Documentation PostgreSQL 15.19

The PostgreSQL Global Development Group

Documentation PostgreSQL 15.19

The PostgreSQL Global Development Group
Copyright © 1996–2026 The PostgreSQL Global Development Group

Legal Notice

PostgreSQL Database Management System (also known as Postgres, formerly known as Postgres95)

Portions Copyright (c) 1996-2026, PostgreSQL Global Development Group

Portions Copyright (c) 1994, The Regents of the University of California

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN « AS-IS » BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Table des matières

Préface xxxii
1. Définition de PostgreSQL xxxii
2. Bref historique de PostgreSQL xxxiii
2.1. Le projet POSTGRES à Berkeley xxxiii
2.2. Postgres95 xxxiii
2.3. PostgreSQL xxxiv
3. Conventions xxxiv
4. Pour plus d'informations xxxiv
5. Lignes de conduite pour les rapports de bogues xxxv
5.1. Identifier les bogues xxxv
5.2. Que rapporter ? xxxvi
5.3. Où rapporter des bogues ? xxxviii
I. Tutoriel 1
1. Démarrage 3
1.1. Installation 3
1.2. Concepts architecturaux de base 3
1.3. Création d'une base de données 4
1.4. Accéder à une base 5
2. Le langage SQL 7
2.1. Introduction 7
2.2. Concepts 7
2.3. Créer une nouvelle table 7
2.4. Remplir une table avec des lignes 8
2.5. Interroger une table 9
2.6. Jointures entre les tables 11
2.7. Fonctions d'agrégat 13
2.8. Mises à jour 15
2.9. Suppressions 15
3. Fonctionnalités avancées 16
3.1. Introduction 16
3.2. Vues 16
3.3. Clés étrangères 16
3.4. Transactions 17
3.5. Fonctions de fenêtrage 19
3.6. Héritage 22
3.7. Conclusion 23
II. Langage SQL 25
4. Syntaxe SQL 33
4.1. Structure lexicale 33
4.2. Expressions de valeurs 42
4.3. Fonctions appelantes 57
5. Définition des données 60
5.1. Notions fondamentales sur les tables 60
5.2. Valeurs par défaut 61
5.3. Colonnes générées 62
5.4. Contraintes 63
5.5. Colonnes système 72
5.6. Modification des tables 73
5.7. Droits 77
5.8. Politiques de sécurité niveau ligne 81
5.9. Schémas 88
5.10. L'héritage 93
5.11. Partitionnement de tables 96
5.12. Données distantes 110
5.13. Autres objets de la base de données 111

5.14. Gestion des dépendances 111
6. Manipulation de données 113
6.1. Insérer des données 113
6.2. Actualiser les données 114
6.3. Supprimer des données 115
6.4. Renvoyer des données provenant de lignes modifiées 115
7. Requêtes 117
7.1. Aperçu 117
7.2. Expressions de table 117
7.3. Listes de sélection 134
7.4. Combiner des requêtes (UNION, INTERSECT, EXCEPT) 135
7.5. Tri des lignes (ORDER BY) 137
7.6. LIMIT et OFFSET 138
7.7. Listes VALUES 138
7.8. Requêtes WITH (*Common Table Expressions*) 139
8. Types de données 149
8.1. Types numériques 150
8.2. Types monétaires 157
8.3. Types caractère 158
8.4. Types de données binaires 160
8.5. Types date/heure 162
8.6. Type booléen 172
8.7. Types énumération 173
8.8. Types géométriques 176
8.9. Types adresses réseau 178
8.10. Type chaîne de bits 181
8.11. Types de recherche plein texte 182
8.12. Type UUID 185
8.13. Type XML 185
8.14. Types JSON 188
8.15. Tableaux 199
8.16. Types composites 209
8.17. Types intervalle de valeurs 215
8.18. Types domaine 222
8.19. Types identifiant d'objet 223
8.20. Type pg_lsn 225
8.21. Pseudo-Types 226
9. Fonctions et opérateurs 228
9.1. Opérateurs logiques 228
9.2. Fonctions et opérateurs de comparaison 229
9.3. Fonctions et opérateurs mathématiques 233
9.4. Fonctions et opérateurs sur des chaînes de caractères 240
9.5. Fonctions et opérateurs pour les chaînes binaires 251
9.6. Fonctions et opérateurs pour les chaînes de bit 256
9.7. Correspondance de motif 257
9.8. Fonctions de formatage de type de données 279
9.9. Fonctions et opérateurs pour date/heure 287
9.10. Fonctions de support d'enum 304
9.11. Fonctions et opérateurs de géométrie 305
9.12. Fonctions et opérateurs pour les adresses réseau 312
9.13. Fonctions et opérateurs de recherche plein texte 315
9.14. Fonctions UUID 322
9.15. Fonctions XML 322
9.16. Fonctions et opérateurs JSON 338
9.17. Fonctions de manipulation de séquence 357
9.18. Expressions conditionnelles 358
9.19. Fonctions et opérateurs tableau 361
9.20. Fonctions et opérateurs sur les intervalles simple/multiple 365

9.21. Fonctions d'agrégat 371
9.22. Fonctions de fenêtrage 378
9.23. Expressions de sous-requêtes 380
9.24. Comparaisons de ligne et de tableau 383
9.25. Fonctions renvoyant des ensembles 386
9.26. Fonctions et opérateurs d'information sur le système 389
9.27. Fonctions d'administration système 408
9.28. Fonctions trigger 426
9.29. Fonctions de trigger d'événement 427
9.30. Fonctions d'informations statistiques 430
10. Conversion de types 431
10.1. Aperçu 431
10.2. Opérateurs 432
10.3. Fonctions 436
10.4. Stockage de valeurs 440
10.5. Constructions UNION, CASE et constructions relatives 441
10.6. Colonnes de sortie du SELECT 443
11. Index 444
11.1. Introduction 444
11.2. Types d'index 445
11.3. Index multicolonnes 447
11.4. Index et ORDER BY 448
11.5. Combiner des index multiples 449
11.6. Index d'unicité 450
11.7. Index d'expressions 450
11.8. Index partiels 451
11.9. Parcours d'index seul et index couvrants 454
11.10. Classes et familles d'opérateurs 457
11.11. Index et collationnements 459
11.12. Examiner l'utilisation des index 459
12. Recherche plein texte 461
12.1. Introduction 461
12.2. Tables et index 465
12.3. Contrôler la recherche plein texte 468
12.4. Fonctionnalités supplémentaires 476
12.5. Analyseurs 482
12.6. Dictionnaires 484
12.7. Exemple de configuration 494
12.8. Tester et déboguer la recherche plein texte 496
12.9. Types d'index préférés pour la recherche plein texte 501
12.10. Support de psql 502
12.11. Limites 505
13. Contrôle d'accès simultané 507
13.1. Introduction 507
13.2. Isolation des transactions 507
13.3. Verrouillage explicite 514
13.4. Vérification de cohérence des données au niveau de l'application 520
13.5. Gestion des échecs de sérialisation 522
13.6. Avertissements 522
13.7. Verrous et index 523
14. Conseils sur les performances 524
14.1. Utiliser EXPLAIN 524
14.2. Statistiques utilisées par le planificateur 536
14.3. Contrôler le planificateur avec des clauses JOIN explicites 541
14.4. Remplir une base de données 543
14.5. Configuration avec une perte acceptée 546
15. Requêtes parallélisées 548
15.1. Comment fonctionne la parallélisation des requêtes 548

15.2. Quand la parallélisation des requêtes peut-elle être utilisée ? 549
15.3. Plans parallélisés 550
15.4. Sécurité de la parallélisation 552
III. Administration du serveur 554
16. Installation depuis les binaires 561
17. Procédure d'installation depuis le code source 562
17.1. Version courte 562
17.2. Prérequis 562
17.3. Obtenir les sources 564
17.4. Procédure d'installation 564
17.5. Initialisation post-installation 579
17.6. Plateformes supportées 581
17.7. Notes spécifiques à des plateformes 581
18. Installation à partir du code source sur Windows 587
18.1. Construire avec Visual C++ ou le Microsoft Windows SDK 587
19. Configuration du serveur et mise en place 593
19.1. Compte utilisateur PostgreSQL 593
19.2. Créer un groupe de base de données 593
19.3. Lancer le serveur de bases de données 596
19.4. Gérer les ressources du noyau 599
19.5. Arrêter le serveur 608
19.6. Mise à jour d'une instance PostgreSQL 609
19.7. Empêcher l'usurpation de serveur (spoofing) 612
19.8. Options de chiffrement 613
19.9. Connexions TCP/IP sécurisées avec SSL 614
19.10. Connexions TCP/IP sécurisées avec le chiffrage GSSAPI 618
19.11. Connexions TCP/IP sécurisées avec des tunnels ssh tunnels 618
19.12. Enregistrer le journal des événements sous Windows 619
20. Configuration du serveur 621
20.1. Paramètres de configuration 621
20.2. Emplacement des fichiers 626
20.3. Connexions et authentification 627
20.4. Consommation des ressources 634
20.5. Write Ahead Log 644
20.6. Réplication 655
20.7. Planification des requêtes 664
20.8. Remonter et tracer les erreurs 671
20.9. Statistiques d'exécution 686
20.10. Nettoyage (`vacuum`) automatique 688
20.11. Valeurs par défaut des connexions client 690
20.12. Gestion des verrous 702
20.13. Compatibilité de version et de plateforme 703
20.14. Gestion des erreurs 705
20.15. Options préconfigurées 706
20.16. Options personnalisées 708
20.17. Options pour les développeurs 708
20.18. Options courtes 713
21. Authentification du client 715
21.1. Le fichier `pg_hba.conf` 715
21.2. Correspondances d'utilisateurs 724
21.3. Méthodes d'authentification 726
21.4. Authentification trust 726
21.5. Authentification par mot de passe 727
21.6. Authentification GSSAPI 728
21.7. Authentification SSPI 730
21.8. Authentification fondée sur ident 731
21.9. Authentification Peer 731
21.10. Authentification LDAP 732

21.11. Authentification RADIUS 735
21.12. Authentification de certificat 736
21.13. Authentification PAM 736
21.14. Authentification BSD 737
21.15. Problèmes d'authentification 737
22. Rôles de la base de données 739
22.1. Rôles de la base de données 739
22.2. Attributs des rôles 740
22.3. Appartenance d'un rôle 742
22.4. Supprimer des rôles 743
22.5. Rôles prédéfinis 744
22.6. Sécurité des fonctions 746
23. Administration des bases de données 747
23.1. Aperçu 747
23.2. Création d'une base de données 748
23.3. Bases de données modèles 749
23.4. Configuration d'une base de données 750
23.5. Détruire une base de données 750
23.6. Tablespaces 751
24. Localisation 753
24.1. Support des locales 753
24.2. Support des collations 756
24.3. Support des jeux de caractères 763
25. Planifier les tâches de maintenance 774
25.1. Nettoyages réguliers 774
25.2. Ré-indexation régulière 784
25.3. Maintenance du fichier de traces 785
26. Sauvegardes et restaurations 787
26.1. Sauvegarde SQL 787
26.2. Sauvegarde de niveau système de fichiers 790
26.3. Archivage continu et récupération d'un instantané (PITR) 791
27. Haute disponibilité, répartition de charge et réplication 803
27.1. Comparaison de différentes solutions 803
27.2. Serveurs secondaires par transfert de journaux 807
27.3. Bascule (*Failover*) 817
27.4. Hot Standby 818
28. Superviser l'activité de la base de données 827
28.1. Outils Unix standard 827
28.2. Le système de statistiques cumulatives 828
28.3. Visualiser les verrous 867
28.4. Rapporter la progression 867
28.5. Traces dynamiques 876
29. Surveiller l'utilisation des disques 888
29.1. Déterminer l'utilisation des disques 888
29.2. Panne pour disque saturé 889
30. Fiabilité et journaux de transaction 890
30.1. Fiabilité 890
30.2. Sommes de contrôle pour les données 892
30.3. Write-Ahead Logging (WAL) 892
30.4. Validation asynchrone (Asynchronous Commit) 893
30.5. Configuration des journaux de transaction 895
30.6. Vue interne des journaux de transaction 898
31. Réplication logique 900
31.1. Publication 900
31.2. Abonnement 901
31.3. Filtres de lignes 906
31.4. Listes de colonnes 914
31.5. Conflits 916

31.6. Restrictions 917
31.7. Architecture 918
31.8. Supervision 919
31.9. Sécurité 919
31.10. Paramètres de configuration 920
31.11. Démarrage rapide 920
32. JIT (compilation à la volée) 922
32.1. Qu'est-ce que le JIT ? 922
32.2. Quand utiliser le JIT ? 922
32.3. Configuration 924
32.4. Extensibilité 924
33. Tests de régression 926
33.1. Lancer les tests 926
33.2. Évaluation des tests 930
33.3. Fichiers de comparaison de variants 932
33.4. Tests TAP 933
33.5. Examen de la couverture des tests 934
IV. Interfaces client 936
34. libpq -- Bibliothèque C 941
34.1. Fonctions de contrôle de connexion à la base de données 941
34.2. Fonctions de statut de connexion 959
34.3. Fonctions d'exécution de commandes 965
34.4. Traitement des commandes asynchrones 982
34.5. Mode pipeline 987
34.6. Récupérer le résultats des requêtes ligne par ligne 991
34.7. Annuler des requêtes en cours d'exécution 992
34.8. Interface rapide (Fast Path) 993
34.9. Notification asynchrone 994
34.10. Fonctions associées à la commande COPY 995
34.11. Fonctions de contrôle 1000
34.12. Fonctions diverses 1002
34.13. Traitement des messages 1006
34.14. Système d'événements 1007
34.15. Variables d'environnement 1014
34.16. Fichier de mots de passe 1016
34.17. Fichier des services de connexion 1017
34.18. Recherche LDAP des paramètres de connexion 1017
34.19. Support de SSL 1018
34.20. Comportement des programmes threadés 1023
34.21. Construire des applications avec libpq 1024
34.22. Exemples de programmes 1025
35. Large Objects 1037
35.1. Introduction 1037
35.2. Fonctionnalités de l'implantation 1037
35.3. Interfaces client 1037
35.4. Fonctions du côté serveur 1042
35.5. Programme d'exemple 1043
36. ECPG -- SQL embarqué en C 1049
36.1. Le Concept 1049
36.2. Gérer les Connexions à la Base de Données 1050
36.3. Exécuter des Commandes SQL 1054
36.4. Utiliser des Variables Hôtes 1057
36.5. SQL Dynamique 1072
36.6. Librairie pgtypes 1074
36.7. Utiliser les Zones de Descripteur 1089
36.8. Gestion des Erreurs 1103
36.9. Directives de Préprocesseur 1110
36.10. Traiter des Programmes en SQL Embarqué 1113

36.11. Fonctions de la Librairie 1114
36.12. Large Objects 1114
36.13. Applications C++ 1116
36.14. Commandes SQL Embarquées 1120
36.15. Mode de Compatibilité Informix 1146
36.16. Mode de compatibilité Oracle 1162
36.17. Fonctionnement Interne 1163
37. Schéma d'information 1166
37.1. Le schéma 1166
37.2. Types de données 1166
37.3. information_schema_catalog_name 1167
37.4. administrable_role_authorizations 1167
37.5. applicable_roles 1167
37.6. attributes 1168
37.7. character_sets 1170
37.8. check_constraint_routine_usage 1171
37.9. check_constraints 1172
37.10. collations 1172
37.11. collation_character_set_applicability 1172
37.12. column_column_usage 1173
37.13. column_domain_usage 1173
37.14. column_options 1174
37.15. column_privileges 1174
37.16. column_udt_usage 1175
37.17. columns 1175
37.18. constraint_column_usage 1179
37.19. constraint_table_usage 1179
37.20. data_type_privileges 1180
37.21. domain_constraints 1180
37.22. domain_udt_usage 1181
37.23. domains 1181
37.24. element_types 1183
37.25. enabled_roles 1186
37.26. foreign_data_wrapper_options 1186
37.27. foreign_data_wrappers 1186
37.28. foreign_server_options 1187
37.29. foreign_servers 1187
37.30. foreign_table_options 1188
37.31. foreign_tables 1188
37.32. key_column_usage 1189
37.33. parameters 1189
37.34. referential_constraints 1191
37.35. role_column_grants 1192
37.36. role_routine_grants 1192
37.37. role_table_grants 1193
37.38. role_udt_grants 1194
37.39. role_usage_grants 1194
37.40. routine_column_usage 1195
37.41. routine_privileges 1196
37.42. routine_routine_usage 1196
37.43. routine_sequence_usage 1197
37.44. routine_table_usage 1198
37.45. routines 1198
37.46. schemata 1203
37.47. sequences 1203
37.48. sql_features 1204
37.49. sql_implementation_info 1205
37.50. sql_parts 1205

37.51. sql_sizing 1205
37.52. table_constraints 1206
37.53. table_privileges 1207
37.54. tables 1207
37.55. transforms 1208
37.56. triggered_update_columns 1209
37.57. triggers 1209
37.58. udt_privileges 1211
37.59. usage_privileges 1211
37.60. user_defined_types 1212
37.61. user_mapping_options 1214
37.62. user_mappings 1214
37.63. view_column_usage 1215
37.64. view_routine_usage 1215
37.65. view_table_usage 1216
37.66. views 1216
V. Programmation serveur 1218
38. Étendre SQL 1224
38.1. L'extensibilité 1224
38.2. Le système des types de PostgreSQL 1224
38.3. Fonctions utilisateur 1228
38.4. Procédures utilisateur 1228
38.5. Fonctions en langage de requêtes (SQL) 1229
38.6. Surcharge des fonctions 1247
38.7. Catégories de volatilité des fonctions 1248
38.8. Fonctions en langage de procédures 1249
38.9. Fonctions internes 1250
38.10. Fonctions en langage C 1250
38.11. Informations d'optimisation de fonction 1272
38.12. Agrégats utilisateur 1273
38.13. Types utilisateur 1281
38.14. Opérateurs définis par l'utilisateur 1285
38.15. Informations sur l'optimisation d'un opérateur 1286
38.16. Interfacer des extensions d'index 1290
38.17. Empaqueter des objets dans une extension 1304
38.18. Outils de construction d'extension 1314
39. Triggers (triggers) 1318
39.1. Aperçu du comportement des triggers 1318
39.2. Visibilité des modifications des données 1321
39.3. Écrire des fonctions triggers en C 1322
39.4. Un exemple complet de trigger 1325
40. Triggers sur événement 1329
40.1. Aperçu du fonctionnement des triggers sur événement 1329
40.2. Matrice de déclenchement des triggers sur événement 1330
40.3. Écrire des fonctions trigger sur événement en C 1334
40.4. Un exemple complet de trigger sur événement 1335
40.5. Un exemple de trigger sur événement de table modifiée 1336
41. Système de règles 1338
41.1. Arbre de requêtes 1338
41.2. Vues et système de règles 1340
41.3. Vues matérialisées 1347
41.4. Règles sur insert, update et delete 1350
41.5. Règles et droits 1362
41.6. Règles et statut de commande 1364
41.7. Règles contre triggers 1365
42. Langages de procédures 1368
42.1. Installation des langages de procédures 1368
43. PL/pgSQL -- Langage de procédures SQL 1371

43.1. Aperçu 1371
43.2. Structure de PL/pgSQL 1372
43.3. Déclarations 1374
43.4. Expressions 1381
43.5. Instructions de base 1382
43.6. Structures de contrôle 1391
43.7. Curseurs 1406
43.8. Gestion des transactions 1412
43.9. Erreurs et messages 1414
43.10. Fonctions trigger 1416
43.11. Les dessous de PL/pgSQL 1425
43.12. Astuces pour développer en PL/pgSQL 1429
43.13. Portage d'Oracle PL/SQL 1433
44. PL/Tcl -- Langage de procédures Tcl 1443
44.1. Aperçu 1443
44.2. Fonctions et arguments PL/Tcl 1443
44.3. Valeurs des données avec PL/Tcl 1445
44.4. Données globales avec PL/Tcl 1446
44.5. Accès à la base de données depuis PL/Tcl 1446
44.6. Fonctions triggers en PL/Tcl 1449
44.7. Fonctions trigger sur événement en PL/Tcl 1451
44.8. Gestion des erreurs avec PL/Tcl 1451
44.9. Sous-transactions explicites dans PL/Tcl 1452
44.10. Gestion des transactions 1453
44.11. Configuration PL/Tcl 1454
44.12. Noms de procédure Tcl 1454
45. PL/Perl -- Langage de procédures Perl 1455
45.1. Fonctions et arguments PL/Perl 1455
45.2. Valeurs en PL/Perl 1460
45.3. Fonctions incluses 1460
45.4. Valeurs globales dans PL/Perl 1465
45.5. Niveaux de confiance de PL/Perl 1466
45.6. Triggers PL/Perl 1468
45.7. Triggers sur événements avec PL/Perl 1469
45.8. PL/Perl sous le capot 1470
46. PL/Python -- Langage de procédures Python 1472
46.1. Fonctions PL/Python 1472
46.2. Valeur des données avec PL/Python 1473
46.3. Partage de données 1479
46.4. Blocs de code anonymes 1479
46.5. Fonctions de triggers 1479
46.6. Accès à la base de données 1480
46.7. Sous-transactions explicites 1484
46.8. Gestion des transactions 1485
46.9. Fonctions outils 1486
46.10. Python 2 versus Python 3 1487
46.11. Variables d'environnement 1487
47. Interface de programmation serveur 1489
47.1. Fonctions d'interface 1489
47.2. Fonctions de support d'interface 1531
47.3. Gestion de la mémoire 1540
47.4. Gestion des transactions 1550
47.5. Visibilité des modifications de données 1553
47.6. Exemples 1553
48. Processus en tâche de fond (background worker) 1557
49. Décodage logique (Logical Decoding) 1561
49.1. Exemples de décodage logique 1561
49.2. Concepts de décodage logique 1565

49.3. Interface du protocole de réplication par flux 1566
49.4. Interface SQL de décodage logique 1566
49.5. Catalogues systèmes liés au décodage logique 1567
49.6. Plugins de sortie de décodage logique 1567
49.7. Écrivains de sortie de décodage logique 1575
49.8. Support de la réplication synchrone pour le décodage logique 1575
49.9. Envoi en flux de grosses transactions pour le décodage logique 1576
49.10. Support du Two-phase commit pour le décodage logique 1577
50. Tracer la progression de la réplication 1579
51. Modules d'archivage 1580
51.1. Fonctions d'initialisation 1580
51.2. Fonctions de support d'un module d'archivage 1580
VI. Référence 1582
I. Commandes SQL 1587
ABORT 1591
ALTER AGGREGATE 1592
ALTER COLLATION 1594
ALTER CONVERSION 1597
ALTER DATABASE 1599
ALTER DEFAULT PRIVILEGES 1602
ALTER DOMAIN 1606
ALTER EVENT TRIGGER 1610
ALTER EXTENSION 1611
ALTER FOREIGN DATA WRAPPER 1615
ALTER FOREIGN TABLE 1617
ALTER FUNCTION 1622
ALTER GROUP 1626
ALTER INDEX 1628
ALTER LANGUAGE 1631
ALTER LARGE OBJECT 1632
ALTER MATERIALIZED VIEW 1633
ALTER OPERATOR 1635
ALTER OPERATOR CLASS 1637
ALTER OPERATOR FAMILY 1638
ALTER POLICY 1642
ALTER PROCEDURE 1644
ALTER PUBLICATION 1647
ALTER ROLE 1650
ALTER ROUTINE 1654
ALTER RULE 1656
ALTER SCHEMA 1657
ALTER SEQUENCE 1658
ALTER SERVER 1662
ALTER STATISTICS 1664
ALTER SUBSCRIPTION 1666
ALTER SYSTEM 1669
ALTER TABLE 1671
ALTER TABLESPACE 1690
ALTER TEXT SEARCH CONFIGURATION 1692
ALTER TEXT SEARCH DICTIONARY 1694
ALTER TEXT SEARCH PARSER 1696
ALTER TEXT SEARCH TEMPLATE 1697
ALTER TRIGGER 1698
ALTER TYPE 1700
ALTER USER 1705
ALTER USER MAPPING 1706
ALTER VIEW 1708
ANALYZE 1711

BEGIN 1714
CALL 1716
CHECKPOINT 1718
CLOSE 1719
CLUSTER 1720
COMMENT 1723
COMMIT 1728
COMMIT PREPARED 1729
COPY 1730
CREATE ACCESS METHOD 1741
CREATE AGGREGATE 1742
CREATE CAST 1750
CREATE COLLATION 1755
CREATE CONVERSION 1758
CREATE DATABASE 1760
CREATE DOMAIN 1765
CREATE EVENT TRIGGER 1768
CREATE EXTENSION 1770
CREATE FOREIGN DATA WRAPPER 1773
CREATE FOREIGN TABLE 1775
CREATE FUNCTION 1780
CREATE GROUP 1790
CREATE INDEX 1791
CREATE LANGUAGE 1801
CREATE MATERIALIZED VIEW 1804
CREATE OPERATOR 1806
CREATE OPERATOR CLASS 1809
CREATE OPERATOR FAMILY 1812
CREATE POLICY 1813
CREATE PROCEDURE 1820
CREATE PUBLICATION 1824
CREATE ROLE 1829
CREATE RULE 1834
CREATE SCHEMA 1837
CREATE SEQUENCE 1840
CREATE SERVER 1844
CREATE STATISTICS 1846
CREATE SUBSCRIPTION 1850
CREATE TABLE 1854
CREATE TABLE AS 1878
CREATE TABLESPACE 1881
CREATE TEXT SEARCH CONFIGURATION 1883
CREATE TEXT SEARCH DICTIONARY 1885
CREATE TEXT SEARCH PARSER 1887
CREATE TEXT SEARCH TEMPLATE 1889
CREATE TRANSFORM 1891
CREATE TRIGGER 1894
CREATE TYPE 1902
CREATE USER 1912
CREATE USER MAPPING 1913
CREATE VIEW 1915
DEALLOCATE 1921
DECLARE 1922
DELETE 1926
DISCARD 1929
DO 1931
DROP ACCESS METHOD 1933
DROP AGGREGATE 1934

DROP CAST 1936
DROP COLLATION 1937
DROP CONVERSION 1938
DROP DATABASE 1939
DROP DOMAIN 1941
DROP EVENT TRIGGER 1942
DROP EXTENSION 1943
DROP FOREIGN DATA WRAPPER 1945
DROP FOREIGN TABLE 1946
DROP FUNCTION 1948
DROP GROUP 1950
DROP INDEX 1951
DROP LANGUAGE 1953
DROP MATERIALIZED VIEW 1954
DROP OPERATOR 1955
DROP OPERATOR CLASS 1957
DROP OPERATOR FAMILY 1959
DROP OWNED 1961
DROP POLICY 1962
DROP PROCEDURE 1963
DROP PUBLICATION 1966
DROP ROLE 1967
DROP ROUTINE 1968
DROP RULE 1970
DROP SCHEMA 1971
DROP SEQUENCE 1973
DROP SERVER 1974
DROP STATISTICS 1975
DROP SUBSCRIPTION 1976
DROP TABLE 1978
DROP TABLESPACE 1979
DROP TEXT SEARCH CONFIGURATION 1980
DROP TEXT SEARCH DICTIONARY 1981
DROP TEXT SEARCH PARSER 1982
DROP TEXT SEARCH TEMPLATE 1983
DROP TRANSFORM 1984
DROP TRIGGER 1986
DROP TYPE 1987
DROP USER 1988
DROP USER MAPPING 1989
DROP VIEW 1990
END 1991
EXECUTE 1992
EXPLAIN 1993
FETCH 1999
GRANT 2003
IMPORT FOREIGN SCHEMA 2009
INSERT 2011
LISTEN 2019
LOAD 2021
LOCK 2022
MERGE 2025
MOVE 2031
NOTIFY 2033
PREPARE 2036
PREPARE TRANSACTION 2039
REASSIGN OWNED 2041
REFRESH MATERIALIZED VIEW 2042

REINDEX 2044
RELEASE SAVEPOINT 2050
RESET 2052
REVOKE 2053
ROLLBACK 2057
ROLLBACK PREPARED 2058
ROLLBACK TO SAVEPOINT 2059
SAVEPOINT 2061
SECURITY LABEL 2063
SELECT 2066
SELECT INTO 2089
SET 2091
SET CONSTRAINTS 2094
SET ROLE 2096
SET SESSION AUTHORIZATION 2098
SET TRANSACTION 2100
SHOW 2103
START TRANSACTION 2105
TRUNCATE 2106
UNLISTEN 2109
UPDATE 2111
VACUUM 2116
VALUES 2121
II. Applications client de PostgreSQL 2124
clusterdb 2125
createdb 2128
createuser 2132
dropdb 2137
dropuser 2140
ecpg 2143
pg_amcheck 2146
pg_basebackup 2152
pgbench 2162
pg_config 2188
pg_dump 2191
pg_dumpall 2206
pg_isready 2214
pg_receivewal 2216
pg_recvlogical 2221
pg_restore 2225
pg_verifybackup 2235
psql 2238
reindexdb 2285
vacuumdb 2289
III. Applications relatives au serveur PostgreSQL 2295
initdb 2296
pg_archivecleanup 2301
pg_checksums 2303
pg_controldata 2305
pg_ctl 2306
pg_resetwal 2312
pg_rewind 2316
pg_test_fsync 2320
pg_test_timing 2321
pg_upgrade 2325
pg_waldump 2335
postgres 2338
postmaster 2346

VII. Internes 2347
52. Présentation des mécanismes internes de PostgreSQL 2353
52.1. Chemin d'une requête 2353
52.2. Établissement des connexions 2353
52.3. Étape d'analyse 2354
52.4. Système de règles de PostgreSQL 2355
52.5. Planificateur/Optimiseur 2355
52.6. Exécuteur 2357
53. Catalogues système 2359
53.1. Aperçu 2359
53.2. pg_aggregate 2361
53.3. pg_am 2362
53.4. pg_amop 2363
53.5. pg_amproc 2364
53.6. pg_attrdef 2364
53.7. pg_attribute 2365
53.8. pg_authid 2367
53.9. pg_auth_members 2368
53.10. pg_cast 2369
53.11. pg_class 2369
53.12. pg_collation 2372
53.13. pg_constraint 2373
53.14. pg_conversion 2375
53.15. pg_database 2375
53.16. pg_db_role_setting 2377
53.17. pg_default_acl 2377
53.18. pg_depend 2378
53.19. pg_description 2380
53.20. pg_enum 2380
53.21. pg_event_trigger 2381
53.22. pg_extension 2381
53.23. pg_foreign_data_wrapper 2382
53.24. pg_foreign_server 2383
53.25. pg_foreign_table 2383
53.26. pg_index 2384
53.27. pg_inherits 2385
53.28. pg_init_privs 2386
53.29. pg_language 2386
53.30. pg_largeobject 2387
53.31. pg_largeobject_metadata 2388
53.32. pg_namespace 2388
53.33. pg_opclass 2388
53.34. pg_operator 2389
53.35. pg_opfamily 2390
53.36. pg_parameter_acl 2390
53.37. pg_partitioned_table 2391
53.38. pg_policy 2391
53.39. pg_proc 2392
53.40. pg_publication 2395
53.41. pg_publication_namespace 2395
53.42. pg_publication_rel 2396
53.43. pg_range 2396
53.44. pg_replication_origin 2397
53.45. pg_rewrite 2397
53.46. pg_seclabel 2398
53.47. pg_sequence 2398
53.48. pg_shdepend 2399
53.49. pg_shdescription 2400

53.50. pg_shseclabel 2400
53.51. pg_statistic 2401
53.52. pg_statistic_ext 2402
53.53. pg_statistic_ext_data 2403
53.54. pg_subscription 2404
53.55. pg_subscription_rel 2405
53.56. pg_tablespace 2405
53.57. pg_transform 2406
53.58. pg_trigger 2406
53.59. pg_ts_config 2408
53.60. pg_ts_config_map 2408
53.61. pg_ts_dict 2409
53.62. pg_ts_parser 2409
53.63. pg_ts_template 2410
53.64. pg_type 2410
53.65. pg_user_mapping 2414
54. Vues système 2415
54.1. Aperçu 2415
54.2. pg_available_extensions 2416
54.3. pg_available_extension_versions 2416
54.4. pg_backend_memory_contexts 2417
54.5. pg_config 2418
54.6. pg_cursors 2418
54.7. pg_file_settings 2419
54.8. pg_group 2420
54.9. pg_hba_file_rules 2420
54.10. pg_ident_file_mappings 2421
54.11. pg_indexes 2422
54.12. pg_locks 2422
54.13. pg_matviews 2425
54.14. pg_policies 2425
54.15. pg_prepared_statements 2426
54.16. pg_prepared_xacts 2426
54.17. pg_publication_tables 2427
54.18. pg_replication_origin_status 2427
54.19. pg_replication_slots 2428
54.20. pg_roles 2429
54.21. pg_rules 2430
54.22. pg_seclabels 2431
54.23. pg_sequences 2431
54.24. pg_settings 2432
54.25. pg_shadow 2434
54.26. pg_shmem_allocations 2435
54.27. pg_stats 2436
54.28. pg_stats_ext 2437
54.29. pg_stats_ext_exprs 2438
54.30. pg_tables 2440
54.31. pg_timezone_abbrevs 2440
54.32. pg_timezone_names 2441
54.33. pg_user 2441
54.34. pg_user_mappings 2442
54.35. pg_views 2442
55. Protocole Frontend/Backend 2444
55.1. Aperçu 2444
55.2. Flot de messages 2446
55.3. Authentification SASL 2460
55.4. Protocole de réplication en flux 2462
55.5. Logical Streaming Replication Protocol 2471

55.6. Types de données des messages 2473
55.7. Formats de message 2473
55.8. Champs des messages d'erreur et de notification 2491
55.9. Formats des messages de la réplication logique 2493
55.10. Résumé des changements depuis le protocole 2.0 2501
56. Conventions de codage pour PostgreSQL 2503
56.1. Formatage 2503
56.2. Reporter les erreurs dans le serveur 2504
56.3. Guide de style des messages d'erreur 2507
56.4. Conventions diverses de codage 2511
57. Support natif des langues 2514
57.1. Pour le traducteur 2514
57.2. Pour le développeur 2517
58. Écrire un gestionnaire de langage procédural 2520
59. Écrire un wrapper de données distantes 2523
59.1. Fonctions d'un wrapper de données distantes 2523
59.2. Routines callback des wrappers de données distantes 2524
59.3. Fonctions d'aide pour les wrapper de données distantes 2541
59.4. Planification de la requête avec un wrapper de données distantes 2543
59.5. Le verrouillage de ligne dans les wrappers de données distantes 2545
60. Écrire une méthode d'échantillonnage de table 2548
60.1. Fonctions de support d'une méthode d'échantillonnage 2549
61. Écrire un module de parcours personnalisé 2552
61.1. Créer des parcours de chemin personnalisés 2552
61.2. Créer des parcours de plans personnalisés 2554
61.3. Exécution de parcours personnalisés 2555
62. Optimiseur génétique de requêtes (*Genetic Query Optimizer*) 2558
62.1. Gérer les requêtes, un problème d'optimisation complexe 2558
62.2. Algorithmes génétiques 2558
62.3. Optimisation génétique des requêtes (GEQO) dans PostgreSQL 2559
62.4. Lectures supplémentaires 2561
63. Définition de l'interface des méthodes d'accès aux tables 2562
64. Définition de l'interface des méthodes d'accès aux index 2564
64.1. Structure basique de l'API pour les index 2564
64.2. Fonctions des méthodes d'accès aux index 2567
64.3. Parcours d'index 2574
64.4. Considérations sur le verrouillage d'index 2575
64.5. Vérification de l'unicité par les index 2576
64.6. Fonctions d'estimation des coûts d'index 2578
65. Enregistrements génériques des journaux de transactions 2581
66. Gestionnaires de ressources WAL personnalisées 2583
67. Index B-Tree 2585
67.1. Introduction 2585
67.2. Comportement des classes d'opérateur B-Tree 2585
67.3. Fonctions de support B-Tree 2586
67.4. Implémentation 2589
68. Index GiST 2593
68.1. Introduction 2593
68.2. Classes d'opérateur internes 2593
68.3. Extensibilité 2596
68.4. Implémentation 2609
68.5. Exemples 2610
69. Index SP-GiST 2611
69.1. Introduction 2611
69.2. Classes d'opérateur internes 2611
69.3. Extensibilité 2613
69.4. Implémentation 2623
69.5. Exemples 2625

70. Index GIN 2626
70.1. Introduction 2626
70.2. Classes d'opérateur natives 2626
70.3. Extensibilité 2627
70.4. Implantation 2630
70.5. Conseils et astuces sur GIN 2631
70.6. Limitations 2632
70.7. Exemples 2632
71. Index BRIN 2633
71.1. Introduction 2633
71.2. Classes d'opérateur intégrées 2634
71.3. Extensibilité 2642
72. Index Hash 2647
72.1. Aperçu 2647
72.2. Implémentation 2648
73. Stockage physique de la base de données 2649
73.1. Emplacement des fichiers de la base de données 2649
73.2. TOAST 2651
73.3. Carte des espaces libres 2654
73.4. Carte de visibilité 2655
73.5. Fichier d'initialisation 2655
73.6. Emplacement des pages de la base de données 2655
73.7. Heap-Only Tuples (HOT) 2658
74. Déclaration du catalogue système et contenu initial 2660
74.1. Règles de déclaration de catalogue système 2660
74.2. Données initiales du catalogue système 2661
74.3. Format des fichiers BKI 2667
74.4. Commandes BKI 2667
74.5. Structure du fichier BKI de « bootstrap » 2668
74.6. Exemple BKI 2669
75. Comment le planificateur utilise les statistiques 2670
75.1. Exemples d'estimation des lignes 2670
75.2. Exemples de statistiques multivariées 2676
75.3. Statistiques de l'optimiseur et sécurité 2679
76. Format de manifeste de sauvegarde 2681
76.1. Objet de niveau supérieur du manifeste de sauvegarde 2681
76.2. Objet fichier du manifeste de sauvegarde 2681
76.3. Objet intervalle des journaux de transaction du manifeste de sauvegarde 2682
VIII. Annexes 2683
A. Codes d'erreur de PostgreSQL 2690
B. Support de date/heure 2699
B.1. Interprétation des Date/Heure saisies 2699
B.2. Gestion des horodatages ambigus ou invalides 2700
B.3. Mots-clés Date/Heure 2701
B.4. Fichiers de configuration date/heure 2702
B.5. Spécification POSIX des fuseaux horaires 2703
B.6. Histoire des unités 2705
B.7. Dates Julien 2706
C. Mots-clés SQL 2708
D. Conformité SQL 2733
D.1. Fonctionnalités supportées 2734
D.2. Fonctionnalités non supportées 2745
D.3. Limitations XML et conformité à SQL/XML 2753
E. Notes de version 2757
E.1. Release 15.19 2757
E.2. Release 15.18 2767
E.3. Release 15.17 2772
E.4. Release 15.16 2773

E.5. Release 15.15 2777
E.6. Release 15.14 2782
E.7. Release 15.13 2788
E.8. Release 15.12 2791
E.9. Release 15.11 2792
E.10. Release 15.10 2796
E.11. Release 15.9 2797
E.12. Release 15.8 2803
E.13. Release 15.7 2808
E.14. Release 15.6 2813
E.15. Release 15.5 2818
E.16. Release 15.4 2824
E.17. Release 15.3 2828
E.18. Release 15.2 2835
E.19. Release 15.1 2841
E.20. Release 15 2843
E.21. Versions précédentes 2865
F. Modules supplémentaires fournis 2866
F.1. adminpack 2867
F.2. amcheck 2868
F.3. auth_delay 2873
F.4. auto_explain 2874
F.5. basebackup_to_shell 2877
F.6. basic_archive 2877
F.7. bloom 2878
F.8. btree_gin 2881
F.9. btree_gist 2882
F.10. citext 2883
F.11. cube 2886
F.12. dblink 2891
F.13. dict_int 2923
F.14. dict_xsyn 2924
F.15. earthdistance 2926
F.16. file_fdw 2927
F.17. fuzzystrmatch 2930
F.18. hstore 2933
F.19. intagg 2941
F.20. intarray 2942
F.21. isn 2946
F.22. lo 2949
F.23. ltree 2951
F.24. old_snapshot 2958
F.25. pageinspect 2959
F.26. passwordcheck 2969
F.27. pg_buffercache 2970
F.28. pgcrypto 2972
F.29. pg_freespacemap 2983
F.30. pg_prewarm 2984
F.31. pgrowlocks 2986
F.32. pg_stat_statements 2987
F.33. pgstattuple 2995
F.34. pg_surgery 3000
F.35. pg_trgm 3001
F.36. pg_visibility 3007
F.37. pg_walinspect 3009
F.38. postgres_fdw 3011
F.39. seg 3021
F.40. sepgsql 3024

F.41. spi 3032
F.42. sslinfo 3035
F.43. tablefunc 3037
F.44. tcn 3047
F.45. test_decoding 3048
F.46. tsm_system_rows 3049
F.47. tsm_system_time 3050
F.48. unaccent 3050
F.49. uuid-ossp 3052
F.50. xml2 3054
G. Programmes supplémentaires fournis 3059
G.1. Applications clients 3059
G.2. Applications serveurs 3066
H. Projets externes 3068
H.1. Interfaces client 3068
H.2. Outils d'administration 3068
H.3. Langages procéduraux 3068
H.4. Extensions 3068
I. Dépôt du code source 3069
I.1. Récupérer les sources via Git 3069
J. Documentation 3070
J.1. DocBook 3070
J.2. Ensemble d'outils 3070
J.3. Construire la documentation 3073
J.4. Écriture de la documentation 3075
J.5. Guide des styles 3075
K. Limites PostgreSQL 3078
L. Acronymes 3080
M. Glossaire 3087
N. Support des couleurs 3102
N.1. Quand les couleurs sont utilisées 3102
N.2. Configurer les couleurs 3102
O. Fonctionnalités obsolètes ou renommées 3103
O.1. fichier `recovery.conf` intégré à `postgresql.conf` 3103
O.2. Les rôles par défaut sont renommés en rôles prédéfinis 3103
O.3. `pg_xlogdump` renommé en `pg_waldump` 3103
O.4. `pg_resetxlog` renommé en `pg_resetwal` 3103
O.5. `pg_receivexlog` renommé en `pg_receivewal` 3104
P. Traduction française 3105
Bibliographie 3108
Index 3110

Liste des illustrations

62.1. Structure d'un algorithme génétique 2559
70.1. Cœur de GIN 2630
73.1. Disposition d'une page 2657

Liste des tableaux

4.1. Séquences d'échappements avec antislash 36
4.2. Précédence des opérateurs (du plus haut vers le plus bas) 41
5.1. Abréviations des droits dans les ACL 80
5.2. Résumé des droits d'accès 80
8.1. Types de données 149
8.2. Types numériques 151
8.3. Types monétaires 157
8.4. Types caractère 158
8.5. Types caractères spéciaux 160
8.6. Types de données binaires 160
8.7. Octets littéraux `bytea` à échapper 161
8.8. Octets échappés en sortie pour `bytea` 162
8.9. Types date et heure 162
8.10. Saisie de date 164
8.11. Saisie d'heure 164
8.12. Saisie des fuseaux horaires 165
8.13. Saisie de dates/heures spéciales 166
8.14. Styles d'affichage de date/heure 167
8.15. Convention de présentation des dates 168
8.16. Abréviations d'unités d'intervalle ISO 8601 170
8.17. Saisie d'intervalle 171
8.18. Exemples de styles d'affichage d'intervalles 172
8.19. Type de données booléen 173
8.20. Types géométriques 176
8.21. Types d'adresses réseau 178
8.22. Exemples de saisie de types `cidr` 179
8.23. Types primitifs JSON et types PostgreSQL correspondants 189
8.24. Variables `jsonpath` 198
8.25. Accesseurs `jsonpath` 199
8.26. Types identifiant d'objet 223
8.27. Pseudo-Types 226
9.1. Opérateurs de comparaison 229
9.2. Prédicats de comparaison 230
9.3. Fonctions de comparaison 233
9.4. Opérateurs mathématiques 233
9.5. Fonctions mathématiques 235
9.6. Fonctions aléatoires 238
9.7. Fonctions trigonométriques 238
9.8. Fonctions hyperboliques 240
9.9. Fonctions et opérateurs SQL pour les chaînes de caractères 241
9.10. Autres fonctions et opérateurs pour les chaînes de caractères 243
9.11. Fonctions et opérateurs SQL pour des chaînes binaires 251
9.12. Autres fonctions de chaînes binaires 252
9.13. Fonctions de conversion texte/binaire 254
9.14. Opérateurs pour les chaînes de bit 256
9.15. Fonctions pour les chaînes de bit 256
9.16. Opérateurs de correspondance d'expressions rationnelles 261
9.17. Atomes d'expression rationnelle 268
9.18. Quantifieur pour expression rationnelle 268
9.19. Contraintes d'expression rationnelle 269
9.20. Échappements d'entrée de caractère pour les expressions rationnelles 271
9.21. Échappements de raccourci de classe pour les expressions rationnelles 272
9.22. Échappements de contraintes pour les expressions rationnelles 273
9.23. Références d'expressions rationnelles 273
9.24. Lettres pour options intégrées des ARE 274

9.25. Équivalences des fonctions sur les expressions rationnelles 278
9.26. Fonctions de formatage 279
9.27. Motifs modèles pour le formatage date/heure 280
9.28. Modificateurs des motifs modèles pour le formatage date/heure 282
9.29. Motifs de modèle pour le formatage numérique 285
9.30. Modificateurs de motifs de modèle pour le formatage numérique 286
9.31. `to_char` Examples 286
9.32. Opérateurs Date/Heure 288
9.33. Fonctions Date/Heure 289
9.34. Variantes `AT TIME ZONE` 301
9.35. Fonctions de support des enum 304
9.36. Opérateurs géométriques 305
9.37. Fonctions géométriques 309
9.38. Fonctions de conversion de type géométrique 310
9.39. Opérateurs d'adresse IP 312
9.40. Fonctions des adresses IP 314
9.41. Fonctions pour les adresses MAC 315
9.42. Opérateurs de recherche plein texte 316
9.43. Fonctions de recherche plein texte 317
9.44. Fonctions de débogage de recherche plein texte 321
9.45. Opérateurs `json` et `jsonb` 338
9.46. Opérateurs `jsonb` supplémentaires 339
9.47. Fonctions de création de JSON 341
9.48. Fonctions de traitement JSON 342
9.49. Opérateurs et méthodes `jsonpath` 352
9.50. Éléments d'expression de filtre `jsonpath` 354
9.51. Fonctions pour séquence 357
9.52. Opérateurs de tableau 362
9.53. Fonctions tableau 362
9.54. Opérateurs sur les intervalles simples 365
9.55. Opérateurs sur les intervalles multiples 366
9.56. Fonctions sur intervalles simples 369
9.57. Fonctions sur intervalles multiples 370
9.58. Fonctions d'agrégat à but général 371
9.59. Fonctions d'agrégat pour les statistiques 374
9.60. Fonctions d'agrégat à ensemble trié 376
9.61. Fonctions d'agrégat à ensemble hypothétique 377
9.62. Opérations de regroupement 377
9.63. Fonctions de fenêtrage à usage général 378
9.64. Fonctions générant des séries 386
9.65. Fonctions générant des indices 387
9.66. Fonctions d'information sur la session 389
9.67. Fonctions d'accès aux droits 392
9.68. Opérateurs `aclitem` 394
9.69. Fonctions `aclitem` 394
9.70. Fonctions de vérification de la visibilité 395
9.71. Fonctions d'information sur les catalogues systèmes 396
9.72. Propriétés des colonnes d'un index 401
9.73. Propriétés des index 401
9.74. Propriétés des méthodes d'accès aux index 402
9.75. Drapeaux des GUC 402
9.76. Fonctions d'information et d'adressage des objets 402
9.77. Fonctions d'informations sur les commentaires 403
9.78. Fonctions d'identifiants de transaction et d'informations sur les images de base 404
9.79. Composants d'image de base 405
9.80. Fonctions obsolètes sur les identifiants de transactions et les informations sur les images de base 405
9.81. Fonctions d'informations sur les transactions validées 406

9.82. Fonctions des données de contrôle 406
9.83. Colonnes en sortie de pg_control_checkpoint 407
9.84. Colonnes en sortie de pg_control_system 407
9.85. Colonnes en sortie de pg_control_init 407
9.86. Colonnes en sortie de pg_control_recovery 408
9.87. Fonctions de configuration 408
9.88. Fonctions d'envoi de signaux 409
9.89. Fonctions de contrôle de la sauvegarde 410
9.90. Fonctions d'information sur la restauration 413
9.91. Fonctions de contrôle de la restauration 413
9.92. Fonctions de synchronisation d'image 415
9.93. Fonctions de gestion de la réplication 415
9.94. Fonctions de taille des objets 418
9.95. Fonctions d'emplacement des objets de la base 419
9.96. Fonctions de gestion des collations 420
9.97. Fonctions d'information sur le partitionnement 420
9.98. Fonctions de maintenance des index 421
9.99. Fonctions génériques d'accès aux fichiers 422
9.100. Fonctions pour les verrous informatifs 424
9.101. Fonctions triggers natives 426
9.102. Fonctions d'informations sur la réécriture de table 429
12.1. Types de jeton de l'analyseur par défaut 482
13.1. Niveaux d'isolation des transactions 508
13.2. Modes de verrou conflictuels 516
13.3. Verrous en conflit au niveau ligne 517
19.1. Paramètres System V IPC 600
19.2. Utilisation des fichiers serveur SSL 616
20.1. Modes pour synchronous_commit 646
20.2. Niveaux de sévérité des messages 677
20.3. Clés et valeurs des enregistrements JSON 685
20.4. Clé d'option courte 713
22.1. Rôles prédéfinis 744
24.1. Jeux de caractères de PostgreSQL 764
24.2. Conversions client/serveur fournies de base pour les jeux de caractères 768
24.3. Toutes les conversions fournies de base entre jeux de caractères 769
27.1. Matrice de fonctionnalités : haute disponibilité, répartition de charge et réplication 806
28.1. Vues statistiques dynamiques 830
28.2. Vues sur les statistiques récupérées 830
28.3. Vue pg_stat_activity 833
28.4. Types de Wait Event 835
28.5. Wait Events de type Activity 836
28.6. Wait Events de type BufferPin 836
28.7. Wait Events de type Client 837
28.8. Wait Events de type Extension 837
28.9. Wait Events de type IO 837
28.10. Wait Events de type IPC 840
28.11. Wait Events de type Lock 843
28.12. Wait Events de type LWLock 843
28.13. Wait Events de type Timeout 847
28.14. Vue pg_stat_replication 848
28.15. pg_stat_replication_slots View 850
28.16. Vue pg_stat_wal_receiver 851
28.17. pg_stat_recovery_prefetch View 852
28.18. Vue pg_stat_subscription 853
28.19. pg_stat_subscription_stats View 853
28.20. Vue pg_stat_ssl 854
28.21. Vue pg_stat_gssapi 854
28.22. Vue pg_stat_archiver 855

28.23. Vue pg_stat_bgwriter 855
28.24. pg_stat_wal View 856
28.25. Vue pg_stat_database 857
28.26. Vue pg_stat_database_conflicts 859
28.27. Vue pg_stat_all_tables 860
28.28. Vue pg_stat_all_indexes 861
28.29. Vue pg_statio_all_tables 862
28.30. Vue pg_statio_all_indexes 862
28.31. Vue pg_statio_all_sequences 863
28.32. Vue pg_stat_user_functions 863
28.33. Vue pg_stat_slru 864
28.34. Fonctions supplémentaires de statistiques 865
28.35. Fonctions statistiques par processus serveur 866
28.36. Vue pg_stat_progress_analyze 868
28.37. Phases ANALYZE 868
28.38. Vue pg_stat_progress_create_index 869
28.39. Phases du CREATE INDEX 870
28.40. Vue pg_stat_progress_vacuum 871
28.41. Phases du VACUUM 872
28.42. Vue pg_stat_progress_cluster 873
28.43. Phases de CLUSTER et VACUUM FULL 874
28.44. Vue pg_stat_progress_basebackup 874
28.45. Phases de la sauvegarde de base 875
28.46. Vue pg_stat_progress_copy 875
28.47. Sondes disponibles pour DTrace 877
28.48. Types définis utilisés comme paramètres de sonde 885
31.1. Résumé des transformations d'un UPDATE 907
34.1. Description des modes SSL 1021
34.2. Utilisation des fichiers SSL libpq/client 1022
35.1. Fonctions SQL orientées « Large Object » 1042
36.1. Correspondance Entre les Types PostgreSQL et les Types de Variables C 1059
36.2. Formats d'Entrée Valides pour PGTYPESdate_from_asc 1079
36.3. Formats d'Entrée Valides pour PGTYPESdate_fmt_asc 1081
36.4. Formats d'Entrée Valides pour rdefmtdate 1082
36.5. Formats d'Entrée Valide pour PGTYPEStimestamp_from_asc 1083
37.1. Colonnes de information_schema_catalog_name 1167
37.2. Colonnes de administrable_role_authorizations 1167
37.3. Colonnes de applicable_roles 1168
37.4. Colonnes de attributes 1168
37.5. Colonnes de character_sets 1171
37.6. Colonnes de check_constraint_routine_usage 1171
37.7. Colonnes de check_constraints 1172
37.8. Colonnes de collations 1172
37.9. Colonnes de collation_character_set_applicability 1172
37.10. Colonnes de column_column_usage 1173
37.11. Colonnes de column_domain_usage 1173
37.12. Colonnes de column_options 1174
37.13. Colonnes de column_privileges 1174
37.14. Colonnes de column_udt_usage 1175
37.15. Colonnes de columns 1175
37.16. Colonnes de constraint_column_usage 1179
37.17. Colonnes de constraint_table_usage 1179
37.18. Colonnes de data_type_privileges 1180
37.19. Colonnes de domain_constraints 1181
37.20. Colonnes de domain_udt_usage 1181
37.21. Colonnes de domains 1182
37.22. Colonnes de element_types 1184
37.23. Colonnes de enabled_roles 1186

37.24. Colonnes de foreign_data_wrapper_options 1186
37.25. Colonnes de foreign_data_wrappers 1186
37.26. Colonnes de foreign_server_options 1187
37.27. Colonnes de foreign_servers 1187
37.28. Colonnes de foreign_table_options 1188
37.29. Colonnes de foreign_tables 1188
37.30. Colonnes de key_column_usage 1189
37.31. Colonnes de parameters 1189
37.32. Colonnes de referential_constraints 1191
37.33. Colonnes de role_column_grants 1192
37.34. Colonnes de role_routine_grants 1193
37.35. Colonnes de role_table_grants 1193
37.36. Colonnes de role_udt_grants 1194
37.37. Colonnes de role_usage_grants 1195
37.38. Colonnes de routine_column_usage 1195
37.39. Colonnes de routine_privileges 1196
37.40. Colonnes de routine_routine_usage 1197
37.41. Colonnes de routine_sequence_usage 1197
37.42. Colonnes de routine_table_usage 1198
37.43. Colonnes de routines 1198
37.44. Colonnes de schemata 1203
37.45. Colonnes de sequences 1203
37.46. Colonnes de sql_features 1204
37.47. Colonnes de sql_implementation_info 1205
37.48. Colonnes de sql_parts 1205
37.49. Colonnes de sql_sizing 1206
37.50. Colonnes de table_constraints 1206
37.51. Colonnes de table_privileges 1207
37.52. Colonnes de tables 1207
37.53. Colonnes de transforms 1208
37.54. Colonnes de triggered_update_columns 1209
37.55. Colonnes de triggers 1209
37.56. Colonnes de udt_privileges 1211
37.57. Colonnes de usage_privileges 1212
37.58. Colonnes de user_defined_types 1212
37.59. Colonnes de user_mapping_options 1214
37.60. Colonnes de user_mappings 1214
37.61. Colonnes de view_column_usage 1215
37.62. Colonnes de view_routine_usage 1215
37.63. Colonnes de view_table_usage 1216
37.64. Colonnes de views 1216
38.1. Types polymorphiques 1225
38.2. Équivalence des types C et des types SQL intégrés 1253
38.3. Stratégies B-tree 1291
38.4. Stratégies de découpage 1291
38.5. Stratégies « R-tree » pour GiST à deux dimensions 1292
38.6. Stratégies point SP-GiST 1292
38.7. Stratégies des tableaux GIN 1292
38.8. Stratégies MinMax pour BRIN 1293
38.9. Fonctions d'appui de B-tree 1293
38.10. Fonctions d'appui pour découpage 1294
38.11. Fonctions d'appui pour GiST 1294
38.12. Fonctions de support SP-GiST 1295
38.13. Fonctions d'appui GIN 1295
38.14. Fonctions de support BRIN 1296
40.1. Support des triggers sur événement par commande 1330
43.1. Éléments de diagnostics disponibles 1389
43.2. Diagnostics et erreurs 1404

287. Politiques appliquées par type de commande 1817
288. Variables automatiques de pgbench 2172
289. Opérateurs pgbench 2174
290. pgbench Functions 2176
53.1. Catalogues système 2359
53.2. Les colonnes de pg_aggregate 2361
53.3. Colonnes de pg_am 2362
53.4. Colonnes de pg_amop 2363
53.5. Colonnes de pg_amproc 2364
53.6. Colonnes de pg_attrdef 2364
53.7. Colonnes de pg_attribute 2365
53.8. Colonnes de pg_authid 2367
53.9. Colonnes de pg_auth_members 2368
53.10. Colonnes de pg_cast 2369
53.11. Colonnes de pg_class 2370
53.12. Colonnes de pg_collation 2372
53.13. Colonnes de pg_constraint 2373
53.14. Colonnes de pg_conversion 2375
53.15. Colonnes de pg_database 2376
53.16. Colonnes de pg_db_role_setting 2377
53.17. Colonnes de pg_default_acl 2377
53.18. Colonnes de pg_depend 2378
53.19. Colonnes de pg_description 2380
53.20. Colonnes de pg_enum 2380
53.21. Colonnes de pg_event_trigger 2381
53.22. Colonnes de pg_extension 2381
53.23. Colonnes de pg_foreign_data_wrapper 2382
53.24. Colonnes de pg_foreign_server 2383
53.25. Colonnes de pg_foreign_table 2383
53.26. Colonnes de pg_index 2384
53.27. Colonnes de pg_inherits 2385
53.28. Colonnes de pg_init_privs 2386
53.29. Colonnes de pg_language 2386
53.30. Colonnes de pg_largeobject 2387
53.31. Colonnes de pg_largeobject_metadata 2388
53.32. Colonnes de pg_namespace 2388
53.33. Colonnes de pg_opclass 2388
53.34. Colonnes de pg_operator 2389
53.35. Colonnes de pg_opfamily 2390
53.36. pg_parameter_acl Columns 2391
53.37. Colonnes de pg_partitioned_table 2391
53.38. Colonnes de pg_policy 2392
53.39. Colonnes de pg_proc 2392
53.40. Colonnes de pg_publication 2395
53.41. pg_publication_namespace Columns 2395
53.42. Colonnes de pg_publication_rel 2396
53.43. Colonnes de pg_range 2396
53.44. Colonnes de pg_replication_origin 2397
53.45. Colonnes de pg_rewrite 2397
53.46. Colonnes de pg_seclabel 2398
53.47. Colonnes de pg_sequence 2398
53.48. Colonnes de pg_shdepend 2399
53.49. Colonnes de pg_shdescription 2400
53.50. Colonnes de pg_shseclabel 2401
53.51. Colonnes de pg_statistic 2401
53.52. Colonnes de pg_statistic_ext 2403
53.53. Colonnes de pg_statistic_ext_data 2403
53.54. Colonnes de pg_subscription 2404

53.55. Colonnes de pg_subscription_rel 2405
53.56. Colonnes de pg_tablespace 2405
53.57. Colonnes de pg_transform 2406
53.58. Colonnes de pg_trigger 2406
53.59. Colonnes de pg_ts_config 2408
53.60. Colonnes de pg_ts_config_map 2408
53.61. Colonnes de pg_ts_dict 2409
53.62. Colonnes de pg_ts_parser 2409
53.63. Colonnes de pg_ts_template 2410
53.64. Colonnes de pg_type 2410
53.65. Codes typcategory 2413
53.66. Colonnes de pg_user_mapping 2414
54.1. Vues système 2415
54.2. Colonnes de pg_available_extensions 2416
54.3. Colonnes de pg_available_extension_versions 2417
54.4. Colonnes de pg_backend_memory_contexts 2417
54.5. Colonnes de pg_config 2418
54.6. Colonnes de pg_cursors 2419
54.7. Colonnes de pg_file_settings 2419
54.8. Colonnes de pg_group 2420
54.9. Colonnes de pg_hba_file_rules 2420
54.10. Colonnes de pg_ident_file_mappings 2421
54.11. Colonnes de pg_indexes 2422
54.12. Colonnes de pg_locks 2422
54.13. Colonnes de pg_matviews 2425
54.14. Colonnes de pg_policies 2425
54.15. Colonnes de pg_prepared_statements 2426
54.16. Colonnes de pg_prepared_xacts 2427
54.17. Colonnes de pg_publication_tables 2427
54.18. Colonnes de pg_replication_origin_status 2428
54.19. Colonnes de pg_replication_slots 2428
54.20. Colonnes de pg_roles 2429
54.21. Colonnes de pg_rules 2430
54.22. Colonnes de pg_seclabels 2431
54.23. Colonnes de pg_sequences 2431
54.24. Colonnes de pg_settings 2432
54.25. Colonnes de pg_shadow 2434
54.26. Colonnes de pg_shmem_allocations 2435
54.27. Colonnes de pg_stats 2436
54.28. Colonnes de pg_stats_ext 2437
54.29. Colonnes de pg_stats_ext_exprs 2438
54.30. Colonnes de pg_tables 2440
54.31. Colonnes de pg_timezone_abbrevs 2440
54.32. Colonnes de pg_timezone_names 2441
54.33. Colonnes de pg_user 2441
54.34. Colonnes de pg_user_mappings 2442
54.35. Colonnes de pg_views 2443
68.1. Classes d'opérateur GiST internes 2593
69.1. Classes d'opérateur SP-GiST internes 2611
70.1. Classes d'opérateur GIN natives 2626
71.1. Classe d'opérateur BRIN intégrée 2634
71.2. Fonctions et numéros de support pour les classes d'opérateurs Minmax 2643
71.3. Fonctions et numéros de support pour les classes d'opérateurs d'inclusion 2644
71.4. Procédures et numéros de support pour les classes d'opérateurs Bloom 2645
71.5. Procédure et numéros de support pour les classes d'opérateurs minmax-multi 2646
73.1. Contenu de PGDATA 2649
73.2. Disposition d'une page 2656
73.3. Disposition de PageHeaderData 2656

73.4. Disposition de HeapTupleHeaderData 2657
A.1. Codes d'erreur de PostgreSQL 2690
B.1. Noms de mois 2701
B.2. Noms des jours de la semaine 2701
B.3. Modificateurs de Champs Date/Heure 2702
C.1. Mots clés SQL 2708
F.1. Fonctions de adminpack 2867
F.2. Représentations externes d'un cube 2886
F.3. Opérateurs pour cube 2887
F.4. Fonctions cube 2888
F.5. Fonctions earthdistance par cubes 2926
F.6. Opérateurs earthdistance par points 2927
F.7. Opérateurshstore 2934
F.8. hstore Functions 2935
F.9. Fonctions intarray 2943
F.10. Opérateurs d'intarray 2943
F.11. Types de données isn 2946
F.12. Fonctions de isn 2948
F.13. Opérateurs ltree 2953
F.14. Fonctions ltree 2954
F.15. Colonnes de pg_buffercache 2970
F.16. Algorithmes supportés par crypt() 2973
F.17. Nombre d'itération pour crypt() 2974
F.18. Vitesse de l'algorithm de hachage 2974
F.19. Colonnes de pgrowlocks 2986
F.20. Colonnes de pg_stat_statements 2988
F.21. Colonnes de pg_stat_statements_info 2991
F.22. Colonnes de pgstattuple 2996
F.23. Colonnes de pgstattuple_approx 2999
F.24. Fonctions de pg_trgm 3002
F.25. Opérateurs de pg_trgm 3003
F.26. Représentations externes de seg 3022
F.27. Exemples d'entrées valides de type seg 3022
F.28. Opérateurs GiST du type Seg 3023
F.29. Fonctions Sepgsql 3031
F.30. Fonctions tablefunc 3037
F.31. Paramètres connectby 3045
F.32. Fonctions pour la génération d'UUID 3053
F.33. Fonctions renvoyant des constantes UUID 3053
F.34. Fonctions xml2 3055
F.35. Paramètres de xpath_table 3056
K.1. Limitations de PostgreSQL 3078

Liste des exemples

8.1. Utilisation des types caractère 159
8.2. Utilisation du type `boolean`. 173
8.3. Utiliser les types de chaînes de bits 181
9.1. Feuille de style XSLT pour convertir un résultat SQL/XML en HTML 337
10.1. Résolution du type d'opérateur racine carrée 434
10.2. Résolution de types pour les opérateurs de concaténation de chaînes 434
10.3. Résolution de types pour les opérateurs de valeur absolue et de négation 435
10.4. Résolution du type d'opérateur avec des inclusions de tableaux 435
10.5. Opérateur personnalisé sur un domaine 436
10.6. Résolution de types pour les arguments de la fonction arrondi 438
10.7. Résolution de fonction à arguments variables 439
10.8. Résolution de types pour les fonctions retournant un segment de chaîne 439
10.9. Conversion de types pour le stockage de `character` 441
10.10. Résolution de types avec des types sous-spécifiés dans une union 442
10.11. Résolution de types dans une union simple 442
10.12. Résolution de types dans une union transposée 442
10.13. Résolution de type dans une union imbriquée 443
11.1. Mettre en place un index partiel pour exclure des valeurs courantes 451
11.2. Mettre en place un index partiel pour exclure les valeurs inintéressantes 452
11.3. Mettre en place un index d'unicité partielle 453
11.4. Ne pas utiliser les index partiels comme substitut au partitionnement 454
21.1. Exemple d'entrées de `pg_hba.conf` 722
21.2. Un exemple de fichier `pg_ident.conf` 725
34.1. Premier exemple de programme pour libpq 1026
34.2. Deuxième exemple de programme pour libpq 1028
34.3. Troisième exemple de programme pour libpq 1031
35.1. Exemple de programme sur les « Large Objects » avec libpq 1044
36.1. Programme de Démonstration SQLDA 1100
36.2. Programme ECPG Accédant à un Large Object 1115
42.1. Installation manuelle de PL/Perl 1369
43.1. Mettre entre guillemets des valeurs dans des requêtes dynamiques 1387
43.2. Exceptions avec `UPDATE`/`INSERT` 1403
43.3. Une fonction trigger PL/pgSQL 1418
43.4. Une fonction d'audit par trigger en PL/pgSQL 1419
43.5. Une fonction trigger en PL/pgSQL sur une vue pour un audit 1420
43.6. Une fonction trigger PL/pgSQL pour maintenir une table résumée 1421
43.7. Auditer avec les tables de transition 1423
43.8. Une fonction PL/pgSQL pour un trigger d'événement 1425
43.9. Portage d'une fonction simple de PL/SQL vers PL/pgSQL 1434
43.10. Portage d'une fonction qui crée une autre fonction de PL/SQL vers PL/pgSQL 1434
43.11. Portage d'une procédure avec manipulation de chaînes et paramètres `OUT` de PL/SQL vers PL/pgSQL 1436
43.12. Portage d'une procédure de PL/SQL vers PL/pgSQL 1438
F.1. Créer une table distante pour les journaux applicatifs PostgreSQL au format CSV 2929

Préface

Cet ouvrage représente l'adaptation française de la documentation officielle de PostgreSQL. La version originale a été rédigée par les développeurs de PostgreSQL et des volontaires, en parallèle du développement du logiciel. Elle décrit toutes les fonctionnalités officiellement supportées par la dernière version de PostgreSQL.

Pour faciliter l'accès à la masse d'informations qu'il contient, cet ouvrage est organisé en plusieurs parties. Chacune est destinée à une classe précise d'utilisateurs, ou à des utilisateurs de niveaux d'expertise différents de PostgreSQL :

- la Partie I est une introduction informelle destinée aux nouveaux utilisateurs ;
- la Partie II présente l'environnement du langage de requêtes SQL, notamment les types de données, les fonctions et les optimisations utilisateurs. Tout utilisateur de PostgreSQL devrait la lire ;
- la Partie III décrit l'installation et l'administration du serveur. Quiconque administre un serveur PostgreSQL, à titre privé ou autre, devrait la lire ;
- la Partie IV décrit les interfaces de programmation pour les outils clients de PostgreSQL ;
- la Partie V, destinée aux utilisateurs expérimentés, présente les possibilités d'extension du serveur, notamment les types de données personnalisés et les fonctions utilisateurs ;
- la Partie VI contient la documentation de référence des commandes SQL et des outils client et serveur. Elle sert de référence par les autres parties ;
- la Partie VII contient diverses informations utiles aux développeurs de PostgreSQL.

1. Définition de PostgreSQL

PostgreSQL est un système de gestion de bases de données relationnelles objet (ORDBMS) fondé sur POSTGRES, Version 4.2[1]. Ce dernier a été développé à l'université de Californie au département des sciences informatiques de Berkeley. POSTGRES est à l'origine de nombreux concepts qui ne seront rendus disponibles au sein de systèmes de gestion de bases de données commerciaux que bien plus tard.

PostgreSQL est un descendant libre du code original de Berkeley. Il supporte une grande partie du standard SQL tout en offrant de nombreuses fonctionnalités modernes :

- requêtes complexes ;
- clés étrangères ;
- triggers ;
- vues modifiables ;
- intégrité transactionnelle ;
- contrôle des versions concurrentes (MVCC, acronyme de « MultiVersion Concurrency Control »).

De plus, PostgreSQL peut être étendu par l'utilisateur de multiples façons, en ajoutant, par exemple :

- de nouveaux types de données ;
- de nouvelles fonctions ;
- de nouveaux opérateurs ;
- de nouvelles fonctions d'agrégat ;
- de nouvelles méthodes d'indexage ;
- de nouveaux langages de procédure.

Et grâce à sa licence libérale, PostgreSQL peut être utilisé, modifié et distribué librement, quel que soit le but visé, qu'il soit privé, commercial ou académique.

[1] https://dsf.berkeley.edu/postgres.html

2. Bref historique de PostgreSQL

Le système de bases de données relationnelles objet PostgreSQL est issu de POSTGRES, programme écrit à l'université de Californie à Berkeley. Après des dizaines d'années de développement, PostgreSQL annonce être devenu la base de données libre de référence.

2.1. Le projet POSTGRES à Berkeley

Le projet POSTGRES, mené par le professeur Michael Stonebraker, était sponsorisé par le DARPA (acronyme de *Defense Advanced Research Projects Agency*), l'ARO (acronyme de *Army Research Office*), la NSF (acronyme de *National Science Foundation*) et ESL, Inc. Le développement de POSTGRES a débuté en 1986. Les concepts initiaux du système ont été présentés dans [ston86] et la définition du modèle de données initial apparut dans [rowe87]. Le système de règles fut décrit dans [ston87a] et l'architecture du gestionnaire de stockage dans [ston87b].

Depuis, plusieurs versions majeures de POSTGRES ont vu le jour. La première « démo » devint opérationnelle en 1987 et fut présentée en 1988 lors de la conférence ACM-SIGMOD. La version 1, décrite dans [ston90a], fut livrée à quelques utilisateurs externes en juin 1989. Suite à la critique du premier mécanisme de règles ([ston89]), celui-ci fut réécrit ([ston90b]) pour la version 2, présentée en juin 1990. La version 3 apparut en 1991. Elle apporta le support de plusieurs gestionnaires de stockage, un exécuteur de requêtes amélioré et une réécriture du gestionnaire de règles. La plupart des versions qui suivirent, jusqu'à Postgres95 (voir plus loin), portèrent sur la portabilité et la fiabilité.

POSTGRES fut utilisé dans plusieurs applications, en recherche et en production. On peut citer, par exemple : un système d'analyse de données financières, un programme de suivi des performances d'un moteur à réaction, une base de données de suivi d'astéroïdes, une base de données médicale et plusieurs systèmes d'informations géographiques. POSTGRES a aussi été utilisé comme support de formation dans plusieurs universités. Illustra Information Technologies (devenu Informix[2], maintenant détenu par IBM[3]) a repris le code et l'a commercialisé. Fin 1992, POSTGRES est devenu le gestionnaire de données principal du projet de calcul scientifique Sequoia 2000 décrit dans [ston92].

La taille de la communauté d'utilisateurs doubla quasiment au cours de l'année 1993. De manière évidente, la maintenance du prototype et le support prenaient un temps considérable, temps qui aurait dû être employé à la recherche en bases de données. Dans un souci de réduction du travail de support, le projet POSTGRES de Berkeley se termina officiellement avec la version 4.2.

2.2. Postgres95

En 1994, Andrew Yu et Jolly Chen ajoutèrent un interpréteur de langage SQL à POSTGRES. Sous le nouveau nom de Postgres95, le projet fut publié sur le Web comme descendant libre (Open Source) du code source initial de POSTGRES, version Berkeley.

Le code de Postgres95 était écrit en pur C ANSI et réduit de 25%. De nombreux changements internes améliorèrent les performances et la maintenabilité. Les versions 1.0.x de Postgres95 passèrent le Wisconsin Benchmark avec des performances meilleures de 30 à 50% par rapport à POSTGRES, version 4.2. À part les correctifs de bogues, les principales améliorations furent les suivantes :

- le langage PostQUEL est remplacé par SQL (implémenté sur le serveur). (La bibliothèque d'interface libpq a été nommée à partir du langage PostQUEL.) Les requêtes imbriquées n'ont pas été supportées avant PostgreSQL (voir plus loin), mais elles pouvaient être imitées dans Postgres95 à l'aide de fonctions SQL utilisateur ; les agrégats furent reprogrammés, la clause GROUP BY ajoutée ;

- un nouveau programme, psql, qui utilise GNU Readline, permet l'exécution interactive de requêtes SQL ; c'est la fin du programme monitor ;

[2] https://www.ibm.com/analytics/informix
[3] https://www.ibm.com/

- une nouvelle bibliothèque cliente, `libpgtcl`, supporte les programmes écrits en Tcl ; un shell exemple, `pgtclsh`, fournit de nouvelles commandes Tcl pour interfacer des programmes Tcl avec Postgres95 ;
- l'interface de gestion des « Large Objects » est réécrite ; jusque-là, le seul mécanisme de stockage de ces objets passait par le système de fichiers Inversion (« Inversion file system ») ; ce système est abandonné ;
- le système de règles d'instance est supprimé ; les règles sont toujours disponibles en tant que règles de réécriture ;
- un bref tutoriel présentant les possibilités du SQL ainsi que celles spécifiques à Postgres95 est distribué avec les sources ;
- la version GNU de make est utilisée pour la construction à la place de la version BSD ; Postgres95 peut également être compilé avec un GCC sans correctif (l'alignement des doubles est corrigé).

2.3. PostgreSQL

En 1996, le nom « Postgres95 » commence à mal vieillir. Le nom choisi, PostgreSQL, souligne le lien entre POSTGRES et les versions suivantes qui intègrent le SQL. En parallèle, la version est numérotée 6.0 pour reprendre la numérotation du projet POSTGRES de Berkeley.

Postgres est toujours considéré comme le nom officiel du projet, à la fois à cause de la tradition et parce que les gens le trouvent plus simple à prononcer Postgres que PostgreSQL.

Lors du développement de Postgres95, l'effort était axé sur l'identification et la compréhension des problèmes dans le code. Avec PostgreSQL, l'accent est mis sur les nouvelles fonctionnalités, sans pour autant abandonner les autres domaines.

L'historique de PostgreSQL à partir de ce moment est disponible dans l'Annexe E.

3. Conventions

Les conventions suivantes sont utilisées dans le synopsis d'une commande : les crochets (`[` et `]`) indiquent des parties optionnelles. Les accolades (`{` et `}`) et les barres verticales (`|`) indiquent un choix entre plusieurs options. Les points de suspension (`...`) signifient que l'élément précédent peut être répété. Tous les autres symboles, ceci incluant les parenthèses, devraient être acceptés directement.

Lorsque cela améliore la clarté, les commandes SQL sont précédées d'une invite `=>`, tandis que les commandes shell le sont par `$`. Dans le cadre général, les invites ne sont pas indiquées.

Un *administrateur* est généralement une personne en charge de l'installation et de la bonne marche du serveur. Un *utilisateur* est une personne qui utilise ou veut utiliser une partie quelconque du système PostgreSQL. Ces termes ne doivent pas être pris trop à la lettre ; cet ouvrage n'a pas d'avis figé sur les procédures d'administration système.

4. Pour plus d'informations

En dehors de la documentation, il existe d'autres ressources concernant PostgreSQL :

Wiki

Le wiki[4] de PostgreSQL contient la FAQ[5] (liste des questions fréquemment posées), la liste TODO[6] et des informations détaillées sur de nombreux autres thèmes.

[4] https://wiki.postgresql.org
[5] https://wiki.postgresql.org/wiki/Frequently_Asked_Questions

Site web

Le site web[7] de PostgreSQL contient des détails sur la dernière version, et bien d'autres informations pour rendre un travail ou un investissement personnel avec PostgreSQL plus productif.

Listes de discussion

Les listes de discussion constituent un bon endroit pour trouver des réponses à ses questions, pour partager ses expériences avec celles d'autres utilisateurs et pour contacter les développeurs. La consultation du site web de PostgreSQL fournit tous les détails.

Soi-même !

PostgreSQL est un projet Open Source. En tant que tel, le support dépend de la communauté des utilisateurs. Lorsque l'on débute avec PostgreSQL, on est tributaire de l'aide des autres, soit au travers de la documentation, soit par les listes de discussion. Il est important de faire partager à son tour ses connaissances par la lecture des listes de discussion et les réponses aux questions. Lorsque quelque chose est découvert qui ne figurait pas dans la documentation, pourquoi ne pas en faire profiter les autres ? De même lors d'ajout de fonctionnalités au code.

5. Lignes de conduite pour les rapports de bogues

Lorsque vous trouvez un bogue dans PostgreSQL, nous voulons en entendre parler. Vos rapports de bogues jouent un rôle important pour rendre PostgreSQL plus fiable, car même avec la plus grande attention, nous ne pouvons pas garantir que chaque partie de PostgreSQL fonctionnera sur toutes les plates-formes et dans toutes les circonstances.

Les suggestions suivantes ont pour but de vous former à la saisie d'un rapport de bogue qui pourra ensuite être géré de façon efficace. Il n'est pas requis de les suivre, mais ce serait à l'avantage de tous.

Nous ne pouvons pas promettre de corriger tous les bogues immédiatement. Si le bogue est évident, critique ou affecte un grand nombre d'utilisateurs, il y a de grandes chances pour que quelqu'un s'en charge. Il se peut que nous vous demandions d'utiliser une version plus récente pour vérifier si le bogue est toujours présent. Ou nous pourrions décider que le bogue ne peut être corrigé avant qu'une réécriture massive, que nous avions planifiée, ne soit faite. Ou peut-être est-ce trop difficile et que des choses plus importantes nous attendent. Si vous avez besoin d'aide immédiatement, envisagez l'obtention d'un contrat de support commercial.

5.1. Identifier les bogues

Avant de rapporter un bogue, merci de lire et relire la documentation pour vérifier que vous pouvez réellement faire ce que vous essayez de faire. Si ce n'est pas clair, rapportez-le aussi ; c'est un bogue dans la documentation. S'il s'avère que le programme fait différemment de ce qu'indique la documentation, c'est un bogue. Ceci peut inclure les circonstances suivantes, sans s'y limiter :

- Un programme se terminant avec un signal fatal ou un message d'erreur du système d'exploitation qui indiquerait un problème avec le programme. (Un contre-exemple pourrait être le message « disk full », disque plein, car vous devez le régler vous-même.)
- Un programme produit une mauvaise sortie pour une entrée donnée.
- Un programme refuse d'accepter une entrée valide (c'est-à-dire telle que définie dans la documentation).

[6] https://wiki.postgresql.org/wiki/Todo
[7] https://www.postgresql.org

- Un programme accepte une entrée invalide sans information ou message d'erreur. Mais gardez en tête que votre idée d'entrée invalide pourrait être notre idée d'une extension ou d'une compatibilité avec les pratiques traditionnelles.

- PostgreSQL échoue à la compilation, à la construction ou à l'installation suivant les instructions des plates-formes supportées.

Ici, « programme » fait référence à un exécutable, pas au moteur du serveur.

Une lenteur ou une absorption des ressources n'est pas nécessairement un bogue. Lisez la documentation ou demandez sur une des listes de discussion de l'aide concernant l'optimisation de vos applications. Ne pas se conformer au standard SQL n'est pas nécessairement un bogue, sauf si une telle conformité est indiquée explicitement.

Avant de continuer, vérifiez sur la liste des choses à faire ainsi que dans la FAQ pour voir si votre bogue n'est pas déjà connu. Si vous n'arrivez pas à décoder les informations sur la liste des choses à faire, écrivez un rapport. Le minimum que nous puissions faire est de rendre cette liste plus claire.

5.2. Que rapporter ?

Le point le plus important à se rappeler avec les rapports de bogues est de donner tous les faits et seulement les faits. Ne spéculez pas sur ce que vous pensez qui ne va pas, sur ce qu'« il semble faire » ou sur quelle partie le programme a une erreur. Si vous n'êtes pas familier avec l'implémentation, vous vous tromperez probablement et vous ne nous aiderez pas. Et même si vous avez raison, des explications complètes sont un bon supplément, mais elles ne doivent pas se substituer aux faits. Si nous pensons corriger le bogue, nous devons toujours le reproduire nous-mêmes. Rapporter les faits stricts est relativement simple (vous pouvez probablement copier/coller à partir de l'écran) mais, trop souvent, des détails importants sont oubliés parce que quelqu'un a pensé qu'ils n'avaient pas d'importance ou que le rapport serait compris.

Les éléments suivants devraient être fournis avec chaque rapport de bogue :

- La séquence exacte des étapes nécessaires pour reproduire le problème *à partir du lancement du programme*. Ceci devrait se suffire ; il n'est pas suffisant d'envoyer une simple instruction `SELECT` sans les commandes `CREATE TABLE` et `INSERT` qui ont précédé, si la sortie devait dépendre des données contenues dans les tables. Nous n'avons pas le temps de comprendre le schéma de votre base de données. Si nous sommes supposés créer nos propres données, nous allons probablement ne pas voir le problème.

  Le meilleur format pour un test suite à un problème relatif à SQL est un fichier qui peut être lancé via l'interface psql et qui montrera le problème. (Assurez-vous de ne rien avoir dans votre fichier de lancement `~/.psqlrc`.) Un moyen facile pour créer ce fichier est d'utiliser pg_dump pour récupérer les déclarations des tables ainsi que les données nécessaires pour mettre en place la scène. Il ne reste plus qu'à ajouter la requête posant problème. Vous êtes encouragé à minimiser la taille de votre exemple, mais ce n'est pas une obligation. Si le bogue est reproductible, nous le trouverons de toute façon.

  Si votre application utilise une autre interface client, telle que PHP, alors essayez d'isoler le problème aux requêtes erronées. Nous n'allons certainement pas mettre en place un serveur web pour reproduire votre problème. Dans tous les cas, rappelez-vous d'apporter les fichiers d'entrée exacts ; n'essayez pas de deviner que le problème se pose pour les « gros fichiers », pour les « bases de données de moyenne taille », etc., car cette information est trop inexacte, subjective pour être utile.

- La sortie que vous obtenez. Merci de ne pas dire que cela « ne fonctionne pas » ou s'est « arrêté brutalement ». S'il existe un message d'erreur, montrez-le même si vous ne le comprenez pas. Si le programme se termine avec une erreur du système d'exploitation, dites-le. Même si le résultat de votre test est un arrêt brutal du programme ou un autre souci évident, il pourrait ne pas survenir sur notre plateforme. Le plus simple est de copier directement la sortie du terminal, si possible.

> **Note**
>
> Si vous rapportez un message d'erreur, merci d'obtenir la forme la plus verbeuse de ce message. Avec psql, exécutez `\set VERBOSITY verbose` avant tout. Si vous récupérez le message des traces du serveur, initialisez la variable d'exécution log_error_verbosity avec `verbose` pour que tous les détails soient tracés.

> **Note**
>
> Dans le cas d'erreurs fatales, le message d'erreur rapporté par le client pourrait ne pas contenir toutes les informations disponibles. Jetez aussi un œil aux traces du serveur de la base de données. Si vous ne conservez pas les traces de votre serveur, c'est le bon moment pour commencer à le faire.

- Il est très important de préciser ce que vous attendez en sortie. Si vous écrivez uniquement « Cette commande m'a donné cette réponse. » ou « Ce n'est pas ce que j'attendais. », nous pourrions le lancer nous-mêmes, analyser la sortie et penser que tout est correct, car cela correspond exactement à ce que nous attendions. Nous ne devrions pas avoir à passer du temps pour décoder la sémantique exacte de vos commandes. Tout spécialement, ne vous contentez pas de dire que « Ce n'est pas ce que SQL spécifie/Oracle fait. » Rechercher le comportement correct à partir de SQL n'est pas amusant et nous ne connaissons pas le comportement de tous les autres serveurs de bases de données relationnelles. (Si votre problème est un arrêt brutal du serveur, vous pouvez évidemment omettre cet élément.)

- Toutes les options en ligne de commande ainsi que les autres options de lancement incluant les variables d'environnement ou les fichiers de configuration que vous avez modifiées. Encore une fois, soyez exact. Si vous utilisez une distribution prépackagée qui lance le serveur au démarrage, vous devriez essayer de retrouver ce que cette distribution fait.

- Tout ce que vous avez fait de différent à partir des instructions d'installation.

- La version de PostgreSQL. Vous pouvez lancer la commande `SELECT version();` pour trouver la version du serveur sur lequel vous êtes connecté. La plupart des exécutables disposent aussi d'une option `--version` ; `postgres --version` et `psql --version` devraient au moins fonctionner. Si la fonction ou les options n'existent pas, alors votre version est bien trop ancienne et vous devez mettre à jour. Si vous avez lancé une version préparée sous forme de paquets, tels que les RPM, dites-le en incluant la sous-version que le paquet pourrait avoir. Si vous êtes sur une version Git, mentionnez-le en indiquant le hachage du commit.

  Si votre version est antérieure à la 15.19, nous allons certainement vous demander de mettre à jour. Beaucoup de corrections de bogues et d'améliorations sont apportées dans chaque nouvelle version, donc il est bien possible qu'un bogue rencontré dans une ancienne version de PostgreSQL soit déjà corrigé. Nous ne fournissons qu'un support limité pour les sites utilisant d'anciennes versions de PostgreSQL ; si vous avez besoin de plus de support que ce que nous fournissons, considérez l'acquisition d'un contrat de support commercial.

- Informations sur la plateforme. Ceci inclut le nom du noyau et sa version, bibliothèque C, processeur, mémoires et ainsi de suite. Dans la plupart des cas, il est suffisant de préciser le vendeur et la version, mais ne supposez pas que tout le monde sait ce que « Debian » contient ou que tout le monde utilise des x86_64. Si vous avez des problèmes à l'installation, des informations sur l'ensemble des outils de votre machine (compilateurs, `make`, etc.) sont aussi nécessaires.

N'ayez pas peur si votre rapport de bogue devient assez long. C'est un fait. Il est préférable de rapporter tous les faits la première fois plutôt que nous ayons à vous tirer les vers du nez. D'un autre côté, si vos

fichiers d'entrée sont trop gros, il est préférable de demander si quelqu'un souhaite s'y plonger. Voici un article[8] qui donne quelques autres conseils sur les rapports de bogues.

Ne passez pas tout votre temps à vous demander quelles modifications apporter pour que le problème s'en aille. Ceci ne nous aidera probablement pas à le résoudre. S'il arrive que le bogue ne puisse pas être corrigé immédiatement, vous aurez toujours l'opportunité de chercher ceci et de partager vos trouvailles. De même, encore une fois, ne perdez pas votre temps à deviner pourquoi le bogue existe. Nous le trouverons assez rapidement.

Lors de la rédaction d'un rapport de bogue, merci de choisir une terminologie qui ne laisse pas place aux confusions. Le paquet logiciel en totalité est appelé « PostgreSQL », quelquefois « Postgres » en court. Si vous parlez spécifiquement du serveur, mentionnez-le, mais ne dites pas seulement « PostgreSQL a planté ». Un arrêt brutal d'un seul processus serveur est assez différent de l'arrêt brutal du « postgres » père ; merci de ne pas dire que « le serveur a planté » lorsque vous voulez dire qu'un seul processus s'est arrêté, ni vice versa. De plus, les programmes clients tels que l'interface interactive « psql » sont complètement séparés du moteur. Essayez d'être précis sur la provenance du problème : client ou serveur.

5.3. Où rapporter des bogues ?

En général, envoyez vos rapports de bogue à la liste de discussion des rapports de bogue (<pgsql-bogues@lists.postgresql.org>). Nous vous demandons d'utiliser un sujet descriptif pour votre courrier électronique, par exemple une partie du message d'erreur.

Une autre méthode consiste à remplir le formulaire web disponible sur le site web[9] du projet. Saisir un rapport de bogue de cette façon fait que celui-ci est envoyé à la liste de discussion <pgsql-bogues@lists.postgresql.org>.

Si votre rapport de bogue a des implications sur la sécurité et que vous préféreriez qu'il ne soit pas immédiatement visible dans les archives publiques, ne l'envoyez pas sur pgsql-bugs. Les problèmes de sécurité peuvent être rapportés de façon privée sur <security@lists.postgresql.org>.

N'envoyez pas de rapports de bogue aux listes de discussion des utilisateurs, comme <pgsql-sql@lists.postgresql.org> ou <pgsql-general@lists.postgresql.org>. Ces listes de discussion servent à répondre aux questions des utilisateurs et les abonnés ne souhaitent pas recevoir de rapports de bogue. Plus important, ils ont peu de chance de les corriger.

De même, n'envoyez *pas* vos rapports de bogue à la liste de discussion des développeurs <pgsql-hackers@lists.postgresql.org>. Cette liste sert aux discussions concernant le développement de PostgreSQL et il serait bon de conserver les rapports de bogue séparément. Nous pourrions choisir de discuter de votre rapport de bogue sur pgsql-hackers si le problème nécessite que plus de personnes s'en occupent.

Si vous avez un problème avec la documentation, le meilleur endroit pour le rapporter est la liste de discussion pour la documentation <pgsql-docs@lists.postgresql.org>. Soyez précis sur la partie de la documentation qui vous déplaît.

Si votre bogue concerne un problème de portabilité sur une plateforme non supportée, envoyez un courrier électronique à <pgsql-hackers@lists.postgresql.org>, pour que nous puissions travailler sur le portage de PostgreSQL sur votre plateforme.

> **Note**
>
> Malheureusement, à cause du grand nombre de pourriels (*spam*), toutes les adresses de courrier électronique ci-dessus sont modérées sauf si vous vous êtes abonné. Ceci signifie qu'il y aura

[8] https://www.chiark.greenend.org.uk/~sgtatham/bugs.html
[9] https://www.postgresql.org/

un certain délai avant que l'email ne soit délivré. Si vous souhaitez souscrire aux listes, merci de visiter https://lists.postgresql.org/ pour les instructions.

Partie I. Tutoriel

Bienvenue dans le tutoriel de PostgreSQL. Les chapitres suivants présentent une courte introduction à PostgreSQL, aux concepts des bases de données relationnelles et au langage SQL à ceux qui débutent dans l'un de ces domaines. Seules sont nécessaires des connaissances générales sur l'utilisation des ordinateurs. Aucune expérience particulière d'Unix ou en programmation n'est requise. Ce tutoriel a surtout pour but de vous faire acquérir une expérience pratique des aspects importants du système PostgreSQL. Il n'est ni exhaustif ni complet.

À la suite de ce tutoriel, la lecture de la Partie II permettra d'acquérir une connaissance plus complète du langage SQL, celle de la Partie IV des informations sur le développement d'applications pour PostgreSQL. Ceux qui montent et gèrent leur propre serveur devraient aussi lire la Partie III.

Table des matières

1. Démarrage 3
    1.1. Installation 3
    1.2. Concepts architecturaux de base 3
    1.3. Création d'une base de données 4
    1.4. Accéder à une base 5
2. Le langage SQL 7
    2.1. Introduction 7
    2.2. Concepts 7
    2.3. Créer une nouvelle table 7
    2.4. Remplir une table avec des lignes 8
    2.5. Interroger une table 9
    2.6. Jointures entre les tables 11
    2.7. Fonctions d'agrégat 13
    2.8. Mises à jour 15
    2.9. Suppressions 15
3. Fonctionnalités avancées 16
    3.1. Introduction 16
    3.2. Vues 16
    3.3. Clés étrangères 16
    3.4. Transactions 17
    3.5. Fonctions de fenêtrage 19
    3.6. Héritage 22
    3.7. Conclusion 23

Chapitre 1. Démarrage

1.1. Installation

Avant de pouvoir utiliser PostgreSQL, vous devez l'installer. Il est possible que PostgreSQL soit déjà installé dans votre environnement, soit parce qu'il est inclus dans votre distribution, soit parce que votre administrateur système s'en est chargé. Dans ce cas, vous devriez obtenir les informations nécessaires pour accéder à PostgreSQL dans la documentation de votre distribution ou de la part de votre administrateur.

Si vous n'êtes pas sûr que PostgreSQL soit déjà disponible ou que vous puissiez l'utiliser pour vos tests, vous avez la possibilité de l'installer vous-même. Le faire n'est pas difficile et peut être un bon exercice. PostgreSQL peut être installé par n'importe quel utilisateur sans droit particulier. Aucun accès administrateur (root) n'est requis.

Si vous installez PostgreSQL vous-même, référez-vous au Chapitre 17, pour les instructions sur l'installation, puis revenez à ce guide quand l'installation est terminée. Nous vous conseillons de suivre attentivement la section sur la configuration des variables d'environnement appropriées.

Si votre administrateur n'a pas fait une installation par défaut, vous pouvez avoir à effectuer un paramétrage supplémentaire. Par exemple, si le serveur de bases de données est une machine distante, vous aurez besoin de configurer la variable d'environnement `PGHOST` avec le nom du serveur de bases de données. Il sera aussi peut-être nécessaire de configurer la variable d'environnement `PGPORT`. La démarche est la suivante : si vous essayez de démarrer un programme et qu'il se plaint de ne pas pouvoir se connecter à la base de données, vous devez consulter votre administrateur ou, si c'est vous, la documentation pour être sûr que votre environnement est correctement paramétré. Si vous n'avez pas compris le paragraphe précédent, lisez donc la prochaine section.

1.2. Concepts architecturaux de base

Avant de continuer, vous devez connaître les bases de l'architecture système de PostgreSQL. Comprendre comment les parties de PostgreSQL interagissent entre elles rendra ce chapitre un peu plus clair.

Dans le jargon des bases de données, PostgreSQL utilise un modèle client/serveur. Une session PostgreSQL est le résultat de la coopération des processus (programmes) suivants :

- Un processus serveur, qui gère les fichiers de la base de données, accepte les connexions à la base de la part des applications clientes et effectue sur la base les actions des clients. Le programme serveur est appelé `postgres`.
- L'application cliente (l'application de l'utilisateur), qui veut effectuer des opérations sur la base de données. Les applications clientes peuvent être de natures très différentes : un client peut être un outil texte, une application graphique, un serveur web qui accède à la base de données pour afficher des pages web ou un outil spécialisé dans la maintenance de bases de données. Certaines applications clientes sont fournies avec PostgreSQL ; la plupart sont développées par les utilisateurs.

Comme souvent avec les applications client/serveur, le client et le serveur peuvent être sur des hôtes différents. Dans ce cas, ils communiquent à travers une connexion réseau TCP/IP. Vous devez garder cela à l'esprit, car les fichiers qui sont accessibles sur la machine cliente peuvent ne pas l'être (ou l'être seulement en utilisant des noms de fichiers différents) sur la machine exécutant le serveur de bases de données.

Le serveur PostgreSQL peut traiter de multiples connexions simultanées depuis les clients. Dans ce but, il démarre un nouveau processus pour chaque connexion. À ce moment, le client et le nouveau processus serveur communiquent sans intervention de la part du processus `postgres` original. Ainsi, le processus serveur principal s'exécute toujours, attendant de nouvelles connexions clientes, tandis

que le client et les processus serveur associés vont et viennent (bien sûr, tout ceci est invisible pour l'utilisateur ; nous le mentionnons ici seulement par exhaustivité).

1.3. Création d'une base de données

Le premier test pour voir si vous pouvez accéder au serveur de bases de données consiste à essayer de créer une base. Un serveur PostgreSQL peut gérer plusieurs bases de données. Généralement, une base de données distincte est utilisée pour chaque projet ou pour chaque utilisateur.

Il est possible que votre administrateur ait déjà créé une base pour vous. Dans ce cas, vous pouvez omettre cette étape et aller directement à la prochaine section.

Pour créer une nouvelle base, nommée `ma_base` dans cet exemple, utilisez la commande suivante :

```
$ createdb ma_base
```

Si cette commande ne fournit aucune réponse, cette étape est réussie et vous pouvez sauter le reste de cette section.

Si vous voyez un message similaire à :

```
createdb: command not found
```

alors PostgreSQL n'a pas été installé correctement. Soit il n'a pas été installé du tout, soit le chemin système n'a pas été configuré pour l'inclure. Essayez d'appeler la commande avec le chemin absolu :

```
$ /usr/local/pgsql/bin/createdb ma_base
```

Le chemin sur votre serveur peut être différent. Contactez votre administrateur ou vérifiez dans les instructions d'installation pour corriger la commande.

Voici une autre réponse possible :

```
createdb: error: connection to server on socket "/
tmp/.s.PGSQL.5432" failed: No such file or directory
        Is the server running locally and accepting connections on
 that socket?
```

Cela signifie que le serveur n'était pas démarré, ou qu'il n'écoutait pas là où `createdb` s'attendait à le contacter. Une fois encore, vérifiez les instructions d'installation ou consultez votre administrateur.

Voici encore une autre réponse possible :

```
createdb: error: connection to server on socket "/
tmp/.s.PGSQL.5432" failed: FATAL:  role "joe" does not exist
```

mais avec votre propre nom de connexion mentionné à la place de joe. Ceci survient si l'administrateur n'a pas créé de compte utilisateur PostgreSQL pour vous (les comptes utilisateurs PostgreSQL sont distincts de ceux du système d'exploitation). Si vous êtes l'administrateur, la lecture du Chapitre 22 vous expliquera comment créer de tels comptes. Vous aurez besoin de prendre l'identité de l'utilisateur du système d'exploitation sous lequel PostgreSQL a été installé (généralement `postgres`) pour créer le compte du premier utilisateur. Cela pourrait aussi signifier que vous avez un nom d'utilisateur PostgreSQL qui est différent de celui de votre compte utilisateur du système d'exploitation. Dans ce cas, vous avez besoin d'utiliser l'option `-U` ou de configurer la variable d'environnement `PGUSER` pour spécifier votre nom d'utilisateur PostgreSQL.

Si vous n'avez pas les droits requis pour créer une base, vous verrez le message suivant :

```
createdb: error: database creation failed: ERROR:  permission
 denied to create database
```

Tous les utilisateurs n'ont pas l'autorisation de créer de nouvelles bases de données. Si PostgreSQL refuse de créer des bases pour vous, alors il faut que l'administrateur vous accorde ce droit. Consultez votre administrateur si cela arrive. Si vous avez installé vous-même l'instance PostgreSQL, alors vous devez ouvrir une session sous le compte utilisateur que vous avez utilisé pour démarrer le serveur. [1]

Vous pouvez aussi créer des bases de données avec d'autres noms. PostgreSQL vous permet de créer un nombre quelconque de bases sur un site donné. Le nom des bases doit avoir comme premier caractère un caractère alphabétique et est limité à 63 octets de longueur. Un choix pratique est de créer une base avec le même nom que votre nom d'utilisateur courant. Beaucoup d'outils utilisent ce nom comme nom par défaut pour la base : cela permet de gagner du temps en saisie. Pour créer cette base, tapez simplement :

```
$ createdb
```

Si vous ne voulez plus utiliser votre base, vous pouvez la supprimer. Par exemple, si vous êtes le propriétaire (créateur) de la base `ma_base`, vous pouvez la détruire en utilisant la commande suivante :

```
$ dropdb ma_base
```

(Pour cette commande, le nom de la base n'est pas par défaut le nom du compte utilisateur. Vous devez toujours en spécifier un.) Cette action supprime physiquement tous les fichiers associés avec la base de données et elle ne peut pas être annulée, donc cela doit se faire avec beaucoup de prudence.

createdb et dropdb apportent beaucoup plus d'informations sur `createdb` et `dropdb`.

1.4. Accéder à une base

Une fois que vous avez créé la base, vous pouvez y accéder :

- Démarrez le programme en ligne de commande de PostgreSQL, appelé *psql*, qui vous permet de saisir, d'éditer et d'exécuter de manière interactive des commandes SQL.
- Utilisez un outil existant avec une interface graphique comme pgAdmin ou une suite bureautique avec un support ODBC ou JDBC pour créer et manipuler une base. Ces possibilités ne sont pas couvertes dans ce tutoriel.
- Écrivez une application personnalisée en utilisant un des nombreux langages disponibles. Ces possibilités sont davantage examinées dans la Partie IV.

Vous aurez probablement besoin de lancer `psql` pour essayer les exemples de ce tutoriel. Pour cela, saisissez la commande suivante :

```
$ psql ma_base
```

Si vous n'indiquez pas le nom de la base, alors `psql` utilisera par défaut le nom de votre compte utilisateur. Vous avez déjà découvert ce principe dans la section précédente en utilisant `createdb`.

Dans `psql`, vous serez accueilli avec le message suivant :

```
psql (15.19)
Type "help" for help.

ma_base=>
```

La dernière ligne peut aussi être :

[1] Quelques explications : les noms d'utilisateurs de PostgreSQL sont différents des comptes utilisateurs du système d'exploitation. Quand vous vous connectez à une base de données, vous pouvez choisir le nom d'utilisateur PostgreSQL que vous utilisez. Si vous ne spécifiez rien, cela sera par défaut le même nom que votre compte système courant. En fait, il existe toujours un compte utilisateur PostgreSQL qui a le même nom que l'utilisateur du système d'exploitation qui a démarré le serveur, et cet utilisateur a toujours le droit de créer des bases. Au lieu de vous connecter au système en tant que cet utilisateur, vous pouvez spécifier partout l'option `-U` pour sélectionner un nom d'utilisateur PostgreSQL sous lequel vous connecter.

```
ma_base=#
```

Cela veut dire que vous êtes le superutilisateur de la base de données, ce qui est souvent le cas si vous avez installé PostgreSQL vous-même. Être superutilisateur ou administrateur signifie que vous n'êtes pas sujet aux contrôles d'accès. Concernant ce tutoriel, cela n'a pas d'importance.

Si vous rencontrez des problèmes en exécutant `psql`, alors retournez à la section précédente. Les diagnostics de `psql` et de `createdb` sont semblables. Si le dernier fonctionnait, alors le premier devrait fonctionner également.

La dernière ligne affichée par `psql` est l'invite. Cela indique que `psql` est à l'écoute et que vous pouvez saisir des requêtes SQL dans l'espace de travail maintenu par `psql`. Essayez ces commandes :

```
ma_base=> SELECT version();
                                              version
-------------------------------------------------------------------------------
 PostgreSQL 15.19 on x86_64-pc-linux-gnu, compiled by gcc (Debian
 4.9.2-10) 4.9.2, 64-bit
(1 row)

ma_base=> SELECT current_date;
    date
------------
 2016-01-07
(1 row)

ma_base=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

Le programme `psql` dispose d'un certain nombre de commandes internes qui ne sont pas des commandes SQL. Elles commencent avec le caractère antislash (une barre oblique inverse, « \ »). Par exemple, vous pouvez obtenir de l'aide sur la syntaxe de nombreuses commandes SQL de PostgreSQL en exécutant :

```
ma_base=> \h
```

Pour sortir de `psql`, saisissez :

```
ma_base=> \q
```

et `psql` se terminera et vous ramènera à votre shell. Pour plus de commandes internes, saisissez `\?` à l'invite de `psql`. Les possibilités complètes de `psql` sont documentées dans psql. Dans ce tutoriel, nous ne verrons pas ces caractéristiques explicitement, mais vous pouvez les utiliser vous-même quand cela vous est utile.

Chapitre 2. Le langage SQL

2.1. Introduction

Ce chapitre fournit un panorama sur la façon d'utiliser SQL pour exécuter des opérations simples. Ce tutoriel est seulement prévu pour vous donner une introduction et n'est, en aucun cas, un tutoriel complet sur SQL. De nombreux livres ont été écrits sur SQL, incluant [melt93] et [date97]. Certaines caractéristiques du langage de PostgreSQL sont des extensions de la norme.

Dans les exemples qui suivent, nous supposons que vous avez créé une base de données appelée `ma_base`, comme cela a été décrit dans le chapitre précédent et que vous avez été capable de lancer psql.

Les exemples dans ce manuel peuvent aussi être trouvés dans le répertoire `src/tutorial/` de la distribution source de PostgreSQL. (Les distributions binaires de PostgreSQL pourraient ne pas fournir ces fichiers.) Pour utiliser ces fichiers, commencez par changer de répertoire et lancez make :

```
$ cd .../src/tutorial
$ make
```

Ceci crée les scripts et compile les fichiers C contenant des fonctions et types définis par l'utilisateur. Puis, pour lancer le tutoriel, faites ce qui suit :

```
$ psql -s ma_base
...

ma_base=> \i basics.sql
```

La commande `\i` de `psql` lit les commandes depuis le fichier spécifié. L'option `-s` vous place dans un mode pas à pas qui fait une pause avant d'envoyer chaque instruction au serveur. Les commandes utilisées dans cette section sont dans le fichier `basics.sql`.

2.2. Concepts

PostgreSQL est un *système de gestion de bases de données relationnelles* (SGBDR). Cela signifie que c'est un système pour gérer des données stockées dans des *relations*. Relation est essentiellement un terme mathématique pour *table*. La notion de stockage de données dans des tables est si commune aujourd'hui que cela peut sembler en soi évident, mais il y a de nombreuses autres manières d'organiser des bases de données. Les fichiers et répertoires dans les systèmes d'exploitation de type Unix forment un exemple de base de données hiérarchique. Un développement plus moderne est une base de données orientée objet.

Chaque table est un ensemble de *lignes*. Chaque ligne d'une table donnée a le même ensemble de *colonnes* et chaque colonne est d'un type de données particulier. Tandis que les colonnes ont un ordre fixé dans chaque ligne, il est important de se rappeler que SQL ne garantit, d'aucune façon, l'ordre des lignes à l'intérieur de la table (bien qu'elles puissent être explicitement triées pour l'affichage).

Les tables sont groupées dans des bases de données et un ensemble de bases gérées par une instance unique du serveur PostgreSQL constitue une *instance* de bases (*cluster* en anglais).

2.3. Créer une nouvelle table

Vous pouvez créer une nouvelle table en spécifiant le nom de la table, suivi du nom de toutes les colonnes et de leur type :

```
CREATE TABLE temps (
    ville           varchar(80),
```

```
    t_basse         int,            -- température basse
    t_haute         int,            -- température haute
    prcp            real,           -- précipitation
    date            date
);
```

Vous pouvez saisir cela dans `psql` avec les sauts de lignes. `psql` reconnaîtra que la commande n'est pas terminée jusqu'à arriver à un point-virgule.

Les espaces blancs (c'est-à-dire les espaces, les tabulations et les retours à la ligne) peuvent être librement utilisés dans les commandes SQL. Cela signifie que vous pouvez saisir la commande ci-dessus alignée différemment ou même sur une seule ligne. Deux tirets (« `--` ») introduisent des commentaires. Ce qui les suit est ignoré jusqu'à la fin de la ligne. SQL est insensible à la casse pour les mots-clés et les identifiants, excepté quand les identifiants sont entre doubles guillemets pour préserver leur casse (non fait ci-dessus).

`varchar(80)` spécifie un type de données pouvant contenir une chaîne de caractères arbitraires de 80 caractères au maximum. `int` est le type entier normal. `real` est un type pour les nombres décimaux en simple précision. `date` devrait s'expliquer de lui-même (oui, la colonne de type `date` est aussi nommée `date` ; cela peut être commode ou porter à confusion, à vous de choisir).

PostgreSQL prend en charge les types SQL standards `int`, `smallint`, `real`, `double precision`, `char(`*`N`*`)`, `varchar(`*`N`*`)`, `date`, `time`, `timestamp` et `interval`, ainsi que d'autres types d'utilité générale et un riche ensemble de types géométriques. PostgreSQL peut être personnalisé avec un nombre arbitraire de types de données définis par l'utilisateur. En conséquence, les noms des types ne sont pas des mots-clés dans la syntaxe sauf lorsqu'il est requis de supporter des cas particuliers dans la norme SQL.

Le second exemple stockera des villes et leur emplacement géographique associé :

```
CREATE TABLE villes (
    nom             varchar(80),
    emplacement     point
);
```

Le type `point` est un exemple d'un type de données spécifique à PostgreSQL.

Pour finir, vous devez savoir que si vous n'avez plus besoin d'une table ou que vous voulez la recréer différemment, vous pouvez la supprimer en utilisant la commande suivante :

```
DROP TABLE nom_table;
```

2.4. Remplir une table avec des lignes

L'instruction `INSERT` est utilisée pour remplir une table avec des lignes :

```
INSERT INTO temps VALUES ('San Francisco', 46, 50, 0.25,
 '1994-11-27');
```

Notez que tous les types utilisent des formats d'entrées plutôt évidents. Les constantes qui ne sont pas des valeurs numériques simples doivent être habituellement entourées par des guillemets simples (`'`) comme dans l'exemple. Le type `date` est en réalité tout à fait flexible dans ce qu'il accepte, mais, pour ce tutoriel, nous collerons au format non ambigu montré ici.

Le type `point` demande une paire de coordonnées en entrée, comme cela est montré ici :

```
INSERT INTO villes VALUES ('San Francisco', '(-194.0, 53.0)');
```

La syntaxe utilisée jusqu'à maintenant nécessite de se rappeler l'ordre des colonnes. Une syntaxe alternative vous autorise à lister les colonnes explicitement :

```
INSERT INTO temps (ville, t_basse, t_haute, prcp, date)
    VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

Vous pouvez lister les colonnes dans un ordre différent si vous le souhaitez ou même omettre certaines colonnes ; par exemple, si la précipitation est inconnue :

```
INSERT INTO temps (date, ville, t_haute, t_basse)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

De nombreux développeurs considèrent que le listage explicite des colonnes est un meilleur style que de compter sur l'ordre implicite.

Merci d'exécuter toutes les commandes vues ci-dessus de façon à avoir des données sur lesquelles travailler dans les prochaines sections.

Vous auriez pu aussi utiliser `COPY` pour charger de grandes quantités de données depuis des fichiers texte. C'est habituellement plus rapide, car la commande `COPY` est optimisée pour cet emploi, mais elle est moins flexible que `INSERT`. Par exemple :

```
COPY temps FROM '/home/utilisateur/temps.txt';
```

où le nom du fichier source doit être disponible sur la machine qui exécute le processus serveur, car le processus serveur lit le fichier directement. Les données insérées ci-dessus dans la table temps peuvent aussi être insérées à partir d'un fichier contenant (les valeurs sont séparées d'une tabulation) :

```
San Francisco    46    50    0.25    1994-11-27
San Francisco    43    57    0.0    1994-11-29
Hayward    37    54    \N    1994-11-29
```

Vous avez plus d'informations sur la commande `COPY` dans COPY.

2.5. Interroger une table

Pour retrouver les données d'une table, elle est *interrogée*. Une instruction SQL `SELECT` est utilisée pour faire cela. L'instruction est divisée en liste de sélection (la partie qui liste les colonnes à retourner), une liste de tables (la partie qui liste les tables à partir desquelles les données seront retrouvées) et une qualification optionnelle (la partie qui spécifie les restrictions). Par exemple, pour retrouver toutes les lignes de la table `temps`, saisissez :

```
SELECT * FROM temps;
```

Ici, `*` est un raccourci pour « toutes les colonnes ». [1] Donc, le même résultat pourrait être obtenu avec :

```
SELECT ville, t_basse, t_haute, prcp, date FROM temps;
```

Le résultat devrait être ceci :

```
     ville     | t_basse | t_haute | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |   50    | 0.25 | 1994-11-27
 San Francisco |      43 |   57    |    0 | 1994-11-29
 Hayward       |      37 |   54    |      | 1994-11-29
(3 rows)
```

Vous pouvez écrire des expressions, pas seulement des références à de simples colonnes, dans la liste de sélection. Par exemple, vous pouvez faire :

```
SELECT ville, (t_haute+t_basse)/2 AS temp_moy, date FROM temps;
```

[1] Alors que `SELECT *` est utile pour des requêtes rapides, c'est généralement considéré comme un mauvais style dans un code en production, car l'ajout d'une colonne dans la table changerait les résultats.

Cela devrait donner :

```
     ville     | temp_moy |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

Notez comment la clause AS est utilisée pour renommer la sortie d'une colonne (cette clause AS est optionnelle).

Une requête peut être « qualifiée » en ajoutant une clause WHERE qui spécifie les lignes souhaitées. La clause WHERE contient une expression booléenne et seules les lignes pour lesquelles l'expression booléenne est vraie sont renvoyées. Les opérateurs booléens habituels (AND, OR et NOT) sont autorisés dans la qualification. Par exemple, ce qui suit recherche le temps à San Francisco les jours pluvieux :

```
SELECT * FROM temps
    WHERE ville = 'San Francisco' AND prcp > 0.0;
```

Résultat :

```
     ville     | t_basse | t_haute | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

Vous pouvez demander à ce que les résultats d'une requête soient renvoyés dans un ordre trié :

```
SELECT * FROM temps
ORDER BY ville;
```

```
 ville        | t_basse | t_haute | prcp |    date
--------------+---------+---------+------+------------
Hayward       |      37 |      54 |      | 1994-11-29
San Francisco |      43 |      57 |    0 | 1994-11-29
San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

Dans cet exemple, l'ordre de tri n'est pas spécifié complètement, donc vous pouvez obtenir les lignes San Francisco dans n'importe quel ordre. Mais, vous auriez toujours obtenu les résultats affichés ci-dessus si vous aviez fait :

```
SELECT * FROM temps
ORDER BY ville, t_basse;
```

Vous pouvez demander que les lignes dupliquées soient supprimées du résultat d'une requête :

```
SELECT DISTINCT ville
    FROM temps;
```

```
     ville
---------------
 Hayward
 San Francisco
(2 rows)
```

De nouveau, l'ordre des lignes résultat pourrait varier. Vous pouvez vous assurer des résultats cohérents en utilisant DISTINCT et ORDER BY ensemble : [2]

[2] Dans certains systèmes de bases de données, ceci incluant les anciennes versions de PostgreSQL, l'implémentation de DISTINCT ordonne automatiquement les lignes. Du coup, ORDER BY n'est pas nécessaire. Mais, ceci n'est pas requis par le standard SQL et PostgreSQL ne vous garantit pas actuellement que DISTINCT ordonne les lignes.

```
SELECT DISTINCT ville
FROM temps
ORDER BY ville;
```

2.6. Jointures entre les tables

Jusqu'ici, nos requêtes avaient seulement consulté une table à la fois. Les requêtes peuvent accéder à plusieurs tables en même temps ou accéder à la même table de façon à ce que plusieurs lignes de la table soient traitées en même temps. Les requêtes qui accèdent à plusieurs tables (ou plusieurs fois la même table) sont appelées des requêtes de *jointure*. Elles combinent les lignes d'une table avec les lignes d'une deuxième table, avec une expression indiquant comment réaliser la jointure entre ces lignes. Par exemple, pour renvoyer toutes les lignes de la table temps avec l'emplacement de la ville associée, la base de données doit comparer la colonne `ville` de chaque ligne de la table `temps` avec la colonne `nom` de toutes les lignes de la table `villes` et que vous choisissiez les paires de lignes où ces valeurs correspondent.[3] Ceci sera accompli avec la requête suivante :

```
SELECT * FROM temps JOIN villes ON ville = nom;

     ville     | t_basse | t_haute | prcp |    date    |     nom
    | emplacement
---------------+---------+---------+------+------------
+---------------+-------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San
 Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San
 Francisco | (-194,53)
(2 rows)
```

Deux remarques à propos du résultat :

- Il n'y a pas de lignes pour la ville de Hayward dans le résultat. C'est parce qu'il n'y a aucune entrée correspondante dans la table `villes` pour Hayward, donc la jointure ignore les lignes n'ayant pas de correspondance avec la table `temps`. Nous verrons rapidement comment cela peut être résolu.

- Il y a deux colonnes contenant le nom des villes. C'est correct, car les listes des colonnes des tables `temps` et `villes` sont concaténées. En pratique, ceci est indésirable, vous voudrez probablement lister les colonnes explicitement plutôt que d'utiliser * :

  ```
  SELECT ville, t_basse, t_haute, prcp, date, emplacement
      FROM temps JOIN villes ON ville = nom;
  ```

Puisque toutes les colonnes ont un nom différent, l'analyseur a automatiquement trouvé à quelle table elles appartiennent. Si des noms de colonnes sont communs entre les deux tables, vous aurez besoin de *qualifier* les noms des colonnes pour préciser celles dont vous parlez. Par exemple :

```
SELECT temps.ville, temps.t_basse, temps.t_haute,
       temps.prcp, temps.date, villes.emplacement
    FROM temps JOIN villes ON villes.nom = temps.ville;
```

La qualification des noms de colonnes dans une requête de jointure est fréquemment considérée comme une bonne pratique. Cela évite l'échec de la requête si un nom de colonne dupliqué est ajouté plus tard dans une des tables.

Les requêtes de jointure vues jusqu'ici peuvent aussi être écrites sous cette forme :

```
SELECT *
```

[3] Ceci est uniquement un modèle conceptuel. La jointure est habituellement exécutée d'une manière plus efficace que la comparaison de chaque paire de lignes, mais c'est invisible pour l'utilisateur.

```
    FROM temps, villes WHERE ville = nom;
```

Cette syntaxe date d'avant la syntaxe `JOIN`/`ON`, qui a été introduite dans SQL-92. Les tables sont simplement listées dans la clause `FROM`, et l'expression de comparaison est ajoutée dans la clause `WHERE`. Les résultats de l'ancienne syntaxe implicite et de la nouvelle syntaxe explicite `JOIN`/`ON` sont identiques. Cependant, pour celui qui lit la requête, la syntaxe explicite est plus facile à comprendre : la condition de jointure est introduite par son propre mot clé alors qu'auparavant, la condition était mélangée dans la clause `WHERE` avec les autres conditions.

Maintenant, nous allons essayer de comprendre comment nous pouvons avoir les entrées de Hayward. Nous voulons que la requête parcoure la table `temps` et que, pour chaque ligne, elle trouve la (ou les) ligne(s) de `villes` correspondante(s). Si aucune ligne correspondante n'est trouvée, nous voulons que les valeurs des colonnes de la table `villes` soient remplacées par des « valeurs vides ». Ce genre de requêtes est appelé *jointure externe* (outer join). (Les jointures que nous avons vues jusqu'ici sont des *jointures internes* -- inner joins). La commande ressemble à cela :

```
SELECT *
    FROM temps LEFT OUTER JOIN villes ON temps.ville = villes.nom;
```

```
     ville     | t_basse | t_haute | prcp |    date    |     nom
    | emplacement
---------------+---------+---------+------+------------
+---------------+-------------
 Hayward       |      37 |      54 |      | 1994-11-29 |
    |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San
 Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San
 Francisco | (-194,53)
(3 rows)
```

Cette requête est appelée une *jointure externe à gauche* (left outer join) parce que la table mentionnée à la gauche de l'opérateur de jointure aura au moins une fois ses lignes dans le résultat, tandis que la table sur la droite aura seulement les lignes qui correspondent à des lignes de la table de gauche. Lors de l'affichage d'une ligne de la table de gauche pour laquelle il n'y a pas de correspondance dans la table de droite, des valeurs vides (appelées NULL) sont utilisées pour les colonnes de la table de droite.

Exercice : Il existe aussi des jointures externes à droite et des jointures externes complètes. Essayez de trouver ce qu'elles font.

Nous pouvons également joindre une table avec elle-même. Ceci est appelé une *jointure réflexive*. Comme exemple, supposons que nous voulions trouver toutes les entrées de temps qui sont dans un intervalle de températures d'autres entrées de temps. Nous avons donc besoin de comparer les colonnes `t_basse` et `t_haute` de chaque ligne de `temps` aux colonnes `t_basse` et `t_haute` de toutes les autres lignes de `temps`. Nous pouvons faire cela avec la requête suivante :

```
SELECT T1.ville, T1.t_basse AS bas, T1.t_haute AS haut,
    T2.ville, T2.t_basse AS bas, T2.t_haute AS haus
    FROM temps T1 JOIN temps T2
    ON T1.t_basse < T2.t_basse AND T1.t_haute > T2.t_haute;
```

```
     ville      | bas | haut |     ville     | bas | haut
----------------+-----+------+---------------+-----+------
  San Francisco |  43 |   57 | San Francisco |  46 |   50
  Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

Dans cet exemple, nous avons renommé la table temps en `T1` et en `T2` pour être capables de distinguer respectivement le côté gauche et le côté droit de la jointure. Vous pouvez aussi utiliser ce genre d'alias dans d'autres requêtes pour économiser de la frappe, c'est-à-dire :

```
SELECT *
    FROM temps t JOIN villes v ON t.ville = v.nom;
```

Vous rencontrerez ce genre d'abréviation assez fréquemment.

2.7. Fonctions d'agrégat

Comme la plupart des autres produits de bases de données relationnelles, PostgreSQL supporte les *fonctions d'agrégat*. Une fonction d'agrégat calcule un seul résultat à partir de plusieurs lignes en entrée. Par exemple, il y a des agrégats pour calculer le nombre (`count`), la somme (`sum`), la moyenne (`avg`), le maximum (`max`) et le minimum (`min`) d'un ensemble de lignes.

Comme exemple, nous pouvons trouver la température la plus haute parmi les températures basses avec :

```
SELECT max(t_basse) FROM temps;

 max
-----
  46
(1 row)
```

Si nous voulons connaître dans quelle ville (ou villes) ces lectures se sont produites, nous pouvons essayer :

```
SELECT ville FROM temps WHERE t_basse = max(t_basse);
FAUX
```

mais cela ne marchera pas puisque l'agrégat `max` ne peut pas être utilisé dans une clause `WHERE` (cette restriction existe parce que la clause `WHERE` détermine les lignes qui seront traitées par l'agrégat ; donc les lignes doivent être évaluées avant que les fonctions d'agrégat ne calculent leur résultat). Cependant, comme cela est souvent le cas, la requête peut être répétée pour arriver au résultat attendu, ici en utilisant une *sous-requête* :

```
SELECT ville FROM temps
    WHERE t_basse = (SELECT max(t_basse) FROM temps);

     ville
---------------
 San Francisco
(1 row)
```

Ceci est correct, car la sous-requête est un calcul indépendant qui traite son propre agrégat séparément à partir de ce qui se passe dans la requête externe.

Les agrégats sont également très utiles s'ils sont combinés avec les clauses `GROUP BY`. Par exemple, nous pouvons obtenir le nombre de prises de température et la température la plus haute parmi les températures basses observées dans chaque ville avec :

```
SELECT ville, count(*), max(t_basse)
    FROM temps
    GROUP BY ville;

     ville     | count | max
---------------+-------|-----
 Hayward       |     1 |  37
```

```
 San Francisco |     2 |  46
(2 rows)
```

ce qui nous donne une ligne par ville dans le résultat. Chaque résultat d'agrégat est calculé avec les lignes de la table correspondant à la ville. Nous pouvons filtrer ces lignes groupées en utilisant `HAVING` :

```
SELECT ville, count(*), max(t_basse)
    FROM temps
    GROUP BY ville
    HAVING max(t_basse) < 40;

  ville  | count | max
---------+-------+-----
 Hayward |     1 |  37
(1 row)
```

ce qui nous donne le même résultat uniquement pour les villes qui ont toutes leurs valeurs de `t_basse` en dessous de 40. Pour finir, si nous nous préoccupons seulement des villes dont le nom commence par « S », nous pouvons faire :

```
SELECT ville, count(*), max(t_basse)
    FROM temps
    WHERE ville LIKE 'S%' -- 1
    GROUP BY ville;


     city      | count | max
---------------+-------+-----
 San Francisco |     2 |  46
(1 row)
```

1 L'opérateur `LIKE` fait la correspondance avec un motif ; cela est expliqué dans la Section 9.7.

Il est important de comprendre l'interaction entre les agrégats et les clauses SQL `WHERE` et `HAVING`. La différence fondamentale entre `WHERE` et `HAVING` est que `WHERE` sélectionne les lignes en entrée avant que les groupes et les agrégats ne soient traités (donc, cette clause contrôle les lignes qui se retrouvent dans le calcul de l'agrégat), tandis que `HAVING` sélectionne les lignes groupées après que les groupes et les agrégats ont été traités. Donc, la clause `WHERE` ne doit pas contenir de fonctions d'agrégat ; cela n'a aucun sens d'essayer d'utiliser un agrégat pour déterminer les lignes en entrée des agrégats. D'un autre côté, la clause `HAVING` contient toujours des fonctions d'agrégat (pour être précis, vous êtes autorisés à écrire une clause `HAVING` qui n'utilise pas d'agrégat, mais c'est rarement utilisé. La même condition pourra être utilisée plus efficacement par un `WHERE`).

Dans l'exemple précédent, nous pouvons appliquer la restriction sur le nom de la ville dans la clause `WHERE` puisque cela ne nécessite aucun agrégat. C'est plus efficace que d'ajouter la restriction dans `HAVING` parce que nous évitons le groupement et les calculs d'agrégat pour toutes les lignes qui ont échoué lors du contrôle fait par `WHERE`.

Une autre façon de sélectionner les lignes qui vont dans le calcul d'un agrégat est d'utiliser la clause `FILTER`, qui est une option par agrégat :

```
SELECT ville, count(*) FILTER (WHERE t_basse < 45), max(t_basse)
    FROM temps
    GROUP BY ville;


     city      | count | max
---------------+-------+-----
```

```
 Hayward       |     1 |  37
 San Francisco |     1 |  46
(2 rows)
```

`FILTER` ressemble beaucoup à `WHERE`, sauf qu'elle supprime les lignes uniquement sur l'entrée de la fonction d'agrégat à laquelle elle est attachée. Dans cet exemple, l'agrégat `count` compte seulement les lignes pour lesquelles la colonne `t_basse` a une valeur inférieure à 45 alors que l'agrégat `max` est toujours appliqué à toutes les lignes, donc il trouve toujours la valeur 46.

2.8. Mises à jour

Vous pouvez mettre à jour une ligne existante en utilisant la commande `UPDATE`. Supposez que vous découvrez que les températures sont toutes excédentes de deux degrés après le 28 novembre. Vous pouvez corriger les données de la façon suivante :

```
UPDATE temps
    SET t_haute = t_haute - 2,  t_basse = t_basse - 2
    WHERE date > '1994-11-28';
```

Regardez le nouvel état des données :

```
SELECT * FROM temps;

     ville      | t_basse | t_haute | prcp |    date
----------------+---------+---------+------+------------
 San Francisco  |      46 |      50 | 0.25 | 1994-11-27
 San Francisco  |      41 |      55 |    0 | 1994-11-29
 Hayward        |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. Suppressions

Les lignes peuvent être supprimées de la table avec la commande `DELETE`. Supposez que vous ne soyez plus intéressé par le temps de Hayward. Vous pouvez faire ce qui suit pour supprimer ses lignes de la table :

```
DELETE FROM temps WHERE ville = 'Hayward';
```

Toutes les entrées de temps pour Hayward sont supprimées.

```
SELECT * FROM temps;

     ville     | t_basse | t_haute | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

Faire très attention aux instructions de la forme

```
DELETE FROM nom_table;
```

Sans une qualification, `DELETE` supprimera *toutes* les lignes de la table donnée, la laissant vide. Le système le fera sans demander de confirmation !

Chapitre 3. Fonctionnalités avancées

3.1. Introduction

Le chapitre précédent couvre les bases de l'utilisation de SQL pour le stockage et l'accès aux données avec PostgreSQL. Il est temps d'aborder quelques fonctionnalités avancées du SQL qui simplifient la gestion et empêchent la perte ou la corruption des données. Quelques extensions de PostgreSQL sont également abordées.

Ce chapitre fait occasionnellement référence aux exemples disponibles dans le Chapitre 2 pour les modifier ou les améliorer. Il est donc préférable d'avoir lu ce chapitre. Quelques exemples de ce chapitre sont également disponibles dans `advanced.sql` situé dans le répertoire du tutoriel. De plus, ce fichier contient quelques données à charger pour utiliser l'exemple. Cela n'est pas repris ici (on peut se référer à la Section 2.1 pour savoir comment utiliser ce fichier).

3.2. Vues

Se référer aux requêtes de la Section 2.6. Si la liste des enregistrements du temps et des villes est d'un intérêt particulier pour l'application considérée, mais qu'il devient contraignant de saisir la requête à chaque utilisation, il est possible de créer une *vue* avec la requête. De ce fait, la requête est nommée et il peut y être fait référence de la même façon qu'il est fait référence à une table :

```
CREATE VIEW ma_vue AS
    SELECT nom, t_basse, t_haute, prcp, date, emplacement
        FROM temps, villes
        WHERE ville = nom;

SELECT * FROM ma_vue;
```

L'utilisation des vues est un aspect clé d'une bonne conception des bases de données SQL. Les vues permettent d'encapsuler les détails de la structure des tables. Celle-ci peut alors changer avec l'évolution de l'application, tandis que l'interface reste constante.

Les vues peuvent être utilisées dans quasiment toutes les situations où une vraie table est utilisable. De plus, il n'est pas inhabituel de construire des vues reposant sur d'autres vues.

3.3. Clés étrangères

Soit les tables `temps` et `villes` définies dans le Chapitre 2. Il s'agit maintenant de s'assurer que personne n'insère de ligne dans la table `temps` qui ne corresponde à une entrée dans la table `villes`. On appelle cela maintenir l'*intégrité référentielle* des données. Dans les systèmes de bases de données simplistes, lorsque c'est possible, cela est parfois obtenu par la vérification préalable de l'existence d'un enregistrement correspondant dans la table `villes`, puis par l'insertion, ou l'interdiction, du nouvel enregistrement dans `temps`. Puisque cette approche, peu pratique, présente un certain nombre d'inconvénients, PostgreSQL peut se charger du maintien de l'*intégrité référentielle*.

La nouvelle déclaration des tables ressemble alors à ceci :

```
CREATE TABLE villes (
 nom       varchar(80) primary key,
 emplacement point
);

CREATE TABLE temps (
 ville      varchar(80) references villes(nom),
```

```
 t_haute    int,
 t_basse    int,
 prcp       real,
 date       date
);
```

Lors d'une tentative d'insertion d'enregistrement non valide :

```
INSERT INTO temps VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');

ERROR:  insert or update on table "temps" violates foreign key
 constraint "temps_ville_fkey"
DETAIL: Key (ville)=(a) is not present in table "villes".
```

Le comportement des clés étrangères peut être adapté très finement à une application particulière. Ce tutoriel ne va pas plus loin que cet exemple simple. De plus amples informations sont accessibles dans le Chapitre 5. Une utilisation efficace des clés étrangères améliore la qualité des applications accédant aux bases de données. Il est donc fortement conseillé d'apprendre à les utiliser.

3.4. Transactions

Les *transactions* sont un concept fondamental de tous les systèmes de bases de données. Une transaction assemble plusieurs étapes en une seule opération tout ou rien. Les états intermédiaires entre les étapes ne sont pas visibles par les transactions concurrentes. De plus, si un échec survient qui empêche le succès de la transaction, alors aucune des étapes n'affecte la base de données.

Si l'on considère, par exemple, la base de données d'une banque qui contient le solde de différents comptes clients et le solde total des dépôts par branches et que l'on veut enregistrer un virement de 100 euros du compte d'Alice vers celui de Bob, les commandes SQL peuvent ressembler à cela (après simplification) :

```
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE nom = (SELECT nom_branche FROM comptes WHERE nom =
 'Alice');
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE nom = (SELECT nom_branche FROM comptes WHERE nom =
 'Bob');
```

Ce ne sont pas les détails des commandes qui importent ici ; le point important est la nécessité de plusieurs mises à jour séparées pour accomplir cette opération assez simple. Les employés de la banque veulent être assurés que, soit toutes les commandes sont effectuées, soit aucune ne l'est. Il n'est pas envisageable que, suite à une erreur du système, Bob reçoive 100 euros qui n'ont pas été débités du compte d'Alice. De la même façon, Alice ne restera pas longtemps une cliente fidèle si elle est débitée du montant sans que celui-ci ne soit crédité sur le compte de Bob. Il est important de garantir que si quelque chose se passe mal, aucune des étapes déjà exécutées n'est prise en compte. Le regroupement des mises à jour au sein d'une *transaction* apporte cette garantie. Une transaction est dite *atomique* : du point de vue des autres transactions, elle passe complètement ou pas du tout.

Il est également nécessaire de garantir qu'une fois la transaction terminée et validée par la base de données, les transactions sont enregistrées définitivement et ne peuvent être perdues, même si une panne survient peu après. Ainsi, si un retrait d'argent est effectué par Bob, il ne faut absolument pas que le débit de son compte disparaisse suite à une panne survenant juste après son départ de la banque. Une base de données transactionnelle garantit que toutes les mises à jour faites lors d'une transaction sont stockées de manière persistante (c'est-à-dire sur disque) avant que la transaction ne soit déclarée validée.

Une autre propriété importante des bases de données transactionnelles est en relation étroite avec la notion de mises à jour atomiques : quand plusieurs transactions sont lancées en parallèle, aucune d'entre elles ne doit être capable de voir les modifications incomplètes effectuées par les autres. Ainsi, si une transaction calcule le total de toutes les branches, inclure le débit de la branche d'Alice sans le crédit de la branche de Bob, ou vice-versa, est une véritable erreur. Les transactions doivent donc être tout ou rien, non seulement pour leur effet persistant sur la base de données, mais aussi pour leur visibilité au moment de leur exécution. Les mises à jour faites jusque-là par une transaction ouverte sont invisibles aux autres transactions jusqu'à la fin de celle-ci. À ce moment, toutes les mises à jour deviennent simultanément visibles.

Sous PostgreSQL, une transaction est déclarée en entourant les commandes SQL de la transaction par les commandes `BEGIN` et `COMMIT`. La transaction bancaire ressemble alors à ceci :

```
BEGIN;
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
-- etc etc
COMMIT;
```

Si, au cours de la transaction, il est décidé de ne pas valider (peut-être la banque s'aperçoit-elle que la balance d'Alice passe en négatif), la commande `ROLLBACK` peut être utilisée à la place de `COMMIT`. Toutes les mises à jour réalisées jusque-là sont alors annulées.

En fait, PostgreSQL traite chaque instruction SQL comme si elle était exécutée dans une transaction. En l'absence de commande `BEGIN` explicite, chaque instruction individuelle se trouve implicitement entourée d'un `BEGIN` et (en cas de succès) d'un `COMMIT`. Un groupe d'instructions entourées par `BEGIN` et `COMMIT` est parfois appelé *bloc transactionnel*.

> **Note**
>
> Quelques bibliothèques clientes lancent les commandes `BEGIN` et `COMMIT` automatiquement. L'utilisateur bénéficie alors des effets des blocs transactionnels sans les demander. Vérifiez la documentation de l'interface que vous utilisez.

Il est possible d'augmenter la granularité du contrôle des instructions au sein d'une transaction en utilisant des *points de retournement* (*savepoint*). Ceux-ci permettent d'annuler des parties de la transaction tout en validant le reste. Après avoir défini un point de retournement à l'aide de `SAVEPOINT`, les instructions exécutées depuis ce point peuvent, au besoin, être annulées avec `ROLLBACK TO`. Toutes les modifications de la base de données effectuées par la transaction entre le moment où le point de retournement a été défini et celui où l'annulation est demandée sont annulées, mais les modifications antérieures à ce point sont conservées.

Le retour à un point de retournement ne l'annule pas. Il reste défini et peut donc être utilisé plusieurs fois. À l'inverse, lorsqu'il n'est plus nécessaire de revenir à un point de retournement particulier, il peut être relâché, ce qui permet de libérer des ressources système. Il faut savoir toutefois que relâcher un point de retournement ou y revenir relâche tous les points de retournement qui ont été définis après.

Tout ceci survient à l'intérieur du bloc de transaction, et n'est donc pas visible par les autres sessions en cours sur la base de données. Si le bloc est validé, et à ce moment-là seulement, toutes les actions validées deviennent immédiatement visibles par les autres sessions, tandis que les actions annulées ne le seront jamais.

Reconsidérant la base de données de la banque, on peut supposer vouloir débiter le compte d'Alice de $100.00, somme à créditer sur le compte de Bob, mais considérer plus tard que c'est le compte de Wally qu'il convient de créditer. À l'aide des points de retournement, cela peut se dérouler ainsi :

```
BEGIN;
```

```
UPDATE comptes SET balance = balance - 100.00
    WHERE nom = 'Alice';
SAVEPOINT mon_pointdesauvegarde;
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Bob';
-- oups ... oublions ça et créditons le compte de Wally
ROLLBACK TO mon_pointdesauvegarde;
UPDATE comptes SET balance = balance + 100.00
    WHERE nom = 'Wally';
COMMIT;
```

Cet exemple est bien sûr très simplifié, mais de nombreux contrôles sont réalisables au sein d'un bloc de transaction grâce à l'utilisation des points de retournement. Qui plus est, `ROLLBACK TO` est le seul moyen de regagner le contrôle d'un bloc de transaction placé dans un état d'annulation par le système du fait d'une erreur. C'est plus rapide que de tout annuler pour tout recommencer.

3.5. Fonctions de fenêtrage

Une *fonction de fenêtrage* effectue un calcul sur un jeu d'enregistrements liés d'une certaine façon à l'enregistrement courant. On peut les rapprocher des calculs réalisables par une fonction d'agrégat. Cependant, les fonctions de fenêtrage n'entraînent pas le regroupement des enregistrements traités en un seul, comme le ferait l'appel à une fonction d'agrégation standard. À la place, chaque enregistrement garde son identité propre. En coulisse, la fonction de fenêtrage est capable d'accéder à d'autres enregistrements que l'enregistrement courant du résultat de la requête.

Voici un exemple permettant de comparer le salaire d'un employé avec le salaire moyen de sa division :

```
SELECT nomdep, noemp, salaire, avg(salaire) OVER (PARTITION BY
 nomdep) FROM salaireemp;
```

```
  nomdep   | noemp | salaire |          avg
-----------+-------+---------+-----------------------
 develop   |    11 |   5200  | 5020.0000000000000000
 develop   |     7 |   4200  | 5020.0000000000000000
 develop   |     9 |   4500  | 5020.0000000000000000
 develop   |     8 |   6000  | 5020.0000000000000000
 develop   |    10 |   5200  | 5020.0000000000000000
 personnel |     5 |   3500  | 3700.0000000000000000
 personnel |     2 |   3900  | 3700.0000000000000000
 ventes    |     3 |   4800  | 4866.6666666666666667
 ventes    |     1 |   5000  | 4866.6666666666666667
 ventes    |     4 |   4800  | 4866.6666666666666667
(10 rows)
```

Les trois premières colonnes viennent directement de la table `salaireemp`, et il y a une ligne de sortie pour chaque ligne de la table. La quatrième colonne représente une moyenne calculée sur tous les enregistrements de la table qui ont la même valeur de `nomdep` que la ligne courante. (Il s'agit effectivement de la même fonction que la fonction d'agrégat classique `avg`, mais la clause `OVER` entraîne son exécution en tant que fonction de fenêtrage et son calcul sur la fenêtre.)

Un appel à une fonction de fenêtrage contient toujours une clause `OVER` qui suit immédiatement le nom et les arguments de la fonction. C'est ce qui permet de la distinguer syntaxiquement d'une fonction simple ou d'une fonction d'agrégat. La clause `OVER` détermine précisément comment les lignes de la requête sont éclatées pour être traitées par la fonction de fenêtrage. La clause `PARTITION`

BY contenue dans OVER divise les enregistrements en groupes, ou partitions, qui partagent les mêmes valeurs pour la (les) expression(s) contenue(s) dans la clause PARTITION BY. Pour chaque enregistrement, la fonction de fenêtrage est calculée sur les enregistrements qui se retrouvent dans la même partition que l'enregistrement courant.

Vous pouvez aussi contrôler l'ordre dans lequel les lignes sont traitées par les fonctions de fenêtrage en utilisant la clause ORDER BY à l'intérieur de la clause OVER (la partition traitée par le ORDER BY n'a de plus pas besoin de correspondre à l'ordre dans lequel les lignes seront affichées). Voici un exemple :

```
SELECT nomdep, noemp, salaire,
       rank() OVER (PARTITION BY nomdep ORDER BY salaire DESC)
FROM salaireemp;
```

```
  nomdep   | noemp | salaire| rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 ventes    |     1 |   5000 |    1
 ventes    |     4 |   4800 |    2
 ventes    |     3 |   4800 |    2
(10 rows)
```

On remarque que la fonction rank produit un rang numérique pour chaque valeur ORDER BY distincte dans la partition de la ligne courante, en utilisant l'ordre défini par la clause ORDER BY. rank n'a pas besoin de paramètre explicite, puisque son comportement est entièrement déterminé par la clause OVER.

Les lignes prises en compte par une fonction de fenêtrage sont celles de la table virtuelle produite par la clause FROM de la requête filtrée par ses clauses WHERE, GROUP BY et HAVING, s'il y en a. Par exemple, une ligne rejetée parce qu'elle ne satisfait pas à la condition WHERE n'est vue par aucune fonction de fenêtrage. Une requête peut contenir plusieurs de ces fonctions de fenêtrage qui découpent les données de façons différentes, par le biais de clauses OVER différentes, mais elles travaillent toutes sur le même jeu d'enregistrements, défini par cette table virtuelle.

ORDER BY peut être omis lorsque l'ordre des enregistrements est sans importance. Il est aussi possible d'omettre PARTITION BY, auquel cas il n'y a qu'une seule partition, contenant tous les enregistrements.

Il y a un autre concept important associé aux fonctions de fenêtrage : pour chaque enregistrement, il existe un jeu d'enregistrements dans sa partition appelé son *window frame* (cadre de fenêtre). Certaines fonctions de fenêtrage travaillent uniquement sur les enregistrements du *window frame*, plutôt que sur l'ensemble de la partition. Par défaut, si on a précisé une clause ORDER BY, la *window frame* contient tous les enregistrements du début de la partition jusqu'à l'enregistrement courant, ainsi que tous les enregistrements suivants qui sont égaux à l'enregistrement courant au sens de la clause ORDER BY. Quand ORDER BY est omis, la *window frame* par défaut contient tous les enregistrements de la partition. [1] Voici un exemple utilisant sum :

```
SELECT salaire, sum(salaire) OVER () FROM salaireemp;
```

[1] Il existe des options pour définir la *window frame* autrement, mais ce tutoriel ne les présente pas. Voir la Section 4.2.8 pour les détails.

```
 salaire|  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

Dans l'exemple ci-dessus, puisqu'il n'y a pas d'`ORDER BY` dans la clause `OVER`, la *window frame* est égale à la partition ; en d'autres termes, chaque somme est calculée sur toute la table, ce qui fait qu'on a le même résultat pour chaque ligne du résultat. Mais si on ajoute une clause `ORDER BY`, on a un résultat très différent :

```
SELECT salaire, sum(salaire) OVER (ORDER BY salaire) FROM
 salaireemp;
```

```
 salaire|  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

Ici, sum est calculé à partir du premier salaire (c'est-à-dire le plus bas) jusqu'au salaire courant, en incluant tous les doublons du salaire courant (remarquez les valeurs pour les salaires identiques).

Les fonctions window ne sont autorisées que dans la liste `SELECT` et la clause `ORDER BY` de la requête. Elles sont interdites ailleurs, comme dans les clauses `GROUP BY`,`HAVING` et `WHERE`. La raison en est qu'elles sont exécutées après le traitement de ces clauses. Par ailleurs, les fonctions de fenêtrage s'exécutent après les fonctions d'agrégat classiques. Cela signifie qu'il est permis d'inclure une fonction d'agrégat dans les arguments d'une fonction de fenêtrage, mais pas l'inverse.

S'il y a besoin de filtrer ou de grouper les enregistrements après le calcul des fonctions de fenêtrage, une sous-requête peut être utilisée. Par exemple :

```
SELECT nomdep, noemp, salaire, date_embauche
FROM
```

```
  (SELECT nomdep, noemp, salaire, date_embauche,
          rank() OVER (PARTITION BY nomdep ORDER BY salaire DESC,
 noemp) AS pos
     FROM salaireemp
  ) AS ss
WHERE pos < 3;
```

La requête ci-dessus n'affiche que les enregistrements de la requête interne ayant un rang inférieur à 3.

Quand une requête met en jeu plusieurs fonctions de fenêtrage, il est possible d'écrire chacune avec une clause `OVER` différente, mais cela entraîne des duplications de code et augmente les risques d'erreurs si on souhaite le même comportement pour plusieurs fonctions de fenêtrage. À la place, chaque comportement de fenêtrage peut être associé à un nom dans une clause `WINDOW` et ensuite être référencé dans `OVER`. Par exemple :

```
SELECT sum(salaire) OVER w, avg(salaire) OVER w
  FROM salaireemp
  WINDOW w AS (PARTITION BY nomdep ORDER BY salaire DESC);
```

Plus de détails sur les fonctions de fenêtrage sont disponibles dans la Section 4.2.8, la Section 9.22, la Section 7.2.5 et la page de référence SELECT.

3.6. Héritage

L'héritage est un concept issu des bases de données orientées objet. Il ouvre de nouvelles possibilités intéressantes en conception de bases de données.

Soit deux tables : une table `villes` et une table `capitales`. Les capitales étant également des villes, il est intéressant d'avoir la possibilité d'afficher implicitement les capitales lorsque les villes sont listées. Un utilisateur particulièrement brillant peut écrire ceci

```
CREATE TABLE capitales (
  nom        text,
  population real,
  elevation  int,    -- (en pieds)
  etat       char(2)
);

CREATE TABLE non_capitales (
  nom        text,
  population real,
  elevation   int     -- (en pieds)
);

CREATE VIEW villes AS
  SELECT nom, population, elevation FROM capitales
    UNION
  SELECT nom, population, elevation FROM non_capitales;
```

Cela fonctionne bien pour les requêtes, mais la mise à jour d'une même donnée sur plusieurs lignes devient vite un horrible casse-tête.

Une meilleure solution peut être :

```
CREATE TABLE villes (
  nom        text,
```

```
  population real,
  elevation  int     -- (en pieds)
);

CREATE TABLE capitales (
  etat      char(2) UNIQUE NOT NULL
) INHERITS (villes);
```

Dans ce cas, une ligne de `capitales` *hérite* de toutes les colonnes (`nom`, `population` et `elevation`) de son *parent*, `villes`. Le type de la colonne `nom` est `text`, un type natif de PostgreSQL pour les chaînes de caractères à longueur variable. La table `capitales` a une colonne supplémentaire, `etat`, qui affiche l'abréviation de l'état. Sous PostgreSQL, une table peut hériter de zéro à plusieurs autres tables.

La requête qui suit fournit un exemple d'extraction des noms de toutes les villes, en incluant les capitales des états, situées à une elevation de plus de 500 pieds :

```
SELECT nom, elevation
  FROM villes
  WHERE elevation > 500;
```

ce qui renvoie :

```
   nom     | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
(3 rows)
```

À l'inverse, la requête qui suit récupère toutes les villes qui ne sont pas des capitales et qui sont situées à une élévation d'au moins 500 pieds :

```
SELECT nom, elevation
    FROM ONLY villes
    WHERE elevation > 500;

   nom     | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
(2 rows)
```

Ici, `ONLY` avant `villes` indique que la requête ne doit être exécutée que sur la table `villes`, et non pas sur les tables en dessous de `villes` dans la hiérarchie des héritages. La plupart des commandes déjà évoquées -- `SELECT`, `UPDATE` et `DELETE` -- supportent cette notation (`ONLY`).

> **Note**
>
> Bien que l'héritage soit fréquemment utile, il n'a pas été intégré avec les contraintes d'unicité et les clés étrangères, ce qui limite son utilité. Voir la Section 5.10 pour plus de détails.

3.7. Conclusion

PostgreSQL dispose d'autres fonctionnalités non décrites dans ce tutoriel d'introduction orienté vers les nouveaux utilisateurs de SQL. Ces fonctionnalités sont discutées plus en détail dans le reste de ce livre.

Si une introduction plus approfondie est nécessaire, le lecteur peut visiter le site web[2] de PostgreSQL qui fournit des liens vers d'autres ressources.

[2] https://www.postgresql.org

Partie II. Langage SQL

Cette partie présente l'utilisation du langage SQL dans PostgreSQL. Nous commençons par la syntaxe générale du SQL, nous expliquerons comment créer des structures de stockage des données, remplir la base, et comment l'interroger. La partie centrale liste les types de données et les fonctions disponibles, et leur utilisation dans les requêtes SQL. Le reste traite de divers points importants dans le paramétrage en vue de performances optimales.

Cette partie est organisée de telle manière qu'un utilisateur novice puisse la suivre du début à la fin, et gagner une compréhension complète des sujets sans avoir à effectuer de fréquents sauts entre les chapitres. Les chapitres sont indépendants pour qu'un utilisateur expérimenté puisse ne lire que les chapitres qui l'intéressent. L'information est présentée dans un style narratif par unité thématique. Les lecteurs cherchant la description complète d'une commande particulière doivent se référer à la Partie VI.

Les lecteurs doivent savoir se connecter à une base PostgreSQL et y exécuter des commandes SQL. Ceux peu familiers avec ces prérequis sont encouragés à lire préalablement la Partie I. Les commandes SQL sont généralement saisies à partir du terminal interactif de PostgreSQL, psql, mais d'autres programmes aux fonctionnalités similaires peuvent aussi être utilisés.

Table des matières

4. Syntaxe SQL 33
4.1. Structure lexicale 33
4.1.1. identificateurs et mots-clés 33
4.1.2. Constantes 35
4.1.3. Opérateurs 40
4.1.4. Caractères spéciaux 40
4.1.5. Commentaires 41
4.1.6. Précédence d'opérateurs 41
4.2. Expressions de valeurs 42
4.2.1. Références de colonnes 43
4.2.2. Paramètres de position 43
4.2.3. Indices 44
4.2.4. Sélection de champs 44
4.2.5. Appels d'opérateurs 45
4.2.6. Appels de fonctions 45
4.2.7. Expressions d'agrégat 45
4.2.8. Appels de fonction de fenêtrage 48
4.2.9. Conversions de type 50
4.2.10. Expressions de collationnement 51
4.2.11. Sous-requêtes scalaires 52
4.2.12. Constructeurs de tableaux 52
4.2.13. Constructeurs de lignes 54
4.2.14. Règles d'évaluation des expressions 55
4.3. Fonctions appelantes 57
4.3.1. En utilisant la notation par position 57
4.3.2. En utilisant la notation par nom 58
4.3.3. En utilisant la notation mixée 58
5. Définition des données 60
5.1. Notions fondamentales sur les tables 60
5.2. Valeurs par défaut 61
5.3. Colonnes générées 62
5.4. Contraintes 63
5.4.1. Contraintes de vérification 63
5.4.2. Contraintes de non-nullité (NOT NULL) 66
5.4.3. Contraintes d'unicité 66
5.4.4. Clés primaires 68
5.4.5. Clés étrangères 68
5.4.6. Contraintes d'exclusion 72
5.5. Colonnes système 72
5.6. Modification des tables 73
5.6.1. Ajouter une colonne 74
5.6.2. Supprimer une colonne 74
5.6.3. Ajouter une contrainte 75
5.6.4. Supprimer une contrainte 76
5.6.5. Modifier la valeur par défaut d'une colonne 76
5.6.6. Modifier le type de données d'une colonne 76
5.6.7. Renommer une colonne 76
5.6.8. Renommer une table 77
5.7. Droits 77
5.8. Politiques de sécurité niveau ligne 81
5.9. Schémas 88
5.9.1. Créer un schéma 88
5.9.2. Le schéma public 89
5.9.3. Chemin de parcours des schémas 89
5.9.4. Schémas et droits 91

5.9.5. Le schéma du catalogue système 91
5.9.6. Utilisation 91
5.9.7. Portabilité 92
5.10. L'héritage 93
5.10.1. Restrictions 96
5.11. Partitionnement de tables 96
5.11.1. Aperçu 96
5.11.2. Partitionnement déclaratif 97
5.11.3. Partitionnement utilisant l'héritage 102
5.11.4. Élagage de partition 107
5.11.5. Partitionnement et contrainte d'exclusion 108
5.11.6. Bonnes pratiques avec le partitionnement déclaratif 109
5.12. Données distantes 110
5.13. Autres objets de la base de données 111
5.14. Gestion des dépendances 111
6. Manipulation de données 113
6.1. Insérer des données 113
6.2. Actualiser les données 114
6.3. Supprimer des données 115
6.4. Renvoyer des données provenant de lignes modifiées 115
7. Requêtes 117
7.1. Aperçu 117
7.2. Expressions de table 117
7.2.1. Clause `FROM` 118
7.2.2. Clause `WHERE` 127
7.2.3. Clauses `GROUP BY` et `HAVING` 128
7.2.4. `GROUPING SETS`, `CUBE` et `ROLLUP` 130
7.2.5. Traitement de fonctions Window 133
7.3. Listes de sélection 134
7.3.1. Éléments de la liste de sélection 134
7.3.2. Labels de colonnes 134
7.3.3. `DISTINCT` 135
7.4. Combiner des requêtes (`UNION`, `INTERSECT`, `EXCEPT`) 135
7.5. Tri des lignes (`ORDER BY`) 137
7.6. `LIMIT` et `OFFSET` 138
7.7. Listes `VALUES` 138
7.8. Requêtes `WITH` (*Common Table Expressions*) 139
7.8.1. `SELECT` dans `WITH` 139
7.8.2. Requêtes récursives 140
7.8.3. Matérialisation des CTE 145
7.8.4. Ordres de modification de données avec `WITH` 146
8. Types de données 149
8.1. Types numériques 150
8.1.1. Types entiers 151
8.1.2. Nombres à précision arbitraire 151
8.1.3. Types à virgule flottante 154
8.1.4. Types seriés 156
8.2. Types monétaires 157
8.3. Types caractère 158
8.4. Types de données binaires 160
8.4.1. Le format hexadécimal `bytea` 160
8.4.2. Le format d'échappement `bytea` 161
8.5. Types date/heure 162
8.5.1. Saisie des dates et heures 163
8.5.2. Affichage des dates et heures 167
8.5.3. Fuseaux horaires 168
8.5.4. Saisie d'intervalle 170
8.5.5. Affichage d'intervalles 172

8.6. Type booléen 172
8.7. Types énumération 173
8.7.1. Déclaration de types énumérés 174
8.7.2. Tri 174
8.7.3. Sûreté du type 175
8.7.4. Détails d'implémentation 175
8.8. Types géométriques 176
8.8.1. Points 176
8.8.2. Lines 176
8.8.3. Segments de droite 177
8.8.4. Boîtes 177
8.8.5. Chemins 177
8.8.6. Polygones 178
8.8.7. Cercles 178
8.9. Types adresses réseau 178
8.9.1. `inet` 179
8.9.2. `cidr` 179
8.9.3. `inet` vs `cidr` 180
8.9.4. `macaddr` 180
8.9.5. `macaddr8` 180
8.10. Type chaîne de bits 181
8.11. Types de recherche plein texte 182
8.11.1. `tsvector` 182
8.11.2. `tsquery` 183
8.12. Type UUID 185
8.13. Type XML 185
8.13.1. Créer des valeurs XML 186
8.13.2. Gestion de l'encodage 187
8.13.3. Accéder aux valeurs XML 187
8.14. Types JSON 188
8.14.1. Syntaxe d'entrée et de sortie JSON 189
8.14.2. Concevoir des documents JSON efficacement 190
8.14.3. Existence et inclusion `jsonb` 191
8.14.4. Indexation `jsonb` 193
8.14.5. Utiliser l'indiçage avec `jsonb` 196
8.14.6. Transformations 197
8.14.7. Type jsonpath 198
8.15. Tableaux 199
8.15.1. Déclaration des types tableaux 199
8.15.2. Saisie de valeurs de type tableau 200
8.15.3. Accès aux tableaux 201
8.15.4. Modification de tableaux 204
8.15.5. Recherche dans les tableaux 206
8.15.6. Syntaxe d'entrée et de sortie des tableaux 208
8.16. Types composites 209
8.16.1. Déclaration de types composites 209
8.16.2. Construire des valeurs composites 210
8.16.3. Accéder aux types composites 211
8.16.4. Modifier les types composites 211
8.16.5. Utiliser des types composites dans les requêtes 212
8.16.6. Syntaxe en entrée et sortie d'un type composite 214
8.17. Types intervalle de valeurs 215
8.17.1. Types internes d'intervalle de valeurs range et multirange 216
8.17.2. Exemples 216
8.17.3. Bornes inclusives et exclusives 217
8.17.4. Intervalles de valeurs infinis (sans borne) 217
8.17.5. Saisie/affichage d'intervalle de valeurs 217
8.17.6. Construire des intervalles de valeurs ranges et multiranges 218

8.17.7. Types intervalle de valeurs discrètes 219
8.17.8. Définir de nouveaux types intervalle de valeurs 220
8.17.9. Indexation 221
8.17.10. Contraintes sur les intervalles de valeurs 221
8.18. Types domaine 222
8.19. Types identifiant d'objet 223
8.20. Type pg_lsn 225
8.21. Pseudo-Types 226
9. Fonctions et opérateurs 228
9.1. Opérateurs logiques 228
9.2. Fonctions et opérateurs de comparaison 229
9.3. Fonctions et opérateurs mathématiques 233
9.4. Fonctions et opérateurs sur des chaînes de caractères 240
9.4.1. format 249
9.5. Fonctions et opérateurs pour les chaînes binaires 251
9.6. Fonctions et opérateurs pour les chaînes de bit 256
9.7. Correspondance de motif 257
9.7.1. LIKE 258
9.7.2. Expressions rationnelles pour SIMILAR TO 259
9.7.3. Expressions rationnelles POSIX 261
9.8. Fonctions de formatage de type de données 279
9.9. Fonctions et opérateurs pour date/heure 287
9.9.1. EXTRACT, date_part 294
9.9.2. date_trunc 299
9.9.3. date_bin 300
9.9.4. AT TIME ZONE 301
9.9.5. Date/Heure actuelle 302
9.9.6. Retarder l'exécution 303
9.10. Fonctions de support d'enum 304
9.11. Fonctions et opérateurs de géométrie 305
9.12. Fonctions et opérateurs pour les adresses réseau 312
9.13. Fonctions et opérateurs de recherche plein texte 315
9.14. Fonctions UUID 322
9.15. Fonctions XML 322
9.15.1. Produire du contenu XML 323
9.15.2. Prédicats XML 327
9.15.3. Traiter du XML 329
9.15.4. Correspondances tables et XML 334
9.16. Fonctions et opérateurs JSON 338
9.16.1. Traiter et créer des données JSON 338
9.16.2. Le langage de chemin SQL/JSON 349
9.17. Fonctions de manipulation de séquence 357
9.18. Expressions conditionnelles 358
9.18.1. CASE 359
9.18.2. COALESCE 360
9.18.3. NULLIF 361
9.18.4. GREATEST et LEAST 361
9.19. Fonctions et opérateurs tableau 361
9.20. Fonctions et opérateurs sur les intervalles simple/multiple 365
9.21. Fonctions d'agrégat 371
9.22. Fonctions de fenêtrage 378
9.23. Expressions de sous-requêtes 380
9.23.1. EXISTS 380
9.23.2. IN 380
9.23.3. NOT IN 381
9.23.4. ANY/SOME 381
9.23.5. ALL 382
9.23.6. Comparaison de ligne simple 383

9.24. Comparaisons de ligne et de tableau 383
9.24.1. IN 383
9.24.2. NOT IN 383
9.24.3. ANY/SOME (array) 384
9.24.4. ALL (array) 384
9.24.5. Comparaison de constructeur de lignes 384
9.24.6. Comparaison de type composite 385
9.25. Fonctions renvoyant des ensembles 386
9.26. Fonctions et opérateurs d'information sur le système 389
9.27. Fonctions d'administration système 408
9.27.1. Fonctions de configuration 408
9.27.2. Fonctions d'envoi de signaux 408
9.27.3. Fonctions de contrôle de la sauvegarde 410
9.27.4. Fonctions de contrôle de la restauration 413
9.27.5. Fonctions de synchronisation d'image 414
9.27.6. Fonctions de gestion de la réplication 415
9.27.7. Fonctions de gestion des objets de la base 418
9.27.8. Fonctions de maintenance des index 421
9.27.9. Fonctions génériques d'accès aux fichiers 422
9.27.10. Fonctions sur les verrous consultatifs 424
9.28. Fonctions trigger 426
9.29. Fonctions de trigger d'événement 427
9.29.1. Capturer les modifications à la fin d'une commande 427
9.29.2. Traitement des objets supprimés par une commande DDL 428
9.29.3. Gérer un événement de réécriture de table 429
9.30. Fonctions d'informations statistiques 430
9.30.1. Inspecter les listes MCV 430
10. Conversion de types 431
10.1. Aperçu 431
10.2. Opérateurs 432
10.3. Fonctions 436
10.4. Stockage de valeurs 440
10.5. Constructions UNION, CASE et constructions relatives 441
10.6. Colonnes de sortie du SELECT 443
11. Index 444
11.1. Introduction 444
11.2. Types d'index 445
11.2.1. B-Tree 445
11.2.2. Hash 446
11.2.3. GiST 446
11.2.4. SP-GiST 446
11.2.5. GIN 446
11.2.6. BRIN 447
11.3. Index multicolonnes 447
11.4. Index et ORDER BY 448
11.5. Combiner des index multiples 449
11.6. Index d'unicité 450
11.7. Index d'expressions 450
11.8. Index partiels 451
11.9. Parcours d'index seul et index couvrants 454
11.10. Classes et familles d'opérateurs 457
11.11. Index et collationnements 459
11.12. Examiner l'utilisation des index 459
12. Recherche plein texte 461
12.1. Introduction 461
12.1.1. Qu'est-ce qu'un document ? 462
12.1.2. Correspondance de base d'un texte 463
12.1.3. Configurations 465

12.2. Tables et index 465
12.2.1. Rechercher dans une table 465
12.2.2. Créer des index 466
12.3. Contrôler la recherche plein texte 468
12.3.1. Analyser des documents 468
12.3.2. Analyser des requêtes 469
12.3.3. Ajouter un score aux résultats d'une recherche 472
12.3.4. Surligner les résultats 474
12.4. Fonctionnalités supplémentaires 476
12.4.1. Manipuler des documents 476
12.4.2. Manipuler des requêtes 477
12.4.3. Triggers pour les mises à jour automatiques 480
12.4.4. Récupérer des statistiques sur les documents 481
12.5. Analyseurs 482
12.6. Dictionnaires 484
12.6.1. Termes courants 485
12.6.2. Dictionnaire simple 486
12.6.3. Dictionnaire des synonymes 487
12.6.4. Dictionnaire thésaurus 489
12.6.5. Dictionnaire Ispell 491
12.6.6. Dictionnaire Snowball 494
12.7. Exemple de configuration 494
12.8. Tester et déboguer la recherche plein texte 496
12.8.1. Test d'une configuration 496
12.8.2. Test de l'analyseur 499
12.8.3. Test des dictionnaires 500
12.9. Types d'index préférés pour la recherche plein texte 501
12.10. Support de psql 502
12.11. Limites 505
13. Contrôle d'accès simultané 507
13.1. Introduction 507
13.2. Isolation des transactions 507
13.2.1. Niveau d'isolation Read committed (lecture uniquement des données validées) 508
13.2.2. Repeatable Read Isolation Level 510
13.2.3. Niveau d'Isolation Serializable 511
13.3. Verrouillage explicite 514
13.3.1. Verrous de niveau table 514
13.3.2. Verrous au niveau ligne 516
13.3.3. Verrous au niveau page 518
13.3.4. Verrous morts (blocage) 518
13.3.5. Verrous informatifs 519
13.4. Vérification de cohérence des données au niveau de l'application 520
13.4.1. Garantir la Cohérence avec des Transactions Serializable 520
13.4.2. Garantir la Cohérence avec des Verrous Bloquants Explicites 521
13.5. Gestion des échecs de sérialisation 522
13.6. Avertissements 522
13.7. Verrous et index 523
14. Conseils sur les performances 524
14.1. Utiliser `EXPLAIN` 524
14.1.1. Concepts d'`EXPLAIN` 524
14.1.2. `EXPLAIN ANALYZE` 531
14.1.3. Avertissements 535
14.2. Statistiques utilisées par le planificateur 536
14.2.1. Statistiques mono-colonne 536
14.2.2. Statistiques étendues 538
14.3. Contrôler le planificateur avec des clauses `JOIN` explicites 541
14.4. Remplir une base de données 543

14.4.1. Désactivez la validation automatique (autocommit) 543
14.4.2. Utilisez COPY 544
14.4.3. Supprimez les index 544
14.4.4. Suppression des contraintes de clés étrangères 544
14.4.5. Augmentez maintenance_work_mem 545
14.4.6. Augmenter max_wal_size 545
14.4.7. Désactiver l'archivage des journaux de transactions et la réplication en flux 545
14.4.8. Lancez ANALYZE après 545
14.4.9. Quelques notes sur pg_dump 545
14.5. Configuration avec une perte acceptée 546
15. Requêtes parallélisées 548
15.1. Comment fonctionne la parallélisation des requêtes 548
15.2. Quand la parallélisation des requêtes peut-elle être utilisée ? 549
15.3. Plans parallélisés 550
15.3.1. Parcours parallélisés 550
15.3.2. Jointures parallélisées 551
15.3.3. Agrégations parallélisées 551
15.3.4. Parallel Append 551
15.3.5. Conseils pour les plans parallélisés 552
15.4. Sécurité de la parallélisation 552
15.4.1. Marquage de parallélisation pour les fonctions et agrégats 553

Chapitre 4. Syntaxe SQL

Ce chapitre décrit la syntaxe de SQL. Il donne les fondements pour comprendre les chapitres suivants qui iront plus en détail sur la façon dont les commandes SQL sont appliquées pour définir et modifier des données.

Nous avertissons aussi nos utilisateurs, déjà familiers avec le SQL, qu'ils doivent lire ce chapitre très attentivement, car il existe plusieurs règles et concepts implémentés différemment suivant les bases de données SQL ou spécifiques à PostgreSQL.

4.1. Structure lexicale

Une entrée SQL consiste en une séquence de *commandes*. Une commande est composée d'une séquence de *jetons*, terminés par un point-virgule (« ; »). La fin du flux en entrée termine aussi une commande. Les jetons valides dépendent de la syntaxe particulière de la commande.

Un jeton peut être un *mot-clé*, un *identificateur*, un *identificateur entre guillemets*, une *constante* ou un symbole de caractère spécial. Les jetons sont normalement séparés par des espaces blancs (espace, tabulation, nouvelle ligne), mais n'ont pas besoin de l'être s'il n'y a pas d'ambiguïté (ce qui est seulement le cas si un caractère spécial est adjacent à des jetons d'autres types).

Par exemple, ce qui suit est (syntaxiquement) valide pour une entrée SQL :

```
SELECT * FROM MA_TABLE;
UPDATE MA_TABLE SET A = 5;
INSERT INTO MA_TABLE VALUES (3, 'salut ici');
```

C'est une séquence de trois commandes, une par ligne (bien que cela ne soit pas requis , plusieurs commandes peuvent se trouver sur une même ligne et une commande peut se répartir sur plusieurs lignes).

De plus, des *commentaires* peuvent se trouver dans l'entrée SQL. Ce ne sont pas des jetons, ils sont réellement équivalents à un espace blanc.

La syntaxe SQL n'est pas très cohérente en ce qui concerne les jetons identificateurs des commandes, lesquels sont des opérandes ou des paramètres. Les premiers jetons sont généralement le nom de la commande. Dans l'exemple ci-dessus, nous parlons d'une commande « SELECT », d'une commande « UPDATE » et d'une commande « INSERT ». Mais en fait, la commande `UPDATE` requiert toujours un jeton `SET` apparaissant à une certaine position, et cette variante particulière d'`INSERT` requiert aussi un `VALUES` pour être complète. Les règles précises de syntaxe pour chaque commande sont décrites dans la Partie VI.

4.1.1. identificateurs et mots-clés

Les jetons tels que `SELECT`, `UPDATE` ou `VALUES` dans l'exemple ci-dessus sont des exemples de *mots-clés*, c'est-à-dire des mots qui ont une signification dans le langage SQL. Les jetons `MA_TABLE` et `A` sont des exemples d'*identificateurs*. Ils identifient des noms de tables, colonnes ou d'autres objets de la base de données, suivant la commande qui a été utilisée. Du coup, ils sont quelques fois simplement nommées des « noms ». Les mots-clés et les identificateurs ont la même structure lexicale, signifiant que quelqu'un ne peut pas savoir si un jeton est un identificateur ou un mot-clé sans connaître le langage. Une liste complète des mots-clés est disponible dans l'Annexe C.

Les identificateurs et les mots-clés SQL doivent commencer avec une lettre (`a-z`, mais aussi des lettres de marques diacritiques différentes et des lettres non latines) ou un tiret bas (`_`). Les caractères suivants dans un identificateur ou dans un mot-clé peuvent être des lettres, des tirets bas, des chiffres (`0-9`) ou des signes dollar (`$`). Notez que les signes dollar ne sont pas autorisés en tant qu'identificateur d'après

le standard SQL, donc leur utilisation pourrait rendre les applications moins portables. Le standard SQL ne définira pas un mot-clé contenant des chiffres ou commençant ou finissant par un tiret bas, donc les identificateurs de cette forme sont sûrs de ne pas entrer en conflit avec les futures extensions du standard.

Le système utilise au plus NAMEDATALEN-1 octets d'un identificateur ; les noms longs peuvent être écrits dans des commandes, mais ils seront tronqués. Par défaut, NAMEDATALEN vaut 64. Du coup, la taille maximale de l'identificateur est de 63 octets. Si cette limite est problématique, elle peut être élevée en modifiant NAMEDATALEN dans src/include/pg_config_manual.h.

Les mots-clés et les identificateurs sans guillemets doubles sont insensibles à la casse. Du coup :

```
UPDATE MA_TABLE SET A = 5;
```

peut aussi s'écrire de cette façon :

```
uPDaTE ma_TabLE SeT a = 5;
```

Une convention couramment utilisée revient à écrire les mots-clés en majuscule et les noms en minuscule, c'est-à-dire :

```
UPDATE ma_table SET a = 5;
```

Voici un deuxième type d'identificateur : l'*identificateur délimité* ou l'*identificateur entre guillemets*. Il est formé en englobant une séquence arbitraire de caractères entre des guillemets doubles ("). Un identificateur délimité est toujours un identificateur, jamais un mot-clé. Donc, "select" pourrait être utilisé pour faire référence à une colonne ou à une table nommée « select », alors qu'un select sans guillemets sera pris pour un mot-clé et, du coup, pourrait provoquer une erreur d'analyse lorsqu'il est utilisé alors qu'un nom de table ou de colonne est attendu. L'exemple peut être écrit avec des identificateurs entre guillemets comme ceci :

```
UPDATE "ma_table" SET "a" = 5;
```

Les identificateurs entre guillemets peuvent contenir tout caractère autre que celui de code 0. (Pour inclure un guillemet double, écrivez deux guillemets doubles.) Ceci permet la construction de noms de tables et de colonnes qui ne seraient pas possibles autrement, comme des noms contenant des espaces ou des arobases. La limitation de la longueur s'applique toujours.

Mettre entre guillemets un identifiant le rend sensible à la casse, alors que les noms sans guillemets sont toujours transformés en minuscule. Par exemple, les identifiants FOO, foo et "foo" sont considérés identiques par PostgreSQL, mais "Foo" et "FOO" sont différents des trois premiers et entre eux. (la mise en minuscule des noms sans guillemets dans PostgreSQL est incompatible avec le standard SQL, qui indique que les noms sans guillemets devraient être convertis en majuscule. De ce fait, foo devrait être équivalent à "FOO", et non pas à "foo", d'après le standard. Si vous voulez écrire des applications portables, il est conseillé de soit toujours mettre les noms entre guillemets, soit ne jamais les mettre entre guillemets.)

Une variante des identificateurs entre guillemets permet d'inclure des caractères Unicode échappés en les identifiant par leur point de code. Cette variante commence par U& (U en majuscule ou minuscule suivi par un « et commercial ») immédiatement suivis par un guillemet double d'ouverture, sans espace entre eux. Par exemple U&"foo". (Notez que c'est source d'ambiguïté avec l'opérateur &. Utilisez les espaces autour de l'opérateur pour éviter ce problème.) À l'intérieur des guillemets, les caractères Unicode peuvent être indiqués dans une forme échappée en écrivant un antislash suivi par le code hexadécimal sur quatre chiffres ou, autre possibilité, un antislash suivi du signe plus suivi d'un code hexadécimal sur six chiffres. Par exemple, l'identificateur "data" peut être écrit ainsi :

```
U&"d\0061t\+000061"
```

L'exemple suivant, moins trivial, écrit le mot russe « slon » (éléphant) en lettres cyrilliques :

```
U&"\0441\043B\043E\043D"
```

Si un caractère d'échappement autre que l'antislash est désiré, il peut être indiqué en utilisant la clause UESCAPE après la chaîne. Par exemple :

```
U&"d!0061t!+000061" UESCAPE '!'
```

La chaîne d'échappement peut être tout caractère simple autre qu'un chiffre hexadécimal, le signe plus, un guillemet simple ou double, ou un espace blanc. Notez que le caractère d'échappement est écrit entre guillemets simples, pas entre guillemets doubles, après UESCAPE.

Pour inclure le caractère d'échappement dans l'identificateur sans interprétation, écrivez-le deux fois.

La forme sur quatre chiffres et la forme sur six chiffres peuvent être utilisées pour indiquer des paires UTF-16, composant ainsi des caractères comprenant des points de code plus grands que U+FFFF (et ce, bien que la disponibilité de la forme sur six chiffres ne le nécessite pas techniquement). (Les paires de substitution ne sont pas stockées directement, mais sont combinées dans un point de code seul.)

Si l'encodage du serveur n'est pas UTF-8, le point code Unicode identifié par une des ces séquences d'échappement est converti vers l'encodage actuel du serveur. Une erreur est renvoyée si ce n'est pas possible.

4.1.2. Constantes

Il existe trois *types implicites de constantes* dans PostgreSQL : les chaînes, les chaînes de bits et les nombres. Les constantes peuvent aussi être spécifiées avec des types explicites, ce qui peut activer des représentations plus précises et gérées plus efficacement par le système. Les constantes implicites sont décrites ci-dessous ; ces constantes sont discutées dans les sous-sections suivantes.

4.1.2.1. Constantes de chaînes

Une constante de type chaîne en SQL est une séquence arbitraire de caractères entourée par des guillemets simples ('), par exemple 'Ceci est une chaîne'. Pour inclure un guillemet simple dans une chaîne constante, saisissez deux guillemets simples adjacents, par exemple 'Le cheval d''Anne'. Notez que ce n'est *pas* identique à un guillemet double (").

Deux constantes de type chaîne séparées par un espace blanc *avec au moins une nouvelle ligne* sont concaténées et traitées réellement comme si la chaîne avait été écrite dans une constante. Par exemple :

```
SELECT 'foo'
'bar';
```

est équivalent à :

```
SELECT 'foobar';
```

mais :

```
SELECT 'foo'      'bar';
```

n'a pas une syntaxe valide (ce comportement légèrement bizarre est spécifié par le standard SQL ; PostgreSQL suit le standard).

4.1.2.2. Constantes chaîne avec des échappements de style C

PostgreSQL accepte aussi les constantes de chaîne utilisant des échappements qui sont une extension au standard SQL. Une constante de type chaîne d'échappement est indiquée en écrivant la lettre `E` (en majuscule ou minuscule) juste avant le guillemet d'ouverture, par exemple `E'foo'`. (Pour continuer une constante de ce type sur plusieurs lignes, écrire `E` seulement avant le premier guillemet d'ouverture.) À l'intérieur d'une chaîne d'échappement, un caractère antislash (`\`) est géré comme une séquence d'*échappement avec antislash* du langage C. La combinaison d'antislash et du (ou des) caractère(s) suivant(s) représente une valeur spéciale, comme indiqué dans le Tableau 4.1.

Tableau 4.1. Séquences d'échappements avec antislash

Séquence d'échappement avec antislash	Interprétation
`\b`	suppression
`\f`	retour en début de ligne
`\n`	saut de ligne
`\r`	saut de ligne
`\t`	tabulation
`\`*o*, `\`*oo*, `\`*ooo* (*o* = 0–7)	valeur octale
`\x`*h*, `\x`*hh* (*h* = 0–9, A–F)	valeur hexadécimale
`\u`*xxxx*, `\U`*xxxxxxxx* (*x* = 0–9, A–F)	caractère Unicode hexadécimal sur 16 ou 32 bits

Tout autre caractère suivi d'un antislash est pris littéralement. Du coup, pour inclure un caractère antislash, écrivez deux antislashs (`\\`). De plus, un guillemet simple peut être inclus dans une chaîne d'échappement en écrivant `\'`, en plus de la façon normale `''`.

Il est de votre responsabilité que les séquences d'octets que vous créez, tout spécialement lorsque vous utilisez les échappements octaux et hexadécimaux, soient des caractères valides dans l'encodage du jeu de caractères du serveur. Une alternative utile est d'utiliser les échappements Unicode ou l'autre syntaxe d'échappement Unicode, expliqués dans la Section 4.1.2.3; ensuite le serveur vérifiera que la conversion de caractères est possible.

> **Attention**
>
> Si le paramètre de configuration standard_conforming_strings est désactivé (`off`), alors PostgreSQL reconnaît les échappements antislashs dans les constantes traditionnelles de type chaînes et celles échappées. Néanmoins, à partir de PostgreSQL 9.1, la valeur par défaut est `on`, ce qui signifie que les échappements par antislash ne sont reconnus que dans les constantes de chaînes d'échappement. Ce comportement est plus proche du standard SQL, mais pourrait causer des problèmes aux applications qui se basent sur le comportement historique où les échappements par antislash étaient toujours reconnus. Pour contourner ce problème, vous pouvez configurer ce paramètre à `off`, bien qu'il soit préférable de ne plus utiliser les échappements par antislash. Si vous avez besoin d'un échappement par antislash pour représenter un caractère spécial, écrivez la chaîne fixe avec un `E`.
>
> En plus de `standard_conforming_strings`, les paramètres de configuration escape_string_warning et backslash_quote imposent le traitement des antislashs dans les constantes de type chaîne.

Le caractère de code zéro ne peut pas être placé dans une constante de type chaîne.

4.1.2.3. Constantes de chaînes avec des échappements Unicode

PostgreSQL supporte aussi un autre type de syntaxe d'échappement pour les chaînes qui permettent d'indiquer des caractères Unicode arbitraires par code. Une constante de chaîne d'échappement

Unicode commence avec U& (U en majuscule ou minuscule suivi par un « et commercial ») immédiatement suivi par un guillemet double d'ouverture, sans espace entre eux. Par exemple U&"foo". (Notez que c'est source d'ambiguïté avec l'opérateur &. Utilisez les espaces autour de l'opérateur pour éviter ce problème.) À l'intérieur des guillemets, les caractères Unicode peuvent être indiqués dans une forme échappée en écrivant un antislash suivi par le code hexadécimal sur quatre chiffres ou, autre possibilité, un antislash suivi du signe plus suivi d'un code hexadécimal sur six chiffres. Par exemple, l'identificateur 'data' peut être écrit ainsi :

```
U&'d\0061t\+000061'
```

L'exemple suivant, moins trivial, écrit le mot russe « slon » (éléphant) en lettres cyrilliques :

```
U&'\0441\043B\043E\043D'
```

Si un caractère d'échappement autre que l'antislash est souhaité, il peut être indiqué en utilisant la clause UESCAPE après la chaîne. Par exemple :

```
U&'d!0061t!+000061' UESCAPE '!'
```

Le caractère d'échappement peut être tout caractère simple autre qu'un chiffre hexadécimal, le signe plus, un guillemet simple ou double, ou un espace blanc.

Pour inclure le caractère d'échappement dans la chaîne, écrivez-la deux fois.

Les formes d'échappement à 4 chiffres et 6 chiffres sont utilisables pour spécifier les paires surrogates UTF-16 pour composer des caractères avec des codes points plus larges que U+FFFF, bien que la disponibilité de la forme à 6 chiffres la rende non nécessaire. (Les paires surrogates ne sont pas enregistrées directement, mais sont combinées dans un seul point code.)

Si l'encodage du serveur n'est pas en UTF-8, le point code Unicode identifié par une de ces séquences d'échappement est converti dans l'encodage réel du serveur ; une erreur est renvoyée quand la conversion n'est pas possible.

De plus, la syntaxe d'échappement de l'Unicode pour les constantes de chaînes fonctionne seulement quand le paramètre de configuration standard_conforming_strings est activé. Dans le cas contraire, cette syntaxe est confuse pour les clients qui analysent les instructions SQL, au point que cela pourrait amener des injections SQL et des problèmes de sécurité similaires. Si le paramètre est désactivé, cette syntaxe sera rejetée avec un message d'erreur.

4.1.2.4. Constantes de chaînes avec guillemet dollar

Alors que la syntaxe standard pour la spécification des constantes de chaînes est généralement agréable, elle peut être difficile à comprendre quand la chaîne désirée contient un grand nombre de guillemets simples car chacun d'entre eux doit être doublé. Pour permettre la saisie de requêtes plus lisibles dans de telles situations, PostgreSQL fournit une autre façon, appelée « guillemet dollar », pour écrire des constantes de chaînes. Une constante de chaîne avec guillemet dollar consiste en un signe dollar ($), une « balise » optionnelle de zéro ou plus de caractères, un autre signe dollar, une séquence arbitraire de caractères qui constitue le contenu de la chaîne, un signe dollar, la même balise et un signe dollar. Par exemple, voici deux façons de spécifier la chaîne « Le cheval d'Anne » en utilisant les guillemets dollar :

```
$$Le cheval d'Anne$$
$UneBalise$Le cheval d'Anne$UneBalise$
```

Notez qu'à l'intérieur de la chaîne avec guillemet dollar, les guillemets simples peuvent être utilisés sans devoir être échappés. En fait, aucun caractère à l'intérieur d'une chaîne avec guillemet dollar n'a besoin d'être échappé : le contenu est toujours écrit littéralement. Les antislashs ne sont pas spéciaux, pas plus que les signes dollar, sauf s'ils font partie d'une séquence correspondant à la balise ouvrante.

Il est possible d'imbriquer les constantes de chaînes avec guillemets dollar en utilisant différentes balises pour chaque niveau d'imbrication. Ceci est habituellement utilisé lors de l'écriture de définition de fonctions. Par exemple :

```
$fonction$
BEGIN
  RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$fonction$
```

Dans cet exemple, la séquence `$q$[\t\r\n\v\\]$q$` représente une chaîne constante avec guillemet dollar `[\t\r\n\v\\]`, qui sera reconnue quand le corps de la fonction est exécuté par PostgreSQL. Mais comme la séquence ne correspond pas au délimiteur `$fonction$`, il s'agit juste de quelques caractères à l'intérieur de la constante pour ce qu'en sait la chaîne externe.

La balise d'une chaîne avec guillemets dollar, si elle existe, suit les mêmes règles qu'un identificateur sans guillemets, sauf qu'il ne peut pas contenir de signes dollar. Les balises sont sensibles à la casse, du coup `$balise$Contenu de la chaîne$balise$` est correct, mais `$BALISE$Contenu de la chaîne$balise$` ne l'est pas.

Une chaîne avec guillemets dollar suivant un mot clé ou un identificateur doit en être séparée par un espace blanc ; sinon, le délimiteur du guillemet dollar serait pris comme faisant partie de l'identificateur précédent.

Le guillemet dollar ne fait pas partie du standard SQL, mais c'est un moyen bien plus agréable pour écrire des chaînes constantes que d'utiliser la syntaxe des guillemets simples, bien que compatible avec le standard. Elle est particulièrement utile pour représenter des constantes de type chaîne à l'intérieur d'autres constantes, comme cela est souvent le cas avec les définitions de fonctions. Avec la syntaxe des guillemets simples, chaque antislash dans l'exemple précédent devrait avoir été écrit avec quatre antislashs, ce qui sera réduit à deux antislashs dans l'analyse de la constante originale, puis à un lorsque la constante interne est analysée de nouveau lors de l'exécution de la fonction.

4.1.2.5. Constantes de chaînes de bits

Les constantes de chaînes de bits ressemblent aux constantes de chaînes standard avec un `B` (majuscule ou minuscule) juste avant le guillemet du début (sans espace blanc), c'est-à-dire `B'1001'`. Les seuls caractères autorisés dans les constantes de type chaîne de bits sont `0` et `1`.

Les constantes de chaînes de bits peuvent aussi être spécifiées en notation hexadécimale en utilisant un `X` avant (minuscule ou majuscule), c'est-à-dire `X'1FF'`. Cette notation est équivalente à une constante de chaîne de bits avec quatre chiffres binaires pour chaque chiffre hexadécimal.

Les deux formes de constantes de chaînes de bits peuvent être continuées sur plusieurs lignes de la même façon que les constantes de chaînes habituelles. Le guillemet dollar ne peut pas être utilisé dans une constante de chaîne de bits.

4.1.2.6. Constantes numériques

Les constantes numériques sont acceptées dans ces formes générales :

chiffres
chiffres.[*chiffres*][e[+-]*chiffres*]
[*chiffres*].*chiffres*[e[+-]*chiffres*]
*chiffres*e[+-]*chiffres*

où *chiffres* est un ou plusieurs chiffres décimaux (de 0 à 9). Au moins un chiffre doit être avant ou après le point décimal, s'il est utilisé. Au moins un chiffre doit suivre l'indicateur d'exponentielle (e), s'il est présent. Il ne peut pas y avoir d'espaces ou d'autres caractères imbriqués dans la constante. Notez que tout signe plus ou moins en avant n'est pas considéré comme faisant part de la constante ; il est un opérateur appliqué à la constante.

Voici quelques exemples de constantes numériques valides :

42
3.5
4.
.001
5e2
1.925e-3

Une constante numérique ne contenant ni un point décimal ni un exposant est tout d'abord présumée du type `integer` si sa valeur est contenue dans le type `integer` (32 bits) ; sinon, il est présumé de type `bigint` si sa valeur entre dans un type `bigint` (64 bits) ; sinon, il est pris pour un type `numeric`. Les constantes contenant des points décimaux et/ou des exposants sont toujours présumées de type `numeric`.

Le type de données affecté initialement à une constante numérique est seulement un point de départ pour les algorithmes de résolution de types. Dans la plupart des cas, la constante sera automatiquement convertie dans le type le plus approprié suivant le contexte. Si nécessaire, vous pouvez forcer l'interprétation d'une valeur numérique sur un type de données spécifique en la convertissant. Par exemple, vous pouvez forcer une valeur numérique à être traitée comme un type `real` (`float4`) en écrivant :

```
REAL '1.23'  -- style chaîne
1.23::REAL   -- style PostgreSQL (historique)
```

Ce sont en fait des cas spéciaux des notations de conversion générales discutées après.

4.1.2.7. Constantes d'autres types

Une constante de type arbitraire peut être saisie en utilisant une des notations suivantes :

```
type 'chaîne'
'chaîne'::type
CAST ( 'chaîne' AS type )
```

Le texte de la chaîne constante est passé dans la routine de conversion pour le type appelé *type*. Le résultat est une constante du type indiqué. La conversion explicite de type peut être omise s'il n'y a pas d'ambiguïté sur le type de la constante (par exemple, lorsqu'elle est affectée directement à une colonne de la table), auquel cas elle est convertie automatiquement.

La constante chaîne peut être écrite en utilisant soit la notation SQL standard soit les guillemets dollar.

Il est aussi possible de spécifier une conversion de type en utilisant une syntaxe style fonction :

```
nom_type ( 'chaîne' )
```

mais tous les noms de type ne peuvent pas être utilisés ainsi ; voir la Section 4.2.9 pour plus de détails.

Les syntaxes `::`, `CAST()` et d'appels de fonctions sont aussi utilisables pour spécifier les conversions de type à l'exécution d'expressions arbitraires, comme discuté dans la Section 4.2.9. Pour éviter une ambiguïté syntaxique, la syntaxe *type* `'chaîne'` peut seulement être utilisée pour spécifier le type d'une constante. Une autre restriction sur la syntaxe *type* `'chaîne'` est qu'elle ne fonctionne pas pour les types de tableau ; utilisez `::` ou `CAST()` pour spécifier le type d'une constante de type tableau.

La syntaxe de `CAST()` est conforme au standard SQL. La syntaxe *`type 'chaine'`* est une généralisation du standard : SQL spécifie cette syntaxe uniquement pour quelques types de données, mais PostgreSQL l'autorise pour tous les types. La syntaxe `::` est un usage historique dans PostgreSQL, comme l'est la syntaxe d'appel de fonction.

4.1.3. Opérateurs

Un nom d'opérateur est une séquence d'au plus `NAMEDATALEN`-1 (63 par défaut) caractères provenant de la liste suivante :

+ - * / < > = ~ ! @ # % ^ & | ` ?

Néanmoins, il existe quelques restrictions sur les noms d'opérateurs :

- `--` et `/*` ne peuvent pas apparaître quelque part dans un nom d'opérateur, car ils seront pris pour le début d'un commentaire.
- Un nom d'opérateur à plusieurs caractères ne peut pas finir avec `+` ou `-`, sauf si le nom contient aussi un de ces caractères :

  ~ ! @ # % ^ & | ` ?

  Par exemple, `@-` est un nom d'opérateur autorisé, mais `*-` ne l'est pas. Cette restriction permet à PostgreSQL d'analyser des requêtes compatibles avec SQL sans requérir des espaces entre les jetons.

Lors d'un travail avec des noms d'opérateurs ne faisant pas partie du standard SQL, vous aurez habituellement besoin de séparer les opérateurs adjacents avec des espaces pour éviter toute ambiguïté. Par exemple, si vous avez défini un opérateur préfixe nommé `@`, vous ne pouvez pas écrire `X*@Y` ; vous devez écrire `X* @Y` pour vous assurer que PostgreSQL le lit comme deux noms d'opérateurs, et non pas comme un seul.

4.1.4. Caractères spéciaux

Quelques caractères non alphanumériques ont une signification spéciale, différente de celle d'un opérateur. Les détails sur leur utilisation sont disponibles à l'endroit où l'élément de syntaxe respectif est décrit. Cette section existe seulement pour avertir de leur existence et pour résumer le but de ces caractères.

- Un signe dollar (`$`) suivi de chiffres est utilisé pour représenter un paramètre de position dans le corps de la définition d'une fonction ou d'une instruction préparée. Dans d'autres contextes, le signe dollar pourrait faire partie d'un identificateur ou d'une constante de type chaîne utilisant le dollar comme guillemet.
- Les parenthèses (`()`) ont leur signification habituelle pour grouper leurs expressions et renforcer la précédence. Dans certains cas, les parenthèses sont requises, car faisant partie de la syntaxe d'une commande SQL particulière.
- Les crochets (`[]`) sont utilisés pour sélectionner les éléments d'un tableau. Voir la Section 8.15 pour plus d'informations sur les tableaux.
- Les virgules (`,`) sont utilisées dans quelques constructions syntaxiques pour séparer les éléments d'une liste.
- Le point-virgule (`;`) termine une commande SQL. Il ne peut pas apparaître quelque part dans une commande, sauf à l'intérieur d'une constante de type chaîne ou d'un identificateur entre guillemets.
- Le caractère deux points (`:`) est utilisé pour sélectionner des « morceaux » de tableaux (voir la Section 8.15). Dans certains dialectes SQL (tel que le SQL embarqué), il est utilisé pour préfixer les noms de variables.

- L'astérisque (`*`) est utilisé dans certains contextes pour indiquer tous les champs de la ligne d'une table ou d'une valeur composite. Elle a aussi une signification spéciale lorsqu'elle est utilisée comme argument d'une fonction d'agrégat. Cela signifie que l'agrégat ne requiert pas de paramètre explicite.

- Le point (`.`) est utilisé dans les constantes numériques et pour séparer les noms de schéma, table et colonne.

4.1.5. Commentaires

Un commentaire est une séquence de caractères commençant avec deux tirets et s'étendant jusqu'à la fin de la ligne, par exemple :

```
-- Ceci est un commentaire standard en SQL
```

Autrement, les blocs de commentaires style C peuvent être utilisés :

```
/* commentaires multilignes
 * et imbriqués: /* bloc de commentaire imbriqué */
 */
```

où le commentaire commence avec `/*` et s'étend jusqu'à l'occurrence de `*/`. Ces blocs de commentaires s'imbriquent, comme spécifié dans le standard SQL, mais pas comme dans le langage C. De ce fait, vous pouvez commenter des blocs importants de code pouvant contenir des blocs de commentaires déjà existants.

Un commentaire est supprimé du flux en entrée avant une analyse plus poussée de la syntaxe et est remplacé par un espace blanc.

4.1.6. Précédence d'opérateurs

Le Tableau 4.2 affiche la précédence et l'associativité des opérateurs dans PostgreSQL. La plupart des opérateurs ont la même précédence et sont associatifs par la gauche. La précédence et l'associativité des opérateurs sont codées en dur dans l'analyseur. Ajoutez des parenthèses si vous voulez qu'une expression avec plusieurs opérateurs soit analysée d'une autre façon que celle que les règles de précédence auraient impliquées.

Tableau 4.2. Précédence des opérateurs (du plus haut vers le plus bas)

Opérateur/Élément	Associativité	Description
`.`	gauche	séparateur de noms de table et de colonne
`::`	gauche	conversion de type, style PostgreSQL
`[ ]`	gauche	sélection d'un élément d'un tableau
`+ -`	droite	plus unaire, moins unaire
`COLLATE`	gauche	sélection de collation
`AT`	gauche	`AT TIME ZONE`
`^`	gauche	exposant
`* / %`	gauche	multiplication, division, modulo
`+ -`	gauche	addition, soustraction
(autres opérateurs)	gauche	tout autre opérateur natif ou défini par l'utilisateur
`BETWEEN IN LIKE ILIKE SIMILAR`		intervalle contenu, recherche d'appartenance, correspondance de chaîne

Opérateur/Élément	Associativité	Description
< > = <= >= <>		opérateurs de comparaison
IS ISNULL NOTNULL		IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM, etc
NOT	droite	négation logique
AND	gauche	conjonction logique
OR	gauche	disjonction logique

Notez que les règles de précédence des opérateurs s'appliquent aussi aux opérateurs définis par l'utilisateur qui ont le même nom que les opérateurs internes mentionnés ici. Par exemple, si vous définissez un opérateur « + » pour un type de données personnalisé, il aura la même précédence que l'opérateur interne « + », peu importe ce que fait le vôtre.

Lorsqu'un nom d'opérateur qualifié par un schéma est utilisé dans la syntaxe OPERATOR, comme dans :

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

la construction OPERATOR est prise pour avoir la précédence par défaut affichée dans le Tableau 4.2 pour les opérateurs « autres ». Ceci est vrai, quel que soit le nom spécifique de l'opérateur apparaissant à l'intérieur de OPERATOR().

Note

Les versions de PostgreSQL antérieures à la 9.5 utilisaient des règles de précédence différentes pour les opérateurs. En particulier, <= >= et <> étaient traités comme des opérateurs génériques ; les tests IS avaient une priorité supérieure ; NOT BETWEEN et les constructions qui en découlent agissaient de façon incohérente, ayant dans certains cas la précédence de NOT plutôt que de BETWEEN. Ces règles étaient modifiées pour un meilleur accord avec le standard SQL et pour réduire la configuration d'un traitement incohérent de constructions équivalentes logiquement. Dans la plupart des cas, ces changements ne résulteront pas en un changement de comportement. Il peut arriver que des échecs du type « opérateur inconnu » surviennent, auquel cas un ajout de parenthèses devrait corriger le problème. Néanmoins, il existe des cas particuliers où une requête pourrait voir son comportement changé sans qu'une erreur d'analyse soit renvoyée.

4.2. Expressions de valeurs

Les expressions de valeurs sont utilisées dans une grande variété de contextes, tels que dans la liste cible d'une commande SELECT, dans les nouvelles valeurs de colonnes d'une commande INSERT ou UPDATE, ou dans les conditions de recherche d'un certain nombre de commandes. Le résultat d'une expression de valeurs est quelquefois appelé *scalaire*, pour le distinguer du résultat d'une expression de table (qui est une table). Les expressions de valeurs sont aussi appelées des *expressions scalaires* (voire simplement des *expressions*). La syntaxe d'expression permet le calcul des valeurs à partir de morceaux primitifs en utilisant les opérations arithmétiques, logiques, d'ensemble et autres.

Une expression de valeur peut être :

- une constante ou une valeur constante ;
- une référence de colonne ;
- une référence de la position d'un paramètre, dans le corps d'une définition de fonction ou d'instruction préparée ;

- une expression indicée ;
- une expression de sélection de champs ;
- un appel d'opérateur ;
- un appel de fonction ;
- une expression d'agrégat ;
- un appel de fonction de fenêtrage ;
- une conversion de type ;
- une expression de collationnement ;
- une sous-requête scalaire ;
- un constructeur de tableau ;
- un constructeur de ligne ;
- toute expression de valeur entre parenthèses, utile pour grouper des sous-expressions et surcharger la précédence.

En plus de cette liste, il existe un certain nombre de constructions pouvant être classées comme une expression, mais ne suivant aucune règle de syntaxe générale. Elles ont généralement la sémantique d'une fonction ou d'un opérateur et sont expliquées au Chapitre 9. Un exemple est la clause `IS NULL`.

Nous avons déjà discuté des constantes dans la Section 4.1.2. Les sections suivantes discutent des options restantes.

4.2.1. Références de colonnes

Une colonne peut être référencée avec la forme :

`correlation.nom_colonne`

`correlation` est le nom d'une table (parfois qualifié par son nom de schéma) ou un alias d'une table définie au moyen de la clause `FROM`. Le nom de corrélation et le point de séparation peuvent être omis si le nom de colonne est unique dans les tables utilisées par la requête courante (voir aussi le Chapitre 7).

4.2.2. Paramètres de position

Un paramètre de position est utilisé pour indiquer une valeur fournie en externe par une instruction SQL. Les paramètres sont utilisés dans des définitions de fonction SQL et dans les requêtes préparées. Quelques bibliothèques clients supportent aussi la spécification de valeurs de données séparément de la chaîne de commandes SQL, auquel cas les paramètres sont utilisés pour référencer les valeurs de données en dehors. Le format d'une référence de paramètre est :

`$`*`numéro`*

Par exemple, considérez la définition d'une fonction : `dept` :

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE nom = $1 $$
    LANGUAGE SQL;
```

Dans cet exemple, `$1` référence la valeur du premier argument de la fonction à chaque appel de cette commande.

4.2.3. Indices

Si une expression récupère une valeur de type tableau, alors un élément spécifique du tableau peut être extrait en écrivant :

expression[*indice*]

Des éléments adjacents (un « morceau de tableau ») peuvent être extraits en écrivant :

expression[*indice_bas*:*indice_haut*]

Les crochets [ ] doivent apparaître réellement. Chaque *indice* est lui-même une expression, dont la valeur sera arrondie à la valeur entière la plus proche.

En général, l'*expression* de type tableau doit être entre parenthèses, mais ces dernières peuvent être omises lorsque l'expression utilisée comme indice est seulement une référence de colonne ou un paramètre de position. De plus, les indices multiples peuvent être concaténés lorsque le tableau original est multidimensionnel. Par exemple :

```
ma_table.colonnetableau[4]
ma_table.colonnes_deux_d[17][34]
$1[10:42]
(fonctiontableau(a,b))[42]
```

Dans ce dernier exemple, les parenthèses sont requises. Voir la Section 8.15 pour plus d'informations sur les tableaux.

4.2.4. Sélection de champs

Si une expression récupère une valeur de type composite (type row), alors un champ spécifique de la ligne est extrait en écrivant :

expression.*nom_champ*

En général, l'*expression* de ligne doit être entre parenthèses, mais les parenthèses peuvent être omises lorsque l'expression à partir de laquelle se fait la sélection est seulement une référence de table ou un paramètre de position. Par exemple :

```
ma_table.macolonne
$1.unecolonne
(fonctionligne(a,b)).col3
```

En fait, une référence de colonne qualifiée est un cas spécial de syntaxe de sélection de champ. Un cas spécial important revient à extraire un champ de la colonne de type composite d'une table :

```
(colcomposite).unchamp
(matable.colcomposite).unchamp
```

Les parenthèses sont requises ici pour montrer que `colcomposite` est un nom de colonne, et non pas un nom de table, ou que `matable` est un nom de table, pas un nom de schéma dans le deuxième cas.

Vous pouvez demander tous les champs d'une valeur composite en écrivant `.*` :

```
(compositecol).*
```

Cette syntaxe se comporte différemment suivant le contexte. Voir Section 8.16.5 pour plus de détails.

4.2.5. Appels d'opérateurs

Il existe deux syntaxes possibles pour l'appel d'un opérateur :

expression opérateur expression (opérateur binaire préfixe)
opérateur expression (opérateur unaire préfixe)

où le jeton *opérateur* suit les règles de syntaxe de la Section 4.1.3, ou est un des mots-clés AND, OR et NOT, ou est un nom d'opérateur qualifié de la forme

OPERATOR(*schema.nom_operateur*)

Le fait qu'opérateur particulier existe et qu'il soit unaire ou binaire dépend des opérateurs définis par le système ou l'utilisateur. Le Chapitre 9 décrit les opérateurs internes.

4.2.6. Appels de fonctions

La syntaxe pour un appel de fonction est le nom d'une fonction (qualifié ou non du nom du schéma) suivi par sa liste d'arguments entre parenthèses :

nom_fonction([*expression* [,*expression* ...]])

Par exemple, ce qui suit calcule la racine carrée de 2 :

```
sqrt(2)
```

La liste des fonctions intégrées se trouve dans le Chapitre 9. D'autres fonctions pourraient être ajoutées par l'utilisateur.

Lors de l'exécution de requêtes dans une base de données où certains utilisateurs ne font pas confiance aux autres utilisateurs, observez quelques mesures de sécurité disponibles dans Section 10.3 lors de l'appel de fonctions.

En option, les arguments peuvent avoir leur nom attaché. Voir la Section 4.3 pour les détails.

> **Note**
>
> Une fonction qui prend un seul argument de type composite peut aussi être appelée en utilisant la syntaxe de sélection de champ. Du coup, un champ peut être écrit dans le style fonctionnel. Cela signifie que les notations `col(table)` et `table.col` sont interchangeables. Ce comportement ne respecte pas le standard SQL, mais il est fourni dans PostgreSQL, car il permet l'utilisation de fonctions émulant les « champs calculés ». Pour plus d'informations, voir la Section 8.16.5.

4.2.7. Expressions d'agrégat

Une *expression d'agrégat* représente l'application d'une fonction d'agrégat à travers les lignes sélectionnées par une requête. Une fonction d'agrégat réduit les nombres entrés en une seule valeur de sortie, comme la somme ou la moyenne des valeurs en entrée. La syntaxe d'une expression d'agrégat est une des suivantes :

```
nom_agregat (expression [ , ... ] [ clause_order_by ] ) [ FILTER
 ( WHERE clause_filtre ) ]
nom_agregat (ALL expression [ , ... ] [ clause_order_by ] )
 [ FILTER ( WHERE clause_filtre ) ]
```

```
nom_agregat (DISTINCT expression [ , ... ] [ clause_order_by ] )
 [ FILTER ( WHERE clause_filtre ) ]
nom_agregat ( * ) [ FILTER ( WHERE clause_filtre ) ]
nom_agregat ( [ expression [ , ... ] ] ) WITHIN GROUP
 ( clause_order_by ) [ FILTER ( WHERE clause_filtre ) ]
```

où *nom_agregat* est un agrégat précédemment défini (parfois qualifié d'un nom de schéma), *expression* est toute expression de valeur qui ne contient pas elle-même une expression d'agrégat ou un appel à une fonction de fenêtrage. Les clauses optionnelles *clause_order_by* et *clause_filtre* sont décrites ci-dessous.

La première forme d'expression d'agrégat appelle l'agrégat une fois pour chaque ligne en entrée. La seconde forme est identique à la première, car `ALL` est une clause active par défaut. La troisième forme fait appel à l'agrégat une fois pour chaque valeur distincte de l'expression (ou ensemble distinct de valeurs, pour des expressions multiples) trouvée dans les lignes en entrée. La quatrième forme appelle l'agrégat une fois pour chaque ligne en entrée ; comme aucune valeur particulière en entrée n'est spécifiée, c'est généralement utile pour la fonction d'agrégat `count(*)`. La dernière forme est utilisée avec les agrégats à *ensemble trié* qui sont décrits ci-dessous.

La plupart des fonctions d'agrégats ignorent les entrées NULL, pour que les lignes qui renvoient une ou plusieurs expressions NULL soient disqualifiées. Ceci peut être considéré comme vrai pour tous les agrégats internes sauf indication contraire.

Par exemple, `count(*)` trouve le nombre total de lignes en entrée, alors que `count(f1)` récupère le nombre de lignes en entrée pour lesquelles `f1` n'est pas NULL. En effet, la fonction `count` ignore les valeurs NULL, mais `count(distinct f1)` retrouve le nombre de valeurs distinctes non NULL de `f1`.

D'habitude, les lignes en entrée sont passées à la fonction d'agrégat dans un ordre non spécifié. Dans la plupart des cas, cela n'a pas d'importance. Par exemple, `min` donne le même résultat quel que soit l'ordre dans lequel il reçoit les données. Néanmoins, certaines fonctions d'agrégat (telles que `array_agg` et `string_agg`) donnent un résultat dépendant de l'ordre des lignes en entrée. Lors de l'utilisation de ce type d'agrégat, la clause *clause_order_by* peut être utilisée pour préciser l'ordre de tri désiré. La clause *clause_order_by* a la même syntaxe que la clause `ORDER BY` d'une requête, qui est décrite dans la Section 7.5, sauf que ses expressions sont toujours des expressions simples et ne peuvent pas être des noms de colonne en sortie ou des numéros. Par exemple :

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

Lors de l'utilisation de fonctions d'agrégat à plusieurs arguments, la clause `ORDER BY` arrive après tous les arguments de l'agrégat. Par exemple, il faut écrire ceci :

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

et non pas ceci :

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- incorrect
```

Ce dernier exemple est syntaxiquement correct, mais il concerne un appel à une fonction d'agrégat à un seul argument avec deux clés pour le `ORDER BY` (le deuxième étant inutile, car il est constant).

Si `DISTINCT` est indiqué en plus de la clause *clause_order_by*, alors toutes les expressions de l'`ORDER BY` doivent correspondre aux arguments de l'agrégat ; autrement dit, vous ne pouvez pas trier sur une expression qui n'est pas incluse dans la liste `DISTINCT`.

> **Note**
>
> La possibilité de spécifier à la fois `DISTINCT` et `ORDER BY` dans une fonction d'agrégat est une extension de PostgreSQL.

Placer la clause ORDER BY dans la liste des arguments standards de l'agrégat, comme décrit jusqu'ici, est utilisé pour un agrégat de type général et statistique pour lequel le tri est optionnel. Il existe une sous-classe de fonctions d'agrégat appelée *agrégat d'ensemble trié* pour laquelle la clause *clause_order_by* est *requise*, habituellement parce que le calcul de l'agrégat est seulement sensible à l'ordre des lignes en entrée. Des exemples typiques d'agrégat avec ensemble trié incluent les calculs de rang et de pourcentage. Pour un agrégat d'ensemble trié, la clause *clause_order_by* est écrite à l'intérieur de WITHIN GROUP (...), comme indiqué dans la syntaxe alternative finale. Les expressions dans *clause_order_by* sont évaluées une fois par ligne en entrée, comme n'importe quel argument d'un agrégat, une fois triées suivant la clause *clause_order_by*, et envoyées à la fonction en tant qu'arguments en entrée. (Ceci est contraire au cas de la clause *clause_order_by* en dehors d'un WITHIN GROUP , qui n'est pas traité comme argument de la fonction d'agrégat.) Les expressions d'argument précédant WITHIN GROUP, s'il y en a, sont appelées des *arguments directs* pour les distinguer des *arguments agrégés* listés dans *clause_order_by*. Contrairement aux arguments normaux d'agrégats, les arguments directs sont évalués seulement une fois par appel d'agrégat et non pas une fois par ligne en entrée. Cela signifie qu'ils peuvent contenir des variables seulement si ces variables sont regroupées par GROUP BY ; cette restriction équivaut à des arguments directs qui ne seraient pas dans une expression d'agrégat. Les arguments directs sont typiquement utilisés pour des fractions de pourcentage, qui n'ont de sens qu'en tant que valeur singulière par calcul d'agrégat. La liste d'arguments directs peut être vide ; dans ce cas, écrivez simplement (), et non pas (*). (PostgreSQL accepte actuellement les deux écritures, mais seule la première est conforme au standard SQL.)

Voici un exemple d'appel d'agrégat à ensemble trié :

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY revenu) FROM
 proprietes;
 percentile_cont
-----------------
           50489
```

qui obtient le 50e pourcentage ou le médian des valeurs de la colonne revenu de la table proprietes. Ici, 0.5 est un argument direct ; cela n'aurait pas de sens si la fraction de pourcentage était une valeur variant suivant les lignes.

Si la clause FILTER est spécifiée, alors seules les lignes en entrée pour lesquelles *filter_clause* est vraie sont envoyées à la fonction d'agrégat ; les autres lignes sont ignorées. Par exemple :

```
SELECT
    count(*) AS nonfiltres,
    count(*) FILTER (WHERE i < 5) AS filtres
FROM generate_series(1,10) AS s(i);
 nonfiltres | filtres
------------+---------
         10 |       4
(1 row)
```

Les fonctions d'agrégat prédéfinies sont décrites dans la Section 9.21. D'autres fonctions d'agrégat pourraient être ajoutées par l'utilisateur.

Une expression d'agrégat peut seulement apparaître dans la liste de résultats ou dans la clause HAVING d'une commande SELECT. Elle est interdite dans d'autres clauses, telles que WHERE, parce que ces clauses sont logiquement évaluées avant que les résultats des agrégats ne soient calculés.

Lorsqu'une expression d'agrégat apparaît dans une sous-requête (voir la Section 4.2.11 et la Section 9.23), l'agrégat est normalement évalué sur les lignes de la sous-requête. Cependant, une exception survient si les arguments de l'agrégat (et *clause_filtre* si fourni) contiennent

seulement des niveaux externes de variables : ensuite, l'agrégat appartient au niveau externe le plus proche et est évalué sur les lignes de cette requête. L'expression de l'agrégat est une référence externe pour la sous-requête dans laquelle il apparaît et agit comme une constante sur toute évaluation de cette requête. La restriction apparaissant seulement dans la liste de résultats ou dans la clause HAVING s'applique avec respect du niveau de requête auquel appartient l'agrégat.

4.2.8. Appels de fonction de fenêtrage

Un *appel de fonction de fenêtrage* représente l'application d'une fonction de type agrégat sur une portion des lignes sélectionnées par une requête. Contrairement aux appels de fonction d'agrégat standard, ce n'est pas lié au groupement des lignes sélectionnées en une seule ligne résultat -- chaque ligne reste séparée dans les résultats. Néanmoins, la fonction de fenêtrage a accès à toutes les lignes qui font partie du groupe de la ligne courante d'après la spécification du groupe (liste PARTITION BY) de l'appel de la fonction de fenêtrage. La syntaxe d'un appel de fonction de fenêtrage est une des suivantes :

```
nom_fonction ([expression [, expression ... ]]) [ FILTER
 ( WHERE clause_filtre ) ] OVER nom_window
nom_fonction ([expression [, expression ... ]]) [ FILTER
 ( WHERE clause_filtre ) ] OVER ( définition_window )
nom_fonction ( * ) [ FILTER ( WHERE clause_filtre ) ]
 OVER nom_window
nom_fonction ( * ) [ FILTER ( WHERE clause_filtre ) ] OVER
 ( définition_window )
```

où *définition_fenêtrage* a comme syntaxe :

```
[ nom_fenêtrage_existante ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING opérateur ] [ NULLS
 { FIRST | LAST } ] [, ...] ]
[ clause_portée ]
```

et la clause *clause_portée* optionnelle fait partie de :

```
{ RANGE | ROWS | GROUPS } début_portée [ frame_exclusion ]
{ RANGE | ROWS | GROUPS } BETWEEN début_portée AND fin_portée
 [ frame_exclusion ]
```

avec *début_portée* et *fin_portée* pouvant faire partie de

```
UNBOUNDED PRECEDING
décalage PRECEDING
CURRENT ROW
décalage FOLLOWING
UNBOUNDED FOLLOWING
```

et *frame_exclusion* peut valoir

```
EXCLUDE CURRENT ROW
```

```
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

Ici, *`expression`* représente toute expression de valeur qui ne contient pas elle-même d'appel à des fonctions de fenêtrage.

`nom_fenêtrage` est une référence à la spécification d'une fenêtre nommée, définie dans la clause `WINDOW` de la requête. Les spécifications de fenêtres nommées sont habituellement référencées avec `OVER` *`nom_fenêtrage`*, mais il est aussi possible d'écrire un nom de fenêtre entre parenthèses, puis de fournir en option une clause de tri et/ou une clause de portée (la fenêtre référencée ne doit pas avoir ces clauses si elles sont fournies ici). Cette dernière syntaxe suit les mêmes règles que la modification d'un nom de fenêtre existant dans une clause `WINDOW` ; voir la page de référence de SELECT pour les détails.

La clause `PARTITION BY` groupe les lignes de la requête en *partitions*, qui sont traitées séparément par la fonction de fenêtrage. `PARTITION BY` fonctionne de la même façon qu'une clause `GROUP BY` au niveau de la requête, sauf que ses expressions sont toujours des expressions et ne peuvent pas être des noms ou des numéros de colonnes en sortie. Sans `PARTITION BY`, toutes les lignes produites par la requête sont traitées comme une seule partition. La clause `ORDER BY` détermine l'ordre dans lequel les lignes d'une partition sont traitées par la fonction de fenêtrage. Cela fonctionne de la même façon que la clause `ORDER BY` d'une requête, mais ne peut pas non plus utiliser les noms ou les numéros des colonnes en sortie. Sans `ORDER BY`, les lignes sont traitées dans n'importe quel ordre.

La clause *`clause_portée`* indique l'ensemble de lignes constituant la *portée de la fenêtre*, qui est un sous-ensemble de la partition en cours, pour les fonctions de fenêtrage qui agissent sur ce sous-ensemble plutôt que sur la partition entière. L'ensemble de lignes dans la portée peut varier suivant la ligne courante. Le sous-ensemble peut être spécifié avec le mode `RANGE`, avec le mode `ROWS` ou avec le mode `GROUPS`. Dans les deux cas, il s'exécute de *`début_portée`* à *`fin_portée`*. Si *`fin_portée`* est omis, la fin vaut par défaut `CURRENT ROW`.

Un *`début_portée`* à `UNBOUNDED PRECEDING` signifie que le sous-ensemble commence avec la première ligne de la partition. De la même façon, un *`fin_portée`* à `UNBOUNDED FOLLOWING` signifie que le sous-ensemble se termine avec la dernière ligne de la partition.

Dans les modes `RANGE` et `GROUPS`, un *`début_portée`* à `CURRENT ROW` signifie que le sous-ensemble commence avec la ligne suivant la ligne courante (une ligne que la clause `ORDER BY` de la fenêtre considère comme équivalente à la ligne courante), alors qu'un *`fin_portée`* à `CURRENT ROW` signifie que le sous-ensemble se termine avec la dernière ligne homologue de la ligne en cours. Dans le mode `ROWS`, `CURRENT ROW` signifie simplement la ligne courante.

Dans les options de portée, *`offset`* de `PRECEDING` et *`offset`* de `FOLLOWING`, le *`offset`* doit être une expression ne contenant ni variables, ni fonctions d'agrégat, ni fonctions de fenêtrage. La signification de *`offset`* dépend du mode de porté :

- Dans le mode `ROWS`, *`offset`* doit renvoyer un entier non négatif non NULL, et l'option signifie que la portée commence ou finit au nombre spécifié de lignes avant ou après la ligne courante.

- Dans le mode `GROUPS`, *`offset`* doit de nouveau renvoyer un entier non négatif non NULL, et l'option signifie que la portée commence ou finit au nombre spécifié de *groupes de lignes équivalentes* avant ou après le groupe de la ligne courante, et où un groupe de lignes équivalentes est un ensemble de lignes équivalentes dans le tri `ORDER BY`. (Il doit y avoir une clause `ORDER BY` dans la définition de la fenêtre pour utiliser le mode `GROUPS`.)

- Dans le mode `RANGE`, ces options requièrent que la clause `ORDER BY` spécifient exactement une colonne. *`offset`* indique la différence maximale entre la valeur de cette colonne dans la ligne courante et sa valeur dans les lignes précédentes et suivantes de la portée. Le type de données de l'expression *`offset`* varie suivant le type de données de la colonne triée. Pour les colonnes ordonnées numériques, il s'agit habituellement du même type que la colonne ordonnée. Mais pour

les colonnes ordonnées de type date/heure, il s'agit d'un `interval`. Par exemple, si la colonne ordonnée est de type `date` ou `timestamp`, on pourrait écrire `RANGE BETWEEN '1 day' PRECEDING AND '10 days' FOLLOWING`. *`offset`* est toujours requis pour être non NULL et non négatif, bien que la signification de « non négatif » dépend de son type de données.

Dans tous les cas, la distance jusqu'à la fin de la portée est limitée par la distance jusqu'à la fin de la partition, pour que les lignes proche de la fin de la partition, la portée puisse contenir moins de lignes qu'ailleurs.

Notez que dans les deux modes `ROWS` et `GROUPS`, `0 PRECEDING` et `0 FOLLOWING` sont équivalents à `CURRENT ROW`. Le mode `RANGE` en fait aussi partie habituellement, pour une signification appropriée de « zéro » pour le type de données spécifique.

L'option *`frame_exclusion`* permet aux lignes autour de la ligne courante d'être exclues de la portée, même si elles seraient incluses d'après les options de début et de fin de portée. `EXCLUDE CURRENT ROW` exclut la ligne courante de la portée. `EXCLUDE GROUP` exclut la ligne courante et ses équivalents dans l'ordre à partir de la portée. `EXCLUDE TIES` exclut de la portée tout équivalent de la ligne courante mais pas la ligne elle-même. `EXCLUDE NO OTHERS` spécifie explicitement le comportement par défaut lors de la non exclusion de la ligne courante ou de ses équivalents.

L'option par défaut est `RANGE UNBOUNDED PRECEDING`, ce qui est identique à `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. Avec `ORDER BY`, ceci configure le sous-ensemble pour contenir toutes les lignes de la partition à partir de la ligne courante. Sans `ORDER BY`, toutes les lignes de la partition sont incluses dans le sous-ensemble de la fenêtre, car toutes les lignes deviennent voisines de la ligne en cours.

Les restrictions sont que *`début_portée`* ne peut pas valoir `UNBOUNDED FOLLOWING`, *`fin_portée`* ne peut pas valoir `UNBOUNDED PRECEDING`, et le choix de *`fin_portée`* ne peut pas apparaître avant la liste ci-dessus des options *`début_portée`* et *`fin_portée`* que le choix de *`frame_start`* -- par exemple, `RANGE BETWEEN CURRENT ROW AND` *`valeur`* `PRECEDING` n'est pas autorisé. Par exemple, `ROWS BETWEEN 7 PRECEDING AND 8 PRECEDING` est autorisé, même s'il ne sélectionnera aucune ligne.

Si `FILTER` est indiqué, seules les lignes en entrée pour lesquelles *`clause_filtre`* est vrai sont envoyées à la fonction de fenêtrage. Les autres lignes sont simplement ignorées. Seules les fonctions de fenêtrage qui sont des agrégats acceptent une clause `FILTER`.

Les fonctions de fenêtrage internes sont décrites dans la Tableau 9.63. D'autres fonctions de fenêtrage peuvent être ajoutées par l'utilisateur. De plus, toute fonction d'agrégat de type général ou statistique peut être utilisée comme fonction de fenêtrage. Néanmoins, les agrégats d'ensemble trié et d'ensemble hypothétique ne peuvent pas être utilisés actuellement comme des fonctions de fenêtrage.

Les syntaxes utilisant * sont utilisées pour appeler des fonctions d'agrégats sans paramètres en tant que fonctions de fenêtrage. Par exemple : `count(*) OVER (PARTITION BY x ORDER BY y)`. Le symbole * n'est habituellement pas utilisé pour les fonctions de fenêtrage. Les fonctions de fenêtrage n'autorisent pas l'utilisation de `DISTINCT` ou `ORDER BY` dans la liste des arguments de la fonction.

Les appels de fonctions de fenêtrage sont autorisés seulement dans la liste `SELECT` et dans la clause `ORDER BY` de la requête.

Il existe plus d'informations sur les fonctions de fenêtrages dans la Section 3.5, dans la Section 9.22 et dans la Section 7.2.5.

4.2.9. Conversions de type

Une conversion de type spécifie une conversion à partir d'un type de données vers un autre. PostgreSQL accepte deux syntaxes équivalentes pour les conversions de type :

```
CAST ( expression AS type )
expression::type
```

La syntaxe `CAST` est conforme à SQL ; la syntaxe avec `::` est historique dans PostgreSQL.

Lorsqu'une conversion est appliquée à une expression de valeur pour un type connu, il représente une conversion de type à l'exécution. Cette conversion réussira seulement si une opération convenable de conversion de type a été définie. Notez que ceci est subtilement différent de l'utilisation de conversion avec des constantes, comme indiqué dans la Section 4.1.2.7. Une conversion appliquée à une chaîne constante représente l'affectation initiale d'un type pour une valeur constante, et donc cela réussira pour tout type (si le contenu de la chaîne constante est une syntaxe acceptée en entrée pour le type de donnée).

Une conversion de type explicite pourrait être habituellement omise s'il n'y a pas d'ambiguïté sur le type qu'une expression de valeur pourrait produire (par exemple, lorsqu'elle est affectée à une colonne de table) ; le système appliquera automatiquement une conversion de type dans de tels cas. Néanmoins, la conversion automatique est réalisée seulement pour les conversions marquées « OK pour application implicite » dans les catalogues système. D'autres conversions peuvent être appelées avec la syntaxe de conversion explicite. Cette restriction a pour but d'empêcher l'exécution silencieuse de conversions surprenantes.

Il est aussi possible de spécifier une conversion de type en utilisant une syntaxe de type fonction :

nom_type (*expression*)

Néanmoins, ceci fonctionne seulement pour les types dont les noms sont aussi valides en tant que noms de fonctions. Par exemple, `double precision` ne peut pas être utilisé de cette façon, mais son équivalent `float8` le peut. De même, les noms `interval`, `time` et `timestamp` peuvent seulement être utilisés de cette façon s'ils sont entre des guillemets doubles, à cause des conflits de syntaxe. Du coup, l'utilisation de la syntaxe de conversion du style fonction amène à des incohérences et devrait probablement être évitée.

> **Note**
>
> La syntaxe par fonction est en fait seulement un appel de fonction. Quand un des deux standards de syntaxe de conversion est utilisé pour faire une conversion à l'exécution, elle appellera en interne une fonction enregistrée pour réaliser la conversion. Par convention, ces fonctions de conversion ont le même nom que leur type de sortie et, du coup, la syntaxe par fonction n'est rien de plus qu'un appel direct à la fonction de conversion sous-jacente. Évidemment, une application portable ne devrait pas s'y fier. Pour plus d'informations, voir la page de manuel de CREATE CAST.

4.2.10. Expressions de collationnement

La clause `COLLATE` surcharge le collationnement d'une expression. Elle est ajoutée à l'expression à laquelle elle s'applique :

expr COLLATE *collationnement*

où *`collationnement`* est un identificateur pouvant être qualifié par son schéma. La clause `COLLATE` a priorité par rapport aux opérateurs ; des parenthèses peuvent être utilisées si nécessaire.

Si aucun collationnement n'est spécifiquement indiqué, le système de bases de données déduit cette information du collationnement des colonnes impliquées dans l'expression. Si aucune colonne ne se trouve dans l'expression, il utilise le collationnement par défaut de la base de données.

Les deux utilisations principales de la clause `COLLATE` sont la surcharge de l'ordre de tri dans une clause `ORDER BY`, par exemple :

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

et la surcharge du collationnement d'une fonction ou d'un opérateur qui produit un résultat sensible à la locale, par exemple :

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

Notez que, dans le dernier cas, la clause COLLATE est attachée à l'argument en entrée de l'opérateur. Peu importe l'argument de l'opérateur ou de la fonction qui a la clause COLLATE, parce que le collationnement appliqué à l'opérateur ou à la fonction est dérivé en considérant tous les arguments, et une clause COLLATE explicite surchargera les collationnements des autres arguments. (Attacher des clauses COLLATE différentes sur les arguments aboutit à une erreur. Pour plus de détails, voir la Section 24.2.) Du coup, ceci donne le même résultat que l'exemple précédent :

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

Mais ceci n'est pas valide :

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

car cette requête cherche à appliquer un collationnement au résultat de l'opérateur >, qui est du type boolean, type non sujet au collationnement.

4.2.11. Sous-requêtes scalaires

Une sous-requête scalaire est une requête SELECT ordinaire entre parenthèses renvoyant exactement une ligne avec une colonne (voir le Chapitre 7 pour plus d'informations sur l'écriture des requêtes). La requête SELECT est exécutée et la seule valeur renvoyée est utilisée dans l'expression de valeur englobante. C'est une erreur d'utiliser une requête qui renvoie plus d'une ligne ou plus d'une colonne comme requête scalaire. Mais si, lors d'une exécution particulière, la sous-requête ne renvoie pas de lignes, alors il n'y a pas d'erreur ; le résultat scalaire est supposé NULL. La sous-requête peut référencer des variables de la requête englobante, qui agiront comme des constantes durant toute évaluation de la sous-requête. Voir aussi la Section 9.23 pour d'autres expressions impliquant des sous-requêtes.

Par exemple, ce qui suit trouve la ville disposant de la population la plus importante dans chaque état :

```
SELECT nom, (SELECT max(pop) FROM villes WHERE villes.etat =
 etat.nom)
    FROM etats;
```

4.2.12. Constructeurs de tableaux

Un constructeur de tableau est une expression qui construit une valeur de tableau à partir de la valeur de ses membres. Un constructeur de tableau simple utilise le mot-clé ARRAY, un crochet ouvrant [, une liste d'expressions (séparées par des virgules) pour les valeurs des éléments du tableau et finalement un crochet fermant]. Par exemple :

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

Par défaut, le type d'élément du tableau est le type commun des expressions des membres, déterminé en utilisant les mêmes règles que pour les constructions UNION ou CASE (voir la Section 10.5). Vous pouvez surcharger ceci en convertissant explicitement le constructeur de tableau vers le type désiré. Par exemple :

```
SELECT ARRAY[1,2,22.7]::integer[];
```

```
  array
----------
 {1,2,23}
(1 row)
```

Ceci a le même effet que la conversion de chaque expression vers le type d'élément du tableau individuellement. Pour plus d'informations sur les conversions, voir la Section 4.2.9.

Les valeurs de tableaux multidimensionnels peuvent être construites par des constructeurs de tableaux imbriqués. Pour les constructeurs internes, le mot-clé ARRAY peut être omis. Par exemple, ces expressions produisent le même résultat :

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

Comme les tableaux multidimensionnels doivent être rectangulaires, les constructeurs internes du même niveau doivent produire des sous-tableaux de dimensions identiques. Toute conversion appliquée au constructeur ARRAY externe se propage automatiquement à tous les constructeurs internes.

Les éléments d'un constructeur de tableau multidimensionnel peuvent être tout ce qui récupère un tableau du bon type, pas seulement une construction d'un tableau imbriqué. Par exemple :

```
CREATE TABLE tab(f1 int[], f2 int[]);

INSERT INTO tab VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM tab;
                     array
-----------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

Vous pouvez construire un tableau vide, mais comme il est impossible d'avoir un tableau sans type, vous devez convertir explicitement votre tableau vide dans le type désiré. Par exemple :

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

Il est aussi possible de construire un tableau à partir des résultats d'une sous-requête. Avec cette forme, le constructeur de tableau est écrit avec le mot-clé ARRAY suivi par une sous-requête entre parenthèses (et non pas des crochets). Par exemple :

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                               array
-----------------------------------------------------------------
```

```
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412}

SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS
 a(i));
              array
----------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

La sous-requête doit renvoyer une seule colonne. Si la sortie de la sous- requête n'est pas de type tableau, le tableau à une dimension résultant aura un élément pour chaque ligne dans le résultat de la sous-requête, avec un type élément correspondant à celui de la colonne en sortie de la sous- requête. Si la colonne en sortie de la sous-requête est de type tableau, le résultat sera un tableau du même type, mais avec une dimension supplémentaire ; dans ce cas, toutes les lignes de la sous-requête doivent renvoyer des tableaux de dimension identique (dans le cas contraire, le résultat ne serait pas rectangulaire).

Les indices d'un tableau construit avec ARRAY commencent toujours à un. Pour plus d'informations sur les tableaux, voir la Section 8.15.

4.2.13. Constructeurs de lignes

Un constructeur de ligne est une expression qui construit une valeur de ligne (aussi appelée une valeur composite) à partir des valeurs de ses membres. Un constructeur de ligne consiste en un mot-clé ROW, une parenthèse gauche, zéro ou une ou plus d'une expression (séparées par des virgules) pour les valeurs des champs de la ligne, et enfin une parenthèse droite. Par exemple :

```
SELECT ROW(1,2.5,'ceci est un test');
```

Le mot-clé ROW est optionnel lorsqu'il y a plus d'une expression dans la liste.

Un constructeur de ligne peut inclure la syntaxe *valeurligne*.*, qui sera étendue en une liste d'éléments de la valeur ligne, ce qui est le comportement habituel de la syntaxe .* utilisée au niveau haut d'une liste SELECT (voir Section 8.16.5). Par exemple, si la table t a les colonnes f1 et f2, ces deux requêtes sont identiques :

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

> **Note**
>
> Avant PostgreSQL 8.2, la syntaxe .* n'était pas étendue dans les constructeurs de lignes. De ce fait, ROW(t.*, 42) créait une ligne à deux champs dont le premier était une autre valeur de ligne. Le nouveau comportement est généralement plus utile. Si vous avez besoin de l'ancien comportement de valeurs de ligne imbriquées, écrivez la valeur de ligne interne sans .*, par exemple ROW(t, 42).

Par défaut, la valeur créée par une expression ROW est d'un type d'enregistrement anonyme. Si nécessaire, il peut être converti en un type composite nommé -- soit le type de ligne d'une table, soit un type composite créé avec CREATE TYPE AS. Une conversion explicite pourrait être nécessaire pour éviter toute ambiguïté. Par exemple :

```
CREATE TABLE ma_table(f1 int, f2 float, f3 text);

CREATE FUNCTION recup_f1(ma_table) RETURNS int AS 'SELECT $1.f1'
 LANGUAGE SQL;
```

```
-- Aucune conversion nécessaire parce que seul un recup_f1() existe
SELECT recup_f1(ROW(1,2.5,'ceci est un test'));
 recup_f1
----------
1
(1 row)

CREATE TYPE mon_typeligne AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION recup_f1(mon_typeligne) RETURNS int AS 'SELECT
 $1.f1' LANGUAGE SQL;

-- Maintenant, nous avons besoin d'une conversion
-- pour indiquer la fonction à appeler
SELECT recup_f1(ROW(1,2.5,'ceci est un test'));
ERROR:  function recup_f1(record) is not unique

SELECT recup_f1(ROW(1,2.5,'ceci est un test')::ma_table);
 getf1
-------
 1
(1 row)

SELECT recup_f1(CAST(ROW(11,'ceci est un test',2.5) AS
 mon_typeligne));
 getf1
-------
 11
(1 row)
```

Les constructeurs de lignes peuvent être utilisés pour construire des valeurs composites à stocker dans une colonne de table de type composite ou pour être passés à une fonction qui accepte un paramètre composite. De plus, il est possible de comparer deux valeurs de lignes ou de tester une ligne avec `IS NULL` ou `IS NOT NULL`, par exemple

```
SELECT ROW(1,2.5,'ceci est un test') = ROW(1, 3, 'pas le même');

SELECT ROW(table.*) IS NULL FROM table; -- détecte toutes les
 lignes non NULL
```

Pour plus de détails, voir la Section 9.24. Les constructeurs de lignes peuvent aussi être utilisés en relation avec des sous-requêtes, comme discuté dans la Section 9.23.

4.2.14. Règles d'évaluation des expressions

L'ordre d'évaluation des sous-expressions n'est pas défini. En particulier, les entrées d'un opérateur ou d'une fonction ne sont pas obligatoirement évaluées de la gauche vers la droite ou dans un autre ordre fixé.

De plus, si le résultat d'une expression peut être déterminé par l'évaluation de certaines parties de celle-ci, alors d'autres sous-expressions devraient ne pas être évaluées du tout. Par exemple, si vous écrivez :

```
SELECT true OR une_fonction();
```

alors `une_fonction()` pourrait (probablement) ne pas être appelée du tout. Pareil dans le cas suivant :

```
SELECT une_fonction() OR true;
```

Notez que ceci n'est pas identique au « court-circuitage » de gauche à droite des opérateurs booléens utilisé par certains langages de programmation.

En conséquence, il est déconseillé d'utiliser des fonctions ayant des effets de bord dans une partie des expressions complexes. Il est particulièrement dangereux de se fier aux effets de bord ou à l'ordre d'évaluation dans les clauses `WHERE` et `HAVING`, car ces clauses sont reproduites de nombreuses fois lors du développement du plan d'exécution. Les expressions booléennes (combinaisons `AND`/`OR`/`NOT`) dans ces clauses pourraient être réorganisées d'une autre façon autorisée dans l'algèbre booléenne.

Quand il est essentiel de forcer l'ordre d'évaluation, une construction `CASE` (voir la Section 9.18) peut être utilisée. Voici un exemple qui ne garantit pas qu'une division par zéro ne soit pas faite dans une clause `WHERE` :

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

Mais ceci est sûr :

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

Une construction `CASE` utilisée de cette façon déjouera les tentatives d'optimisation, donc cela ne sera à faire que si c'est nécessaire (dans cet exemple particulier, il serait sans doute mieux de contourner le problème en écrivant `y > 1.5*x`).

Néanmoins, `CASE` n'est pas un remède à tout. Une limitation à la technique illustrée ci-dessus est qu'elle n'empêche pas l'évaluation en avance des sous-expressions constantes. Comme décrit dans Section 38.7, les fonctions et les opérateurs marqués `IMMUTABLE` peuvent être évalués quand la requête est planifiée plutôt que quand elle est exécutée. Donc, par exemple :

```
 SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

va produire comme résultat un échec pour division par zéro, car le planificateur a essayé de simplifier la sous-expression constante, même si chaque ligne de la table a `x > 0` de façon à ce que la condition `ELSE` ne soit jamais exécutée.

Bien que cet exemple particulier puisse sembler stupide, il existe de nombreux cas moins évidents, n'impliquant pas de constantes, mais plutôt des requêtes exécutées par des fonctions, quand les valeurs des arguments des fonctions et de variables locales peuvent être insérées dans les requêtes en tant que constantes toujours dans le but de la planification. À l'intérieur de fonctions PL/pgSQL, par exemple, utiliser une instruction `IF-THEN- ELSE` pour protéger un calcul risqué est beaucoup plus sûr qu'une expression `CASE`.

Une autre limitation de cette technique est qu'une expression `CASE` ne peut pas empêcher l'évaluation d'une expression d'agrégat contenue dans cette expression, car les expressions d'agrégat sont calculées avant les expressions « scalaires » dans une liste `SELECT` ou dans une clause `HAVING`. Par exemple, la requête suivante peut provoquer une erreur de division par zéro bien qu'elle semble protégée contre ce type d'erreurs :

```
SELECT CASE WHEN min(employees) > 0
            THEN avg(expenses / employees)
       END
    FROM departments;
```

Les agrégats `min()` et `avg()` sont calculés en même temps avec toutes les lignes en entrée, donc si une ligne a une valeur 0 pour la colonne `employees`, l'erreur de division par zéro surviendra avant d'avoir pu tester le résultat de `min()`. Il est préférable d'utiliser une clause `WHERE` ou une clause `FILTER` pour empêcher les lignes problématiques en entrée d'atteindre la fonction d'agrégat.

4.3. Fonctions appelantes

PostgreSQL permet aux fonctions qui ont des paramètres nommés d'être appelées en utilisant soit la notation par *position* soit la notation par *nom*. La notation par nom est particulièrement utile pour les fonctions qui ont un grand nombre de paramètres, car elle rend l'association entre paramètre et argument plus explicite et fiable. Dans la notation par position, un appel de fonction précise les valeurs en argument dans le même ordre que ce qui a été défini à la création de la fonction. Dans la notation nommée, les arguments sont précisés par leur nom et peuvent du coup être intégrés dans n'importe quel ordre. Pour chaque notation, considérez aussi l'effet des types d'argument de la fonction, documenté dans Section 10.3.

Quel que soit la notation, les paramètres qui ont des valeurs par défaut dans leur déclaration n'ont pas besoin d'être précisés dans l'appel. Ceci est particulièrement utile dans la notation nommée, car toute combinaison de paramètre peut être omise, alors que dans la notation par position, les paramètres peuvent seulement être omis de la droite vers la gauche.

PostgreSQL supporte aussi la notation *mixée*. Elle combine la notation par position avec la notation par nom. Dans ce cas, les paramètres de position sont écrits en premier, les paramètres nommés apparaissent après.

Les exemples suivants illustrent l'utilisation des trois notations, en utilisant la définition de fonction suivante :

```
CREATE FUNCTION assemble_min_ou_maj(a text, b text, majuscule
 boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

La fonction `assemble_min_ou_maj` a deux paramètres obligatoires, `a` et `b`. Il existe en plus un paramètre optionnel, `majuscule`, qui vaut par défaut `false`. Les arguments `a` et `b` seront concaténés et forcés soit en majuscule soit en minuscule suivant la valeur du paramètre `majuscule`. Les détails restants ne sont pas importants ici (voir le Chapitre 38 pour plus d'informations).

4.3.1. En utilisant la notation par position

La notation par position est le mécanisme traditionnel pour passer des arguments aux fonctions avec PostgreSQL. En voici un exemple :

```
SELECT assemble_min_ou_maj('Hello', 'World', true);
 assemble_min_ou_maj
---------------------
 HELLO WORLD
(1 row)
```

Tous les arguments sont indiqués dans l'ordre. Le résultat est en majuscule, car l'argument `majuscule` est indiqué à `true`. Voici un autre exemple :

```
SELECT assemble_min_ou_maj('Hello', 'World');
 assemble_min_ou_maj
----------------------
 hello world
(1 row)
```

Ici, le paramètre `majuscule` est omis, donc il récupère la valeur par défaut, soit `false`, ce qui a pour résultat une sortie en minuscule. Dans la notation par position, les arguments peuvent être omis de la droite à la gauche à partir du moment où ils ont des valeurs par défaut.

4.3.2. En utilisant la notation par nom

Dans la notation par nom, chaque nom d'argument est précisé en utilisant `=>` pour le séparer de l'expression de la valeur de l'argument. Par exemple :

```
SELECT assemble_min_ou_maj(a => 'Hello', b => 'World');
 assemble_min_ou_maj
---------------------
 hello world
(1 row)
```

Encore une fois, l'argument `majuscule` a été omis, donc il dispose de sa valeur par défaut, `false`, implicitement. Un avantage à utiliser la notation par nom est que les arguments peuvent être saisis dans n'importe quel ordre. Par exemple :

```
SELECT assemble_min_ou_maj(a => 'Hello', b => 'World', uppercase =>
 true);
 assemble_min_ou_maj
---------------------
 HELLO WORLD
(1 row)

SELECT assemble_min_ou_maj(a => 'Hello', uppercase => true, b =>
 'World');
 assemble_min_ou_maj
---------------------
 HELLO WORLD
(1 row)
```

Une syntaxe plus ancienne basée sur « := » est supportée pour des raisons de compatibilité ascendante :

```
SELECT assemble_min_ou_maj(a := 'Hello', uppercase := true, b :=
 'World');
 assemble_min_ou_maj
---------------------
 HELLO WORLD
(1 row)
```

4.3.3. En utilisant la notation mixée

La notation mixée combine les notations par position et par nom. Néanmoins, comme cela a déjà été expliqué, les arguments par nom ne peuvent pas précéder les arguments par position. Par exemple :

```
SELECT assemble_min_ou_maj('Hello', 'World', majuscule => true);
 assemble_min_ou_maj
----------------------
 HELLO WORLD
(1 row)
```

Dans la requête ci-dessus, les arguments `a` et `b` sont précisés par leur position alors que `majuscule` est indiqué par son nom. Dans cet exemple, cela n'apporte pas grand-chose, sauf pour une documentation de la fonction. Avec une fonction plus complexe, comprenant de nombreux paramètres avec des valeurs par défaut, les notations par nom et mixées améliorent l'écriture des appels de fonction et permettent de réduire les risques d'erreurs.

> **Note**
>
> Les notations par appel nommé ou mixe ne peuvent pas être utilisées lors de l'appel d'une fonction d'agrégat (mais elles fonctionnent quand une fonction d'agrégat est utilisée en tant que fonction de fenêtrage).

Chapitre 5. Définition des données

Ce chapitre couvre la création des structures de données amenées à contenir les données. Dans une base relationnelle, les données brutes sont stockées dans des tables. De ce fait, une grande partie de ce chapitre est consacrée à des explications sur la création et la modification des tables et aux fonctionnalités disponibles pour contrôler les données stockées dans les tables. L'organisation des tables dans des schémas et l'attribution de droits sur les tables sont ensuite décrites. Pour finir, d'autres fonctionnalités, telles que l'héritage, le partitionnement de tables, les vues, les fonctions et les triggers sont passés en revue.

5.1. Notions fondamentales sur les tables

Une table dans une base relationnelle ressemble beaucoup à un tableau sur papier : elle est constituée de lignes et de colonnes. Le nombre et l'ordre des colonnes sont fixes et chaque colonne a un nom. Le nombre de lignes est variable -- il représente le nombre de données stockées à un instant donné. Le SQL n'apporte aucune garantit sur l'ordre des lignes dans une table. Quand une table est lue, les lignes apparaissent dans un ordre non spécifié, sauf si un tri est demandé explicitement. Tout cela est expliqué dans le Chapitre 7. De plus, le SQL n'attribue pas d'identifiant unique aux lignes. Il est donc possible d'avoir plusieurs lignes identiques au sein d'une table. C'est une conséquence du modèle mathématique sur lequel repose le SQL, même si cela n'est habituellement pas souhaitable. Il est expliqué plus bas dans ce chapitre comment traiter ce problème.

Chaque colonne a un type de données. Ce type limite l'ensemble de valeurs qu'il est possible d'attribuer à une colonne. Il attribue également une sémantique aux données stockées dans la colonne pour permettre les calculs sur celles-ci. Par exemple, une colonne déclarée dans un type numérique n'accepte pas les chaînes textuelles ; les données stockées dans une telle colonne peuvent être utilisées dans des calculs mathématiques. Par opposition, une colonne déclarée de type chaîne de caractères accepte pratiquement n'importe quel type de données, mais ne se prête pas aux calculs mathématiques. D'autres types d'opérations, telle la concaténation de chaînes, sont cependant disponibles.

PostgreSQL inclut un ensemble important de types de données intégrés pour s'adapter à diverses applications. Les utilisateurs peuvent aussi définir leurs propres types de données.

La plupart des types de données intégrés ont des noms et des sémantiques évidents. C'est pourquoi leur explication détaillée est reportée au Chapitre 8.

Parmi les types les plus utilisés, on trouve `integer` pour les entiers, `numeric` pour les éventuelles fractions, `text` pour les chaînes de caractères, `date` pour les dates, `time` pour les heures et `timestamp` pour les valeurs qui contiennent à la fois une date et une heure.

Pour créer une table, on utilise la commande nommée CREATE TABLE. Dans cette commande, il est nécessaire d'indiquer, au minimum, le nom de la table, les noms des colonnes et le type de données de chacune d'elles. Par exemple :

```
CREATE TABLE ma_premiere_table (
    premiere_colonne text,
    deuxieme_colonne integer
);
```

Cela crée une table nommée `ma_premiere_table` avec deux colonnes. La première colonne, nommée `premiere_colonne`, est de type `text` ; la seconde colonne, nommée `deuxieme_colonne`, est de type `integer`. Les noms des tables et des colonnes se conforment à la syntaxe des identifiants expliquée dans la Section 4.1.1. Les noms des types sont souvent aussi des identifiants, mais il existe des exceptions. Le séparateur de la liste des colonnes est la virgule. La liste doit être entre parenthèses.

L'exemple qui précède est à l'évidence extrêmement simpliste. On donne habituellement aux tables et aux colonnes des noms qui indiquent les données stockées. L'exemple ci-dessous est un peu plus réaliste :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric
);
```

(Le type `numeric` peut stocker des fractions telles que les montants.)

> **Astuce**
>
> Quand de nombreuses tables liées sont créées, il est préférable de définir un motif cohérent pour le nommage des tables et des colonnes. On a ainsi la possibilité d'utiliser le pluriel ou le singulier des noms, chacune ayant ses fidèles et ses détracteurs.

Le nombre de colonnes d'une table est limité. En fonction du type de données des colonnes, il oscille entre 250 et 1600. Définir une table avec un nombre de colonnes proche de cette limite est, cependant, très inhabituel et doit conduire à se poser des questions quant à la conception du modèle.

Lorsqu'une table n'est plus utile, elle peut être supprimée à l'aide de la commande DROP TABLE. Par exemple :

```
DROP TABLE ma_premiere_table;
DROP TABLE produits;
```

Tenter de supprimer une table qui n'existe pas lève une erreur. Il est néanmoins habituel, dans les fichiers de scripts SQL, d'essayer de supprimer chaque table avant de la créer. Les messages d'erreur sont alors ignorés afin que le script fonctionne, que la table existe ou non. (La variante `DROP TABLE IF EXISTS` peut aussi être utilisée pour éviter les messages d'erreur, mais elle ne fait pas partie du standard SQL.)

Pour la procédure de modification d'une table qui existe déjà, voir la Section 5.6 plus loin dans ce chapitre.

Les outils précédemment décrits permettent de créer des tables fonctionnelles. Le reste de ce chapitre est consacré à l'ajout de fonctionnalités à la définition de tables pour garantir l'intégrité des données, la sécurité ou l'ergonomie. Le lecteur impatient d'insérer des données dans ses tables peut sauter au Chapitre 6 et lire le reste de ce chapitre plus tard.

5.2. Valeurs par défaut

Une valeur par défaut peut être attribuée à une colonne. Quand une nouvelle ligne est créée et qu'aucune valeur n'est indiquée pour certaines de ses colonnes, celles-ci sont remplies avec leurs valeurs par défaut respectives. Une commande de manipulation de données peut aussi demander explicitement que la valeur d'une colonne soit positionnée à la valeur par défaut, sans qu'il lui soit nécessaire de connaître cette valeur (les détails concernant les commandes de manipulation de données sont donnés dans le Chapitre 6).

Si aucune valeur par défaut n'est déclarée explicitement, la valeur par défaut est la valeur NULL. Cela a un sens dans la mesure où l'on peut considérer que la valeur NULL représente des données inconnues.

Dans la définition d'une table, les valeurs par défaut sont listées après le type de données de la colonne. Par exemple :

```
CREATE TABLE produits (
```

```
    no_produit integer,
    nom text,
    prix numeric DEFAULT 9.99
);
```

La valeur par défaut peut être une expression, alors évaluée à l'insertion de cette valeur (*pas* à la création de la table). Un exemple commun est la colonne de type `timestamp` dont la valeur par défaut est `now()`. Elle se voit ainsi attribuer l'heure d'insertion. Un autre exemple est la génération d'un « numéro de série » pour chaque ligne. Dans PostgreSQL, cela s'obtient habituellement par quelque chose comme

```
CREATE TABLE produits (
    no_produit integer DEFAULT nextval('produits_no_produit_seq'),
    ...
);
```

où la fonction `nextval()` fournit des valeurs successives à partir d'un *objet séquence* (voir la Section 9.17). Cet arrangement est suffisamment commun pour qu'il ait son propre raccourci :

```
CREATE TABLE produits (
    no_produit SERIAL,
    ...
);
```

Le raccourci `SERIAL` est discuté plus tard dans la Section 8.1.4.

5.3. Colonnes générées

Une colonne générée est une colonne spéciale, toujours calculée à partir d'autres colonnes. Elle est donc aux colonnes ce qu'une vue est aux tables. Il y a deux types de colonnes générées : stockée et virtuelle. Une colonne générée stockée est calculée quand elle est écrite (insérée ou mise à jour) et occupe de l'espace de stockage comme si elle était une colonne normale. Une colonne virtuelle générée n'occupe pas d'espace et est calculée à la lecture. Une colonne générée virtuelle est donc équivalente à une vue, et une colonne générée stockée est équivalente à une vue matérialisée (sauf qu'elle sera toujours mise à jour automatiquement). PostgreSQL n'implémente actuellement que les colonnes générées stockées.

Pour créer une colonne générée, utilisez la clause `GENERATED ALWAYS AS` de la commande `CREATE TABLE`, par exemple :

```
CREATE TABLE people (
    ...,
    height_cm numeric,
    height_in numeric GENERATED ALWAYS AS (height_cm / 2.54) STORED
);
```

Le mot-clé `STORED` doit être spécifié pour choisir le type de colonne générée. Voir CREATE TABLE pour plus de détails.

On ne peut écrire directement dans une colonne générée. Une valeur ne peut pas y être affectée dans les commandes `INSERT` ou `UPDATE`, mais le mot-clé `DEFAULT` peut l'être.

Voyons les différences entre une colonne avec une valeur par défaut et une colonne générée. La colonne par défaut est calculée une seule fois à la première insertion de la ligne si aucune autre valeur n'est fournie ; une colonne générée est mise à jour à chaque fois que la ligne change et on ne peut y déroger. Une valeur par défaut d'une colonne ne peut se référer à d'autres colonnes de la table ; mais c'est ce que fait normalement une expression générée. Une valeur par défaut d'une colonne peut utiliser des fonctions volatiles, par exemple `random()` ou des fonctions se référant au temps actuel ; ce n'est pas permis pour les colonnes générées.

Il existe plusieurs restrictions dans la définition des colonnes générées et des tables qui les utilisent :

- L'expression pour générer les valeurs ne peut utiliser que des fonctions immutables, ne peut pas utiliser de sous-requêtes, ni référencer d'aucune manière quoi que ce soit hors de la ligne en cours.
- Une expression ne peut référencer une autre colonne générée.
- Une expression ne peut référencer une colonne système, sauf `tableoid`.
- Une colonne générée ne peut avoir une valeur par défaut ou être définie comme colonne identité.
- Une colonne générée ne peut faire partie d'une clé de partitionnement.
- Les tables distantes peuvent porter des colonnes générées. Voir CREATE FOREIGN TABLE pour les détails.
- Pour l'héritage :
  - Si une colonne parent est une colonne générée, une colonne enfant doit aussi être une colonne générée utilisant la même expression. Dans la définition de la colonne enfant, laissez la clause `GENERATED` car elle sera copiée du parent.
  - Dans le cas d'héritage multiple, si une colonne parent est une colonne générée, alors toutes les colonnes parents doivent être des colonnes générées et avec la même expression.
  - Si une colonne parent n'est pas une colonne générée, une colonne enfant pourrait être définie comme colonne générée ou pas.

D'autres considérations s'appliquent à l'utilisation des colonnes générées.

- Les colonnes générées maintiennent les droits d'accès séparément des colonnes sur lesquelles elles sont basées. On peut donc s'arranger pour qu'un rôle défini puisse lire une colonne générée mais pas la colonne de base sous-jacente.
- Conceptuellement, les colonnes générées sont mises à jour après le déclenchement des triggers `BEFORE`. Les changements dans les colonnes de base au sein d'un trigger `BEFORE` seront donc répercutés dans les colonnes générées. Mais à l'inverse, il n'est pas permis d'accéder aux colonnes générées dans les triggers `BEFORE`.

5.4. Contraintes

Les types de données sont un moyen de restreindre la nature des données qui peuvent être stockées dans une table. Pour beaucoup d'applications, toutefois, la contrainte fournie par ce biais est trop grossière. Par exemple, une colonne qui contient le prix d'un produit ne doit accepter que des valeurs positives. Mais il n'existe pas de type de données natif qui n'accepte que des valeurs positives. Un autre problème peut provenir de la volonté de contraindre les données d'une colonne par rapport aux autres colonnes ou lignes. Par exemple, dans une table contenant des informations de produit, il ne peut y avoir qu'une ligne par numéro de produit.

Pour cela, SQL permet de définir des contraintes sur les colonnes et les tables. Les contraintes donnent autant de contrôle sur les données des tables qu'un utilisateur peut le souhaiter. Si un utilisateur tente de stocker des données dans une colonne en violation d'une contrainte, une erreur est levée. Cela s'applique même si la valeur vient de la définition de la valeur par défaut.

5.4.1. Contraintes de vérification

La contrainte de vérification est la contrainte la plus générique qui soit. Elle permet d'indiquer que la valeur d'une colonne particulière doit satisfaire une expression booléenne (valeur de vérité). Par exemple, pour obliger les prix des produits à être positifs, on peut utiliser :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric CHECK (prix > 0)
);
```

La définition d'une contrainte vient après le type de données, comme pour les définitions de valeur par défaut. Les valeurs par défaut et les contraintes peuvent être données dans n'importe quel ordre. Une contrainte de vérification s'utilise avec le mot-clé `CHECK` suivi d'une expression entre parenthèses. L'expression de la contrainte implique habituellement la colonne à laquelle elle s'applique, la contrainte n'ayant dans le cas contraire que peu de sens.

La contrainte peut prendre un nom distinct. Cela clarifie les messages d'erreur et permet de faire référence à la contrainte lorsqu'elle doit être modifiée. La syntaxe est :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric CONSTRAINT prix_positif CHECK (prix > 0)
);
```

Pour indiquer une contrainte nommée, on utilise le mot-clé `CONSTRAINT` suivi d'un identifiant et de la définition de la contrainte (si aucun nom n'est précisé, le système en choisit un).

Une contrainte de vérification peut aussi faire référence à plusieurs colonnes. Dans le cas d'un produit, on peut vouloir stocker le prix normal et un prix réduit en s'assurant que le prix réduit soit bien inférieur au prix normal.

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric CHECK (prix > 0),
    prix_promotion numeric CHECK (prix_promotion > 0),
    CHECK (prix > prix_promotion)
);
```

Si les deux premières contraintes n'offrent pas de nouveauté, la troisième utilise une nouvelle syntaxe. Elle n'est pas attachée à une colonne particulière, mais apparaît comme un élément distinct dans la liste des colonnes. Les définitions de colonnes et ces définitions de contraintes peuvent être définies dans un ordre quelconque.

Les deux premières contraintes sont appelées contraintes de colonne, tandis que la troisième est appelée contrainte de table parce qu'elle est écrite séparément d'une définition de colonne particulière. Les contraintes de colonne peuvent être écrites comme des contraintes de table, mais l'inverse n'est pas forcément possible puisqu'une contrainte de colonne est supposée ne faire référence qu'à la colonne à laquelle elle est attachée (PostgreSQL ne vérifie pas cette règle, mais il est préférable de la suivre pour s'assurer que les définitions de tables fonctionnent avec d'autres systèmes de bases de données). L'exemple ci-dessus peut aussi s'écrire :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric,
    CHECK (prix > 0),
    prix_promotion numeric,
    CHECK (prix_promotion > 0),
    CHECK (prix > prix_promotion)
);
```

ou même :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric CHECK (prix > 0),
    prix_promotion numeric,
    CHECK (prix_promotion > 0 AND prix > prix_promotion)
);
```

C'est une question de goût.

Les contraintes de table peuvent être nommées, tout comme les contraintes de colonne :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric,
    CHECK (prix > 0),
    prix_promotion numeric,
    CHECK (prix_promotion > 0),
    CONSTRAINT promo_valide CHECK (prix > prix_promotion)
);
```

Une contrainte de vérification est satisfaite si l'expression est évaluée vraie ou NULL. Puisque la plupart des expressions sont évaluées NULL si l'un des opérandes vaut NULL, elles n'interdisent pas les valeurs NULL dans les colonnes contraintes. Pour s'assurer qu'une colonne ne contient pas de valeurs NULL, la contrainte NOT NULL décrite dans la section suivante peut être utilisée.

> **Note**
>
> PostgreSQL ne gère pas les contraintes `CHECK` référençant des données situées ailleurs que dans la ligne vérifiée, nouvelle ou mise à jour. Bien qu'une contrainte `CHECK` violant cette règle puisse sembler fonctionner dans des tests simples, on ne peut garantir que la base de données n'atteindra pas un état dans lequel la condition de contrainte serait fausse (suite à des changements sur d'autres lignes impliquées). Cela entraînerait l'échec d'une restauration de sauvegarde logique. Elle échouerait même si l'état de la base entière était cohérent avec la contrainte, à cause des lignes non encore chargées dans un ordre qui satisferait la contrainte. Si possible, utilisez les contraintes `UNIQUE`, `EXCLUDE` ou `FOREIGN KEY` pour exprimer des restrictions entre lignes ou entre tables.
>
> Si vous recherchez une vérification uniquement à l'insertion de la ligne par rapport à d'autres lignes, et non une garantie de cohérence à maintenir en permanence, vous pouvez utiliser un trigger personnalisé. (Cette approche évite le problème de la restauration logique car pg_dump ne réinstalle pas les triggers avant d'avoir fini de recharger les données ; ainsi la vérification ne sera pas appliquée à la restauration.)

> **Note**
>
> PostgreSQL suppose que les conditions des contraintes `CHECK` sont immutables, c'est-à-dire qu'elles donneront toujours le même résultat pour la même ligne en entrée. Ainsi on s'autorise à n'examiner les contraintes `CHECK` qu'à l'insertion ou la suppression des lignes, et pas à d'autres moments. (L'avertissement ci-dessus sur la référence aux données d'autres tables n'est qu'un cas particulier de cette restriction.)
>
> Un exemple d'une manière courante de passer outre cette supposition est de faire référence à une fonction utilisateur dans l'expression d'un `CHECK`, puis de changer le comportement de cette fonction. PostgreSQL ne l'interdit pas, mais il ne remarquera pas qu'il y a des lignes dans

la table qui violent à présent la contrainte CHECK. Cela provoquerait l'échec d'une sauvegarde/restauration logique subséquente. La manière recommandée pour traiter un tel changement est de supprimer la contrainte (avec ALTER TABLE), d'ajuster la définition de la fonction et de ré-appliquer la contrainte, la revalidant ainsi sur toutes les lignes de la table.

5.4.2. Contraintes de non-nullité (NOT NULL)

Une contrainte NOT NULL indique simplement qu'une colonne ne peut pas prendre la valeur NULL. Par exemple :

```
CREATE TABLE produits (
    no_produit integer NOT NULL,
    nom text NOT NULL,
    prix numeric
);
```

Une contrainte NOT NULL est toujours écrite comme une contrainte de colonne. Elle est fonctionnellement équivalente à la création d'une contrainte de vérification CHECK (*nom_colonne* IS NOT NULL). Toutefois, dans PostgreSQL, il est plus efficace de créer explicitement une contrainte NOT NULL. L'inconvénient est que les contraintes de non-nullité ainsi créées ne peuvent pas être explicitement nommées.

Une colonne peut évidemment avoir plusieurs contraintes. Il suffit d'écrire les contraintes les unes après les autres :

```
CREATE TABLE produits (
    no_produit integer NOT NULL,
    nom text NOT NULL,
    prix numeric NOT NULL CHECK (prix > 0)
);
```

L'ordre n'a aucune importance. Il ne détermine pas l'ordre de vérification des contraintes.

La contrainte NOT NULL a un contraire ; la contrainte NULL. Elle ne signifie pas que la colonne doit être NULL, ce qui est assurément inutile, mais sélectionne le comportement par défaut, à savoir que la colonne peut être NULL. La contrainte NULL n'est pas présente dans le standard SQL et ne doit pas être utilisée dans des applications portables(elle n'a été ajoutée dans PostgreSQL que pour assurer la compatibilité avec d'autres bases de données). Certains utilisateurs l'apprécient néanmoins, car elle permet de basculer aisément d'une contrainte à l'autre dans un fichier de script. On peut, par exemple, commencer avec :

```
CREATE TABLE produits (
    no_produit integer NULL,
    nom text NULL,
    prix numeric NULL
);
```

puis insérer le mot-clé NOT en fonction des besoins.

Astuce

Dans la plupart des bases de données, il est préférable que la majorité des colonnes soient marquées NOT NULL.

5.4.3. Contraintes d'unicité

Les contraintes d'unicité garantissent l'unicité des données contenues dans une colonne ou un groupe de colonnes par rapport à toutes les lignes de la table. La syntaxe est :

```
CREATE TABLE produits (
    no_produit integer UNIQUE,
    nom text,
    prix numeric
);
```

lorsque la contrainte est écrite comme contrainte de colonne et :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric,
    UNIQUE (no_produit)
);
```

lorsqu'elle est écrite comme contrainte de table.

Pour définir une contrainte d'unicité pour un groupe de colonnes, saisissez-la en tant que contrainte de table avec les noms des colonnes séparés par des virgules :

```
CREATE TABLE exemple (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

Cela précise que la combinaison de valeurs dans les colonnes indiquées est unique sur toute la table. Sur une colonne prise isolément, ce n'est pas nécessairement le cas (et habituellement, cela ne l'est pas).

Une contrainte d'unicité peut être nommée, de la façon habituelle :

```
CREATE TABLE produits (
    no_produit integer CONSTRAINT doit_etre_different UNIQUE,
    nom text,
    prix numeric
);
```

Ajouter une contrainte d'unicité va automatiquement créer un index unique B-tree sur la colonne ou le groupe de colonnes listées dans la contrainte. Une restriction d'unicité couvrant seulement certaines lignes ne peut pas être écrite comme une contrainte d'unicité, mais il est possible de forcer ce type de restriction en créant un index partiel unique.

En général, une contrainte d'unicité est violée si plus d'une ligne de la table possède des valeurs identiques sur toutes les colonnes de la contrainte. Par défaut, deux valeurs NULL ne sont jamais considérées égales. Cela signifie qu'il est possible de stocker des lignes dupliquées contenant une valeur NULL dans au moins une des colonnes contraintes. Ce comportement peut être modifié en ajoutant la clause `NULLS NOT DISTINCT`, comme :

```
CREATE TABLE products (
    product_no integer UNIQUE NULLS NOT DISTINCT,
    name text,
    price numeric
);
```

ou :

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE NULLS NOT DISTINCT (product_no)
);
```

Le comportement par défaut peut être spécifié explicitement en utilisant `NULLS DISTINCT`. Le traitement par défaut des NULL dans les contraintes d'unicité est défini par l'implémentation suivant le standard SQL, alors que d'autres implémentations ont un comportement différent. Donc faites attention quand vous développez des applications qui ont pour but d'être portables.

5.4.4. Clés primaires

Une contrainte de type clé primaire indique qu'une colonne, ou un groupe de colonnes, peuvent être utilisés comme un identifiant unique de ligne pour cette table. Ceci nécessite que les valeurs soient à la fois uniques et non NULL. Les définitions de table suivantes acceptent de ce fait les mêmes données :

```
CREATE TABLE produits (
 no_produit integer UNIQUE NOT NULL,
 nom text,
 prix numeric
);
```

```
CREATE TABLE produits (
 no_produit integer PRIMARY KEY,
 nom text,
 prix numeric
);
```

Les clés primaires peuvent également contraindre plusieurs colonnes ; la syntaxe est semblable aux contraintes d'unicité :

```
CREATE TABLE exemple (
 a integer,
 b integer,
 c integer,
 PRIMARY KEY (a, c)
);
```

Ajouter une clé primaire créera automatiquement un index unique B-tree sur la colonne ou le groupe de colonnes listés dans la clé primaire, et forcera les colonnes à être marquées `NOT NULL`.

L'ajout d'une clé primaire créera automatiquement un index B-tree unique sur la colonne ou le groupe de colonnes utilisé dans la clé primaire.

Une table a, au plus, une clé primaire. (Le nombre de contraintes UNIQUE NOT NULL, qui assurent pratiquement la même fonction, n'est pas limité, mais une seule peut être identifiée comme clé primaire.) La théorie des bases de données relationnelles impose que chaque table ait une clé primaire. Cette règle n'est pas forcée par PostgreSQL, mais il est préférable de la respecter.

Les clés primaires sont utiles pour la documentation et pour les applications clientes. Par exemple, une application graphique qui permet de modifier des valeurs des lignes a probablement besoin de connaître la clé primaire d'une table pour être capable d'identifier les lignes de façon unique. Le système de bases de données utilise une clé primaire de différentes façons. Par exemple, la clé primaire définit les colonnes cibles par défaut pour les clés étrangères référençant cette table.

5.4.5. Clés étrangères

Une contrainte de clé étrangère stipule que les valeurs d'une colonne (ou d'un groupe de colonnes) doivent correspondre aux valeurs qui apparaissent dans les lignes d'une autre table. On dit que cela maintient l'*intégrité référentielle* entre les deux tables.

Soit la table de produits, déjà utilisée plusieurs fois :

```
CREATE TABLE produits (
 no_produit integer PRIMARY KEY,
 nom text,
 prix numeric
);
```

Soit également une table qui stocke les commandes de ces produits. Il est intéressant de s'assurer que la table des commandes ne contient que des commandes de produits qui existent réellement. Pour cela, une contrainte de clé étrangère est définie dans la table des commandes qui référence la table « produits » :

```
CREATE TABLE commandes (
 id_commande integer PRIMARY KEY,
 no_produit integer REFERENCES produits (no_produit),
 quantite integer
);
```

Il est désormais impossible de créer des commandes pour lesquelles les valeurs non NULL de `no_produit` n'apparaissent pas dans la table « produits ».

Dans cette situation, on dit que la table des commandes est la table *qui référence* et la table des produits est la table *référencée*. De la même façon, il y a des colonnes qui référencent et des colonnes référencées.

La commande précédente peut être raccourcie en

```
CREATE TABLE commandes (
 id_commande integer PRIMARY KEY,
 no_produit integer REFERENCES produits,
 quantite integer
);
```

parce qu'en l'absence de liste de colonnes, la clé primaire de la table de référence est utilisée comme colonne de référence.

Une contrainte de clé étrangère peut être nommée de la façon habituelle.

Une clé étrangère peut aussi contraindre et référencer un groupe de colonnes. Comme cela a déjà été évoqué, il faut alors l'écrire sous forme d'une contrainte de table. Exemple de syntaxe :

```
CREATE TABLE t1 (
      a integer PRIMARY KEY,
      b integer,
      c integer,
      FOREIGN KEY (b, c) REFERENCES autre_table (c1, c2)
);
```

Le nombre et le type des colonnes contraintes doivent correspondre au nombre et au type des colonnes référencées.

Parfois, il est utile que l'« autre table » d'une clé étrangère soit la même table ; elle est alors appelée une clé étrangère *auto-référencée*. Par exemple, si vous voulez que les lignes d'une table représentent les nœuds d'une structure en arbre, vous pouvez écrire

```
CREATE TABLE tree (
    node_id integer PRIMARY KEY,
    parent_id integer REFERENCES tree,
    name text,
    ...
);
```

Un nœud racine aura la colonne `parent_id` à NULL, et les enregistrements non NULL de `parent_id` seront contraints de référencer des enregistrements valides de la table.

Une table peut contenir plusieurs contraintes de clé étrangère. Les relations n-n entre tables sont implémentées ainsi. Soit des tables qui contiennent des produits et des commandes, avec la possibilité d'autoriser une commande à contenir plusieurs produits (ce que la structure ci-dessus ne permet pas). On peut pour cela utiliser la structure de table suivante :

```
CREATE TABLE produits (
 no_produit integer PRIMARY KEY,
 nom text,
 prix numeric
);

CREATE TABLE commandes (
 id_commande integer PRIMARY KEY,
 adresse_de_livraison text,
 ...
);

CREATE TABLE commande_produits (
 no_produit integer REFERENCES produits,
 id_commande integer REFERENCES commandes,
 quantite integer,
 PRIMARY KEY (no_produit, id_commande)
);
```

La clé primaire de la dernière table recouvre les clés étrangères.

Les clés étrangères interdisent désormais la création de commandes qui ne sont pas liées à un produit. Qu'arrive-t-il si un produit est supprimé alors qu'une commande y fait référence ? SQL permet aussi de le gérer. Intuitivement, plusieurs options existent :

- interdire d'effacer un produit référencé ;
- effacer aussi les commandes ;
- autre chose ?

Pour illustrer ce cas, la politique suivante est implémentée sur l'exemple de relations n-n évoqué plus haut :

- quand quelqu'un veut retirer un produit qui est encore référencé par une commande(au travers de `commande_produits`), on l'interdit ;
- si quelqu'un supprime une commande, les éléments de la commande sont aussi supprimés.

```
CREATE TABLE produits (
 no_produit integer PRIMARY KEY,
 nom text,
 prix numeric
);

CREATE TABLE commandes (
```

```
 id_commande integer PRIMARY KEY,
 adresse_de_livraison text,
 ...
);

CREATE TABLE commande_produits (
 no_produit integer REFERENCES produits ON DELETE RESTRICT,
 id_commande integer REFERENCES commandes ON DELETE CASCADE,
 quantite integer,
 PRIMARY KEY (no_produit, id_commande)
);
```

Restreindre les suppressions et les réaliser en cascade sont les deux options les plus communes. `RESTRICT` empêche la suppression d'une ligne référencée. `NO ACTION` impose la levée d'une erreur si des lignes référençant existent lors de la vérification de la contrainte. Il s'agit du comportement par défaut en l'absence de précision. La différence entre `RESTRICT` et `NO ACTION` est l'autorisation par `NO ACTION` du report de la vérification à la fin de la transaction, ce que `RESTRICT` ne permet pas. `CASCADE` indique que, lors de la suppression d'une ligne référencée, les lignes la référençant doivent être automatiquement supprimées. Il existe deux autres options : `SET NULL` et `SET DEFAULT`. Celles-ci imposent que les colonnes qui référencent dans les lignes référencées soient réinitialisées à NULL ou à leur valeur par défaut, respectivement, lors de la suppression d'une ligne référencée. Elles ne dispensent pas pour autant d'observer les contraintes. Par exemple, si une action précise `SET DEFAULT`, mais que la valeur par défaut ne satisfait pas la clé étrangère, l'opération échoue.

Le choix approprié de l'action `ON DELETE` dépend du type d'objets que les tables ciblées représentent. Quand la table référençante représente quelque chose qui est un composant de ce qui est représenté par la table référencée et ne peut pas exister indépendamment, alors `CASCADE` pourrait être approprié. Si les deux tables représentent des objets indépendants, alors `RESTRICT` ou `NO ACTION` est plus approprié ; une application qui souhaite réellement supprimer les deux objets devra être explicite là-dessus et exécuter les deux commandes de suppression. Dans l'exemple ci-dessus, les éléments d'une commande font partie de cette commande, et il est pratique qu'ils soient supprimés automatiquement si une commande est supprimée. Mais les produits et les commandes sont deux choses différentes, et de ce fait, faire en sorte que supprimer un produit cause la suppression automatique de certains éléments de commandes peut être considéré comme problématique. Les actions `SET NULL` ou `SET DEFAULT` peuvent être appropriées si la relation d'une clé étrangère représente des informations supplémentaires. Par exemple, si la table produits contenait une référence à un gestionnaire de produit et que l'enregistrement correspondant à ce gestionnaire était supprimé, alors configurer le gestionnaire du produit à NULL ou à une valeur par défaut pourrait être utile.

Les actions `SET NULL` et `SET DEFAULT` peuvent prendre une liste de colonnes pour spécifier les colonnes à configurer. Normalement, toutes les colonnes de la clé étrangère sont configurés ; configurer uniquement un sous-ensemble est utile dans quelques cas spéciaux. Considérez l'exemple suivant :

```
CREATE TABLE tenants (
    tenant_id integer PRIMARY KEY
);

CREATE TABLE users (
    tenant_id integer REFERENCES tenants ON DELETE CASCADE,
    user_id integer NOT NULL,
    PRIMARY KEY (tenant_id, user_id)
);

CREATE TABLE posts (
    tenant_id integer REFERENCES tenants ON DELETE CASCADE,
    post_id integer NOT NULL,
    author_id integer,
```

```
    PRIMARY KEY (tenant_id, post_id),
    FOREIGN KEY (tenant_id, author_id) REFERENCES users ON DELETE
 SET NULL (author_id)
);
```

Sans la spécification de la colonne, la clé étrangère aurait aussi configurer la colonne `tenant_id` à null, mais la valeur de cette colonne est toujours essentielle, vu qu'elle fait partie de la clé primaire.

À l'instar de `ON DELETE`, il existe `ON UPDATE`, évoqué lorsqu'une colonne référencée est modifiée (actualisée). Les actions possibles sont les mêmes, sauf que les listes de colonnes ne peuvent pas être utilisées avec `SET NULL` et `SET DEFAULT`. Dans ce cas, `CASCADE` signifie que les valeurs mises à jour dans la colonne référencée doivent être copiées dans les lignes de référence.

Habituellement, une ligne de référence n'a pas besoin de satisfaire la clé étrangère si une de ses colonnes est NULL. Si la clause `MATCH FULL` est ajoutée à la déclaration de la clé étrangère, une ligne de référence échappe à la clé étrangère seulement si toutes ses colonnes de référence sont NULL (donc un mélange de valeurs NULL et non NULL échoue forcément sur une contrainte `MATCH FULL`). Si vous ne voulez pas que les lignes de référence soient capables d'empêcher la satisfaction de la clé étrangère, déclarez les colonnes de référence comme `NOT NULL`.

Une clé étrangère doit référencer les colonnes qui soit sont une clé primaire, soit forment une contrainte d'unicité, soit sont les colonnes d'un index d'unicité non partiel. Cela signifie que les colonnes référencées ont toujours un index pour permettre des recherches efficaces pour savoir si une ligne référençante a une correspondance. Comme la suppression d'une ligne de la table référencée ou la mise à jour d'une colonne référencée nécessitera un parcours de la table référée pour trouver les lignes correspondant à l'ancienne valeur, il est souvent intéressant d'indexer les colonnes référencées. Comme cela n'est pas toujours nécessaire et qu'il y a du choix sur la façon d'indexer, l'ajout d'une contrainte de clé étrangère ne crée pas automatiquement un index sur les colonnes référencées.

Le Chapitre 6 contient de plus amples informations sur l'actualisation et la suppression de données. Voir aussi la description de la syntaxe des clés étrangères dans la documentation de référence sur CREATE TABLE.

Une clé étrangère peut faire référence à des colonnes qui constituent une clé primaire ou forment une contrainte d'unicité. Si la clé étrangère référence une contrainte d'unicité, des possibilités supplémentaires sont offertes concernant la correspondance des valeurs NULL. Celles-ci sont expliquées dans la documentation de référence de CREATE TABLE.

5.4.6. Contraintes d'exclusion

Les contraintes d'exclusion vous assurent que si deux lignes sont comparées sur les colonnes ou expressions spécifiées en utilisant les opérateurs indiqués, au moins une de ces comparaisons d'opérateurs renverra false ou NULL. La syntaxe est :

```
CREATE TABLE cercles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

Voir aussi `CREATE TABLE ... CONSTRAINT ... EXCLUDE` pour plus de détails.

L'ajout d'une contrainte d'exclusion créera automatiquement un index du type spécifié dans la déclaration de la contrainte.

5.5. Colonnes système

Chaque table contient plusieurs *colonnes système* implicitement définies par le système. De ce fait, leurs noms ne peuvent pas être utilisés comme noms de colonnes utilisateur (ces restrictions sont

distinctes de celles sur l'utilisation de mot-clés ; mettre le nom entre guillemets ne permet pas d'échapper à cette règle). Il n'est pas vraiment utile de se préoccuper de ces colonnes, mais au minimum de savoir qu'elles existent.

`tableoid`

L'OID de la table contenant la ligne. Cette colonne est particulièrement utile pour les requêtes qui utilisent des tables partitionnées (voir Section 5.11) ou des hiérarchies d'héritage (voir Section 5.10). En effet, il est difficile, en son absence, de savoir de quelle table provient une ligne. `tableoid` peut être joint à la colonne `oid` de `pg_class` pour obtenir le nom de la table.

`xmin`

L'identifiant (ID) de la transaction qui a inséré cette version de la ligne. (Une version de ligne est un état individuel de la ligne ; toute mise à jour d'une ligne crée une nouvelle version de ligne pour la même ligne logique.)

`cmin`

L'identifiant de commande (à partir de zéro) au sein de la transaction d'insertion.

`xmax`

L'identifiant (ID) de la transaction de suppression, ou zéro pour une version de ligne non effacée. Il est possible que la colonne ne soit pas nulle pour une version de ligne visible ; cela indique habituellement que la transaction de suppression n'a pas été effectuée, ou qu'une tentative de suppression a été annulée.

`cmax`

L'identifiant de commande au sein de la transaction de suppression, ou zéro.

`ctid`

La localisation physique de la version de ligne au sein de sa table. Bien que le `ctid` puisse être utilisé pour trouver la version de ligne très rapidement, le `ctid` d'une ligne change si la ligne est actualisée ou déplacée par un `VACUUM FULL`. `ctid` est donc inutilisable comme identifiant de ligne sur le long terme. Il est préférable d'utiliser la clé primaire pour identifier les lignes logiques.

Les identifiants de transaction sont des nombres sur 32 bits. Dans une base de données âgée, il est possible que les identifiants de transaction bouclent. Cela n'est pas un problème fatal avec des procédures de maintenance appropriées ; voir le Chapitre 25 pour les détails. Il est, en revanche, imprudent de considérer l'unicité des identifiants de transaction sur le long terme (plus d'un milliard de transactions).

Les identifiants de commande sont aussi des nombres sur 32 bits. Cela crée une limite dure de 2^{32} (4 milliards) commandes SQL au sein d'une unique transaction. En pratique, cette limite n'est pas un problème -- la limite est sur le nombre de commandes SQL, pas sur le nombre de lignes traitées. De plus, seules les commandes qui modifient réellement le contenu de la base de données consomment un identifiant de commande.

5.6. Modification des tables

Lorsqu'une table est créée et qu'une erreur a été commise ou que les besoins de l'application changent, il est alors possible de la supprimer et de la récréer. Cela n'est toutefois pas pratique si la table contient déjà des données ou qu'elle est référencée par d'autres objets de la base de données (une contrainte de clé étrangère, par exemple). C'est pourquoi PostgreSQL offre une série de commandes permettant de modifier une table existante. Cela n'a rien à voir avec la modification des données contenues dans la table ; il ne s'agit ici que de modifier la définition, ou structure, de la table.

Il est possible :

- d'ajouter des colonnes ;
- de supprimer des colonnes ;
- d'ajouter des contraintes ;
- de supprimer des contraintes ;
- de modifier des valeurs par défaut ;
- de modifier les types de données des colonnes ;
- de renommer des colonnes ;
- de renommer des tables.

Toutes ces actions sont réalisées à l'aide de la commande ALTER TABLE, dont la page de référence est bien plus détaillée.

5.6.1. Ajouter une colonne

La commande d'ajout d'une colonne ressemble à :

```
ALTER TABLE produits ADD COLUMN description text;
```

La nouvelle colonne est initialement remplie avec la valeur par défaut précisée (NULL en l'absence de clause `DEFAULT`).

> **Astuce**
>
> À partir de PostgreSQL 11, ajouter une colonne avec une valeur par défaut constante ne signifie plus que chaque ligne de la table doit être mise à jour quand l'instruction `ALTER TABLE` doit être exécutée. À la place, la valeur par défaut sera renvoyée à chaque accès à la ligne et appliquée quand la table est réécrite, rendant ainsi la commande `ALTER TABLE` bien plus rapide, même sur des tables volumineuses.
>
> Néanmoins, si la valeur par défaut est volatile (par exemple `clock_timestamp()`), chaque ligne devra être mise à jour avec la valeur calculée à l'exécution du `ALTER TABLE`. Pour éviter une opération de mise à jour potentiellement longue, et en particulier si vous avez de toute façon l'intention de remplir la colonne avec des valeurs qui ne sont pas par défaut, il pourrait être préférable d'ajouter la colonne sans valeur par défaut, d'insérer les valeurs correctes en utilisant l'instruction `UPDATE`, et enfin d'ajouter la valeur par désirée comme décrit ci-dessous.

Des contraintes de colonne peuvent être définies dans la même commande, à l'aide de la syntaxe habituelle :

```
ALTER TABLE produits ADD COLUMN description text CHECK (description
 <> '');
```

En fait, toutes les options applicables à la description d'une colonne dans `CREATE TABLE` peuvent être utilisées ici. Il ne faut toutefois pas oublier que la valeur par défaut doit satisfaire les contraintes données. Dans le cas contraire, `ADD` échoue. Il est aussi possible d'ajouter les contraintes ultérieurement (voir ci-dessous) après avoir rempli la nouvelle colonne correctement.

5.6.2. Supprimer une colonne

La commande de suppression d'une colonne ressemble à celle-ci :

```
ALTER TABLE produits DROP COLUMN description;
```

Toute donnée dans cette colonne disparaît. Les contraintes de table impliquant la colonne sont également supprimées. Néanmoins, si la colonne est référencée par une contrainte de clé étrangère

d'une autre table, PostgreSQL ne supprime pas silencieusement cette contrainte. La suppression de tout ce qui dépend de la colonne peut être autorisée en ajoutant CASCADE :

```
ALTER TABLE produits DROP COLUMN description CASCADE;
```

Voir la Section 5.14 pour une description du mécanisme général.

> **Note**
>
> PostgreSQL ne supporte pas les contraintes CHECK qui référencent les données d'autres tables que celle contenant la nouvelle ligne ou la ligne mise à jour en cours de vérification. Alors qu'une contrainte CHECK qui viole cette règle pourrait apparaitre fonctionner dans des tests simples, il est possible que la base de données atteigne un état dans lequel la condition de la contrainte est fausse (à cause de changements supplémentaires en dehors de la ligne impliquée). Ceci sera la cause d'un échec du rechargement de la sauvegarde d'une base. La restauration pourrait échouer même quand l'état complet de la base est cohérent avec la contrainte, à cause de lignes chargées dans un autre différent qui satisferait la contrainte. Si possible, utilisez les contraintes UNIQUE, EXCLUDE, et FOREIGN KEY pour exprimer des restrictions inter-lignes et inter-tables.
>
> Si ce que vous désirez est une vérification unique avec certaines lignes au moment de l'insertion, plutôt qu'une garantie de cohérence maintenue en permanence, un trigger personnalisé peut être utilisé pour l'implémenter. (Cette approche évite le problème de sauvegarde/restauration car pg_dump ne réinstalle les triggers qu'après chargement des données, donc cette vérification ne sera pas effectuée pendant une sauvegarde/restauration.)

> **Note**
>
> PostgreSQL suppose que les conditions des contraintes CHECK sont immutables, c'est-à-dire qu'elles donneront toujours le même résultat pour la même ligne en entrée. Cette supposition est ce qui justifie l'examen des contraintes CHECK uniquement quand les lignes sont insérées ou mises à jour, et non pas à d'autres moments. (Cet avertissement sur la non référence aux données d'autres tables est en fait un cas particulier de cette restriction.)
>
> Un exemple d'une façon habituelle de casser cette supposition est de référencer une fonction utilisateur dans une expression CHECK, puis de changer le comportement de cette fonction. PostgreSQL n'interdit pas cela, mais il ne notera pas qu'il y a des lignes dans la table qui violent maintenant la contrainte CHECK. Ceci sera la cause d'un échec de la restauration d'une sauvegarde de cette base. La façon recommandée de gérer de tels changements revient à supprimer la contrainte (en utilisant ALTER TABLE), d'ajuster la définition de la fonction, et d'ajouter de nouveau la contrainte, ce qui causera une nouvelle vérification des lignes de la table.

5.6.3. Ajouter une contrainte

Pour ajouter une contrainte, la syntaxe de contrainte de table est utilisée. Par exemple :

```
ALTER TABLE produits ADD CHECK (nom <> '');
ALTER TABLE produits ADD CONSTRAINT autre_nom UNIQUE (no_produit);
ALTER TABLE produits ADD FOREIGN KEY (id_groupe_produit) REFERENCES
 groupes_produits;
```

Pour ajouter une contrainte NOT NULL, qui ne peut pas être écrite sous forme d'une contrainte de table, la syntaxe suivante est utilisée :

```
ALTER TABLE produits ALTER COLUMN no_produit SET NOT NULL;
```

La contrainte étant immédiatement vérifiée, les données de la table doivent satisfaire la contrainte avant qu'elle ne soit ajoutée.

5.6.4. Supprimer une contrainte

Pour supprimer une contrainte, il faut connaître son nom. Si elle a été explicitement nommée, il n'y a aucune difficulté. Dans le cas contraire, le système a affecté un nom généré qu'il faudra identifier. La commande `\d` *`table`* de psql peut être utile ici ; d'autres interfaces offrent aussi la possibilité d'examiner les détails de table. La commande est :

```
ALTER TABLE produits DROP CONSTRAINT un_nom;
```

Comme pour la suppression d'une colonne, `CASCADE` peut être ajouté pour supprimer une contrainte dont dépendent d'autres objets. Une contrainte de clé étrangère, par exemple, dépend d'une contrainte de clé primaire ou d'unicité sur la(les) colonne(s) référencée(s).

Cela fonctionne de la même manière pour tous les types de contraintes, à l'exception des contraintes NOT NULL. Pour supprimer une contrainte NOT NULL, on écrit :

```
ALTER TABLE produits ALTER COLUMN no_produit DROP NOT NULL;
```

(Les contraintes NOT NULL n'ont pas de noms.)

5.6.5. Modifier la valeur par défaut d'une colonne

La commande de définition d'une nouvelle valeur par défaut de colonne ressemble à celle-ci :

```
ALTER TABLE produits ALTER COLUMN prix SET DEFAULT 7.77;
```

Cela n'affecte pas les lignes existantes de la table, mais uniquement la valeur par défaut pour les futures commandes `INSERT`.

Pour retirer toute valeur par défaut, on écrit :

```
ALTER TABLE produits ALTER COLUMN prix DROP DEFAULT;
```

C'est équivalent à mettre la valeur par défaut à NULL. En conséquence, il n'y a pas d'erreur à retirer une valeur par défaut qui n'a pas été définie car NULL est la valeur par défaut implicite.

5.6.6. Modifier le type de données d'une colonne

La commande de conversion du type de données d'une colonne ressemble à celle-ci :

```
ALTER TABLE produits ALTER COLUMN prix TYPE numeric(10,2);
```

Elle ne peut réussir que si chaque valeur de la colonne peut être convertie dans le nouveau type par une conversion implicite. Si une conversion plus complexe est nécessaire, une clause `USING` peut être ajoutée qui indique comment calculer les nouvelles valeurs à partir des anciennes.

PostgreSQL tente de convertir la valeur par défaut de la colonne le cas échéant, ainsi que toute contrainte impliquant la colonne. Mais ces conversions peuvent échouer ou produire des résultats surprenants. Il est souvent préférable de supprimer les contraintes de la colonne avant d'en modifier le type, puis d'ajouter ensuite les contraintes convenablement modifiées.

5.6.7. Renommer une colonne

Pour renommer une colonne :

```
ALTER TABLE produits RENAME COLUMN no_produit TO numero_produit;
```

5.6.8. Renommer une table

Pour renommer une table :

```
ALTER TABLE produits RENAME TO elements;
```

5.7. Droits

Quand un objet est créé, il se voit affecter un propriétaire. Le propriétaire est normalement le rôle qui a exécuté la requête de création. Pour la plupart des objets, l'état initial est que seul le propriétaire et les superutilisateurs peuvent faire quelque chose avec cet objet. Pour permettre aux autres rôles de l'utiliser, des *droits* doivent être donnés.

Il existe un certain nombre de droits différents : `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES`, `TRIGGER`, `CREATE`, `CONNECT`, `TEMPORARY`, `EXECUTE`, `USAGE`, `SET` et `ALTER SYSTEM`. Les droits applicables à un objet particulier varient selon le type d'objet (table, fonction...). Plus de détails sur la signification de ces droits sont donnés ci-dessous. La section et les chapitres suivants présentent l'utilisation de ces droits.

Le droit de modifier ou détruire un objet est inhérent au propriétaire de l'objet et ne peut ni être donné ni être enlevé. Néanmoins, comme tous les droits, il peut être hérité pour les membres du rôle propriétaire ; voir Section 22.3.)

Un objet peut se voir affecter un nouveau propriétaire avec la commande `ALTER` correspondant à l'objet, par exemple

```
ALTER TABLE nom_table OWNER TO nouveau_propriétaire;
```

Les superutilisateurs peuvent toujours le faire. Les rôles ordinaires ne peuvent le faire que s'ils sont le propriétaire actuel de l'objet (ou un membre du rôle propriétaire) et un membre du nouveau rôle propriétaire. Tous les droits sur les objets de l'ancien propriétaire sont transférés au nouveau propriétaire avec l'appartenance.

La commande GRANT est utilisée pour accorder des droits. Par exemple, si `joe` est un rôle et `comptes` une table, le droit de modifier la table `comptes` peut être accordé à `joe` avec :

```
GRANT UPDATE ON comptes TO joe;
```

Écrire `ALL` à la place d'un droit spécifique accorde tous les droits applicables à ce type d'objet.

Le nom de « rôle » spécial `PUBLIC` peut être utilisé pour donner un droit à tous les rôles du système. De plus, les rôles de type « group » peuvent être configurés pour aider à la gestion des droits quand il y a beaucoup d'utilisateurs dans une base -- pour les détails, voir Chapitre 22.

Pour révoquer un droit précédemment donné, on utilise la commande nommée REVOKE, comme dans l'exemple ci-dessous :

```
REVOKE ALL ON comptes FROM PUBLIC;
```

Habituellement, seul le propriétaire de l'objet ou un superutilisateur peuvent accorder ou révoquer les droits sur un objet. Néanmoins, il est possible de donner un droit « avec possibilité de transmission »(*« with grant option »*), qui donne à celui qui le reçoit le droit de donner ce droit à d'autres. Si cette option est ensuite révoquée, alors tous ceux qui ont reçu ce droit par cet utilisateur(directement ou indirectement via la chaîne des dons) perdent ce droit. Pour les détails, voir les pages de références GRANT et REVOKE.

Le propriétaire d'un objet peut décider de révoquer ses propres droits standards. Par exemple, il peut rendre une table en lecture seule pour eux comme pour les autres. Mais les propriétaires sont toujours traités comme ayant le droit de donner des droits, pour qu'ils puissent toujours retrouver leur anciens droits.

Les droits disponibles sont :

`SELECT`

Autorise `SELECT` de n'importe quelle colonne, ou colonnes, désignée(s) d'une table, vue, vue matérialisée ou tout autre objet utilisable comme une table. Permet aussi l'utilisation de `COPY TO`. Ce droit est aussi nécessaire pour référencer les valeurs actuelles d'une colonne dans `UPDATE`, `DELETE` ou `MERGE`. Pour les séquences, ce droit permet aussi d'utiliser la fonction `currval`. Pour les « Large Objects », ce droit permet de lire l'objet binaire.

`INSERT`

Permet l'`INSERT` d'une nouvelle ligne dans une table, vue, etc. Peut être accordé sur des colonnes spécifiques, auquel cas seules ces colonnes pourront être affectées dans l'ordre `INSERT` (les autres commandes recevront alors les valeurs par défaut). Permet aussi d'utiliser `COPY FROM`.

`UPDATE`

Autorise l'`UPDATE` de n'importe quelle colonne, ou colonnes, désignée(s) d'une table, vue, etc. (En pratique, toute commande `UPDATE` non triviale requiert en plus le droit `SELECT` puisqu'il faut se référer aux colonnes de la table pour déterminer les lignes à mettre à jour et/ou calculer les nouvelles valeurs des colonnes.) `SELECT ... FOR UPDATE` et `SELECT ... FOR SHARE` requièrent aussi ce droit sur au moins une colonne, en plus du droit `SELECT`. Pour les séquences, ce droit autorise l'usage des fonctions `nextval` et `setval`. Pour les « Large Objects », ce droit permet d'écrire dans l'objet ou de le tronquer.

`DELETE`

Permet le `DELETE` d'une ligne dans une table, vue, etc. (En pratique, toute commande `DELETE` non triviale nécessite aussi le droit `SELECT` puisqu'il faut lire les colonnes de la table pour déterminer les lignes à supprimer.)

`TRUNCATE`

Permet `TRUNCATE` sur une table.

`REFERENCES`

Permet la création de clés étrangères référençant une table, ou des colonnes spécifiques d'une table. Il faut faire particulièrement attention à donner ce droit car un utilisateur qui crée une clé étrangère peut forcer l'appel d'une fonction arbitraire, comme une fonction de conversion, et de telles fonctions seront appelées avec les droits du propriétaire de la table.

`TRIGGER`

Permet la création d'un trigger sur une table, vue, etc. Il faut faire particulièrement attention à donner ce droit car tout trigger ajouté à une table ou à une vue sera ajouté avec les droits des utilisateurs qui la modifient.

`CREATE`

Pour les bases de données, autorise la création de nouveaux schémas et publications dans la base, et autorise l'installation d'extensions de confiance dans la base de données.

Pour les schémas, autorise la création de nouveaux objets dans le schéma. Pour renommer un objet existant, vous devez posséder l'objet *et* posséder ce droit pour le schéma de l'objet.

Pour les tablespaces, permet de créer des tables, index et fichiers temporaires dans le tablespace, et de créer des bases de données ayant ce tablespace comme tablespace par défaut.

Notez que révoquer ce droit ne modifiera pas l'emplacement des objets.

`CONNECT`

Permet au bénéficiaire de ce droit de se connecter à la base de données. Ce droit est vérifié au démarrage de la connexion(en plus de la vérification de toute restriction imposée par le fichier `pg_hba.conf`).

`TEMPORARY`

Permet aux tables temporaires d'être créées dans la base de données.

`EXECUTE`

Permet d'appeler une fonction ou procédure, y compris en utilisant des opérateurs implémentés par-dessus la fonction. C'est le seul type de droit applicable aux fonctions et procédures.

`USAGE`

Pour les langages procéduraux, permet d'utiliser le langage pour la création de fonctions. C'est le seul type de droit applicable aux langages procéduraux.

Pour les schémas, permet l'accès aux objets contenus dans le schéma (à supposer que les droit d'accès requis par les objets soient aussi respectés). Cela autorise essentiellement le bénéficiaire à « rechercher » des objets dans le schéma. Sans ce droit, il reste possible de voir les noms des objets, par exemple en consultant les catalogues système. Après révocation de ce droit, les sessions existantes peuvent avoir des ordres ayant précédemment effectué cette recherche, donc ce n'est pas une manière totalement sûre d'interdire l'accès à un objet.

Pour les séquences, permet l'utilisation des fonctions `currval` et `nextval`.

Pour les types et domaines, permet l'utilisation du type ou domaine dans la création de tables, fonctions et autres objets. (Notez que ce droit ne contrôle pas l'« utilisation » de ce type, comme les valeurs apparaissant dans les requêtes. Il se limite à interdire la création d'objets qui dépendent de ce type. Le but principal de ce droit est de contrôler quels utilisateurs peuvent créer des dépendances envers un type, qui pourraient empêcher le propriétaire de modifier le type plus tard.)

Pour les foreign data wrappers, permet la création de nouveaux serveurs avec ce wrapper.

Pour les serveurs distants (*foreign servers*), permet la création de tables distantes utilisant le serveur. Les bénéficiaires du droit peuvent créer, modifier, supprimer leurs propres correspondances d'utilisateurs (*user mappings*) associées à ce serveur.

`SET`

Permet la configuration d'un paramètre à une nouvelle valeur dans la session en cours. (Bien que ce droit puisse être donné pour tout paramètre, ce ne serait pas sensé sauf pour les paramètres qui nécessitent l'attribut `SUPERUSER` pour le configurer.)

`ALTER SYSTEM`

Permet la configuration d'un paramètre du serveur avec une nouvelle valeur en utilisant la commande ALTER SYSTEM.

Les droits requis par d'autres commandes sont listés sur la page de référence de la commande.

Par défaut, PostgreSQL accorde des droits sur certains types d'objets à `PUBLIC` dès la création des objets. Aucun droit n'est accordé à `PUBLIC` par défaut sur les tables, colonnes de table, séquences, foreign data wrappers, serveurs distants, large objects, schémas, tablespaces et paramètres de configuration. Pour les autres types d'objets, les droits par défaut accordés à `PUBLIC` sont les suivants : sur les bases de données : les droits `CONNECT` et `TEMPORARY`(création de table temporaire) ; sur les fonctions et procédures : le droit `EXECUTE` ; et sur les langages et types de données (y compris les domaines) : le droit `USAGE`. Bien sûr, le propriétaire de l'objet peut révoquer, à l'aide de

la commande REVOKE, les droits par défaut comme ceux expressément accordés. (Pour une sécurité maximale, ordonnez REVOKE dans la même transaction que celle qui crée l'objet ; il n'y a alors aucune fenêtre où un autre utilisateur peut utiliser l'objet.) Ces droits par défaut peuvent aussi être remplacés avec la commande ALTER DEFAULT PRIVILEGES.

Tableau 5.1 liste les abréviations à une lettre utilisées pour les types de droits dans les valeurs des *ACL* (listes de contrôle d'accès). Vous verrez ces lettres dans la sortie des commandes psql plus bas, ou en consultant les colonnes ACL des catalogues système.

Tableau 5.1. Abréviations des droits dans les ACL

Droit	Abréviation	Types d'objets concernés
SELECT	r (« read »)	LARGE OBJECT, SEQUENCE, TABLE (et objets similaires à des tables), colonne de table
INSERT	a (« append »)	TABLE, colonne de table
UPDATE	w (« write »)	LARGE OBJECT, SEQUENCE, TABLE, colonne de table
DELETE	d	TABLE
TRUNCATE	D	TABLE
REFERENCES	x	TABLE, colonne de table
TRIGGER	t	TABLE
CREATE	C	DATABASE, SCHEMA, TABLESPACE
CONNECT	c	DATABASE
TEMPORARY	T	DATABASE
EXECUTE	X	FUNCTION, PROCEDURE
USAGE	U	DOMAIN, FOREIGN DATA WRAPPER, FOREIGN SERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE
SET	s	PARAMETER
ALTER SYSTEM	A	PARAMETER

Tableau 5.2 résume les droits disponibles pour chaque objet SQL, avec les abréviations ci-dessus. Il affiche aussi la commande psql à utiliser pour consulter les valeurs des droits de chaque type d'objet.

Tableau 5.2. Résumé des droits d'accès

Type d'objet	Tous les droits	Droits par défaut pour PUBLIC	Commande psql
DATABASE	CTc	Tc	\l
DOMAIN	U	U	\dD+
FUNCTION ou PROCEDURE	X	X	\df+
FOREIGN DATA WRAPPER	U	aucun	\dew+
FOREIGN SERVER	U	aucun	\des+
LANGUAGE	U	U	\dL+
LARGE OBJECT	rw	aucun	\dl+
PARAMETER	sA	aucun	\dconfig+
SCHEMA	UC	aucun	\dn+
SEQUENCE	rwU	aucun	\dp
TABLE (et objets similaires)	arwdDxt	aucun	\dp

Type d'objet	Tous les droits	Droits par défaut pour PUBLIC	Commande psql
Colonne de table	arwx	aucun	\dp
TABLESPACE	C	aucun	\db+
TYPE	U	U	\dT+

Les droits accordés à un objet particulier sont affichés comme une liste d'entrées aclitem, chacune ayant le format :

grantee=privilege-abbreviation[*]*.../grantor*

Chaque aclitem liste tous les droits d'un rôle qui ont été donnés par un rôle particulier. Les droits spécifiques sont représentés par des abréviations sur une lettre à partir de Tableau 5.1, avec un * ajouté si le droit a été donné avec l'option grant. Par exemple, calvin=r*w/hobbes indique que le rôle calvin a le droit SELECT (r) avec possibilité de transmission (*« with grant option »*) ainsi que le droit UPDATE (w), sans retransmission possible ; droits accordés tous les deux par le rôle hobbes. Si calvin a aussi des droits sur le même objet accordés par un autre utilisateur, ils apparaîtraient comme une entrée aclitem séparée. Un champ vide dans un aclitem indique PUBLIC.

Par exemple, supposons que l'utilisateur miriam crée une table mytable et fasse :

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam_rw;
```

alors dans psql, la commande \dp afficherait :

```
=> \dp mytable
                                        Droits d'accès
 Schéma |   Nom   | Type  |     Droits d'accès    | Droits d'accès
 à la colonne | Politiques
--------+---------+-------+-----------------------
+-----------------------------+-----------
 public | mytable | table | miriam=arwdDxt/miriam+| col1:
           +|
        |         |       | =r/miriam            +|   miriam_rw=rw/
miriam       |
        |         |       | admin=arw/miriam      |
             |
(1 ligne)
```

Si la colonne « Droits d'accès » est vide pour un objet donné, cela signifie que l'objet a les droits par défaut(c'est-à-dire que l'entrée des droits dans les catalogues systèmes est NULL). Les droits par défaut incluent toujours tous les droits pour le propriétaire, et peuvent inclure certains droits pour PUBLIC en fonction du type d'objet comme expliqué plus haut. La première commande GRANT ou REVOKE sur un objet va instancier les droits par défaut (en produisant, par exemple, miriam=arwdDxt/miriam) puis les modifier selon la requête. De manière similaire, des entrées ne sont affichées dans « Droits d'accès à la colonne » que pour les colonnes qui n'ont pas les droits par défaut.(Note : dans ce contexte, on entend par « droits par défaut » les droits par défaut intégrés à PostgreSQL pour le type d'objet. Un objet dont les droits auront été affectés par une commande ALTER DEFAULT PRIVILEGES montrera toujours une entrée de droits explicite qui inclue les effets du ALTER.)

Notez que les droits de transmission (*grant option*) implicites ne sont pas indiqués dans l'affichage des droits d'accès. Un * n'apparaîtra que si la transmission a été explicitement accordée à quelqu'un.

5.8. Politiques de sécurité niveau ligne

En plus des systèmes de droits du standard SQL disponibles via GRANT, les tables peuvent avoir des *politiques de sécurité pour l'accès aux lignes* qui restreignent, utilisateur par utilisateur, les lignes qui peuvent être renvoyées par les requêtes d'extraction ou les commandes d'insertions, de mises à jour ou de suppressions. Cette fonctionnalité est aussi connue sous le nom *Row-Level Security*. Par défaut, les tables n'ont aucune politique de ce type pour que, si un utilisateur a accès à une table selon les droits du standard SQL, toutes les lignes de la table soient accessibles aux requêtes de lecture ou d'écriture.

Lorsque la protection des lignes est activée sur une table (avec l'instruction ALTER TABLE ... ENABLE ROW LEVEL SECURITY), tous les accès classiques à la table pour sélectionner ou modifier des lignes doivent être autorisés par une politique de sécurité. Cependant, le propriétaire de la table n'est typiquement pas soumis aux politiques de sécurité. Si aucune politique n'existe pour la table, une politique de rejet est utilisée par défaut, ce qui signifie qu'aucune ligne n'est visible ou ne peut être modifiée. Les opérations qui s'appliquent pour la table dans sa globalité, comme `TRUNCATE` et `REFERENCES`, ne sont pas soumises à ces restrictions de niveau ligne.

Les politiques de sécurité niveau ligne peuvent s'appliquer en particulier soit à des commandes, soit à des rôles, soit aux deux. Une politique est indiquée comme s'appliquant à toutes les commandes par `ALL`, ou seulement à `SELECT`, `INSERT`, `UPDATE` ou `DELETE`. Plusieurs rôles peuvent être affectés à une politique donnée, et les règles normales d'appartenance et d'héritage s'appliquent.

Pour indiquer les lignes visibles ou modifiables pour une politique, une expression renvoyant un booléen est requise. Cette expression sera évaluée pour chaque ligne avant toutes conditions ou fonctions qui seraient indiquées dans les requêtes de l'utilisateur. (La seule exception à cette règle concerne les fonctions marquées `leakproof`, qui annoncent ne pas dévoiler d'information ; l'optimiseur peut choisir d'appliquer de telles fonctions avant les vérifications de sécurité niveau ligne). Les lignes pour lesquelles l'expression ne renvoie pas `true` ne sont pas traitées. Des expressions différentes peuvent être indiquées pour fournir des contrôles indépendants pour les lignes qui sont visibles et pour celles qui sont modifiées. Les expressions attachées à la politique sont exécutées dans le cours de la requête et avec les droits de l'utilisateur qui exécute la commande, bien que les fonctions définies avec l'attribut `SECURITY DEFINER` puissent être utilisées pour accéder à des données qui ne seraient pas disponibles à l'utilisateur effectuant la requête.

Les superutilisateurs et les rôles avec l'attribut `BYPASSRLS` ne sont pas soumis au système de sécurité niveau ligne lorsqu'ils accèdent une table. Il en est de même par défaut du propriétaire d'une table, bien qu'il puisse choisir de se soumettre à ces contrôles avec ALTER TABLE ... FORCE ROW LEVEL SECURITY.

L'activation ou la désactivation de la sécurité niveau ligne, comme l'ajout des politiques à une table, est toujours le privilège du seul propriétaire de la table.

Les politiques sont créées en utilisant l'instruction CREATE POLICY, modifiées avec la commande ALTER POLICY et supprimées avec la commande DROP POLICY. Pour activer et désactiver la sécurité niveau ligne pour une table donnée, utilisez la commande ALTER TABLE.

Chaque politique possède un nom et de multiples politiques peuvent être définies pour une table. Comme les politiques sont spécifiques à une table, chaque politique pour une même table doit avoir un nom différent. Différentes tables peuvent avoir des noms de politique de même nom.

Lorsque plusieurs politiques sont applicables pour une même requête, elles sont combinées en utilisant `OR` (pour les politiques permissives, ce qui est le comportement par défaut) ou en utilisant `AND` (pour les politiques restrictives). C'est similaire à la règle qu'un rôle donné a les droits de tous les rôles dont il est membre. Les politiques permissives et restrictives sont discutées plus en détail ci-dessous.

À titre de simple exemple, nous allons ici créer une politique sur la relation `comptes` pour autoriser seulement les membres du rôle `admins` à accéder seulement aux lignes de leurs propres comptes :

```
CREATE TABLE comptes (admin text, societe text, contact_email
 text);
ALTER TABLE comptes ENABLE ROW LEVEL SECURITY;
```

```
CREATE POLICY compte_admins ON comptes TO admins
    USING (admin = current_user);
```

La politique ci-dessus prévoit implicitement une clause `WITH CHECK` identique à sa clause `USING`, de sorte que la contrainte s'applique à la fois aux lignes sélectionnées par une commande (un gestionnaire ne peut donc pas utiliser `SELECT`, `UPDATE`, ou `DELETE` sur des lignes existantes appartenant à un gestionnaire différent) et aux lignes modifiées par une commande (les lignes appartenant à un gestionnaire différent ne peuvent donc être créées avec `INSERT` ou `UPDATE`).

Si aucun rôle n'est indiqué ou si le nom de pseudo rôle `PUBLIC` est utilisé, alors la politique s'applique à tous les utilisateurs du système. Pour autoriser tous les utilisateurs à accéder à leurs propres lignes dans une table `utilisateurs`, une politique simple peut être utilisée :

```
CREATE POLICY police_utilisateur ON utilisateurs
    USING (user_name = current_user);
```

Cela fonctionne de la même manière que dans l'exemple précédent.

Pour utiliser une politique différente pour les lignes ajoutées à la table de celle appliquée pour les lignes visibles, plusieurs politiques peuvent être combinées. Cette paire de politiques autorisera tous les utilisateurs à voir toutes les lignes de la table `utilisateurs`, mais seulement à modifier les leurs :

```
CREATE POLICY user_sel_policy ON users
    FOR SELECT
    USING (true);
CREATE POLICY user_mod_policy ON users
    USING (user_name = current_user);
```

Pour une commande `SELECT`, ces deux politique sont combinées à l'aide de `OR`, ayant pour effet que toutes les lignes peuvent être sélectionnées. Pour les autres types de commandes, seule la deuxième politique s'applique, de sorte que les effets sont les mêmes qu'auparavant.

La sécurité niveau ligne peut également être désactivée avec la commande `ALTER TABLE`. La désactivation de la sécurité niveau ligne ne supprime pas les polices qui sont définies pour la table ; elles sont simplement ignorées. L'ensemble des lignes sont alors visibles et modifiables, selon le système standard des droits SQL.

Ci-dessous se trouve un exemple plus important de la manière dont cette fonctionnalité peut être utilisée en production. La table `passwd` simule le fichier des mots de passe d'un système Unix.

```
-- Simple exemple basé sur le fichier passwd
CREATE TABLE passwd (
  user_name             text UNIQUE NOT NULL,
  pwhash                text,
  uid                   int  PRIMARY KEY,
  gid                   int  NOT NULL,
  real_name             text NOT NULL,
  home_phone            text,
  extra_info            text,
  home_dir              text NOT NULL,
  shell                 text NOT NULL
);

CREATE ROLE admin;  -- Administrateur
CREATE ROLE bob;    -- Utilisateur normal
CREATE ROLE alice;  -- Utilisateur normal

-- Chargement de la table
```

```
INSERT INTO passwd VALUES
  ('admin','xxx',0,0,'Admin','111-222-3333',null,'/root','/bin/
dash');
INSERT INTO passwd VALUES
  ('bob','xxx',1,1,'Bob','123-456-7890',null,'/home/bob','/bin/
zsh');
INSERT INTO passwd VALUES
  ('alice','xxx',2,1,'Alice','098-765-4321',null,'/home/alice','/
bin/zsh');

-- Assurez-vous d'activer le row level security pour la table
ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;

-- Créer les polices
-- L'administrateur peut voir toutes les lignes et en ajouter comme
 il le souhaite
CREATE POLICY admin_all ON passwd TO admin USING (true) WITH CHECK
 (true);
-- Les utilisateurs normaux peuvent voir toutes les lignes
CREATE POLICY all_view ON passwd FOR SELECT USING (true);
-- Les utilisateurs normaux peuvent mettre à jour leurs propres
 lignes,
-- tout en limitant les shells qu'ils peuvent choisir
CREATE POLICY user_mod ON passwd FOR UPDATE
  USING (current_user = user_name)
  WITH CHECK (
    current_user = user_name AND
    shell IN ('/bin/bash','/bin/sh','/bin/dash','/bin/zsh','/bin/
tcsh')
  );

-- Donner à admin tous les droits normaux
GRANT SELECT, INSERT, UPDATE, DELETE ON passwd TO admin;
-- Les utilisateurs ne peuvent que sélectionner des colonnes
 publiques
GRANT SELECT
  (user_name, uid, gid, real_name, home_phone, extra_info,
 home_dir, shell)
  ON passwd TO public;
-- Autoriser les utilisateurs à mettre à jour certaines colonnes
GRANT UPDATE
  (pwhash, real_name, home_phone, extra_info, shell)
  ON passwd TO public;
```

Comme avec tous les réglages de sécurité, il est important de tester et de s'assurer que le système se comporte comme attendu. En utilisant l'exemple ci-dessus, les manipulations ci-dessous montrent que le système des droits fonctionne correctement.

```
-- admin peut voir toutes les lignes et les colonnes
postgres=> set role admin;
SET
postgres=> table passwd;
 user_name | pwhash | uid | gid | real_name |  home_phone  |
 extra_info | home_dir    |   shell
-----------+--------+-----+-----+-----------+--------------
+------------+-------------+-----------
 admin     | xxx    |   0 |   0 | Admin     | 111-222-3333 |
     | /root       | /bin/dash
```

```
 bob       | xxx    |   1 |   1 | Bob       | 123-456-7890 |
     | /home/bob   | /bin/zsh
 alice     | xxx    |   2 |   1 | Alice     | 098-765-4321 |
     | /home/alice | /bin/zsh
(3 rows)

-- Tester ce que Alice est capable de faire:
postgres=> set role alice;
SET
postgres=> table passwd;
ERROR:  permission denied for table passwd
postgres=> select
 user_name,real_name,home_phone,extra_info,home_dir,shell from
 passwd;
 user_name | real_name |  home_phone  | extra_info | home_dir    |
  shell
-----------+-----------+--------------+------------+-------------
+-----------
 admin     | Admin     | 111-222-3333 |            | /root
 | /bin/dash
 bob       | Bob       | 123-456-7890 |            | /home/bob
 | /bin/zsh
 alice     | Alice     | 098-765-4321 |            | /home/alice
 | /bin/zsh
(3 rows)

postgres=> update passwd set user_name = 'joe';
ERROR:  permission denied for table passwd
-- Alice est autorisée à modifier son propre nom (real_name), mais
 pas celui des autres
postgres=> update passwd set real_name = 'Alice Doe';
UPDATE 1
postgres=> update passwd set real_name = 'John Doe' where user_name
 = 'admin';
UPDATE 0
postgres=> update passwd set shell = '/bin/xx';
ERROR:  new row violates WITH CHECK OPTION for "passwd"
postgres=> delete from passwd;
ERROR:  permission denied for table passwd
postgres=> insert into passwd (user_name) values ('xxx');
ERROR:  permission denied for table passwd
-- Alice peut modifier son propre mot de passe; RLS empêche
 silencieusement la mise à jour d'autres lignes
postgres=> update passwd set pwhash = 'abc';
UPDATE 1
```

Toutes les politiques construites jusqu'à maintenant étaient des politiques permissives, ce qui veut dire que quand plusieurs politiques sont appliquées, elles sont combinées en utilisant l'opérateur booléen « OR ». Bien que les politiques permissives puissent être construites pour autoriser l'accès à des lignes dans les cas attendus, il peut être plus simple de combiner des politiques permissives avec des politiques restrictives (que l'enregistrement doit passer et qui sont combinées en utilisant l'opérateur booléen « AND »). En continuant sur l'exemple ci-dessus, nous ajoutons une politique restrictive pour exiger que l'administrateur soit connecté via un socket unix local pour accéder aux enregistrements de la table `passwd` :

```
CREATE POLICY admin_local_only ON passwd AS RESTRICTIVE TO admin
    USING (pg_catalog.inet_client_addr() IS NULL);
```

Nous pouvons alors voir qu'un administrateur se connectant depuis le réseau ne verra aucun enregistrement, du fait de la politique restrictive :

```
=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> select inet_client_addr();
 inet_client_addr
------------------
 127.0.0.1
(1 row)

=> TABLE passwd;
 user_name | pwhash | uid | gid | real_name | home_phone |
 extra_info | home_dir | shell
-----------+--------+-----+-----+-----------+------------
+------------+----------+-------
(0 rows)

=> UPDATE passwd set pwhash = NULL;
UPDATE 0
```

Les vérifications d'intégrité référentielle, telles que les contraintes d'unicité ou de clefs primaires et les références de clefs étrangères, passent toujours outre la sécurité niveau ligne pour s'assurer que l'intégrité des données est maintenue. Une attention particulière doit être prise lors de la mise en place des schémas et des politiques de sécurité de niveau ligne pour éviter qu'un canal caché (`covert channel`) ne dévoile des informations au travers de telles vérifications d'intégrité référentielle.

Dans certains contextes, il est important de s'assurer que la sécurité niveau ligne n'est pas appliquée. Par exemple, lors d'une sauvegarde, il serait désastreux si la sécurité niveau ligne avait pour conséquence de soustraire silencieusement certaines lignes de la sauvegarde. Dans une telle situation, vous pouvez positionner le paramètre de configuration row_security à `off`. En lui-même, ce paramètre ne contourne pas la sécurité niveau ligne ; ce qu'il fait, c'est qu'il lève une erreur si le résultat d'une des requêtes venait à être filtrée par une politique. La raison de l'erreur peut alors être recherchée et résolue.

Dans les exemples ci-dessus, les expressions attachées aux politiques considèrent uniquement les valeurs de la ligne courante accédée ou modifiée. C'est le cas le plus simple et le plus performant ; lorsque c'est possible, il est préférable de concevoir les applications qui utilisent cette fonctionnalité de la sorte. S'il est nécessaire de consulter d'autres lignes ou tables pour que la politique puisse prendre une décision, ceci peut être réalisé en utilisant dans les expressions des politiques des sous-requêtes `SELECT` ou des fonctions qui contiennent des commandes `SELECT`. Cependant, faites attention que de tels accès ne créent pas des accès concurrents qui pourraient permettre une fuite d'informations si aucune précaution n'est prise. À titre d'exemple, considérez la création de la table suivante :

```
-- définition des droits de groupes
CREATE TABLE groupes (groupe_id int PRIMARY KEY,
                      nom_groupe text NOT NULL);

INSERT INTO groupes VALUES
  (1, 'bas'),
  (2, 'moyen'),
  (5, 'haut');

GRANT ALL ON groupes TO alice;  -- alice est l'administratrice
GRANT SELECT ON groupes TO public;
```

```
-- définition des niveaux de droits utilisateurs
CREATE TABLE utilisateurs (nom_utilisateur text PRIMARY KEY,
                   groupe_id int NOT NULL REFERENCES groupes);

INSERT INTO utilisateurs VALUES
  ('alice', 5),
  ('bob', 2),
  ('mallory', 2);

GRANT ALL ON utilisateurs TO alice;
GRANT SELECT ON utilisateurs TO public;

-- table contenant l'information à protéger
CREATE TABLE information (info text,
                          groupe_id int NOT NULL REFERENCES
 groupes);

INSERT INTO information VALUES
  ('peu secret', 1),
  ('légèrement secret', 2),
  ('très secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;

-- une ligne devrait être visible et modifiable pour les
 utilisateurs
-- dont le groupe_id est égal ou plus grand au groupe_id de la
 ligne
CREATE POLICY fp_s ON information FOR SELECT
  USING (groupe_id <= (SELECT groupe_id FROM utilisateurs WHERE
 nom_utilisateur = current_user));
CREATE POLICY fp_u ON information FOR UPDATE
  USING (groupe_id <= (SELECT groupe_id FROM utilisateurs WHERE
 nom_utilisateur = current_user));

-- nous comptons sur les RLS pour protéger la table information
GRANT ALL ON information TO public;
```

Maintenant, supposez qu'`alice` souhaite modifier l'information « légèrement secrète », mais décide que `mallory` ne devrait pas pouvoir obtenir ce nouveau contenu, elle le fait ainsi :

```
BEGIN;
UPDATE utilisateurs SET groupe_id = 1 WHERE nom_utilisateur =
 'mallory';
UPDATE information SET info = 'caché à mallory' WHERE groupe_id =
 2;
COMMIT;
```

Ceci semble correct, il n'y a pas de fenêtre pendant laquelle `mallory` devrait pouvoir accéder à la chaîne « caché à mallory ». Cependant, il y a une situation de compétition ici. Si `mallory` fait en parallèle, disons :

```
SELECT * FROM information WHERE groupe_id = 2 FOR UPDATE;
```

et sa transaction est en mode `READ COMMITED`, il est possible qu'elle voit « caché à mallory ». C'est possible si sa transaction accède la ligne `information` juste après qu'`alice` l'ait fait. Elle est bloquée en attendant que la transaction d'`alice` valide, puis récupère la ligne mise à jour grâce à la clause `FOR UPDATE`. Cependant, elle ne récupère *pas* une ligne mise à jour pour la commande implicite `SELECT` sur la table utilisateurs parce que cette sous-commande n'a pas la clause `FOR`

`UPDATE` ; à la place, la ligne `utilisateurs` est lue avec une image de la base de données prise au début de la requête. Ainsi, l'expression de la politique teste l'ancienne valeur du niveau de droit de `mallory` et l'autorise à voir la valeur mise à jour.

Il y a plusieurs solutions à ce problème. Une simple réponse est d'utiliser `SELECT ... FOR SHARE` dans la sous-commande `SELECT` de la politique de sécurité niveau ligne. Cependant, ceci demande de donner le droit `UPDATE` sur la table référencée (ici `utilisateurs`) aux utilisateurs concernés, ce qui peut ne pas être souhaité. (Une autre politique de sécurité niveau ligne pourrait être mise en place pour les empêcher d'exercer ce droit ; ou la sous-commande `SELECT` pourrait être incluse dans une fonction marquée `security definer`.) De plus, l'utilisation intensive et concurrente de verrous partagés sur les lignes de la table référencée pourrait poser un problème de performance, tout spécialement si des mises à jour de cette table sont fréquentes. Une autre solution envisageable, si les mises à jour de la table référencée ne sont pas fréquentes, est de prendre un verrou de type `ACCESS EXCLUSIVE` sur la table référencée lors des mises à jour, de telle manière qu'aucune autre transaction concurrente ne pourrait consulter d'anciennes valeurs. Ou une transaction pourrait attendre que toutes les transactions se terminent après avoir validé une mise à jour de la table référencée et avant de faire des modifications qui reposent sur la nouvelle politique de sécurité.

Pour plus de détails, voir CREATE POLICY et ALTER TABLE.

5.9. Schémas

Une instance de bases de données PostgreSQL contient une ou plusieurs base(s) nommée(s). Les rôles et quelques autres types d'objets sont partagés sur l'ensemble de l'instance. Une connexion cliente au serveur ne peut accéder qu'aux données d'une seule base, celle indiquée dans la requête de connexion.

> **Note**
>
> Les rôles d'une instance n'ont pas obligatoirement le droit d'accéder à toutes les bases de l'instance. Le partage des noms de rôles signifie qu'il ne peut pas y avoir plusieurs rôles nommés `joe`, par exemple, dans deux bases de la même instance ; mais le système peut être configuré pour n'autoriser `joe` à accéder qu'à certaines bases.

Une base de données contient un (ou plusieurs) *schéma*(s) nommé(s) qui, eux, contiennent des tables. Les schémas contiennent aussi d'autres types d'objets nommés (types de données, fonctions et opérateurs, par exemple). Le même nom d'objet peut être utilisé dans différents schémas sans conflit ; par exemple, `schema1` et `mon_schema` peuvent tous les deux contenir une table nommée `ma_table`. À la différence des bases de données, les schémas ne sont pas séparés de manière rigide : un utilisateur peut accéder aux objets de n'importe quel schéma de la base de données à laquelle il est connecté, sous réserve qu'il en ait le droit.

Il existe plusieurs raisons d'utiliser les schémas :

- autoriser de nombreux utilisateurs à utiliser une base de données sans interférer avec les autres ;
- organiser les objets de la base de données en groupes logiques afin de faciliter leur gestion ;
- les applications tierces peuvent être placées dans des schémas séparés pour éviter les collisions avec les noms d'autres objets.

Les schémas sont comparables aux répertoires du système d'exploitation, à ceci près qu'ils ne peuvent pas être imbriqués.

5.9.1. Créer un schéma

Pour créer un schéma, on utilise la commande CREATE SCHEMA. Le nom du schéma est libre. Par exemple :

```
CREATE SCHEMA
    mon_schema;
```

Pour créer les objets d'un schéma ou y accéder, on écrit un *nom qualifié* constitué du nom du schéma et du nom de la table séparés par un point :

schema.table

Cela fonctionne partout où un nom de table est attendu, ce qui inclut les commandes de modification de la table et les commandes d'accès aux données discutées dans les chapitres suivants. (Pour des raisons de simplification, seules les tables sont évoquées, mais les mêmes principes s'appliquent aux autres objets nommés, comme les types et les fonctions.)

La syntaxe encore plus générale

base.schema.table

peut aussi être utilisée, mais à l'heure actuelle, cette syntaxe n'existe que pour des raisons de conformité avec le standard SQL. Si un nom de base de données est précisé, ce doit être celui de la base à laquelle l'utilisateur est connecté.

Pour créer une table dans le nouveau schéma, on utilise :

```
CREATE TABLE mon_schema.ma_table (
 ...
);
```

Pour effacer un schéma vide (tous les objets qu'il contient ont été supprimés), on utilise :

```
DROP SCHEMA mon_schema;
```

Pour effacer un schéma et les objets qu'il contient, on utilise :

```
DROP SCHEMA mon_schema CASCADE;
```

La Section 5.14 décrit le mécanisme général sous-jacent.

Il n'est pas rare de vouloir créer un schéma dont un autre utilisateur est propriétaire (puisque c'est l'une des méthodes de restriction de l'activité des utilisateurs à des *schémas* prédéfinis). La syntaxe en est :

```
CREATE SCHEMA nom_schema AUTHORIZATION nom_utilisateur;
```

Le nom du schéma peut être omis, auquel cas le nom de l'utilisateur est utilisé. Voir la Section 5.9.6 pour en connaître l'utilité.

Les noms de schéma commençant par `pg_` sont réservés pour les besoins du système et ne peuvent être créés par les utilisateurs.

5.9.2. Le schéma public

Dans les sections précédentes, les tables sont créées sans qu'un nom de schéma soit indiqué. Par défaut, ces tables (et les autres objets) sont automatiquement placées dans un schéma nommé « public ». Toute nouvelle base de données contient un tel schéma. Les instructions suivantes sont donc équivalentes :

```
CREATE TABLE produits ( ... );
```

et :

```
CREATE TABLE public.produits ( ... );
```

5.9.3. Chemin de parcours des schémas

Non seulement l'écriture de noms qualifiés est contraignante, mais il est, de toute façon, préférable de ne pas fixer un nom de schéma dans les applications. De ce fait, les tables sont souvent appelées par des *noms non qualifiés*, autrement dit le seul nom de la table. Le système détermine la table appelée en suivant un *chemin de recherche*, liste de schémas dans lesquels chercher. La première table correspondante est considérée comme la table voulue. S'il n'y a pas de correspondance, une erreur est remontée, quand bien même il existerait des tables dont le nom corresponde dans d'autres schémas de la base.

La possibilité de créer des objets de même nom dans différents schémas complique l'écriture d'une requête qui référence précisément les mêmes objets à chaque fois. Cela ouvre aussi la possibilité aux utilisateurs de modifier le comportement des requêtes des autres utilisations, par accident ou volontairement. À cause de la prévalence des noms non qualifiés dans les requêtes et de leur utilisation des schémas natifs de PostgreSQL, ajouter un schéma à `search_path` demande en effet à tous les utilisateurs d'avoir le droit `CREATE` sur ce schéma. Quand vous exécutez une requête ordinaire, un utilisateur mal intentionné capable de créer des objets dans un schéma de votre chemin de recherche peut prendre le contrôle et exécuter des fonctions SQL arbitraires comme si vous les exécutiez.

Le premier schéma du chemin de recherche est appelé schéma courant. En plus d'être le premier schéma parcouru, il est aussi le schéma dans lequel les nouvelles tables sont créées si la commande `CREATE TABLE` ne précise pas de nom de schéma.

Le chemin de recherche courant est affiché à l'aide de la commande :

```
SHOW search_path;
```

Dans la configuration par défaut, ceci renvoie :

```
 search_path
--------------
 "$user", public
```

Le premier élément précise qu'un schéma de même nom que l'utilisateur courant est recherché. En l'absence d'un tel schéma, l'entrée est ignorée. Le deuxième élément renvoie au schéma public précédemment évoqué.

C'est, par défaut, dans le premier schéma du chemin de recherche qui existe que sont créés les nouveaux objets. C'est la raison pour laquelle les objets sont créés, par défaut, dans le schéma public. Lorsqu'il est fait référence à un objet, dans tout autre contexte, sans qualification par un schéma (modification de table, modification de données ou requêtes), le chemin de recherche est traversé jusqu'à ce qu'un objet correspondant soit trouvé. C'est pourquoi, dans la configuration par défaut, tout accès non qualifié ne peut que se référer au schéma public.

Pour ajouter un schéma au chemin, on écrit :

```
SET search_path TO mon_schema,public;
```

(`$user` est omis à ce niveau, car il n'est pas immédiatement nécessaire.) Il est alors possible d'accéder à la table sans qu'elle soit qualifiée par un schéma :

```
DROP TABLE ma_table;
```

Puisque `mon_schema` est le premier élément du chemin, les nouveaux objets sont, par défaut, créés dans ce schéma.

On peut aussi écrire :

```
SET search_path TO mon_schema;
```

Dans ce cas, le schéma public n'est plus accessible sans qualification explicite. Hormis le fait qu'il existe par défaut, le schéma public n'a rien de spécial. Il peut même être effacé.

On peut également se référer à la Section 9.26 qui détaille les autres façons de manipuler le chemin de recherche des schémas.

Le chemin de recherche fonctionne de la même façon pour les noms de type de données, les noms de fonction et les noms d'opérateur que pour les noms de table. Les noms des types de données et des fonctions peuvent être qualifiés de la même façon que les noms de table. S'il est nécessaire d'écrire un nom d'opérateur qualifié dans une expression, il y a une condition spéciale. Il faut écrire :

```
OPERATOR(schéma.opérateur)
```

Cela afin d'éviter toute ambiguïté syntaxique. Par exemple :

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

En pratique, il est préférable de s'en remettre au chemin de recherche pour les opérateurs, afin de ne pas avoir à écrire quelque chose d'aussi étrange.

5.9.4. Schémas et droits

Par défaut, les utilisateurs ne peuvent pas accéder aux objets présents dans les schémas qui ne leur appartiennent pas. Pour le permettre, le propriétaire du schéma doit donner le droit `USAGE` sur le schéma. Par défaut, tout le monde a ce droit sur le schéma `public`. Pour autoriser les utilisateurs à manipuler les objets d'un schéma, des droits supplémentaires doivent éventuellement être accordés, en fonction de l'objet.

Un utilisateur peut aussi être autorisé à créer des objets dans le schéma d'un autre utilisateur. Pour cela, le droit `CREATE` doit être donné sur ce schéma. Dans les bases de données mises à jour à partir de PostgreSQL 14 ou les versions précédentes, tout le monde a ce droit sur le schéma `public`. Certaines méthodes d'usage demandent à révoquer ce droit :

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

Le premier « public » est le schéma, le second « public » signifie « tout utilisateur ». Dans le premier cas, c'est un identifiant, dans le second, un mot-clé, d'où la casse différente. (Se reporter aux règles de la Section 4.1.1.)

5.9.5. Le schéma du catalogue système

En plus du schéma `public` et de ceux créés par les utilisateurs, chaque base de données contient un schéma `pg_catalog`. Celui-ci contient les tables système et tous les types de données, fonctions et opérateurs intégrés. `pg_catalog` est toujours dans le chemin de recherche. S'il n'est pas nommé explicitement dans le chemin, il est parcouru implicitement *avant* le parcours des schémas du chemin. Cela garantit que les noms internes sont toujours accessibles. En revanche, `pg_catalog` peut être explicitement placé à la fin si les noms utilisateur doivent surcharger les noms internes.

Comme les noms des catalogues système commencent par `pg_`, il est préférable d'éviter d'utiliser de tels noms pour se prémunir d'éventuels conflits si une version ultérieure devait définir une table système qui porte le même nom que la table créée. (Le chemin de recherche par défaut implique qu'une référence non qualifiée à cette table pointe sur la table système). Les tables système continueront de suivre la convention qui leur impose des noms préfixés par `pg_`. Il n'y a donc pas de conflit possible avec des noms de table utilisateur non qualifiés, sous réserve que les utilisateurs évitent le préfixe `pg_`.

5.9.6. Utilisation

Les schémas peuvent être utilisés de différentes façons pour organiser les données. Un *modèle d'utilisation de schéma sécurisé* empêche tous les utilisateurs pour lesquels nous n'avons pas confiance de modifier le comportement des requêtes des autres utilisateurs. Quand une base de données n'utilise pas de modèle d'utilisation de schéma sécurisé, les utilisateurs souhaitant interroger cette base de

données en toute sécurité devront prendre des mesures de protection au début de chaque session. Plus précisément, ils commenceraient chaque session par la configuration du paramètre `search_path` en une chaîne vide ou en supprimant de leur `search_path` les schémas accessibles en écriture par des utilisateurs standards. Il existe quelques modèles d'utilisation facilement pris en charge par la configuration par défaut :

- Contraindre les utilisateurs à de schémas privés. Pour implémenter cela, assurez-vous tout d'abord que les schémas n'ont pas le droit `CREATE` pour public. Ensuite, pour chaque utilisateur devant créer des objets permanents, créez un schéma de même nom que l'utilisateur, par exemple `CREATE SCHEMA alice AUTHORIZATION alice`. (Rappelez-vous que le chemin de recherche par défaut commence avec `$user`, ce qui est remplacé par le nom de l'utilisateur. De ce fait, si chaque utilisateur a un schéma séparé, ils accèdent à leur schéma par défaut.) Cette méthode est une méthode d'utilisation de schéma sécurisé sauf si un utilisateur malin est le propriétaire de la base de données ou dispose de l'attribut `CREATEROLE`, auquel cas aucune méthode d'utilisation sécurisée des schémas n'existe.

  Avec PostgreSQL 15 et les versions suivantes, la configuration par défaut accepte cette méthode d'utilisation. Dans les versions précédentes, ou lors de l'utilisation d'une base de donnée mise à jour d'une version précédente, vous aurez besoin de supprimer l'attribut `CREATE` sur public à partir du schéma `public` (lancez `REVOKE CREATE ON SCHEMA public FROM PUBLIC`). Puis considérez la réalisation d'un audit du schéma `public` pour des objets nommés comme les objets du schéma `pg_catalog`.

- Supprimer le schéma public du chemin de recherche par défaut, en modifiant le fichier `postgresql.conf` ou en exécutant `ALTER ROLE ALL SET search_path = "$user"`. Puis donnez les droits pour créer des objets dans le schéma public. Seuls les noms qualifiés choisiront des objets du schéma public. Bien que les références de table qualifiées soient correctes, les appels aux fonctions dans le schéma public seront dangereux ou peu fiables. Si vous créez des fonctions ou des extensions dans le schéma public, utilisez le premier modèle à la place. Sinon, tout comme le premier modèle, c'est sécurisé sauf si un utilisateur non fiable est le propriétaire de la base de données ou détient l'attribut `CREATEROLE`.

- Conserver le chemin de recherche par défaut, et donner les droits de création sur le schéma public. Tous les utilisateurs ont accès au schéma public implicitement. Ceci simule la situation où les schémas ne sont pas du tout disponibles, réalisant ainsi une transition en douceur vers un monde qui ne connait pas les schémas. Néanmoins, ceci ne sera jamais un modèle sécurisé. C'est uniquement acceptable si la base de données ne contient qu'un seul utilisateur ou quelques utilisateurs qui se font mutuellement confiance. Dans les bases mises à jour à partir de PostgreSQL 14 ou antérieures, c'est le comportement par défaut.

Pour chaque méthode, pour installer des applications partagées (tables utilisées par tout le monde, fonctions supplémentaires fournies par des tiers, etc.), placez-les dans des schémas séparés. N'oubliez pas d'accorder les droits appropriés pour permettre aux autres utilisateurs d'y accéder. Les utilisateurs peuvent ensuite faire référence à ces objets supplémentaires en les qualifiant avec le nom du schéma, ou bien ils peuvent placer les schémas supplémentaires dans leur chemin de recherche, suivant leur préférence.

5.9.7. Portabilité

Dans le standard SQL, la notion d'objets d'un même schéma appartenant à des utilisateurs différents n'existe pas. De plus, certaines implantations ne permettent pas de créer des schémas de nom différent de celui de leur propriétaire. En fait, les concepts de schéma et d'utilisateur sont presque équivalents dans un système de base de données qui n'implante que le support basique des schémas tel que spécifié dans le standard. De ce fait, beaucoup d'utilisateurs considèrent les noms qualifiés comme correspondant en réalité à *`utilisateur.table`*. C'est comme cela que PostgreSQL se comporte si un schéma utilisateur est créé pour chaque utilisateur.

Le concept de schéma `public` n'existe pas non plus dans le standard SQL. Pour plus de conformité au standard, le schéma `public` ne devrait pas être utilisé.

Certains systèmes de bases de données n'implantent pas du tout les schémas, ou fournissent le support de *namespace* en autorisant (peut-être de façon limitée) l'accès inter-bases de données. Dans ce cas, la portabilité maximale est obtenue en n'utilisant pas les schémas.

5.10. L'héritage

PostgreSQL implante l'héritage des tables, qui peut s'avérer très utile pour les concepteurs de bases de données. (SQL:1999 et les versions suivantes définissent une fonctionnalité d'héritage de type qui diffère par de nombreux aspects des fonctionnalités décrites ici.)

Soit l'exemple d'un modèle de données de villes. Chaque état comporte plusieurs villes, mais une seule capitale. Pour récupérer rapidement la ville capitale d'un état donné, on peut créer deux tables, une pour les capitales et une pour les villes qui ne sont pas des capitales. Mais, que se passe-t-il dans le cas où toutes les données d'une ville doivent être récupérées, qu'elle soit une capitale ou non ? L'héritage peut aider à résoudre ce problème. La table `capitales` est définie pour hériter de `villes` :

```
CREATE TABLE villes (
    nom             text,
    population      float,
    elevation        int     -- (en pied)
);

CREATE TABLE capitales (
    etat           char(2)
) INHERITS (villes);
```

Dans ce cas, la table `capitales` *hérite* de toutes les colonnes de sa table parente, `villes`. Les capitales ont aussi une colonne supplémentaire, `etat`, qui indique l'état dont elles sont capitales.

Dans PostgreSQL, une table peut hériter de zéro à plusieurs autres tables et une requête faire référence aux lignes d'une table ou à celles d'une table et de ses descendantes. Ce dernier comportement est celui par défaut.

Par exemple, la requête suivante retourne les noms et élévations de toutes les villes, y compris les capitales, situées à une élévation supérieure à 500 pieds :

```
SELECT nom, elevation
FROM villes
WHERE elevation > 500;
```

Avec les données du tutoriel de PostgreSQL (voir Section 2.1), ceci renvoie :

```
   nom     | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
```

D'un autre côté, la requête suivante retourne les noms et élévations de toutes les villes, qui ne sont pas des capitales, situées à une élévation supérieure à 500 pieds :

```
SELECT nom, elevation
FROM ONLY villes
WHERE elevation > 500;

   nom     | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
```

Le mot-clé ONLY indique que la requête s'applique uniquement aux villes, et non pas à toutes les tables en dessous de villes dans la hiérarchie de l'héritage. Un grand nombre des commandes déjà évoquées -- SELECT, UPDATE et DELETE -- supportent le mot-clé ONLY.

Vous pouvez aussi écrire le nom de la table avec un astérisque (*) à la fin pour indiquer spécifiquement que les tables filles sont incluses :

```
SELECT name, elevation
FROM cities*
WHERE elevation > 500;
```

Écrire l'astérisque (*) n'est pas nécessaire, puisque ce comportement est toujours le comportement par défaut. Toutefois, cette syntaxe est toujours supportée pour raison de compatibilité avec les anciennes versions où le comportement par défaut pouvait être changé.

Dans certains cas, il peut être intéressant de savoir de quelle table provient une ligne donnée. Une colonne système appelée TABLEOID, présente dans chaque table, donne la table d'origine :

```
SELECT v.tableoid, v.nom, v.elevation
FROM villes v
WHERE v.elevation > 500;
```

qui renvoie :

```
 tableoid |   nom     | elevation
----------+-----------+-----------
   139793 | Las Vegas |      2174
   139793 | Mariposa  |      1953
   139798 | Madison   |       845
```

(Reproduire cet exemple conduit probablement à des OID numériques différents). Une jointure avec pg_class, permet d'obtenir les noms réels des tables :

```
SELECT p.relname, v.nom, v.elevation
FROM villes v, pg_class p
WHERE v.elevation > 500 AND v.tableoid = p.oid;
```

ce qui retourne :

```
 relname   |   nom     | elevation
-----------+-----------+-----------
 villes    | Las Vegas |      2174
 villes    | Mariposa  |      1953
 capitales | Madison   |       845
```

Une autre manière d'obtenir le même effet est d'utiliser le pseudo-type regclass qui affichera l'OID de la table de façon symbolique :

```
SELECT v.tableoid::regclass, v.nom, v.elevation
FROM villes v
WHERE v.elevation > 500;
```

L'héritage ne propage pas automatiquement les données des commandes INSERT ou COPY aux autres tables de la hiérarchie de l'héritage. Dans l'exemple considéré, l'instruction INSERT suivante échoue :

```
INSERT INTO villes (nom, population, elevation, etat)
VALUES ('Albany', NULL, NULL, 'NY');
```

On pourrait espérer que les données soient d'une manière ou d'une autre acheminées vers la table capitales, mais ce n'est pas le cas : INSERT insère toujours dans la table indiquée. Dans certains cas, il est possible de rediriger l'insertion en utilisant une règle (voir Chapitre 41). Néanmoins, cela

n'est d'aucune aide dans le cas ci-dessus, car la table `villes` ne contient pas la colonne `etat`. La commande est donc rejetée avant que la règle ne puisse être appliquée.

Toutes les contraintes de vérification et toutes les contraintes `NOT NULL` sur une table parent sont automatiquement héritées par les tables enfants, sauf si elles sont spécifiées explicitement avec des clauses `NO INHERIT`. Les autres types de contraintes (unicité, clé primaire, clé étrangère) ne sont pas hérités.

Une table peut hériter de plusieurs tables, auquel cas elle possède l'union des colonnes définies par les tables mères. Toute colonne déclarée dans la définition de la table enfant est ajoutée à cette dernière. Si le même nom de colonne apparaît dans plusieurs tables mères, ou à la fois dans une table mère et dans la définition de la table enfant, alors ces colonnes sont « assemblées » pour qu'il n'en existe qu'une dans la table enfant. Pour être assemblées, les colonnes doivent avoir le même type de données, sinon une erreur est levée. Les contraintes de vérification et les contraintes non NULL héritables sont assemblées de façon similaire. De ce fait, par exemple, une colonne assemblée sera marquée non NULL si une des définitions de colonne d'où elle provient est marquée non NULL. Les contraintes de vérification sont assemblées si elles ont le même nom, et l'assemblage échouera si leurs conditions sont différentes.

L'héritage de table est établi à la création de la table enfant, à l'aide de la clause `INHERITS` de l'instruction `CREATE TABLE`. Alternativement, il est possible d'ajouter à une table, définie de façon compatible, une nouvelle relation de parenté à l'aide de la clause `INHERIT` de `ALTER TABLE`. Pour cela, la nouvelle table enfant doit déjà inclure des colonnes de mêmes nom et de même type que les colonnes de la table parent. Elle doit aussi contenir des contraintes de vérification de mêmes nom et expression que celles de la table parent.

De la même façon, un lien d'héritage peut être supprimé d'un enfant à l'aide de la variante `NO INHERIT` d'`ALTER TABLE`. Ajouter et supprimer dynamiquement des liens d'héritage de cette façon est utile quand cette relation d'héritage est utilisée pour le partitionnement des tables (voir Section 5.11).

Un moyen pratique de créer une table compatible en vue d'en faire ultérieurement une table enfant est d'utiliser la clause `LIKE` dans `CREATE TABLE`. Ceci crée une nouvelle table avec les mêmes colonnes que la table source. S'il existe des contraintes `CHECK` définies sur la table source, l'option `INCLUDING CONSTRAINTS` de `LIKE` doit être indiquée, car le nouvel enfant doit avoir des contraintes qui correspondent à celles du parent pour être considéré compatible.

Une table mère ne peut pas être supprimée tant qu'elle a des enfants. De même, les colonnes ou les contraintes de vérification des tables enfants ne peuvent être supprimées ou modifiées si elles sont héritées. La suppression d'une table et de tous ses descendants peut être aisément obtenue en supprimant la table mère avec l'option `CASCADE` (voir Section 5.14).

`ALTER TABLE` propage toute modification dans les définitions des colonnes et contraintes de vérification à travers la hiérarchie d'héritage. Là encore, supprimer des colonnes qui dépendent d'autres tables mères n'est possible qu'avec l'option `CASCADE`. `ALTER TABLE` suit les mêmes règles d'assemblage de colonnes dupliquées et de rejet que l'instruction `CREATE TABLE`.

Les requêtes sur tables héritées réalisent des vérifications de droit sur la table parent seulement. De ce fait, par exemple, donner le droit `UPDATE` sur la table `villes` implique que les droits de mise à jour des lignes dans la table `capitales` soient elles aussi vérifiées quand elles sont accédées via la table `villes`. Ceci préserve l'apparence que les données proviennent (aussi) de la table parent. Mais la table `capitales` ne pouvait pas être mise à jour directement sans droit supplémentaire. De façon similaire, les politiques de sécurité au niveau ligne de la table parent (voir Section 5.8) sont appliquées aux lignes provenant des tables filles avec une requête héritée. Les politiques de tables enfant sont appliquées seulement quand la table enfant est explicitement nommée dans la requête. Dans ce cas, toute politique attachée à ses parents est ignorée.

Les tables distantes (voir Section 5.12) peuvent aussi participer aux hiérarchies d'héritage, soit comme table parent, soit comme table enfant, comme les tables standards peuvent l'être. Si une table distante fait partie d'une hiérarchie d'héritage, toutes les opérations non supportées par la table étrangère ne sont pas non plus supportées sur l'ensemble de la hiérarchie.

5.10.1. Restrictions

Notez que toutes les commandes SQL fonctionnent avec les héritages. Les commandes utilisées pour récupérer des données, pour modifier des données ou pour modifier le schéma (autrement dit `SELECT`, `UPDATE`, `DELETE`, la plupart des variantes de `ALTER TABLE`, mais pas `INSERT` ou `ALTER TABLE ... RENAME`) incluent par défaut les tables filles et supportent la notation `ONLY` pour les exclure. Les commandes qui font de la maintenance de bases de données et de la configuration (par exemple `REINDEX`, `VACUUM`) fonctionnent typiquement uniquement sur les tables physiques, individuelles et ne supportent pas la récursion sur les tables de l'héritage. Le comportement respectif de chaque commande individuelle est documenté dans la référence (Commandes SQL).

Il existe une réelle limitation à la fonctionnalité d'héritage : les index (dont les contraintes d'unicité) et les contraintes de clés étrangères ne s'appliquent qu'aux tables mères, pas à leurs héritiers. Cela est valable pour le côté référençant et le côté référencé d'une contrainte de clé étrangère. Ce qui donne, dans les termes de l'exemple ci-dessus :

- si `villes.nom` est déclarée `UNIQUE` ou clé primaire (`PRIMARY KEY`), cela n'empêche pas la table `capitales` de posséder des lignes avec des noms dupliqués dans `villes`. Et ces lignes dupliquées s'affichent par défaut dans les requêtes sur `villes`. En fait, par défaut, `capitales` n'a pas de contrainte d'unicité du tout et, du coup, peut contenir plusieurs lignes avec le même nom. Une contrainte d'unicité peut être ajoutée à `capitales`, mais cela n'empêche pas la duplication avec `villes` ;

- de façon similaire, si `villes.nom` fait référence (`REFERENCES`) à une autre table, cette contrainte n'est pas automatiquement propagée à `capitales`. Il est facile de contourner ce cas de figure en ajoutant manuellement la même contrainte `REFERENCES` à `capitales` ;

- si une autre table indique `REFERENCES villes (nom)`, cela l'autorise à contenir les noms des villes, mais pas les noms des capitales. Il n'existe pas de contournement efficace de ce cas.

Certaines fonctionnalité non implémentées pour l'héritage le sont pour le partitionnement déclaratif. Réfléchissez soigneusement avant de décider de l'utilité de l'ancien partitionnement par héritage pour votre application.

5.11. Partitionnement de tables

PostgreSQL offre un support basique du partitionnement de table. Cette section explique pourquoi et comment implanter le partitionnement lors de la conception de la base de données.

5.11.1. Aperçu

Le partitionnement fait référence à la division d'une table logique volumineuse en plusieurs parties physiques plus petites. Le partitionnement comporte de nombreux avantages :

- les performances des requêtes peuvent être significativement améliorées dans certaines situations, particulièrement lorsque la plupart des lignes fortement accédées d'une table se trouvent sur une seule partition ou sur un petit nombre de partitions. Le partitionnement se substitue aux niveaux élevés de index, facilitant la tenue en mémoire des parties les plus utilisées de l'index ;

- lorsque les requêtes ou les mises à jour accèdent à un pourcentage important d'une seule partition, les performances peuvent être grandement améliorées par l'utilisation avantageuse d'un parcours séquentiel sur cette partition plutôt que d'utiliser un index qui nécessiterait des lectures aléatoires réparties sur toute la table ;

- les chargements et suppressions importants de données peuvent être obtenus par l'ajout ou la suppression de partitions, sous réserve que ce besoin ait été pris en compte lors de la conception du partitionnement. Supprimer une partition individuelle en utilisant `DROP TABLE` ou en exécutant `ALTER TABLE DETACH PARTITION` est bien plus rapide qu'une opération groupée. Cela évite également la surcharge due au `VACUUM` causé par un `DELETE` massif ;

- les données peu utilisées peuvent être déplacées sur un média de stockage moins cher et plus lent.

Ces bénéfices ne sont réellement intéressants que si cela permet d'éviter une table autrement plus volumineuse. Le point d'équilibre exact à partir duquel une table tire des bénéfices du partitionnement dépend de l'application. Toutefois, le partitionnement doit être envisagé si la taille de la table peut être amenée à dépasser la taille de la mémoire physique du serveur.

PostgreSQL offre un support natif pour les formes suivantes de partitionnement :

Partitionnement par intervalles

La table est partitionnée en « intervalles » (ou échelles) définis par une colonne clé ou par un ensemble de colonnes clés, sans recouvrement entre les intervalles de valeurs affectées aux différentes partitions. Il est possible, par exemple, de partitionner par intervalles de date ou par intervalles d'identifiants pour des objets métier particuliers. Chaque limite de l'intervalle est comprise comme étant inclusive au point initial et exclusive au point final. Par exemple, si l'intervalle d'une partition va de `1` à `10`, et que le prochain intervalle va de `10` à `20`, alors la valeur `10` appartient à la deuxième partition, et non pas à la première.

Partitionnement par liste

La table est partitionnée en listant explicitement les valeurs clés qui apparaissent dans chaque partition.

Partitionnement par hachage

La table est partitionnée en spécifiant un modulo et un reste pour chaque partition. Chaque partition contiendra les lignes pour lesquelles la valeur de hachage de la clé de partition divisée par le modulo spécifié produira le reste spécifié.

Si votre application nécessite d'utiliser d'autres formes de partitionnement qui ne sont pas listées au-dessus, des méthodes alternatives comme l'héritage et des vues `UNION ALL` peuvent être utilisées à la place. De telles méthodes offrent de la flexibilité, mais n'ont pas certains des bénéfices de performance du partitionnement déclaratif natif.

5.11.2. Partitionnement déclaratif

PostgreSQL donne un moyen de déclarer qu'une table est divisée en partitions. La table qui est divisée est appelée *table partitionnée*. La déclaration inclut la *méthode de partitionnement*, comme décrite ci-dessus, et une liste de colonnes ou d'expressions à utiliser comme *clé de partitionnement*.

La table partitionnée est elle-même une table « virtuelle » sans stockage propre. À la place, le stockage se fait dans les *partitions*, qui sont en fait des tables ordinaires mais associées à la table partitionnée. Chaque partition enregistre un sous-ensemble de données correspondant à la définition de ses *limites de partition*. Tous les lignes insérées dans une table partitionnée seront transférées sur la partition appropriée suivant les valeurs des colonnes de la clé de partitionnement. Mettre à jour la clé de partitionnement d'une ligne causera son déplacement dans une partition différente si elle ne satisfait plus les limites de sa partition originale.

Les partitions peuvent elles-mêmes être définies comme des tables partitionnées, ce qui aboutirait à du *sous-partitionnement*. Bien que toutes les partitions doivent avoir les mêmes colonnes que leur parent partitionné, les partitions peuvent avoir leurs propres index, contraintes et valeurs par défaut, différents de ceux des autres partitions. Voir CREATE TABLE pour plus de détails sur la création des tables partitionnées et des partitions.

Il n'est pas possible de transformer une table standard en table partitionnée et inversement. Par contre, il est possible d'ajouter une table standard ou une table partitionnée existante comme une partition d'une table partitionnée, ou de supprimer une partition d'une table partitionnée, pour la transformer en table standard ; ceci peut simplifier et accélérer de nombreux traitements de maintenance. Voir

ALTER TABLE pour en apprendre plus sur les sous-commandes ATTACH PARTITION et DETACH PARTITION.

Les partitions peuvent également être des tables étrangères, mais il faut faire très attention car c'est de la responsabilité de l'utilisateur que le contenu de la table distante satisfasse la clé de partitionnement. Il existe aussi d'autres restrictions. Voir CREATE FOREIGN TABLE pour plus d'informations.

5.11.2.1. Exemple

Imaginons que nous soyons en train de construire une base de données pour une grande société de crème glacée. La société mesure les pics de températures chaque jour, ainsi que les ventes de crème glacée dans chaque région. Conceptuellement, nous voulons une table comme ceci :

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

La plupart des requêtes n'accèdent qu'aux données de la dernière semaine, du dernier mois ou du dernier trimestre, car cette table est essentiellement utilisée pour préparer des rapports en ligne pour la direction. Pour réduire le nombre de données anciennes à stocker, seules les trois dernières années sont conservées. Au début de chaque mois, les données du mois le plus ancien sont supprimées. Dans cette situation, le partitionnement permet de répondre aux différents besoins identifiés sur la table des mesures.

Pour utiliser le partitionnement déclaratif dans ce cas d'utilisation, il faut suivre les étapes suivantes :

1. Créer une table measurement comme une table partitionnée en spécifiant la clause PARTITION BY, ce qui inclut la méthode de partitionnement (RANGE dans ce cas) ainsi que la colonne ou la liste de colonnes à utiliser comme clé de partitionnement.

   ```
   CREATE TABLE measurement (
       city_id         int not null,
       logdate         date not null,
       peaktemp        int,
       unitsales       int
   ) PARTITION BY RANGE (logdate);
   ```

2. Créez les partitions. La définition de chaque partition doit spécifier les limites qui correspondent à la méthode de partitionnement ainsi qu'à la clé de partitionnement du parent. Veuillez noter que spécifier des limites telles que les valeurs de la nouvelle partition pourront se chevaucher avec celles d'une ou plusieurs autres partitions retournera une erreur.

   Les partitions ainsi créées sont de tous les points de vue des tables PostgreSQL normales (ou, potentiellement, des tables distantes). Il est possible de spécifier un tablespace et des paramètres de stockage pour chacune des partitions séparément.

   Pour notre exemple, chaque partition devrait contenir un mois de données pour correspondre au besoin de supprimer un mois de données à la fois. Les commandes pourraient ressembler à ceci :

   ```
   CREATE TABLE measurement_y2006m02 PARTITION OF measurement
       FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');

   CREATE TABLE measurement_y2006m03 PARTITION OF measurement
       FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');

    ...
   CREATE TABLE measurement_y2007m11 PARTITION OF measurement
   ```

```
    FOR VALUES FROM ('2007-11-01') TO ('2007-12-01');

CREATE TABLE measurement_y2007m12 PARTITION OF measurement
    FOR VALUES FROM ('2007-12-01') TO ('2008-01-01')
    TABLESPACE fasttablespace;

CREATE TABLE measurement_y2008m01 PARTITION OF measurement
    FOR VALUES FROM ('2008-01-01') TO ('2008-02-01')
    WITH (parallel_workers = 4)
    TABLESPACE fasttablespace;
```

(Pour rappel, les partitions adjacentes peuvent partager une valeur de limite car les limites hautes sont traitées comme des limites exclusive.)

Si vous voulez mettre en place du sous-partitionnement, spécifiez la clause `PARTITION BY` dans les commandes utilisées pour créer des partitions individuelles, par exemple :

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
    FOR VALUES FROM ('2006-02-01') TO ('2006-03-01')
    PARTITION BY RANGE (peaktemp);
```

Après avoir créé les partitions de `measurement_y2006m02`, toute donnée insérée dans `measurement` qui correspond à `measurement_y2006m02` (ou donnée qui est directement insérée dans `measurement_y2006m02`, ce qui est autorisé à condition que la contrainte de partition soit respectée) sera redirigée vers l'une de ses partitions en se basant sur la colonne `peaktemp`. La clé de partition spécifiée pourrait se chevaucher avec la clé de partition du parent, il faut donc faire spécialement attention lorsque les limites d'une sous-partition sont spécifiées afin que l'ensemble de données qu'elle accepte constitue un sous-ensemble de ce que les propres limites de la partition acceptent ; le système n'essayera pas de vérifier si c'est vraiment le cas.

Insérer des données dans la table parent, données qui ne correspondent pas à une des partitions existantes, causera une erreur ; une partition appropriée doit être ajoutée manuellement.

Il n'est pas nécessaire de créer manuellement les contraintes de table décrivant les conditions des limites de partition pour les partitions. De telles contraintes seront créées automatiquement.

3. Créez un index sur la ou les colonnes de la clé, ainsi que tout autre index que vous pourriez vouloir pour chaque partition. (L'index sur la clé n'est pas strictement nécessaire, mais c'est utile dans la plupart des scénarios.) Ceci crée automatiquement un index correspondant sur chaque partition, et toutes les partitions que vous créerez ou attacherez plus tard auront elles-aussi cet index. Un index ou une contrainte unique déclarée sur une table partitionnée est « virtuel » de la même façon que la table partitionnée l'est : les données réelles sont dans les index enfants sur les partitions individuelles.

   ```
   CREATE INDEX ON measurement (logdate);
   ```

   Assurez-vous que le paramètre de configuration enable_partition_pruning ne soit pas désactivé dans `postgresql.conf`. S'il l'est, les requêtes ne seront pas optimisées comme voulu.

Dans l'exemple ci-dessus, nous créerions une nouvelle partition chaque mois, il serait donc avisé d'écrire un script qui génère le DDL nécessaire automatiquement.

5.11.2.2. Maintenance des partitions

Normalement, l'ensemble des partitions établies lors de la définition initiale de la table n'a pas vocation à demeurer statique. Il est courant de vouloir supprimer les partitions contenant d'anciennes données et d'ajouter périodiquement de nouvelles partitions pour de nouvelles données. Un des avantages les plus importants du partitionnement est précisément qu'il permet d'exécuter instantanément cette tâche de maintenance normalement pénible, en manipulant la structure partitionnée, plutôt que de bouger physiquement de grands ensembles de données.

Le moyen le plus simple pour supprimer d'anciennes données est de supprimer la partition qui n'est plus nécessaire :

```
DROP TABLE measurement_y2006m02;
```

Cela peut supprimer des millions d'enregistrements très rapidement, car il n'est pas nécessaire de supprimer chaque enregistrement séparément. Veuillez noter toutefois que la commande ci-dessus nécessite de prendre un verrou de type ACCESS EXCLUSIVE sur la table parente.

Une autre possibilité, généralement préférable, est de ne pas supprimer la partition de la table partitionnée, mais de la conserver en tant que table à part entière. Cela peut prendre deux formes :

```
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02;
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02
 CONCURRENTLY;
```

Elles permettent d'effectuer ensuite d'autres opérations sur les données avant la suppression. Par exemple, c'est souvent le moment idéal pour sauvegarder les données en utilisant COPY, pg_dump, ou des outils similaires. Ce peut aussi être le bon moment pour agréger les données dans un format moins volumineux, effectuer d'autres manipulations des données, ou lancer des rapports. La première forme de la commande requiert un verrou ACCESS EXCLUSIVE sur la table parent. Ajouter la clause CONCURRENTLY, comme indiqué dans la deuxième forme, permet que l'opération de détachement ne réclame qu'un verrou SHARE UPDATE EXCLUSIVE sur la table parent. Cependant, voir ALTER TABLE ... DETACH PARTITION pour des détails sur les restrictions.

De la même manière, nous pouvons ajouter une nouvelle partition pour gérer les nouvelles données. Nous pouvons créer une partition vide dans la table partitionnée exactement comme les partitions originales ont été créées précédemment :

```
CREATE TABLE measurement_y2008m02 PARTITION OF measurement
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')
    TABLESPACE fasttablespace;
```

De manière alternative, il est parfois plus pratique de créer la nouvelle table en dehors de la structure partitionnée, et de l'attacher comme partition plus tard. Cela permet de charger de nouvelles données, de les vérifier et d'y effectuer des transformations avant que les données apparaissent dans la table partitionnée. De plus, l'opération ATTACH PARTITION ne requiert que le verrou SHARE UPDATE EXCLUSIVE sur la table partitionnée contrairement au verrou ACCESS EXCLUSIVE qui est nécessaire pour CREATE TABLE ... PARTITION OF, donc cette opération est préférable sur la table partitionnée. L'option CREATE TABLE ... LIKE est utile pour éviter de répéter à chaque fois la définition de la table parent :

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
  TABLESPACE fasttablespace;

ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE
 '2008-03-01' );

\copy measurement_y2008m02 from 'measurement_y2008m02'
-- et éventuellement d'autres étapes de préparation

ALTER TABLE measurement ATTACH PARTITION measurement_y2008m02
    FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );
```

Avant d'exécuter une commande ATTACH PARTITION, il est recommandé de créer une contrainte CHECK sur la table qui doit être attachée correspondant à la contrainte de la partition désirée, comme indiquée ci-dessus. De cette manière, le système pourra ignorer le parcours de table qui est

habituellement nécessaire pour valider la contrainte implicite de partition. Sans la contrainte CHECK, la table sera parcourue pour valider la contrainte de partition, alors qu'elle aura pris un verrou ACCESS EXCLUSIVE sur cette partition. Il est recommandé de supprimer la contrainte CHECK redondante après la fin de la commande ATTACH PARTITION. Si la table en cours d'attachement est elle-même une table partitionnée, alors chacune de ses sous-partitions sera verrouillée récursivement et parcourue jusqu'à ce qu'une contrainte CHECK convenable soit rencontrée ou que les partitions feuilles sont atteintes.

De façon similaire, si la table partitionnée a une partition par défaut (DEFAULT), il est recommandé de créer une contrainte CHECK qui exclut la contrainte de la partition à attacher. Si cela n'est pas fait, alors la partition DEFAULT sera parcourue pour vérifier qu'elle ne contienne aucun enregistrement qui devrait être placé dans la partition en cours d'attachement. Cette opération sera réalisée en détenant un verrou ACCESS EXCLUSIVE sur la partition par défaut. Si la partition par défaut est elle-même une table partitionnées, alors chacune de ses partitions sera vérifiée récursivement de la même façon que la table en cours d'attachement, comme indiqué ci-dessus.

Comme expliqué ci-dessus, il est possible de créer des index sur des tables partitionnées, pour qu'ils soient appliqués automatiquement sur la hiérarchie entière. C'est très pratique, car non seulement les partitions déjà créées seront indexées, mais aussi toutes les partitions créées dans le futur le seront. Une limitation est qu'il n'est pas possible d'utiliser CONCURRENTLY pour créer un index partitionné. Pour éviter des verrous trop longs, il est possible d'utiliser CREATE INDEX ON ONLY sur la table partitionnée ; un tel index sera marqué invalide, et il ne sera pas appliqué automatiquement sur les partitions. Les index sur les partitions peuvent être créés individuellement avec CONCURRENTLY, et plus tard rattachés à l'index sur le parent avec ALTER INDEX .. ATTACH PARTITION. Une fois les index de toutes les partitions attachés à l'index parent, celui-ci sera automatiquement marqué comme valide. Exemple :

```
CREATE INDEX measurement_usls_idx ON ONLY measurement (unitsales);

CREATE INDEX CONCURRENTLY measurement_usls_200602_idx
    ON measurement_y2006m02 (unitsales);
ALTER INDEX measurement_usls_idx
    ATTACH PARTITION measurement_usls_200602_idx;
...
```

Cette technique peut aussi être utilisée avec des contraintes UNIQUE et PRIMARY KEY ; les index sont créés implicitement quand la contrainte est créée. Exemple :

```
ALTER TABLE ONLY measurement ADD UNIQUE (city_id, logdate);

ALTER TABLE measurement_y2006m02 ADD UNIQUE (city_id, logdate);
ALTER INDEX measurement_city_id_logdate_key
    ATTACH PARTITION measurement_y2006m02_city_id_logdate_key;
...
```

5.11.2.3. Limitations

Les limitations suivantes s'appliquent aux tables partitionnées :

- Pour créer une contrainte d'unicité ou une clé primaire sur une table partitionnée, la clé de partitionnement ne doit pas inclure des expressions ou des appels de fonction, et les colonnes de la contrainte doivent inclure toutes les colonnes de la clé de partitionnement. Cette limitation existe parce que les index individuels créant la contrainte peuvent seulement forcer l'unicité sur leur propre partition ; de ce fait, la structure même de la partition doit garantir qu'il n'existe pas de duplicats dans les différentes partitions.

- Il n'existe aucun moyen de créer une contrainte d'exclusion sur toute la table partitionnée. Il est seulement possible de placer une telle contrainte sur chaque partition individuellement. Cette limitation vient là-aussi de l'impossibilité de fixer les restrictions entre partitions.

- Les triggers BEFORE ROW ne peuvent pas changer la partition de destination d'une nouvelle ligne.

- Mélanger des relations temporaires et permanentes dans la même arborescence de partitions n'est pas autorisé. Par conséquent, si une table partitionnée est permanente, ses partitions doivent l'être aussi ; de même si la table partitionnée est temporaire, ses partitions doivent l'être aussi. Lors de l'utilisation de relations temporaires, tous les membres de l'arborescence des partitions doivent être issus de la même session.

- Les colonnes générées sont ignorées par la réplication logique.

Les partitions individuelles sont liées à leur table partitionnée en utilisant l'héritage en arrière plan. Néanmoins, il n'est pas possible d'utiliser toutes les fonctionnalités génériques de l'héritage avec les tables en partitionnement déclaratif et leurs partitions, comme indiqué ci-dessous. Notamment, une partition ne peut pas avoir d'autres parents que leur table partitionnée. Une table ne peut pas non plus hériter d'une table partitionnée et d'une table normale. Cela signifie que les tables partitionnées et leur partitions ne partagent jamais une hiérarchie d'héritage avec des tables normales.

Comme une hiérarchie de partitionnement consistant en la table partitionnée et ses partitions est toujours une hiérarchie d'héritage, tableoid et toutes les règles normales d'héritage s'appliquent comme décrites dans Section 5.10, avec quelques exceptions :

- Les partitions ne peuvent pas avoir des colonnes qui ne sont pas présentes chez le parent. Il n'est pas possible d'indiquer des colonnes lors de la création de partitions avec CREATE TABLE, pas plus qu'il n'est possible d'ajouter des colonnes aux partitions après leur création en utilisant ALTER TABLE. Les tables pourraient être ajoutées en tant que partition avec ALTER TABLE ... ATTACH PARTITION seulement si leurs colonnes correspondent exactement à leur parent, en incluant toute colonne oid.

- Les contraintes CHECK et NOT NULL d'une table partitionnée sont toujours héritées par toutes ses partitions. La création des contraintes CHECK marquées NO INHERIT n'est pas autorisée sur les tables partitionnées. Vous ne pouvez pas supprimer une contrainte NOT NULL de la colonne d'une partition si la même contrainte est présente dans la table parent.

- Utiliser ONLY pour ajouter ou supprimer une contrainte uniquement sur la table partitionnée est supportée tant qu'il n'y a pas de partitions. Dès qu'une partition existe, utiliser ONLY renverra une erreur pour toute contrainte autre que UNIQUE et PRIMARY KEY. À la place, des contraintes sur les partitions elles-mêmes peuvent être ajoutées et (si elles ne sont pas présentes sur la table parent) supprimées.

- Comme une table partitionnée n'a pas de données elle-même, toute tentative d'utiliser TRUNCATE ONLY sur une table partitionnée renverra systématiquement une erreur.

5.11.3. Partitionnement utilisant l'héritage

Bien que le partitionnement déclaratif natif soit adapté pour la plupart des cas d'usage courant, il reste certains cas où une approche plus flexible peut être utile. Le partitionnement peut être implémenté en utilisant l'héritage de table, ce qui permet d'autres fonctionnalités non supportées par le partitionnement déclaratif, comme :

- Pour le partitionnement déclaratif, les partitions doivent avoir exactement les mêmes colonnes que la table partitionnée, alors qu'avec l'héritage de table, les tables filles peuvent avoir des colonnes supplémentaires non présentes dans la table parente.

- L'héritage de table permet l'héritage multiple.

- Le partitionnement déclaratif ne prend en charge que le partitionnement par intervalle, par liste et par hachage, tandis que l'héritage de table permet de diviser les données de la manière choisie par l'utilisateur. (Notez, cependant, que si l'exclusion de contrainte n'est pas en mesure d'élaguer efficacement les tables filles, la performance de la requête peut être faible).

5.11.3.1. Exemple

Cet exemple construit une structure de partitionnement équivalente à l'exemple de partitionnement déclaratif ci-dessus. Procédez aux étapes suivantes :

1. Créez la table « mère » (ou racine, ou principale), de laquelle toutes les tables « filles » hériteront. Cette table ne contiendra aucune donnée. Ne définissez aucune contrainte de vérification sur cette table, à moins que vous n'ayez l'intention de l'appliquer de manière identique sur toutes les tables filles. Il n'y a aucun intérêt à définir d'index ou de contrainte unique sur elle non plus. Pour notre exemple, la table mère correspond à la table `measurement` définie à l'origine :

   ```
   CREATE TABLE measurement (
       city_id         int not null,
       logdate         date not null,
       peaktemp        int,
       unitsales       int
   );
   ```
2. Créez plusieurs tables « filles », chacune héritant de la table mère. Normalement, ces tables n'ajouteront aucune colonne à celles héritées de la table mère. Comme avec le partitionnement déclaratif, ces tables filles sont des tables PostgreSQL à part entière (ou des tables distantes).

   ```
   CREATE TABLE measurement_y2006m02 () INHERITS (measurement);
   CREATE TABLE measurement_y2006m03 () INHERITS (measurement);
   ...
   CREATE TABLE measurement_y2007m11 () INHERITS (measurement);
   CREATE TABLE measurement_y2007m12 () INHERITS (measurement);
   CREATE TABLE measurement_y2008m01 () INHERITS (measurement);
   ```
3. Ajoutez les contraintes de tables, sans qu'elles se chevauchent, sur les tables filles pour définir les valeurs de clé autorisées dans chacune.

   Des exemples typiques seraient :

   ```
   CHECK ( x = 1 )
   CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire',
    'Warwickshire' ))
   CHECK ( outletID >= 100 AND outletID < 200 )
   ```

   Assurez-vous que les contraintes garantissent qu'il n'y a pas de chevauchement entre les valeurs de clés permises dans différentes tables filles. Une erreur fréquente est de mettre en place des contraintes d'intervalle comme ceci :

   ```
   CHECK ( outletID BETWEEN 100 AND 200 )
   CHECK ( outletID BETWEEN 200 AND 300 )
   ```

   Cet exemple est faux puisqu'il n'est pas possible de savoir à quelle table fille appartient la valeur de clé 200. À la place, les intervalles devraient être définis ainsi :

   ```
   CREATE TABLE measurement_y2006m02 (
       CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE
    '2006-03-01' )
   ) INHERITS (measurement);

   CREATE TABLE measurement_y2006m03 (
       CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE
    '2006-04-01' )
   ) INHERITS (measurement);

   ...
   ```

```
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE
 '2007-12-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2007m12 (
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE
 '2008-01-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE
 '2008-02-01' )
) INHERITS (measurement);
```

4. Pour chaque table fille, créez un index sur la ou les colonnes de la clé, ainsi que tout autre index que vous voudriez.

```
CREATE INDEX measurement_y2006m02_logdate ON
 measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON
 measurement_y2006m03 (logdate);
CREATE INDEX measurement_y2007m11_logdate ON
 measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON
 measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON
 measurement_y2008m01 (logdate);
```

5. Nous voulons que notre application soit capable de dire `INSERT INTO measurement ...`, et de voir ses données redirigées dans la table fille appropriée. Nous pouvons réaliser cela en ajoutant un trigger sur la table mère. Si les données doivent être ajoutées sur la dernière table fille uniquement, nous pouvons utiliser un trigger avec une fonction très simple :

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

Après avoir créé la fonction, nous créons le trigger qui appelle la fonction trigger :

```
CREATE TRIGGER insert_mesure_trigger
    BEFORE INSERT ON mesure
    FOR EACH ROW EXECUTE FUNCTION mesure_insert_trigger();
```

Une telle fonction doit être redéfinie chaque mois pour toujours insérer sur la table fille active. La définition du trigger n'a pas besoin d'être redéfinie.

Il est également possible de laisser le serveur localiser la table fille dans laquelle doit être insérée la ligne. Une fonction plus complexe peut alors être utilisée :

```
CREATE OR REPLACE FUNCTION mesure_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF ( NEW.date_trace >= DATE '2006-02-01' AND
         NEW.date_trace < DATE '2006-03-01' ) THEN
        INSERT INTO mesure_a2006m02 VALUES (NEW.*);
```

```
    ELSIF ( NEW.date_trace >= DATE '2006-03-01' AND
            NEW.date_trace < DATE '2006-04-01' ) THEN
        INSERT INTO mesure_a2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.date_trace >= DATE '2008-01-01' AND
            NEW.date_trace < DATE '2008-02-01' ) THEN
        INSERT INTO mesure_a2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date en dehors de l''intervalle.
 Corrigez la fonction mesure_insert_trigger() !';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

La définition du trigger est la même qu'avant. Notez que chaque test `IF` doit correspondre exactement à la contrainte `CHECK` de la table fille correspondante.

Bien que cette fonction soit plus complexe que celle pour un seul mois, il n'est pas nécessaire de l'actualiser aussi fréquemment, les branches pouvant être ajoutées en avance.

> **Note**
>
> En pratique, il vaudrait mieux vérifier d'abord la dernière table fille créée si la plupart des insertions lui sont destinées. Pour des raisons de simplicité, les tests du trigger sont présentés dans le même ordre que les autres parties de l'exemple.

Une approche différente du trigger est la redirection des insertions par des règles sur la table mère. Par exemple :

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE
 '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE
 '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

Une règle a un surcoût bien plus important qu'un trigger, mais il n'est payé qu'une fois par requête plutôt qu'une fois par ligne. Cette méthode peut donc être avantageuse pour les insertions en masse. Toutefois, dans la plupart des cas, la méthode du trigger offrira de meilleures performances.

Soyez conscient que `COPY` ignore les règles. Si vous voulez utiliser `COPY` pour insérer des données, vous devrez les copier dans la bonne table fille plutôt que dans la table mère. `COPY` déclenche les triggers, vous pouvez donc l'utiliser normalement si vous utilisez l'approche par trigger.

Un autre inconvénient à l'approche par règle est qu'il n'y a pas de moyen simple de forcer une erreur si l'ensemble de règles ne couvre pas la date d'insertion ; les données iront silencieusement dans la table mère à la place.

6. Assurez-vous que le paramètre de configuration constraint_exclusion ne soit pas désactivé dans le fichier `postgresql.conf` ; sinon il pourrait y avoir des accès inutiles aux autres tables.

Comme nous pouvons le voir, une hiérarchie complexe de tables peut nécessiter une quantité de DDL non négligeable. Dans l'exemple ci-dessus, nous créerions une nouvelle table fille chaque mois, il serait donc sage d'écrire un script qui génère le DDL automatiquement.

5.11.3.2. Maintenance du partitionnement par héritage

Pour supprimer les anciennes données rapidement, il suffit de supprimer la table fille qui n'est plus nécessaire :

```
DROP TABLE measurement_a2006m02;
```

Pour enlever une table fille de la hiérarchie d'héritage, mais en en gardant l'accès en tant que table normale :

```
ALTER TABLE measurement_a2006m02 NO INHERIT measurement;
```

Pour ajouter une nouvelle table fille pour gérer les nouvelles données, créez une table fille vide, tout comme les tables filles originales ont été créées ci-dessus :

```
CREATE TABLE measurement_a2008m02 (
    CHECK ( date_trace >= DATE '2008-02-01' AND date_trace < DATE
 '2008-03-01' )
) INHERITS (measurement);
```

Une autre alternative est de créer et de remplir la nouvelle table enfant avant de l'ajouter à la hiérarchie de la table. Ceci permet aux données d'être chargées, vérifiées et transformées avant d'être rendues visibles aux requêtes sur la table parente.

```
CREATE TABLE measurement_a2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_a2008m02 ADD CONSTRAINT y2008m02
   CHECK ( date_trace >= DATE '2008-02-01' AND date_trace < DATE
 '2008-03-01' );
\copy measurement_a2008m02 from 'measurement_a2008m02'
-- quelques travaux de préparation des données
ALTER TABLE measurement_a2008m02 INHERIT measurement;
```

5.11.3.3. Restrictions

Les restrictions suivantes s'appliquent au partitionnement par héritage :

- Il n'existe pas de moyen automatique de vérifier que toutes les contraintes de vérification (`CHECK`) sont mutuellement exclusives. Il est plus sûr de créer un code qui fabrique les tables filles, et crée et/ou modifie les objets associés plutôt que de les créer manuellement ;

- Les contraintes d'index et de clés étrangères s'appliquent à des tables seules et non à leurs enfants par héritage, il y a donc des limitations à connaître.

- Les schémas montrés ici supposent que les colonnes clés du partitionnement d'une ligne ne changent jamais ou, tout du moins, ne changent pas suffisamment pour nécessiter un déplacement vers une autre partition. Une commande `UPDATE` qui tentera de le faire échouera à cause des contraintes `CHECK`. Si vous devez gérer ce type de cas, des triggers sur mise à jour peuvent être placés sur les tables filles, mais cela rend la gestion de la structure beaucoup plus complexe.

- Si `VACUUM` ou `ANALYZE` sont lancés manuellement, n'oubliez pas de les lancer sur chaque table fille. Une commande comme :

```
ANALYZE measurement;
```

ne traitera que la table mère.

- Les commandes `INSERT` avec des clauses `ON CONFLICT` ont peu de chances de fonctionner comme attendu, puisque l'action du `ON CONFLICT` n'est effectuée que dans le cas de violations d'unicité dans la table cible, pas dans les filles.
- Des triggers ou des règles seront nécessaires pour rediriger les lignes vers la table fille voulue, à moins que l'application ne soit explicitement au courant du schéma de partitionnement. Les triggers peuvent être compliqués à écrire, et seront bien plus lents que la redirection de ligne effectuée en interne par le partitionnement déclaratif.

5.11.4. Élagage de partition

L'*élagage des partitions* (`Partition pruning`) est une technique d'optimisation des requêtes qui vise à améliorer les performances des tables à partitionnement déclaratif. À titre d'exemple :

```
SET enable_partition_pruning = on;                 -- défaut
SELECT count(*) FROM measurement WHERE date_trace >= DATE
 '2008-01-01';
```

Sans l'élagage de partition, la requête ci-dessus parcourrait chacune des partitions de la table `measurement`. Avec l'élagage de partition activé, le planificateur examinera la définition de chaque partition, et montrera qu'il n'est pas nécessaire de la parcourir puisqu'elle ne contient aucune ligne respectant la clause WHERE de la requête. Lorsque le planificateur peut l'établir, il exclut (élague) la partition du plan de recherche.

En utilisant la commande `EXPLAIN` et le paramètre de configuration enable_partition_pruning, il est possible de voir la différence entre un plan pour lequel des partitions ont été élaguées et celui pour lequel elles ne l'ont pas été. Un plan typique non optimisé pour ce type de configuration de table serait :

```
SET enable_partition_pruning = off;
EXPLAIN SELECT count(*) FROM measurement WHERE date_trace >= DATE
 '2008-01-01';

                                     QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=188.76..188.77 rows=1 width=8)
   ->  Append  (cost=0.00..181.05 rows=3085 width=0)
         ->  Seq Scan on measurement_y2006m02  (cost=0.00..33.12
 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03  (cost=0.00..33.12
 rows=617 width=0)
                Filter: (logdate >= '2008-01-01'::date)
 ...
         ->  Seq Scan on measurement_y2007m11  (cost=0.00..33.12
 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2007m12  (cost=0.00..33.12
 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12
 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

Quelques partitions, voire toutes, peuvent utiliser des parcours d'index à la place des parcours séquentiels de la table complète, mais le fait est qu'il n'est pas besoin de parcourir les plus vieilles

partitions pour répondre à cette requête. Lorsque l'élagage de partitions est activé, nous obtenons un plan significativement moins coûteux, pour le même résultat :

```
SET enable_partition_pruning = on;
EXPLAIN SELECT count(*) FROM measurement WHERE date_trace >= DATE
 '2008-01-01';

                                       QUERY PLAN
-------------------------------------------------------------------------------
 Aggregate  (cost=37.75..37.76 rows=1 width=8)
   ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617
 width=0)
         Filter: (logdate >= '2008-01-01'::date)
```

Il est à noter que l'élagage des partitions n'est piloté que par les contraintes définies implicitement par les clés de partition, et non par la présence d'index. Il n'est donc pas nécessaire de définir des index sur les colonnes clés. Pour savoir si un index doit être créé sur une partition donnée, il vous faut juger si les requêtes sur cette partition en parcourent généralement une grande partie, ou seulement une petite. Un index sera utile dans ce dernier cas, mais pas dans le premier.

L'élagage des partitions peut être effectué non seulement lors de la planification d'une requête, mais aussi lors de son exécution. C'est utile pour élaguer plus de partitions lorsque les clauses contiennent des expressions de valeurs inconnues au moment de la planification de la requête, par exemple des paramètres définis dans une instruction PREPARE, utilisant une valeur obtenue d'une sous-requête, ou une valeur paramétrée sur la partie interne d'une jointure en boucle imbriquée (`nested loop join`). L'élagage de partition pendant l'exécution peut être réalisé à l'un des moments suivant :

- Lors de l'initialisation du plan d'exécution. L'élagage de partition peut être effectué pour les valeurs de paramètres connues dès cette phase. Les partitions élaguées pendant cette étape n'apparaîtront pas dans l'EXPLAIN ou l'EXPLAIN ANALYZE de la requête. Il est même possible de déterminer le nombre de partitions supprimées pendant cette phase en observant la propriété « Subplans Removed »(sous-plans supprimés) dans la sortie d'EXPLAIN. Il est important de noter que toute partition supprimée par l'élagage fait à cette étape est toujours verrouillée au début de l'exécution.

- Pendant l'exécution effective du plan d'exécution. L'élagage des partitions peut également être effectué pour supprimer des partitions en utilisant des valeurs qui ne sont connues que pendant l'exécution de la requête. Cela inclut les valeurs des sous-requêtes et des paramètres issus de l'exécution, comme des jointures par boucle imbriquée(`nested loop join`) paramétrées. Comme la valeur de ces paramètres peut changer plusieurs fois pendant l'exécution de la requête, l'élagage de partitions est effectué chaque fois que l'un des paramètres d'exécution utilisés pour celui-ci change. Déterminer si les partitions ont été élaguées pendant cette phase nécessite une inspection minutieuse de la propriété `loops` de la sortie d'EXPLAIN ANALYZE. Les sous-plans correspondant aux différentes partitions peuvent avoir différentes valeurs dépendant du nombre de fois où chacun d'eux a été élagué lors de l'exécution. Certains peuvent être affichés comme `(never executed)` (littéralement, `jamais exécuté`) s'ils sont élagués à chaque fois.

L'élagage des partitions peut être désactivé à l'aide du paramètre enable_partition_pruning.

5.11.5. Partitionnement et contrainte d'exclusion

Une *contrainte d'exclusion* est une technique d'optimisation de requêtes similaire à l'élagage de partitions. Bien qu'elle soit principalement utilisée pour les tables partitionnées avec l'ancienne méthode par héritage, elle peut être utilisée à d'autres fins, y compris avec le partitionnement déclaratif.

Les contraintes d'exclusion fonctionnent d'une manière très similaire à l'élagage de partitions, sauf qu'elles utilisent les contraintes CHECK de chaque table (d'où le nom) alors que l'élagage de partition utilise les limites de partition de la table, qui n'existent que dans le cas d'un partitionnement déclaratif. Une autre différence est qu'une contrainte d'exclusion n'est appliquée qu'à la planification ; il n'y a donc pas de tentative d'écarter des partitions à l'exécution.

Le fait que les contraintes d'exclusion utilisent les contraintes `CHECK` les rend plus lentes que l'élagage de partitions, mais peut être un avantage : puisque les contraintes peuvent être définies même sur des tables avec partitionnement déclaratif, en plus de leurs limites internes, les contraintes d'exclusion peuvent être capables de supprimer des partitions supplémentaires pendant la phase de planification de la requête.

La valeur par défaut (et donc recommandée) de constraint_exclusion n'est ni `on` ni `off`, mais un état intermédiaire appelé `partition`, qui fait que la technique n'est appliquée qu'aux requêtes qui semblent fonctionner avec des tables partitionnées par héritage. La valeur `on` entraîne que le planificateur examine les contraintes `CHECK` dans toutes les requêtes, y compris les requêtes simples qui ont peu de chance d'en profiter.

Les avertissement suivants s'appliquent à l'exclusion de contraintes :

- Les contraintes d'exclusion ne sont appliquées que lors de la phase de planification de la requête, contrairement à l'élagage de partition, qui peut être appliqué lors de la phase d'exécution.
- La contrainte d'exclusion ne fonctionne que si la clause `WHERE` de la requête contient des constantes (ou des paramètres externes). Par exemple, une comparaison avec une fonction non immutable comme `CURRENT_TIMESTAMP` ne peut pas être optimisée car le planificateur ne peut pas savoir dans quelle table fille la valeur de la fonction ira lors de l'exécution.
- Les contraintes de partitionnement doivent rester simples. Dans le cas contraire, le planificateur peut rencontrer des difficultés à déterminer les tables filles qu'il n'est pas nécessaire de parcourir. Des conditions simples d'égalité pour le partitionnement de liste, ou des tests d'intervalle simples lors de partitionnement par intervalles sont recommandées, comme illustré dans les exemples précédents. Une règle générale est que les contraintes de partitionnement ne doivent contenir que des comparaisons entre les colonnes partitionnées et des constantes, à l'aide d'opérateurs utilisables par les index B-tree, car seules les colonnes indexables avec un index B-tree sont autorisées dans la clé de partitionnement.
- Toutes les contraintes sur toutes les tables filles de la table parente sont examinées lors de l'exclusion de contraintes. De ce fait, un grand nombre de filles augmente considérablement le temps de planification de la requête. Ainsi, l'ancien partitionnement par héritage fonctionnera bien jusqu'à, peut-être, une centaine de tables enfant ; n'essayez pas d'en utiliser plusieurs milliers.

5.11.6. Bonnes pratiques avec le partitionnement déclaratif

Il faut choisir avec soin le partitionnement d'une table car les performances à la planification et à l'exécution peuvent pâtir d'une mauvaise conception.

Un des choix les plus cruciaux portera sur la ou les colonnes par lesquelles vous partitionnerez. Souvent le meilleur choix sera la colonne ou l'ensemble de colonnes qui apparaissent le plus souvent dans les clauses `WHERE` des requêtes exécutées sur la table partitionnée. Les clauses `WHERE` qui sont compatibles avec les contraintes des limites des partitions peuvent être utilisées pour élaguer les partitions inutiles. Cependant, le choix peut vous être imposé par des exigences sur la `PRIMARY KEY` ou une contrainte `UNIQUE`. La suppression de données indésirables est aussi un facteur à considérer pour préparer votre stratégie de partitionnement. Une partition entière peut être détachée assez vite, et cela peut valoir le coup de concevoir votre partitionnement pour que toutes les données à supprimer en même temps soient situées dans la même partition.

Choisir le nombre cible de partitions par lequel diviser la table est aussi une décision critique à prendre. Ne pas avoir assez de partitions peut signifier que les index resteront trop gros, et que la localité des données restera faible, ce qui entraînera de mauvais *hit ratios*. Cependant, diviser la table en trop de partitions a aussi ses inconvénients. Trop de partitions peuvent entraîner des temps de planification plus longs et une plus grande consommation de mémoire pendant la planification comme pendant l'exécution, comme indiqué plus bas. Lors du choix du partitionnement de votre table, il est aussi important de considérer ce qui pourrait changer dans le futur. Par exemple, si vous choisissez

d'avoir une partition par client alors que vous n'avez actuellement qu'un petit nombre de gros clients, considérez les implications si, dans quelques années, vous vous retrouvez avec un grand nombre de petits clients. Dans ce cas, il serait mieux de choisir une partition par `HASH` et de choisir un nombre raisonnable de partitions plutôt que d'essayer de partitionner par `LIST` et d'espérer que le nombre de clients n'augmente pas au-delà de ce qu'il est en pratique possible de partitionner.

Sous-partitionner peut être utile pour diviser encore des partitions qui devraient devenir plus grandes que d'autres partitions. Une autre option est d'utiliser le partitionnement par intervalle avec plusieurs colonnes dans la clé de partitionnement. Chacune de ses solutions peut facilement amener à un nombre excessif de partitions, il convient donc de rester prudent.

Il est important de considérer le surcroît de travail pour la planification et l'exécution dû au partitionnement. Le planificateur de requêtes est généralement capable de manipuler correctement des hiérarchies jusqu'à plusieurs milliers de partitions, pourvu que les requêtes courantes lui permettent d'élaguer toutes les partitions à l'exception d'un petit nombre. Les temps de planification s'allongent et la consommation de mémoire augmente s'il reste beaucoup de partitions une fois que le planificateur a fait l'élagage. Une autre raison de se méfier d'un grand nombre de partitions est que la consommation mémoire du serveur peut augmenter significativement au fil du temps, particulièrement si beaucoup de sessions touchent de nombreuses partitions. La cause en est que chaque partition a besoin que ses métadonnées soient chargées dans la mémoire locale d'une session qui y touche.

Avec une charge de type entrepôt de données, il y a plus de sens à utiliser un grand nombre de partitions que pour une charge de type OLTP. Généralement, en décisionnel, le temps de planification est moins un souci puisque la majorité du temps de traitement est dépensé pendant l'exécution. Avec l'un comme l'autre de ces types de charge, il est important de prendre tôt la bonne décision, car re-partitionner de grandes quantités de données peut être douloureusement long. Des simulation de la charge attendue sont souvent souhaitables pour optimiser la stratégie de partitionnement. Ne supposez jamais que plus de partitions valent mieux que moins de partitions et vice-versa.

5.12. Données distantes

PostgreSQL implémente des portions de la norme SQL/MED, vous permettant d'accéder à des données qui résident en dehors de PostgreSQL en utilisant des requêtes SQL standards. On utilise le terme de *données distantes* pour de telles données. (Notez qu'en anglais il y a ambiguïté : les données distantes (*foreign data*) n'ont rien à voir avec les clés étrangères (*foreign keys*), qui sont un type de contrainte à l'intérieur de la base de données.)

Les données distantes sont accédées grâce à un *wrapper de données distantes*. Ce dernier est une bibliothèque qui peut communiquer avec une source de données externe, cachant les détails de la connexion vers la source de données et de la récupération des données à partir de cette source. Il existe des wrappers de données distantes disponibles en tant que modules `contrib`. D'autres types de wrappers de données distantes peuvent faire partie de produits tiers. Si aucun des wrappers de données distantes ne vous convient, vous pouvez écrire le vôtre. Voir Chapitre 59.

Pour accéder aux données distantes, vous devez créer un objet de type *serveur distant* qui définit la façon de se connecter à une source de données externes particulière suivant un ensemble d'options utilisées par un wrapper de données distantes. Ensuite, vous aurez besoin de créer une ou plusieurs *tables distantes*, qui définissent la structure des données distantes. Une table distante peut être utilisée dans des requêtes comme toute autre table, mais une table distante n'est pas stockée sur le serveur PostgreSQL. À chaque utilisation, PostgreSQL demande au wrapper de données distantes de récupérer les données provenant de la source externe, ou de transmettre les données à la source externe dans le cas de commandes de mise à jour.

Accéder à des données distantes pourrait nécessiter une authentification auprès de la source de données externes. Cette information peut être passée par une *correspondance d'utilisateur*, qui peut fournir des données comme les noms d'utilisateurs et mots de passe en se basant sur le rôle PostgreSQL actuel.

Pour plus d'informations, voir CREATE FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE et IMPORT FOREIGN SCHEMA.

5.13. Autres objets de la base de données

Les tables sont les objets centraux dans une structure de base de données relationnelle, car elles contiennent les données. Mais ce ne sont pas les seuls objets qui existent dans une base de données. De nombreux autres types d'objets peuvent être créés afin de rendre l'utilisation et la gestion des données plus efficace ou pratique. Ils ne sont pas abordés dans ce chapitre, mais une liste en est dressée à titre d'information.

- Vues
- Fonctions, procédures et opérateurs
- Types de données et domaines
- Triggers et règles de réécriture

Des informations détaillées sur ces sujets apparaissent dans la Partie V.

5.14. Gestion des dépendances

Lorsque des structures de base complexes sont créées et qu'ils impliquent beaucoup de tables avec des contraintes de clés étrangères, des vues, des triggers, des fonctions, etc., un réseau de dépendances entre les objets est implicitement créé. Par exemple, une table avec une contrainte de clé étrangère dépend de la table à laquelle elle fait référence.

Pour garantir l'intégrité de la structure entière de la base, PostgreSQL s'assure qu'un objet dont d'autres objets dépendent ne peut pas être supprimé. Ainsi, toute tentative de suppression de la table des produits utilisée dans la Section 5.4.5, sachant que la table des commandes en dépend, lève un message d'erreur comme celui-ci :

```
DROP TABLE produits;

ERROR:  cannot drop table produits because other objects depend on
 it
DETAIL:  constraint commandes_no_produit_fkey on table commandes
 depends on table produits
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

Le message d'erreur contient un indice utile : pour ne pas avoir à supprimer individuellement chaque objet dépendant, on peut lancer

```
DROP TABLE produits CASCADE;
```

et tous les objets dépendants sont ainsi effacés, comme tous les objets dépendant de ces derniers, récursivement. Dans ce cas, la table des commandes n'est pas supprimée, mais seulement la contrainte de clé étrangère. Elle s'arrête là, car rien ne dépend d'une contrainte de clé étrangère. (Pour vérifier ce que fait `DROP ... CASCADE`, on peut lancer `DROP` sans `CASCADE` et lire les messages `DETAIL`.)

Pratiquement toutes les commandes `DROP` dans PostgreSQL supportent l'utilisation de `CASCADE`. La nature des dépendances est évidemment fonction de la nature des objets. On peut aussi écrire `RESTRICT` au lieu de `CASCADE` pour obtenir le comportement par défaut, à savoir interdire les suppressions d'objets dont dépendent d'autres objets.

> **Note**
>
> D'après le standard SQL, il est nécessaire d'indiquer `RESTRICT` ou `CASCADE` dans une commande `DROP`. Aucun système de base de données ne force cette règle, en réalité, mais le choix du comportement par défaut, `RESTRICT` ou `CASCADE`, varie suivant le système.

Si une commande DROP liste plusieurs objets, CASCADE est seulement requis quand il existe des dépendances en dehors du groupe spécifié. Par exemple, en indiquant DROP TABLE tab1, tab2, l'existence d'une clé étrangère référençant tab1 à partir de tab2 ne signifie pas que CASCADE est nécessaire pour réussir.

Pour les fonctions définies par les utilisateurs ou les procédures dont le corps est défini comme une chaîne, PostgreSQL trace les dépendances associées avec les propriétés de la fonction visibles en externe, comme les types de données des arguments et du résultat. Par contre, il ne trace *pas* les dépendances seulement connues en examinant le corps de la fonction. Par exemple :

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',
                             'green', 'blue', 'purple');

CREATE TABLE my_colors (color rainbow, note text);

CREATE FUNCTION get_color_note (rainbow) RETURNS text AS
  'SELECT note FROM my_colors WHERE color = $1'
  LANGUAGE SQL;
```

(Voir Section 38.5 pour une explication sur les fonctions en SQL.) PostgreSQL aura connaissance du fait que la fonction get_color_note dépend du type rainbow : supprimer ce type de données forcera la suppression de la fonction parce que le type de son argument ne serait plus défini. Mais PostgreSQL ne considérera pas que la fonction get_color_note dépende de la table my_colors, et donc ne supprimera pas la fonction si la table est supprimée. Bien qu'il y ait des inconvénients à cette approche, il y a aussi des avantages. La fonction est toujours valide d'une certaine façon si la table est manquante, bien que son exécution causera une erreur. Créer une nouvelle table de même nom permettra à la fonction d'être valide de nouveau.

D'un autre côté, pour une fonction en langage SQL ou une procédure dont le corps est écrit dans un style SQL, le corps est analysé à la création de la fonction et toutes les dépendances reconnues par l'analyseur sont enregistrées. De ce fait, si nous écrivons la fonction ci-dessus ainsi :

```
CREATE FUNCTION get_color_note (rainbow) RETURNS text
BEGIN ATOMIC
  SELECT note FROM my_colors WHERE color = $1;
END;
```

alors la dépendance de la fonction avec la table my_colors sera connu et respecté par une instruction DROP.

Chapitre 6. Manipulation de données

Le chapitre précédent présente la création des tables et des autres structures de stockage des données. Il est temps de remplir ces tables avec des données. Le présent chapitre couvre l'insertion, la mise à jour et la suppression des données des tables. Après cela, le chapitre présente l'élimination des données perdues.

6.1. Insérer des données

Quand une table est créée, elle ne contient aucune donnée. La première chose à faire, c'est d'y insérer des données. Sans quoi la base de données n'est pas d'une grande utilité. Les données sont insérées ligne par ligne. Vous pouvez aussi insérer plus d'une ligne en une seule commande, mais il n'est pas possible d'insérer une ligne partielle. Même lorsque seules les valeurs d'une partie des colonnes sont connues, une ligne complète doit être créée.

Pour créer une nouvelle ligne, la commande INSERT est utilisée. La commande a besoin du nom de la table et des valeurs des colonnes.

Soit la table des produits du Chapitre 5 :

```
CREATE TABLE produits (
    no_produit integer,
    nom text,
    prix numeric
);
```

Une commande d'insertion d'une ligne peut être :

```
INSERT INTO produits VALUES (1, 'Fromage', 9.99);
```

Les données sont listées dans l'ordre des colonnes de la table, séparées par des virgules. Souvent, les données sont des libellés (constantes), mais les expressions scalaires sont aussi acceptées.

La syntaxe précédente oblige à connaître l'ordre des colonnes. Pour éviter cela, les colonnes peuvent être explicitement listées. Les deux commandes suivantes ont, ainsi, le même effet que la précédente :

```
INSERT INTO produits (no_produit, nom, prix) VALUES (1, 'Fromage',
 9.99);
INSERT INTO produits (nom, prix, no_produit) VALUES ('Fromage',
 9.99, 1);
```

Beaucoup d'utilisateurs recommandent de toujours lister les noms de colonnes.

Si les valeurs de certaines colonnes ne sont pas connues, elles peuvent être omises. Dans ce cas, elles sont remplies avec leur valeur par défaut. Par exemple :

```
INSERT INTO produits (no_produit, nom) VALUES (1, 'Fromage');
INSERT INTO produits VALUES (1, 'Fromage');
```

La seconde instruction est une extension PostgreSQL. Elle remplit les colonnes de gauche à droite avec toutes les valeurs données, et les autres prennent leur valeur par défaut.

Il est possible, pour plus de clarté, d'appeler explicitement les valeurs par défaut pour des colonnes particulières ou pour la ligne complète.

```
INSERT INTO produits (no_produit, nom, prix) VALUES (1, 'Fromage',
 DEFAULT);
INSERT INTO produits DEFAULT VALUES;
```

Plusieurs lignes peuvent être insérées en une seule commande :

```
INSERT INTO produits (no_produit, nom, prix) VALUES
    (1, 'Fromage', 9.99),
    (2, 'Pain', 1.99),
    (3, 'Lait', 2.99);
```

Il est aussi possible d'insérer le résultat d'une requête (qui pourrait renvoyer aucune ligne, une ligne ou plusieurs lignes) :

```
INSERT INTO produits (no_produit, nom, prix)
  SELECT no_produit, nom, prix FROM nouveaux_produits
    WHERE date_sortie = 'today';
```

Ceci montre la grande puissance du mécanisme des requêtes SQL (Chapitre 7) sur le traitement des lignes à insérer.

> **Astuce**
>
> Lors de l'insertion d'une grande quantité de données en même temps, il est préférable d'utiliser la commande COPY. Elle n'est pas aussi flexible que la commande INSERT, mais elle est plus efficace. Se référer à Section 14.4 pour plus d'informations sur l'amélioration des performances lors de gros chargements de données.

6.2. Actualiser les données

La modification de données présentes en base est appelée mise à jour ou actualisation (*update* en anglais). Il est possible de mettre à jour une ligne spécifique, toutes les lignes ou un sous-ensemble de lignes de la table. Chaque colonne peut être actualisée séparément ; les autres colonnes ne sont alors pas modifiées.

Pour mettre à jour les lignes existantes, utilisez la commande UPDATE. Trois informations sont nécessaires :

1. le nom de la table et de la colonne à mettre à jour ;
2. la nouvelle valeur de la colonne ;
3. les lignes à mettre à jour.

Comme cela a été vu dans le Chapitre 5, le SQL ne donne pas, par défaut, d'identifiant unique pour les lignes. Il n'est, de ce fait, pas toujours possible d'indiquer directement la ligne à mettre à jour. On précise plutôt les conditions qu'une ligne doit remplir pour être mise à jour. Si la table possède une clé primaire (qu'elle soit déclarée ou non), une ligne unique peut être choisie en précisant une condition sur la clé primaire. Les outils graphiques d'accès aux bases de données utilisent ce principe pour permettre les modifications de lignes individuelles.

La commande suivante, par exemple, modifie tous les produits dont le prix est 5 en le passant à 10.

```
UPDATE produits SET prix = 10 WHERE prix = 5;
```

Cela peut mettre à jour zéro, une, ou plusieurs lignes. L'exécution d'une commande `UPDATE` qui ne met à jour aucune ligne ne représente pas une erreur.

Dans le détail de la commande, on trouve tout d'abord, le mot-clé `UPDATE` suivi du nom de la table. Le nom de la table peut toujours être préfixé par un nom de schéma ; dans le cas contraire, elle est

recherchée dans le chemin. On trouve ensuite le mot-clé SET suivi du nom de la colonne, un signe égal et la nouvelle valeur de la colonne, qui peut être une constante ou une expression scalaire.

Par exemple, pour augmenter de 10% le prix de tous les produits, on peut exécuter :

```
UPDATE produits SET prix = prix * 1.10;
```

L'expression donnant la nouvelle valeur peut faire référence aux valeurs courantes de la ligne.

Il n'a pas été indiqué ici de clause WHERE. Si elle est omise, toutes les lignes de la table sont modifiées. Si elle est présente, seules les lignes qui remplissent la condition WHERE sont mises à jour. Le signe égal dans la clause SET réalise une affectation, alors que celui de la clause WHERE permet une comparaison. Pour autant, cela ne crée pas d'ambiguïté. La condition WHERE n'est pas nécessairement un test d'égalité ; de nombreux autres opérateurs existent (voir le Chapitre 9). Mais le résultat de l'expression est booléen.

Il est possible d'actualiser plusieurs colonnes en une seule commande UPDATE par l'indication de plusieurs colonnes dans la clause SET.

Par exemple :

```
UPDATE ma_table SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. Supprimer des données

Les parties précédentes présentent l'ajout et la modification de données. Il reste à voir leur suppression quand elles ne sont plus nécessaires. Comme pour l'insertion, la suppression ne peut se faire que par ligne entière. Le SQL ne propose pas de moyen d'accéder à une ligne particulière. C'est pourquoi la suppression de lignes se fait en indiquant les conditions à remplir par les lignes à supprimer. S'il y a une clé primaire dans la table, alors il est possible d'indiquer précisément la ligne à supprimer. Mais on peut aussi supprimer un groupe de lignes qui remplissent une condition, ou même toutes les lignes d'une table en une fois.

Pour supprimer des lignes, on utilise la commande DELETE ; la syntaxe est très similaire à la commande UPDATE.

Par exemple, pour supprimer toutes les lignes de la table produits qui ont un prix de 10, on exécute :

```
DELETE FROM produits WHERE prix = 10;
```

En indiquant simplement :

```
DELETE FROM produits;
```

on supprime toutes les lignes de la table. Attention aux mauvaises manipulations !

6.4. Renvoyer des données provenant de lignes modifiées

Parfois, il est intéressant d'obtenir des données de lignes modifiées pendant qu'elles sont manipulées. Les commandes INSERT, UPDATE et DELETE ont toutes une clause RETURNING optionnelle qui le permet. L'utilisation de la clause RETURNING évite l'exécution d'une requête supplémentaire pour coller les données, et est particulièrement intéressante quand il serait difficile d'identifier autrement les lignes modifiées.

Le contenu autorisé d'une clause RETURNING est identique à celui de la liste de sortie d'une commande SELECT (voir Section 7.3). Elle peut contenir les noms des colonnes de la table cible ou des

expressions utilisant ces colonnes. Un raccourci habituel est `RETURNING *`, qui sélectionne toutes les colonnes de la table cible, dans l'ordre de définition.

Avec un `INSERT`, les données disponibles à `RETURNING` sont la ligne qui a été insérée. Ceci n'est pas utile pour les insertions simples, car cela ne fera que répéter les données fournies par le client, mais cela peut devenir très utile si la commande se base sur les valeurs calculées par défaut. Par exemple, lors de l'utilisation d'une colonne `serial` fournissant des identifiants uniques, `RETURNING` peut renvoyer l'identifiant affecté à une nouvelle ligne :

```
CREATE TABLE utilisateurs (prenom text, nom text, id serial primary
 key);

INSERT INTO utilisateurs (prenom, nom) VALUES ('Joe', 'Cool')
 RETURNING id;
```

La clause `RETURNING` est aussi très utile avec un `INSERT ... SELECT`

Dans un `UPDATE`, les données disponibles pour la clause `RETURNING` correspondent au nouveau contenu de la ligne modifiée. Par exemple :

```
UPDATE produits SET prix = prix * 1.10
  WHERE prix <= 99.99
  RETURNING nom, prix AS nouveau_prix;
```

Dans un `DELETE`, les données disponibles pour la clause `RETURNING` correspondent au contenu de la ligne supprimée. Par exemple :

```
DELETE FROM produits
  WHERE date_perime = 'today'
  RETURNING *;
```

Si des triggers (Chapitre 39) sont définis sur la table cible, les données disponibles pour la clause `RETURNING` correspondent à la ligne modifiée par les triggers. De ce fait, une utilisation courante de la clause `RETURNING` est d'inspecter les colonnes calculées par les triggers.

Chapitre 7. Requêtes

Les précédents chapitres ont expliqué comment créer des tables, comment les remplir avec des données et comment manipuler ces données. Maintenant, nous discutons enfin de la façon de récupérer ces données depuis la base de données.

7.1. Aperçu

Le processus et la commande de récupération des données sont appelés une *requête*. En SQL, la commande `SELECT` est utilisée pour spécifier des requêtes. La syntaxe générale de la commande `SELECT` est

```
[WITH with_requêtes] SELECT liste_select FROM expression_table
 [specification_tri]
```

Les sections suivantes décrivent le détail de la liste de sélection, l'expression des tables et la spécification du tri. Les requêtes `WITH` sont traitées en dernier, car il s'agit d'une fonctionnalité avancée.

Un type de requête simple est de la forme :

```
SELECT * FROM table1;
```

En supposant qu'il existe une table appelée `table1`, cette commande récupérera toutes les lignes et toutes les colonnes, définies par l'utilisateur, de `table1`. La méthode de récupération dépend de l'application cliente. Par exemple, le programme psql affichera une table, façon art ASCII, alors que les bibliothèques du client offriront des fonctions d'extraction de valeurs individuelles à partir du résultat de la requête. * comme liste de sélection signifie que toutes les colonnes de l'expression de table seront récupérées. Une liste de sélection peut aussi être un sous-ensemble des colonnes disponibles ou effectuer un calcul en utilisant les colonnes. Par exemple, si `table1` dispose des colonnes nommées `a`, `b` et `c` (et peut-être d'autres), vous pouvez lancer la requête suivante :

```
SELECT a, b + c FROM table1;
```

(en supposant que `b` et `c` soient de type numérique). Voir la Section 7.3 pour plus de détails.

`FROM table1` est un type très simple d'expression de tables : il lit une seule table. En général, les expressions de tables sont des constructions complexes de tables de base, de jointures et de sous-requêtes. Mais vous pouvez aussi entièrement omettre l'expression de table et utiliser la commande `SELECT` comme une calculatrice :

```
SELECT 3 * 4;
```

Ceci est plus utile si les expressions de la liste de sélection renvoient des résultats variants. Par exemple, vous pouvez appeler une fonction de cette façon :

```
SELECT random();
```

7.2. Expressions de table

Une *expression de table* calcule une table. L'expression de table contient une clause `FROM` qui peut être suivie des clauses `WHERE`, `GROUP BY` et `HAVING`. Les expressions triviales de table font simplement référence à une table sur le disque, une table de base, mais des expressions plus complexes peuvent être utilisées pour modifier ou combiner des tables de base de différentes façons.

Les clauses optionnelles `WHERE`, `GROUP BY` et `HAVING` dans l'expression de table spécifient un tube de transformations successives réalisées sur la table dérivée de la clause `FROM`. Toutes ces transformations produisent une table virtuelle fournissant les lignes à passer à la liste de sélection qui choisira les lignes à afficher de la requête.

7.2.1. Clause `FROM`

La clause `FROM` dérive une table à partir d'une ou plusieurs tables données dans une liste de référence dont les tables sont séparées par des virgules.

```
FROM reference_table [, reference_table [, ...]]
```

Une référence de table pourrait être un nom de table (avec en option le nom du schéma) ou de table dérivée, telle qu'une sous-requête, une construction `JOIN` ou une combinaison complexe de ces possibilités. Si plus d'une référence de table est listée dans la clause `FROM`, les tables sont jointes en croisé (autrement dit, cela réalise un produit cartésien de leurs lignes ; voir ci-dessous). Le résultat de la liste `FROM` est une table virtuelle intermédiaire pouvant être sujette aux transformations des clauses `WHERE`, `GROUP BY` et `HAVING`, et est finalement le résultat des expressions de table.

Lorsqu'une référence de table nomme une table qui est la table parent d'une table suivant la hiérarchie de l'héritage, la référence de table produit les lignes non seulement de la table, mais aussi des descendants de cette table, sauf si le mot-clé `ONLY` précède le nom de la table. Néanmoins, la référence produit seulement les colonnes qui apparaissent dans la table nommée... Toute colonne ajoutée dans une sous-table est ignorée.

Au lieu d'écrire `ONLY` avant le nom de la table, vous pouvez écrire `*` après le nom de la table pour indiquer spécifiquement que les tables filles sont incluses. Il n'y a plus de vraie raison pour encore utiliser cette syntaxe, car chercher dans les tables descendantes est maintenant le comportement par défaut. C'est toutefois supporté pour compatibilité avec des versions plus anciennes.

7.2.1.1. Tables jointes

Une table jointe est une table dérivée de deux autres tables (réelles ou dérivées) suivant les règles du type de jointure particulier. Les jointures internes (inner), externes (outer) et croisées (cross) sont disponibles. La syntaxe générale d'une table jointe est :

```
T1 type_jointure T2 [ condition_jointure ]
```

Des jointures de tous types peuvent être chaînées ensemble ou imbriquées : une des deux tables ou les deux tables peuvent être des tables jointes. Des parenthèses peuvent être utilisées autour des clauses `JOIN` pour contrôler l'ordre de jointure. Dans l'absence des parenthèses, les clauses `JOIN` s'imbriquent de gauche à droite.

Types de jointures

Jointure croisée (cross join)

```
T1 CROSS JOIN T2
```

Pour chaque combinaison possible de lignes provenant de *`T1`* et *`T2`* (c'est-à-dire un produit cartésien), la table jointe contiendra une ligne disposant de toutes les colonnes de *`T1`* suivies par toutes les colonnes de *`T2`*. Si les tables ont respectivement N et M lignes, la table jointe en aura N * M.

`FROM` *`T1`* `CROSS JOIN` *`T2`* est équivalent à `FROM` *`T1`* `INNER JOIN` *`T2`* `ON TRUE` (voir ci-dessous). C'est aussi équivalent à : `FROM` *`T1`*`,` *`T2`*.

> **Note**
>
> Cette dernière équivalence ne convient pas exactement quand plusieurs tables apparaissent, car `JOIN` lie de façon plus profonde que la virgule. Par exemple, `FROM T1 CROSS JOIN T2 INNER JOIN T3 ON condition` n'est pas identique à `FROM T1, T2 INNER JOIN T3 ON condition`, car `condition` peut faire référence à `T1` dans le premier cas, mais pas dans le second.

Jointures qualifiées (qualified joins)

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
 ON expression_booleenne
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING
 ( liste des colonnes jointes )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

Les mots `INNER` et `OUTER` sont optionnels dans toutes les formes. `INNER` est la valeur par défaut ; `LEFT`, `RIGHT` et `FULL` impliquent une jointure externe.

La *condition de la jointure* est spécifiée dans la clause `ON` ou `USING`, ou implicitement par le mot `NATURAL`. La condition de jointure détermine les lignes des deux tables sources considérées comme « correspondante », comme l'explique le paragraphe ci-dessous.

Les types possibles de jointures qualifiées sont :

`INNER JOIN`

Pour chaque ligne R1 de T1, la table jointe a une ligne pour chaque ligne de T2 satisfaisant la condition de jointure avec R1.

`LEFT OUTER JOIN`

Tout d'abord, une jointure interne est réalisée. Puis, pour chaque ligne de T1 qui ne satisfait pas la condition de jointure avec les lignes de T2, une ligne jointe est ajoutée avec des valeurs NULL dans les colonnes de T2. Du coup, la table jointe a toujours au moins une ligne pour chaque ligne de T1, quelles que soient les conditions.

`RIGHT OUTER JOIN`

Tout d'abord, une jointure interne est réalisée. Puis, pour chaque ligne de T2 qui ne satisfait pas la condition de jointure avec les lignes de T1, une ligne jointe est ajoutée avec des valeurs NULL dans les colonnes de T1. C'est l'inverse d'une jointure gauche : la table résultante aura toujours une ligne pour chaque ligne de T2, quelles que soient les conditions.

`FULL OUTER JOIN`

Tout d'abord, une jointure interne est réalisée. Puis, pour chaque ligne de T1 qui ne satisfait pas la condition de jointure avec les lignes de T2, une ligne jointe est ajoutée avec des valeurs NULL dans les colonnes de T2. De plus, pour chaque ligne de T2 qui ne satisfait pas la condition de jointure avec les lignes de T1, une ligne jointe est ajoutée avec des valeurs NULL dans les colonnes de T1.

La clause `ON` est le type de condition de jointure le plus utilisé : elle prend une valeur booléenne du même type que celle utilisée dans une clause `WHERE`. Une paire de lignes provenant de *`T1`* et de *`T2`* correspondent si l'expression de la clause `ON` vaut true.

La clause `USING` est un raccourci qui vous permet de prendre avantage d'une situation spécifique où les deux côtés de la jointure utilisent le même nom pour la colonne jointe. Elle prend une liste de noms de colonnes partagées, en séparant les noms par des virgules et forme une condition de

jointure qui inclut une comparaison d'égalité entre chaque. Par exemple, joindre *T1* et *T2* avec `USING (a, b)` produit la même condition de jointure que la condition `ON T1.a = T2.a AND T1.b = T2.b`.

De plus, la sortie de `JOIN USING` supprime les colonnes redondantes : il n'est pas nécessaire d'imprimer les colonnes de correspondance, puisqu'elles doivent avoir des valeurs identiques. Alors que `JOIN ON` produit toutes les colonnes de *T2*, `JOIN USING` produit une seule colonne pour chaque paire de colonnes listées (dans l'ordre listé), suivi par chaque colonne restante provenant de *T1*, suivi par chaque colonne restante provenant de *T2*.

Enfin, `NATURAL` est un raccourci de `USING` : il forme une liste `USING` constituée de tous les noms de colonnes apparaissant dans les deux tables en entrée. Comme avec `USING`, ces colonnes apparaissent une fois seulement dans la table en sortie. S'il n'existe aucun nom commun de colonne, `NATURAL JOIN` se comporte comme `JOIN ... ON TRUE` et produit une jointure croisée.

> **Note**
>
> `USING` est raisonnablement protégé contre les changements de colonnes dans les relations jointes, car seuls les noms de colonnes listés sont combinés. `NATURAL` est considéré comme plus risqué, car toute modification de schéma causant l'apparition d'un nouveau nom de colonne correspondant fera en sorte de joindre la nouvelle colonne.

Pour rassembler tout ceci, supposons que nous avons une table `t1` :

```
 no | nom
----+------
  1 | a
  2 | b
  3 | c
```

et une table `t2` :

```
 no | valeur
----+-------
  1 | xxx
  3 | yyy
  5 | zzz
```

Nous obtenons les résultats suivants pour les différentes jointures :

```
=> SELECT * FROM t1 CROSS JOIN t2;
 no | nom | no | valeur
----+-----+----+-------
  1 | a   |  1 | xxx
  1 | a   |  3 | yyy
  1 | a   |  5 | zzz
  2 | b   |  1 | xxx
  2 | b   |  3 | yyy
  2 | b   |  5 | zzz
  3 | c   |  1 | xxx
  3 | c   |  3 | yyy
  3 | c   |  5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.no = t2.no;
 no | nom | no | valeur
```

```
----+-----+----+-------
  1 | a   |  1 | xxx
  3 | c   |  3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (no);
 no | nom | valeur
----+-----+-------
  1 | a   | xxx
  3 | c   | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 no | nom | valeur
----+-----+-------
  1 | a   | xxx
  3 | c   | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.no = t2.no;
 no | nom | no | valeur
----+-----+----+-------
  1 | a   |  1 | xxx
  2 | b   |    |
  3 | c   |  3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (no);
 no | nom | valeur
----+-----+-------
  1 | a   | xxx
  2 | b   |
  3 | c   | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.no = t2.no;
 no | nom | no | valeur
----+-----+----+-------
  1 | a   |  1 | xxx
  3 | c   |  3 | yyy
    |     |  5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.no = t2.no;
 no | nom | no | valeur
----+-----+----+-------
  1 | a   |  1 | xxx
  2 | b   |    |
  3 | c   |  3 | yyy
    |     |  5 | zzz
(4 rows)
```

La condition de jointure spécifiée avec `ON` peut aussi contenir des conditions sans relation directe avec la jointure. Ceci est utile pour quelques requêtes, mais son utilisation doit avoir été réfléchie. Par exemple :

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.no = t2.no AND t2.valeur =
 'xxx';
```

```
 no | nom | no | valeur
----+-----+----+-------
  1 | a   |  1 | xxx
  2 | b   |    |
  3 | c   |    |
(3 rows)
```

Notez que placer la restriction dans la clause `WHERE` donne un résultat différent :

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value
 = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
(1 row)
```

Ceci est dû au fait qu'une restriction placée dans la clause `ON` est traitée *avant* la jointure, alors qu'une restriction placée dans la clause `WHERE` est traitée *après* la jointure. Ceci n'a pas d'importance avec les jointures internes, mais en a une grande avec les jointures externes.

7.2.1.2. Alias de table et de colonne

Un nom temporaire peut être donné aux tables et aux références de tables complexes, nom qui sera ensuite utilisé pour référencer la table dérivée dans la suite de la requête. Cela s'appelle un *alias de table*.

Pour créer un alias de table, écrivez

```
FROM reference_table AS alias
```

ou

```
FROM reference_table alias
```

Le mot-clé `AS` n'est pas obligatoire. *`alias`* peut être tout identifiant.

Une application typique des alias de table est l'affectation d'identifiants courts pour les noms de tables longs, ce qui permet de garder des clauses de jointures lisibles. Par exemple :

```
SELECT * FROM nom_de_table_tres_tres_long s
  JOIN un_autre_nom_tres_long a ON s.id = a.no;
```

L'alias devient le nouveau nom de la table en ce qui concerne la requête en cours -- il n'est pas autorisé de faire référence à la table par son nom original où que ce soit dans la requête. Du coup, ceci n'est pas valide :

```
SELECT * FROM mon_table AS m WHERE mon_table.a > 5;    -- mauvais
```

Les alias de table sont disponibles principalement pour aider à l'écriture de requête, mais ils deviennent nécessaires pour joindre une table avec elle-même, par exemple :

```
SELECT * FROM personnes AS mere JOIN personnes AS enfant ON mere.id
 = enfant.mere_id;
```

De plus, un alias est requis si la référence de la table est une sous-requête (voir la Section 7.2.1.3).

Les parenthèses sont utilisées pour résoudre les ambiguïtés. Dans l'exemple suivant, la première instruction affecte l'alias `b` à la deuxième instance de `ma_table`, mais la deuxième instruction affecte l'alias au résultat de la jonction :

```
SELECT * FROM ma_table AS a CROSS JOIN ma_table AS b ...
SELECT * FROM (ma_table AS a CROSS JOIN ma_table) AS b ...
```

Une autre forme d'alias de tables donne des noms temporaires aux colonnes de la table ainsi qu'à la table :

```
FROM reference_table [AS] alias ( colonne1 [, colonne2 [, ...]] )
```

Si le nombre d'alias de colonnes spécifié est plus petit que le nombre de colonnes dont dispose la table réelle, les colonnes suivantes ne sont pas renommées. Cette syntaxe est particulièrement utile dans le cas de jointures avec la même table ou dans le cas de sous-requêtes.

Quand un alias est appliqué à la sortie d'une clause `JOIN`, l'alias cache le nom original référencé à l'intérieur du `JOIN`. Par exemple :

```
SELECT a.* FROM ma_table AS a JOIN ta_table AS b ON ...
```

est du SQL valide, mais :

```
SELECT a.* FROM (ma_table AS a JOIN ta_table AS b ON ...) AS c
```

n'est pas valide ; l'alias de table `a` n'est pas visible en dehors de l'alias `c`.

7.2.1.3. Sous-requêtes

Une sous-requête spécifiant une table dérivée doit être enfermée dans des parenthèses et *doit* se voir affecter un alias de table (comme dans Section 7.2.1.2). Par exemple :

```
FROM (SELECT * FROM table1) AS nom_alias
```

Cet exemple est équivalent à `FROM table1 AS nom_alias`. Des cas plus intéressants, qui ne peuvent pas être réduits à une jointure pleine, surviennent quand la sous-requête implique un groupement ou un agrégat.

Une sous-requête peut aussi être une liste `VALUES` :

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
     AS noms(prenom, nom)
```

De nouveau, un alias de table est requis. Affecter des noms d'alias aux colonnes de la liste `VALUES` est optionnel, mais c'est une bonne pratique. Pour plus d'informations, voir Section 7.7.

7.2.1.4. Fonctions de table

Les fonctions de table sont des fonctions produisant un ensemble de lignes composées de types de données de base (types scalaires) ou de types de données composites (lignes de table). Elles sont utilisées comme une table, une vue ou une sous-requête de la clause `FROM` d'une requête. Les colonnes renvoyées par les fonctions de table peuvent être incluses dans une clause `SELECT`, `JOIN` ou `WHERE` de la même manière que les colonnes d'une table, vue ou sous-requête.

Les fonctions de table peuvent aussi être combinées en utilisant la syntaxe `ROWS FROM`, avec les résultats renvoyés dans des colonnes parallèles ; le nombre de lignes résultantes dans ce cas est celui du résultat de fonction le plus large. Les résultats ayant moins de colonnes sont alignés avec des valeurs NULL.

```
appel_fonction [WITH ORDINALITY] [[AS] alias_table [(alias_colonne
 [, ... ])]]
ROWS FROM( appel_fonction [, ... ] ) [WITH ORDINALITY]
 [[AS] alias_table [(alias_colonne [, ... ])]]
```

Si la clause `WITH ORDINALITY` est ajoutée, une colonne supplémentaire de type `bigint` sera ajoutée aux colonnes de résultat de la fonction. Cette colonne numérote les lignes de l'ensemble de résultats de la fonction, en commençant à 1. (Ceci est une généralisation de la syntaxe du standard SQL pour `UNNEST ... WITH ORDINALITY`.) Par défaut, la colonne ordinale est appelée `ordinality`, mais un nom de colonne différent peut être affecté en utilisant une clause `AS`.

La fonction de table `UNNEST` peut être appelée avec tout nombre de paramètres tableaux, et envoie un nombre correspondant de colonnes comme si la fonction `UNNEST` avait été appelée sur chaque paramètre séparément (Section 9.19) et combinée en utilisant la construction `ROWS FROM`.

```
UNNEST( expression_tableau [, ... ] ) [WITH ORDINALITY]
 [[AS] alias_table [(alias_colonne [, ... ])]]
```

Si aucun *`alias_table`* n'est précisé, le nom de la fonction est utilisé comme nom de table ; dans le cas d'une construction `ROWS FROM()`, le nom de la première fonction est utilisé.

Si des alias de colonnes ne sont pas fournis pour une fonction renvoyant un type de données de base, alors le nom de la colonne est aussi le même que le nom de la fonction. Pour une fonction renvoyant un type composite, les colonnes résultats obtiennent les noms des attributs individuels du type.

Quelques exemples :

```
CREATE TABLE truc (trucid int, trucsousid int, trucnom text);

CREATE FUNCTION recuptruc(int) RETURNS SETOF truc AS $$
    SELECT * FROM truc WHERE trucid = $1;
$$ LANGUAGE SQL;

SELECT * FROM recuptruc(1) AS t1;

SELECT * FROM truc
    WHERE trucsousid IN (
                        SELECT trucsousid
                        FROM recuptruc(truc.trucid) z
                        WHERE z.trucid = truc.trucid);

CREATE VIEW vue_recuptruc AS SELECT * FROM recuptruc(1);
SELECT * FROM vue_recuptruc;
```

Dans certains cas, il est utile de définir des fonctions de table pouvant renvoyer des ensembles de colonnes différentes suivant la façon dont elles sont appelées. Pour supporter ceci, la fonction de table est déclarée comme renvoyant le pseudotype `record` sans paramètres `OUT`. Quand une telle fonction est utilisée dans une requête, la structure de ligne attendue doit être spécifiée dans la requête elle-même, de façon à ce que le système sache comment analyser et planifier la requête. Cette syntaxe ressemble à ceci :

```
appel_fonction [AS] alias (définition_colonne [, ... ])
appel_fonction AS [alias] (définition_colonne [, ... ])
ROWS FROM( ... appel_fonction AS (définition_colonne [, ... ])
 [, ... ] )
```

Lorsque la syntaxe ROWS FROM() n'est pas utilisée, la liste *définition_colonne* remplace la liste d'alias de colonnes qui aurait été autrement attachée à la clause FROM ; les noms dans les définitions de colonnes servent comme alias de colonnes. Lors de l'utilisation de la syntaxe ROWS FROM(), une liste *définition_colonne* peut être attachée à chaque fonction membre séparément ; ou s'il existe seulement une fonction membre et pas de clause WITH ORDINALITY, une liste *column_definition* peut être écrite au lieu de la liste d'alias de colonnes suivant ROWS FROM().

Considérez cet exemple :

```
SELECT *
    FROM dblink('dbname=mabd', 'SELECT proname, prosrc FROM
 pg_proc')
      AS t1(proname nom, prosrc text)
    WHERE proname LIKE 'bytea%';
```

La fonction dblink (qui fait partie du module dblink) exécute une requête distante. Elle déclare renvoyer le type record, car elle pourrait être utilisée pour tout type de requête. L'ensemble de colonnes réelles doit être spécifié dans la requête appelante de façon à ce que l'analyseur sache, par exemple, comment étendre *.

Cet exemple utilise ROWS FROM :

```
SELECT *
FROM ROWS FROM
    (
        json_to_recordset('[{"a":40,"b":"foo"},
{"a":"100","b":"bar"}]')
            AS (a INTEGER, b TEXT),
        generate_series(1, 3)
    ) AS x (p, q, s)
ORDER BY p;

  p  |  q  | s
-----+-----+---
  40 | foo | 1
 100 | bar | 2
     |     | 3
```

Il joint deux fonctions en une seule cible FROM. json_to_recordset() doit renvoyer deux colonnes, la première de type integer et la seconde de type text. Le résultat de generate_series() est utilisé directement. La clause ORDER BY trie les valeurs de la colonne en tant qu'entiers.

7.2.1.5. Sous-requêtes LATERAL

Les sous-requêtes apparaissant dans la clause FROM peuvent être précédées du mot-clé LATERAL. Ceci leur permet de référencer les colonnes fournies par les éléments précédents dans le FROM. (Sans LATERAL, chaque sous-requête est évaluée indépendamment et ne peut donc pas référencer les autres éléments de la clause FROM.)

Les fonctions renvoyant des ensembles et apparaissant dans le FROM peuvent aussi être précédées du mot-clé LATERAL, mais, pour les fonctions, le mot-clé est optionnel. Les arguments de la fonction peuvent contenir des références aux colonnes fournies par les éléments précédents dans le FROM.

Un élément LATERAL peut apparaître au niveau haut dans la liste FROM ou dans un arbre de jointures (JOIN). Dans ce dernier cas, cela peut aussi faire référence à tout élément qui est sur le côté gauche d'un JOIN, alors qu'il est positionné sur sa droite.

Quand un élément FROM contient des références croisées LATERAL, l'évaluation se fait ainsi : pour chaque ligne d'un élément FROM fournissant les colonnes référencées, ou pour chaque ensemble de lignes de plusieurs éléments FROM fournissant les colonnes, l'élément LATERAL est évalué en utilisant cette valeur de ligne ou cette valeur d'ensemble de lignes. Les lignes résultantes sont jointes comme d'habitude aux lignes résultant du calcul. C'est répété pour chaque ligne ou ensemble de lignes provenant de la table source.

Un exemple trivial de LATERAL est

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id =
 foo.bar_id) ss;
```

Ceci n'est pas vraiment utile, car cela revient exactement au même résultat que cette écriture plus conventionnelle :

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

LATERAL est principalement utile lorsqu'une colonne référencée est nécessaire pour calculer la colonne à joindre. Une utilisation habituelle est de fournir une valeur d'un argument à une fonction renvoyant un ensemble de lignes. Par exemple, supposons que vertices(polygon) renvoie l'ensemble de sommets d'un polygone, nous pouvons identifier les sommets proches des polygones stockés dans une table avec la requête suivante :

```
SELECT p1.id, p2.id, v1, v2
FROM polygones p1, polygones p2,
     LATERAL vertices(p1.poly) v1,
     LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

Cette requête pourrait aussi être écrite ainsi :

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,
     polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

ou dans diverses autres formulations équivalentes. (Nous l'avons déjà mentionné, le mot-clé LATERAL est inutile dans cet exemple, mais nous l'utilisons pour plus de clarté.)

Il est souvent particulièrement utile d'utiliser LEFT JOIN sur une sous-requête LATERAL, pour que les lignes sources apparaissent dans le résultat même si la sous-requête LATERAL ne produit aucune ligne pour elles. Par exemple, si get_product_names() renvoie les noms des produits réalisés par un manufacturier, mais que quelques manufacturiers dans notre table ne réalisent aucun produit, nous pourrions les trouver avec cette requête :

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id)
 pname ON true
WHERE pname IS NULL;
```

7.2.2. Clause WHERE

La syntaxe de la clause WHERE est

```
WHERE condition_recherche
```

où *condition_recherche* est toute expression de valeur (voir la Section 4.2) renvoyant une valeur de type boolean.

Après le traitement de la clause FROM, chaque ligne de la table virtuelle dérivée est vérifiée avec la condition de recherche. Si le résultat de la vérification est positif (true), la ligne est conservée dans la table de sortie, sinon (c'est-à-dire si le résultat est faux ou nul), la ligne est abandonnée. La condition de recherche référence typiquement au moins une colonne de la table générée dans la clause FROM ; ceci n'est pas requis, mais, dans le cas contraire, la clause WHERE n'aurait aucune utilité.

> **Note**
>
> La condition de jointure d'une jointure interne peut être écrite soit dans la clause WHERE soit dans la clause JOIN. Par exemple, ces expressions de tables sont équivalentes :
>
> ```
> FROM a, b WHERE a.id = b.id AND b.val > 5
> ```
>
> et :
>
> ```
> FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
> ```
>
> ou même peut-être :
>
> ```
> FROM a NATURAL JOIN b WHERE b.val > 5
> ```
>
> Laquelle utiliser est plutôt une affaire de style. La syntaxe JOIN dans la clause FROM n'est probablement pas aussi portable vers les autres systèmes de gestion de bases de données SQL, même si cela fait partie du standard SQL. Pour les jointures externes, il n'y a pas d'autres choix : elles doivent être faites dans la clause FROM. La clause ON ou USING d'une jointure externe n'est *pas* équivalente à une condition WHERE parce qu'elle détermine l'ajout de lignes (pour les lignes qui ne correspondent pas en entrée) ainsi que pour la suppression de lignes dans le résultat final.

Voici quelques exemples de clauses WHERE :

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 =
 fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 =
 fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 >
 fdt.c1)
```

fdt est la table dérivée dans la clause FROM. Les lignes qui ne correspondent pas à la condition de recherche de la clause WHERE sont éliminées de la table fdt. Notez l'utilisation de sous-requêtes

scalaires en tant qu'expressions de valeurs. Comme n'importe quelle autre requête, les sous-requêtes peuvent employer des expressions de tables complexes. Notez aussi comment `fdt` est référencée dans les sous-requêtes. Qualifier `c1` comme `fdt.c1` est seulement nécessaire si `c1` est aussi le nom d'une colonne dans la table d'entrée dérivée de la sous-requête. Mais qualifier le nom de colonne ajoute de la clarté même lorsque cela n'est pas nécessaire. Cet exemple montre comment le nom de colonne d'une requête externe est étendu dans les requêtes internes.

7.2.3. Clauses `GROUP BY` et `HAVING`

Après avoir passé le filtre `WHERE`, la table d'entrée dérivée peut être sujette à un regroupement en utilisant la clause `GROUP BY` et à une élimination de groupe de lignes avec la clause `HAVING`.

```
SELECT liste_selection
    FROM ...
    [WHERE ...]
    GROUP
 BY reference_colonne_regroupement[,reference_colonne_regroupement]...
```

La clause `GROUP BY` est utilisée pour regrouper les lignes d'une table qui ont les mêmes valeurs dans toutes les colonnes précisées. L'ordre dans lequel ces colonnes sont indiquées importe peu. L'effet est de combiner chaque ensemble de lignes partageant des valeurs communes en un seul groupe de lignes représentant toutes les lignes du groupe. Ceci est fait pour éliminer les redondances dans la sortie et/ou pour calculer les agrégats s'appliquant à ces groupes. Par exemple :

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

Dans la seconde requête, nous n'aurions pas pu écrire `SELECT * FROM test1 GROUP BY x` parce qu'il n'existe pas une seule valeur pour la colonne `y` pouvant être associée avec chaque autre groupe. Les colonnes de regroupement peuvent être référencées dans la liste de sélection, car elles ont une valeur constante unique par groupe.

En général, si une table est groupée, les colonnes qui ne sont pas listées dans le `GROUP BY` ne peuvent pas être référencées sauf dans les expressions d'agrégats. Voici un exemple d'expression d'agrégat :

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Ici, `sum` est la fonction d'agrégat qui calcule une seule valeur pour le groupe entier. La Section 9.21 propose plus d'informations sur les fonctions d'agrégats disponibles.

> **Astuce**
>
> Le regroupement sans expressions d'agrégats calcule effectivement l'ensemble des valeurs distinctes d'une colonne. Ceci peut aussi se faire en utilisant la clause `DISTINCT` (voir la Section 7.3.3).

Voici un autre exemple : il calcule les ventes totales pour chaque produit (plutôt que le total des ventes sur tous les produits) :

```
SELECT id_produit, p.nom, (sum(v.unite) * p.prix) AS ventes
    FROM produits p LEFT JOIN ventes v USING (id_produit)
    GROUP BY id_produit, p.nom, p.prix;
```

Dans cet exemple, les colonnes `id_produit`, `p.nom` et `p.prix` doivent être dans la clause `GROUP BY`, car elles sont référencées dans la liste de sélection de la requête (mais voir plus loin). La colonne `v.unite` n'a pas besoin d'être dans la liste `GROUP BY`, car elle est seulement utilisée dans l'expression de l'agrégat (`sum(...)`) représentant les ventes d'un produit. Pour chaque produit, la requête renvoie une ligne de résumé sur les ventes de ce produit.

Si la table produits est configurée de façon à ce que `id_produit` soit la clé primaire, alors il serait suffisant de grouper par la colonne `id_produit` dans l'exemple ci-dessus, car le nom et le prix seraient *dépendants fonctionnellement* de l'identifiant du produit, et donc il n'y aurait pas d'ambiguïté sur le nom et le prix à renvoyer pour chaque groupe d'identifiants de produits.

En SQL strict, `GROUP BY` peut seulement grouper les colonnes de la table source, mais PostgreSQL étend ceci en autorisant `GROUP BY` à grouper aussi les colonnes de la liste de sélection. Grouper par expressions de valeurs au lieu de simples noms de colonnes est aussi permis.

Si une table a été groupée en utilisant la clause `GROUP BY`, mais que seuls certains groupes sont intéressants, la clause `HAVING` peut être utilisée, comme une clause `WHERE`, pour éliminer les groupes du résultat. Voici la syntaxe :

```
SELECT liste_selection FROM ... [WHERE ...] GROUP BY ...
 HAVING expression_booléenne
```

Les expressions de la clause `HAVING` peuvent référer à la fois aux expressions groupées et aux expressions non groupées (ce qui implique nécessairement une fonction d'agrégat).

Exemple :

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)
```

De nouveau, un exemple plus réaliste :

```
SELECT id_produit, p.nom, (sum(v.unite) * (p.prix - p.cout)) AS
 profit
```

```
    FROM produits p LEFT JOIN ventes v USING (id_produit)
    WHERE v.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY id_produit, p.nom, p.prix, p.cout
    HAVING sum(p.prix * s.unite) > 5000;
```

Dans l'exemple ci-dessus, la clause `WHERE` sélectionne les lignes par une colonne qui n'est pas groupée (l'expression est vraie seulement pour les ventes des quatre dernières semaines) alors que la clause `HAVING` restreint la sortie aux groupes dont le total des ventes dépasse 5000. Notez que les expressions d'agrégats n'ont pas besoin d'être identiques dans toutes les parties d'une requête.

Si une requête contient des appels à des fonctions d'agrégat, mais pas de clause `GROUP BY`, le regroupement a toujours lieu : le résultat est une seule ligne de regroupement (ou peut-être pas de ligne du tout si la ligne unique est ensuite éliminée par la clause `HAVING`). Ceci est vrai aussi si elle comporte une clause `HAVING`, même sans fonction d'agrégat ou `GROUP BY`.

7.2.4. `GROUPING SETS`, `CUBE` et `ROLLUP`

Des opérations de regroupements plus complexes que celles décrites ci-dessus sont possibles en utilisant la notion d'*ensembles de regroupement*. Les données sélectionnées par les clauses `FROM` et `WHERE` sont regroupées séparément pour chaque ensemble de regroupement indiqué, les agrégats calculés pour chaque ensemble de la même manière que pour la clause simple `GROUP BY`, puis le résultat est retourné. Par exemple:

```
=> SELECT * FROM ventes;
 produit | taille | vendus
-------+------+-------
 Foo   | L    |  10
 Foo   | M    |  20
 Bar   | M    |  15
 Bar   | L    |  5
(4 rows)

=> SELECT produit, taille, sum(vendus) FROM ventes GROUP BY
 GROUPING SETS ((produit), (taille), ());
 produit | taille | sum
-------+------+-----
 Foo   |      |  30
 Bar   |      |  20
       | L    |  15
       | M    |  35
       |      |  50
(5 rows)
```

Chaque sous-liste de `GROUPING SETS` peut indiquer 0 ou plusieurs colonnes ou expressions et est interprétée de la même manière que si elle était directement dans la clause `GROUP BY`. Un ensemble de regroupement vide signifie que toutes les lignes sont agrégées pour former un simple groupe (qui est renvoyé quand bien même aucune ligne ne serait sélectionnée), comme décrit ci-dessus dans le cas de fonctions d'agrégat sans clause `GROUP BY`.

Les références aux colonnes de regroupement ou expressions sont remplacées par des valeurs NULL dans les lignes renvoyées pour les ensembles de regroupement où ces colonnes n'apparaissent pas. Pour identifier à quel ensemble de regroupement une ligne en particulier appartient, référez-vous à Tableau 9.62.

Une notation raccourcie est fournie pour indiquer deux types classiques d'ensembles de regroupement. Une clause sous la forme

```
ROLLUP ( e1, e2, e3, ... )
```

représente la liste indiquée d'expressions ainsi que l'ensemble des préfixes de la liste, y compris la liste vide. C'est donc équivalent à

```
GROUPING SETS (
    ( e1, e2, e3, ... ),
    ...
    ( e1, e2 ),
    ( e1 ),
    ( )
)
```

Cette notation est communément utilisée avec des données hiérarchiques ; par exemple, le total des salaires par département, division et sur l'ensemble de l'entreprise.

Une clause sous la forme

```
CUBE ( e1, e2, ... )
```

représente la liste indiquée ainsi que l'ensemble des sous-ensembles possibles. De ce fait,

```
CUBE ( a, b, c )
```

est équivalent à

```
GROUPING SETS (
    ( a, b, c ),
    ( a, b    ),
    ( a,    c ),
    ( a       ),
    (    b, c ),
    (    b    ),
    (       c ),
    (         )
)
```

Les éléments individuels des clauses `CUBE` ou `ROLLUP` peuvent être des expressions individuelles, ou des sous-listes d'éléments entre parenthèses. Dans ce dernier cas, les sous-listes sont traitées comme simple élément pour la génération des ensembles de regroupements individuels. Par exemple :

```
CUBE ( (a, b), (c, d) )
```

est équivalent à

```
GROUPING SETS (
    ( a, b, c, d ),
```

```
    ( a, b       ),
    (       c, d ),
    (            )
)
```

et

```
ROLLUP ( a, (b, c), d )
```

est équivalent à

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b, c    ),
    ( a          ),
    (            )
)
```

Les éléments `CUBE` et `ROLLUP` peuvent être utilisés directement dans la clause `GROUP BY`, ou imbriqués à l'intérieur d'une clause `GROUPING SETS`. Si une clause `GROUPING SETS` est imbriquée dans une autre, l'effet est le même que si tous les éléments de la clause la plus imbriquée avaient été écrits directement dans la clause de niveau supérieur.

Si de multiples clauses de regroupement sont indiquées dans une simple clause `GROUP BY`, alors la liste finale des ensembles de regroupements est le produit cartésien des éléments individuels. Par exemple :

```
GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))
```

est équivalent à

```
GROUP BY GROUPING SETS (
    (a, b, c, d), (a, b, c, e),
    (a, b, d),    (a, b, e),
    (a, c, d),    (a, c, e),
    (a, d),       (a, e)
)
```

Lors de la spécificiation de plusieurs éléments de regroupement, l'ensemble final des ensembles de regroupement pourrait contenir des duplicats. Par exemple :

```
GROUP BY ROLLUP (a, b), ROLLUP (a, c)
```

est équivalent à :

```
GROUP BY GROUPING SETS (
    (a, b, c),
    (a, b),
    (a, b),
```

```
    (a, c),
    (a),
    (a),
    (a, c),
    (a),
    ()
)
```

Si ces duplicats ne sont pas désirables, ils peuvent être supprimés en utilisant la clause `DISTINCT` directement sur le `GROUP BY`. De ce fait :

```
GROUP BY DISTINCT ROLLUP (a, b), ROLLUP (a, c)
```

est équivalent à :

```
GROUP BY GROUPING SETS (
    (a, b, c),
    (a, b),
    (a, c),
    (a),
    ()
)
```

Ceci n'est pas identique à utiliser `SELECT DISTINCT` car les lignes en sortie pourraient toujours contenir des duplicats. Si une des colonnes non groupées contient NULL, cela ne sera plus distinguable du NULL utilisé quand la même colonne est regroupée.

> **Note**
>
> La syntaxe `(a, b)` est normalement reconnue dans les expressions comme un constructeur de ligne. À l'intérieur d'une clause `GROUP BY`, cette règle ne s'applique pas au premier niveau d'expressions, et `(a, b)` est reconnu comme une liste d'expressions, comme décrit ci-dessus. Si pour une quelconque raison vous avez *besoin* d'un constructeur de ligne dans une expression de regroupement, utilisez `ROW(a, b)`.

7.2.5. Traitement de fonctions Window

Si la requête contient une des fonctions Window (voir Section 3.5, Section 9.22 et Section 4.2.8), ces fonctions sont évaluées après que sont effectués les regroupements, les agrégations, les filtrages par `HAVING`. C'est-à-dire que si la requête comporte des agrégats, `GROUP BY` ou `HAVING`, alors les enregistrements vus par les fonctions Window sont les lignes regroupées à la place des enregistrements originaux provenant de `FROM`/`WHERE`.

Quand des fonctions Window multiples sont utilisées, toutes les fonctions Window ayant des clauses `PARTITION BY` et `ORDER BY` équivalentes dans leurs définitions verront à coup sûr le même ordre des données en entrées, même si `ORDER BY` ne détermine pas de façon unique un tri. Toutefois, aucune garantie n'est faite à propos de l'évaluation de fonctions ayant des spécifications de `PARTITION BY` ou `ORDER BY` différentes. (Dans ces cas, une étape de tri est généralement nécessaire entre les passes d'évaluations de fonctions Window, et le tri ne garantit pas la préservation de l'ordre des enregistrements que son `ORDER BY` estime comme identiques.)

À l'heure actuelle, les fonctions Window nécessitent toujours des données prétriées, ce qui fait que la sortie de la requête sera triée suivant l'une ou l'autre des clauses `PARTITION BY`/`ORDER BY` des fonctions Window. Il n'est toutefois pas recommandé de s'en servir. Utilisez une clause `ORDER BY` au plus haut niveau de la requête si vous voulez être sûr que vos résultats soient triés d'une certaine façon.

7.3. Listes de sélection

Comme montré dans la section précédente, l'expression de table pour la commande SELECT construit une table virtuelle intermédiaire en combinant les tables, vues, en éliminant les lignes, en groupant, etc. Cette table est finalement passée à la réalisation de la *liste de sélection*. Cette liste détermine les *colonnes* de la table intermédiaire à afficher.

7.3.1. Éléments de la liste de sélection

La forme la plus simple de liste de sélection est *. C'est un raccourci pour indiquer toutes les colonnes que l'expression de table produit. Sinon, une liste de sélection est une liste d'expressions de valeurs séparées par des virgules (comme défini dans la Section 4.2). Par exemple, cela pourrait être une liste des noms de colonnes :

```
SELECT a, b, c FROM ...
```

Les noms de colonnes a, b et c sont soit les noms actuels des colonnes des tables référencées dans la clause FROM, soit les alias qui leur ont été donnés (voir l'explication dans Section 7.2.1.2). L'espace de nom disponible dans la liste de sélection est le même que dans la clause WHERE sauf si le regroupement est utilisé, auquel cas c'est le même que dans la clause HAVING.

Si plus d'une table a une colonne du même nom, le nom de la table doit aussi être donné, comme dans :

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

En travaillant avec plusieurs tables, il est aussi utile de demander toutes les colonnes d'une table particulière :

```
SELECT tbl1.*, tbl2.a FROM ...
```

Voir Section 8.16.5 pour plus d'informations sur la syntaxe *nom_table*.*.

Si une expression de valeur arbitraire est utilisée dans la liste de sélection, il ajoute conceptuellement une nouvelle colonne virtuelle dans la table renvoyée. L'expression de valeur est évaluée une fois pour chaque ligne avec une substitution des valeurs de lignes avec les références de colonnes. Mais les expressions de la liste de sélection n'ont pas à référencer les colonnes dans l'expression de la table de la clause FROM ; elles pourraient être des expressions arithmétiques constantes, par exemple.

7.3.2. Labels de colonnes

Les entrées de la liste de sélection peuvent se voir affecter des noms pour la suite de l'exécution, peut-être pour référence dans une clause ORDER BY ou pour affichage par l'application cliente. Par exemple :

```
SELECT a AS valeur, b + c AS sum FROM ...
```

Si aucun nom de colonne en sortie n'est spécifié en utilisant AS, le système affecte un nom de colonne par défaut. Pour les références de colonne simple, c'est le nom de la colonne référencée. Pour les appels de fonction, il s'agit du nom de la fonction. Pour les expressions complexes, le système générera un nom générique.

Le mot clé AS est habituellement optionnel mais dans certains cas où le nom de la colonne désirée correspond à un mot clé PostgreSQL, vous devez écrire AS ou placer le nom de la colonne entre guillemets doubles pour éviter toute ambiguïté. (Annexe C affiche les mots clés qui nécessitent que AS soit utilisé comme label de colonne.) Par exemple, FROM est un de ces mots clés, donc ceci ne fonctionnera pas :

```
SELECT a, b + c AS somme FROM ...
```

mais ces deux-là fonctionnent :

```
SELECT a AS from, b + c AS sum FROM ...
SELECT a "from", b + c AS sum FROM ...
```

Pour mieux vous protéger de possibles ajouts futurs de mots-clés, il est recommandé de toujours écrire AS ou de mettre le nom de colonne de sortie entre guillemets.

> **Note**
>
> Le nom des colonnes en sortie est différent ici de ce qui est fait dans la clause FROM (voir la Section 7.2.1.2). Il est possible de renommer deux fois la même colonne, mais le nom affecté dans la liste de sélection est celui qui sera passé.

7.3.3. DISTINCT

Après le traitement de la liste de sélection, la table résultante pourrait être optionnellement sujette à l'élimination des lignes dupliquées. Le mot-clé DISTINCT est écrit directement après SELECT pour spécifier ceci :

```
SELECT DISTINCT liste_selection ...
```

(au lieu de DISTINCT, le mot-clé ALL peut être utilisé pour spécifier le comportement par défaut, la récupération de toutes les lignes).

Évidemment, les deux lignes sont considérées distinctes si elles diffèrent dans au moins une valeur de colonne. Les valeurs NULL sont considérées égales dans cette comparaison.

Autrement, une expression arbitraire peut déterminer quelles lignes doivent être considérées distinctes :

```
SELECT DISTINCT ON (expression [, expression ...]) liste_selection
 ...
```

Ici, *expression* est une expression de valeur arbitraire, évaluée pour toutes les lignes. Les lignes dont toutes les expressions sont égales sont considérées comme dupliquées et seule la première ligne de cet ensemble est conservée dans la sortie. Notez que la « première ligne » d'un ensemble est non prévisible sauf si la requête est triée sur assez de colonnes pour garantir un ordre unique des colonnes arrivant dans le filtre DISTINCT (le traitement de DISTINCT ON parvient après le tri de ORDER BY).

La clause DISTINCT ON ne fait pas partie du standard SQL et est quelques fois considérée comme étant un mauvais style à cause de la nature potentiellement indéterminée de ses résultats. Avec l'utilisation judicieuse de GROUP BY et de sous-requêtes dans FROM, la construction peut être évitée, mais elle représente souvent l'alternative la plus agréable.

7.4. Combiner des requêtes (UNION, INTERSECT, EXCEPT)

Les résultats de deux requêtes peuvent être combinés en utilisant les opérations d'ensemble : union, intersection et différence. La syntaxe est

```
requete1 UNION [ALL] requete2
requete1 INTERSECT [ALL] requete2
requete1 EXCEPT [ALL] requete2
```

où *requete1* et *requete2* sont les requêtes pouvant utiliser toutes les fonctionnalités discutées ici.

`UNION` ajoute effectivement le résultat de *requete2* au résultat de *requete1* (bien qu'il n'y ait pas de garantie qu'il s'agisse de l'ordre dans lequel les lignes sont réellement renvoyées). De plus, il élimine les lignes dupliquées du résultat, de la même façon que `DISTINCT`, sauf si `UNION ALL` est utilisée.

`INTERSECT` renvoie toutes les lignes qui sont à la fois dans le résultat de *requete1* et dans le résultat de *requete2*. Les lignes dupliquées sont éliminées sauf si `INTERSECT ALL` est utilisé.

`EXCEPT` renvoie toutes les lignes qui sont dans le résultat de *requete1* mais pas dans le résultat de *requete2* (ceci est quelquefois appelé la *différence* entre deux requêtes). De nouveau, les lignes dupliquées sont éliminées sauf si `EXCEPT ALL` est utilisé.

Pour calculer l'union, l'intersection ou la différence de deux requêtes, les deux requêtes doivent être « compatibles pour une union », ce qui signifie qu'elles doivent renvoyer le même nombre de colonnes et que les colonnes correspondantes doivent avoir des types de données compatibles, comme décrit dans la Section 10.5.

Les opérations sur les ensembles peuvent être combinées, par exemple :

```
requete1 UNION requete2 EXCEPT requete3
```

qui est équivalent à :

```
(requete1 UNION requete2) EXCEPT requete3
```

Comme indiqué ici, vous pouvez utiliser les parenthèses pour contrôler l'ordre d'évaluation. Sans les parenthèses, `UNION` et `EXCEPT` font une association de gauche à droite, mais `INTERSECT` a une priorité plus forte que ces deux opérateurs. De ce fait :

```
requete1 UNION requete2 INTERSECT requete3
```

signifie

```
requete1 UNION (requete2 INTERSECT requete3)
```

Vous pouvez aussi entourer une *requête* individuelle avec des parenthèses. C'est important si la *requête* a besoin d'utiliser une des clauses discutées dans les sections suivantes, telles que `LIMIT`. Sans les parenthèses, vous obtiendrez soit une erreur de syntaxe soit une interprétation de cette clausse comme s'appliquant à la sortie de l'opération ensembliste plutôt que sur une de ses entrées. Par exemple :

```
SELECT a FROM b UNION SELECT x FROM y LIMIT 10
```

est acceptée, mais signifie :

```
(SELECT a FROM b UNION SELECT x FROM y) LIMIT 10
```

et non pas :

```
SELECT a FROM b UNION (SELECT x FROM y LIMIT 10)
```

7.5. Tri des lignes (ORDER BY)

Après qu'une requête a produit une table en sortie (après que la liste de sélection a été traitée), elle peut être optionnellement triée. Si le tri n'a pas été choisi, les lignes sont renvoyées dans un ordre non spécifié. Dans ce cas, l'ordre réel dépendra des types de plan de parcours et de jointure et de l'ordre sur le disque, mais vous ne devez pas vous y fier. Un tri particulier en sortie peut seulement être garanti si l'étape de tri est choisie explicitement.

La clause ORDER BY spécifie l'ordre de tri :

```
SELECT liste_selection
    FROM expression_table
    ORDER BY expression_tri1 [ASC | DESC] [NULLS { FIRST | LAST }]
[, expression_tri2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

Les expressions de tri peuvent être toute expression qui serait valide dans la liste de sélection des requêtes. Voici un exemple :

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

Quand plus d'une expression est indiquée, les valeurs suivantes sont utilisées pour trier les lignes qui sont identiques aux valeurs précédentes. Chaque expression pourrait être suivie d'un ASC ou DESC optionnel pour configurer la direction du tri (ascendant ou descendant). L'ordre ASC est la valeur par défaut. L'ordre ascendant place les plus petites valeurs en premier, où « plus petit » est défini avec l'opérateur <. De façon similaire, l'ordre descendant est déterminé avec l'opérateur >. [1]

Les options NULLS FIRST et NULLS LAST sont utilisées pour déterminer si les valeurs NULL apparaissent avant ou après les valeurs non NULL après un tri. Par défaut, les valeurs NULL sont triées comme si elles étaient plus grandes que toute valeur non NULL. Autrement dit, NULLS FIRST est la valeur par défaut pour l'ordre descendant (DESC) et NULLS LAST est la valeur utilisée sinon.

Notez que les options de tri sont considérées indépendamment pour chaque colonne triée. Par exemple, ORDER BY x, y DESC signifie en fait ORDER BY x ASC, y DESC, ce qui est différent de ORDER BY x DESC, y DESC.

Une *expression_tri* peut aussi être, à la place, le nom ou le numéro d'une colonne en sortie, par exemple :

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

les deux triant par la première colonne en sortie. Notez qu'un nom de colonne en sortie doit être unique, il ne doit pas être utilisé dans une expression -- par exemple, ceci n'est *pas* correct :

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          --
 mauvais
```

[1] En fait, PostgreSQL utilise la *classe d'opérateur B-tree par défaut* pour le type de données de l'expression pour déterminer l'ordre de tri avec ASC et DESC. De façon conventionnelle, les types de données seront initialisés de façon à ce que les opérateurs < et > correspondent à cet ordre de tri, mais un concepteur des types de données définis par l'utilisateur pourrait choisir de faire quelque chose de différent.

Cette restriction est là pour réduire l'ambiguïté. Il y en a toujours si un élément ORDER BY est un simple nom qui pourrait correspondre soit à un nom de colonne en sortie soit à une colonne d'une expression de table. La colonne en sortie est utilisée dans de tels cas. Cela causera seulement de la confusion si vous utilisez AS pour renommer une colonne en sortie qui correspondra à un autre nom de colonne d'une table.

ORDER BY peut être appliqué au résultat d'une combinaison UNION, d'une combinaisonINTERSECT ou d'une combinaison EXCEPT, mais, dans ce cas, il est seulement permis de trier par les noms ou numéros de colonnes, pas par les expressions.

7.6. LIMIT et OFFSET

LIMIT et OFFSET vous permettent de retrouver seulement une portion des lignes générées par le reste de la requête :

```
SELECT liste_selection
    FROM expression_table
    [ ORDER BY ...]
    [ LIMIT { nombre | ALL } ] [OFFSET nombre]
```

Si un nombre limite est donné, pas plus que ce nombre de lignes ne sera renvoyé (mais peut-être moins si la requête récupère moins de lignes). LIMIT ALL revient à ne pas spécifier la clause LIMIT.

OFFSET indique de passer ce nombre de lignes avant de renvoyer les lignes restantes. OFFSET 0 revient à oublier la clause OFFSET, tout comme OFFSET avec un argument NULL.

Si à la fois OFFSET et LIMIT apparaissent, alors les OFFSET lignes sont laissées avant de commencer le renvoi des LIMIT lignes.

Lors de l'utilisation de LIMIT, il est important d'utiliser une clause ORDER BY contraignant les lignes résultantes dans un ordre unique. Sinon, vous obtiendrez un sous-ensemble non prévisible de lignes de la requête. Vous pourriez demander les lignes de 10 à 20, mais dans quel ordre ? L'ordre est inconnu si vous ne spécifiez pas ORDER BY.

L'optimiseur de requêtes prend LIMIT en compte lors de la génération des plans de requêtes, de façon à ce que vous obteniez différents plans (avec différents ordres de lignes) suivant ce que vous donnez à LIMIT et OFFSET. Du coup, utiliser des valeurs LIMIT/OFFSET différentes pour sélectionner des sous-ensembles différents d'un résultat de requête *donnera des résultats inconsistants* sauf si vous forcez un ordre de résultat prévisible avec ORDER BY. Ceci n'est pas un bogue ; c'est une conséquence inhérente au fait que le SQL ne promette pas de délivrer les résultats d'une requête dans un ordre particulier sauf si ORDER BY est utilisé pour contraindre l'ordre.

Les lignes passées par une clause OFFSET devront toujours être traitées à l'intérieur du serveur ; du coup, un OFFSET important peut être inefficace.

7.7. Listes VALUES

VALUES fournit une façon de générer une table de « constantes » qui peut être utilisée dans une requête sans avoir à réellement créer et peupler une table sur disque. La syntaxe est

```
VALUES ( expression [, ...] ) [, ...]
```

Chaque liste d'expressions entre parenthèses génère une ligne dans la table. Les listes doivent toutes avoir le même nombre d'éléments (c'est-à-dire une liste de colonnes dans la table), et les entrées correspondantes dans chaque liste doivent avoir des types compatibles. Le type réel affecté à chaque colonne du résultat est déterminé en utilisant les mêmes règles que pour UNION (voir Section 10.5).

Voici un exemple :

```
VALUES (1, 'un'), (2, 'deux'), (3, 'trois');
```

renverra une table de deux colonnes et trois lignes. C'est équivalent à :

```
SELECT 1 AS column1, 'un' AS column2
UNION ALL
SELECT 2, 'deux'
UNION ALL
SELECT 3, 'trois';
```

Par défaut, PostgreSQL affecte les noms `column1`, `column2`, etc. aux colonnes d'une table `VALUES`. Les noms des colonnes ne sont pas spécifiés par le standard SQL et les différents SGBD le font de façon différente. Donc, il est généralement mieux de surcharger les noms par défaut avec une liste d'alias, comme ceci :

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t
 (num,letter);
 num | letter
-----+--------
   1 | one
   2 | two
   3 | three
(3 rows)
```

Syntaxiquement, `VALUES` suivi par une liste d'expressions est traité de la même façon que

```
SELECT liste_select FROM expression_table
```

et peut apparaître partout où un `SELECT` le peut. Par exemple, vous pouvez l'utiliser comme élément d'un `UNION` ou y attacher une *spécification de tri* (`ORDER BY`, `LIMIT` et/ou `OFFSET`). `VALUES` est habituellement utilisée comme source de données dans une commande `INSERT` command, mais aussi dans une sous-requête.

Pour plus d'informations, voir VALUES.

7.8. Requêtes `WITH` (*Common Table Expressions*)

`WITH` fournit un moyen d'écrire des ordres auxiliaires pour les utiliser dans des requêtes plus importantes. Ces requêtes, qui sont souvent appelées Common Table Expressions ou CTE, peuvent être vues comme des tables temporaires qui n'existent que pour une requête. Chaque ordre auxiliaire dans une clause `WITH` peut être un `SELECT`, `INSERT`, `UPDATE`, ou `DELETE` ; et la clause `WITH` elle-même est attachée à un ordre primaire qui peut lui aussi être un `SELECT`, `INSERT`, `UPDATE`, `DELETE` ou `MERGE`.

7.8.1. `SELECT` dans `WITH`

L'intérêt de `SELECT` dans `WITH` est de diviser des requêtes complexes en parties plus simples. Un exemple est:

```
WITH ventes_regionales AS (
```

```
        SELECT region, SUM(montant) AS ventes_totales
        FROM commandes
        GROUP BY region
    ), meilleures_regions AS (
        SELECT region
        FROM ventes_regionales
        WHERE ventes_totales > (SELECT SUM(ventes_totales)/10 FROM
 ventes_regionales)
    )
SELECT region,
       produit,
       SUM(quantite) AS unites_produit,
       SUM(montant) AS ventes_produit
FROM commandes
WHERE region IN (SELECT region FROM meilleures_regions)
GROUP BY region, produit;
```

qui affiche les totaux de ventes par produit seulement dans les régions ayant les meilleures ventes. La clause `WITH` définit deux ordres auxiliaires appelés `ventes_regionales` et `meilleures_regions`, où la sortie de `ventes_regionales` est utilisée dans `meilleures_regions` et la sortie de `meilleures_regions` est utilisée dans la requête `SELECT` primaire. Cet exemple aurait pu être écrit sans `WITH`, mais aurait alors nécessité deux niveaux de sous-`SELECT` imbriqués. Les choses sont un peu plus faciles à suivre de cette façon.

7.8.2. Requêtes récursives

Le modificateur optionnel `RECURSIVE` fait passer `WITH` du statut de simple aide syntaxique à celui de quelque chose qu'il serait impossible d'accomplir avec du SQL standard. Grâce à `RECURSIVE`, une requête `WITH` peut utiliser sa propre sortie. Un exemple très simple se trouve dans cette requête, qui ajoute les nombres de 1 à 100 :

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

La forme générale d'une requête `WITH` est toujours un *terme non récursif*, puis `UNION` (ou `UNION ALL`), puis un *terme récursif*. Seul le terme récursif peut contenir une référence à la sortie propre de la requête. Une requête de ce genre est exécutée comme suit :

Évaluation de requête récursive

1. Évaluer le terme non récursif. Pour `UNION` (mais pas `UNION ALL`), supprimer les enregistrements en double. Inclure le reste dans le résultat de la requête récursive et le mettre aussi dans une table temporaire de travail (*working table*.)

2. Tant que la table de travail n'est pas vide, répéter ces étapes :

    a. Évaluer le terme récursif, en substituant à la référence récursive le contenu courant de la table de travail. Pour `UNION` (mais pas `UNION ALL`), supprimer les doublons, ainsi que les enregistrements en doublon des enregistrements déjà obtenus. Inclure les enregistrements restants dans le résultat de la requête récursive, et les mettre aussi dans une table temporaire intermédiaire (*intermediate table*).

    b. Remplacer le contenu de la table de travail par celui de la table intermédiaire, puis supprimer la table intermédiaire.

> **Note**
>
> Alors que `RECURSIVE` autorise que les requêtes soient spécifiées récursivement, en interne, ce type de requêtes est évalué itérativement.

Dans l'exemple précédent, la table de travail a un seul enregistrement à chaque étape, et il prend les valeurs de 1 à 100 en étapes successives. À la centième étape, il n'y a plus de sortie en raison de la clause `WHERE`, ce qui met fin à la requête.

Les requêtes récursives sont utilisées généralement pour traiter des données hiérarchiques ou sous forme d'arbres. Cette requête est un exemple utile pour trouver toutes les sous-parties directes et indirectes d'un produit, si seule une table donne toutes les inclusions immédiates :

```
WITH RECURSIVE parties_incluses(sous_partie, partie, quantite) AS (
    SELECT sous_partie, partie, quantite FROM parties WHERE partie
 = 'notre_produit'
  UNION ALL
    SELECT p.sous_partie, p.partie, p.quantite * pr.quantite
    FROM parties_incluses pr, parties p
    WHERE p.partie = pr.sous_partie
  )
SELECT sous_partie, SUM(quantite) as quantite_totale
FROM parties_incluses
GROUP BY sous_partie
```

7.8.2.1. Ordre de recherche

Lors du calcul d'un parcours d'arbre en utilisant une requête récursive, vous pourriez vouloir trier les résultats soit en *depth-first* soit en *breadth-first*. Ceci peut se faire en calculant une colonne de tri parmi les autres colonnes de données et en l'utilisant pour trier les résultats à la fin. Notez que cela ne contrôle pas réellement dans qel ordre l'évaluation de la requête visite les lignes ; ceci est toujours dépendant de l'implémentation SQL. Cette approche fournit simplement une façon agréable de trier les résultats après coup.

Pour créer un ordre *depth-first*, nous calculons pour chaque résultat un tableau de lignes que nous avons déjà visité. Par exemple, considérez la requête suivante qui recherche dansune table `tree` en utilisant un champ `link` :

```
WITH RECURSIVE search_tree(id, link, data) AS (
    SELECT t.id, t.link, t.data
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree;
```

Pour ajouter l'information de tri *depth-first*, vous pouvez écrire ceci :

```
WITH RECURSIVE search_tree(id, link, data, path) AS (
    SELECT t.id, t.link, t.data, ARRAY[t.id]
    FROM tree t
  UNION ALL
```

```
    SELECT t.id, t.link, t.data, path || t.id
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree ORDER BY path;
```

Dans le cas général où plus d'un champ a besoin d'être utilisé pour identifier une ligne, utilisez un tableau de lignes. Par exemple, si vous avez besoin de tracer les champs `f1` et `f2` :

```
WITH RECURSIVE search_tree(id, link, data, path) AS (
    SELECT t.id, t.link, t.data, ARRAY[ROW(t.f1, t.f2)]
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data, path || ROW(t.f1, t.f2)
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree ORDER BY path;
```

> **Astuce**
>
> Omettez la syntaxe `ROW()` dans le cas commun où seulement un champ a besoin d'être tracé. Ceci permet l'utilisation d'un tableau simple plutôt que d'un tableau de type composite, ce qui permet de gagner en efficacité.

Pour créer un ordre *breadth-first*, vous pouvez utiliser un colonne qui trace la profondeur de la recherche, par exemple :

```
WITH RECURSIVE search_tree(id, link, data, depth) AS (
    SELECT t.id, t.link, t.data, 0
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data, depth + 1
    FROM tree t, search_tree st
    WHERE t.id = st.link
)
SELECT * FROM search_tree ORDER BY depth;
```

Pour obtenir un tri stable, ajoutez des colonnes de données comme colonnes secondaires de tri.

> **Astuce**
>
> L'algorithme d'évaluation de la requête récursive produit sa sortie dans l'ordre de recherche *breadth-first*. Néanmoins, ceci est un détail d'implémentation et il n'est pas conseillé de se baser deuss. L'ordre des lignes à l'intérieur de chaque niveau n'est pas défini, donc un tri explicite pourrait être désiré dans tous les cas.

Une syntaxe native permet de calculer une colonne de tri *depth-first* ou *breadth-first*. Par exemple :

```
WITH RECURSIVE search_tree(id, link, data) AS (
    SELECT t.id, t.link, t.data
```

```
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data
    FROM tree t, search_tree st
    WHERE t.id = st.link
) SEARCH DEPTH FIRST BY id SET ordercol
SELECT * FROM search_tree ORDER BY ordercol;

WITH RECURSIVE search_tree(id, link, data) AS (
    SELECT t.id, t.link, t.data
    FROM tree t
  UNION ALL
    SELECT t.id, t.link, t.data
    FROM tree t, search_tree st
    WHERE t.id = st.link
) SEARCH BREADTH FIRST BY id SET ordercol
SELECT * FROM search_tree ORDER BY ordercol;
```

Cette syntaxe est étendue en interne pour obtenir quelque chose de similaire aux formes écrites manuellement. La clause SEARCH indique le type de recherche désiré, la liste des colonnes à tracer pour le tri et un nom de colonnes qui contiendra les données résults pouvant être utilisée pour le tri. Cette colonne sera ajoutée implicitement aux lignes en sortie de la CTE.

7.8.2.2. Détection de cycles

Quand on travaille avec des requêtes récursives, il est important d'être sûr que la partie récursive de la requête finira par ne retourner aucun enregistrement, au risque sinon de voir la requête boucler indéfiniment. Quelquefois, utiliser `UNION` à la place de `UNION ALL` peut résoudre le problème en supprimant les enregistrements qui doublonnent ceux déjà retournés. Toutefois, souvent, un cycle ne met pas en jeu des enregistrements de sortie qui sont totalement des doublons : il peut s'avérer nécessaire de vérifier juste un ou quelques champs, afin de s'assurer que le même point a déjà été atteint précédemment. La méthode standard pour gérer ces situations est de calculer un tableau de valeurs déjà visitées. Par exemple, observez de nouveau la requête suivante, qui parcourt une table `graphe` en utilisant un champ `lien` :

```
WITH RECURSIVE parcourt_graphe(id, lien, donnee, profondeur) AS (
        SELECT g.id, g.lien, g.donnee, 0
        FROM graphe g
      UNION ALL
        SELECT g.id, g.lien, g.donnee, sg.profondeur + 1
        FROM graphe g, parcourt_graphe sg
        WHERE g.id = sg.lien
)
SELECT * FROM parcourt_graphe;
```

Cette requête va boucler si la liaison `lien` contient des boucles. Parce que nous avons besoin de la sortie « profondeur », simplement remplacer `UNION ALL` par `UNION` ne résoudra pas le problème. À la place, nous avons besoin d'identifier si nous avons atteint un enregistrement que nous avons déjà traité pendant notre parcours des liens. Nous ajoutons deux colonnes `is_cycle` et `path` à la requête :

```
WITH RECURSIVE search_graph(id, link, data, depth, is_cycle, path)
 AS (
    SELECT g.id, g.link, g.data, 0,
      false,
      ARRAY[g.id]
    FROM graph g
```

```
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      g.id = ANY(path),
      path || g.id
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT is_cycle
)
SELECT * FROM search_graph;
```

En plus de prévenir les boucles, cette valeur de tableau est souvent pratique en elle-même pour représenter le « chemin » pris pour atteindre chaque enregistrement.

De façon plus générale, quand plus d'un champ a besoin d'être vérifié pour identifier une boucle, utilisez un tableau d'enregistrements. Par exemple, si nous avions besoin de comparer les champs `f1` et `f2` :

```
WITH RECURSIVE search_graph(id, link, data, depth, is_cycle, path)
 AS (
    SELECT g.id, g.link, g.data, 0,
      false,
      ARRAY[ROW(g.f1, g.f2)]
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      ROW(g.f1, g.f2) = ANY(path),
      path || ROW(g.f1, g.f2)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT is_cycle
)
SELECT * FROM search_graph;
```

> **Astuce**
>
> Omettez la syntaxe `ROW()` dans le cas courant où un seul champ a besoin d'être testé pour déterminer une boucle. Ceci permet, par l'utilisation d'un tableau simple plutôt que d'un tableau de type composite, de gagner en efficacité.

Il existe une syntaxe interne pour simplifier la détection de cycles. La requête ci-dessus peut aussi être écrite ainsi :

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 1
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
) CYCLE id SET is_cycle USING path
SELECT * FROM search_graph;
```

et elle sera réécrite en interne sous le forme ci-dessus. La clause `CYCLE` indique tout d'abord la liste des colonnes à tracer pour une détection de cycle, puis le nom de la colonne qui indiquera si un cycle a été détecté, et enfin le nom d'une autre colonne qui tracera le chemin. Les colonnes cycle et chemin seront automatiquement ajoutées aux lignes en sortie de la CTE.

> **Astuce**
>
> La colonne du chemindu cycle est calculée de la même façon que l'affiche la colonne de tri *depth-first* dans la section précédente. Une requête peut avoir à la fois une clause `SEARCH` et une clause `CYCLE`, mais une spécification de recherche *depth-first* et une spécification de recherche de cyle vont créer des calculs redondants, donc il est plus efficace d'utiliser juste la clause `CYCLE` et trier par la colonne du chemin. Si un tri *breadth-first* est voulu, alors indiquer les deux, `SEARCH` and `CYCLE`, peut être utile.

Si vous n'êtes pas certain qu'une requête puisse boucler, une astuce pratique pour la tester est d'utiliser `LIMIT` dans la requête parente. Par exemple, cette requête bouclerait indéfiniment sans un `LIMIT` :

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

Ceci fonctionne parce que l'implémentation de PostgreSQL n'évalue que le nombre d'enregistrements de la requête `WITH` récupérés par la requête parente. L'utilisation de cette astuce en production est déconseillée parce que d'autres systèmes pourraient fonctionner différemment. Par ailleurs, cela ne fonctionnera pas si vous demandez à la requête externe de trier les résultats de la requête récursive, ou si vous les joignez à une autre table, parce dans ces cas, la requête extérieure essaiera habituellement de récupérer toute la sortie de la requête `WITH` de toute façon.

7.8.3. Matérialisation des CTE

Une propriété intéressante des requêtes `WITH` est qu'elles ne sont évaluées qu'une seule fois par exécution de la requête parente ou des requêtes `WITH` sœurs. Par conséquent, les calculs coûteux qui sont nécessaires à plusieurs endroits peuvent être placés dans une requête `WITH` pour éviter le travail redondant. Un autre intérêt peut être d'éviter l'exécution multiple d'une fonction ayant des effets de bord. Néanmoins, le revers de la médaille est que l'optimiseur n'est pas en mesure de faire descendre les restrictions de la requête parent dans une requête `WITH` à références multiples, car cela pourrait affecter toutes les utilisations de la sortie de la requête `WITH` alors que cela ne devrait en affecter qu'une seule. La requête `WITH` sera généralement exécutée telle quelle, sans suppression d'enregistrements, que la requête parente devra supprimer ensuite. (Mais, comme mentionné précédemment, l'évaluation pourrait s'arrêter rapidement si la (les) référence(s) à la requête ne demande(nt) qu'un nombre limité d'enregistrements).

Néanmoins, si une requête `WITH` est non récursive et qu'elle est libre de tout effet de bord (autrement dit un `SELECT` ne contenant aucune fonction volatile), alors elle peut être intégrée dans la requête parent, permettant ainsi une optimisation de la jointure sur les deux niveaux de la requête. Par défaut, ceci survient si la requête parente fait référence une seule fois à la requête `WITH` mais si elle y fait référence plusieurs fois. Vous pouvez surcharger cette décision en indiquant `MATERIALIZED` pour forcer un calcul séparé de la requête `WITH` ou en spécifiant `NOT MATERIALIZED` pour la forcer pour être intégrée dans la requête parente. Ce dernier choix risque de dupliquer des calculs sur la requête `WITH`, mais cela peut apporter un gain net si chaque utilisation de la requête `WITH` ne nécessite qu'une petite partie de la sortie complète de la requête `WITH`.

Un exemple simple de ces règles est le suivant :

```
WITH w AS (
    SELECT * FROM big_table
```

```
)
SELECT * FROM w WHERE key = 123;
```

Cette requête `WITH` va être intégrée, produisant le même plan d'exécution que :

```
SELECT * FROM big_table WHERE key = 123;
```

EN particulier, s'il existe un index sur `key`, il sera probablement utilisé pour récupérer les lignes pour lesquelles `key = 123`. D'un autre côté, dans

```
WITH w AS (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

la requête `WITH` sera matérialisée, produisant une copie temporaire de `big_table` qui est ensuite jointe avec elle-même -- sans intérêt pour un index. Cette requête sera exécutée bien plus efficacement s'il est écrite ainsi :

```
WITH w AS NOT MATERIALIZED (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

pour que les restrictions de la requête parent puissent être appliquées directement aux parcours de `big_table`.

Voici un exemple où `NOT MATERIALIZED` pourrait être indésirable :

```
WITH w AS (
    SELECT key, very_expensive_function(val) as f FROM some_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.f = w2.f;
```

Ici, la matérialisation de la requête `WITH` assure que la `very_expensive_function` est évaluée uniquement une fois par ligne de table, et non pas deux fois.

Les exemples précédents ne montrent que des cas d'utilisation de `WITH` avec `SELECT`, mais on peut les attacher de la même façon à un `INSERT`, `UPDATE`, `DELETE` ou `MERGE`. Dans chaque cas, le mécanisme fournit en fait des tables temporaires auxquelles on peut faire référence dans la commande principale.

7.8.4. Ordres de modification de données avec `WITH`

Vous pouvez utiliser des ordres de modification de données (`INSERT`, `UPDATE` ou `DELETE`, mais pas `MERGE`) dans `WITH`. Cela vous permet d'effectuer plusieurs opérations différentes dans la même requête. Par exemple:

```
WITH lignes_deplacees AS (
    DELETE FROM produits
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
INSERT INTO log_produits
SELECT * FROM lignes_deplacees;
```

Cette requête déplace les enregistrements de `produits` vers `log_produits`. Le `DELETE` du `WITH` supprime les enregistrements spécifiés de `produits`, en retournant leurs contenus par la clause `RETURNING`; puis la requête primaire lit cette sortie et l'insère dans `log_produits`.

Un point important à noter de l'exemple précédent est que la clause `WITH` est attachée à l'`INSERT`, pas au sous-`SELECT` de l' `INSERT`. C'est nécessaire parce que les ordres de modification de données ne sont autorisés que dans les clauses `WITH` qui sont attachées à l'ordre de plus haut niveau. Toutefois, les règles de visibilité normales de `WITH` s'appliquent, il est donc possible de faire référence à la sortie du `WITH` dans le sous-`SELECT`.

Les ordres de modification de données dans `WITH` ont habituellement des clauses `RETURNING` (voir Section 6.4), comme dans l'exemple précédent. C'est la sortie de la clause `RETURNING`, *pas* la table cible de l'ordre de modification de données, qui forme la table temporaire à laquelle on pourra faire référence dans le reste de la requête. Si un ordre de modification de données dans `WITH` n'a pas de clause `RETURNING`, alors il ne produit pas de table temporaire et ne peut pas être utilisé dans le reste de la requête. Un ordre de ce type sera toutefois exécuté. En voici un exemple (dénué d'intérêt) :

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

Cet exemple supprimerait tous les éléments des tables `foo` et `bar`. Le nombre d'enregistrements retourné au client n'inclurait que les enregistrements supprimés de `bar`.

Les autoréférences récursives dans les ordres de modification de données ne sont pas autorisées. Dans certains cas, il est possible de contourner cette limitation en faisant référence à la sortie d'un `WITH`, par exemple:

```
WITH RECURSIVE pieces_incluses(sous_piece, piece) AS (
    SELECT sous_piece, piece FROM pieces WHERE piece =
 'notre_produit'
  UNION ALL
    SELECT p.sous_piece, p.piece
    FROM pieces_incluses pr, pieces p
    WHERE p.piece = pr.sous_piece
  )
DELETE FROM pieces
  WHERE piece IN (SELECT piece FROM pieces_incluses);
```

Cette requête supprimerait toutes les pièces directes et indirectes d'un produit.

Les ordres de modification de données dans `WITH` sont exécutés exactement une fois, et toujours jusqu'à la fin, indépendamment du fait que la requête primaire lise tout (ou même une partie) de leur sortie. Notez que c'est différent de la règle pour `SELECT` dans `WITH`: comme précisé dans la section

précédente, l'exécution d'un SELECT n'est poursuivie que tant que la requête primaire consomme sa sortie.

Les sous-requêtes du WITH sont toutes exécutées simultanément et simultanément avec la requête principale. Par conséquent, quand vous utilisez un ordre de modification de données avec WITH, l'ordre dans lequel les mises à jour sont effectuées n'est pas prévisible. Toutes les requêtes sont exécutées dans le même *instantané* (voyez Chapitre 13), elles ne peuvent donc pas voir les effets des autres sur les tables cibles. Ceci rend sans importance le problème de l'imprévisibilité de l'ordre des mises à jour, et signifie que RETURNING est la seule façon de communiquer les modifications entre les différentes sous-requêtes WITH et la requête principale. En voici un exemple :

```
WITH t AS (
    UPDATE produits SET prix = prix * 1.05
    RETURNING *
)
SELECT * FROM produits;
```

Le SELECT externe retournerait les prix originaux avant l'action de UPDATE, alors qu'avec :

```
WITH t AS (
    UPDATE produits SET prix = prix * 1.05
    RETURNING *
)
SELECT * FROM t;
```

le SELECT externe retournerait les données mises à jour.

Essayer de mettre à jour le même enregistrement deux fois dans le même ordre n'est pas supporté. Seule une des deux modifications a lieu, mais il n'est pas aisé (et quelquefois impossible) de déterminer laquelle. Ceci s'applique aussi pour la suppression d'un enregistrement qui a déjà été mis à jour dans le même ordre : seule la mise à jour est effectuée. Par conséquent, vous devriez éviter en règle générale de mettre à jour le même enregistrement deux fois en un seul ordre. En particulier, évitez d'écrire des sous-requêtes qui modifieraient les mêmes enregistrements que la requête principale ou une autre sous-requête. Les effets d'un ordre de ce type seraient imprévisibles.

À l'heure actuelle, les tables utilisées comme cibles d'un ordre modifiant les données dans un WITH ne doivent avoir ni règle conditionnelle, ni règle ALSO, ni une règle INSTEAD qui génère plusieurs ordres.

Chapitre 8. Types de données

PostgreSQL offre un large choix de types de données disponibles nativement. Les utilisateurs peuvent ajouter de nouveaux types à PostgreSQL en utilisant la commande CREATE TYPE.

Le Tableau 8.1 montre tous les types de données généraux disponibles nativement. La plupart des types de données alternatifs listés dans la colonne « Alias » sont les noms utilisés en interne par PostgreSQL pour des raisons historiques. Il existe également d'autres types de données internes ou obsolètes, mais ils ne sont pas listés ici.

Tableau 8.1. Types de données

Nom	Alias	Description
`bigint`	`int8`	Entier signé sur huit octets
`bigserial`	`serial8`	Entier sur huit octets à incrémentation automatique
`bit [ (n) ]`		Suite de bits de longueur fixe
`bit varying [ (n) ]`	`varbit [ (n) ]`	Suite de bits de longueur variable
`boolean`	`bool`	Booléen (Vrai/Faux)
`box`		Boîte rectangulaire dans le plan
`bytea`		Donnée binaire (« tableau d'octets »)
`character [ (n) ]`	`char [ (n) ]`	Chaîne de caractères de longueur fixe
`character varying [ (n) ]`	`varchar [ (n) ]`	Chaîne de caractères de longueur variable
`cidr`		Adresse réseau IPv4 ou IPv6
`circle`		Cercle dans le plan
`date`		Date du calendrier (année, mois, jour)
`double precision`	`float`, `float8`	Nombre à virgule flottante de double précision (sur huit octets)
`inet`		Adresse d'ordinateur IPv4 ou IPv6
`integer`	`int`, `int4`	Entier signé sur quatre octets
`interval [ champs ] [ (p) ]`		Intervalle de temps
`json`		Données texte JSON
`jsonb`		Données binaires JSON, décomposées
`line`		Droite (infinie) dans le plan
`lseg`		Segment de droite dans le plan
`macaddr`		Adresse MAC (pour *Media Access Control*)
`macaddr8`		Adresse MAC (pour *Media Access Control*) (format EUI-64)
`money`		Montant monétaire
`numeric [ (p, s) ]`	`decimal [ (p, s) ]`	Nombre exact dont la précision peut être spécifiée
`path`		Chemin géométrique dans le plan

Nom	Alias	Description
pg_lsn		Séquence numérique de journal (Log Sequence Number) de PostgreSQL
pg_snapshot		image (*snapshot*) de l'identifiant de transaction niveau utilisateur
point		Point géométrique dans le plan
polygon		Chemin géométrique fermé dans le plan
real	float4	Nombre à virgule flottante de simple précision (sur quatre octets)
smallint	int2	Entier signé sur deux octets
smallserial	serial2	Entier sur deux octets à incrémentation automatique
serial	serial4	Entier sur quatre octets à incrémentation automatique
text		Chaîne de caractères de longueur variable
time [(*p*)] [without time zone]		Heure du jour (sans fuseau horaire)
time [(*p*)] with time zone	timetz	Heure du jour, avec fuseau horaire
timestamp [(*p*)] [without time zone]		Date et heure (sans fuseau horaire)
timestamp [(*p*) with time zone	timestamptz	Date et heure, avec fuseau horaire
tsquery		requête pour la recherche plein texte
tsvector		document pour la recherche plein texte
txid_snapshot		image de l'identifiant de transaction au niveau utilisateur (déprécié ; voir pg_snapshot)
uuid		identifiant unique universel
xml		données XML

Compatibilité

Les types suivants sont conformes à la norme SQL : bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (avec et sans fuseau horaire), timestamp (avec et sans fuseau horaire), xml.

Chaque type de données a une représentation externe déterminée par ses fonctions d'entrée et de sortie. De nombreux types de données internes ont un format externe évident. Cependant, certains types sont spécifiques à PostgreSQL, comme les chemins géométriques, ou acceptent différents formats, comme les types de données de date et d'heure. Certaines fonctions d'entrée et de sortie ne sont pas inversables : le résultat de la fonction de sortie peut manquer de précision comparé à l'entrée initiale.

8.1. Types numériques

Les types numériques sont constitués d'entiers de deux, quatre ou huit octets, de nombres à virgule flottante de quatre ou huit octets et de décimaux dont la précision peut être indiquée. Le Tableau 8.2 précise les types disponibles.

Tableau 8.2. Types numériques

Nom	Taille de stockage	Description	Étendue
`smallint`	2 octets	entier de faible étendue	de -32768 à +32767
`integer`	4 octets	entier habituel	de -2147483648 à +2147483647
`bigint`	8 octets	grand entier	de -9223372036854775808 à +9223372036854775807
`decimal`	variable	précision indiquée par l'utilisateur, valeur exacte	jusqu'à 131072 chiffres avant le point décimal ; jusqu'à 16383 après le point décimal
`numeric`	variable	précision indiquée par l'utilisateur, valeur exacte	jusqu'à 131072 chiffres avant le point décimal ; jusqu'à 16383 après le point décimal
`real`	4 octets	précision variable, valeur inexacte	précision de 6 décimales
`double precision`	8 octets	précision variable, valeur inexacte	précision de 15 décimales
`smallserial`	2 bytes	Entier sur 2 octets à incrémentation automatique	1 to 32767
`serial`	4 octets	entier à incrémentation automatique	de 1 à 2147483647
`bigserial`	8 octets	entier de grande taille à incrémentation automatique	de 1 à 9223372036854775807

La syntaxe des constantes pour les types numériques est décrite dans la Section 4.1.2. Les types numériques ont un ensemble complet d'opérateurs arithmétiques et de fonctions. On peut se référer au Chapitre 9 pour plus d'informations. Les sections suivantes décrivent ces types en détail.

8.1.1. Types entiers

Les types `smallint`, `integer` et `bigint` stockent des nombres entiers, c'est-à-dire sans décimale, de différentes étendues. Toute tentative d'y stocker une valeur en dehors de l'échelle produit une erreur.

Le type `integer` est le plus courant. Il offre un bon compromis entre capacité, espace utilisé et performance. Le type `smallint` n'est utilisé que si l'économie d'espace disque est le premier critère de choix. Le type `bigint` est conçu pour n'être utilisé que si l'échelle de valeurs du type `integer` n'est pas suffisante.

SQL ne définit que les types de données `integer` (ou `int`), `smallint` et `bigint`. Les noms de types `int2`, `int4`, et `int8` sont des extensions, partagées par d'autres systèmes de bases de données SQL.

8.1.2. Nombres à précision arbitraire

Le type `numeric` peut stocker des nombres contenant un très grand nombre de chiffres. Il est spécialement recommandé pour stocker les montants financiers et autres quantités pour lesquels l'exactitude est indispensable. Les calculs avec des valeurs `numeric` renvoient des résultats exacts quand c'est possible (addition, soustraction, multiplication). Néanmoins, les calculs sur les valeurs `numeric` sont très lents comparés aux types entiers ou aux types à virgule flottante décrits dans la section suivante.

Dans ce qui suit, on utilise les termes suivants. La *précision* d'un `numeric` est le nombre total de chiffres significatifs dans le nombre complet, c'est-à-dire le nombre de chiffres de part et d'autre du séparateur. L'*échelle* d'un `numeric` est le nombre de chiffres décimaux de la partie fractionnaire, à droite du séparateur de décimales. Donc, le nombre 23.5141 a une précision de 6 et une échelle de 4. On peut considérer que les entiers ont une échelle de 0.

La précision maximale et l'échelle maximale d'une colonne `numeric` peuvent être toutes deux réglées. Pour déclarer une colonne de type numérique, il faut utiliser la syntaxe :

`NUMERIC(`*`précision`*`, `*`échelle`*`)`

La précision doit être strictement positive, alors que l'échelle peut être positive ou négative (voir ci-dessous). Alternativement :

`NUMERIC(`*`précision`*`)`

indique une échelle de 0.

`NUMERIC`

sans précision ni échelle crée une « colonne non contrainte » dans laquelle on peut stocker des valeurs de n'importe quelle longueur, dans les limites de l'implémentation. Une colonne de ce type n'impose aucune précision à la valeur entrée, alors que les colonnes `numeric` ayant une échelle forcent les valeurs entrées à cette échelle. (Le standard SQL demande une précision par défaut de 0, c'est-à-dire de forcer la transformation en entier. Les auteurs trouvent cela inutile. Dans un souci de portabilité, il est préférable de toujours indiquer explicitement la précision et l'échelle.)

> **Note**
>
> La précision maximale autorisée qui peut être explicitement spécifiée dans la déclaration du type est de 1000. Une donnée `numeric` sans contrainte est sujet aux limites décrites dans Tableau 8.2.

Si l'échelle d'une valeur à stocker est supérieure à celle de la colonne, le système arrondit la valeur au nombre de décimales indiqué pour la colonne. Si le nombre de chiffres à gauche du point décimal est supérieur à la différence entre la précision déclarée et l'échelle déclarée, une erreur est levée. Par exemple, une colonne déclarée comme

```
NUMERIC(3, 1)
```

voit ses valeurs arrondies à une place décimale et peut enregistrer des valeurs entre -99,9 et 99,9, valeurs inclues.

À partir de PostgreSQL, il est autorisé de déclarer une colonne de type `numeric` avec une échelle négative. Les valeurs sont alors arrondies à partir de la gauche du point décimal. La précision représente toujours le nombre maximum de chiffres non arrondis. De ce fait, une colonne déclarée ainsi

```
NUMERIC(2, -3)
```

arrondira les valeurs au millier le plus proche et peut enregistrer des valeurs entre -99000 et 99000 inclus. Il est aussi autorisé de déclarer une échelle plus grande que la précision déclarée. Une telle colonne peut seulement contenir des valeurs fractionnelles, et elle nécessite le nombre de chiffres zéro juste à droite du point décimal pour être au moins l'échelle déclarée moins la précision déclarée. Par exemple, une colonne déclarée ainsi

```
NUMERIC(3, 5)
```

arrondira les valeurs aux 5 places décimales et peut enregistrer des valeurs comprises entre -0,00999 et 0,00999, valeurs inclues.

> **Note**
>
> PostgreSQL permet à l'échelle dans une déclaration de type `numeric` d'être toute valeur dans l'échelle de -1000 à 1000. Néanmoins, le standard SQL requiert que l'échelle soit dans l'intervalle 0 à *`precision`*. Utiliser des échelles en dehors de cet interval peut ne pas autre portable auprès d'autres systèmes de bases de données.

Les valeurs numériques sont stockées physiquement sans zéro avant ou après. Du coup, la précision déclarée et l'échelle de la colonne sont des valeurs maximales, pas des allocations fixes (en ce sens, le type numérique est plus proche de `varchar(`*`n`*`)` que de `char(`*`n`*`)`). Le besoin pour le stockage réel est de deux octets pour chaque groupe de quatre chiffres décimaux, plus trois à huit octets d'en-tête.

En plus des valeurs numériques ordinaires, le type `numeric` a plusieurs valeurs spéciales :

```
Infinity
-Infinity
NaN
```

Ces valeurs sont adaptées du standard IEEE 754, et représentent respectivement « l'infini », « l'infini négatif » et « pas-un-nombre ». Pour écrire ces valeurs sous forme de constantes dans une commande SQL, vous devez les mettre entre guillemets, par exemple `UPDATE table SET x = '-Infinity'`. En entrée, ces chaînes sont reconnues quelque soit leur casse. Les valeurs infinis peuvent s'écrire alternativement `inf` et `-inf`.

Les valeurs infinies se comportent comme attendues mathématiquement. Par exemple, `Infinity` plus n'importe quelle valeur finie donne `Infinity` ; mais `Infinity` moins `Infinity` retourne `NaN` (pas un nombre) parce qu'il n'y a pas d'interprétation correcte possible. Notez que l'infini peut être seulement employé dans une colonne `numeric` sans contrainte, parce que sa notion dépasse n'importe quelle précision finie.

La valeur `NaN` est utilisée pour représenter un résultat non défini d'un calcul. En général, n'importe quelle opération avec une entrée `NaN` donne un autre `NaN`. La seule exception est quand les autres entrées de l'opération sont telles que le même résultat serait obtenu si les valeurs `NaN` étaient remplacées par n'importe quelle valeur numérique finie ou infinie ; ainsi, ce résultat est utilisé aussi pour `NaN`. (Un exemple de cette notion est que `NaN` élevé à la puissance zéro donne un.)

> **Note**
>
> Dans la plupart des implémentations du concept « not-a-number », `NaN` est considéré différent de toute valeur numérique (ceci incluant `NaN`). Pour autoriser le tri des valeurs de type

> `numeric` et les utiliser dans des index basés sur le tri, PostgreSQL traite les valeurs `NaN` comme identiques entre elles, mais toutes supérieures aux valeurs non `NaN`.

Les types `decimal` et `numeric` sont équivalents. Les deux types sont dans le standard SQL.

Lors de l'arrondissement de valeurs, le type `numeric` arrondit en s'éloignant de zéro, alors que (sur la plupart des machines) les types `real` et `double precision` arrondissent vers le nombre le plus proche. Par exemple :

```
SELECT x,
  round(x::numeric) AS num_round,
  round(x::double precision) AS dbl_round
FROM generate_series(-3.5, 3.5, 1) as x;
  x   | num_round | dbl_round
------+-----------+-----------
 -3.5 |        -4 |        -4
 -2.5 |        -3 |        -2
 -1.5 |        -2 |        -2
 -0.5 |        -1 |        -0
  0.5 |         1 |         0
  1.5 |         2 |         2
  2.5 |         3 |         2
  3.5 |         4 |         4
(8 rows)
```

8.1.3. Types à virgule flottante

Les types de données `real` et `double precision` sont des types numériques inexacts de précision variable. Sur toutes les plateformes actuellement supportées, ces types sont une implémentation du « IEEE Standard 754 for Binary Floating-Point Arithmetic » (respectivement, simple et double précision), suivant le support fourni par le processeur, le système d'exploitation et le compilateur.

Inexact signifie que certaines valeurs ne peuvent être converties exactement dans le format interne. Elles sont, de ce fait, stockées sous une forme approchée. Ainsi, stocker puis réafficher ces valeurs peut faire apparaître de légers écarts. Prendre en compte ces erreurs et la façon dont elles se propagent au cours des calculs est le sujet d'une branche entière des mathématiques et de l'informatique, qui n'est pas le sujet de ce document, à l'exception des points suivants :

- pour un stockage et des calculs exacts, comme pour les valeurs monétaires, le type `numeric` doit être privilégié ;
- pour des calculs compliqués avec ces types pour quoi que ce soit d'important, et particulièrement pour le comportement aux limites (infini, zéro), l'implantation spécifique à la plateforme doit être étudiée avec soin ;
- tester l'égalité de deux valeurs à virgule flottante peut ne pas donner le résultat attendu.

Sur toutes les plates-formes supportées, le type `real` a une étendue d'au moins 1E-37 à 1E37 avec une précision d'au moins six chiffres décimaux. Le type `double precision` a une étendue de 1E-307 à 1E+308 avec une précision d'au moins quinze chiffres. Les valeurs trop grandes ou trop petites produisent une erreur. Un arrondi peut avoir lieu si la précision d'un nombre en entrée est trop grande. Les nombres trop proches de zéro qui ne peuvent être représentés autrement que par zéro produisent une erreur (underflow).

Par défaut, les valeurs à virgule flottante sont renvoyées sous forme de texte dans leur représentation décimale la plus courte ; la valeur décimale produite est plus proche de la vraie valeur binaire

enregistrée que toute autre valeur représentable dans la même précision binaire. (Néanmoins, la valeur en sortie n'est en fait jamais *exactement* entre deux valeurs représentables, pour éviter un bug fréquent où les routines en entrée ne respectent pas la règle d'arrondi.) Cette valeur utilisera au plus les 17 chiffres décimaux significatifs pour des valeurs `float8`, et au plus 9 chiffres pour des valeurs `float4`.

> **Note**
>
> Ce format de sortie le plus précis tout en étant le plus court est bien plus rapide à générer que le format arrondi historique.

Pour la compatibilité avec la sortie générée par les anciennes versions de PostgreSQL et pour permettre de réduire la précision de la sortie, le paramètre extra_float_digits peut être utilisé pour sélectionner une sortie décimale arrondie à la place. Configurer une valeur 0 restaure le précédent comportement par défaut avec un arrondi à de la valeur aux 6 (pour `float4`) ou 15 (pour `float8`) chiffres décimaux significatifs. Configurer une valeur négative réduit encore plus le nombre de chiffres ; par exemple, -2 arrondirait la sortie à, respectivement, 4 ou 13 chiffres.

Toute valeur de extra_float_digits supérieure à 0 sélectionne le format précis le plus court.

> **Note**
>
> Les applications qui voulaient des valeurs précises devaient historiquement configurer extra_float_digits à 3 pour les obtenir. Pour un maximum de compatibilité entre les versions, elles doivent continuer à le faire.

> **Note**
>
> Le paramètre extra_float_digits contrôle le nombre de chiffres significatifs inclus lorsqu'une valeur à virgule flottante est convertie en texte. Avec la valeur par défaut de `0`, la sortie est la même sur chaque plateforme supportée par PostgreSQL. L'augmenter va produire une sortie représentant plus précisément la valeur stockée, mais il est possible que la sortie soit différente suivant les plates-formes.

En plus des valeurs numériques ordinaires, les types à virgule flottante ont plusieurs valeurs spéciales :

```
Infinity -Infinity
  NaN
```

Elles représentent les valeurs spéciales de l'IEEE 754, respectivement « infinity » (NdT : infini), « negative infinity » (NdT : infini négatif) et « not-a-number » (NdT : pas un nombre). Lorsqu'elles sont saisies en tant que constantes dans une commande SQL, ces valeurs doivent être placées entre guillemets. Par exemple, `UPDATE table SET x = '-Infinity'`. En entrée, ces valeurs sont reconnues, quelle que soit la casse utilisée. Les valeurs infinies peuvent être alternativement écrites `inf` et `-inf`.

> **Note**
>
> IEEE 754 spécifie que `NaN` ne devrait pas être considéré égale à toute autre valeur en virgule flottante (ceci incluant `NaN`). Pour permettre le tri des valeurs en virgule flottante et leur

utilisation dans des index basés sur des arbres, PostgreSQL traite les valeurs `NaN` comme identiques entre elles, mais supérieures à toute valeur différente de `NaN`.

PostgreSQL autorise aussi la notation `float` du standard SQL, ainsi que `float(`*p*`)` pour indiquer des types numériques inexacts. *p* indique la précision minimale acceptable en *chiffres binaires*. PostgreSQL accepte de `float(1)` à `float(24)`, qu'il transforme en type `real`, et de `float(25)` à `float(53)`, qu'il transforme en type `double precision`. Toute valeur de *p* hors de la zone des valeurs possibles produit une erreur. `float` sans précision est compris comme `double precision`.

8.1.4. Types seriés

Note

Cette section décrit une façon spécifique à PostgreSQL de créer une colonne autoincrémentée. Une autre façon revient à utiliser les colonnes d'identité, décrite sur CREATE TABLE.

Les types de données `smallserial`, `serial` et `bigserial` ne sont pas de vrais types, mais plutôt un raccourci de notation pour créer des colonnes d'identifiants uniques (similaires à la propriété `AUTO_INCREMENT` utilisée par d'autres SGBD). Dans la version actuelle, indiquer :

```
CREATE TABLE nom_de_table (
    nom_de_colonne SERIAL
);
```

est équivalent à écrire :

```
CREATE SEQUENCE nom_de_table_nom_de_colonne_seq AS integer;
CREATE TABLE nom_de_table (
    nom_de_colonne integer NOT NULL DEFAULT
 nextval('nom_de_table_nom_de_colonne_seq') NOT NULL
);
ALTER SEQUENCE nom_de_table_nom_de_colonne_seq OWNED
 BY nom_de_table.nom_de_colonne;
```

Ainsi a été créée une colonne d'entiers dont la valeur par défaut est assignée par un générateur de séquence. Une contrainte `NOT NULL` est ajoutée pour s'assurer qu'une valeur NULL ne puisse pas être insérée. (Dans la plupart des cas, une contrainte `UNIQUE` ou `PRIMARY KEY` peut être ajoutée pour interdire que des doublons soient créés par accident, mais ce n'est pas automatique.) Enfin, la séquence est marquée « owned by » (possédée par) la colonne pour qu'elle soit supprimée si la colonne ou la table est supprimée.

Note

Comme `smallserial`, `serial` et `bigserial` sont implémentés en utilisant des séquences, il peut y avoir des trous dans la séquence de valeurs qui apparait dans la colonne, même si aucune ligne n'est jamais supprimée. Une valeur allouée à partir de la séquence est toujours utilisée même si la ligne contenant cette valeur n'est pas insérée avec succès dans la colonne de la table. Cela peut survenir si la transaction d'insertion est annulée. Voir `nextval()` dans Section 9.17 pour plus de détails.

Pour insérer la valeur suivante de la séquence dans la colonne `serial`, il faut préciser que la valeur par défaut de la colonne doit être utilisée. Cela peut se faire de deux façons : soit en excluant cette colonne de la liste des colonnes de la commande `INSERT`, soit en utilisant le mot-clé `DEFAULT`.

Les types `serial` et `serial4` sont identiques : ils créent tous les deux des colonnes `integer`. Les types `bigserial` et `serial8` fonctionnent de la même façon, mais créent des colonnes `bigint`. `bigserial` doit être utilisé si plus de 2^{31} identifiants sont prévus sur la durée de vie de la table. Les noms de type `smallserial` et `serial2` fonctionnent de la même façon, sauf qu'ils créent une colonne de type `smallint`.

La séquence créée pour une colonne `serial` est automatiquement supprimée quand la colonne correspondante est supprimée. La séquence peut être détruite sans supprimer la colonne, mais la valeur par défaut de la colonne est alors également supprimée.

8.2. Types monétaires

Le type `money` stocke un montant en devise avec un nombre fixe de décimales. Voir le Tableau 8.3. La précision de la partie fractionnée est déterminée par le paramètre lc_monetary de la base de données. L'échelle indiquée dans la table suppose qu'il y a deux chiffres dans la partie fractionnée. De nombreux formats sont acceptés en entrée, dont les entiers et les nombres à virgule flottante, ainsi que les formats classiques de devises, comme `'$1,000.00'`. Le format de sortie est généralement dans le dernier format, mais dépend de la locale.

Tableau 8.3. Types monétaires

Nom	Taille de stockage	Description	Étendue
`money`	8 octets	montant monétaire	-92233720368547758.08 à +92233720368547758.07

Comme la sortie de type de données est sensible à la locale, la recharge de données de type `money` dans une base de données pourrait ne pas fonctionner si la base a une configuration différente pour `lc_monetary`. Pour éviter les problèmes, avant de restaurer une sauvegarde dans une nouvelle base de données, assurez-vous que `lc_monetary` a la même valeur ou une valeur équivalente à celle de la base qui a été sauvegardée.

Les valeurs de types `numeric`, `int` et `bigint` peuvent être converties en type `money`. La conversion à partir du type `real` et `double precision` peut être faite en convertissant tout d'abord vers le type `numeric`. Par exemple :

```
SELECT '12.34'::float8::numeric::money;
```

Néanmoins, ce n'est pas recommandé. Les nombres à virgules flottantes ne doivent pas être utilisés pour gérer de la monnaie à cause des erreurs potentielles d'arrondis.

Une valeur `money` peut être convertie en `numeric` sans perdre de précision. Les conversions vers d'autres types peuvent potentiellement perdre en précision et doivent aussi se faire en deux étapes :

```
SELECT '52093.89'::money::numeric::float8;
```

La division d'une valeur de type `money` par une valeur de type entier est réalisée en tronquant la partie décimale. Pour obtenir un résultat arrondi, il faut diviser par une valeur en virgule flottante ou convertir la valeur de type `money` en `numeric` avant de réaliser la division. Il faudra ensuite convertir vers le type `money`. (Cette dernière méthode est préférable pour éviter de perdre en précision.) Quand une valeur de type `money` est divisée par une autre valeur de type `money`, le résultat est du type `double precision` (c'est-à-dire un nombre pur, pas une monnaie). Les unités de monnaie s'annulent dans la division.

8.3. Types caractère

Tableau 8.4. Types caractère

Nom	Description
`character varying(`*n*`)`, `varchar(`*n*`)`	Longueur variable avec limite
`character(`*n*`)`, `char(`*n*`)`	longueur fixe, complété par des espaces
`text`	longueur variable illimitée

Le Tableau 8.4 présente les types génériques disponibles dans PostgreSQL.

SQL définit deux types de caractères principaux : `character varying(`*n*`)` et `character(`*n*`)` où *n* est un entier positif. Ces deux types permettent de stocker des chaînes de caractères de taille inférieure ou égale à *n* (ce ne sont pas des octets). Toute tentative d'insertion d'une chaîne plus longue conduit à une erreur, à moins que les caractères en excès ne soient tous des espaces, auquel cas la chaîne est tronquée à la taille maximale (cette exception étrange est imposée par la norme SQL). Si la chaîne à stocker est plus petite que la taille déclarée, les valeurs de type `character` sont complétées par des espaces, celles de type `character varying` sont stockées en l'état.

Si une valeur est explicitement transtypée en `character varying(`*n*`)` ou en `character(`*n*`)`, une valeur trop longue est tronquée à *n* caractères sans qu'aucune erreur ne soit levée (ce comportement est aussi imposé par la norme SQL.)

Les notations `varchar(`*n*`)` et `char(`*n*`)` sont des alias de `character varying(`*n*`)` et `character(`*n*`)`, respectivement. Si indiqué, la longueur doit être supérieur à zéro et ne peut pas excéder 10485760. `character` sans indication de taille est équivalent à `character(1)`. Si `character varying` est utilisé sans indicateur de taille, le type accepte des chaînes de toute taille. Il s'agit là d'une spécificité de PostgreSQL.

De plus, PostgreSQL propose aussi le type `text`, qui permet de stocker des chaînes de n'importe quelle taille. Bien que le type `text` ne soit pas dans le standard SQL, plusieurs autres systèmes de gestion de bases de données SQL le proposent également.

Les valeurs de type `character` sont complétées physiquement à l'aide d'espaces pour atteindre la longueur *n* indiquée. Ces valeurs sont également stockées et affichées de cette façon. Cependant, les espaces de remplissage sont traités comme sémantiquement non significatifs et sont donc ignorés lors de la comparaison de deux valeurs de type `character`. Dans les collationnements où les espaces de remplissage sont significatifs, ce comportement peut produire des résultats inattendus, par exemple `SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2)` retourne vrai, même si la locale `C` considérerait qu'un espace est plus grand qu'un retour chariot. Les espaces de remplissage sont supprimés lors de la conversion d'une valeur `character` vers l'un des autres types chaîne. Ces espaces *ont* une signification sémantique pour les valeurs de type `character varying` et `text`, et lors de l'utilisation de la correspondance de motifs, par exemple avec `LIKE` ou avec les expressions rationnelles.

Les caractères pouvant être enregistrés dans chacun de ces types de données sont déterminés par le jeu de caractères de la base de données, qui a été sélectionné à la création de la base. Quelque soit le jeu de caractères spécifique, le caractère de code zéro (quelque fois appelé NUL) ne peut être enregistré. Pour plus d'informations, voir Section 24.3.

L'espace nécessaire pour une chaîne de caractères courte (jusqu'à 126 octets) est de un octet, plus la taille de la chaîne qui inclut le remplissage avec des espaces dans le cas du type `character`. Les chaînes plus longues ont quatre octets d'en-tête au lieu d'un seul. Les chaînes longues sont automatiquement compressées par le système, donc le besoin pourrait être moindre. Les chaînes vraiment très longues sont stockées dans des tables supplémentaires, pour qu'elles n'empêchent pas d'accéder rapidement à des valeurs plus courtes. Dans tous les cas, la taille maximale possible pour une chaîne de caractères est de l'ordre de 1 Go. (La taille maximale pour *n* dans la déclaration de type est

inférieure. Il ne sert à rien de modifier ce comportement, car avec les encodages sur plusieurs octets, les nombres de caractères et d'octets peuvent être très différents. Pour stocker de longues chaînes sans limite supérieure précise, il est préférable d'utiliser les types `text` et `character varying` sans taille, plutôt que d'indiquer une limite de taille arbitraire.)

> **Astuce**
>
> Il n'y a aucune différence de performance parmi ces trois types, si ce n'est la place disque supplémentaire requise pour le type à remplissage et quelques cycles CPU supplémentaires pour vérifier la longueur lors du stockage dans une colonne contrainte par la taille. Bien que `character(`*n*`)` ait des avantages en termes de performance sur certains autres systèmes de bases de données, il ne dispose pas de ce type d'avantages dans PostgreSQL ; en fait, `character(`*n*`)` est habituellement le plus lent des trois à cause des coûts de stockage supplémentaires. Dans la plupart des situations, les types `text` et `character varying` peuvent être utilisés à leur place.

On peut se référer à la Section 4.1.2.1 pour obtenir plus d'informations sur la syntaxe des libellés de chaînes, et le Chapitre 9 pour des informations complémentaires sur les opérateurs et les fonctions.

Exemple 8.1. Utilisation des types caractère

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- 1

  a   | char_length
------+-------------
 ok   |           2


CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('bien      ');
INSERT INTO test2 VALUES ('trop long');
ERROR:  value too long for type character varying(5)
INSERT INTO test2 VALUES ('trop long'::varchar(5)); -- troncature
 explicite
SELECT b, char_length(b) FROM test2;

   b   | char_length
-------+-------------
 ok    |           2
 bien  |           5
 trop  |           5
```

1 La fonction `char_length` est décrite dans la Section 9.4.

Il y a deux autres types caractère de taille fixe dans PostgreSQL. Ils sont décrits dans le Tableau 8.5. Ils ne sont pas destinés à une utilisation générale, mais seulement par les catalogues systèmes internes. Le type `name` est utilisé pour le stockage des identifiants. Sa taille est actuellement définie à 64 octets (63 utilisables plus le terminateur), mais doit être référencée en utilisant la constante `NAMEDATALEN` en code source `C`. La taille est définie à la compilation (et est donc ajustable pour des besoins particuliers). La taille maximale par défaut peut éventuellement être modifiée dans une prochaine version. Le type `"char"` (attention aux guillemets) est différent de `char(1)` car il n'utilise qu'un seul octet de stockage et, de ce fait ne peut stocker qu'un seul caractère ASCII. Il est utilisé dans les catalogues système comme un type d'énumération simpliste.

Tableau 8.5. Types caractères spéciaux

Nom	Taille de stockage	Description
`"char"`	1 octet	type interne d'un octet
`name`	64 octets	type interne pour les noms d'objets

8.4. Types de données binaires

Le type de données `bytea` permet de stocker des chaînes binaires ; voir le Tableau 8.6.

Tableau 8.6. Types de données binaires

Nom	Espace de stockage	Description
`bytea`	un à quatre octets plus la taille de la chaîne binaire à stocker	Chaîne binaire de longueur variable

Une chaîne binaire est une séquence d'octets. Les chaînes binaires se distinguent des chaînes de caractères de deux façons : tout d'abord, les chaînes binaires permettent de stocker des octets de valeurs zéro ainsi que les autres caractères « non imprimables » (habituellement, les octets en dehors de l'intervalle décimal de 32 à 126). Les chaînes de caractères interdisent les octets de valeur zéro et interdisent aussi toute valeur d'octet ou séquence d'octets invalide selon l'encodage sélectionné pour la base de données. Ensuite, les opérations sur les chaînes binaires traitent réellement les octets alors que le traitement de chaînes de caractères dépend de la configuration de la locale. En résumé, les chaînes binaires sont appropriées pour le stockage de données que le développeur considère comme des « octets bruts », alors que les chaînes de caractères sont appropriées pour le stockage de texte.

Le type `bytea` accepte deux formats en entrée et en sortie le format « hex » et le format historique de PostgreSQL, « escape ». Les deux sont acceptés en entrée. Le format de sortie dépend du paramètre de configuration bytea_output ; ce dernier sélectionne par défaut le format hexadécimal. (Notez que le format hexadécimal est disponible depuis PostgreSQL 9.0 ; les versions antérieures et certains outils ne le comprennent pas.)

Le standard SQL définit un type de chaîne binaire différent, appelé `BLOB` ou `BINARY LARGE OBJECT`. Le format en entrée est différent du `bytea`, mais les fonctions et opérateurs fournis sont pratiquement les mêmes.

8.4.1. Le format hexadécimal `bytea`

Le format « hex » code les données binaires sous la forme de deux chiffres hexadécimaux par octet, le plus significatif en premier. La chaîne complète est précédée par la séquence `\x` (pour la distinguer du format d'échappement). Dans certains cas, l'antislash initial peut avoir besoin d'être échappé par un doublage du caractère (voir Section 4.1.2.1). En saisie, les chiffres hexadécimaux peuvent être soit en majuscules, soit en minuscules, et les espaces blancs sont permis entre les paires de chiffres (mais pas à l'intérieur d'une paire ni dans la séquence `\x` de début). Le format hexadécimal est compatible avec une grande variété d'applications et de protocoles externes, et il a tendance à être plus rapide à convertir que le format d'échappement. Son utilisation est donc préférée.

Exemple :

```
SET bytea_output = 'hex';

SELECT '\xDEADBEEF'::bytea;
   bytea
------------
 \xdeadbeef
```

8.4.2. Le format d'échappement bytea

Le format d'échappement (« escape ») est le format traditionnel de PostgreSQL pour le type bytea. Son approche est de représenter une chaîne binaire comme un séquence de caractères ASCII et de convertir les données qui ne peuvent pas être représentées en ASCII en une séquence spéciale d'échappement. Si, du point de vue de l'application, représenter les octets sous la forme de caractères revêt un sens, alors cette représentation est intéressante. En pratique, c'est généralement source de confusion, car cela diminue la distinction entre chaînes binaires et chaînes textuelles. De plus, le mécanisme particulier de l'échappement qui a été choisi est quelque peu complexe. Donc ce format devrait probablement être évité pour la plupart des nouvelles applications.

Lors de la saisie de valeurs bytea dans le format d'échappement, les octets de certaines valeurs *doivent* être échappés alors que les autres valeurs d'octets *peuvent* être échappés. En général, pour échapper un octet, il suffit de le convertir dans sa valeur octale composée de trois chiffres et de la faire précéder d'un antislash (ou de deux antislashs s'il faut utiliser la syntaxe d'échappement de chaînes). L'antislash lui-même (octet en valeur décimal, 92) peut alternativement être représenté par un double antislash. Le Tableau 8.7 affiche les caractères qui doivent être échappés et donne les séquences d'échappement possibles.

Tableau 8.7. Octets littéraux bytea à échapper

Valeur décimale de l'octet	Description	Représentation échappée en entrée	Exemple	Représentation hexadécimale
0	octet zéro	`'\000'`	`'\000'::bytea;`	`\x00`
39	apostrophe	`''''` ou `'\047'`	`''''::bytea;`	`\x27`
92	antislash	`'\\'` or `'\134'`	`'\\'::bytea;`	`\x5c`
de 0 à 31 et de 127 à 255	octets « non affichables »	`'\`*xxx*`'` (valeur octale)	`'\001'::bytea;`	`\x01`

La nécessité d'échapper les octets *non affichables* dépend des paramétrages de la locale. Il est parfois possible de s'en sortir sans échappement.

La raison pour laquelle les guillemets simples doivent être doublés, comme indiqué dans Tableau 8.7, est que cela est vrai pour toute chaîne litérale dans une commande SQL. L'analyseur générique des chaînes litérales utilise les guillemets simples externes et réduit toute paire de guillemets simples en un seul caractère. La fonction en entrée du type bytea ne voit qu'un guillemet simple, qu'il traire comme un caractère standard. Néanmoins, la fonction en entrée du type bytea traite les antislashs de façon spéciale et les autres comportements montrés dans Tableau 8.7 sont implémentés par cette fonction.

Dans certains contextes, les antislashs doivent être doublés par rapport à ce qui est montré ci-dessus car l'analyseur générique de chaîne litérale réduira aussi les paires d'antislashs en un seul caractère de données ; voir Section 4.1.2.1.

Les octets Bytea sont affichés par défaut dans le format hex. Si vous modifiez bytea_output à escape, les octets « non affichables » sont convertis dans leur équivalent sous la forme d'une valeur octale à trois chiffres et précédé d'un antislash. La plupart des octets « affichables » sont affichés dans leur représentation standard pour le jeu de caractères du client :

```
SET bytea_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124'::bytea;
```

```
     bytea
----------------
 abc klm *\251T
```

L'octet de valeur décimale 92 (antislash) est doublé en sortie. Les détails sont dans le Tableau 8.8.

Tableau 8.8. Octets échappés en sortie pour `bytea`

Valeur décimale de l'octet	Description	Représentation de sortie échappée	Exemple	Résultat en sortie
92	antislash	`\\`	`'\134'::bytea;`	`\\`
0 à 31 et 127 à 255	octets« non affichables »	`\`*xxx* (valeur octale)	`'\001'::bytea;`	`\001`
32 à 126	octets « affichables »	Représentation dans le jeu de caractères du client	`'\176'::bytea;`	~

En fonction de l'interface utilisée pour accéder à PostgreSQL, un travail supplémentaire d'échappement/de « déséchappement » des chaînes `bytea` peut être nécessaire. Il faut également échapper les sauts de lignes et retours à la ligne si l'interface les traduit automatiquement, par exemple.

8.5. Types date/heure

PostgreSQL supporte l'ensemble des types date et heure du SQL. Ces types sont présentés dans le Tableau 8.9. Les opérations disponibles sur ces types de données sont décrites dans la Section 9.9. Les dates sont comptées suivant le calendrier grégorien, même dans le cas des dates antérieures à l'introduction du calendrier (voir) Section B.6 pour plus d'informations).

Tableau 8.9. Types date et heure

Nom	Taille de stockage	Description	Valeur minimale	Valeur maximale	Résolution
`timestamp [ (`*p*`) ] [ without time zone ]`	8 octets	date et heure (sans fuseau horaire)	4713 avant JC	294276 après JC	1 microseconde
`timestamp [ (`*p*`) ] with time zone`	8 octets	date et heure, avec fuseau horaire	4713 avant JC	294276 après JC	1 microseconde
`date`	4 octets	date seule (pas d'heure)	4713 avant JC	5874897 après JC	1 jour
`time [ (`*p*`) ] [ without time zone ]`	8 octets	heure seule (pas de date)	00:00:00.00	24:00:00	1 microseconde
`time [ (`*p*`) ]`	12 octets	heure (sans date), avec fuseau horaire	00:00:00+1559	24:00:00-1559	1 microseconde

Nom	Taille de stockage	Description	Valeur minimale	Valeur maximale	Résolution
`with time zone`					
`interval [ champs ] [ (p) ]`	16 octets	intervalles de temps	-178000000 années	178000000 années	1 microseconde

> **Note**
>
> Le standard SQL impose que `timestamp` soit un équivalent de `timestamp without time zone`. `timestamptz` est accepté comme abréviation pour `timestamp with time zone` ; c'est une extension PostgreSQL.

`time`, `timestamp`, et `interval` acceptent une précision optionnelle *p*, qui indique le nombre de décimales pour les secondes. Il n'y a pas, par défaut, de limite explicite à cette précision. Les valeurs acceptées pour *p* s'étendent de 0 à 6.

Le type `interval` a une option supplémentaire, qui permet de restreindre le jeu de champs stockés en écrivant une de ces expressions :

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

Notez que si *champs* et *p* sont tous les deux indiqués, *champs* doit inclure `SECOND`, puisque la précision s'applique uniquement aux secondes.

Le type `time with time zone` est défini dans le standard SQL, mais sa définition lui prête des propriétés qui font douter de son utilité. Dans la plupart des cas, une combinaison de `date`, `time`, `timestamp without time zone` et `timestamp with time zone` devrait permettre de résoudre toutes les fonctionnalités de date et heure nécessaires à une application.

8.5.1. Saisie des dates et heures

La saisie de dates et heures peut se faire dans la plupart des formats raisonnables, dont ISO8601, tout format compatible avec SQL, le format POSTGRES traditionnel ou autres. Pour certains formats, l'ordre des jours, mois et années en entrée est ambigu. Il est alors possible de préciser l'ordre attendu pour ces champs. Le paramètre datestyle peut être positionné à `MDY` pour choisir une interprétation mois-jour-année, à `DMY` pour jour-mois-année ou à `YMD` pour année-mois-jour.

PostgreSQL est plus flexible que la norme SQL ne l'exige pour la manipulation des dates et des heures. Voir l'Annexe B pour connaître les règles exactes de reconnaissance des dates et heures et les formats reconnus pour les champs texte comme les mois, les jours de la semaine et les fuseaux horaires.

Tout libellé de date ou heure saisi doit être placé entre apostrophes, comme les chaînes de caractères. La Section 4.1.2.7 peut être consultée pour plus d'information. SQL requiert la syntaxe suivante :

```
type [ (p) ] 'valeur'
```

où *p*, précision optionnelle, est un entier correspondant au nombre de décimales du champ secondes. La précision peut être spécifiée pour les types `time`, `timestamp` et `interval`, et peut aller de 0 à 6. Si aucune précision n'est indiquée dans une déclaration de constante, celle de la valeur littérale est utilisée (mais pas plus de 6 chiffres).

8.5.1.1. Dates

Le Tableau 8.10 regroupe les formats de date possibles pour la saisie de valeurs de type `date`.

Tableau 8.10. Saisie de date

Exemple	Description
1999-01-08	ISO-8601 ; 8 janvier, quel que soit le mode (format recommandé)
January 8, 1999	sans ambiguïté quel que soit le style de date (`datestyle`)
1/8/1999	8 janvier en mode `MDY` ; 1er août en mode `DMY`
1/18/1999	18 janvier en mode `MDY` ; rejeté dans les autres modes
01/02/03	2 janvier 2003 en mode `MDY` ; 1er février 2003 en mode `DMY` ; 3 février 2001 en mode `YMD`
1999-Jan-08	8 janvier dans tous les modes
Jan-08-1999	8 janvier dans tous les modes
08-Jan-1999	8 janvier dans tous les modes
99-Jan-08	8 janvier en mode `YMD`, erreur sinon
08-Jan-99	8 janvier, sauf en mode `YMD` : erreur
Jan-08-99	8 janvier, sauf en mode `YMD` : erreur
19990108	ISO-8601 ; 8 janvier 1999 dans tous les modes
990108	ISO-8601 ; 8 janvier 1999 dans tous les modes
1999.008	Année et jour de l'année
J2451187	Date du calendrier Julien
January 8, 99 BC	Année 99 avant Jésus Christ

8.5.1.2. Heures

Les types « heure du jour » sont `time [ (p) ] without time zone` et `time [ (p) ] with time zone`. `time` est équivalent à `time without time zone`.

Les saisies valides pour ces types sont constituées d'une heure suivie éventuellement d'un fuseau horaire (voir le Tableau 8.11 et le Tableau 8.12). Si un fuseau est précisé pour le type `time without time zone`, il est ignoré sans message d'erreur. Si une date est indiquée, elle est ignorée, sauf si un fuseau horaire impliquant une règle de changement d'heure (heure d'été/heure d'hiver) est précisé, `America/New_York` par exemple. Dans ce cas, la date est nécessaire pour pouvoir déterminer la règle de calcul de l'heure qui s'applique. Le décalage approprié du fuseau horaire est enregistré dans la valeur de `time with time zone` et est affiché de la façon dont il est stocké ; il n'est pas converti vers le fuseau horaire actif.

Tableau 8.11. Saisie d'heure

Exemple	Description
`04:05:06.789`	ISO 8601

Exemple	Description
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	Identique à 04:05 ; AM n'affecte pas la valeur
04:05 PM	Identique à 16:05 ; l'heure doit être <= 12
04:05:06.789-8	ISO 8601, avec le décalage UTC comme fuseau horaire
04:05:06-08:00	ISO 8601, avec le décalage UTC comme fuseau horaire
04:05-08:00	ISO 8601, avec le décalage UTC comme fuseau horaire
040506+0730	ISO 8601, avec le décalage UTC avec un fuseau horaire en heure fractionnée
040506+07:30:00	décalage UTC exprimé en secondes (non autorisé dans ISO 8601)
040506-08	ISO 8601
04:05:06 PST	fuseau horaire abrégé
2003-04-12 04:05:06 America/New_York	fuseau horaire en nom complet

Tableau 8.12. Saisie des fuseaux horaires

Exemple	Description
PST	Abréviation pour l'heure standard du Pacifique (Pacific Standard Time)
America/New_York	Nom complet du fuseau horaire
PST8PDT	Nommage POSIX du fuseau horaire
-8:00:00	Décalage UTC pour la zone PST
-8:00	Décalage ISO-8601 pour la zone PST (format étendu ISO 8601)
-800	Décalage ISO-8601 pour la zone PST (format basique ISO 8601)
-8	Décalage ISO-8601 pour la zone PST (format basique ISO 8601)
zulu	Abréviation militaire de GMT
z	Version courte de zulu (aussi dans ISO 8601)

La Section 8.5.3 apporte des précisions quant à la façon d'indiquer les fuseaux horaires.

8.5.1.3. Horodatage

Les saisies valides sont constituées de la concaténation d'une date et d'une heure, éventuellement suivie d'un fuseau horaire et d'un qualificatif AD (après Jésus Christ) ou BC (avant Jésus Christ). (AD/BC peut aussi apparaître avant le fuseau horaire, mais ce n'est pas l'ordre préféré.) Ainsi :

```
1999-01-08 04:05:06
```

et :

```
1999-01-08 04:05:06 -8:00
```

sont des valeurs valides, qui suivent le standard ISO 8601. Le format très courant :

```
January 8 04:05:06 1999 PST
```

est également supporté.

Le standard SQL différencie les libellés `timestamp without time zone` et `timestamp with time zone` par la présence d'un symbole « + » ou d'un « - » et le décalage du fuseau horaire après l'indication du temps. De ce fait, d'après le standard,

```
TIMESTAMP '2004-10-19 10:23:54'
```

est du type `timestamp without time zone` alors que

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

est du type `timestamp with time zone`. PostgreSQL n'examine jamais le contenu d'un libellé avant de déterminer son type. Du coup, il traite les deux ci-dessus comme des valeurs de type `timestamp without time zone`. Pour s'assurer qu'un littéral est traité comme une valeur de type `timestamp with time zone`, il faut préciser explicitement le bon type :

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

Dans une valeur de type `timestamp without time zone`, PostgreSQL ignore silencieusement toute indication de fuseau horaire. C'est-à-dire que la valeur résultante est dérivée des champs date/heure de la valeur en entrée et n'est pas corrigée par le fuseau horaire.

Pour les valeurs de type `timestamp with time zone`, une chaîne en entrée qui inclut un fuseau horaire explicite sera convertie en UTC (*Universal Coordinated Time* ou Temps Universel Coordonné) en utilisant le décalage approprié pour ce fuseau horaire. connu sous le nom de GMT (*Greenwich Mean Time*). Les valeurs saisies avec un fuseau horaire explicite sont converties en UTC à l'aide du décalage approprié. Si aucun fuseau horaire n'est précisé, alors le système considère que la date est dans le fuseau horaire indiqué par le paramètre système TimeZone, et la convertit en UTC en utilisant le décalage de la zone `timezone`. Dans tous les cas, la valeur est enregistrée en UTC en interne et le fuseau horaire indiqué à l'origine ou supposé n'est pas conservé.

Quand une valeur `timestamp with time zone` est affichée, elle est toujours convertie de l'UTC vers le fuseau horaire courant (variable `timezone`), et affichée comme une heure locale. Pour voir l'heure dans un autre fuseau horaire, il faut, soit changer la valeur de `timezone`, soit utiliser la construction `AT TIME ZONE` (voir la Section 9.9.4).

Les conversions entre `timestamp without time zone` et `timestamp with time zone` considèrent normalement que la valeur `timestamp without time zone` utilise le fuseau horaire `timezone`. Un fuseau différent peut être choisi en utilisant `AT TIME ZONE`.

8.5.1.4. Valeurs spéciales

PostgreSQL supporte plusieurs valeurs de dates spéciales, dans un souci de simplification. Ces valeurs sont présentées dans le Tableau 8.13. Les valeurs `infinity` et `-infinity` ont une représentation spéciale dans le système et sont affichées ainsi ; les autres ne sont que des raccourcies de notation convertis en dates/heures ordinaires lorsqu'ils sont lus. (En particulier, `now` et les chaînes relatives sont converties en une valeur de temps spécifique à leur lecture). Toutes ces valeurs doivent être écrites entre simples quotes lorsqu'elles sont utilisées comme des constantes dans les commandes SQL.

Tableau 8.13. Saisie de dates/heures spéciales

Saisie	Types valides	Description
`epoch`	`date`, `timestamp`	1970-01-01 00:00:00+00 (date système zéro d'Unix)

Saisie	Types valides	Description
`infinity`	`date`, `timestamp`	plus tard que toutes les autres dates
`-infinity`	`date`, `timestamp`	plus tôt que toutes les autres dates
`now`	`date`, `time`, `timestamp`	heure de démarrage de la transaction courante
`today`	`date`, `timestamp`	aujourd'hui minuit (`00:00`)
`tomorrow`	`date`, `timestamp`	demain minuit (`00:00`)
`yesterday`	`date`, `timestamp`	hier minuit (`00:00`)
`allballs`	`time`	00:00:00.00 UTC

Les fonctions suivantes, compatibles avec le standard SQL, peuvent aussi être utilisées pour obtenir l'heure courante pour le type de données correspondant : `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, `LOCALTIMESTAMP`. (Voir la Section 9.9.5). Ce sont là des fonctions SQL qui ne sont *pas* reconnues comme chaînes de saisie de données.

> **Attention**
>
> Bien qu'il n'y ait pas de problèmes à utiliser les chaînes `now`, `today`, `tomorrow` et `yesterday` dans des commandes SQL interactives, elles peuvent avoir un comportement surprenant quand la commande est sauvegardée pour une exécution ultérieure, par exemple dans des requêtes préparées, des vues ou des fonctions. La chaîne peut être convertie en une valeur spécifique qui continue à être utilisée bien après qu'elle ne soit obsolète. Dans de tels contextes, utilisez plutôt une des fonctions SQL. Par exemple, `CURRENT_DATE + 1` est plus sûr que `'tomorrow'::date`.

8.5.2. Affichage des dates et heures

Le format de sortie des types date/heure peut être positionné à l'un des quatre formats de date suivants : ISO 8601, SQL (Ingres), traditionnel POSTGRES (date au format Unix date) ou German (germanique). Le format par défaut est le format ISO. (Le standard `SQL` impose l'utilisation du format ISO 8601. Le nom du format d'affichage « SQL » est mal choisi, un accident historique.) Le Tableau 8.14 présente des exemples de chaque format d'affichage. La sortie d'un type `date` ou `time` n'est évidemment composée que de la partie date ou heure, comme montré dans les exemples. Néanmoins, le style POSTGRES affiche seulement les dates dans le format ISO.

Tableau 8.14. Styles d'affichage de date/heure

Spécification de style	Description	Exemple
`ISO`	standard SQL ISO 8601	`1997-12-17 07:37:16-08`
`SQL`	style traditionnel	`12/17/1997 07:37:16.00 PST`
`Postgres`	style original	`Wed Dec 17 07:37:16 1997 PST`
`German`	style régional	`17.12.1997 07:37:16.00 PST`

> **Note**
>
> ISO 8601 spécifie l'utilisation d'une lettre `T` en majuscule pour séparer la date et l'heure. PostgreSQL accepte ce format en entrée. En sortie, il utilise un espace plutôt qu'un `T`, comme

indiqué ci-dessus. C'est à la fois plus lisible et cohérent avec la RFC 3339[1] ainsi qu'avec d'autres systèmes de bases de données.

Dans les styles SQL et POSTGRES, les jours apparaissent avant le mois si l'ordre des champs DMY a été précisé, sinon les mois apparaissent avant les jours (voir la Section 8.5.1 pour savoir comment ce paramètre affecte l'interprétation des valeurs en entrée). Le Tableau 8.15 présente des exemples.

Tableau 8.15. Convention de présentation des dates

Valeur de `datestyle` (style de date)	**Ordre de saisie**	**Exemple d'affichage**
`SQL, DMY`	*jour*/*mois*/*année*	`17/12/1997 15:37:16.00 CET`
`SQL, MDY`	*mois*/*jour*/*année*	`12/17/1997 07:37:16.00 PST`
`Postgres, DMY`	*jour*/*mois*/*année*	`Wed 17 Dec 07:37:16 1997 PST`

Dans le style ISO, le fuseau horaire est toujours affiché sous la forme d'un décalage numérique signé de UTC, avec un signe positif utilisé pour les zones à l'est de Greenwich. Le décalage sera affiché sous la forme *hh* (heures seulement) s'il s'agit d'un nombre intégral d'heures, ou sous la forme *hh*:*mm* s'il s'agit d'un nombre intégral de minutes, et enfin sous la forme *hh*:*mm*:*ss*. (Le troisième cas n'est pas possible pour tout standard moderne de fuseau horaire, mais il peut apparaître en travaillant sur des jours antérieurs à l'adoption des fuseaux horaires standardisés.) Pour les autres styles de dates, le fuseau horaire est affiché comme une abréviation alphabétique si l'une d'entre elles est d'utilisation commune dans le fuseau actuel. Sinon, il apparaît comme un décalage numérique signé dans le format basique ISO 8601 (*hh* ou *hhmm*).

Le style de date/heure peut être sélectionné à l'aide de la commande `SET datestyle`, du paramètre datestyle du fichier de configuration `postgresql.conf` ou par la variable d'environnement `PGDATESTYLE` sur le serveur ou le client.

La fonction de formatage `to_char` (voir Section 9.8) permet de formater les affichages de date/heure de manière plus flexible.

8.5.3. Fuseaux horaires

Les fuseaux horaires et les conventions liées sont influencés par des décisions politiques, pas uniquement par la géométrie de la Terre. Les fuseaux horaires se sont quelque peu standardisés au cours du vingtième siècle, mais continuent à être soumis à des changements arbitraires, particulièrement en respect des règles de changement d'heure (heure d'été/heure d'hiver). PostgreSQL utilise la très répandue base de données de fuseaux horaires IANA (Olson) pour gérer les informations sur les règles historiques de fuseau horaire. Pour les dates se situant dans le futur, PostgreSQL part de l'assomption que les dernières règles connues pour un fuseau continueront à s'appliquer dans le futur.

PostgreSQL se veut compatible avec les définitions standard SQL pour un usage typique. Néanmoins, le standard SQL possède un mélange étrange de types de date/heure et de possibilités. Deux problèmes évidents sont :

- bien que le type `date` ne puisse pas se voir associer un fuseau horaire, le type `heure` peut en avoir un. Les fuseaux horaires, dans le monde réel, ne peuvent avoir de sens qu'associés à une date et à une heure, vu que l'écart peut varier avec l'heure d'été ;

- le fuseau horaire par défaut est précisé comme un écart numérique constant avec l'UTC. Il n'est, de ce fait, pas possible de s'adapter à l'heure d'été ou d'hiver lorsque l'on fait des calculs arithmétiques qui passent les limites de l'heure d'été et de l'heure d'hiver.

[1] https://datatracker.ietf.org/doc/html/rfc3339

Pour éviter ces difficultés, il est recommandé d'utiliser des types date/heure qui contiennent à la fois une date et une heure lorsque les fuseaux horaires sont utilisés. Il est également préférable de *ne pas* utiliser le type `time with time zone`. (Ce type est néanmoins proposé par PostgreSQL pour les applications existantes et pour assurer la compatibilité avec le standard SQL.) PostgreSQL utilise le fuseau horaire local pour tous les types qui ne contiennent qu'une date ou une heure.

Toutes les dates et heures liées à un fuseau horaire sont stockées en interne en UTC. Elles sont converties en heure locale dans le fuseau indiqué par le paramètre de configuration TimeZone avant d'être affichées sur le client.

PostgreSQL permet d'indiquer les fuseaux horaires de trois façons différentes :

- un nom complet de fuseau horaire, par exemple `America/New_York`. Les noms reconnus de fuseau horaire sont listés dans la vue `pg_timezone_names` (voir Section 54.32). PostgreSQL utilise les données IANA pour cela, les mêmes noms sont donc reconnus par de nombreux autres logiciels ;

- une abréviation de fuseau horaire, par exemple `PST`. Une telle indication ne définit qu'un décalage particulier à partir d'UTC, en contraste avec les noms complets de fuseau horaire qui peuvent aussi impliquer un ensemble de dates pour le changement d'heure. Les abréviations reconnues sont listées dans la vue `pg_timezone_abbrevs` (voir Section 54.31). Les paramètres de configuration TimeZone et log_timezone ne peuvent pas être configurés à l'aide d'une abréviation de fuseau horaire, mais ces abréviations peuvent être utilisées dans les saisies de date/heure et avec l'opérateur `AT TIME ZONE` ;

- En plus de noms et abréviations de fuseaux horaires, PostgreSQL acceptera les spécifications de fuseaux horaire au format POSIX, comme décrites dans Section B.5. Cette option n'est normalement pas préférable à utiliser un nom de fuseau horaire, mais pourrait se révéler nécessaire si aucune entrée IANA de fuseau horaire n'est disponible.

Les abréviations représentent un décalage spécifique depuis UTC, alors qu'un grand nombre des noms complets implique une règle de changement d'heure, et donc potentiellement deux décalages UTC. Par exemple, `2014-06-04 12:00 America/New_York` représente minuit à New York, ce qui, pour cette date particulière, sera le fuseau Eastern Daylight Time (UTC-4). Donc `2014-06-04 12:00 EDT` stipule ce moment précis. Mais `2014-06-04 12:00 EST` représente minuit pour le fuseau Eastern Standard Time (UTC-5), quel que soit le changement d'heure en effet à cette date.

Pour compliquer encore plus, certaines juridictions ont utilisé les mêmes abréviations de fuseau horaire pour signifier des décalages UTC différents. Par exemple, Moscow `MSK` correspondait à UTC +3 certaines années et UTC+4 à d'autres. PostgreSQL interprète ces abréviations suivant ce à quoi elles correspondent (ou ont correspondu récemment) pour la date indiquée. Mais, comme le montre l'exemple `EST` ci-dessus, ce n'est pas nécessairement la même chose que l'heure civile locale à ce moment.

Dans tous les cas, les noms et les abréviations des fuseaux horaires sont insensibles à la casse. (C'est un changement par rapport aux versions de PostgreSQL antérieures à la 8.2 qui étaient sensibles à la casse dans certains cas et pas dans d'autres.)

Ni les noms ni les abréviations des fuseaux horaires ne sont codés en dur dans le serveur ; ils sont obtenus à partir des fichiers de configuration stockés sous `.../share/timezone/` et `.../share/timezonesets/` du répertoire d'installation (voir Section B.4).

Le paramètre de configuration TimeZone peut être fixé dans le fichier `postgresql.conf` ou par tout autre moyen standard décrit dans le Chapitre 20. Il existe aussi quelques manières spéciales de le configurer :

- la commande SQL `SET TIME ZONE` configure le fuseau horaire pour une session. C'est une autre façon d'indiquer `SET TIMEZONE TO` avec une syntaxe plus compatible avec les spécifications SQL ;

- la variable d'environnement PGTZ est utilisée par les applications clientes fondées sur libpq pour envoyer une commande SET TIME ZONE au serveur lors de la connexion.

8.5.4. Saisie d'intervalle

Les valeurs de type interval peuvent être saisies en utilisant la syntaxe verbeuse suivante :

```
[@] quantité
unité [quantité
unité...]
[direction]
```

où *quantité* est un nombre (éventuellement signé) ; *unité* est microsecond millisecond, second, minute, hour, day, week, month, year, decade, century, millennium, ou des abréviations ou pluriels de ces unités ; *direction* peut être ago (pour indiquer un intervalle négatif) ou vide. Le signe @ est du bruit optionnel. Les quantités de chaque unité différente sont implicitement ajoutées, avec prise en compte appropriée des signes (+ et -). ago inverse tous les champs. Cette syntaxe est aussi utilisée pour les sorties d'intervalles, si IntervalStyle est positionné à postgres_verbose.

Les quantités de jours, heures, minutes et secondes peuvent être spécifiées sans notations explicites d'unités. Par exemple '1 12:59:10' est comprise comme '1 day 12 hours 59 min 10 sec'. Par ailleurs, une combinaison d'années et de mois peut être spécifiée avec un tiret ; par exemple, '200-10' est compris comme '200 years 10 months'. (Ces formes raccourcies sont en fait les seules autorisées par le standard SQL, et sont utilisées pour la sortie quand la variable IntervalStyle est positionnée à sql_standard.)

Les valeurs d'intervalles peuvent aussi être écrites en tant qu'intervalles de temps ISO 8601, en utilisant soit le « format avec désignateurs » de la section 4.4.3.2 ou le « format alternatif » de la section 4.4.3.3. Le format avec désignateurs ressemble à ceci :

```
P quantité unité [ quantité unité ...] [ T [ quantité unité ...]]
```

La chaîne doit commencer avec un P, et peut inclure un T qui introduit les unités de ce type. Les abréviations d'unité disponibles sont données dans Tableau 8.16. Des unités peuvent être omises, et peuvent être spécifiées dans n'importe quel ordre, mais les unités inférieures à un jour doivent apparaître après T. En particulier, la signification de M dépend de son emplacement, c'est-à-dire avant ou après T.

Tableau 8.16. Abréviations d'unités d'intervalle ISO 8601

Abréviation	Signification
Y	Années
M	Mois (dans la zone de date)
W	Semaines
D	Jours
H	Heures
M	Minutes (dans la zone de temps)
S	Secondes

Dans le format alternatif :

```
P [ années-mois-jours ] [ T heures:minutes:secondes ]
```

la chaîne doit commencer par `P`, et un `T` sépare la zone de date et la zone de temps de l'intervalle. Les valeurs sont données comme des nombres, de façon similaire aux dates ISO 8601.

Lors de l'écriture d'une constante d'intervalle avec une spécification de *`champs`*, ou lors de l'assignation d'une chaîne à une colonne d'intervalle qui a été définie avec une spécification de *`champs`*, l'interprétation de quantité sans unité dépend des *`champs`*. Par exemple, `INTERVAL '1' YEAR` est interprété comme 1 an, alors que `INTERVAL '1'` est interprété comme 1 seconde. De plus, les valeurs du champ « à droite » du champ le moins significatif autorisé par la spécification de *`champs`* sont annulées de façon silencieuse. Par exemple, écrire `INTERVAL '1 day 2:03:04' HOUR TO MINUTE` implique la suppression du champ des secondes, mais pas celui des journées.

D'après le standard SQL, toutes les valeurs de tous les champs d'un intervalle doivent avoir le même signe, ce qui entraîne qu'un signe négatif initial s'applique à tous les champs ; par exemple, le signe négatif dans l'expression d'intervalle `'-1 2:03:04'` s'applique à la fois aux jours et aux heures/minutes/secondes. PostgreSQL permet que les champs aient des signes différents, et traditionnellement traite chaque champ de la représentation textuelle comme indépendamment signé, ce qui fait que la partie heure/minute/seconde est considérée comme positive dans l'exemple. Si `IntervalStyle` est positionné à `sql_standard`, alors un signe initial est considéré comme s'appliquant à tous les champs (mais seulement si aucun autre signe n'apparaît). Sinon, l'interprétation traditionnelle de PostgreSQL est utilisée. Pour éviter les ambiguïtés, il est recommandé d'attacher un signe explicite à chaque partie, si au moins un champ est négatif.

En interne, les valeurs `interval` sont enregistrées sous la forme de trois champs entiers : mois, jours et microsecondes. Ces champs sont gardés séparément car le nomnbre de jours varie dans un mois, alors qu'un jour peut avoir 23 ou 25 heures si un changement d'heure survient. Une chaîne en entrée pour un interval qui utilise d'autres unités est normalisée dans ce format, puis reconstruit d'une façon standardisée en sortie. Par exemple :

```
SELECT '2 years 15 months 100 weeks 99 hours 123456789
 milliseconds'::interval;
               interval
--------------------------------------
 3 years 3 mons 700 days 133:17:36.789
```

Ici les semaines, qui sont compris comme sept jours (« 7 days »), sont séparées et les unités de temps plus petites et plus grandes sont combinées et normalisées.

Les valeurs des champs en entrée peuvent avoir des parties fractionnelles : par exemple, `'1.5 weeks'` ou `'01:02:03.45'`. Néanmoins, comme le type `interval` stocke en interne seulement les trois unités sous forme d'entier (mois, jours, microsecondes), les unités fractionelles doivent être divisées en plus petites unités. Les parties fractionnelles des unités supérieures aux mois sont arrondies en un nombre entier de mois, par exemple `'1.5 years'` devient `'1 year 6 mons'`. Les parties fractionnelles des semaines et jours sont calculées comme un nombre entier de jours et de microsecondes, en supposant 30 jours par mois et 24 heures par jour, par exemple `'1.75 months'` devient `1 mon 22 days 12:00:00`. Seules les secondes seront affichées en fractionné en sortie.

Tableau 8.17 présente des exemples de saisies d'`interval` valides.

Tableau 8.17. Saisie d'intervalle

Exemple	Description
`1-2`	Format SQL standard : 1 an 2 mois
`3 4:05:06`	Format SQL standard : 3 jours 4 heures 5 minutes 6 secondes

Exemple	Description
`1 year 2 months 3 days 4 hours 5 minutes 6 seconds`	Format PostgreSQL traditionnel : 1 an 2 mois 3 jours 4 heures 5 minutes 6 secondes
`P1Y2M3DT4H5M6S`	« format avec désignateurs » ISO 8601 : signification identique à ci-dessus
`P0001-02-03T04:05:06`	« format alternatif » ISO 8601 : signification identique à ci-dessus

8.5.5. Affichage d'intervalles

Comme indiqué précédemment, PostgreSQL enregistre les valeurs de type `interval` sous le forme de mois, jours et microsecondes. En sortie, le champ `months` est converti en années et mois en divisant par 12. Le champ `days` est affiché tel quel. Le champ `microseconds` est converti en heures, minutes, secondes et secondes fractionnelles. De ce fait, les mois, minutes et secondes ne dépasseront jamais les intervalles, respectivement, 0–11, 0–59 et 0–59, alors que les années, jours et heures affichées peuvent être très larges. (Les fonctions `justify_days` et `justify_hours` peuvent être utilisées s'il est souhaitable de transposer un grand nombre de jours ou heures dans le champ supérieur suivant.)

Le format de sortie du type interval peut être positionné à une de ces quatre valeurs : `sql_standard`, `postgres`, `postgres_verbose` ou `iso_8601`, en utilisant la commande `SET intervalstyle`. La valeur par défaut est le format `postgres`. Tableau 8.18 donne des exemples de chaque style de format de sortie.

Le style `sql_standard` produit une sortie qui se conforme à la spécification du standard SQL pour les chaînes littérales d'intervalle, si la valeur de l'intervalle reste dans les restrictions du standard (soit année-mois seul, ou jour-temps seul, et sans mélanger les composants positifs et négatifs). Sinon, la sortie ressemble au standard littéral année-mois suivi par une chaîne jour-temps littérale, avec des signes explicites ajoutés pour désambiguer les intervalles dont les signes seraient mélangés.

La sortie du style `postgres` correspond à la sortie des versions de PostgreSQL précédant la 8.4, si le paramètre datestyle était positionné à `ISO`.

La sortie du style `postgres_verbose` correspond à la sortie des versions de PostgreSQL précédant la 8.4, si le paramètre datestyle était positionné à autre chose que `ISO`.

La sortie du style `iso_8601` correspond au « format avec designateurs » décrit dans la section 4.4.3.2 du standard ISO 8601.

Tableau 8.18. Exemples de styles d'affichage d'intervalles

Spécification de style	Intervalle année-mois	Intervalle date-temps	Interval Mixte
`sql_standard`	1-2	3 4:05:06	-1-2 +3 -4:05:06
`postgres`	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
`postgres_verbose`	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
`iso_8601`	P1Y2M	P3DT4H5M6S	`P-1Y-2M3DT-4H-5M-6S`

8.6. Type booléen

PostgreSQL fournit le type `boolean` du standard SQL ; voir Tableau 8.19. Ce type dispose de plusieurs états :« true » (vrai), « false » (faux) et un troisième état, « unknown » (inconnu), qui est représenté par la valeur SQL NULL.

Tableau 8.19. Type de données booléen

Nom	Taille du stockage	Description
boolean	1 octet	état vrai ou faux

Les constantes booléennes peuvent être représentées dans les requêtes SQL avec les mots clés SQL TRUE, FALSE et NULL.

La fonction en entrée pour le type boolean accepte ces représentations, sous forme de chaîne de caractères, pour l'état « true » :

```
true
yes
on
1
```

et ces représentations pour l'état « false » :

```
false
no
off
0
```

Les préfixes uniques de ces chaînes sont aussi acceptés, par exemple t ou n. Les espaces avant ou après, ainsi que la casse, sont ignorés.

La fonction en sortie pour le boolean renvoie toujours soit t soit f, comme indiqué dans Exemple 8.2.

Exemple 8.2. Utilisation du type boolean.

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

Les mots clés TRUE et FALSE sont la méthode préférée (compatible SQL) pour l'écriture des constantes booléennes dans les requêtes SQL. Cependant, vous pouvez aussi utiliser les représentations sous forme de chaîne de caractères en suivant la syntaxe générique décrite dans Section 4.1.2.7, par exemple 'yes'::boolean.

Notez que l'analyseur comprend automatiquement que TRUE et FALSE sont du type boolean, mais ce n'est pas le cas pour NULL car il peut avoir tout type. Donc, dans certains contextes, vous devrez convertir explicitement NULL vers le type boolean, par exemple NULL::boolean. À l'inverse, la conversion peut être omise d'une valeur booléenne représentée sous la forme d'une chaîne de caractères dans les contextes où l'analyseur peut déduire que la constante doit être de type boolean.

8.7. Types énumération

Les types énumérés (enum) sont des types de données qui comprennent un ensemble statique, prédéfini de valeurs dans un ordre spécifique. Ils sont équivalents aux types `enum` dans de nombreux langages de programmation. Les jours de la semaine ou un ensemble de valeurs de statut pour un type de données sont de bons exemples de type enum.

8.7.1. Déclaration de types énumérés

Les types enum sont créés en utilisant la commande CREATE TYPE. Par exemple :

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Une fois créé, le type enum peut être utilisé dans des définitions de table et de fonction, comme tous les autres types :

```
CREATE TYPE humeur AS ENUM ('triste', 'ok', 'heureux');
CREATE TABLE personne (
    nom text,
    humeur_actuelle humeur
);
INSERT INTO personne VALUES ('Moe', 'heureux');
SELECT * FROM personne WHERE humeur_actuelle = 'heureux';
 name | humeur_actuelle
------+-----------------
 Moe  | heureux
(1 row)
```

8.7.2. Tri

L'ordre des valeurs dans un type enum correspond à l'ordre dans lequel les valeurs sont créées lors de la déclaration du type. Tous les opérateurs de comparaison et les fonctions d'agrégats relatives peuvent être utilisés avec des types enum. Par exemple :

```
INSERT INTO personne VALUES ('Larry', 'triste');
INSERT INTO personne VALUES ('Curly', 'ok');
SELECT * FROM personne WHERE humeur_actuelle > 'triste';
 nom   | humeur_actuelle
-------+-----------------
 Moe   | heureux
 Curly | ok
(2 rows)

SELECT * FROM personne WHERE humeur_actuelle > 'triste' ORDER BY
 humeur_actuelle;
 nom   | humeur_actuelle
-------+--------------
 Curly | ok
 Moe   | heureux
(2 rows)

SELECT nom
FROM personne
WHERE humeur_actuelle = (SELECT MIN(humeur_actuelle) FROM
 personne);
```

```
 nom
-------
 Larry
(1 row)
```

8.7.3. Sûreté du type

Chaque type de données énuméré est séparé et ne peut pas être comparé aux autres types énumérés. Par exemple :

```
CREATE TYPE niveau_de_joie AS ENUM ('heureux', 'très heureux',
 'ecstatique');
CREATE TABLE vacances (
    nombre_de_semaines integer,
    niveau_de_joie niveau_de_joie
);
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (4,
 'heureux');
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (6,
 'très heureux');
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (8,
 'ecstatique');
INSERT INTO vacances(nombre_de_semaines,niveau_de_joie) VALUES (2,
 'triste');
ERROR:  invalid input value for enum niveau_de_joie: "triste"
SELECT personne.nom, vacances.nombre_de_semaines FROM personne,
 vacances
  WHERE personne.humeur_actuelle = vacances.niveau_de_joie;
ERROR:  operator does not exist: humeur = niveau_de_joie
```

Si vous avez vraiment besoin de ce type de conversion, vous pouvez soit écrire un opérateur personnalisé soit ajouter des conversions explicites dans votre requête :

```
SELECT personne.nom, vacances.nombre_de_semaines FROM personne,
 vacances
  WHERE personne.humeur_actuelle::text =
 vacances.niveau_de_joie::text;
 nom  | nombre_de_semaines
------+--------------------
 Moe  |         4
(1 row)
```

8.7.4. Détails d'implémentation

Les labels enum sont sensibles à la casse, donc `'heureux'` n'est pas identique à `'HEUREUX'`. Les espaces blancs dans les labels sont aussi pris en compte.

Bien que les types enum aient principalement pour but d'être des ensembles statiques de valeurs, il est possible d'ajouter de nouvelles valeurs à un type enum existant et de renommer les valeurs existantes (voir ALTER TYPE). Les valeurs existantes ne peuvent pas être supprimées d'un type enum, pas plus qu'il n'est possible de modifier l'ordre de tri de ces valeurs, si ce n'est en supprimant puis en re- créant le type enum.

Une valeur enum occupe quatre octets sur disque. La longueur du label texte d'une valeur enum est limité au paramètre `NAMEDATALEN` codé en dur dans PostgreSQL ; dans les constructions standard, cela signifie un maximum de 63 octets.

Les traductions des valeurs enum internes vers des labels texte sont gardées dans le catalogue système `pg_enum`. Interroger ce catalogue directement peut s'avérer utile.

8.8. Types géométriques

Les types de données géométriques représentent des objets à deux dimensions. Le Tableau 8.20 liste les types disponibles dans PostgreSQL.

Tableau 8.20. Types géométriques

Nom	Taille de stockage	Description	Représentation
`point`	16 octets	Point du plan	(x,y)
`line`	24 octets	Ligne infinie	{A,B,C}
`lseg`	32 octets	Segment de droite fini	((x1,y1),(x2,y2))
`box`	32 octets	Boîte rectangulaire	((x1,y1),(x2,y2))
`path`	16+16n octets	Chemin fermé (similaire à un polygone)	((x1,y1),...)
`path`	16+16n octets	Chemin ouvert	[(x1,y1),...]
`polygon`	40+16n octets	Polygone (similaire à un chemin fermé)	((x1,y1),...)
`circle`	24 octets	Cercle	<(x,y),r> (point central et rayon)

Pour tous ces types, les coordonnées individuelles sont enregistrées comme des nombres `double precision` (`float8`).

Un large ensemble de fonctions et d'opérateurs permettent d'effectuer différentes opérations géométriques, comme l'échelonnage, la translation, la rotation, la détermination des intersections. Elles sont expliquées dans la Section 9.11.

8.8.1. Points

Les points sont les briques fondamentales des types géométriques. Les valeurs de type `point` sont indiquées à l'aide d'une des syntaxes suivantes :

```
( x , y )
  x , y
```

où *x* et *y* sont les coordonnées respectives sous forme de nombre à virgule flottante.

Les points sont affichés en utilisant la première syntaxe.

8.8.2. Lines

Les lignes sont représentées par l'équation linéaire $Ax + By + C = 0$, où A et B ne valent pas zéro tous les deux. Les valeurs de type `line` sont fournies et récupérées sous la forme suivante :

```
{ A, B, C }
```

Il est également possible d'utiliser n'importe laquelle des formes suivantes pour la saisie :

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

où `(x1,y1)` et `(x2,y2)` sont deux points différents sur la ligne.

8.8.3. Segments de droite

Les segments de ligne sont représentés par des paires de points qui sont les points finaux du segment. Les valeurs de type `lseg` sont précisées en utilisant une des syntaxes suivantes :

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

où `(x1,y1)` et `(x2,y2)` sont les points aux extrémités du segment.

Les segments de ligne sont affichés en utilisant la première syntaxe.

8.8.4. Boîtes

Les boîtes (rectangles) sont représentées par les paires de points des coins opposés de la boîte selon une des syntaxes suivantes :

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

où `(x1,y1)` et `(x2,y2)` sont les coins opposés du rectangle.

Les rectangles sont affichés selon la deuxième syntaxe.

Les deux coins opposés peuvent être fournis en entrée, mais les valeurs seront réordonnées pour stocker les coins en haut à droite et en bas à gauche, dans cet ordre.

8.8.5. Chemins

Les chemins (type `path`) sont représentés par des listes de points connectés. Ils peuvent être *ouverts*, si le premier et le dernier point ne sont pas considérés comme connectés, ou *fermés*, si le premier et le dernier point sont considérés comme connectés.

Les valeurs de type `path` sont saisies selon une des syntaxes suivantes :

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

où les points sont les extrémités des segments de droite qui forment le chemin. Les crochets (`[]`) indiquent un chemin ouvert alors que les parenthèses (`()`) indiquent un chemin fermé. Quand les parenthèses externes sont omises, comme dans les syntaxes trois à cinq, un chemin fermé est utilisé.

Les chemins sont affichés selon la première ou la seconde syntaxe appropriée.

8.8.6. Polygones

Les polygones (type `polygon`) sont représentés par des listes de points (les vertex du polygone). Ils sont très similaires à des chemins fermés ; la différence sémantique essentielle est qu'un polygone est considéré comme incluant l'aire à l'intérieur alors que ce n'est pas le cas pour un chemin.

Une différence d'implémentation importante entre les polygones et les chemins est que la représentation enregistrée d'un polygone inclut sa boite la plus petite. Ceci accélère certaines opérations de recherche, bien que le calcul de la boîte ajoute une surcharge lors de la construction de nouveaux polygones.

Les valeurs de type `polygon` sont saisies selon une des syntaxes suivantes :

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

où les points sont les extrémités des segments de droite qui forment les limites du polygone.

Les polygones sont affichés selon la première syntaxe.

8.8.7. Cercles

Les cercles (type `circle`) sont représentés par un point central et un rayon. Les valeurs de type `circle` sont saisies selon une des syntaxes suivantes :

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

où `(x,y)` est le point central et `r` le rayon du cercle.

Les cercles sont affichés selon la première syntaxe.

8.9. Types adresses réseau

PostgreSQL propose des types de données pour stocker des adresses IPv4, IPv6 et MAC. Ceux-ci sont décrits dans le Tableau 8.21. Il est préférable d'utiliser ces types plutôt que des types texte standard pour stocker les adresses réseau, car ils offrent un contrôle de syntaxe lors de la saisie et plusieurs opérateurs et fonctions spécialisés (voir la Section 9.12).

Tableau 8.21. Types d'adresses réseau

Nom	Taille de stockage	Description
`cidr`	7 ou 19 octets	réseaux IPv4 et IPv6
`inet`	7 ou 19 octets	hôtes et réseaux IPv4 et IPv6
`macaddr`	6 octets	adresses MAC

Nom	Taille de stockage	Description
`macaddr8`	8 bytes	adresses MAC (format EUI-64)

Lors du tri de données de types `inet` ou `cidr`, les adresses IPv4 apparaissent toujours avant les adresses IPv6, y compris les adresses IPv4 encapsulées, comme ::10.2.3.4 ou ::ffff:10.4.3.2.

8.9.1. `inet`

Le type `inet` stocke une adresse d'hôte IPv4 ou IPv6 et, optionnellement, son sous-réseau, le tout dans un seul champ. Le sous-réseau est représenté par le nombre de bits de l'adresse hôte constituant l'adresse réseau (le « masque réseau »). Si le masque réseau est 32 et l'adresse de type IPv4, alors la valeur n'indique pas un sous-réseau, juste un hôte. En IPv6, la longueur de l'adresse est de 128 bits, si bien que 128 bits définissent une adresse réseau unique. Pour n'accepter que des adresses réseau, il est préférable d'utiliser le type `cidr` plutôt que le type `inet`.

Le format de saisie pour ce type est *`adresse/y`* où *`adresse`* est l'adresse la plus basse du réseau en IPv4 ou en IPv6 et *`y`* est le nombre de bits du masque réseau. Si *`y`* est omis, alors le masque vaut 32 pour IPv4 et 128 pour IPv6, et la valeur représente un hôte unique. À l'affichage, la portion *`/y`* est supprimée si le masque réseau indique un hôte unique.

8.9.2. `cidr`

Le type `cidr` stocke une définition de réseau IPv4 ou IPv6. La saisie et l'affichage suivent les conventions Classless Internet Domain Routing. Le format de saisie d'un réseau est *`address/y`* où *`address`* est le réseau représenté sous forme d'une adresse IPv4 ou IPv6 et *`y`* est le nombre de bits du masque réseau. Si *`y`* est omis, il calculé en utilisant les règles de l'ancien système de classes d'adresses, à ceci près qu'il est au moins assez grand pour inclure tous les octets saisis. Saisir une adresse réseau avec des bits positionnés à droite du masque indiqué est une erreur.

Tableau 8.22 présente quelques exemples.

Tableau 8.22. Exemples de saisie de types `cidr`

Saisie `cidr`	Affichage `cidr`	`abbrev(cidr)`
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba/64
2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.9.3. `inet` vs `cidr`

La différence principale entre les types de données `inet` et `cidr` réside dans le fait que `inet` accepte des valeurs avec des bits non nuls à droite du masque de réseau, alors que `cidr` ne l'accepte pas. Par exemple, `192.168.0.1/24` est valide pour `inet`, mais pas pour `cidr`.

> **Astuce**
>
> Les fonctions `host`, `text` et `abbrev` permettent de modifier le format d'affichage des valeurs `inet` et `cidr`.

8.9.4. `macaddr`

Le type `macaddr` stocke des adresses MAC, connues par exemple pour les adresses de cartes réseau Ethernet (mais les adresses MAC sont aussi utilisées dans d'autres cas). Les saisies sont acceptées dans les formats suivants :

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800-2b01-0203'
'08002b010203'
```

Ces exemples indiquent tous la même adresse. Les majuscules et les minuscules sont acceptées pour les chiffres `a` à `f`. L'affichage se fait toujours selon le premier des formats ci-dessus.

Le standard IEEE 802-2001 spécifie la seconde forme affichée (avec les tirets) comme forme canonique pour les adresses MAC, et la première forme (avec les :) comme utilisé avec la notation à bits retournés, MSB en premier, ce qui donne l'équivalence 08-00-2b-01-02-03 = 10:00:D4:80:40:C0. Cette convention est largement ignorée aujourd'hui et n'a de sens que pour des protocoles réseau obsolètes (comme Token Ring). PostgreSQL ne tient pas compte des bits retournés ; tous les formats acceptés utilisent l'ordre canonique LSB.

Les cinq derniers formats ne font partie d'aucun standard.

8.9.5. `macaddr8`

Le type `macaddr8` stocke des adresses MAC au format EUI-64, connu par exemple pour les adresses de cartes réseau Ethernet (mais les adresses MAC sont aussi utilisées dans d'autres cas). Ce type accepte à la fois des adresses MAC d'une longueur de six et huit octets. Les adresses MAC fournies dans un format de six octets seront stockées dans un format de huit octets avec les quatrième et cinquième octets respectivement positionnés à FF et FE. Veuillez noter qu'IPv6 utilise un format modifié de EUI-64 où le septième bit devrait être positionné à un après la conversion depuis EUI-48. La fonction `macaddr8_set7bit` est fournie pour réaliser ce changement. De manière générale, n'importe quelle valeur en entrée constituée de paires de chiffres au format hexadécimal (dans les limites d'un octet), systématiquement séparées ou non d'un de ces caractères `':'`, `'-'` ou `'.'` est acceptée. Le nombre de chiffres hexadécimaux doit être 16 (huit octets) ou 12 (six octets). Les espaces non significatifs présents avant ou après sont ignorés. Voici un ensemble d'exemples de formats acceptés en entrée :

```
'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
'08002b:0102030405'
'08002b-0102030405'
```

```
'0800.2b01.0203.0405'
'0800-2b01-0203-0405'
'08002b01:02030405'
'08002b0102030405'
```

Ces exemples spécifient tous la même adresse. Les majuscules et les minuscules sont acceptées pour les caractères de `a` jusqu'à `f`. La sortie sera toujours au même format que le premier exemple.

Les six derniers formats en entrée qui sont mentionnés au-dessus ne font partie d'aucun standard.

Pour convertir une adresse MAC traditionnelle de 48 bits au format EUI-48 vers le format modifié EUI-64 pour pouvoir être incluse dans la partie hôte d'une adresse IPv6, utilisez `macaddr8_set7bit` comme ceci :

```
SELECT macaddr8_set7bit('08:00:2b:01:02:03');

    macaddr8_set7bit
-------------------------
 0a:00:2b:ff:fe:01:02:03
(1 row)
```

8.10. Type chaîne de bits

Les chaînes de bits sont des chaînes de 0 et de 1. Elles peuvent être utilisées pour stocker ou visualiser des masques de bits. Il y a deux types bits en SQL : `bit(`*n*`)` et `bit varying(`*n*`)`, avec *n* un entier positif.

Les données de type `bit` doivent avoir une longueur de *n* bits exactement. Essayer de lui affecter une chaîne de bits plus longue ou plus courte déclenche une erreur. Les données de type `bit varying` ont une longueur variable, d'au maximum *n* bits ; les chaînes plus longues sont rejetées. Écrire `bit` sans longueur est équivalent à `bit(1)`, alors que `bit varying` sans longueur indique une taille illimitée.

> **Note**
>
> Lors du transtypage explicite (cast) d'une chaîne de bits en champ de type `bit(`*n*`)`, la chaîne obtenue est complétée avec des zéros ou bien tronquée pour obtenir une taille de *n* bits exactement, sans que cela ne produise une erreur. De la même façon, si une chaîne de bits est explicitement transtypée en un champ de type `bit varying(`*n*`)`, elle est tronquée si sa longueur dépasse *n* bits.

Voir la Section 4.1.2.5 pour plus d'information sur la syntaxe des constantes en chaîne de bits. Les opérateurs logiques et les fonctions de manipulation de chaînes sont décrits dans la Section 9.6.

Exemple 8.3. Utiliser les types de chaînes de bits

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');

ERROR:  bit string length 2 does not match type bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');
```

```
SELECT * FROM test;

  a  |  b
-----+-----
 101 | 00
 100 | 101
```

Une valeur pour une chaîne de bits nécessite un octet pour chaque groupe de huit bits, plus cinq ou huit octets d'en-tête suivant la longueur de la chaîne (les valeurs longues peuvent être compressées ou déplacées, comme expliqué dans Section 8.3 pour les chaînes de caractères).

8.11. Types de recherche plein texte

PostgreSQL fournit deux types de données conçus pour supporter la recherche plein texte qui est l'activité de recherche via une collection de *documents* en langage naturel pour situer ceux qui correspondent le mieux à une *requête*. Le type `tsvector` représente un document dans une forme optimisée pour la recherche plein texte alors que le type `tsquery` représente de façon similaire une requête. Chapitre 12 fournit une explication détaillée de cette capacité et Section 9.13 résume les fonctions et opérateurs en relation.

8.11.1. `tsvector`

Une valeur `tsvector` est une liste triée de *lexemes* distincts, qui sont des mots qui ont été *normalisés* pour fusionner différentes variantes du même mot apparaissant (voir Chapitre 12 pour plus de détails). Trier et éliminer les duplicats se font automatiquement lors des entrées, comme indiqué dans cet exemple :

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

Pour représenter des lexèmes contenant des espaces blancs ou des signes de ponctuation, entourez-les avec des guillemets simples :

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

(Nous utilisons les valeurs littérales entre guillemets simples dans cet exemple et dans le prochain pour éviter une confusion en ayant à doubler les guillemets à l'intérieur des valeurs littérales.) Les guillemets imbriqués et les antislashs doivent être doublés :

```
SELECT $$the lexeme 'Joe''s' contains a quote$$::tsvector;
                    tsvector
-----------------------------------------------
 'Joe''s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

En option, les *positions* peuvent être attachées aux lexèmes :

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10
 fat:11 rat:12'::tsvector;
                                  tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5
 'rat':12 'sat':4
```

Une position indique normalement l'emplacement du mot source dans le document. Les informations de position sont utilisables pour avoir un *score de proximité*. Les valeurs des positions peuvent aller de 1 à 16383 ; les grands nombres sont limités silencieusement à 16383. Les positions dupliquées du même lexème sont rejetées.

Les lexèmes qui ont des positions peuvent aussi avoir un label d'un certain *poids*. Les labels possibles sont `A`, `B`, `C` ou `D`. `D` est la valeur par défaut et n'est du coup pas affiché en sortie :

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

Les poids sont typiquement utilisés pour refléter la structure du document en marquant les mots du titre de façon différente des mots du corps. Les fonctions de score de la recherche plein texte peuvent assigner des priorités différentes aux marqueurs de poids différents.

Il est important de comprendre que le type `tsvector` lui-même ne réalise aucune normalisation de mots ; il suppose que les mots qui lui sont fournis sont normalisés correctement pour l'application. Par exemple,

```
SELECT 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
```

Pour la plupart des applications de recherche en anglais, les mots ci-dessus seraient considérés comme non normalisés, mais `tsvector` n'y prête pas attention. Le texte des documents bruts doit habituellement passer via `to_tsvector` pour normaliser les mots de façon appropriée pour la recherche :

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

De nouveau, voir Chapitre 12 pour plus de détails.

8.11.2. `tsquery`

Une valeur `tsquery` enregistre les lexèmes qui doivent être recherchés, et peut les combiner en utilisant les opérateurs booléens `&` (AND), `|` (OR) et `!` (NOT), ainsi que l'opérateur de recherche de phrase `<->` (FOLLOWED BY). Il existe aussi une variante de l'opérateur FOLLOWED BY, `<N>`, où *N* est une constante entière indiquant la distance maximale entre les deux lexèmes recherchés. `<->` est équivalent à `<1>`.

Les parenthèses peuvent être utilisées pour forcer le regroupement des opérateurs. En l'absence de parenthèses, `!` (NOT) est prioritaire, `<->` (FOLLOWED BY) suit, et enfin & (AND) et | (OR) sont les moins prioritaires.

Voici quelques exemples :

```
 SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

En option, les lexèmes dans une `tsquery` peuvent être labelisés avec une lettre de poids ou plus, ce qui les restreint à une correspondance avec les seuls lexèmes `tsvector` pour un de ces poids :

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

Par ailleurs, les lexèmes d'une `tsquery` peuvent être marqués avec `*` pour spécifier une correspondance de préfixe :

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

Cette requête fera ressortir tout mot dans un `tsvector` qui commence par « super ».

Les règles de guillemets pour les lexèmes sont identiques à celles décrites ci-dessus pour les lexèmes de `tsvector` ; et, comme avec `tsvector`, toute normalisation requise des mots doit se faire avant de les placer dans le type `tsquery`. La fonction `to_tsquery` est convenable pour réaliser une telle normalisation :

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

Notez que `to_tsquery` traitera les préfixes de la même façon que les autres mots, ce qui signifie que cette comparaison renvoie true :

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
----------
 t
```

parce que `postgres` devient `postgr` :

```
SELECT to_tsvector( 'postgraduate' ), to_tsquery( 'postgres:*' );
  to_tsvector  | to_tsquery
---------------+------------
 'postgradu':1 | 'postgr':*
```

qui correspondra à la forme native de `postgraduate`.

8.12. Type UUID

Le type de données `uuid` stocke des identifiants universels uniques (UUID, acronyme de *Universally Unique Identifiers*) décrits dans les standards RFC 4122[2], ISO/IEC 9834-8:2005, et d'autres encore. (Certains systèmes font référence à ce type de données en tant qu'identifiant unique global (ou GUID).) Un identifiant de ce type est une quantité sur 128 bits générée par un algorithme adéquat qui a peu de chances d'être reproduit par quelqu'un d'autre utilisant le même algorithme. Du coup, pour les systèmes distribués, ces identifiants fournissent une meilleure garantie d'unicité que ce que pourrait fournir une séquence, dont la valeur est unique seulement au sein d'une base de données.

Un UUID est écrit comme une séquence de chiffres hexadécimaux en minuscule, répartis en différents groupes, séparés par un tiret. Plus précisément, il s'agit d'un groupe de huit chiffres suivis de trois groupes de quatre chiffres terminés par un groupe de douze chiffres, ce qui fait un total de 32 chiffres représentant les 128 bits. Voici un exemple d'UUID dans sa forme standard :

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

PostgreSQL accepte aussi d'autres formes en entrée : utilisation des majuscules, de crochets englobant le nombre, suppression d'une partie ou de tous les tirets, ajout d'un tiret après n'importe quel groupe de quatre chiffres. Voici quelques exemples :

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

L'affichage est toujours dans la forme standard.

Voir Section 9.14 pour savoir comment générer un UUID dans PostgreSQL.

8.13. Type XML

Le type de données `xml` est utilisé pour stocker des données au format XML. Son avantage sur un champ de type `text` est qu'il vérifie que les valeurs sont bien formées. De plus, il existe de nombreuses

[2] https://datatracker.ietf.org/doc/html/rfc4122

fonctions pour réaliser des opérations de vérification à partir de ce type ; voir la Section 9.15. L'utilisation de ce type de données requiert que l'étape de compilation ait utilisé l'option `--with-libxml`.

Le type `xml` peut stocker des « documents » bien formés, suivant la définition du standard XML, ainsi que des fragments de contenu (« content »), en référence au « nœud de document »[3] plus permissif des modèle de données XQuery et XPath. Cela signifie que les fragments de contenu peuvent avoir plus d'un élément racine ou nœud caractère. L'expression *`valeur`*`xml IS DOCUMENT` permet d'évaluer si une valeur `xml` particulière est un document complet ou seulement un fragment de contenu.

Les limites et notes de compatibilité pour le type de données `xml` sont disponibles dans Section D.3.

8.13.1. Créer des valeurs XML

Pour produire une valeur de type `xml` à partir d'une donnée de type caractère, utilisez la fonction `xmlparse` :

```
XMLPARSE ( { DOCUMENT | CONTENT } valeur)
```

Quelques exemples :

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</
title><chapter>...</chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

Bien que cela soit la seule façon de convertir des chaînes de caractères en valeurs XML d'après le standard XML, voici des syntaxes spécifiques à PostgreSQL :

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

Le type `xml` ne valide pas les valeurs en entrée par rapport à une déclaration de type de document (DTD), même quand la valeur en entrée indique une DTD. Il n'existe pas encore de support pour la validation avec d'autres langages de schéma XML, comme XML Schema.

L'opération inverse, produisant une chaîne de caractères à partir d'une valeur au type `xml`, utilise la fonction `xmlserialize`:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

`type` peut être `character`, `character varying` ou `text` (ou un alias de ces derniers). Encore une fois, d'après le standard SQL, c'est le seul moyen de convertir le type `xml` vers les types caractère, mais PostgreSQL autorise aussi la conversion simple de la valeur.

Lorsque les valeurs des chaînes de caractères sont converties vers ou à partir du type `xml` sans passer par `XMLPARSE` ou `XMLSERIALIZE`, respectivement, le choix de `DOCUMENT` ou de `CONTENT` est déterminé par un paramètre de configuration niveau session, « XML OPTION » , qui peut être configuré par la commande habituelle :

```
SET XML OPTION { DOCUMENT | CONTENT };
```

[3] https://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/#DocumentNode

ou la syntaxe PostgreSQL :

```
SET xmloption TO { DOCUMENT | CONTENT };
```

La valeur par défaut est `CONTENT`, donc toutes les formes de données XML sont autorisées.

8.13.2. Gestion de l'encodage

Une grande attention doit prévaloir lors de la gestion de plusieurs encodages sur le client, le serveur ou dans les données XML qui passent entre eux. Lors de l'utilisation du mode texte pour passer les requêtes au serveur et pour renvoyer les résultats au client (qui se trouve dans le mode normal), PostgreSQL convertit toutes les données de type caractère passées entre le client et le serveur et vice-versa suivant l'encodage spécifique de la destination finale ; voir la Section 24.3. Cela inclut les représentations textuelles des valeurs XML, comme dans les exemples ci-dessus, ce qui signifie que les déclarations d'encodage contenues dans les données XML pourraient devenir invalides lorsque les données sont converties vers un autre encodage lors du transfert entre le client et le serveur, alors que la déclaration de l'encodage n'est pas modifiée. Pour s'en sortir, une déclaration d'encodage contenue dans une chaîne de caractères présentée en entrée du type `xml` est *ignorée*, et le contenu est toujours supposé être de l'encodage du serveur. En conséquence, pour un traitement correct, ces chaînes de caractères de données XML doivent être envoyées du client dans le bon encodage. C'est de la responsabilité du client de soit convertir le document avec le bon encodage client avant de l'envoyer au serveur, soit d'ajuster l'encodage client de façon appropriée. En sortie, les valeurs du type `xml` n'auront pas une déclaration d'encodage et les clients devront supposer que les données sont dans l'encodage du client.

Lors de l'utilisation du mode binaire pour le passage des paramètres de la requête au serveur et des résultats au client, aucune conversion de l'encodage n'est réalisée, donc la situation est différente. Dans ce cas, une déclaration d'encodage dans les données XML sera observée et, si elle est absente, les données seront supposées être en UTF-8 (comme requis par le standard XML ; notez que PostgreSQL ne supporte pas du tout UTF-16). En sortie, les données auront une déclaration d'encodage spécifiant l'encodage client, sauf si l'encodage client est UTF-8, auquel cas elle sera omise.

Le traitement des données XML avec PostgreSQL sera moins complexe et plus efficace si l'encodage des données, l'encodage client et l'encodage serveur sont identiques. Comme les données XML sont traitées en interne en UTF-8, les traitements seront plus efficaces si l'encodage serveur est aussi en UTF-8.

> **Attention**
>
> Certaines fonctions relatives à XML pourraient ne pas fonctionner du tout sur des données non ASCII quand l'encodage du serveur n'est pas UTF-8. C'est un problème connu pour `xmltable()` et `xpath()` en particulier.

8.13.3. Accéder aux valeurs XML

Le type de données `xml` est inhabituel dans le sens où il ne dispose pas d'opérateurs de comparaison. Ceci est dû au fait qu'il n'existe pas d'algorithme de comparaison bien défini et utile pour des données XML. Une conséquence de ceci est que vous ne pouvez pas récupérer des lignes en comparant une colonne `xml` avec une valeur de recherche. Les valeurs XML doivent du coup être typiquement accompagnées par un champ clé séparé comme un identifiant. Une autre solution pour la comparaison de valeurs XML est de les convertir en des chaînes de caractères, mais notez que la comparaison de chaînes n'a que peu à voir avec une méthode de comparaison XML utile.

Comme il n'y a pas d'opérateurs de comparaison pour le type de données `xml`, il n'est pas possible de créer un index directement sur une colonne de ce type. Si une recherche rapide est souhaitée dans des données XML, il est toujours possible de convertir l'expression en une chaîne de caractères et d'indexer cette conversion. Il est aussi possible d'indexer une expression XPath. La vraie requête devra bien sûr être ajustée à une recherche sur l'expression indexée.

La fonctionnalité de recherche plein texte peut aussi être utilisée pour accélérer les recherches dans des données XML. Le support du prétraitement nécessaire n'est cependant pas disponible dans la distribution PostgreSQL.

8.14. Types JSON

Les types de données JSON sont faits pour stocker des données JSON (JavaScript Object Notation), comme spécifié dans la RFC 7159[4]. De telles données peuvent également être stockées comme `text` , mais les types de données JSON ont l'avantage d'assurer que chaque valeur stockée est valide d'après les règles JSON. Il y a également des fonctions et opérateurs spécifiques à JSON associés disponibles pour les données stockées dans ces types de données. Voir Section 9.16.

PostgreSQL propose deux types de données pour stocker des données JSON : `json` et `jsonb`. Pour implémenter des mécanismes de requêtage efficaces pour ces types de données, PostgreSQL propose aussi le type `jsonpath` décrit dans Section 8.14.7.

Les types de données `json` et `jsonb` acceptent *quasiment* des ensembles de valeurs identiques en entrée. La différence majeure réside dans l'efficacité. Le type de données `json` stocke une copie exacte du texte en entrée, que chaque fonction doit analyser à chaque exécution, alors que le type de données `jsonb` est stocké dans un format binaire décomposé qui rend l'insertion légèrement plus lente du fait du surcoût de la conversion, mais est significativement plus rapide pour traiter les données, puisqu'aucune analyse n'est nécessaire. `jsonb` gère également l'indexation, ce qui peut être un avantage significatif.

Puisque le type `json` stocke une copie exacte du texte en entrée, il conservera les espaces sémantiquement non significatifs entre les jetons, ainsi que l'ordre des clés au sein de l'objet JSON. De plus, si un objet JSON contient dans sa valeur la même clé plus d'une fois, toutes les paires clé/valeur sont conservées (les fonctions de traitement considèrent la dernière valeur comme celle significative). À l'inverse, `jsonb` ne conserve ni les espaces non significatifs, ni l'ordre des clés d'objet, ni ne conserve les clés d'objet dupliquées. Si des clés dupliquées sont présentées en entrée, seule la dernière valeur est conservée.

En général, la plupart des applications devraient préférer stocker les données JSON avec `jsonb`, à moins qu'il y ait des besoins spécifiques, comme la supposition légitime de l'ordre des clés d'objet.

La RFC 7159 spécifie que les chaînes JSON devraient être encodées en UTF8. Il n'est donc pas possible pour les types JSON de se conformer de manière rigoureuse à la spécification JSON, à moins que l'encodage de la base de données soit UTF8. Tenter d'inclure directement des caractères qui ne peuvent pas être représentés dans l'encodage de la base de données échouera ; inversement, des caractères qui peuvent être représentés dans l'encodage de la base de données, mais pas en UTF8, seront autorisés.

La RFC 7159 autorise les chaînes JSON à contenir des séquences Unicode échappées, indiquées avec `\u`*`XXXX`*. Dans la fonction d'entrée pour le type `json`, les échappements Unicode sont autorisés quel que soit l'encodage de la base de données, et sont vérifiés uniquement pour l'exactitude de la syntaxe (qui est quatre chiffres hexadécimaux précédés d'un `\u`). Toutefois, la fonction d'entrée pour `jsonb` est plus stricte : elle interdit les échappements Unicode qui ne peuvent pas être représentés dans l'encodage de la base. Le type `jsonb` rejette aussi `\u0000` (parce qu'il ne peut pas être représenté avec le type `text` de PostgreSQL), et il insiste pour que chaque utilisation de paires de substitution Unicode désignant des caractères en dehors du *Unicode Basic Multilingual Plane* soit correcte. Les échappements Unicode valides sont convertis en leur caractère ASCII ou UTF8 équivalent pour du stockage ; ceci inclut les « folding surrogate pairs » sur un seul caractère.

[4] https://datatracker.ietf.org/doc/html/rfc7159

> **Note**
>
> De nombreuses fonctions de traitement JSON décrites dans Section 9.16 convertiront les échappements Unicode vers des caractères standards, et généreront donc le même type d'erreurs décrit juste avant si leur entrée est de type `json` et non `jsonb`. Le fait que la fonction d'entrée `json` ne fasse pas ces vérifications peut être considéré comme un artefact historique, bien qu'elle n'autorise pas un simple stockage (sans traitement) d'échappements Unicode JSON dans un encodage de base qui ne supporte pas les caractères représentés.

Lors de la conversion de données texte JSON vers `jsonb`, les types primitifs décrits par la RFC 7159 sont transcrits efficacement vers des types PostgreSQL natifs, comme indiqué dans Tableau 8.23. Par conséquent, il y a quelques contraintes additionnelles mineures sur ce qui constitue des données `jsonb` valides qui ne s'appliquent ni au type `json`, ni à JSON en définitive, correspondant aux limites de ce qui peut être représenté par le type de données sous-jacent. Spécifiquement, `jsonb` rejettera les nombres qui sont en dehors de la portée du type de données `numeric` de PostgreSQL, alors que `json` les acceptera. De telles restrictions définies par l'implémentation sont permises par la RFC 7159. Cependant, en pratique, de tels problèmes ont beaucoup plus de chances de se produire dans d'autres implémentations, puisqu'il est habituel de représenter les types primitifs `number` JSON comme des nombres flottants à double précision (IEEE 754 double precision floating point), ce que la RFC 7159 anticipe explicitement et autorise. Lorsque JSON est utilisé comme format d'échange avec de tels systèmes, le risque de perte de précision pour les valeurs numériques comparées aux données stockées à l'origine par PostgreSQL devrait être considéré.

À l'inverse, comme indiqué dans le tableau, il y a quelques restrictions mineures sur le format d'entrée de types primitifs JSON qui ne s'appliquent pas aux types PostgreSQL correspondants.

Tableau 8.23. Types primitifs JSON et types PostgreSQL correspondants

Type primitif JSON	Type PostgreSQL	Notes
`string`	`text`	`\u0000` est interdit, comme le sont les échappements Unicode représentant des caractères non disponibles dans l'encodage de la base
`number`	`numeric`	Les valeurs `NaN` et `infinity` sont interdites
`boolean`	`boolean`	Seules les versions en minuscule de `true` et `false` sont acceptées
`null`	(none)	`NULL` dans SQL est un concept différent

8.14.1. Syntaxe d'entrée et de sortie JSON

La syntaxe d'entrée/sortie pour les types de données JSON est identique à celle spécifiée dans la RFC 7159.

Les exemples suivants sont tous des expressions `json` (ou `jsonb`) valides :

```
-- Simple valeur scalaire/primitive
-- Les valeurs primitives peuvent être des nombres, chaînes entre
 guillemets, true, false ou null
SELECT '5'::json;

-- Tableau de zéro ou plus éléments (les éléments doivent être du
 même type)
SELECT '[1, 2, "foo", null]'::json;
```

```
-- Objets contenant des paires de clé et valeurs
-- À noter que les clés d'objets doivent toujours être des chaînes
 entre guillemets
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- Tableaux et objets peuvent être imbriqués arbitrairement
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

Comme dit précédemment, quand une valeur JSON est renseignée puis affichée sans traitement additionnel, `json` renvoie le même texte qui était fourni en entrée, alors que `jsonb` ne préserve pas les détails sémantiquement non significatifs comme les espaces. Par exemple, il faut noter la différence ici :

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
                      json
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
                      jsonb
-------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

un détail sémantiquement non significatif qu'il faut souligner est qu'avec `jsonb`, les nombres seront affichés en fonction du type `numeric` sous-jacent. En pratique, cela signifie que les nombres renseignés avec la notation `E` seront affichés sans. Par exemple :

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading":
 1.230e-5}'::jsonb;
         json          |          jsonb
-----------------------+-------------------------
 {"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

Toutefois, `jsonb` préservera les zéros en fin de partie fractionnaire, comme on peut le voir dans cet exemple, même si ceux-ci ne sont pas sémantiquement significatifs, pour des besoins tels que des tests d'égalité.

Pour la liste de fonctions et opérateurs intégrés disponibles pour construire et traiter des valeurs JSON, voir Section 9.16.

8.14.2. Concevoir des documents JSON efficacement

Représenter des données en JSON peut être considérablement plus flexible que le modèle de données relationnel traditionnel, qui est contraignant dans des environnements où les exigences sont souples. Il est tout à fait possible que ces deux approches puissent coexister, et qu'elles soient complémentaires au sein de la même application. Toutefois, même pour les applications où on désire le maximum de flexibilité, il est toujours recommandé que les documents JSON aient une structure quelque peu fixée. La structure est typiquement non vérifiée (bien que vérifier des règles métier de manière déclarative soit possible), mais le fait d'avoir une structure prévisible rend plus facile l'écriture de requêtes qui résument utilement un ensemble de « documents » (datums) dans une table.

Les données JSON sont sujettes aux mêmes considérations de contrôle de concurrence que pour n'importe quel autre type de données quand elles sont stockées en table. Même si stocker de gros documents est prévisible, il faut garder à l'esprit que chaque mise à jour acquiert un verrou de niveau ligne sur toute la ligne. Il faut envisager de limiter les documents JSON à une taille gérable pour réduire les contentions sur verrou lors des transactions en mise à jour. Idéalement, les documents JSON devraient chacun représenter une donnée atomique, que les règles métiers imposent de ne pas pouvoir subdiviser en données plus petites qui pourraient être modifiées séparément.

8.14.3. Existence et inclusion `jsonb`

Tester l'*inclusion* est une capacité importante de `jsonb`. Il n'y a pas d'ensemble de fonctionnalités parallèles pour le type `json`. L'inclusion teste si un des documents `jsonb` est contenu dans un autre. Ces exemples renvoient vrai, sauf note explicite :

```
-- Simple valeur scalaire/primitive qui contient une seule valeur
 identique :
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

-- Le tableau de droite est contenu dans celui de gauche :
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- L'ordre des éléments d'un tableau n'est pas significatif, donc
 ceci est tout
-- aussi vrai :
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- Les éléments dupliqués d'un tableau n'ont pas plus
 d'importance :
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- L'objet avec une seule paire à droite est contenu
-- dans l'objet sur le côté gauche :
SELECT '{"product": "PostgreSQL", "version": 9.4,
 "jsonb":true}'::jsonb @> '{"version":9.4}'::jsonb;

-- Le tableau du côté droit n'est <emphasis>pas</emphasis>
 considéré comme contenu
-- dans le tableau du côté gauche, même si un tableau similaire est
 imbriqué dedans :
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  -- renvoie faux

-- Mais avec une couche d'imbrication, il est contenu :
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

-- De la même manière, l'inclusion n'est pas valable ici :
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;
  -- renvoie faux

-- Une clé du niveau racine et un objet vide sont contenus :
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

Le principe général est que l'objet inclus doit correspondre à l'objet devant le contenir à la fois pour la structure et pour les données, peut-être après la suppression d'éléments de tableau ou d'objets paires clé/valeur ne correspondant pas à l'objet contenant. Mais rappelez-vous que l'ordre des éléments dans un tableau n'est pas significatif lors d'une recherche de contenance, et que les éléments dupliqués d'un tableau ne sont réellement considérés qu'une seule fois.

Comme exception qui confirme la règle que les structures doivent correspondre, un tableau peut inclure une valeur primitive :

```
-- Ce tableau inclut la valeur primitive chaîne :
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- Cette exception n'est pas réciproque, la non-inclusion est
 rapportée ici :
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- renvoie faux
```

`jsonb` a également un opérateur d'*existence*, qui est une variation sur le thème de l'inclusion : il teste si une chaîne (sous forme de valeur `text`) apparaît comme une clé d'objet ou un élément de tableau au niveau supérieur de la valeur `jsonb`. Ces exemples renvoient vrai; sauf note explicite :

```
-- La chaîne existe comme un élément de tableau :
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- La chaîne existe comme une clé d'objet :
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- Les valeurs d'objets ne sont pas examinées :
SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- renvoie faux

-- Comme pour l'inclusion, l'existence doit correspondre au niveau
 supérieur :
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- renvoie faux

-- Une chaîne est examinée pour l'existence si elle correspond à
 une primitive chaîne JSON :
SELECT '"foo"'::jsonb ? 'foo';
```

Les objets JSON sont plus adaptés que les tableaux pour tester l'inclusion ou l'existence quand il y a de nombreux éléments ou clés impliqués, car contrairement aux tableaux, ils sont optimisés de manière interne pour la recherche et n'ont pas besoin d'être parcourus linéairement.

Astuce

Comme les documents JSON sont imbriqués, une requête appropriée peut ignorer une sélection explicite de sous-objets. Par exemple, supposons que nous ayons une colonne `doc` contenant des objets au plus haut niveau, avec la plupart des objets contenant les champs `tags` qui contiennent eux-mêmes des tableaux de sous-objets. Cette requête trouve des entrées dans lesquelles les sous-objets contiennent à la fois `"term":"paris"` et `"term":"food"`, tout en ignorant ces clés en dehors du tableau `tags` :

```
SELECT doc->'site_name' FROM websites
  WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';
```

Cela pourrait s'accomplir aussi ainsi :

```
SELECT doc->'site_name' FROM websites
  WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';
```

mais cette approche est moins flexible, et souvent bien moins efficace.

Mais l'opérateur JSON d'existence n'est pas imbriqué : il cherchera seulement pour la clé ou l'élément de tableau spécifié à la racine de la valeur JSON.

Les différents opérateurs d'inclusion d'existence, avec tous les autres opérateurs et fonctions JSON, sont documentés dans Section 9.16.

8.14.4. Indexation `jsonb`

Les index GIN peuvent être utilisés pour chercher efficacement des clés ou des paires clé/valeur se trouvant parmi un grand nombre de documents (datums) `jsonb`. Deux « classes d'opérateurs » GIN sont fournies, offrant différents compromis entre performances et flexibilité.

La classe d'opérateurs GIN par défaut pour `jsonb` supporte les requêtes avec des opérateurs clé-existe `?`, `?|` et `?&` et l'opérateur de contenance `@>` et les opérations de correspondance `jsonpath` `@?` et `@@`. (Pour des détails sur la sémantique que ces opérateurs implémentent, voir Tableau 9.46.) Un exemple de création d'index avec cette classe d'opérateurs est :

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

La classe d'opérateurs GIN qui n'est pas par défaut `jsonb_path_ops` n'accepte pas les opérateurs clé-existe, mais accepte `@>`, `@?` et `@@`. seulement. Un exemple de création d'index avec cette classe d'opérateurs est :

```
CREATE INDEX idxginp ON api USING GIN (jdoc jsonb_path_ops);
```

En étudiant l'exemple d'une table qui stocke des documents JSON récupérés par un service web tiers, avec une définition de schéma documentée, un document typique serait :

```
{
    "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
    "name": "Angela Barton",
    "is_active": true,
    "company": "Magnafone",
    "address": "178 Howard Place, Gulf, Washington, 702",
    "registered": "2009-11-07T08:53:22 +08:00",
    "latitude": 19.793713,
    "longitude": 86.513373,
    "tags": [
        "enim",
        "aliquip",
        "qui"
    ]
}
```

Ces documents sont stockés dans une table nommée `api`, dans une colonne de type `jsonb` nommée `jdoc`. Si un index GIN est créé sur cette colonne, des requêtes semblables à l'exemple suivant peuvent utiliser cet index :

```
-- Trouver les documents dans lesquels la clé "company" a pour
 valeur "Magnafone"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @>
 '{"company": "Magnafone"}';
```

Toutefois, cet index ne pourrait pas être utilisé pour des requêtes comme dans l'exemple suivant, car bien que l'opérateur `?` soit indexable, il n'est pas appliqué directement sur la colonne indexée `jdoc` :

```
-- Trouver les documents dans lesquels la clé "tags" contient une
 clé ou un élément tableau "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ?
 'qui';
```

Toutefois, avec l'utilisation appropriée d'index sur expression, la requête ci-dessus peut utiliser un index. Si le requêtage d'éléments particuliers de la clé `"tags"` est fréquent, définir un index comme ceci pourrait être particulièrement bénéfique :

```
-- À noter que l'opérateur "jsonb -> text" ne peut être appelé que
 sur un
-- objet JSON, donc la conséquence de créer cet index est que le
 premier niveau de
-- chaque valeur "jdoc" doit être un objet. Ceci est vérifié lors
 de chaque insertion.
CREATE INDEX idxgintags ON api USING GIN ((jdoc -> 'tags'));
```

Dorénavant, la clause `WHERE jdoc -> 'tags' ? 'qui'` sera reconnue comme une application de l'opérateur indexable `?` pour l'expression indexée `jdoc -> 'tags'`. (Plus d'informations sur les index sur expression peuvent être trouvées dans Section 11.7.)

Une autre approche pour le requêtage et l'exploitation de l'inclusion, par exemple :

```
-- Trouver les documents dans lesquels la clé "tags" inclut
 l'élément tableau "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags":
 ["qui"]}';
```

Un simple index GIN sur la colonne `jdoc` peut répondre à cette requête. Mais il faut noter qu'un tel index stockera des copies de chaque clé et chaque valeur de la colonne `jdoc`, alors que l'index sur expression de l'exemple précédent ne stockera que les données trouvées pour la clé `tags`. Alors que l'approche d'index simple est bien plus souple (puisqu'elle supporte les requêtes sur n'importe quelle clé), les index sur des expressions ciblées ont bien plus de chances d'être plus petits et plus rapides pour la recherche qu'un simple index.

Les index GIN supportent aussi les opérateurs `@?` et `@@` qui réalisent la correspondance `jsonpath`. Voici des exemples :

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @@ '$.tags[*]
 == "qui"';
```

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @?
 '$.tags[*] ? (@ == "qui")';
```

Pour ces opérateurs, l'index GIN extrait les clauses de la forme suivante du motif `jsonpath` : *`chaîne des accesseurs`* = *`constante`* et effectue la recherche d'index en se basant sur les clés et valeurs mentionnées dans ces clauses. La chaîne des accesseurs peut être les accesseurs `.key`, `[*]`, et `[`*`index`*`]`. La classe d'opérateurs `jsonb_ops` accepte aussi les accesseurs `.*` et `.**`, contrairement à la classe d'opérateur `jsonb_path_ops`.

Bien que la classe d'opérateurs `jsonb_path_ops` ne supporte que les requêtes avec les opérateurs `@>`, `@?` et `@@`, elle a des avantages de performances notables par rapport à la classe d'opérateurs par défaut `jsonb_ops`. Un index `jsonb_path_ops` est généralement bien plus petit qu'un index `jsonb_ops` pour les mêmes données, et la spécificité de la recherche est meilleure, particulièrement quand les requêtes contiennent des clés qui apparaissent fréquemment dans les données. Par conséquent, les opérations de recherche sont généralement plus performantes qu'avec la classe d'opérateurs par défaut.

La différence technique entre des index GIN `jsonb_ops` et `jsonb_path_ops` est que le premier crée des éléments d'index indépendants pour chaque clé et valeur dans les données, alors que le second crée des éléments d'index uniquement pour chaque valeur dans les données. [5] Fondamentalement, chaque élément d'index `jsonb_path_ops` est un hachage de la valeur et de la ou des clés y menant ; par exemple pour indexer `{"foo": {"bar": "baz"}}`, un seul élément dans l'index sera créé, incorporant les trois `foo`, `bar` et `baz` dans une valeur hachée. Ainsi, une requête d'inclusion cherchant cette structure résulterait en une recherche d'index extrêmement spécifique, mais il n'y a pas d'autre moyen de savoir si `foo` apparaît en tant que clé. D'un autre côté, un index `jsonb_ops` créerait trois éléments d'index représentant `foo`, `bar` et `baz` séparément ; ainsi, pour faire la requête d'inclusion, il faudrait rechercher les lignes contenant chacun des trois éléments. Bien que les index GIN puissent effectuer de telles recherches et de manière tout à fait efficace, cela sera toujours moins spécifique et plus lent que la recherche équivalente `jsonb_path_ops`, surtout s'il y a un très grand nombre de lignes contenant n'importe lequel des trois éléments d'index.

Un désavantage de l'approche `jsonb_path_ops` est qu'elle ne produit d'entrées d'index que pour les structures JSON ne contenant aucune valeur, comme `{"a": {}}`. Si une recherche pour des documents contenant une telle structure est demandée, elle nécessitera un parcours de la totalité de l'index, ce qui peut être assez long. `jsonb_path_ops` est donc mal adapté pour des applications qui effectuent souvent de telles recherches.

`jsonb` supporte également les index `btree` et `hash`. Ceux-ci ne sont généralement utiles que s'il est important de vérifier l'égalité de documents JSON entiers. Le tri `btree` pour des données `jsonb` est rarement d'un grand intérêt, mais afin d'être exhaustif, il est :

`Objet > Tableau > Booléen > Nombre > Chaîne > null`

`Objet avec n paires > objet avec n - 1 paires`

`Tableau avec n éléments > tableau avec n - 1 éléments`

avec l'exception que, pour des raisons historiques, un tableau vide au niveau est trié plus bas qu'un *`null`*. Les objets avec le même nombre de paires sont comparés dans cet ordre :

`clé-1, valeur-1, clé-2 ...`

[5] Dans ce contexte, le terme « valeur » inclut les éléments de tableau, bien que la terminologie JSON considère parfois que les éléments de tableaux soient distincts des valeurs dans les objets.

À noter que les clés d'objet sont comparées dans leur ordre de stockage ; en particulier, puisque les clés les plus courtes sont stockées avant les clés les plus longues, cela peut amener à des résultats contre-intuitifs, tels que :

```
{ "aa": 1, "c": 1} > {"b": 1, "d": 1}
```

De la même manière, les tableaux avec le même nombre d'éléments sont comparés dans l'ordre :

```
élément-1, élément-2 ...
```

Les valeurs JSON primitives sont comparées en utilisant les mêmes règles de comparaison que pour les types de données PostgreSQL sous-jacents. Les chaînes sont comparées en utilisant la collation par défaut de la base de données.

8.14.5. Utiliser l'indiçage avec `jsonb`

Le type de données `jsonb` accepte les expressions avec indice, du style tableau, pour extraire et modifier les éléments. Les valeurs imbriquées peuvent être indiquées en chaînant les expressions à indice, en suivant les mêmes règles que l'argument `path` dans la fonction `jsonb_set`. Si une valeur `jsonb` est un tableau, les indices numériques commencent à zéro, et les nombres négatifs comptent à partir du dernier élément du tableau. Les expressions d'intervalle ne sont pas acceptées. Le résultat d'une expression par indice est toujours du type de données jsonb.

Les requêtes `UPDATE` peuvent utiliser l'indiçage dans la clause `SET` pour modifier les valeurs `jsonb`. Les chemins d'indice doivent être parcourables pour toutes les valeurs affectées. Par exemple, le chemin `val['a'] ['b']['c']` peut être entièrement traversé jusqu'à `c` si `val`, `val ['a']` et `val['a']['b']` sont des objets. Si `val['a']` ou `val['a']['b']` ne sont pas définis, il sera créé comme un objet vide et rempli comme nécessaire. Néanmoins, si `val` lui-même ou une des valeurs intermédiaires est défini comme un non-objet (une chaîne, un nombre ou le `null` `jsonb`), la traversée ne peut pas continuer, donc une erreur est levée et la transaction est annulée.

Un exemple de syntaxe avec indiçage :

```
-- Extraire la valeur d'un objet par sa clé
SELECT ('{"a": 1}'::jsonb)['a'];

-- Extraire une valeur d'objet imbriqué par le chemin de la clé
SELECT ('{"a": {"b": {"c": 1}}}'::jsonb)['a']['b']['c'];

-- Extraire l'élément du tableau par indice
SELECT ('[1, "2", null]'::jsonb)[1];

-- Mise à jour de la valeur de l'objet par sa clé.
-- Notez les guillemets simples autour de '1' : la valeur affectée
-- doit aussi être du type jsonb
UPDATE table_name SET jsonb_field['key'] = '1';

-- Ceci lèvera une erreur si un enregistrement de jsonb_field['a']
['b'] se révèle
-- être autre chose qu'un objet. Par exemple, la valeur {"a": 1} a
 une valeur numérique
-- de la clé 'a'.
```

```
UPDATE table_name SET jsonb_field['a']['b']['c'] = '1';

-- Filtrer lesenregistrements en utilisant une clause WHERE avec
 des indices.
-- Comme le résultat de l'indiçage est jsonb, la valeur à laquelle
 nous la comparons
-- doit aussi être jsonb.
-- Les guillemets doubles rendent valide "value" comme chaîne
 jsonb.
SELECT * FROM table_name WHERE jsonb_field['key'] = '"value"';
```

L'affectation de `jsonb` via l'indiçage gère quelques cas particuliers différemment de `jsonb_set`. Quand une valeur source `jsonb` vaut `NULL`, l'affectation via l'indiçage continuera comme s'il s'agissait d'une valeur JSON vide (objet ou tableau) impliquée par la clé d'indiçage :

```
-- Là où jsonb_field valait NULL, il vaut maintenant {"a": 1}
UPDATE table_name SET jsonb_field['a'] = '1';

-- Là où jsonb_field valait NULL, il vaut maintenant [1]
UPDATE table_name SET jsonb_field[0] = '1';
```

Si un index est indiqué pour un tableau contenant trop peu d'éléments, les éléments `NULL` seront ajoutés jusqu'à ce que l'index soit atteignable et que la valeur puisse être initialisée.

```
-- Là où jsonb_field valait [], il vaut maintenant [null, null, 2];
-- Là où jsonb_field valait [0], il vaut maintenant [0, null, 2]
UPDATE table_name SET jsonb_field[2] = '2';
```

Une valeur de type `jsonb` acceptera des affectations sur des chemins indicés non existants car le dernier élément existant à être traversé est un objet ou un tableau, comme impliqué par l'indice correspondant (l'élément indiqué par le dernier indice dans le chemin n'est pas traversé et pourrait être n'improte quoi). Le tableau imbriqué et les structures d'objet seront créés et, dans le premier cas, aligné avec des `null`, comme spécifié par le chemin d'indice jusqu'à ce que la valeur affectée puisse être placée.

```
-- Là où jsonb_field valait {}, il vaut maintenant {"a": [{"b":
 1}]}
UPDATE table_name SET jsonb_field['a'][0]['b'] = '1';

-- Là où jsonb_field valait [], il vaut maintenant [null, {"a": 1}]
UPDATE table_name SET jsonb_field[1]['a'] = '1';
```

8.14.6. Transformations

Des extensions supplémentaires sont disponibles pour implémenter des transformations pour le type `jsonb` pour différents langages de procédure stockée.

Les extensions pour PL/Perl sont appelées `jsonb_plperl` et `jsonb_plperlu`. Si vous les utilisez, les valeurs `jsonb` sont transformées en tableaux, hachages et scalaires Perl, suivant le cas.

L'extension pour PL/Python est appelée `jsonb_plpython3u`. Si vous l'utilisez, les valeurs `jsonb` sont transformées en dictionnaires, listes et scalaires Python, suivant le cas.

Pour ces extensions, `jsonb_plperl` est considéré de confiance (« trusted »), autrement dit il peut être installé par des utilisateurs qui n'ont pas l'attribut `SUPERUSER` mais qui ont le droit `CREATE` sur la base actuelle. Le reste nécessite d'être superutilisateur pour être installé.

8.14.7. Type jsonpath

Le type `jsonpath` implémente le support du langage SQL/JSONpath dans PostgreSQL de façon à requêter correctement dans les données JSON. Cela fournit une représentation binaire de l'expression SQL/JSONpath analysée qui spécifie les valeurs qui doivent être récupérées par le moteur de chemin depuis les données JSON pour être traités par la suite avec les fonctions de requêtage SQL/JSON.

La sémantique des prédicats et opérateurs suit généralement celle du SQL. Dans le même temps, il propose une manière naturelle de travailler avec les données JSON. En effet, la syntaxe SQL/JSONpath utilise certaines conventions JavaScript :

- Le point (`.`) est utilisé pour accéder à une valeur.
- Les crochets (`[]`) sont utilisés pour accéder à un tableau.
- Les tableaux SQL/JSON commencent à partir de 0, contrairement aux tableaux SQL classiques qui commencent à 1.

Une expression de chemin SQL/JSON est typiquement écrite dans une requête SQL comme une constante chaîne de caractères SQL, donc elle doit être entourée par des guillemets simples, et tout guillemet simple souhaité dans la valeur doit être doublé (voir Section 4.1.2.1). Certaines formes d'expressions de chemin requièrent des constantes de chaîne à l'intérieur. Ces constantes de chaînes intégrées suivent les conventions JavaScript/ECMAScript : elles doivent être entourées par des guillemets doubles et les échappements doivent être utilisées pour représenter les caractères difficiles à saisir. En particulier, la façon d'écrire un guillemet double dans une constante de type chaîne est `\"`, et pour écrire un antislash, vous devez écrire `\\`. Les autres séquences spéciales d'échappement incluent celles reconnues dans les chaînes JavaScript : `\b`, `\f`, `\n`, `\r`, `\t`, `\v` pour les différents caractères de contrôle ASCII, `\x`*`NN`* pour un code caractère écrit avec deux chiffres hexadécimaux, `\u`*`NNNN`* pour un point code caractère Unicode écrit avec 1 à 6 chiffres hexadécimaux.

Une expression path consiste en une séquence d'éléments path, qui peuvent être une des suivants :

- Path litéral de type primitifs JSON : texte Unicode, numeric, true, false, ou null.
- Variables Path listées dans Tableau 8.24.
- Opérateurs d'accès listés dans Tableau 8.25.
- Opérateurs et méthodes `jsonpath` listés dans Section 9.16.2.2.
- Les parenthèses, qui peuvent être utilisées pour filtrer des expressions ou définir l'ordre d'évaluation des Path.

Pour plus de détails concernant l'utilisation des expressions `jsonpath` avec des fonctions de requête SQL/JSON, voir Section 9.16.2.

Tableau 8.24. Variables `jsonpath`

Variable	Description
`$`	Une variable qui représente le texte JSON en cours de traitement (l'*élément de contexte*).
`$varname`	Une variable nommée. Le paramètre *`vars`* présent dans de nombreuses fonctions de traitement JSON, permet de définir sa valeur. Voir Tableau 9.48 pour plus de détails.
`@`	Une variable qui représente le résultat de l'évaluation du path dans un filtre.

Tableau 8.25. Accesseurs `jsonpath`

Opérateurs d'accès	Description
.clé ."$*nom de variable*"	La valeur accédée renvoie un objet avec la clé spécifiée. Si le nom de la clé correspond à certaines variables nommées commençant avec $ ou qui n'est pas compatible avec les règles JavaScript d'un identifiant, ce nom doit être entouré avec des guillemets double comme une chaîne de caractères.
.*	L'accès joker renvoie les valeurs de tous les membres se trouvant au plus haut niveau de l'objet courant.
.**	L'accès joker récursif traite tous les niveaux de l'arborescence JSON de l'objet courant, et retourne toutes les valeurs, peu importe leur niveau d'imbrication. Ceci est une extension PostgreSQL du standard SQL/JSON
.**{*level*} .**{*start_level* au *end_level*}	Identique à .**, mais sélectionne seulement les niveaux indiqués de la hiérarchie JSON. Les niveaux d'imbrication sont spécifiés en entier. Le niveau zéro correspond à l'objet courant. Pour accèder au dernier niveau d'imbrication, vous pouvez utiliser le mot clé `last`. Ceci est une extension PostgreSQL du standard SQL/JSON.
[*subscript*, ...]	Accès aux valeurs d'un tableau. *subscript* peut être donné sous deux formes : *index* ou *start_index* à *end_index*. La première forme renvoie une simple valeur du tableau par son index. La seconde forme renvoie un morceau de tableau provenant d'un intervalle d'index, en incluant les valeurs qui correspondent à celles fournies dans *start_index* et *end_index*. L'*index* spécifié peut être un entier, aussi bien qu'une expression renvoyant une simple valeur numérique, qui sera automatiquement transtypé vers un entier. L'index zéro correspond à la première valeur d'un tableau. Vous pouvez aussi utiliser le mot clé `last` pour indiquer le dernier élément, ce qui est utile pour manipuler des tableaux de longueur inconnue.
[*]	L'accès joker tableau qui renvoie toutes les valeurs du tableau.

8.15. Tableaux

PostgreSQL permet de définir des colonnes de table comme des tableaux multidimensionnels de longueur variable. Il est possible de créer des tableaux de n'importe quel type utilisateur : de base, énuméré, composé, intervalle, domaine.

8.15.1. Déclaration des types tableaux

La création de la table suivante permet d'illustrer l'utilisation des types tableaux :

```
CREATE TABLE sal_emp (
    nom              text,
    paye_par_semaine integer[],
    planning         text[][]
);
```

Comme indiqué ci-dessus, un type de données tableau est nommé en ajoutant des crochets (`[]`) au type de données des éléments du tableau. La commande ci-dessus crée une table nommée `sal_emp` avec une colonne de type `text` (`nom`), un tableau à une dimension de type `integer`

(paye_par_semaine), représentant le salaire d'un employé par semaine et un tableau à deux dimensions de type text (planning), représentant le planning hebdomadaire de l'employé.

La syntaxe de CREATE TABLE permet de préciser la taille exacte des tableaux, par exemple :

```
CREATE TABLE tictactoe (
    carres   integer[3][3]
);
```

Néanmoins, l'implantation actuelle ignore toute limite fournie pour la taille du tableau, c'est-à-dire que le comportement est identique à celui des tableaux dont la longueur n'est pas précisée.

De plus, l'implantation actuelle n'oblige pas non plus à déclarer le nombre de dimensions. Les tableaux d'un type d'élément particulier sont tous considérés comme étant du même type, quelles que soient leur taille et leur nombre de dimensions. Déclarer la taille du tableau ou le nombre de dimensions dans CREATE TABLE n'a qu'un but documentaire. Le comportement de l'application n'en est pas affecté.

Une autre syntaxe, conforme au standard SQL via l'utilisation du mot-clé ARRAY, peut être employée pour les tableaux à une dimension. paye_par_semaine peut être défini ainsi :

```
paye_par_semaine  integer ARRAY[4],
```

ou si aucune taille du tableau n'est spécifiée :

```
    paye_par_semaine  integer ARRAY,
```

Néanmoins, comme indiqué précédemment, PostgreSQL n'impose aucune restriction sur la taille dans tous les cas.

8.15.2. Saisie de valeurs de type tableau

Pour écrire une valeur de type tableau comme une constante littérale, on encadre les valeurs des éléments par des accolades et on les sépare par des virgules (ce n'est pas différent de la syntaxe C utilisée pour initialiser les structures). Des guillemets doubles peuvent être positionnés autour des valeurs des éléments. C'est d'ailleurs obligatoire si elles contiennent des virgules ou des accolades (plus de détails ci-dessous). Le format général d'une constante de type tableau est donc le suivant :

```
'{ val1 delim val2 delim ... }'
```

où *delim* est le caractère de délimitation pour ce type, tel qu'il est enregistré dans son entrée pg_type. Parmi les types de données standards fournis par la distribution PostgreSQL, tous utilisent une virgule (,), sauf pour le type box qui utilise un point-virgule (;). Chaque *val* est soit une constante du type des éléments du tableau soit un sous-tableau.

Exemple de constante tableau :

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

Cette constante a deux dimensions, un tableau 3 par 3 consistant en trois sous-tableaux d'entiers.

Pour initialiser un élément d'un tableau à NULL, on écrit NULL pour la valeur de cet élément. (Toute variante majuscule et/ou minuscule de NULL est acceptée.) Si « NULL » doit être utilisé comme valeur de chaîne, on place des guillemets doubles autour.

Ces types de constantes tableau sont en fait un cas particulier des constantes de type générique abordées dans la Section 4.1.2.7. La constante est traitée initialement comme une chaîne et passée à la routine de conversion d'entrées de tableau. Une spécification explicite du type peut être nécessaire.

Quelques instructions INSERT :

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"rendez-vous", "repas"}, {"entrainement",
 "présentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"petit-déjeuner", "consultation"}, {"rendez-vous",
 "repas"}}');
```

Le résultat des deux insertions précédentes ressemble à :

```
SELECT * FROM sal_emp;
 nom   |      paye_par_semaine     |      planning
-------+---------------------------+--------------------
Bill   | {10000,10000,10000,10000} | {{rendez-vous,repas},
{entrainement,présentation}}
Carol  | {20000,25000,25000,25000} | {{petit-
déjeuner,consultation},{rendez-vous,repas}}
(2 rows)
```

Les tableaux multidimensionnels doivent avoir des échelles correspondantes pour chaque dimension. Une différence cause la levée d'une erreur. Par exemple :

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"rendez-vous", "repas"}, {"rendez-vous"}}');
ERROR:  multidimensional arrays must have array expressions with
 matching dimensions
```

La syntaxe du constructeur ARRAY peut aussi être utilisée :

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['rendez-vous', 'repas'],
 ['entrainement','présentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['petit-déjeuner', 'consultation'], ['rendez-vous',
 'repas']]);
```

Les éléments du tableau sont des constantes SQL ordinaires ou des expressions ; par exemple, les chaînes de caractères littérales sont encadrées par des guillemets simples au lieu de guillemets doubles comme cela est le cas dans un tableau littéral. La syntaxe du constructeur ARRAY est discutée plus en profondeur dans la Section 4.2.12.

8.15.3. Accès aux tableaux

Quelques requêtes lancées sur la table permettent d'éclairer le propos précédent. Tout d'abord, l'accès à un seul élément du tableau. Cette requête retrouve le nom des employés dont la paye a changé au cours de la deuxième semaine :

```
SELECT nom FROM sal_emp WHERE paye_par_semaine[1] <>
 paye_par_semaine[2];

 nom
-------
 Carol
(1 row)
```

Les indices du tableau sont écrits entre crochets. Par défaut, PostgreSQL utilise la convention des indices commençant à 1 pour les tableaux, c'est-à-dire un tableau à *n* éléments commence avec `array[1]` et finit avec `array[`*n*`]`.

Récupérer la paye de la troisième semaine de tous les employés :

```
SELECT paye_par_semaine[3] FROM sal_emp;

 paye_par_semaine
------------------
          10000
          25000
(2 rows)
```

Il est également possible d'accéder à des parties rectangulaires arbitraires ou à des sous-tableaux. Une partie de tableau est indiquée par l'écriture *`extrémité basse`*`:`*`extrémité haute`* sur n'importe quelle dimension. Ainsi, la requête suivante retourne le premier élément du planning de Bill pour les deux premiers jours de la semaine :

```
SELECT planning[1:2][1:1] FROM sal_emp WHERE nom = 'Bill';

      planning
--------------------
 {{rendez-vous},{entrainement}}
(1 row)
```

Si l'une des dimensions est écrite comme une partie, c'est-à-dire si elle contient le caractère deux-points, alors toutes les dimensions sont traitées comme des parties. Toute dimension qui n'a qu'un numéro (pas de deux-points), est traitée comme allant de 1 au nombre indiqué. Par exemple, `[2]` est traitée comme `[1:2]`, comme le montre cet exemple :

```
SELECT planning[1:2][2] FROM sal_emp WHERE nom = 'Bill';

         planning
---------------------------
 {{rendez-vous,repas},{entrainement,présentation}}
(1 row)
```

Pour éviter toute confusion avec le cas sans indice, il est préférable d'utiliser la syntaxe avec indice pour toutes les dimensions, c'est-à-dire `[1:2][1:1]` et non pas `[2][1:1]`.

Il est possible d'omettre la *`limite basse`* et/ou la *`limite haute`* dans les indices. La limite manquante est remplacée par la limite basse ou haute des dimensions du tableau. Par exemple :

```
SELECT planning[:2][2:] FROM sal_emp WHERE nom = 'Bill';

        planning
------------------------
 {{lunch},{presentation}}
(1 row)
```

```
SELECT planning[:][1:1] FROM sal_emp WHERE nom = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

Une expression indicée de tableau retourne NULL si le tableau ou une des expressions est NULL. De plus, NULL est renvoyé si un indice se trouve en dehors de la plage du tableau (ce cas n'amène pas d'erreur). Par exemple, si `planning` a les dimensions `[1:3][1:2]`, faire référence à `planning[3][3]` donne un résultat NULL. De la même façon, une référence sur un tableau avec une valeur d'indices incorrecte retourne une valeur NULL plutôt qu'une erreur.

Une expression de découpage d'un tableau est aussi NULL si, soit le tableau, soit une des expressions indicées est NULL. Néanmoins, dans certains cas particuliers comme la sélection d'une partie d'un tableau complètement en dehors de la plage de ce dernier, l'expression de cette partie est un tableau vide (zéro dimension) et non pas un tableau NULL. (Ceci ne correspond pas au comportement sans indice, et est fait pour des raisons historiques.) Si la partie demandée surcharge partiellement les limites du tableau, alors elle est réduite silencieusement à la partie surchargée au lieu de renvoyer NULL.

Les dimensions actuelles de toute valeur de type tableau sont disponibles avec la fonction `array_dims` :

```
SELECT array_dims(planning) FROM sal_emp WHERE nom = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

`array_dims` donne un résultat de type `text`, ce qui est pratique à lire, mais peut s'avérer plus difficile à interpréter par les programmes. Les dimensions sont aussi récupérables avec `array_upper` et `array_lower`, qui renvoient respectivement la limite haute et la limite basse du tableau précisé :

```
SELECT array_upper(planning, 1) FROM sal_emp WHERE nom = 'Carol';

 array_upper
-------------
           2
(1 row)
```

`array_length` renverra la longueur de la dimension indiquée pour le tableau :

```
SELECT array_length(planning, 1) FROM sal_emp WHERE nom = 'Carol';

 array_length
--------------
            2
(1 row)
```

`cardinality` renvoie le nombre total d'éléments d'un tableau sur toutes ses dimensions. Autrement dit, c'est le nombre de lignes que renverrait un appel à la fonction `unnest` :

```
SELECT cardinality(planning) FROM sal_emp WHERE nom = 'Carol';
```

```
 cardinality
-------------
           4
(1 row)
```

8.15.4. Modification de tableaux

La valeur d'un tableau peut être complètement remplacée :

```
UPDATE sal_emp SET paye_par_semaine = '{25000,25000,27000,27000}'
    WHERE nom = 'Carol';
```

ou en utilisant la syntaxe de l'expression ARRAY :

```
UPDATE sal_emp SET paye_par_semaine =
 ARRAY[25000,25000,27000,27000]
    WHERE nom = 'Carol';
```

On peut aussi mettre à jour un seul élément d'un tableau :

```
UPDATE sal_emp SET paye_par_semaine[4] = 15000
    WHERE nom = 'Bill';
```

ou faire une mise à jour par tranche :

```
UPDATE sal_emp SET paye_par_semaine[1:2] = '{27000,27000}'
    WHERE nom = 'Carol';
```

Les syntaxes des indices avec la *`limite basse`* et/ou la limite *`upper-bound`* omise peuvent aussi être utilisées lors de la mise à jour d'une valeur d'un tableau qui est différent de NULL ou à plus de zéro dimension (sinon, il n'existe pas de limite à substituer).

Un tableau peut être agrandi en y stockant des éléments qui n'y sont pas déjà présents. Toute position entre ceux déjà présents et les nouveaux éléments est remplie avec la valeur NULL. Par exemple, si le tableau `mon_tableau` a actuellement quatre éléments, il en aura six après une mise à jour qui affecte `mon_tableau[6]`, car `mon_tableau[5]` est alors rempli avec une valeur NULL. Actuellement, l'agrandissement de cette façon n'est autorisé que pour les tableaux à une dimension, pas pour les tableaux multidimensionnels.

L'affectation par parties d'un tableau permet la création de tableaux dont l'indice de départ n'est pas 1. On peut ainsi affecter, par exemple, `mon_tableau[-2:7]` pour créer un tableau avec des valeurs d'indices allant de -2 à 7.

Les valeurs de nouveaux tableaux peuvent aussi être construites en utilisant l'opérateur de concaténation, `||` :

```
SELECT ARRAY[1,2] || ARRAY[3,4];
   ?column?
---------------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

L'opérateur de concaténation autorise un élément à être placé au début ou à la fin d'un tableau à une dimension. Il accepte aussi deux tableaux à *N* dimensions, ou un tableau à *N* dimensions et un à *N +1* dimensions.

Quand un élément seul est poussé soit au début soit à la fin d'un tableau à une dimension, le résultat est un tableau avec le même indice bas que l'opérande du tableau. Par exemple :

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

Lorsque deux tableaux ayant un même nombre de dimensions sont concaténés, le résultat conserve la limite inférieure de l'opérande gauche. Le résultat est un tableau comprenant chaque élément de l'opérande gauche suivi de chaque élément de l'opérande droit. Par exemple :

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

Lorsqu'un tableau à *N* dimensions est placé au début ou à la fin d'un tableau à *N+1* dimensions, le résultat est analogue au cas ci-dessus. Chaque sous-tableau de dimension *N* est en quelque sorte un élément de la dimension externe d'un tableau à *N+1* dimensions. Par exemple :

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

Un tableau peut aussi être construit en utilisant les fonctions `array_prepend`, `array_append` ou `array_cat`. Les deux premières ne supportent que les tableaux à une dimension alors que `array_cat` supporte les tableaux multidimensionnels. Quelques exemples :

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
```

```
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
   array_cat
---------------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

Dans les cas simples, l'opération de concaténation discutée ci-dessus est préférée à l'utilisation directe de ces fonctions. Néanmoins, comme l'opérateur de concaténation est surchargé pour servir les trois cas, certaines utilisations peuvent bénéficier de l'utilisation d'une fonction pour éviter toute ambiguïté. Par exemple :

```
SELECT ARRAY[1, 2] || '{3, 4}';  -- le littéral non typé est pris
 pour un tableau
 ?column?
-----------
 {1,2,3,4}

SELECT ARRAY[1, 2] || '7';                -- idem pour celui-ci
ERROR:  malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL;               -- pareil pour un NULL
 ?column?
----------
 {1,2}
(1 row)

SELECT array_append(ARRAY[1, 2], NULL);    -- ceci peut être voulu
 array_append
--------------
 {1,2,NULL}
```

Dans l'exemple ci-dessus, l'analyseur voit un tableau d'entiers d'un côté de l'opérateur de concaténation et une constante de type indéterminé de l'autre. L'heuristique utilisée pour résoudre le type de la constante revient à assumer qu'elle est de même type que l'autre entrée de l'opérateur -- dans ce cas, un tableau d'entiers. Donc, l'opérateur de concaténation est supposé représenter `array_cat`, et non pas `array_append`. Quand le choix est erroné, cela peut se corriger en convertissant la constante dans le type de données d'un élément du tableau. L'utilisation de la fonction `array_append` peut être préférable.

8.15.5. Recherche dans les tableaux

Pour rechercher une valeur dans un tableau, il faut vérifier chaque valeur dans le tableau. Ceci peut se faire à la main lorsque la taille du tableau est connue. Par exemple :

```
SELECT * FROM sal_emp WHERE paye_par_semaine[1] = 10000 OR
                            paye_par_semaine[2] = 10000 OR
                            paye_par_semaine[3] = 10000 OR
                            paye_par_semaine[4] = 10000;
```

Ceci devient toutefois rapidement fastidieux pour les gros tableaux et n'est pas très utile si la taille du tableau n'est pas connue. Une autre méthode est décrite dans la Section 9.24. La requête ci-dessus est remplaçable par :

```
SELECT * FROM sal_emp WHERE 10000 = ANY (paye_par_semaine);
```

De la même façon, on trouve les lignes où le tableau n'a que des valeurs égales à 10000 avec :

```
SELECT * FROM sal_emp WHERE 10000 = ALL (paye_par_semaine);
```

Sinon, la fonction `generate_subscripts` peut être utilisée. Par exemple :

```
SELECT * FROM
   (SELECT paye_par_semaine,
           generate_subscripts(paye_par_semaine, 1) AS s
      FROM sal_emp) AS foo
 WHERE paye_par_semaine[s] = 10000;
```

Cette fonction est décrite dans Tableau 9.65.

Vous pouvez aussi chercher dans un tableau en utilisant l'opérateur &&, qui vérifie si l'opérande gauche a des éléments communs avec l'opérande droit. Par exemple :

```
SELECT * FROM sal_emp WHERE paye_par_semaine && ARRAY[10000];
```

Les opérateurs sur les tableaux sont décrits plus en profondeur dans Section 9.19. Leurs performances peuvent profiter d'un index approprié, comme décrit dans Section 11.2.

Vous pouvez aussi rechercher des valeurs spécifiques dans un tableau en utilisant les fonctions `array_position` et `array_positions`. La première renvoie l'indice de la première occurrence d'une valeur dans un tableau. La seconde renvoie un tableau avec les indices de toutes les occurrences de la valeur dans le tableau. Par exemple :

```
SELECT
 array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'],
 'mon');
 array_position
----------------
              2
(1 row)

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
-----------------
 {1,4,8}
(1 row)
```

> **Astuce**
>
> Les tableaux ne sont pas des ensembles ; rechercher des éléments spécifiques dans un tableau peut être un signe d'une mauvaise conception de la base de données. On utilise plutôt une

table séparée avec une ligne pour chaque élément faisant partie du tableau. Cela simplifie la recherche et fonctionne mieux dans le cas d'un grand nombre d'éléments.

8.15.6. Syntaxe d'entrée et de sortie des tableaux

La représentation externe du type texte d'une valeur de type tableau consiste en des éléments interprétés suivant les règles de conversion d'entrées/sorties pour le type de l'élément du tableau, plus des décorations indiquant la structure du tableau. L'affichage est constitué d'accolades (`{` et `}`) autour des valeurs du tableau et de caractères de délimitation entre éléments adjacents. Le caractère délimiteur est habituellement une virgule (`,`) mais peut différer : il est déterminé par le paramètre `typdelim` du type de l'élément tableau. Parmi les types de données standards supportés par l'implantation de PostgreSQL, seul le type `box` utilise un point-virgule (`;`), tous les autres utilisant la virgule. Dans un tableau multidimensionnel, chaque dimension (row, plane, cube, etc.) utilise son propre niveau d'accolades et les délimiteurs doivent être utilisés entre des entités adjacentes au sein d'accolades de même niveau.

La routine de sortie du tableau place des guillemets doubles autour des valeurs des éléments si ce sont des chaînes vides, si elles contiennent des accolades, des caractères délimiteurs, des guillemets doubles, des antislashs ou des espaces ou si elles correspondent à `NULL`. Les guillemets doubles et les antislashs intégrés aux valeurs des éléments sont échappés à l'aide d'un antislash. Pour les types de données numériques, on peut supposer sans risque que les doubles guillemets n'apparaissent jamais, mais pour les types de données texte, il faut être préparé à gérer la présence et l'absence de guillemets.

Par défaut, la valeur de la limite basse d'un tableau est initialisée à 1. Pour représenter des tableaux avec des limites basses différentes, les indices du tableau doivent être indiqués explicitement avant d'écrire le contenu du tableau. Cet affichage est constitué de crochets (`[]`) autour de chaque limite basse et haute d'une dimension avec un délimiteur deux-points (`:`) entre les deux. L'affichage des dimensions du tableau est suivi par un signe d'égalité (`=`). Par exemple :

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1)
 AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

La routine de sortie du tableau inclut les dimensions explicites dans le résultat uniquement lorsqu'au moins une limite basse est différente de 1.

Si la valeur écrite pour un élément est `NULL` (toute variante), l'élément est considéré NULL. La présence de guillemets ou d'antislashs désactive ce fonctionnement et autorise la saisie de la valeur littérale de la chaîne « NULL ». De plus, pour une compatibilité ascendante avec les versions antérieures à la version 8.2 de PostgreSQL, le paramètre de configuration array_nulls doit être désactivé (`off`) pour supprimer la reconnaissance de `NULL` comme un NULL.

Comme indiqué précédemment, lors de l'écriture d'une valeur de tableau, des guillemets doubles peuvent être utilisés autour de chaque élément individuel du tableau. Il *faut* le faire si leur absence autour d'un élément induit en erreur l'analyseur de tableau. Par exemple, les éléments contenant des crochets, virgules (ou tout type de données pour le caractère délimiteur correspondant), guillemets doubles, antislashs ou espace (en début comme en fin) doivent avoir des guillemets doubles. Les chaînes vides et les chaînes `NULL` doivent aussi être entre guillemets. Pour placer un guillemet double ou un antislash dans une valeur d'élément d'un tableau, faites le précéder d'un antislash. Alternativement, il est possible de se passer de guillemets et d'utiliser l'échappement par antislash pour

protéger tous les caractères de données qui seraient autrement interprétés en tant que caractères de syntaxe de tableau.

Des espaces peuvent être ajoutées avant un crochet gauche ou après un crochet droit. Comme avant tout élément individuel. Dans tous ces cas-là, les espaces sont ignorées. En revanche, les espaces à l'intérieur des éléments entre guillemets doubles ou entourées de caractères autres que des espaces ne sont pas ignorées.

> **Astuce**
>
> La syntaxe du constructeur ARRAY (voir Section 4.2.12) est souvent plus facile à utiliser que la syntaxe de tableau littéral lors de l'écriture des valeurs du tableau en commandes SQL. Avec ARRAY, les valeurs de l'élément individuel sont écrites comme elles le seraient si elles ne faisaient pas partie d'un tableau.

8.16. Types composites

Un *type composite* représente la structure d'une ligne ou d'un enregistrement ; il est en essence une simple liste de noms de champs et de leurs types de données. PostgreSQL autorise l'utilisation de types composites identiques de plusieurs façons à l'utilisation des types simples. Par exemple, une colonne d'une table peut être déclarée comme étant de type composite.

8.16.1. Déclaration de types composites

Voici deux exemples simples de définition de types composites :

```
CREATE TYPE complexe AS (
    r       double precision,
    i       double precision
);

CREATE TYPE element_inventaire AS (
    nom             text,
    id_fournisseur  integer,
    prix            numeric
);
```

La syntaxe est comparable à CREATE TABLE, sauf que seuls les noms de champs et leurs types peuvent être spécifiés ; aucune contrainte (telle que NOT NULL) ne peut être incluse actuellement. Notez que le mot-clé AS est essentiel ; sans lui, le système penserait à un autre genre de commande CREATE TYPE et vous obtiendriez d'étranges erreurs de syntaxe.

Après avoir défini les types, nous pouvons les utiliser pour créer des tables :

```
CREATE TABLE disponible (
    element   element_inventaire,
    nombre    integer
);

INSERT INTO disponible VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

ou des fonctions :

```
CREATE FUNCTION prix_extension(element_inventaire, integer) RETURNS
 numeric
AS 'SELECT $1.prix * $2' LANGUAGE SQL;
```

```
SELECT prix_extension(element, 10) FROM disponible;
```

Quand vous créez une table, un type composite est automatiquement créé, avec le même nom que la table, pour représenter le type de ligne de la table. Par exemple, si nous avions dit :

```
CREATE TABLE element_inventaire (
    nom             text,
    id_fournisseur  integer REFERENCES fournisseur,
    prix            numeric CHECK (prix > 0)
);
```

alors le même type composite `element_inventaire` montré ci-dessus aurait été créé et pourrait être utilisé comme ci-dessus. Néanmoins, notez une restriction importante de l'implémentation actuelle : comme aucune contrainte n'est associée avec un type composite, les contraintes indiquées dans la définition de la table *ne sont pas appliquées* aux valeurs du type composite en dehors de la table. (Pour contourner ceci, créer un *domaine* sur le type composite, et appliquer les contraintes désirées en tant que contraintes `CHECK` du domaine.)

8.16.2. Construire des valeurs composites

Pour écrire une valeur composite comme une constante littérale, englobez les valeurs du champ dans des parenthèses et séparez-les par des virgules. Vous pouvez placer des guillemets doubles autour de chaque valeur de champ et vous devez le faire si elle contient des virgules ou des parenthèses (plus de détails ci-dessous). Donc, le format général d'une constante composite est le suivant :

```
'( val1 , val2 , ... )'
```

Voici un exemple :

```
'("fuzzy dice",42,1.99)'
```

qui serait une valeur valide du type `element_inventaire` défini ci-dessus. Pour rendre un champ NULL, n'écrivez aucun caractère dans sa position dans la liste. Par exemple, cette constante spécifie un troisième champ NULL :

```
'("fuzzy dice",42,)'
```

Si vous voulez un champ vide au lieu d'une valeur NULL, saisissez deux guillemets :

```
'("",42,)'
```

Ici, le premier champ est une chaîne vide non NULL alors que le troisième est NULL.

(Ces constantes sont réellement seulement un cas spécial de constantes génériques de type discutées dans la Section 4.1.2.7. La constante est initialement traitée comme une chaîne et passée à la routine de conversion de l'entrée de type composite. Une spécification explicite de type pourrait être nécessaire pour préciser le type à utiliser pour la conversion de la constante.)

La syntaxe d'expression `ROW` pourrait aussi être utilisée pour construire des valeurs composites. Dans la plupart des cas, ceci est considérablement plus simple à utiliser que la syntaxe de chaîne littérale, car vous n'avez pas à vous inquiéter des multiples couches de guillemets. Nous avons déjà utilisé cette méthode ci-dessus :

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

Le mot-clé ROW est optionnel si vous avez plus d'un champ dans l'expression, donc ceci peut être simplifié avec

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

La syntaxe de l'expression ROW est discutée avec plus de détails dans la Section 4.2.13.

8.16.3. Accéder aux types composites

Pour accéder à un champ d'une colonne composite, vous pouvez écrire un point et le nom du champ, un peu comme la sélection d'un champ à partir d'un nom de table. En fait, c'est tellement similaire que vous pouvez souvent utiliser des parenthèses pour éviter une confusion de l'analyseur. Par exemple, vous pouvez essayer de sélectionner des sous-champs à partir de notre exemple de table, `disponible`, avec quelque chose comme :

```
SELECT element.nom FROM disponible WHERE element.prix > 9.99;
```

Ceci ne fonctionnera pas, car le nom `element` est pris pour le nom d'une table, et non pas d'une colonne de `disponible`, suivant les règles de la syntaxe SQL. Vous devez l'écrire ainsi :

```
SELECT (element).nom FROM disponible WHERE (element).prix > 9.99;
```

ou si vous avez aussi besoin d'utiliser le nom de la table (par exemple dans une requête multitable), de cette façon :

```
SELECT (disponible.element).nom FROM disponible WHERE
 (disponible.element).prix > 9.99;
```

Maintenant, l'objet entre parenthèses est correctement interprété comme une référence à la colonne `element`, puis le sous-champ peut être sélectionné à partir de lui.

Des problèmes syntaxiques similaires s'appliquent quand vous sélectionnez un champ à partir d'une valeur composite. En fait, pour sélectionner un seul champ à partir du résultat d'une fonction renvoyant une valeur composite, vous aurez besoin d'écrire quelque chose comme :

```
SELECT (ma_fonction(...)).champ FROM ...
```

Sans les parenthèses supplémentaires, ceci provoquera une erreur.

Le nom du champ spécial * signifie « tous les champs », comme expliqué dans Section 8.16.5.

8.16.4. Modifier les types composites

Voici quelques exemples de la bonne syntaxe pour insérer et mettre à jour des colonnes composites. Tout d'abord, pour insérer ou modifier une colonne entière :

```
INSERT INTO matab (col_complexe) VALUES((1.1,2.2));

UPDATE matab SET col_complexe = ROW(1.1,2.2) WHERE ...;
```

Le premier exemple omet ROW, le deuxième l'utilise ; nous pouvons le faire des deux façons.

Nous pouvons mettre à jour un sous-champ individuel d'une colonne composite :

```
UPDATE matab SET col_complexe.r = (col_complexe).r + 1 WHERE ...;
```

Notez ici que nous n'avons pas besoin de (et, en fait, ne pouvons pas) placer des parenthèses autour des noms de colonnes apparaissant juste après SET, mais nous avons besoin de parenthèses lors de la référence à la même colonne dans l'expression à droite du signe d'égalité.

Et nous pouvons aussi spécifier des sous-champs comme cibles de la commande INSERT :

```
INSERT INTO matab (col_complexe.r, col_complexe.i) VALUES(1.1,
 2.2);
```

Si tous les sous-champs d'une colonne ne sont pas spécifiés, ils sont remplis avec une valeur NULL.

8.16.5. Utiliser des types composites dans les requêtes

Il existe différentes règles spéciales de syntaxe et de différents comportements associés avec les types composites dans les requêtes. Ces règles fournissent des raccourcis utiles, mais peuvent être difficiles à appréhender si vous ne connaissez pas la logique qui y est associée.

Dans PostgreSQL, une référence à un nom de table (ou à un alias) dans une requête est réellement une référence au type composite de la ligne courante de la table. Par exemple, si nous avons une table `element_inventaire` comme définie ci-dessus, nous pouvons écrire :

```
SELECT c FROM element_inventaire c;
```

Cette requête renvoie une seule colonne comprenant une valeur composite, et nous pourrions obtenir l'affichage suivant :

```
           c
------------------------
 ("fuzzy dice",42,1.99)
(1 row)
```

Il faut noter néanmoins que les noms simples (c.-à-d. sans qualifiant) sont traités comme des noms de colonnes puis comme des noms de table s'il n'y a pas de correspondance avec les noms de colonnes. Donc cet exemple fonctionne seulement parce qu'il n'existe pas de colonne nommée `c` dans les tables de la requête.

La syntaxe habituelle avec des noms de colonne qualifiés (comme *`nom_table.nom_colonne`*) peut se comprendre en appliquant la sélection de champs à la valeur composite de la ligne actuelle de la table. (Pour des raisons d'efficacité, ce n'est pas réellement implémenté de cette façon.)

Quand nous écrivons

```
SELECT c.* FROM element_inventaire c;
```

alors, d'après le standard SQL, nous devrions obtenir le contenu de la table étendu en des colonnes séparées :

```
    nom     | id_fournisseur | prix
------------+----------------+-------
 fuzzy dice |             42 |  1.99
(1 row)
```

comme si la requête avait été écrite ainsi :

```
SELECT c.nom, c.id_fournisseur, c.prix FROM element_inventaire c;
```

PostgreSQL appliquera ce comportement étendu à toute expression de valeur composite, bien que, comme indiqué ci-dessus, il est nécessaire d'ajouter des parenthèses autour de la valeur à qui `.*` est appliquée à chaque fois qu'il ne s'agit pas d'un nom de table. Par exemple, si `ma_fonction()` est une fonction renvoyant un type composite avec les colonnes `a`, `b` et `c`, alors ces deux requêtes donnent le même résultat :

```
SELECT (ma_fonction(x)).* FROM une_table;
SELECT (ma_fonction(x)).a, (ma_fonction(x)).b, (ma_fonction(x)).c
 FROM une_table;
```

> **Astuce**
>
> PostgreSQL gère le fait d'étendre les colonnes en transformant la première forme en la seconde. De ce fait, dans cet exemple, `ma_fonction()` serait appelé trois fois par ligne, quelle que soit la syntaxe utilisée. S'il s'agit d'une fonction peu performante, vous pourriez souhaiter éviter cela, ce que vous pouvez faire avec une requête de ce type :
>
> ```
> SELECT (m).* FROM (SELECT ma_fonction(x) AS m FROM une_table
>  OFFSET 0) ss;
> ```
>
> Placer la fonction dans un élément `LATERAL` du `FROM` l'aide à ne pas être invoquée plus d'une fois par ligne. `m.*` est toujours étendu en `m.a, m.b, m.c`, mais maintenant ces variables sont juste des références à la sortie de l'élément `FROM`. (Le mot-clé `LATERAL` est optionnel ici, mais nous le montrons pour clarifier que la fonction obtient `x` de la `some_table`.)

La syntaxe *`valeur_composite`*`.*` étend les colonnes avec un résultat de ce type quand il apparaît au niveau haut d'une liste en sortie du `SELECT`, d'une liste `RETURNING` dans des commandes `INSERT`/`UPDATE`/`DELETE`, d'une clause `VALUES`, ou d'un constructeur de ligne. Dans tous les autres contextes (incluant l'imbrication dans une de ces constructions), attacher `.*` à une valeur composite ne change pas la valeur, car cela signifie « toutes les colonnes » et donc la valeur composite est produite de nouveau. Par exemple, si `une_fonction()` accepte un argument de valeur composite, ces requêtes ont un résultat identique :

```
SELECT une_fonction(c.*) FROM element_inventaire c;
SELECT une_fonction(c) FROM element_inventaire c;
```

Dans les deux cas, la ligne courante de `element_inventaire` est passée à la fonction sous la forme d'un seul argument de type composite. Même si `.*` ne fait rien dans de tels cas, l'utiliser est intéressant, car il est clair à sa lecture qu'on attend une valeur composite. En particulier, l'analyseur considérera `c` dans `c.*` comme une référence au nom de la table ou de l'alias, et non pas comme un nom de colonne, pour qu'il n'y ait pas d'ambiguïté. Sans le `.*`, il n'est pas clair si `c` est un nom de table ou de colonne et, de ce fait, l'interprétation préférée sera celle d'un nom de colonne si une colonne nommée `c` existe.

Voici un autre exemple démontrant ces concepts avec toutes ces requêtes qui ont la même signification :

```
SELECT * FROM element_inventaire c ORDER BY c;
SELECT * FROM element_inventaire c ORDER BY c.*;
```

```
SELECT * FROM element_inventaire c ORDER BY ROW(c.*);
```

Toutes ces clauses `ORDER BY` indiquent la valeur composite de la ligne. Néanmoins, si `element_inventaire` contenait une colonne nommée `c`, le premier cas serait différent des autres, car le tri se ferait uniquement sur cette colonne. Avec les noms de colonne indiqués précédemment, ces requêtes sont aussi équivalentes à celles-ci :

```
SELECT * FROM element_inventaire c ORDER BY ROW(c.nom,
 c.id_fournisseur, c.prix);
SELECT * FROM element_inventaire c ORDER BY (c.nom,
 c.id_fournisseur, c.prix);
```

(Le dernier cas utilise un constructeur de ligne avec le mot-clé `ROW` omis.)

Un autre comportement syntaxique spécial avec les valeurs composites est que nous pouvons utiliser la *notation fonctionnelle* pour extraire un champ d'une valeur composite. La façon simple d'expliquer ceci est que les notations *`champ`*`(`*`table`*`)` et *`table.champ`* sont interchangeables. Par exemple, ces requêtes sont équivalentes :

```
SELECT c.nom FROM element_inventaire c WHERE c.prix > 1000;
SELECT nom(c) FROM element_inventaire c WHERE prix(c) > 1000;
```

De plus, si nous avons une fonction qui accepte un seul argument de type composite, nous pouvons l'appeler avec une de ces notations. Ces requêtes sont toutes équivalentes :

```
SELECT une_fonction(c) FROM element_inventaire c;
SELECT une_fonction(c.*) FROM element_inventaire c;
SELECT c.une_fonction FROM element_inventaire c;
```

Cette équivalence entre la notation fonctionnelle et la notation par champ rend possible l'utilisation de fonctions sur les types composites pour implémenter les « champs calculés ». Une application utilisant la dernière requête ci-dessus n'aurait pas besoin d'être directement attentive au fait que `une_fonction` n'est pas une vraie colonne de la table.

> **Astuce**
>
> À cause de ce comportement, il est déconseillé de donner une fonction qui prend un argument de type composite simple du même nom que n'importe quel champ de ce type composite. S'il existe une ambiguïté, l'interprétation du nom de champ sera choisie si la syntaxe de nom de champ est utilisée, alors que la fonction sera choisie si la syntaxe d'appel de fonction est utilisée. Néanmoins, les versions de PostgreSQL antérieures à la 11 choisiront toujours l'interprétation du nom de champ, sauf si la syntaxe de l'appel requiert un appel de fonction. Une façon de forcer l'interprétation en fonction pour les versions antérieures est de qualifier le nom de la fonction avec le nom du schéma, autrement dit *`schéma.fonction`*`(`*`valeurcomposite`*`)`.

8.16.6. Syntaxe en entrée et sortie d'un type composite

La représentation texte externe d'une valeur composite consiste en des éléments qui sont interprétés suivant les règles de conversion d'entrées/sorties pour les types de champs individuels, plus des

décorations indiquant la structure composite. Ces décorations consistent en des parenthèses (`(` et `)`) autour de la valeur entière, ainsi que des virgules (`,`) entre les éléments adjacents. Des espaces blancs en dehors des parenthèses sont ignorés, mais à l'intérieur des parenthèses, ils sont considérés comme faisant partie de la valeur du champ et pourraient ou non être significatifs suivant les règles de conversion de l'entrée pour le type de données du champ. Par exemple, dans :

```
'(  42)'
```

l'espace blanc sera ignoré si le type du champ est un entier, mais pas s'il s'agit d'un champ de type texte.

Comme indiqué précédemment, lors de l'écriture d'une valeur composite, vous pouvez utiliser des guillemets doubles autour de chaque valeur de champ individuel. Vous *devez* le faire si la valeur du champ était susceptible de gêner l'analyseur de la valeur du champ composite. En particulier, les champs contenant des parenthèses, des virgules, des guillemets doubles ou des antislashs doivent être entre guillemets doubles. Pour placer un guillemet double ou un antislash dans la valeur d'un champ composite entre guillemets, faites-le précéder d'un antislash. (De plus, une paire de guillemets doubles à l'intérieur d'une valeur de champ à guillemets doubles est prise pour représenter un caractère guillemet double, en analogie avec les règles des guillemets simples dans les chaînes SQL littérales.) Autrement, vous pouvez éviter les guillemets et utiliser l'échappement par antislash pour protéger tous les caractères de données qui auraient été pris pour une syntaxe composite.

Une valeur de champ composite vide (aucun caractère entre les virgules ou parenthèses) représente une valeur NULL. Pour écrire une valeur qui est une chaîne vide plutôt qu'une valeur NULL, écrivez `""`.

La routine de sortie composite placera des guillemets doubles autour des valeurs de champs si elles sont des chaînes vides ou si elles contiennent des parenthèses, virgules, guillemets doubles, antislash ou espaces blancs. (Faire ainsi pour les espaces blancs n'est pas essentiel, mais aide à la lecture.) Les guillemets doubles et antislashs dans les valeurs des champs seront doublés.

> **Note**
>
> Rappelez-vous que ce que vous allez saisir dans une commande SQL sera tout d'abord interprété comme une chaîne littérale, puis comme un composite. Ceci double le nombre d'antislash dont vous avez besoin (en supposant que la syntaxe d'échappement des chaînes soit utilisée). Par exemple, pour insérer un champ `text` contenant un guillemet double et un antislash dans une valeur composite, vous devez écrire :
>
> ```
> INSERT ... VALUES ('("\"\\")');
> ```
>
> Le processeur des chaînes littérales supprime un niveau d'antislash de façon que ce qui arrive à l'analyseur de valeurs composites ressemble à `("\"\\")`. À son tour, la chaîne remplie par la routine d'entrée du type de données `text` devient `"\`. (Si nous étions en train de travailler avec un type de données dont la routine d'entrée traite aussi les antislashs spécialement, `bytea` par exemple, nous pourrions avoir besoin d'au plus huit antislashs dans la commande pour obtenir un antislash dans le champ composite stocké.) Le guillemet dollar (voir Section 4.1.2.4) pourrait être utilisé pour éviter le besoin des antislashs doublés.

> **Astuce**
>
> La syntaxe du constructeur `ROW` est habituellement plus simple à utiliser que la syntaxe du littéral composite lors de l'écriture de valeurs composites dans des commandes SQL. Dans `ROW`, les valeurs individuelles d'un champ sont écrites de la même façon qu'elles l'auraient été en n'étant pas membres du composite.

8.17. Types intervalle de valeurs

Les types intervalle de valeurs sont des types de données représentant un intervalle de valeurs d'un certain type d'élément (appelé *sous-type* de l'intervalle). Par exemple, des intervalles de `timestamp` pourraient être utilisés pour représenter les intervalles de temps durant lesquels une salle de réunion est réservée. Dans ce cas, le type de données est `tsrange` (la version abrégée de « timestamp range »), et `timestamp` est le sous-type. Le sous-type doit avoir un tri complet pour que les valeurs d'élément incluses soient bien définies, avant ou après l'intervalle de valeurs.

Les types intervalle de valeurs sont utiles parce qu'ils représentent de nombreuses valeurs d'élément en une seule valeur d'intervalle, et que des concepts comme le chevauchement d'intervalles peuvent être exprimés clairement. L'utilisation d'intervalle de temps et de date pour des besoins de planification est l'exemple le plus parlant ; mais les intervalles de prix, intervalles de mesure pour un instrument et ainsi de suite peuvent également être utiles.

Chaque type range dispose d'un type multirange correspondant. Un multirange est une liste triée de ranges non contigus, non vides et non NULL. La plupart des opérateurs ranges fonctionnent aussi sur les multiranges, et ils ont quelques fonctions à eux.

8.17.1. Types internes d'intervalle de valeurs range et multirange

PostgreSQL fournit nativement les types intervalle de valeurs suivants :

- `INT4RANGE` -- Intervalle d'`integer`, `int4multirange` -- correspondance Multirange
- `INT8RANGE` -- Intervalle de `bigint`, `int8multirange` -- correspondance Multirange
- `NUMRANGE` -- Intervalle de `numeric`, `nummultirange` -- correspondance Multirange
- `TSRANGE` -- Intervalle de `timestamp without time zone`, `tsmultirange` -- correspondance Multirange
- `TSTZRANGE` -- Intervalle de `timestamp with time zone`, `tstzmultirange` -- correspondance Multirange
- `DATERANGE` -- Intervalle de `date`, `datemultirange` -- correspondance Multirange

Vous pouvez en plus définir vos propres types intervalle de valeurs ; voir CREATE TYPE pour plus d'informations.

8.17.2. Exemples

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
  ( 1108, '[2010-01-01 14:30, 2010-01-01 15:30)' );

-- Inclusion
SELECT int4range(10, 20) @> 3;

-- Chevauchement
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Extraire la borne inférieure
SELECT upper(int8range(15, 25));

-- Calculer l'intersection
SELECT int4range(10, 20) * int4range(15, 25);
```

```
-- Est-ce que l'intervalle est vide ?
SELECT isempty(numrange(1, 5));
```

Voir Tableau 9.54 et Tableau 9.56 pour la liste complète des opérateurs et fonctions sur les types intervalle de valeurs.

8.17.3. Bornes inclusives et exclusives

Chaque intervalle de valeurs non vide a deux bornes, la borne inférieure et la borne supérieure. Tous les points entre ces valeurs sont inclus dans l'intervalle. Une borne inclusive signifie que le point limite lui-même est également inclus dans l'intervalle, alors qu'une borne exclusive signifie que ce point limite n'est pas inclus dans l'intervalle.

Dans un intervalle affiché sous la forme de texte, une borne inclusive inférieure est représentée par « `[` » tandis qu'une borne exclusive inférieure est représentée par « `(` ». De la même façon, une borne inclusive supérieure est représentée par « `]` » tandis qu'une borne exclusive supérieure est représentée par « `)` ». (Voir Section 8.17.5 pour plus de détails.)

Les fonctions `lower_inc` et `upper_inc` testent respectivement si les bornes inférieures et supérieures d'une valeur d'intervalle sont inclusives.

8.17.4. Intervalles de valeurs infinis (sans borne)

La limite basse d'un intervalle peut être omise, signifiant que toutes les valeurs inférieures à la limite haute sont incluses dans l'intervalle, par exemple `(,3]`. De la même façon, si la limite haute d'un intervalle est omise, alors toutes les valeurs supérieures à la limite basse sont incluses dans l'intervalle. Si les limites basse et haute sont omises, toutes les valeurs du type de l'élément sont considérées faire partie de l'intervalle. Indiquer une limite manquante comme inclus fait qu'elle est automatique convertie en exclus, autrement dit `[,]` est converti en `(,)`. Vous pouvez penser à ces valeurs manquantes comme +/-infinity, mais ce sont des valeurs spéciales du type intervalle et sont considérées au delà des valeurs +/-infinity du type de l'élément.

Les types de l'élément qui ont une notion de « infinity » peuvent les utiliser comme limites explicites. Par exemple, pour les intervalles du type timestamp, `[today,infinity)` exclut la valeur `infinity` du type `timestamp`, alors que `[today,infinity]` l'inclut, comme le font `[today,)` et `[today,]`.

Les fonctions `lower_inf` et `upper_inf` testent respectivement si les bornes inférieure et supérieure sont infinies.

8.17.5. Saisie/affichage d'intervalle de valeurs

La saisie d'un intervalle de valeurs doit suivre un des modèles suivants:

```
(borne-inférieure,borne-supérieure)
(borne-inférieure,borne-supérieure]
[borne-inférieure,borne-supérieure)
[borne-inférieure,borne-supérieure]
empty
```

Les parenthèses ou crochets indiquent si les bornes inférieure et supérieure sont exclusives ou inclusives, comme décrit précédemment. Notez que le modèle final est `empty`, ce qui représente un intervalle vide (un intervalle qui ne contient aucun point).

La *`borne-inférieure`* peut être une chaîne de caractères valide pour la saisie du sous-type, ou vide pour indiquer qu'il n'y a pas de borne inférieure. De la même façon, la *`borne-supérieure`*

peut être une chaîne de caractères valide pour la saisie du sous-type, ou vide pour indiquer qu'il n'y a pas de borne supérieure.

Chaque borne peut être protégée en entourant la valeur de guillemet double (`"`). C'est nécessaire si la valeur de borne contient des parenthèses, crochets, virgules, guillemets doubles, antislash, puisque, sans cela, ces caractères seraient considérés comme faisant partie de la syntaxe de l'intervalle de valeurs. Pour mettre un guillemet double ou un antislash dans une valeur de borne protégée, faites-le précéder d'un antislash. (Une paire de guillemets doubles dans une borne protégée est également valable pour représenter un caractère guillemet double, de la même manière que la règle pour les guillemets simples dans les chaînes SQL littérales.) Vous pouvez éviter l'emploi des guillemets doubles en échappant avec un antislash tous les caractères qui, sans cela, seraient pris comme une syntaxe d'intervalle de valeurs. De plus, pour écrire une valeur de borne qui est une chaîne vide, écrivez `""`, puisque ne rien écrire signifie une borne infinie.

Des espaces sont autorisés avant et après la valeur de borne, mais chaque espace entre les parenthèses ou les crochets fera partie de la valeur de limite inférieure ou supérieure. (Selon le type d'élément, cela peut être ou ne pas être significatif.)

> **Note**
>
> Ces règles sont très proches de celles de l'écriture de valeurs de champs pour les types composites. Voir Section 8.16.6 pour des commentaires supplémentaires.

Exemples :

```
-- inclut 3, n'inclut pas 7, et inclut tous les points entre
SELECT '[3,7)'::int4range;

-- n'inclut ni 3 ni 7, mais inclut tous les points entre
SELECT '(3,7)'::int4range;

-- n'inclut que l'unique point 4
SELECT '[4,4]'::int4range;

-- n'inclut aucun point (et sera normalisé à 'empty')
SELECT '[4,4)'::int4range;
```

La saisie d'un multirange se fait avec des accolades (`{` et `}`) contenant zéro ou plusieurs ranges valides, séparés par des virgules. Les espaces blancs sont autorisés autour des accolades et des virgules. Ceci a pour but de rappeler la syntaxe des tableaux, bien que les multiranges sont bien plus simples : ils ont juste une dimension et il n'est pas nécessaire de mettre entre guillemets leur contenu. (Les limites de leur ranges pourraient être mises entre guillemets néanmoins.)

Exemples :

```
SELECT '{}'::int4multirange;
SELECT '{[3,7)}'::int4multirange;
SELECT '{[3,7), [8,9)}'::int4multirange;
```

8.17.6. Construire des intervalles de valeurs ranges et multiranges

Chaque type intervalle de valeurs a une fonction constructeur du même nom que le type intervalle. Utiliser le constructeur est souvent plus pratique que d'écrire une constante d'intervalle littérale puisque

cela évite d'avoir à ajouter des guillemets doubles sur les valeurs de borne. Le constructeur accepte deux ou trois arguments. La forme à deux arguments construit un intervalle dans sa forme standard (borne inférieure inclusive, borne supérieure exclusive), alors que la version à trois arguments construit un intervalle avec des bornes de la forme spécifiée par le troisième argument. Le troisième argument doit être la chaîne « `()` », « `(]` », « `[)` » ou « `[]` ». Par exemple :

```
-- La forme complète est : borne inférieure, borne supérieure et
 argument texte indiquant
-- inclusivité/exclusivité des bornes.
SELECT numrange(1.0, 14.0, '(]');

-- Si le troisième argument est omis, '[)' est supposé.
SELECT numrange(1.0, 14.0);

-- Bien que  '(]' soit ici spécifié, à l'affichage la valeur sera
 convertie en sa forme
-- canonique puisque int8range est un type intervalle discret (voir
 ci-dessous).
SELECT int8range(1, 14, '(]');

-- Utiliser NULL pour n'importe laquelle des bornes a pour effet de
 ne pas avoir de borne de ce côté.
SELECT numrange(NULL, 2.2);
```

Chaque type range a aussi un constructeur multirange du même nom que le type multirange. La fonction constructeur prend zéro ou plus d'argumentsqui sont tous des intervalles du type approprié. Par exemple :

```
SELECT nummultirange();
SELECT nummultirange(numrange(1.0, 14.0));
SELECT nummultirange(numrange(1.0, 14.0), numrange(20.0, 25.0));
```

8.17.7. Types intervalle de valeurs discrètes

Un type d'intervalle de valeurs discrètes est un intervalle dont le type d'élément a un « pas » bien défini, comme `integer` ou `date`. Pour ces types, deux éléments peuvent être dits comme étant adjacents, quand il n'y a pas de valeur valide entre eux. Cela contraste avec des intervalles continus, où il y a toujours (ou presque toujours) des valeurs d'autres éléments possibles à identifier entre deux valeurs données. Par exemple, un intervalle de type `numeric` est continu, comme l'est un intervalle de type `timestamp`. (Même si `timestamp` a une limite de précision, et pourrait théoriquement être traité comme discret, il est préférable de le considérer comme continu puisque la taille du pas n'a normalement pas d'intérêt.)

Une autre façon d'imaginer un type d'intervalle de valeurs discrètes est qu'il est possible de déterminer clairement une valeur « suivante » ou « précédente » pour chaque valeur d'élément. En sachant cela, il est possible de convertir des représentations inclusives et exclusives d'une borne d'intervalle, en choisissant la valeur d'élément suivante ou précédente à la place de celle d'origine. Par exemple, dans un type d'intervalle entier, `[4,8]` et `(3,9)` représentent le même ensemble de valeurs, mais cela ne serait pas le cas pour un intervalle de numeric.

Un type d'intervalle discret devrait avoir une fonction de *mise en forme canonique* consciente de la taille du pas désiré pour le type d'élément. La fonction de mise en forme canonique est chargée de convertir des valeurs équivalentes du type d'intervalle pour avoir des représentations identiques, surtout aux voisinages de bornes inclusives ou exclusives. Si une fonction de mise en forme canonique n'est pas spécifiée, alors les intervalles de notations différentes seront toujours traités comme étant différents, même s'ils peuvent en réalité représenter le même ensemble de valeurs.

Les types d'intervalle prédéfinis `int4range`, `int8range`, et `daterange` utilisent tous une forme canonique qui inclut les bornes inférieures et exclut les bornes supérieures ; c'est-à-dire `[)`. Les types intervalles définis par l'utilisateur peuvent cependant utiliser d'autres conventions.

8.17.8. Définir de nouveaux types intervalle de valeurs

Les utilisateurs peuvent définir leurs propres types intervalle de valeurs. La raison la plus commune de le faire est d'utiliser des intervalles de sous-types non prédéfinis. Par exemple, pour définir un nouveau type d'intervalle de valeurs du sous-type `float8` :

```
CREATE TYPE floatrange AS RANGE (
  subtype = float8,
  subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

Puisque `float8` n'a pas de « pas » significatif, nous ne définissons pas de fonction de mise en forme canonique dans cet exemple.

Quand vous définissez votre propre intervalle, vous obtenez automatiquement un type multirange correspondant.

Définir votre propre type intervalle vous permet aussi de spécifier une classe différente d'opérateur ou un collationnement différent, à utiliser, pour modifier l'ordre de tri qui détermine les valeurs tombant dans un intervalle donné.

Si l'on considère que le sous-type est discret plutôt que continu, la commande `CREATE TYPE` devrait spécifier une fonction `canonique`. La fonction de mise en forme canonique prend une valeur d'intervalle en entrée, et doit retourner une valeur d'intervalle équivalente qui peut avoir des bornes et une représentation différente. Les sorties canoniques de deux intervalles qui représentent le même ensemble de valeurs, par exemple les intervalles d'entier `[1, 7]` et `[1,8)` doivent être identiques. La représentation choisie n'a pas d'importance, du moment que deux valeurs équivalentes avec des représentations différentes sont toujours liées à la même valeur avec la même représentation. En plus d'ajuster le format des bornes inclusives et exclusives, une fonction de mise en forme canonique peut arrondir une valeur de borne, dans le cas où la taille de pas désirée est plus grande que ce que le sous-type est capable de stocker. Par exemple, un intervalle de `timestamp` pourrait être défini pour avoir une taille de pas d'une heure, et dans ce cas la fonction de mise en forme canonique nécessiterait d'arrondir les bornes qui ne sont pas multiples d'une heure, ou peut-être déclencher une erreur à la place.

De plus, tout type intervalle devant être utilisé avec des index GiST ou SP-GiST doit définir une différence de sous-type ou une fonction `subtype_diff`. (L'index fonctionnera toujours sans fonction `subtype_diff`, mais il y a de fortes chances qu'il soit considérablement moins efficace qu'avec une fonction de différence.) La fonction de différence du sous-type prend deux valeurs en entrée et renvoie leur différence (par exemple, *X* moins *Y*) représentée sous la forme d'une valeur de type `float8`. Dans notre exemple ci-dessus, la fonction `float8mi` qui soutient l'opérateur moins du type `float8` peut être utilisée ; mais pour tout autre sous-type, une conversion de type serait nécessaire. Un peu de créativité peut se révéler nécessaire pour représenter la différence sous une forme numérique. Dans la mesure du possible, la fonction `subtype_diff` devrait être en accord avec l'ordre de tri impliqué par la classe d'opérateurs et le collationnement sélectionnés ; autrement dit, son résultat doit être positif quand le premier argument est supérieur au second d'après l'ordre de tri.

Voici un exemple moins simplifié d'une fonction `subtype_diff` :

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;
```

```
CREATE TYPE timerange AS RANGE (
    subtype = time,
    subtype_diff = time_subtype_diff
);

SELECT '[11:10, 23:00]'::timerange;
```

Voir CREATE TYPE pour plus d'informations sur la façon de créer des types intervalle de valeurs.

8.17.9. Indexation

Des index GiST et SP-GiST peuvent être créés pour des colonnes de table de type intervalle de valeurs ranges. Des index GiST peuvent aussi être créés pour les colonnes de types multirange. Par exemple, pour créer un index GiST :

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

Un index GiST ou SP-GiST sur un type range peut accélérer les requêtes impliquant ces opérateurs d'intervalle de valeurs : =, &&, <@, @>, <<, >>, -|-, &< et &>. Un index GiST sur des multiranges peut accélérer les requêtes impliquant le même ensemble d'opérateurs multirange. Un index GiST sur des ranges et un index GiST sur des multiranges peut aussi accélérer les requêtes impliquant les opérateurs inter types range vers multirange et multirange vers range : &&, <@, @>, <<, >>, -|-, &< et &>. voir Tableau 9.54 pour plus d'informations.

De plus, les index B-tree et hash peuvent être créés pour des colonnes d'une table de type intervalle de valeurs. Pour ces types d'index, la seule opération d'intervalle véritablement utile est l'égalité. Il y a un ordre de tri pour les index B-tree définis pour les valeurs d'intervalle, correspondant aux opérateurs < et >, mais le tri est plutôt arbitraire et généralement inutile dans la réalité. Le support de B-tree et hash pour les types intervalle de valeurs est à la base destiné à permettre le tri et le hachage de façon interne dans les requêtes, plutôt que pour la création d'un vrai index.

8.17.10. Contraintes sur les intervalles de valeurs

Bien que `UNIQUE` soit une contrainte naturelle pour des valeurs scalaires, c'est en générale inutilisable pour des types intervalle de valeurs. À la place, une contrainte d'exclusion est souvent plus appropriée (voir CREATE TABLE ... CONSTRAINT ... EXCLUDE). Les contraintes d'exclusion permettent la spécification de contraintes telles que le « non chevauchement » sur un type intervalle de valeurs. Par exemple :

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING GIST (during WITH &&)
);
```

Cette contrainte empêchera toute valeur chevauchant une autre présente dans la table à la même heure :

```
INSERT INTO reservation VALUES
   ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO reservation VALUES
   ('[2010-01-01 14:45, 2010-01-01 15:45)');
```

```
ERROR:  conflicting key value violates exclusion constraint
 "reservation_during_excl"
DETAIL:  Key (during)=(["2010-01-01 14:45:00","2010-01-01
 15:45:00")) conflicts
with existing key (during)=(["2010-01-01 11:30:00","2010-01-01
 15:00:00")).
```

Vous pouvez utiliser l'extension `btree_gist` pour définir une contrainte d'exclusion sur des types de données scalaires, qui peuvent alors être combinés avec des exclusions d'intervalle de valeurs pour un maximum de flexibilité. Par exemple, une fois que `btree_gist` est installé, la contrainte suivante ne rejettera les intervalles de valeurs se chevauchant que si le numéro de la salle de conférence est identique :

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation
(
  room TEXT,
  during TSRANGE,
  EXCLUDE USING GIST (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
  ( '123A', '[2010-01-01 14:00, 2010-01-01 15:00)' );
INSERT 0 1

INSERT INTO room_reservation VALUES
  ( '123A', '[2010-01-01 14:30, 2010-01-01 15:30)' );
ERROR:  conflicting key value violates exclusion constraint
 "room_reservation_room_during_excl"
DETAIL:  Key (room, during)=(123A, ["2010-01-01
 14:30:00","2010-01-01 15:30:00")) conflicts
with existing key (room, during)=(123A, ["2010-01-01
 14:00:00","2010-01-01 15:00:00")).

INSERT INTO room_reservation VALUES
  ( '123B', '[2010-01-01 14:30, 2010-01-01 15:30)' );
INSERT 0 1
```

8.18. Types domaine

Un *domaine* est un type de données défini par l'utilisateur. Il est basé sur un autre *type sous-jacent*. En option, il peut avoir des contraintes qui restreignent les valeurs valides à un sous-ensemble de ce que permettrait le type sous-jacent. Pour le reste, il se comporte comme le type sous-jacent -- par exemple, tout opérateur ou fonction qui peut être appliqué au type sous-jacent fonctionne avec le domaine. Le type sous-jacent peut être tout type, interne ou défini par l'utilisateur, type enum, type tableau, type composé, type intervalle ou autre domaine.

Par exemple, nous pouvons créer un domaine sur des entiers qui n'accepte que des valeurs positives :

```
CREATE DOMAIN posint AS integer CHECK (VALUE > 0);
CREATE TABLE mytable (id posint);
INSERT INTO mytable VALUES(1);   -- works
INSERT INTO mytable VALUES(-1);  -- fails
```

Quand un opérateur ou une fonction du type sous-jacent est appliqué à la valeur d'un domaine, le domaine est automatiquement converti vers le type sous-jacent. Donc, par exemple, le résultat de `mytable.id - 1` est considéré être de type `integer`, et non pas `posint`. Nous pouvons écrire `(mytable.id - 1)::posint` pour convertir le résultat avec le type `posint`, causant une nouvelle vérification des contraintes du domaine. Dans ce cas, cela résultera en une erreur si l'expression a été appliquée à une valeur de 1 pour `id`. Affecter une valeur du type sous-jacent à un champ ou variable du type domaine est autorisé sans forcer une conversion explicite, mais les contraintes du domaine seront vérifiées.

Pour plus d'informations, voir CREATE DOMAIN.

8.19. Types identifiant d'objet

Les identifiants d'objets (OID) sont utilisés en interne par PostgreSQL comme clés primaires de différentes tables système. Le type `oid` représente un identifiant d'objet. Il existe aussi différents types alias pour un `oid`, chacun nommés `reg`*`quelquechose`*. Le Tableau 8.26 en donne un aperçu.

Le type `oid` est à ce jour un entier non signé sur quatre octets. Il n'est, de ce fait, pas suffisamment large pour garantir l'unicité au sein d'une base de données volumineuse, voire au sein d'une très grosse table.

Le type `oid` lui-même dispose de peu d'opérations en dehors de la comparaison. Il peut toutefois être converti en entier (integer) et manipulé par les opérateurs habituels des entiers (attention aux possibles confusions entre les entiers signés et non signés dans ce cas).

Les types alias d'OID ne disposent pas d'opérations propres à l'exception des routines spécialisées de saisie et d'affichage. Ces routines acceptent et affichent les noms symboliques des objets système, plutôt que la valeur numérique brute que le type `oid` utilise. Les types alias permettent de simplifier la recherche des valeurs OID des objets. Par exemple, pour examiner les lignes `pg_attribute` en relation avec une table `ma_table`, on peut écrire :

```
SELECT * FROM pg_attribute WHERE attrelid = 'ma_table'::regclass;
```

plutôt que :

```
SELECT * FROM pg_attribute
WHERE attrelid = (SELECT oid FROM pg_class WHERE relname =
 'ma_table');
```

Bien que cela semble une bonne solution, c'est un peu trop simplifié. Un sous-select bien plus compliqué peut être nécessaire pour sélectionner le bon OID s'il existe plusieurs tables nommées `ma_table` dans différents schémas. Le convertisseur de saisie `regclass` gère la recherche de la table en fonction du paramétrage du parcours des schémas et effectue donc la « bonne recherche » automatiquement. De façon similaire, la conversion d'un OID de table en `regclass` pour l'affichage d'un OID numérique est aisée.

Tableau 8.26. Types identifiant d'objet

Nom	Référence	Description	Exemple
`oid`	tous	identifiant d'objet numérique	`564182`
`regclass`	`pg_class`	relation name	`pg_type`
`regcollation`	`pg_collation`	collation name	`"POSIX"`
`regconfig`	`pg_ts_config`	text search configuration	`english`
`regdictionary`	`pg_ts_dict`	text search dictionary	`simple`
`regnamespace`	`pg_namespace`	namespace name	`pg_catalog`
`regoper`	`pg_operator`	nom d'opérateur	`+`
`regoperator`	`pg_operator`	opérateur avec types d'arguments	`*(integer,integer)` ou `-(NONE,integer)`

Nom	Référence	Description	Exemple
regproc	pg_proc	nom de fonction	sum
regprocedure	pg_proc	fonction avec les types des arguments	sum(int4)
regrole	pg_authid	nom de rôle	smithee
regtype	pg_type	nom du type de données	integer

Tous les types alias d'OID pour des objets groupés par schéma acceptent des noms qualifiés par le schéma, et affichent des noms préfixés par un schéma si l'objet ne peut être trouvé dans le chemin de recherche courant sans être qualifié. Par exemple, `myschema.mytable` est une saisie acceptée pour `regclass` (s'il y a une telle table). Cette valeur peut être une valeur retournée comme `myschema.mytable`, ou juste `mytable`, selon le chemin de parcours des schémas courant. Les types alias `regproc` et `regoper` n'acceptent que des noms uniques en entrée (sans surcharge), si bien qu'ils sont d'un usage limité ; dans la plupart des cas, `regprocedure` et `regoperator` sont plus appropriés. Pour `regoperator`, les opérateurs unaires sont identifiés en écrivant `NONE` pour les opérandes non utilisés.

Les entrées de fonction pour ces types permettent les espaces entre tokens, et interpréteront les lettres en majuscules par des minuscules, sauf si elles sont entre guillemets ; ceci est effectué pour rendre les règles de syntaxe similaires à la façon dont les noms d'objets sont écrits en SQL. De même, les retours de fonctions utiliseront les guillements s'il est besoin que la sortie soit un identifiant SQL valide. Par exemple, l'OID d'une fonction nommée `Foo` (avec le `F` majuscule) et prenant deux arguments entiers peut être saisie comme `' "Foo" ( int, integer ) '::regprocedure`. La sortie ressemblera à `"Foo"(integer,integer)`. Les noms de fonction et les noms des types d'arguments peuvent aussi être qualifiés par un nom de schéma.

De nombreuses fonctions internes de PostgreSQL acceptent l'OID d'une table, ou un autre type d'objet de base de données, et pour des raisons de simplicité sont déclarées comme prenant un `regclass` (ou le type d'alias OID approprié). Ceci signifie qu'il n'est pas nécessaire de chercher soi-même l'OID de l'objet, mais qu'il suffit juste de saisir son nom sous sa forme littérale de chaîne. Par exemple, la fonction `nextval(regclass)` prend un OID de relation de séquence, ainsi elle peut être appelée :

```
nextval('foo')              opère sur la séquence foo
nextval('FOO')              comme ci-dessus
nextval('"Foo"')            opère sur la séquence Foo
nextval('myschema.foo')     opère sur myschema.foo
nextval('"myschema".foo')   comme ci-dessus
nextval('foo')              cherche dans le chemin de parcours des
 schémas, foo
```

> **Note**
>
> Quand vous écrivez l'argument d'une de ces fonctions sous forme de chaîne littérale sans fioritures, il devient une constante de type `regclass` (ou du type approprié). Puisqu'il n'est en fait qu'un OID, l'objet originellement identifié sera tracé même s'il est renommé après, réassigné à un schéma, etc. Cette « liaison précoce » (*early binding* dans la version originale) est en général désirée pour les références aux objets dans les valeurs par défaut des colonnes et des vues. Mais parfois, une « liaison tardive » (*late binding*) peut être souhaitée quand la référence à l'objet est résolue à l'exécution. Pour avoir un comportement de liaison tardive, forcez la constante à être stockée en constante `text` au lieu de `regclass` :
>
> ```
> nextval('foo'::text)      foo est interprété au runtime
> ```
>
> La fonction `to_regclass()` et ses sœurs peuvent aussi être utilisées pour effectuer des interprétations de paramètres à l'exécution. Voir Tableau 9.71.

Un autre exemple pratique de cas d'usage de `regclass` est de rechercher l'OID d'une table listée dans les vues `information_schema`, qui demandent l'OID de table. En tenant comptes des règles ci-dessus, la façon adéquate de faire cela est

```
SELECT table_schema, table_name,
       pg_relation_size((quote_ident(table_schema) || '.' ||
                         quote_ident(table_name))::regclass)
FROM information_schema.tables
WHERE ...
```

La fonction `quote_ident()` gère les identifiants entre guillements quand il y a besoin. La façon suivante qui apparait plus simple

```
SELECT pg_relation_size(table_name)
FROM information_schema.tables
WHERE ...
```

n'est *pas recommandée*, car elle échouera pour les tables en dehors du chemin de parcours des schémas courant ou qui ont un nom qui nécessite des guillemets.

Une propriété supplémentaire de pratiquement tous les types alias d'OID est la création de dépendances. Si une constante d'un de ces types apparaît dans une expression stockée (telle que l'expression par défaut d'une colonne ou une vue), elle crée une dépendance sur l'objet référencé. Par exemple, si une colonne a une expression par défaut `nextval('ma_seq'::regclass)`, PostgreSQL comprend que l'expression par défaut dépend de la séquence `ma_seq` ; le système ne permet alors pas la suppression de la séquence si l'expression par défaut n'est pas elle-même supprimée au préalable. La version alternative `nextval('ma_seq'::text)` ne crée pas une dépendance. `regrole` est une exception à cette propriété. Les constantes de ce type ne sont pas autorisées dans les expressions enregistrées.

Un autre type d'identifiant utilisé par le système est `xid`, ou identifiant de transaction (abrégée xact). C'est le type de données des colonnes système `xmin` et `xmax`. Les identifiants de transactions sont stockés sur 32 bits. Dans certains contextes, une variante 64-bit `xid8` est utilisée. Contrairement aux valeurs `xid`, les valeurs `xid8` sont strictement à accroissement monotone et ne peuvent être réemployées sur la durée de vie d'une instance de base de données.

Un troisième type d'identifiant utilisé par le système est `cid`, ou identifiant de commande. C'est le type de données des colonnes système `cmin` et `cmax`. Les identifiants de commandes sont aussi stockés sur 32 bits.

Le dernier type d'identifiant utilisé par le système est `tid`, ou identifiant de ligne (tuple). C'est le type de données des colonnes système `ctid`. Un identifiant de tuple est une paire (numéro de bloc, index de tuple dans le bloc) qui identifie l'emplacement physique de la ligne dans sa table.

Les colonnes système sont expliquées plus en détail dans la Section 5.5.

8.20. Type `pg_lsn`

Le type de données `pg_lsn` peut être utilisé pour stocker des données LSN (Log Sequence Number ou Numéro de Séquence de Journal), qui sont un pointeur vers une position dans les journaux de transactions. Ce type est une représentation de `XLogRecPtr` et un type système interne de PostgreSQL.

En interne, un LSN est un entier sur 64 bits, représentant une position d'octet dans le flux des journaux de transactions. Il est affiché comme deux nombres hexadécimaux allant jusqu'à 8 caractères chacun, séparés par un slash. Par exemple, `16/B374D848`. Le type `pg_lsn` gère les opérateurs de comparaison standard, comme `=` et `>`. Deux LSN peuvent être soustraits en utilisant l'opérateur `-`. Le résultat est le nombre d'octets séparant ces deux emplacements dans les journaux de transactions. De plus, un nombre d'octets peut être additionné ou soustrait à un LSN en utilisant respectivement

les opérateurs `+ (pg_lsn,numeric)` et `-(pg_lsn,numeric)`. Notez que le LSN calculé sera dans l'intervalle de type `pg_lsn`, i.e. entre `0/0` et `FFFFFFFF/FFFFFFFF`.

8.21. Pseudo-Types

Le système de types de PostgreSQL contient un certain nombre de types à usage spécial qui sont collectivement appelés des *pseudo-types*. Un pseudo-type ne peut être utilisé comme type d'une colonne de table, mais peut l'être pour déclarer un argument de fonction ou un type de résultat. Tous les pseudo-types disponibles sont utiles dans des situations où une fonction ne se contente pas d'accepter et retourner des valeurs d'un type de données SQL particulier. Le Tableau 8.27 liste les différents pseudo-types.

Tableau 8.27. Pseudo-Types

Nom	Description
`any`	Indique qu'une fonction accepte tout type de données, quel qu'il soit.
`anyelement`	Indique qu'une fonction accepte tout type de données (voir la Section 38.2.5).
`anyarray`	Indique qu'une fonction accepte tout type de tableau (voir la Section 38.2.5).
`anynonarray`	Indique que la fonction accepte tout type de données non-array (voir Section 38.2.5).
`anyenum`	Indique que la fonction accepte tout type de données enum (voir Section 38.2.5 et Section 8.7).
`anyrange`	Indique qu'une fonction accepte tout type de données intervalle (voir Section 38.2.5 et Section 8.17).
`anymultirange`	Indique qu'une fonction accepte tout type de données intervalle multiple (voir Section 38.2.5 et Section 8.17).
`anycompatible`	Indique qu'une fonction accepte tout type de données, avec une promotion automatique des arguments multiples vers un type de données commun (voir Section 38.2.5).
`anycompatiblearray`	Indique que la fonction accepte tout type de données tableau, avec une promotion automatique des arguments multiples vers un type de données commun (voir Section 38.2.5).
`anycompatiblenonarray`	Indique que la fonction accepte tout type de données autre que tableau, avec une promotion automatique des arguments multiples vers un type de données commun (voir Section 38.2.5).
`anycompatiblerange`	Indique qu'une fonction accepte tout type de données intervalle, avec une promotion automatique des arguments multiples vers un type de données commun (voir Section 38.2.5 et Section 8.17).
`anycompatiblemultirange`	Indique qu'une fonction accepte tout type de données intervalle multiple, avec une promotion automatique des arguments multiples vers un type de données commun(voir Section 38.2.5 et Section 8.17).
`cstring`	Indique qu'une fonction accepte ou retourne une chaîne de caractères C (terminée par un NULL).
`internal`	Indique qu'une fonction accepte ou retourne un type de données interne du serveur de bases de données.

Nom	Description
language_handler	Une fonction d'appel de langage procédural est déclarée retourner un language_handler.
fdw_handler	Une fonction de gestion pour le wrapper de données distantes est déclarée retourner un fdw_handler.
table_am_handler	Un gestionnaire de méthode d'accès de table est déclaré comme retournant table_am_handler.
index_am_handler	Un gestionnaire pour une méthode d'accès d'index est déclaré renvoyer index_am_handler.
tsm_handler	Un gestionnaire de méthode d'échantillonnage est déclaré comme renvoyant le type tsm_handler.
record	Identifie une fonction qui prend ou retourne un type de ligne non spécifié.
trigger	Une fonction trigger est déclarée comme retournant un type trigger.
event_trigger	Une fonction pour un trigger d'événement est déclarée comme renvoyant une donnée de type event_trigger.
pg_ddl_command	Identifie une représention de commandes DDL qui est disponible pour les triggers d'événement.
void	Indique qu'une fonction ne retourne aucune valeur.
unknown	Identifie un type non encore résolu, par exemple une chaîne de texte non décorée.

Les fonctions codées en C (incluses ou chargées dynamiquement) peuvent être déclarées comme acceptant ou retournant tout pseudo-type. Il est de la responsabilité de l'auteur de la fonction de s'assurer du bon comportement de la fonction lorsqu'un pseudo-type est utilisé comme type d'argument.

Les fonctions codées en langage procédural ne peuvent utiliser les pseudo-types que dans les limites imposées par l'implantation du langage. À ce jour, la plupart des langages procéduraux interdisent l'usage d'un pseudo-type comme argument et n'autorisent que void et record comme type de retour (plus trigger ou event_trigger lorsque la fonction est utilisée respectivement comme trigger ou triggers d'événement). Certains supportent également les fonctions polymorphes qui utilisent les pseudo-types polymorphes, qui sont décrits ci-dessus et discutés en détail dans Section 38.2.5.

Le pseudo-type internal sert à déclarer des fonctions qui ne sont appelées que par le système en interne, et non pas directement par une requête SQL. Si une fonction accepte au minimum un argument de type internal, alors elle ne peut être appelée depuis SQL. Pour préserver la sécurité du type de cette restriction, il est important de suivre la règle de codage suivante : ne jamais créer de fonction qui retourne un internal si elle n'accepte pas au moins un argument de type internal.

Chapitre 9. Fonctions et opérateurs

PostgreSQL fournit un grand nombre de fonctions et d'opérateurs pour les types de données natifs. Ce chapitre en décrit la plupart, bien que certaines fonctions spéciales apparaissent dans des sections plus adéquates du manuel. Les utilisateurs peuvent aussi définir leurs propres fonctions et opérateurs, comme décrit dans Partie V. Les méta-commandes `\df` et `\do` de psql peuvent être utilisées pour lister, respectivement, toutes les fonctions et tous les opérateurs disponibles.

La notation utilisée dans ce chapitre pour décrire les types de données des arguments et du résultat d'une fonction ou d'un opérateur ressemble à ceci

```
repeat ( text, integer ) → text
```

qui signifie que la fonction `repeat` prend un premier argument de type `text` et un deuxième argument de type `integer`. Elle renvoie un résultat de type text. La flèche droite est aussi utilisée pour indiquer le résultat d'un exemple, par exemple :

```
repeat('Pg', 4) → PgPgPgPg
```

Si vous êtes préoccupé par la portabilité, notez que la plupart des fonctions et opérateurs décrits dans ce chapitre, à l'exception des opérateurs arithmétiques et de comparaison triviaux, et de certaines fonctions indiquées explicitement, ne sont pas indiqués dans le standard SQL. Certaines sont présentes dans d'autres moteurs de bases de données. Dans certains cas, elles sont compatibles et cohérentes sur les différentes implémentations.

9.1. Opérateurs logiques

Les opérateurs logiques habituels sont disponibles :

```
boolean AND boolean → boolean
boolean OR boolean → boolean
NOT boolean → boolean
```

Le SQL utilise un système logique en trois valeurs, avec true, false et `null`, qui représente une valeur « inconnue ». Voici les tables de vérités :

a	***b***	***a* AND *b***	***a* OR *b***
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	**NOT *a***
TRUE	FALSE

a	**NOT** *a*
FALSE	TRUE
NULL	NULL

Les opérateurs `AND` et `OR` sont commutatifs, c'est-à-dire que vous pouvez intervertir les opérandes gauche et droit sans changer le résultat. (Néanmoins, il n'est pas garanti que l'opérande gauche est évaluée avant l'opérande droit. Voir Section 4.2.14 pour plus d'informations sur l'ordre d'évaluation des sous-expressions.)

9.2. Fonctions et opérateurs de comparaison

Les opérateurs de comparaison habituels sont disponibles comme indiqués dans Tableau 9.1.

Tableau 9.1. Opérateurs de comparaison

Opérateur	Description
datatype < *datatype* → `boolean`	Inférieur à
datatype > *datatype* → `boolean`	Supérieur à
datatype <= *datatype* → `boolean`	Inférieur ou égal à
datatype >= *datatype* → `boolean`	Supérieur ou égal à
datatype = *datatype* → `boolean`	Égal
datatype <> *datatype* → `boolean`	Différent
datatype != *datatype* → `boolean`	Différent

> **Note**
>
> <> est la notation du standard SQL « non égal ». != est un alias, converti en <> au tout début de l'analyse. De ce fait, il n'est pas possible d'implémenter des opérateurs != et <> qui se comporteraient différemment.

Ces opérateurs de comparaison sont disponibles pour les types de données natifs qui ont un tri naturel, incluant les types des nombres, des chaînes et des dates/heures. De plus, les tableaux, types composites et intervalles peuvent être comparés si le type de données de leurs éléments sont comparables.

Il est généralement possible de comparer les valeurs de types de données en relation ; par exemple `integer > bigint` fonctionnera. Certains cas de ce type sont implémentés directement par des opérateurs de comparaison « inter-type », mais si aucun opérateur de ce type n'est disponible, l'analyseur va forcer une conversion de type du type le plus spécifique au type le plus global., et appliquer l'opérateur de comparaison correspondant.

Comme indiqué ci-dessus, tous les opérateurs de comparaison sont des opérateurs binaires, renvoyant des valeurs de type `boolean`. De ce fait, les expressions comme `1 < 2 < 3` ne sont pas valides (parce qu'il n'existe pas d'opérateur < comparant une valeur booléenne avec `3`). Utilisez les prédicats `BETWEEN` affichés ci-dessous pour réaliser des tests sur des intervalles.

Il existe aussi des prédicats de comparaison, comme indiqués dans Tableau 9.2. Ils se comportent comme des opérateurs, mais disposent d'une syntaxe spéciale mandatée par le standard SQL.

Tableau 9.2. Prédicats de comparaison

Prédicat **Description** **Exemple(s)**
datatype `BETWEEN` *datatype* `AND` *datatype* → `boolean` Entre (les points de départ et d'arrêt sont inclus). `2 BETWEEN 1 AND 3` → `t` `2 BETWEEN 3 AND 1` → `f`
datatype `NOT BETWEEN` *datatype* `AND` *datatype* → `boolean` Pas entre (la négation de `BETWEEN`). `2 NOT BETWEEN 1 AND 3` → `f`
datatype `BETWEEN SYMMETRIC` *datatype* `AND` *datatype* → `boolean` Entre, après le tri des valeurs de départ et d'arrêt. `2 BETWEEN SYMMETRIC 3 AND 1` → `t`
datatype `NOT BETWEEN SYMMETRIC` *datatype* `AND` *datatype* → `boolean` Pas entre, après le tri des valeurs de départ et d'arrêt. `2 NOT BETWEEN SYMMETRIC 3 AND 1` → `f`
datatype `IS DISTINCT FROM` *datatype* → `boolean` Différent, en traitant les valeurs NULL comme des valeurs comparables entre elles. `1 IS DISTINCT FROM NULL` → `t` (plutôt que `NULL`) `NULL IS DISTINCT FROM NULL` → `f` (plutôt que `NULL`)
datatype `IS NOT DISTINCT FROM` *datatype* → `boolean` Égal, en traitant les valeurs NULL comme des valeurs comparables entre elles. `1 IS NOT DISTINCT FROM NULL` → `f` (plutôt que `NULL`) `NULL IS NOT DISTINCT FROM NULL` → `t` (plutôt que `NULL`)
datatype `IS NULL` → `boolean` Teste si la valeur est NULL. `1.5 IS NULL` → `f`
datatype `IS NOT NULL` → `boolean` Teste si la valeur n'est pas NULL `'null' IS NOT NULL` → `t`
datatype `ISNULL` → `boolean` Teste si la valeur est NULL (syntaxe non standard).
datatype `NOTNULL` → `boolean` Teste si la valeur n'est pas NULL (syntaxe non standard).
`boolean IS TRUE` → `boolean` Teste si l'expression booléenne vaut true. `true IS TRUE` → `t` `NULL::boolean IS TRUE` → `f` (plutôt que `NULL`)
`boolean IS NOT TRUE` → `boolean` Teste si l'expression booléenne ne vaut pas true (donc false ou NULL). `true IS NOT TRUE` → `f` `NULL::boolean IS NOT TRUE` → `t` (plutôt que `NULL`)

Prédicat **Description** **Exemple(s)**
`boolean IS FALSE → boolean` Teste si l'expression booléenne vaut false. `true IS FALSE → f` `NULL::boolean IS FALSE → f` (plutôt que `NULL`)
`boolean IS NOT FALSE → boolean` Teste si l'expression booléenne ne vaut pas false (donc soit true soit NULL). `true IS NOT FALSE → t` `NULL::boolean IS NOT FALSE → t` (plutôt que `NULL`)
`boolean IS UNKNOWN → boolean` Teste si l'expression booléenne renvoie inconnu. `true IS UNKNOWN → f` `NULL::boolean IS UNKNOWN → t` (plutôt que `NULL`)
`boolean IS NOT UNKNOWN → boolean` Teste si l'expression booléenne renvoie true ou false. `true IS NOT UNKNOWN → t` `NULL::boolean IS NOT UNKNOWN → f` (plutôt que `NULL`)

Le prédicat `BETWEEN` simplifie les tests d'intervalle :

```
a BETWEEN x AND y
```

est équivalent à :

```
a >= x AND a <= y
```

Notez que `BETWEEN` traite les valeurs de départ et d'arrêt comme inclus à l'intervalle. `BETWEEN SYMMETRIC` est identique à `BETWEEN` sauf qu'il n'y a pas d'obligation que l'argument de gauche de `AND` soit inférieur ou égale à l'argument de droite. Dans le cas contraire, les deux arguments sont automatiquement inversés pour qu'un intervalle non vide soit toujours de rigueur.

Les différentes variantes de `BETWEEN` sont implémentées dans les termes des opérateurs de comparaison standard et, de ce fait, fonctionneront pour tout type de données qui peut être comparé.

> **Note**
>
> L'utilisation de `AND` dans la syntaxe `BETWEEN` crée une ambiguité avec l'utilisation de `AND` comme opérateur logique. Pour résoudre ceci, seul un ensemble limité de types d'expression est autorisé comme deuxième argument d'une clause `BETWEEN`. Si vous avez besoin d'écrire une sous-expression plus complexe dans `BETWEEN`, ajoutez des parenthèses autour de la sous-expression.

Les opérateurs de comparaison habituels renvoient NULL (signifiant « inconnu »), et non pas true ou false, si l'argument est NULL. Par exemple, `7 = NULL` renvoie NULL, tout autant que `7 <> NULL`. Si ce comportement n'est pas adéquat, utilisez les prédicats `IS [ NOT ] DISTINCT FROM` :

```
a IS DISTINCT FROM b
a IS NOT DISTINCT FROM b
```

Pour les valeurs non NULL, `IS DISTINCT FROM` est identique à l'opérateur `<>`. Néanmoins, si les deux arguments sont NULL, il renvoie false et si un seul des deux arguments est NULL, il renvoie true. De la même façon, `IS NOT DISTINCT FROM` est identique à `=` pour les arguments non NULL, mais il renvoie true quand les deux arguments sont NULL et false quand un seul argument est NULL. De ce fait, ces prédicats agissent comme si NULL était une valeur normale, plus qu'« inconnue ».

Pour vérifier si une valeur est NULL ou non, utilisez le prédicat :

```
expression IS NULL
expression IS NOT NULL
```

ou les prédicats équivalents, mais non standards :

```
expression ISNULL
expression NOTNULL
```

N'écrivez *pas* `expression = NULL` car `NULL` n'est pas « égal à » `NULL`. (La valeur NULL représente une valeur inconnue, et il n'est pas possible de savoir si deux valeurs inconnues sont égales.)

> **Astuce**
>
> Certaines applications pourraient s'attendre que `expression = NULL` renvoient true si `expression` s'évalue à la valeur NULL. Il est hautement recommandé que ces applications soient modifiées pour se conformer au standard SQL. Néanmoins, si cela ne peut pas se faire, le paramètre transform_null_equals est disponible. S'il est activé, PostgreSQL convertira les clauses `x = NULL` en `x IS NULL`.

Si l'`expression` est une valeur ligne, alors `IS NULL` est true quand l'expression de ligne elle-même est NULL ou quand tous les champs de la ligne sont NULL, alors que `IS NOT NULL` est true quand l'expression de ligne elle-même est non NULL et que tous les champs de la ligne sont non NULL. À cause de ce comportement, `IS NULL` et `IS NOT NULL` ne renvoient pas toujours des résultats inversés pour les expressions de ligne ; en particulier, une expression de ligne qui contient des champs NULL et non NULL renverra false pour les deux tests. Dans certains cas, il pourrait être préférable d'écrire `ligne IS DISTINCT FROM NULL` ou `ligne IS NOT DISTINCT FROM NULL`, qui va simplement vérifier si la valeur de ligne globale est NULL, sans tester les champs de la ligne.

Les valeurs booléennes peuvent aussi être testées en utilisant les prédicats

```
boolean_expression IS TRUE
boolean_expression IS NOT TRUE
boolean_expression IS FALSE
boolean_expression IS NOT FALSE
boolean_expression IS UNKNOWN
boolean_expression IS NOT UNKNOWN
```

Ils renverront toujours true ou false, jamais une valeur NULL, même quand l'opérande est NULL. Un argument NULL est traité comme la valeur logique « inconnue ». Notez que `IS UNKNOWN` et `IS NOT UNKNOWN` sont identiques à `IS NULL` et `IS NOT NULL`, respectivement, sauf que l'expression en argument doit être de type booléen.

Quelques fonctions de comparaison sont aussi disponibles, comme indiquées dans Tableau 9.3.

Tableau 9.3. Fonctions de comparaison

Fonction **Description** **Exemple(s)**
`num_nonnulls ( VARIADIC "any" ) → integer` Renvoie le nombre d'arguments non NULL. `num_nonnulls(1, NULL, 2) → 2`
`num_nulls ( VARIADIC "any" ) → integer` Renvoie le nombre d'arguments NULL. `num_nulls(1, NULL, 2) → 1`

9.3. Fonctions et opérateurs mathématiques

Les opérateurs mathématiques sont fournies pour de nombreux types de données pour PostgreSQL. Pour les types sans conventions mathématiques standards (par exemple les types dates/heures), nous décrivons le comportement réel dans les sections suivantes.

Tableau 9.4 affiche les opérateurs mathématiques disponibles pour les types numériques standards. Sauf note explicite, les opérateurs affichés comme acceptant *`numeric_type`* sont disponibles pour tous les types `smallint`, `integer`, `bigint`, `numeric`, `real` et `double precision`. Les opérateurs affichés comme acceptant *`integral_type`* sont disponibles pour les types `smallint`, `integer` et `bigint`. Sauf note explicite, chaque forme d'un opérateur renvoie le même type de données que ses arguments, Les appels impliquant plusieurs types de données en argument, tel que `integer + numeric`, sont résolus en utilisant le type apparaissant en dernier dans ces listes.

Tableau 9.4. Opérateurs mathématiques

Opérateur **Description** **Exemple(s)**
`numeric_type` `+` *`numeric_type`* `→` *`numeric_type`* Addition `2 + 3 → 5`
`+` *`numeric_type`* `→` *`numeric_type`* Plus unitaire (pas d'opération) `+ 3.5 → 3.5`
`numeric_type` `-` *`numeric_type`* `→` *`numeric_type`* Soustraction `2 - 3 → -1`
`-` *`numeric_type`* `→` *`numeric_type`* Négation `- (-4) → 4`
`numeric_type` `*` *`numeric_type`* `→` *`numeric_type`* Multiplication `2 * 3 → 6`
`numeric_type` `/` *`numeric_type`* `→` *`numeric_type`* Division (pour les types entiers, la division tronque le résultat vers zéro)

Opérateur **Description** **Exemple(s)**
`5.0 / 2` → `2.5000000000000000` `5 / 2` → `2` `(-5) / 2` → `-2`
numeric_type `%` *numeric_type* → *numeric_type* Modulo (reste) ; disponible pour `smallint`, `integer`, `bigint` et `numeric` `5 % 4` → `1`
`numeric ^ numeric` → `numeric` `double precision ^ double precision` → `double precision` Exponentiel `2 ^ 3` → `8` Contrairement aux pratiques mathématiques typiques, plusieurs utilisations de `^` feront une association de gauche à droite par défaut : `2 ^ 3 ^ 3` → `512` `2 ^ (3 ^ 3)` → `134217728`
`|/ double precision` → `double precision` Racine carrée `|/ 25.0` → `5`
`||/ double precision` → `double precision` Racine cubique `||/ 64.0` → `4`
`@` *numeric_type* → *numeric_type* Valeur absolue `@ -5.0` → `5.0`
integral_type `&` *integral_type* → *integral_type* AND bit à bit `91 & 15` → `11`
integral_type `|` *integral_type* → *integral_type* OR bit à bit `32 | 3` → `35`
integral_type `#` *integral_type* → *integral_type* OR exclusif bit à bit `17 # 5` → `20`
`~` *integral_type* → *integral_type* NOT bit à bit `~1` → `-2`
integral_type `<<` `integer` → *integral_type* Décalage à gauche bit à bit `1 << 4` → `16`
integral_type `>>` `integer` → *integral_type* Décalage à droite bit à bit `8 >> 2` → `2`

Tableau 9.5 affiche les fonctions mathématiques disponibles. Beaucoup de ces fonctions sont fournies dans des formes différentes avec des types d'argument différents. Sauf note explicite, toute forme donnée d'une fonction renvoie le même type de donnée que ses arguments ; les cas inter-type sont résolus de la même façon qu'expliqué ci-dessus pour les opérateurs. Les fonctions travaillant avec des données de type `double precision` sont implémentées majoritairement grâce à la bibliothèque C du système hôte ; la précision et le comportement dans les cas limites peuvent donc varier suivant le système hôte.

Tableau 9.5. Fonctions mathématiques

Fonction **Description** **Exemple(s)**
`abs ( numeric_type ) → numeric_type` Valeur absolue `abs(-17.4) → 17.4`
`cbrt ( double precision ) → double precision` Racine cubique `cbrt(64.0) → 4`
`ceil ( numeric ) → numeric` `ceil ( double precision ) → double precision` Entier le plus proche plus grand ou égal à l'argument `ceil(42.2) → 43` `ceil(-42.8) → -42`
`ceiling ( numeric ) → numeric` `ceiling ( double precision ) → double precision` Entier le plus proche plus grand ou égal à l'argument (identique à `ceil`) `ceiling(95.3) → 96`
`degrees ( double precision ) → double precision` Convertit des radians en degrés `degrees(0.5) → 28.64788975654116`
`div ( y numeric, x numeric ) → numeric` Quotient entier de *y*/*x* (tronque vers zéro) `div(9, 4) → 2`
`exp ( numeric ) → numeric` `exp ( double precision ) → double precision` Exponentiel (`e` élevé à la puissance donnée) `exp(1.0) → 2.7182818284590452`
`factorial ( bigint ) → numeric` Factorielle `factorial(5) → 120`
`floor ( numeric ) → numeric` `floor ( double precision ) → double precision` Entier le plus proche plus petit ou égal à l'argument `floor(42.8) → 42` `floor(-42.8) → -43`

Fonction **Description** **Exemple(s)**
`gcd` (*`numeric_type`*, *`numeric_type`*) → *`numeric_type`* Plus grand commun diviseur (plus grand nombre positif qui divise les arguments en entrée sans reste) ; renvoie `0` si les deux arguments valent zéro ; disponible pour `integer`, `bigint`et `numeric` `gcd(1071, 462)` → `21`
`lcm` (*`numeric_type`*, *`numeric_type`*) → *`numeric_type`* Plus petit commun multiple (plus petit nombre strictement positif qui est un multiple entier des deux arguments en entrée) ; renvoie `0` si les deux arguments valent zéro ; disponible pour `integer`, `bigint`et `numeric` `lcm(1071, 462)` → `23562`
`ln` (`numeric`) → `numeric` `ln` (`double precision`) → `double precision` Logarithme `ln(2.0)` → `0.6931471805599453`
`log` (`numeric`) → `numeric` `log` (`double precision`) → `double precision` Logarithme base 10 `log(100)` → `2`
`log10` (`numeric`) → `numeric` `log10` (`double precision`) → `double precision` Logarithme base 10 (identique à `log`) `log10(1000)` → `3`
`log` (*`b`* `numeric`, *`x`* `numeric`) → `numeric` Logarithme de *`x`* en base *`b`* `log(2.0, 64.0)` → `6.0000000000000000`
`min_scale` (`numeric`) → `integer` Échelle minimale (nombre de chiffres décimaux en fraction) nécessaire pour représenter la valeur fournie précisément `min_scale(8.4100)` → `2`
`mod` (*`y numeric_type`*, *`x numeric_type`*) → *`numeric_type`* Reste de *`y`*/*`x`* ; disponible pour `smallint`, `integer`, `bigint` et `numeric` `mod(9, 4)` → `1`
`pi` ( ) → `double precision` Valeur approximative de ## `pi()` → `3.141592653589793`
`power` (*`a`* `numeric`, *`b`* `numeric`) → `numeric` `power` (*`a`* `double precision`, *`b`* `double precision`) → `double precision` *`a`* élevé à la puissance de *`b`* `power(9, 3)` → `729`
`radians` (`double precision`) → `double precision` Convertit des degrés en radians

Fonction **Description** **Exemple(s)**
`radians(45.0)` → `0.7853981633974483`
`round` (`numeric`) → `numeric` `round` (`double precision`) → `double precision` Arrondit à l'entier le plus proche. Pour le type `numeric`, les nombres à mi-chemin sont arrondis en s'éloignant de zéro. Pour le type `double precision`, le comportement de l'arrondi est dépendant de la plateforme, mais « l'arrondi vers le plus proche » est la règle la plus commune. `round(42.4)` → `42`
`round` (*v* `numeric`, *s* `integer`) → `numeric` Arrondit *v* à *s* positions décimales. Les nombres à mi-chemin sont arrondis en s'éloignant de zéro. `round(42.4382, 2)` → `42.44` `round(1234.56, -1)` → `1230`
`scale` (`numeric`) → `integer` Échelle de l'argument (le nombre de chiffres décimaux dans la partie fractionnelle) `scale(8.4100)` → `4`
`sign` (`numeric`) → `numeric` `sign` (`double precision`) → `double precision` Signe de l'argument (-1, 0 ou +1) `sign(-8.4)` → `-1`
`sqrt` (`numeric`) → `numeric` `sqrt` (`double precision`) → `double precision` Racine carrée `sqrt(2)` → `1.4142135623730951`
`trim_scale` (`numeric`) → `numeric` Réduit l'échelle de la valeur (nombre de chiffres décimaux fractionnels) en supprimant les zéro à la fin `trim_scale(8.4100)` → `8.41`
`trunc` (`numeric`) → `numeric` `trunc` (`double precision`) → `double precision` Tronque à l'entier (vers zéro) `trunc(42.8)` → `42` `trunc(-42.8)` → `-42`
`trunc` (*v* `numeric`, *s* `integer`) → `numeric` Tronque *v* à *s* places décimales `trunc(42.4382, 2)` → `42.43`
`width_bucket` (*operand* `numeric`, *low* `numeric`, *high* `numeric`, *count* `integer`) → `integer` `width_bucket` (*operand* `double precision`, *low* `double precision`, *high* `double precision`, *count* `integer`) → `integer` Renvoie le numéro du compartiment dans lequel *operand* serait affecté dans un histogramme ayant *count* compartiments d'égale longueur répartis entre *low* et *high*.

Fonction **Description** **Exemple(s)**
Les compartiments ont des limits basses inclusives et des limites hautes exclusives. Renvoie `0` pour un argument inférieur à *`low`*, ou *`count`*`+1` pour un argument supérieur ou égal à *`high`*. Si *`low`* > *`high`*, le comportement est inversé avec le compartiment `1` juste en dessous de *`low`*, et les limites inclusives étant maintenant du côté haut. `width_bucket(5.35, 0.024, 10.06, 5)` → `3` `width_bucket(9, 10, 0, 10)` → `2`
`width_bucket(` *`operand`* `anycompatible,` *`thresholds`* `anycompatiblearray )` → `integer` Renvoie le numéro du compartiment dans lequel *`operand`* serait affecté pour un tableau listant les limites basses inclusives des compartiments. Renvoie `0` pour un argument inférieur à la première limite basse. *`operand`* et les éléments du tableau peuvent être de tout type ayant des opérateurs de comparaison standards. Le tableau *`thresholds`* *doit être triés*, les plus petits en premier. Sinon, des résultats inattendus pourraient être obtenus. `width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[])` → `2`

Tableau 9.6 affiche les fonctions pour générer des nombres aléatoires.

Tableau 9.6. Fonctions aléatoires

Fonction **Description** **Exemple(s)**
`random ( )` → `double precision` Renvoie une valeur aléatoire comprise dans l'intervalle 0.0 <= x < 1.0 `random()` → `0.897124072839091`
`setseed (double precision )` → `void` Configure le sel pour les appels suivants à `random()` ; l'argument doit être compris entre -1.0 et 1.0, valeurs comprises. `setseed(0.12345)`

La fonction `random()` utilise un générateur de nombres pseudo aléatoires déterministique. Il est rapide mais non convenable pour des applications cryptographiques. Voir le module pgcrypto pour une alternative plus sécurisée. Si `setseed()` est appelé, la série de résultats d'appels en bouicle à `random()` dans la même session peut être répétée en ré-exécutant `setseed()` avec le même argument. Sans un appel précédent à `setseed()` dans la même session, le premier appel à `random()` obtient une graine d'une source dépendante de la plateforme de bits aléatoires.

Tableau 9.7 montre les fonctions trigonométriques disponibles. Chacune de ces fonctions vient en deux variantes, une pour mesurer les angles en radians et une pour les mesurer en degrés.

Tableau 9.7. Fonctions trigonométriques

Fonction **Description** **Exemple(s)**
`acos (double precision )` → `double precision` Cosinus inverse, résultat en radians `acos(1)` → `0`

Fonction **Description** **Exemple(s)**
`acosd ( double precision ) → double precision` Cosinus inverse, résultat en degrés `acosd(0.5) → 60`
`asin ( double precision ) → double precision` Sinus inverse, résultat en radians `asin(1) → 1.5707963267948966`
`asind ( double precision ) → double precision` Sinus inverse, résultat en degrés `asind(0.5) → 30`
`atan ( double precision ) → double precision` Tangente inverse, résultat en radians `atan(1) → 0.7853981633974483`
`atand ( double precision ) → double precision` Tangente inverse, résultat en degrés `atand(1) → 45`
`atan2 ( y double precision, x double precision ) → double precision` Tangente inverse de *y*/*x*, résultat en radians `atan2(1, 0) → 1.5707963267948966`
`atan2d ( y double precision, x double precision ) → double precision` Tangente inverse de *y*/*x*, résultat en degrés `atan2d(1, 0) → 90`
`cos ( double precision ) → double precision` Cosinus, argument en radians `cos(0) → 1`
`cosd ( double precision ) → double precision` Cosinus, argument en degrés `cosd(60) → 0.5`
`cot ( double precision ) → double precision` Cotangente, argument en radians `cot(0.5) → 1.830487721712452`
`cotd ( double precision ) → double precision` Cotangente, argument en degrés `cotd(45) → 1`
`sin ( double precision ) → double precision` Sinus, argument en radians `sin(1) → 0.8414709848078965`
`sind ( double precision ) → double precision` Sinus, argument en degrés `sind(30) → 0.5`
`tan ( double precision ) → double precision`

Fonction Description Exemple(s)
Tangente, argument en radians `tan(1)` → `1.5574077246549023`
`tand(double precision)` → `double precision` Tangente, argument en degrés `tand(45)` → `1`

> **Note**
>
> Une autre façon de travailler avec des angles mesurés en degrés est d'utiliser les fonctions de transformation d'unité `radians()` et `degrees()` montrées plus tôt. Néanmoins, utiliser les fonctions trigonométriques basées sur les degrés est préféré, car cette façon évite les erreurs d'arrondi pour les cas spéciaux comme `sind(30)`.

Tableau 9.8 affiche les fonctions hyperboliques.

Tableau 9.8. Fonctions hyperboliques

Fonction Description Exemple(s)
`sinh(double precision)` → `double precision` Sinus hyperbolique `sinh(1)` → `1.1752011936438014`
`cosh(double precision)` → `double precision` Cosinus hyperbolique `cosh(0)` → `1`
`tanh(double precision)` → `double precision` Tangente hyperbolique `tanh(1)` → `0.7615941559557649`
`asinh(double precision)` → `double precision` Sinus inverse hyperbolique `asinh(1)` → `0.881373587019543`
`acosh(double precision)` → `double precision` Cosinus inverse hyperbolique `acosh(1)` → `0`
`atanh(double precision)` → `double precision` Tangente inverse hyperbolique `atanh(0.5)` → `0.5493061443340548`

9.4. Fonctions et opérateurs sur des chaînes de caractères

Cette section décrit les fonctions et opérateurs pour l'examen et la manipulation de valeurs de type chaîne de caractères. Dans ce contexte, les chaînes incluent les valeurs de type `character`,

`character varying` et `text`. Sauf note explicite, ces fonctions et opérateurs sont déclarés comme acceptant et renvoyant le type `text`. Ils accepteront directement des arguments de type `character varying`. Les valeurs de type `character` seront converties en `text` avant que la fonction ou l'opérateur ne soit exécuté, résultant en une suppression des espaces en fin de la valeur de type `character`.

Le standard SQL définit certaines fonctions de chaîne de caractères comme utilisant des mots clés plutôt que des virgules pour séparer les arguments. Les détails sont disponibles dans Tableau 9.9. PostgreSQL fournit aussi des versions de ces fonctions qui utilisent la syntaxe d'appel de fonctions standards (voir Tableau 9.10).

> **Note**
>
> L'opérateur de concaténation (`||`) acceptera les entrées qui ne sont pas de type chaîne, tant qu'au moins une entrée est de type chaîne, comme indiqué dans Tableau 9.9. Dans d'autres cas, le fait d'ajouter une conversion explicite en `text` peut faire accepter les entrées qui ne sont pas de type chaîne.

Tableau 9.9. Fonctions et opérateurs SQL pour les chaînes de caractères

Fonction/Opérateur **Description** **Exemple(s)**
`text || text → text` Concatène deux chaînes de caractères. `'Post' || 'greSQL' → PostgreSQL`
`text || anynonarray → text` `anynonarray || text → text` Convertit l'argument qui n'est pas une chaîne de caractère en texte, puis concatène les deux chapines. (L'argument qui n'est pas une chaîne ne peut pas être un tableau car cela causerait une ambiguité avec les opérateurs `||` pour les tableaux. Si vous voulez concaténer l'équivalent texte d'un tableau, vous devez préalablement le convertir explicitement vers le type `text`.) `'Valeur: ' || 42 → Valeur: 42`
`text IS [NOT] [form] NORMALIZED → boolean` Vérifie si la chaîne est dans la forme de normalisation Unicode précisée. Le mot clé optionnel *`form`* indique la forme : `NFC` (valeur par défaut), `NFD`, `NFKC` ou `NFKD`. Cette expression peut seulement être utilisée quand l'encodage serveur est `UTF8`. Il faut noter que la vérification de la normalisation en utilisant cette expression est généralement plus rapide que de normaliser des chaînes déjà normalisées. `U&'\0061\0308bc' IS NFD NORMALIZED → t`
`bit_length(text) → integer` Renvoie le nombre de bits dans la chaîne de caractères (8 fois la valeur de `octet_length`). `bit_length('jose') → 32`
`char_length(text) → integer` `character_length(text) → integer` Renvoie le nombre de caractères dans la chaîne. `char_length('josé') → 4`
`lower(text) → text`

Fonction/Opérateur **Description** **Exemple(s)**
Convertit la chaîne en minuscule, suivant les règles de la locale de la base de données. `lower('TOM')` → `tom`
`normalize` (`text` [, *`form`*]) → `text` Convertit la chaîne vers la forme de normalisation Unicode précisée. Le mot clé optionnel *`form`* indique la forme : `NFC` (valeur par défaut), `NFD`, `NFKC` ou `NFKD`. Cette fonction peut seulement être utilisée quand l'encodage serveur est `UTF8`. `normalize(U&'\0061\0308bc', NFC)` → `U&'\00E4bc'`
`octet_length` (`text`) → `integer` Renvoie le nombre d'octets dans la chaîne. `octet_length('josé')` → `5` (if server encoding is UTF8)
`octet_length` (`character`) → `integer` Renvoie le nombre d'octets dans la chaîne. Comme cette version de la fonction accepte directement le type `character`, il ne supprimera pas les espaces en fin de chaîne. `octet_length('abc '::character(4))` → `4`
`overlay` (*`string`* `text` `PLACING` *`newsubstring`* `text` `FROM` *`start`* `integer` [`FOR` *`count`* `integer`]) → `text` Remplace la sous-chaîne de *`string`* qui commence au *`start`*-ième caractère et continue pendant *`count`* caractères avec *`newsubstring`*. Si *`count`* est omis, sa valeur par défaut est la longueur de la chaîne *`newsubstring`*. `overlay('Txxxxas' placing 'hom' from 2 for 4)` → `Thomas`
`position` (*`substring`* `text` `IN` *`string`* `text`) → `integer` Renvoie le premier index de début de la chaîne *`substring`* dans *`string`*, ou zéro si elle n'est pas présente. `position('om' in 'Thomas')` → `3`
`substring` (*`string`* `text` [`FROM` *`start`* `integer`] [`FOR` *`count`* `integer`]) → `text` Extrait la sous-chaîne de *`string`* commençant au *`start`*-ième caractère s'il est spécifié, et arrêtant après *`count`* caractère s'il est spécifié. Il est nécessaire de fournir au moins un des deux arguments, *`start`* et *`count`*. `substring('Thomas' from 2 for 3)` → `hom` `substring('Thomas' from 3)` → `omas` `substring('Thomas' for 2)` → `Th`
`substring` (*`string`* `text` `FROM` *`pattern`* `text`) → `text` Extrait la première sous-chaîne correspondant à une expression rationnelle POSIX ; voir Section 9.7.3. `substring('Thomas' from '...$')` → `mas`
`substring` (*`string`* `text` `SIMILAR` *`pattern`* `text` `ESCAPE` *`escape`* `text`) → `text` `substring` (*`string`* `text` `FROM` *`pattern`* `text` `FOR` *`escape`* `text`) → `text` Extrait la première sous-chaîne correspondant à l'expression rationnelle SQL ; voir Section 9.7.2. La première forme a été spécifiée depuis SQL:2003 ; la seconde forme n'existe que dans SQL:1999 et doit être considérée comme obsolète. `substring('Thomas' similar '%#"o_a#"_' for '#')` → `oma`
`trim` ([`LEADING` | `TRAILING` | `BOTH`] [*`characters`* `text`] `FROM` *`string`* `text`) → `text`

Fonction/Opérateur **Description** **Exemple(s)**
Supprime la chaîne la plus longue contenant seulement des caractères parmi *characters* (un espace par défaut) à partir du début, de la fin ou des deux bouts (BOTH est la valeur par défaut) de *string*. `trim(both 'xyz' from 'yxTomxx')` → `Tom`
`trim` ([`LEADING` | `TRAILING` | `BOTH`] [`FROM`] *string* `text` [, *characters* `text`]) → `text` Ceci est la syntaxe non standard pour `trim()`. `trim(both from 'yxTomxx', 'xyz')` → `Tom`
`upper` (`text`) → `text` Convertit la chaîne en majuscule, suivant les règles de la locale de la base de données. `upper('tom')` → `TOM`

Des fonctions et opérateurs supplémentaires de manipulation des chaînes de caractères sont disponibles et listées dans Tableau 9.10. Certaines sont utilisées en interne pour implémenter les fonctions de chaîne du standard SQL listées dans Tableau 9.9. Il existe aussi des opérateurs de correspondance de motif, décrits dans Section 9.7, et des opérateurs operators de recherche plein texte, décrits dans Chapitre 12.

Tableau 9.10. Autres fonctions et opérateurs pour les chaînes de caractères

Fonction / Opérateur **Description** **Exemple(s)**
`text ^@ text` → `boolean` Renvoie true si la première chaîne commence avec la deuxième chaîne (équivalent à la fonction `starts_with()`). `'alphabet' ^@ 'alph'` → `t`
`ascii` (`text`) → `integer` Renvoie le code numérique du premier caractère de l'argument. Dans l'encodage UTF8, renvoie le point code Unicode du caractère. Dans les autres encodages multioctets, l'argument doit être un caractère ASCII. `ascii('x')` → `120`
`btrim` (*string* `text` [, *characters* `text`]) → `text` Supprime la plus longue chaîne contenant seulement les caractères compris dans *characters* (un espace par défaut) au début et à la fin de *string*. `btrim('xyxtrimyyx', 'xyz')` → `trim`
`chr` (`integer`) → `text` Renvoie le caractère avec le code donné. Dans l'encodage UTF8, l'argument est traité qu'un point code Unicode. Dans les autres encodages multi-octets, l'argument doit désigner un caractère ASCII. `chr(0)` est interdit parce que les types de données texte ne peuvent pas enregistrer ce caractère. `chr(65)` → `A`
`concat` (*val1* `"any"` [, *val2* `"any"` [, ...]]) → `text` Concatène les représentations textuelles de tous les arguments. Les arguments NULL sont ignorés. `concat('abcde', 2, NULL, 22)` → `abcde222`

Fonction / Opérateur **Description** **Exemple(s)**
`concat_ws` (*`sep`* `text`, *`val1`* `"any"` [, *`val2`* `"any"` [, ...]]) → `text` Concatène tous les arguments sauf le premier, avec des séparateurs. Le premier argument est utilisé comme séparateur, et ne doit pas valoir NULL. Les autres arguments NULL sont ignorés. `concat_ws(',', 'abcde', 2, NULL, 22)` → `abcde,2,22`
`format` (*`formatstr`* `text` [, *`formatarg`* `"any"` [, ...]]) → `text` Formate les arguments suivant une chaîne de formatage ; voir Section 9.4.1. Cette fonction est similaire à la fonction C `sprintf`. `format('Hello %s, %1$s', 'World')` → `Hello World, World`
`initcap` (`text`) → `text` Convertit la première lettre de chaque mot en majuscule et le reste en minuscule. Les mots sont des séquences de caractères alpha-numériques séparées par des caractères non alpha-numériques. `initcap('hi THOMAS')` → `Hi Thomas`
`left` (*`string`* `text`, *`n`* `integer`) → `text` Renvoie les *`n`* premiers caractères de la chaîne. Si *`n`* est négatif, renvoie tous les caractères sauf les |*`n`*| derniers. `left('abcde', 2)` → `ab`
`length` (`text`) → `integer` Renvoie le nombre de caractères dans la chaîne. `length('jose')` → `4`
`lpad` (*`string`* `text`, *`length`* `integer` [, *`fill`* `text`]) → `text` Étend la chaîne *`string`* à la longueur *`length`* en ajoutant les caractères de la chaîne *`fill`* (un espace par défaut). Si la chaîne *`string`* est déjà plus longue que *`length`*, alors elle est tronquée (à partir de la droite). `lpad('hi', 5, 'xy')` → `xyxhi`
`ltrim` (*`string`* `text` [, *`characters`* `text`]) → `text` Supprime la chaîne la plus longue contenant seulement les caractères dans *`characters`* (un espace par défaut) à partir du début de *`string`*. `ltrim('zzzytest', 'xyz')` → `test`
`md5` (`text`) → `text` Calcule le hachage MD5 de l'argument, le résultat étant écrit en hexadécimal. `md5('abc')` → `900150983cd24fb0d6963f7d28e17f72`
`parse_ident` (*`qualified_identifier`* `text` [, *`strict_mode`* `boolean` `DEFAULT` `true`]) → `text[]` Divise *`qualified_identifier`* en un tableau d'identifiants, supprimant tout guillemet des identifiants individuels. Par défaut, les caractères supplémentaires après le dernier identifiant sont considérés comme une erreur ; si le second paramètre est `false`, alors ces caractères supplémentaires sont ignorés. (Ce comportement est utile pour analuser les noms d'objets comme des fonctions.) Il faut noter que cette fonction ne tronque pas les identifiants dont les noms sont trop longs. Si vous voulez cette troncature, vous pouvez convertir le résultat avec le type `name[]`. `parse_ident('"SomeSchema".someTable')` → `{SomeSchema,sometable}`

Fonction / Opérateur **Description** **Exemple(s)**
`pg_client_encoding ( ) → name` Renvoie le nom de l'encodage client actuel. `pg_client_encoding() → UTF8`
`quote_ident ( text ) → text` Renvoie la chaîne donnée correctement placée entre guillemets comme un identifiant dans une requête SQL. Les guillemets ne sont ajoutés que s'ils sont nécessaires (c'est-à-dire si la chaîne contient des caractères qui ne font pas partie de ceux autorisés habituellement pour les identifiants ou des caractères majuscules). Les guillemets dans le nom sont aussi doublés. Voir aussi Exemple 43.1. `quote_ident('Foo bar') → "Foo bar"`
`quote_literal ( text ) → text` Renvoie la chaîne donnée correctement placée entre guillemets pour être utilisé comme une valeur littérale de type chaîne dans une requête SQL. Les guillemets simples et antislashs présents dans la chaîne sont correctement doublés. Il faut noter que `quote_literal` renvoie NULL en cas d'argument NULL. Si l'argument pourrait être NULL, `quote_nullable` est généralement plus indiqué. Voir aussi Exemple 43.1. `quote_literal(E'O\'Reilly') → 'O''Reilly'`
`quote_literal ( anyelement ) → text` Convertit la valeur donnée dans le type text et ajoute des guillemets comme pour une valeur. Les guillemets simples et antislashs présents dans la chaîne sont correctement doublés. `quote_literal(42.5) → '42.5'`
`quote_nullable ( text ) → text` Renvoie la chaîne donnée convenablement placée entre guillemets pour être utilisée comme un littéral de type chaîne dans une requête SQL ; ou, si l'argument vaut NULL, renvoie `NULL`. Les guillemets simples et antislashs présents dans la chaîne sont correctement doublés. Voir aussi Exemple 43.1. `quote_nullable(NULL) → NULL`
`quote_nullable ( anyelement ) → text` Convertit la valeur donnée dans le type text puis la place entre guillemets comme une valeur littérale ; ou, si l'argument vaut NULL, renvoie `NULL`. Les guillemets simples et antislashs présents dans la chaîne sont correctement doublés. `quote_nullable(42.5) → '42.5'`
`regexp_count ( string text, pattern text [, start integer [, flags text ] ] )` `→ integer` Renvoie le nombre de fois où l'expression rationnelle POSIX *`pattern`* correspond à *`string`* ; voir Section 9.7.3. `regexp_count('123456789012', '\d\d\d', 2) → 3`
`regexp_instr ( string text, pattern text [, start integer [, N integer [, endoption integer [, flags text [, subexpr integer ] ] ] ] ] ) → integer` Renvoie la position dans *`string`* où la *`N`*-ième correspondance de l'expression rationnelle POSIX *`pattern`* survient, ou zéro s'il n'y a pas de correspondance ; voir Section 9.7.3. `regexp_instr('ABCDEF', 'c(.)(..)', 1, 1, 0, 'i') → 3` `regexp_instr('ABCDEF', 'c(.)(..)', 1, 1, 0, 'i', 2) → 5`

Fonction / Opérateur **Description** **Exemple(s)**
`regexp_like ( string text, pattern text [, flags text ] ) → boolean` Vérifie si une correspondance de l'expression rationnelle POSIX *pattern* survient dans *string* ; voir Section 9.7.3. `regexp_like('Hello World', 'world$', 'i') → t`
`regexp_match ( string text, pattern text [, flags text ] ) → text[]` Renvoie les sous-chaînes dans la première correspondance de l'expression rationnelle POSIX *pattern* avec *string* ; voir Section 9.7.3. `regexp_match('foobarbequebaz', '(bar)(beque)') → {bar,beque}`
`regexp_matches ( string text, pattern text [, flags text ] ) → setof text[]` Renvoie les sous-chaînes dans la première correspondance de l'expression rationnelle POSIX *pattern* avec *string*, ou sous-chaînes parmi toutes les correspondances si l'option g est utilisée ; voir Section 9.7.3. `regexp_matches('foobarbequebaz', 'ba.', 'g') →` `{bar}` `{baz}`
`regexp_replace ( string text, pattern text, replacement text [, start integer ] [, flags text ] ) → text` Remplace la sous-chaîne qui est la première correspondance de l'expression rationnelle POSIX *pattern*, ou toutes les correspondances si l'option g est utilisée ; voir Section 9.7.3. `regexp_replace('Thomas', '.[mN]a.', 'M') → ThM`
`regexp_replace ( string text, pattern text, replacement text, start integer, N integer [, flags text ] ) → text` Remplace la sous-chaîne qui est la *N*-ième correspondance de l'expression rationnelle POSIX *pattern*, ou toutes les correspondances si *N* vaut zéro ; voir Section 9.7.3. `regexp_replace('Thomas', '.', 'X', 3, 2) → ThoXas`
`regexp_split_to_array ( string text, pattern text [, flags text ] ) → text[]` Divise *string* en utilisant une expression rationnelle POSIX comme délimiteur, produisant un tableau de résultats ; voir Section 9.7.3. `regexp_split_to_array('hello world', '\s+') → {hello,world}`
`regexp_split_to_table ( string text, pattern text [, flags text ] ) → setof text` Divise *string* en utilisant une expression rationnelle POSIX comme délimiteur, produisant un tableau de résultats ; voir Section 9.7.3. `regexp_split_to_table('hello world', '\s+') →` `hello` `world`
`regexp_substr ( string text, pattern text [, start integer [, N integer [, flags text [, subexpr integer ] ] ] ] ) → text`

Fonction / Opérateur **Description** **Exemple(s)**
Renvoie la sous-chaîne de *string* qui correspond à la *N*-ième occurrence de l'expression rationnelle POSIX *pattern*, ou NULL s'il n'y a pas de correspondance ; voir Section 9.7.3. `regexp_substr('ABCDEF', 'c(.)(..)', 1, 1, 'i')` → `CDEF` `regexp_substr('ABCDEF', 'c(.)(..)', 1, 1, 'i', 2)` → `EF`
`repeat` (*string* `text`, *number* `integer`) → `text` Répète *string* le nombre de fois spécifié par *number*. `repeat('Pg', 4)` → `PgPgPgPg`
`replace` (*string* `text`, *from* `text`, *to* `text`) → `text` Remplace toutes les occurrences dans *string* de la sous-chaîne *from* avec la sous-chaîne *to*. `replace('abcdefabcdef', 'cd', 'XX')` → `abXXefabXXef`
`reverse` (`text`) → `text` Inverse l'ordre des caractères dans la chaîne. `reverse('abcde')` → `edcba`
`right` (*string* `text`, *n* `integer`)) → `text` Renvoie les *n* derniers caractères dans la chaîne, ou, si *n* est négatif, renvoie tous les caractères sauf les |*n*| premiers caractères. `right('abcde', 2)` → `de`
`rpad` (*string* `text`, *length* `integer` [, *fill* `text`])) → `text` Étend *string* à la longueur *length* en ajoutant les caractères *fill* (un espace par défaut). Si *string* est déjà plus long que *length*, alors elle est tronquée. `rpad('hi', 5, 'xy')` → `hixyx`
`rtrim` (*string* `text` [, *characters* `text`]) → `text` Supprime la plus longue chaîne contenant seulement les caractères compris dans *characters* (un espace par défaut) à partir de la fin de *string*. `rtrim('testxxzx', 'xyz')` → `test`
`split_part` (*string* `text`, *delimiter* `text`, *n* `integer`) → `text` Divise *string* à chaque occurrence de *delimiter* et renvoie le *n*-ième champ (en comptant à partir de 1), ou quand *n* est négatif, renvoie le |*n*|-ième champ depuis la fin. `split_part('abc~@~def~@~ghi', '~@~', 2)` → `def` `split_part('abc,def,ghi,jkl', ',', -2)` → `ghi`
`starts_with` (*string* `text`, *prefix* `text`) → `boolean` Renvoie true si *string* commence avec *prefix*. `starts_with('alphabet', 'alph')` → `t`
`string_to_array` (*string* `text`, *delimiter* `text` [, *null_string* `text`]) → `text[]` Divise *string* à chaque occurrence de *delimiter* et retourne les champs résultants dans un tableau de type `text`. Si *delimiter* vaut NULL, chaque caractère dans *string* deviendra un élément à part du tableau final. Si *delimiter* est une chaîne vide, alors la chaîne *string* est traitée comme un champ entier. Si le paramètre *null_string* est renseigné et ne vaut pas NULL, les champs vérifiant ce paramètre seront remplacés par NULL. Voir aussi `array_to_string`. `string_to_array('xx~~yy~~zz', '~~', 'yy')` → `{xx,NULL,zz}`

Fonction / Opérateur
Description
Exemple(s)

`string_to_table` (*`string`* `text`, *`delimiter`* `text` [, *`null_string`* `text`]) → `setof text`

Divise *`string`* à chaque occurrence de *`delimiter`* et retourne les champs résultants dans un ensemble de lignes de type `text`. Si *`delimiter`* vaut `NULL`, chaque caractère dans *`string`* deviendra une ligne séparée du résultat. Si *`delimiter`* est une chaîne vide, alors la chaîne *`string`* est traitée comme un champ entier. Si le paramètre *`null_string`* est renseigné et ne vaut pas `NULL`, les champs vérifiant ce paramètre seront remplacés par `NULL`.

`string_to_table('xx~^~yy~^~zz', '~^~', 'yy')` →

```
xx
NULL
zz
```

`strpos` (*`string`* `text`, *`substring`* `text`) → `integer`

Renvoie la première position du *`substring`* indiqué à l'intérieur de *`string`*, ou zéro s'il n'est pas présent. (Identique à `position(`*`substring`* `in` *`string`*`)`, mais notez l'ordre inverse des arguments.)

`strpos('high', 'ig')` → `2`

`substr` (*`string`* `text`, *`start`* `integer` [, *`count`* `integer`]) → `text`

Extrait la sous-chaîne de *`string`* commençant au *`start`*-ième caractère et s'étendant sur *`count`* caractères si ce dernier est spécifié. (Identique à `substring(`*`string`* `from` *`start`* `for` *`count`*`)`.)

`substr('alphabet', 3)` → `phabet`

`substr('alphabet', 3, 2)` → `ph`

`to_ascii` (*`string`* `text`) → `text`

`to_ascii` (*`string`* `text`, *`encoding`* `name`) → `text`

`to_ascii` (*`string`* `text`, *`encoding`* `integer`) → `text`

Convertit *`string`* en ASCII à partir d'un autre encodage, qui peut être identifié par nom ou numéro. Si *`encoding`* est omis, l'encodage de la base est utilisé (ce qui, en réalité, est le seul cas utile). La conversion consiste principalement en la suppression des accents. La conversion est seulement supportée à partir des encodages `LATIN1`, `LATIN2`, `LATIN9` et `WIN1250`. (Voir le module unaccent pour une autre solution, plus flexible.)

`to_ascii('Karél')` → `Karel`

`to_hex` (`integer`) → `text`

`to_hex` (`bigint`) → `text`

Convertit le nombre en sa représentation hexadécimale équivalente.

`to_hex(2147483647)` → `7fffffff`

`translate` (*`string`* `text`, *`from`* `text`, *`to`* `text`) → `text`

Remplace chaque caractère de *`string`* qui correspond à un caractère dans l'ensemble *`from`* avec le caractère correspondant dans l'ensemble *`to`*. Si *`from`* est plus long que *`to`*, les occurrences des caractères supplémentaires dans *`from`* sont supprimées.

`translate('12345', '143', 'ax')` → `a2x5`

`unistr` (`text`) → `text`

Fonction / Opérateur Description Exemple(s)
Évalue les caractères Unicode échappés dans l'argument. Les caractères Unicode peuvent être spécifiés comme `\`*`XXXX`* (hexadécimal à 4 chiffres), `\+`*`XXXXXX`* (hexadécimal à 6 chiffres), `\u`*`XXXX`* (hexadécimal à 4 chiffres), ou `\U`*`XXXXXXXX`* (hexadécimal à 6 chiffres). Pour spécifier un antislash, écrire deux antislashs. Tous les autres caractères sont conservés littéralement. Si l'encodage du serveur n'est pas UTF-8, le code Unicode identifié par une de ces séquences échappées est converti dans l'encodage actuel du serveur ; une erreur est retournée quand la conversion n'est pas possible. Cette fonction founit une alternative (non standard) aux constantes de type chaîne avec des échappements en Unicode (voir Section 4.1.2.3). `unistr('d\0061t\+000061')` → `data` `unistr('d\u0061t\U00000061')` → `data`

Les fonctions `concat`, `concat_ws` et `format` ont un nombre d'arguments variables, donc il est possible de passer les valeurs à concaténer ou à formater sous la forme d'un tableau marqué avec le mot-clé `VARIADIC` (voir Section 38.5.6). Les éléments du tableau sont traités comme s'ils étaient des arguments séparés dans la fonction. Si l'argument tableau est NULL, `concat` et `concat_ws` renvoient NULL, mais `format` traite un NULL comme un tableau à zéro élément.

Voir aussi la fonction d'agrégat `string_agg` in Section 9.21, et les fonctions de conversions entre des chaînes de caractères et le type `bytea` dans Tableau 9.13.

9.4.1. `format`

La fonction `format` produit une sortie formatée suivant une chaîne de formatage, dans un style similaire à la fonction C `sprintf`.

```
format(formatstr text [, formatarg "any" [, ...] ])
```

`formatstr` est une chaîne de formatage qui indique comment le résultat doit être formaté. Le texte de la chaîne de formatage est copié directement dans le résultat, à l'exception des *jokers de format*. Les jokers de format agissent comme des espaces réservés dans la chaîne définissant comment les arguments de la fonction doivent être formatés et insérés dans le résultat. Chaque argument *`formatarg`* est converti en texte suivant les règles d'affichage habituel pour son type de données, puis formaté et inséré dans la chaîne résultante suivant le joker de format.

Les jokers de format commencent par un caractère % et ont la forme suivante

```
%[position][flags][width]type
```

où les champs composants sont :

`position` (optionnel)

Une chaîne de la forme *`n`*$ où *`n`* est l'index de l'argument à afficher. L'index 1 correspond au premier argument après *`formatstr`*. Si *`position`* est omis, la valeur par défaut est d'utiliser le prochain argument en séquence.

`flags` (optionnel)

Des options supplémentaires contrôlant comme la sortie du joker est formatée. Actuellement, le seul drapeau supporté est un signe moins (`-`) qui causera l'alignement à gauche de la valeur du champ. Ceci n'a d'effet que si *`width`* est précisé.

width (optionnel)

Indique le nombre *minimum* de caractères à utiliser pour afficher la sortie du joker. La sortie est remplie à gauche ou à droite (suivant la présence du drapeau `-`) avec autant d'espaces que nécessaire pour remplir la largeur indiquée. Une largeur trop petite ne cause pas la trincature de la sortie, mais est tout simplement ignorée. La largeur pourrait être indiquée en utilisant soit un entier positif, soit une astérisque (`*`) pour utiliser le prochain argument de la fonction soit une chaîne de la forme `*`*n*`$` pour utiliser le *n*-ième argument de la fonction comme information de largeur.

Si la largeur vient d'un argument de la fonction, cet argument est consommé avant l'argument utilisé pour la valeur du joker de format. Si l'argument de largeur est négatif, le résultat est un alignement à gauche (comme si le drapeau `-` avait été précisé) à l'intérieur du champ de longueur `abs(`*width*`)`.

type (requis)

Le type de conversion de format à utiliser pour produire le résultat du joker de format. Les types suivants sont supportés :

- `s` formate ma valeur en argument comme une simple chaîne. Une valeur NULL est traitée comme une chaîne vide.
- `I` traite la valeur en argument comme un identifiant SQL, la plaçant entre guillemets doubles si nécessaire. Une valeur NULL est traitée comme une erreur (équivalent à `quote_ident`).
- `L` traite la valeur en argument comme un littérale SQL. Une valeur NULL est affichée comme la chaîne `NULL`, sans guillemets (équivalent à `quote_nullable`).

En plus des jokers de format décrits ci-dessus, la séquence spéciale `%%` est utilisable pour ajouter un caractère `%`.

Voici quelques exemples des conversions basiques de format :

```
SELECT format('Hello %s', 'World');
Résultat : Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
Résultat : Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O
\'Reilly');
Résultat : INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program
 Files');
Résultat : INSERT INTO locations VALUES('C:\Program Files')
```

Voici quelques exemples d'utilisation du champ *width* et du drapeau `-` :

```
SELECT format('|%10s|', 'foo');
Résultat : |       foo|

SELECT format('|%-10s|', 'foo');
Résultat : |foo       |

SELECT format('|%*s|', 10, 'foo');
Résultat : |       foo|
```

```
SELECT format('|%*s|', -10, 'foo');
Résultat : |foo       |

SELECT format('|%-*s|', 10, 'foo');
Résultat : |foo       |

SELECT format('|%-*s|', -10, 'foo');
Résultat : |foo       |
```

Ces exemples montrent l'utilisation du champ *position* :

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
Résultat : Testing three, two, one

SELECT format('|%*2$s|', 'foo', 10, 'bar');
Résultat : |       bar|

SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
Résultat : |       foo|
```

Contrairement à la fonction C standard `sprintf`, la fonction `format` de PostgreSQL autorise les jokers de format sans et avec des champs *position* mixés dans la même champs de formatage. Un joker de format sans champ *position* utilise toujours le prochain argument après le dernier argument consommé. De plus, la fonction `format` ne requiert pas que tous les arguments de la fonction soient utilisés dans la chaîne de format. Par exemple :

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
Résultat : Testing three, two, three
```

Les jokers de format `%I` et `%L` sont particulièrement utiles pour construire en toute sécurité des requêtes SQL dynamiques. Voir Exemple 43.1.

9.5. Fonctions et opérateurs pour les chaînes binaires

Cette section décrit les fonctions et opérateurs pour examiner et manipuler des chaînes binaires, c'est-à-dire des valeurs de type `bytea`. Beaucoup sont équivalentes, dans le but et dans la syntaxe, aux fonctions de chaînes de caractères décrites dans la section précédente.

Le standard SQL définit certaines fonctions de chaînes de caractères utilisant des mots-clés plutôt que des virgules, pour séparer des arguments. Les détails sont dans Tableau 9.11. PostgreSQL fournit aussi des versions de ces fonctions utilisant la syntaxe d'appel de fonctions standards (voir Tableau 9.12).

Tableau 9.11. Fonctions et opérateurs SQL pour des chaînes binaires

Fonction/Opérateur **Description** **Exemple(s)**
`bytea || bytea → bytea` Concatène deux chaînes binaires.

Fonction/Opérateur **Description** **Exemple(s)**
`'\x123456'::bytea || '\x789a00bcde'::bytea` → `\x123456789a00bcde`
`bit_length ( bytea )` → `integer` Renvoie le nombre de bits dans la chaîne binaire (8 fois le résultat de `octet_length`). `bit_length('\x123456'::bytea)` → `24`
`octet_length ( bytea )` → `integer` Renvoie le nombre d'octets dans la chaîne binaire. `octet_length('\x123456'::bytea)` → `3`
`overlay ( bytes bytea PLACING newsubstring bytea FROM start integer [ FOR count integer ] )` → `bytea` Remplace la sous-chaîne de `bytes` commençant au `start`-ième octet et s'étendant sur `count` octets avec `newsubstring`. Si `count` est omis, sa valeur par défaut est la taille de `newsubstring`. `overlay('\x1234567890'::bytea placing '\002\003'::bytea from 2 for 3)` → `\x12020390`
`position ( substring bytea IN bytes bytea )` → `integer` Renvoie la première position de début de `substring` dans `bytes`, ou zéro s'il n'est pas présent. `position('\x5678'::bytea in '\x1234567890'::bytea)` → `3`
`substring ( bytes bytea [ FROM start integer ] [ FOR count integer ] )` → `bytea` Extrait la sous-chaîne de `bytes` commençant au `start`-ième octet s'il est indiqué, et s'arrêtant après `count` octets s'il est indiqué. Il est nécessaire de fournir au moins soit `start` soit `count`. `substring('\x1234567890'::bytea from 3 for 2)` → `\x5678`
`trim ( [ LEADING | TRAILING | BOTH ] bytesremoved bytea FROM bytes bytea )` → `bytea` Supprime la plus longue chaîne contenant seulement les octets apparaissant dans `bytesremoved` à partir du début, de la fin ou des deux de `bytes`. `trim('\x9012'::bytea from '\x1234567890'::bytea)` → `\x345678`
`trim ( [ LEADING | TRAILING | BOTH ] [ FROM ] bytes bytea, bytesremoved bytea )` → `bytea` Ceci est une syntaxe non standard pour `trim()`. `trim(both from '\x1234567890'::bytea, '\x9012'::bytea)` → `\x345678`

Des fonctions supplémentaires de manipulation de chaîne binaire sont disponibles et listées dans Tableau 9.12. Certaines d'entre elles sont utilisées en interne pour implémenter les fonctions de chaînes du standard SQL listées dans Tableau 9.11.

Tableau 9.12. Autres fonctions de chaînes binaires

Fonction **Description** **Exemple(s)**
`bit_count ( bytes bytea )` → `bigint`

Fonction **Description** **Exemple(s)**
Renvoie le nombre de bits initialisés dans une chaîne de bits (également appelée « popcount »). `bit_count('\x1234567890'::bytea)` → `15`
`btrim` (*`bytes`* `bytea`, *`bytesremoved`* `bytea`) → `bytea` Supprime la chaîne la plus longue contenant seulement des octets apparaissant dans *`bytesremoved`* à partir du début et à la fin de *`bytes`*. `btrim('\x1234567890'::bytea, '\x9012'::bytea)` → `\x345678`
`get_bit` (*`bytes`* `bytea`, *`n`* `bigint`) → `integer` Extrait le n-ième bit de la chaîne binaire. `get_bit('\x1234567890'::bytea, 30)` → `1`
`get_byte` (*`bytes`* `bytea`, *`n`* `integer`) → `integer` Extrait le n-ième octet de la chaîne binaire. `get_byte('\x1234567890'::bytea, 4)` → `144`
`length` (`bytea`) → `integer` Renvoie le nombre d'octets de la chaîne binaire. `length('\x1234567890'::bytea)` → `5`
`length` (*`bytes`* `bytea`, *`encoding`* `name`) → `integer` Renvoie le nombre de caractères de la chaîne binaire, en supposant qu'il s'agit de texte dans l'encodage indiqué par *`encoding`*. `length('jose'::bytea, 'UTF8')` → `4`
`ltrim` (*`bytes`* `bytea`, *`bytesremoved`* `bytea`) → `bytea` Supprime la plus longue chaîne contenant seulement les octets apparaissant dans *`bytesremoved`* depuis le début de *`bytes`*. `ltrim('\x1234567890'::bytea, '\x9012'::bytea)` → `\x34567890`
`md5` (`bytea`) → `text` Calcule le hachage MD5 dans la chaîne binaire, le résultat étant écrit en hexadécimal. `md5('Th\000omas'::bytea)` → `8ab2d3c9689aaf18b4958c334c82d8b1`
`rtrim` (*`bytes`* `bytea`, *`bytesremoved`* `bytea`) → `bytea` Supprime la plus longue chaîne contenant seulement les octets apparaissant dans *`bytesremoved`* depuis la fin de *`bytes`*. `rtrim('\x1234567890'::bytea, '\x9012'::bytea)` → `\x12345678`
`set_bit` (*`bytes`* `bytea`, *`n`* `bigint`, *`newvalue`* `integer`) → `bytea` Initialise le n-ième bit dans la chaîne binaire à la valeur *`newvalue`*. `set_bit('\x1234567890'::bytea, 30, 0)` → `\x1234563890`
`set_byte` (*`bytes`* `bytea`, *`n`* `integer`, *`newvalue`* `integer`) → `bytea` Initialise le n-ième octet dans la chaîne binaire à la valeur *`newvalue`*. `set_byte('\x1234567890'::bytea, 4, 64)` → `\x1234567840`
`sha224` (`bytea`) → `bytea` Calcule le hachage SHA-224 de la chaîne binaire. `sha224('abc'::bytea)` → `\x23097d223405d8228642a477bda2` `55b32aadbce4bda0b3f7e36c9da7`
`sha256` (`bytea`) → `bytea`

Fonction **Description** **Exemple(s)**
Calcule le hachage SHA-256 de la chaîne binaire. `sha256('abc'::bytea)` → `\xba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad`
`sha384` (`bytea`) → `bytea` Calcule le hachage SHA-384 de la chaîne binaire. `sha384('abc'::bytea)` → `\xcb00753f45a35e8bb5a03d699ac65007272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7`
`sha512` (`bytea`) → `bytea` Calcule le hachage SHA-512 de la chaîne binaire. `sha512('abc'::bytea)` → `\xddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f`
`substr` (*`bytes`* `bytea`, *`start`* `integer` [, *`count`* `integer`]) → `bytea` Extrait la sous-chaîne de *`bytes`* commençant au *`start`*-ième octet, et s'étendant sur *`count`* octets si ce dernier est indiqué. (Identique à `substring(`*`bytes`* `from` *`start`* `for` *`count`*`)`.) `substr('\x1234567890'::bytea, 3, 2)` → `\x5678`

Les fonctions `get_byte` et `set_byte` numérotent le premier octet d'une chaîne binaire comme l'octet 0. Les fonctions `get_bit` et `set_bit` numérotent les bits à partir de la droite dans chaque octet. Par exemple, le bit 0 est le bit le moins significatif du premier octet, et le bit 15 est le bit le plus significatif du deuxième octet.

Pour des raisons historiques, la fonction `md5` renvoie une valeur encodée en hexadécimal de type `text` alors que les fonctions SHA-2 renvoient le type `bytea`. Utilisez les fonctions `encode` et `decode` pour convertir entre les deux. Par exemple, écrivez `encode(sha256('abc'), 'hex')` pour obtenir une représentation textuelle encodée en hexadécimal ou `decode(md5('abc'), 'hex')` pour obtenir une valeur de type `bytea`.

Les fonctions de conversion de chaîne entre des ensembles de caractères différents (encodage), et pour représenter des données binaires arbitraires dans leur forme textuelle, sont affichées dans Tableau 9.13. Pour ces fonctions, un argument ou un résultat de type `text` est exprimé dans l'encodage par défaut de la base de données alors que les arguments ou résultats de type `bytea` sont dans un encodage nommé par un autre argument.

Tableau 9.13. Fonctions de conversion texte/binaire

Fonction **Description** **Exemple(s)**
`convert` (*`bytes`* `bytea`, *`src_encoding`* `name`, *`dest_encoding`* `name`) → `bytea` Convertit une chaîne binaire représentant du texte dans l'encodage *`src_encoding`* en une chaîne binaire dans l'encodage *`dest_encoding`* (voir Section 24.3.4 pour les conversions disponibles). `convert('text_in_utf8', 'UTF8', 'LATIN1')` → `\x746578745f696e5f75746638`
`convert_from` (*`bytes`* `bytea`, *`src_encoding`* `name`) → `text`

Fonction **Description** **Exemple(s)**
Convertit une chaîne binaire représentant du texte dans l'encodage *src_encoding* vers le type text dans l'encodage de la base de données (voir Section 24.3.4 pour les conversions disponibles). `convert_from('text_in_utf8', 'UTF8')` → `text_in_utf8`
`convert_to` (*string* `text`, *dest_encoding* `name`) → `bytea` Convertit une chaîne de type text (dans l'encodage de la base) en une chaîne binaire encodée dans l'encodage *dest_encoding* (voir Section 24.3.4 pour les conversions disponibles). `convert_to('some_text', 'UTF8')` → `\x736f6d655f74657874`
`encode` (*bytes* `bytea`, *format* `text`) → `text` Encode les données binaires dans leur représentation textuelle ; les valeurs *format* disponibles sont : base64, escape, hex. `encode('123\000\001', 'base64')` → `MTIzAAE=`
`decode` (*string* `text`, *format* `text`) → `bytea` Décode les données binaires à partir d'une représentation textuelle ; les valeurs supportées de *format* sont les mêmes que pour encode. `decode('MTIzAAE=', 'base64')` → `\x3132330001`

Les fonctions encode et decode supportent les formats textuels suivants :

base64

Le format base64 est celui de la RFC 2045 Section 6.8[1]. D'après la RFC, les lignes encodées sont réparties sur 76 caractères. Néanmoins, à la place du marqueur de fin de ligne CRLF MIME, seul un caractère nouvelle ligne est utilisée pour une fin de ligne. La fonction decode ignore les caractères de retour chariot, de nouvelle ligne, l'espace et la tabulation. Sinon une erreur est levée quand decode se voit fourni des données base64 invalides -- ceci incluant quand le remplissage en fin est incorrect.

escape

Le format escape convertit les octets zéro et les octets avec le bit de poids fort configuré en séquences d'échappement octales (*nnn*), et il double les antislashs. Les autres valeurs d'octet sont représentées littéralement. La fonction decode lèvera une erreur si un antislash n'est pas suivi soit par un deuxième antislash ou par trois chiffres octals. Il accepte les autres valeurs sans changement.

hex

Le format hex représente chacun 4 bits de données sous la forme d'un chiffre hexadécimal, de 0 à f, en écrivant le chiffre de haut niveau de chaque octet en premier. La fonction encode affiche les chiffres hexadécimaux a-f en minuscule. Comme l'unité la plus petite des données est de 8 bits, un nombre pair de caractères est renvoyé par encode. La fonction decode accepte les caractères a-f en minuscule comme en majuscule. Une erreur est levée quand decode se voit fourni des données hexadécimales invalides -- ceci incluant un nombre impair de caractères.

Voir aussi la fonction d'agrégat string_agg dans Section 9.21 et les fonctions sur les Large Objects dans Section 35.4.

[1] https://datatracker.ietf.org/doc/html/rfc2045#section-6.8

9.6. Fonctions et opérateurs pour les chaînes de bit

Cette section décrit les fonctions et opérateurs pour examiner et manipuler les chaînes de bit, c'est-à-dire des valeurs de type `bit` et `bit varying`. (Alors que seul le type `bit` est mentionné dans ces tables, les valeurs de type `bit varying` peuvent être utilisées de façon interchangeable.) Les chaînes de bit supportent les opérateurs de comparaison habituels indiqués dans Tableau 9.1, ainsi que les opérateurs montrés dans Tableau 9.14.

Tableau 9.14. Opérateurs pour les chaînes de bit

Opérateur **Description** **Exemple(s)**
`bit || bit → bit` Concaténation `B'10001' || B'011' → 10001011`
`bit & bit → bit` AND bit à bit (les arguments doivent être de même longueur) `B'10001' & B'01101' → 00001`
`bit | bit → bit` OR bit à bit (les arguments doivent être de même longueur) `B'10001' | B'01101' → 11101`
`bit # bit → bit` OR exclusif bit à bit (les arguments doivent être de même longueur) `B'10001' # B'01101' → 11100`
`~ bit → bit` NOT bit à bit `~ B'10001' → 01110`
`bit << integer → bit` Décalage à gauche bit à bit (la longueur de la chaîne est préservée) `B'10001' << 3 → 01000`
`bit >> integer → bit` Décalage à droite bit à bit (la longueur de la chaîne est préservée) `B'10001' >> 2 → 00100`

Certaines des fonctions disponibles pour les chaînes binaires sont aussi disponibles pour les chaînes de bit, comme indiquées dans Tableau 9.15.

Tableau 9.15. Fonctions pour les chaînes de bit

Fonction **Description** **Exemple(s)**
`bit_count ( bit ) → bigint` Retourne le nombre de bits initialisés dans la chaîne de bits (aussi appelée « popcount »). `bit_count(B'10111') → 4`
`bit_length ( bit ) → integer`

Fonction **Description** **Exemple(s)**
Renvoie le nombre de bits d'une chaîne de bit. `bit_length(B'10111')` → `5`
`length` (`bit`) → `integer` Renvoie le nombre de bits d'une chaîne de bit. `length(B'10111')` → `5`
`octet_length` (`bit`) → `integer` Renvoie le nombre d'octets d'une chaîne de bit. `octet_length(B'1011111011')` → `2`
`overlay` (*`bits`* `bit PLACING` *`newsubstring`* `bit FROM` *`start`* `integer` [`FOR` *`count`* `integer`]) → `bit` Remplace la sous-chaîne de *`bits`* commençant au *`start`*-ième bit et s'étendant sur *`count`* bits avec *`newsubstring`*. Si *`count`* est omis, sa valeur par défaut est la longueur de *`newsubstring`*. `overlay(B'01010101010101010' placing B'11111' from 2 for 3)` → `0111110101010101010`
`position` (*`substring`* `bit IN` *`bits`* `bit`) → `integer` Renvoie la position de départ de *`substring`* dans *`bits`*, ou zéro s'il n'est pas présent. `position(B'010' in B'000001101011')` → `8`
`substring` (*`bits`* `bit` [`FROM` *`start`* `integer`] [`FOR` *`count`* `integer`]) → `bit` Extrait la sous-chaîne de *`bits`* commençant au *`start`*-ième bit s'il est précisé, et s'arrêtant après *`count`* bits s'il est précisé. Il est requis qu'au moins soit *`start`* soit *`count`* soient renseignés. `substring(B'110010111111' from 3 for 2)` → `00`
`get_bit` (*`bits`* `bit`, *`n`* `integer`) → `integer` Extrait le *`n`*-ième bit de la chaîne de bits. Le premier bit (le plus à gauche) est le bit 0. `get_bit(B'101010101010101010', 6)` → `1`
`set_bit` (*`bits`* `bit`, *`n`* `integer`, *`newvalue`* `integer`) → `bit` Initialise le *`n`*-ième bit dans la chaîne de bit avec *`newvalue`*. Le premier bit (le plus à gauche) est le bit 0. `set_bit(B'101010101010101010', 6, 0)` → `101010001010101010`

De plus, il est possible de convertir les valeurs intégrales à partir ou vers le type `bit`. Convertir un entier en `bit(n)` copie les `n` bits les plus à droite. Convertir un entier en une chaîne de bits, dont la largeur est plus importante que l'entier lui-même, l'étendra sur la gauche. Quelques exemples :

```
44::bit(10)                    0000101100
44::bit(3)                     100
cast(-44 as bit(12))           111111010100
'1110'::bit(4)::integer        14
```

Il faut noter que la conversion vers simplement « bit » signifie une conversion vers `bit(1)`, et ne renverra donc que le bit le moins significatif de l'entier.

9.7. Correspondance de motif

Il existe trois approches séparées concernant la correspondance de motif fournies par PostgreSQL : l'opérateur SQL `LIKE` habituel, l'opérateur `SIMILAR TO` plus récent (ajouté dans SQL:1999), et les expressions rationnelles style POSIX. En dehors des opérateurs basiques du type « est-ce que cette chaîne correspond à ce motif ? », des fonctions sont disponibles pour extraire ou remplacer des sous-chaînes et pour diviser une chaîne aux emplacements correspondant.

> **Astuce**
>
> Si vous avez des besoins sur la correspondance de motif qui vont au dela de ceci, pensez à écrire une fonction en Perl ou Tcl.

> **Attention**
>
> Bien que la plupart des recherches par expression rationnelles peut être exécutée très rapidement, des expressions rationnelles peuvent être conçues de telle façon qu'elles récupèrent des quantités arbitraires de temps et de mémoire pour le traitement. Faites attention si vous acceptez des expressions rationnelles à partir de sources hostiles. Si vous devez le faire, il est conseillé d'imposer une durée limite d'exécution pour la requête.
>
> Les recherches utilisant des motifs `SIMILAR TO` ont les mêmes problèmes de sécurité car `SIMILAR TO` fournit un grand nombre de fonctionnalités identiques à celles des expressions rationnelles style POSIX.
>
> Les recherches `LIKE`, étant plus simples que les deux autres options, sont plus sûres à utiliser avec des sources potentiellement hostiles.

Les opérateurs de correspondance de motifs des trois types ne supportent pas les collations non déterministes. Si nécessaire, appliquez une collation différente à l'expression pour contourner cette limitation.

9.7.1. `LIKE`

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

L'expression `LIKE` renvoie true si la chaîne *`string`* correspond au *`pattern`* fourni. (Comme attendu, l'expression `NOT LIKE` renvoie false si `LIKE` renvoie true, et vice versa. Une expression équivalente est `NOT (`*`string`* `LIKE` *`pattern`*`)`.)

Si *`pattern`* ne contient pas de signe pourcent ou de tiret bas, alors le motif représente seulement la chaîne elle-même ; dans ce cas, `LIKE` agit comme l'opérateur d'égalité. Un tiret bas (_) dans *`pattern`* établit une correspondance avec n'importe quel caractère, mais un seul ; un signe pourcent (%) établit une correspondance avec une séquence de zéro ou plusieurs caractères.

Quelques exemples :

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

La correspondance de motif de LIKE couvre toujours la chaîne complète. Du coup, s'il est préféré de réaliser une correspondance sur une séquences n'importe où dans une chaîne, le motif doit commencer et finir avec un signe pourcent.

Pour faire une correspondance avec un tiret bas ou un signe pourcent, le caractère respectif dans *pattern* doit être précédé d'un caractère d'échappement. Le caractère d'échappement par défaut est l'antislash mais il est possible d'en sélectionner un autre en utilisant la clause ESCAPE. Pour faire une correspondance avec l'un de ces deux caractères, il faut écrire deux fois les caractères d'échappement.

> **Note**
>
> Si vous avez désactivé standard_conforming_strings, tout antislash écrit dans les constantes de chaîne aura besoin d'être doublé. Voir Section 4.1.2.1 pour plus d'informations.

Il est aussi possible de sélectionner un autre caractère d'échappement en écrivant ESCAPE ''. Ceci a pour conséquence de désactiver le mécanisme d'échappement, ce qui rend impossible de désactiver la signification spéciale des signes tiret bas et pourcent dans le motif.

D'après le standard SQL, omettre ESCAPE signifie qu'il n'y a pas de caractère d'échappement (plutôt que d'avoir antislash comme valeur par défaut), et une valeur ESCAPE de longueur zéro est interdite. Le comportement de PostgreSQL à ce niveau est donc contraire au standard.

Le mot clé ILIKE peut être utilisé à la place de LIKE pour rendre la correspondance insensible à la casse d'après la locale active. Ce mot clé n'est pas défini dans le standard SQL, il s'agit d'une extension de PostgreSQL.

L'opérateur ~~ est équivalent à LIKE, et ~~* correspond à ILIKE. Il existe aussi les opérateurs !~~ et !~~* qui représentent respectivement NOT LIKE et NOT ILIKE. Tous ces opérateurs sont spécifiques à PostgreSQL. Vous pourriez voir les noms de ces opérateurs dans le résultat de la commande EXPLAIN et dans des emplacements similaires, car l'analyseur traduit en fait LIKE et les autres en ces opérateurs.

Les phrases LIKE, ILIKE, NOT LIKE et NOT ILIKE sont généralement traitées comme des opérateurs dans la syntaxe PostgreSQL ; par exemple, elles peuvent être utilisées dans des constructions du style *expression operateur* ANY (*sous-requete*), bien qu'une clause ESCAPE ne puisse pas être inclus ici. Dans certains cas obscurs, il pourrait être nécessaire d' utiliser les noms d'opérateur sous-jacent à la place.

De plus, voir l'opérateur commence-avec ^@ et la fonction correspondante starts_with, qui sont utiles dans les cas où une simple correspondance avec le début d'une chaîne est nécessaire.

9.7.2. Expressions rationnelles pour SIMILAR TO

```
string SIMILAR TO pattern [ESCAPE escape-character]
string NOT SIMILAR TO pattern [ESCAPE escape-character]
```

L'opérateur SIMILAR TO renvoie true ou false suivant que le motif correspond ou non à la chaîne donnée. C'est similaire à LIKE, sauf qu'il interprète le motif en utilisant la définition du standard SQL d'une expression rationnelle. Les expressions rationnelles SQL sont un mixe étrange entre la notation LIKE et la notation d'expression rationnelle POSIX.

Tout comme LIKE, l'opérateur SIMILAR TO réussit seulement si son motif correspond à la chaîne entière ; c'est contraire au comportement commun des expressions rationnelles où le motif peut

correspondre à toute partie de la chaîne. Encore une fois comme `LIKE`, `SIMILAR TO` utilise _ et % comme caractère joker dénotant respectivement tout caractère simple et toute (ils sont comparables à `.` et `.*` dans les expressions rationnelles POSIX).

En plus de ces possibilités empruntées à `LIKE`, `SIMILAR TO` accepte les méta-caractères de correspondance de motif empruntés aux expressions rationnelles POSIX :

- `|` dénote le choix (entre deux alternatives).
- `*` dénote la répétition de l'élément précédent zéro ou plusieurs fois.
- `+` dénote la répétition de l'élément précédent une ou plusieurs fois.
- `?` dénote la répétition de l'élément précédent zéro ou une fois.
- `{`*m*`}` dénote la répétition de l'élément précédent exactement *m* fois.
- `{`*m*`,}` dénote la répétition de l'élément précédent *m* fois ou plus.
- `{`*m*`,`*n*`}` dénote la répétition de l'élément précédent au moins *m* fois mais plus que que *n* fois.
- Les parenthèses `()` peuvent être utilisées pour grouper des éléments en un seul élément logique.
- Une expression entre crochets `[...]` indique une classe de caractères, tout comme dans les expressions rationnelles POSIX.

Notez que le point (`.`) n'est pas un méta-caractère pour `SIMILAR TO`.

Comme avec `LIKE`, un antislash désactive la signification spéciale d'un de ces méta-caractères. Un caractère d'échappement différent peut être indiqué avec `ESCAPE`, et la possibilité d'échappement peut être désactivé avec `ESCAPE ''`.

D'après le standard SQL, omettre `ESCAPE` signifie qu'il n'y a pas de caractère d'échappement (plutôt que d'avoir comme valeur par défaut un antislash), et une valeur `ESCAPE` de longueur nulle est interdite. Le comportement de PostgreSQL dans ce cas est contraire au standard.

Une autre extension non standard est que faire suivre le caractère d'échappement d'une lettre ou d'un chiffre fournit un accès aux séquences d'échappement définies par les expressions rationnelles POSIX ; voir Tableau 9.20, Tableau 9.21 et Tableau 9.22 ci-dessous.

Quelques exemples :

```
'abc' SIMILAR TO 'abc'          true
'abc' SIMILAR TO 'a'            false
'abc' SIMILAR TO '%(b|d)%'      true
'abc' SIMILAR TO '(b|c)%'       false
'-abc-' SIMILAR TO '%\mabc\M%'  true
'xabcy' SIMILAR TO '%\mabc\M%'  false
```

La fonction `substring` avec trois paramètres fournit une extraction d'une sous-chaîne qui correspond à un motif d'expression rationnelle SQL. La fonction peut être écrite d'après la syntaxe du standard SQL :

```
substring(string similar pattern escape escape-character)
```

ou en utilisant la syntaxe maintenant obsolète SQL:1999 :

```
substring(string from pattern for escape-character)
```

ou comme une fonction simple avec trois arguments :

```
substring(string, pattern, escape-character)
```

Comme avec `SIMILAR TO`, le motif spécifié doit correspondre à la chaîne de données entière. Dans le cas contraire, la fonction échoue et renvoie NULL. Pour indiquer la partie du motif pour laquelle la sous-chaîne de données correspondante est intéressante, le motif doit contenir deux occurrences du caractère d'échappement suivi par un guillemet double (`"`). Le texte correspondant à la portion du motif entre ces séparateurs est renvoyé quand la correspondance est réussie.

Les séparateurs échappement-guillemet-double divisent en fait le motif de `substring` en trois expressions rationnelles indépendantes ; par exemple, une barre verticale (`|`) dans une des trois sections affecte seulement uniquement cette section. De plus, la première et la troisième des expressions rationnelles sont définies pour correspondre à la plus petite quantité de texte, pas à la plus grande, quand il y a une ambiguité sur quelle quantité de données il y a correspondance avec le motif. (Dans la langue POSIX, on dit que la première et la troisième expressions rationnelles sont restreintes pour ne pas être gourmandes, *non-greedy* en anglais.)

En tant qu'extension au standard SQL, PostgreSQL permet qu'il y ait juste un caractère pour l'échappement des guillemets doubles, auquel cas la troisième expression rationnelle est comprise comme vide ; ou sans séparateur, auquel cas la première et la troisième expressions rationnelles sont comprises comme vides.

Quelques exemples, avec `#"` délimitant la chaîne en retour :

```
substring('foobar' similar '%#"o_b#"%' escape '#')   oob
substring('foobar' similar '#"o_b#"%' escape '#')    NULL
```

9.7.3. Expressions rationnelles POSIX

Tableau 9.16 liste les opérateurs disponibles pour faire de la correspondance de motif en utilisant les expressions rationnelles.

Tableau 9.16. Opérateurs de correspondance d'expressions rationnelles

Opérateur **Description** **Exemple(s)**
`text ~ text → boolean` La chaîne correspond à l'expression rationnelle, casse comprise `'thomas' ~ 't.*ma' → t`
`text ~* text → boolean` La chaîne correspond à l'expression rationnelle, sans prise en compte de la casse `'thomas' ~* 'T.*ma' → t`
`text !~ text → boolean` La chaîne ne correspond pas à l'expression rationnelle, casse comprise `'thomas' !~ 't.*max' → t`

Opérateur **Description** **Exemple(s)**
`text !~* text → boolean` La chaîne ne correspond pas à l'expression rationnelle, sans prise en compte de la casse `'thomas' !~* 'T.*ma' → f`

Les expressions rationnelles POSIX fournit un moyen bien plus puissant pour la correspondance de motif que les opérateurs `LIKE` et `SIMILAR TO`. Un grand nombre d'outils Unix comme `egrep`, `sed` ou `awk` utilise un langage de correspondance de motif similaire à celui décrit ici.

Une expression rationnelle est une séquence de caractères qui est une définition courte d'un ensemble de chaînes (un *ensemble rationnel*). Une chaîne est dite correspondre à une expression rationnelle si elle est un membre de l'ensemble rationnel décrit par l'expression rationnelle. Tout comme avec `LIKE`, les caractères du motif correspondent exactement aux caractères de la chaîne sauf dans le cas de caractères spéciaux du langage d'expression rationnelle -- mais les expressions rationnelles utilisent différents caractères spéciaux comme le fait `LIKE`. Contrairement aux motifs `LIKE`, une expression rationnelle est autorisée à établir une correspondance n'importe où dans une chaîne, sauf si l'expression rationnelle est explicitement ancrée au début ou à la fin de la chaîne.

Quelques exemples :

```
+'abcd' ~ 'bc'     true
+'abcd' ~ 'a.c'    true  --  le point correspond à tout caractère
+'abcd' ~ 'a.*d'   true  --  * répète l'élément de motif précédent
+'abcd' ~ '(b|x)'  true  --  | signifie OU, pour le groupe entre
 parenthèses
+'abcd' ~ '^a'     true  --  ^ attache au début de la chaîne
+'abcd' ~ '^(b|c)' false  --  correspondrait sauf pour l'attache
```

Le langage de motif POSIX est décrit avec bien plus de détails ci-dessous.

La fonction `substring` avec deux arguments, `substring(`*`string`* `from` *`pattern`*`)`, propose l'extraction d'une sous-chaîne correspondant à un motif d'expression rationnelle POSIX. Elle renvoie NULL s'il n'y a aucune correspondance, et renvoie la première portion du texte correspondant au motif. Mais si le motif contient des parenthèses, la portion du texte correspondant à la première sous-expression entre parenthèses (celle où la parenthèse gauche arrive en premier) est renvoyée. Vous pouvez placer les parenthèses autour de l'expression entière si vous voulez utiliser des parenthèses dans le motif sans déclencher cette exception. Si vous avez besoin des parenthèses dans le motif avant la sous-expression que vous voulez extraire, voir les parenthèses sans capture décrites ci-dessous.

Quelques exemples :

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

La fonction `regexp_count` compte le nombre de places où un motif d'expression rationnelle POSIX correspond à une chaîne. Elle a la syntaxe `regexp_count(`*`string`*`,` *`pattern`* `[, ` *`start`* `[, ` *`flags`* `]])`. *`pattern`* est recherché dans *`string`*, normalement à partir du début de la chaîne mais si le paramètre *`start`* est fourni, alors la recherche commence à partir de cette position. Le paramètre *`flags`* est une chaîne de texte optionnelle contenant zéro ou plus d'options sous la forme de lettres individuelles, modifiant le comportement de la fonction. Par exemple, inclure `i` dans *`flags`* force une correspondance sans sensibilité à la casse. Les options acceptées sont décrites dans Tableau 9.24.

Voici quelques exemples :

```
regexp_count('ABCABCAXYaxy', 'A.')          3
regexp_count('ABCABCAXYaxy', 'A.', 1, 'i')  4
```

La fonction `regexp_instr` renvoie la position de départ ou de fin de la *`N`*-ième correspondance d'un motif d'expression rationnelle POSIX dans une chaîne, ou zéro s'il n'y a pas de correspondance. La syntaxe est la suivante : `regexp_instr(`*`string`*, *`pattern`* [, *`start`* [, *`N`* [, *`endoption`* [, *`flags`* [, *`subexpr`*]]]]]). *`pattern`* est recherché dans *`string`*, normalement à partir du début de la chaîne, mais si le paramètre *`start`* est fourni, alors la recherche commence à partir de cette position. Si *`N`* est spécifié, alors la *`N`*-ième correspondance du motif est recherché, sinon ce sera la première. Si le paramètre *`endoption`* est omis ou vaut zéro, la fonction renvoie la position du premier caractère de la correspondance. Sinon, *`endoption`* doit valoir un, et la fonction renvoie la position du caractère suivant la correspondance. Le paramètre *`flags`* est une chaîne de texte optionnelle contenant zéro ou plus options, sous la forme de lettres individuelles, changeant le comportement de la fonction. Les options acceptées sont décrites dans Tableau 9.24. Pour un motif contenant des sous-expressions entre parenthèses, *`subexpr`* est un entier indiquant la sous-expression d'intérêt : le résultat identifie la position de la sous-chaîne correspondant à cette sous-expression. Les sous-expressions sont numérotées dans l'ordre de leur première parenthèse. Quand *`subexpr`* est omis ou vaut zéro, le résultat identifie la position de la correspondance totale plutôt que celle des sous-expressions entre parenthèses.

Voici quelques exemples :

```
regexp_instr('number of your street, town zip, FR', '[^,]+', 1, 2)
                                    23
regexp_instr('ABCDEFGHI', '(c..)(...)', 1, 1, 0, 'i', 2)
                                    6
```

La fonction `regexp_like` vérifie si une correspondance d'un motif d'expression rationnelle POSIX survient dans une chaîne, renvoyant un booléen true ou false. Voici sa syntaxe : `regexp_like(`*`string`*, *`pattern`* `regexp_replace(`*`source`*, *`pattern`*, *`replacement`* [, *`flags`*]). Le paramètre *`flags`* est une chaîne de texte optionnelle contenant zéro ou plus options, indiquées par des lettres individuelles qui modifient le comportement de la fonction. Les options acceptées sont décrites dans Tableau 9.24. Cette fonction a les mêmes résultats que l'opérateur `~` si aucune option n'est indiquée. Si seule l'option `i` est indiquée, elle a les mêmes résultats que l'opérateur `~*`.

Quelques exemples :

```
regexp_like('Hello World', 'world')       false
regexp_like('Hello World', 'world', 'i')  true
```

La fonction `regexp_match` renvoie un tableau texte des sous-chaînes capturées résultant de la première correspondance au motif d'une expression rationnelle POSIX avec une chaîne. Elle a la syntaxe `regexp_match(`*`string`*, *`pattern`* [, *`flags`*]). S'il n'y a aucune correspondance, le résultat est `NULL`. Si une correspondance est trouvée et que *`pattern`* ne contient aucune sous-expression entre parenthèses, alors le résultat est un tableau texte à un seul élément contenant la sous-chaîne correspondance au motif entier. Si une correspondance est trouvée et que *`pattern`* contient des sous-expressions entre parenthèses, alors le résultat est un tableau texte où le *`n`*-ième élément est la sous-chaîne correspondant à la *`n`*-ième sous-expression entre parenthèses de *`pattern`* (sans compter les parenthèses « sans capture » ; voir ci-dessous pour les détails). Le paramètre *`flags`* est une chaîne de texte optionnel contenant zéro ou plusieurs drapeaux de lettres individuelles qui changent le comportement de la fonction. Les drapeaux acceptés sont décrit dans Tableau 9.24.

Quelques exemples :

```
SELECT regexp_match('foobarbequebaz', 'bar.*que');
 regexp_match
--------------
 {barbeque}
(1 row)

SELECT regexp_match('foobarbequebaz', '(bar)(beque)');
 regexp_match
--------------
 {bar,beque}
(1 row)
```

> **Astuce**
>
> Dans le cas commun où vous voulez seulement la sous-chaîne complète correspondante ou `NULL` s'il n'y a pas de correspondance, la meilleure solution revient à utiliser `regexp_substr()`. Néanmoins, `regexp_substr()` existe seulement dans PostgreSQL version 15 et les versions ultérieures. Si vous utilisez une version plus ancienne, vous pouvez extraire le premier élément du résultat de `regexp_match()`, par exemple :
>
> ```
> SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];
>  regexp_match
> --------------
>  barbeque
> (1 row)
> ```

La fonction `regexp_matches` renvoie un ensemble de tableaux de texte des sous-chaînes correspondantes résultant d'une correspondance d'un motif d'expression rationnelle POSIX à une chaîne. Elle a la même syntaxe que `regexp_match`. Cette fonction ne renvoie aucune ligne s'il n'y a pas de correspondance, une ligne s'il y a une correspondance et que le drapeau `g` n'a pas été utilisé, ou *`N`* lignes s'il y a *`N`* correspondances et que le drapeau `g` a été utilisé. Chaque ligne renvoyée est un tableau texte contenant la sous-chaîne entière correspondante ou les sous-chaînes correspondant aux sous-expressions entre parenthèses de *`pattern`*, comme décrit ci-dessus pour `regexp_match`. `regexp_matches` accepte tous les drapeaux discutés dans Tableau 9.24, plus le drapeau `g` qui requiert le renvoi de toutes les correspondances, et non pas seulement de la première.

Quelques exemples :

```
SELECT regexp_matches('foo', 'not there');
 regexp_matches
----------------
(0 rows)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)
(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)
```

> **Astuce**
>
> Dans la plupart des cas, `regexp_matches()` devrait être utilisé avec le drapeau `g` car, si vous voulez seulement la première correspondance, il est plus simple et plus efficace d'utiliser `regexp_match()`. Néanmoins, `regexp_match()` existe seulement à partir de la version 10 de PostgreSQL. Sur les anciennes versions, une astuce connue est de placer un appel à `regexp_matches()` dans une sous-requête, par exemple :
>
> ```
> SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)'))
>  FROM tab;
> ```
>
> Ceci produit un tableau de texte s'il y a une correspondance ou `NULL` dans le cas contraire, comme le ferait `regexp_match()`. Sans la sous-requête, cette requête ne produirait aucun résultat pour les lignes de la table sans correspondance, ce qui n'est pas le comportement désiré habituellement.

La fonction `regexp_replace` fournit la substitution par un nouveau texte pour les sous-chaînes qui correspondent aux motifs d'expression rationnelle POSIX. Elle a cette syntaxe : `regexp_replace(`*`source`*`, `*`pattern`*`, `*`replacement`* [, *`start`* [, *`N`*]] [, *`flags`*]). (Notez que that *`N`* ne peut être indiqué que si *`start`* l'est, mais *`flags`* peut être donné dans tous les cas.) La chaîne *`source`* est renvoyée non modifiée s'il n'y a pas de correspondance avec *`pattern`*. S'il existe une correspondance, la chaîne *`source`* est renvoyée avec la chaîne *`replacement`* substituant la sous-chaîne qui a établi la correspondance. La chaîne *`replacement`* peut contenir `\`*`n`*, où *`n`* vaut de 1 à 9, pour indiquer que la sous-chaîne source correspondant à la *`n`*-ième sous-expression entre parenthèses du motif doit être insérée. Il peut aussi contenir `\&` pour indiquer que la sous-chaîne correspondant au motif entier doit être insérée. Écrire `\\` si vous avez besoin d'un antislash litéral dans le texte de remplacement. *`pattern`* est recherché dans *`string`*, normalement à partir du début de la chaîne mais si le paramètre *`start`* est fourni, alors la recherche commence à cette position de la chaîne. Par défaut, seule la première occurrence du motif est remplacée. Si *`N`* est indiqué et est plus grand que zéro, alors la *`N`*-ième correspondance du motif est remplacée. Si l'option `g` est utilisé ou si *`N`* est indiqué et vaut zéro, alors toutes les correspondances sur ou après la position *`start`* sont remplacées. (L'option `g` est ignorée quand *`N`* est spécifiée.) Le paramètre *`flags`* est une chaîne de texte optionnelle contenu zéro ou plus options, indiquées par des lettres individuelles, modifiant le comportement de la fonction. Les options acceptées (sauf `g`) sont décrites dans Tableau 9.24.

Voici quelques exemples :

```
regexp_replace('foobarbaz', 'b..', 'X')
                                    fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                    fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
                                    fooXarYXazY
regexp_replace('A PostgreSQL function', 'a|e|i|o|u', 'X', 1, 0,
 'i')
                                    X PXstgrXSQL fXnctXXn
regexp_replace('A PostgreSQL function', 'a|e|i|o|u', 'X', 1, 3,
 'i')
                                    A PostgrXSQL function
```

La fonction `regexp_split_to_table` divise une chaîne utilisant un motif d'expression rationnelle POSIX comme délimiteur. Elle a la syntaxe `regexp_split_to_table(`*`string`*`, `*`pattern`* [, *`flags`*]). S'il n'y a aucune correspondance avec *`pattern`*, la fonction renvoie *`string`*. S'il y a au moins une correspondance, pour chaque correspondance, elle renvoie le texte à partir de la fin

de la dernière correspondance (ou du début de la chaîne) jusqu'au début de la correspondance. Quand il n'y a plus de correspondance, elle renvoie le texte de la fin de la dernière correspondance à la fin de la chaîne. Le paramètre *`flags`* est une chaîne de texte optionnelle contenant zéro ou plusieurs drapeaux de lettres individuelles qui modifient le comportement de la fonction. `regexp_split_to_table` accepte les drapeaux décrits dans Tableau 9.24.

La fonction `regexp_split_to_array` se comporte de la même façon que `regexp_split_to_table`, sauf que `regexp_split_to_array` renvoie son résultat comme un tableau de `text`. Elle a la syntaxe `regexp_split_to_array(`*`string`*`, `*`pattern`*` [, `*`flags`*` ])`. Les paramètres sont les mêmes que pour `regexp_split_to_table`.

Quelques exemples :

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps
 over the lazy dog', '\s+') AS foo;
  foo
-------
 the
 quick
 brown
 fox
 jumps
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumps over the
 lazy dog', '\s+');
              regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumps,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*')
 AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
 k
 b
 r
 o
 w
 n
 f
 o
 x
(16 rows)
```

Comme le démontre le dernier exemple, les fonctions de division d'expressions rationnelles ignorent les correspondances de longueur zéro survenant au début et à la fin de la chaîne ou immédiatement après la correspondance précédente. Ceci est contraire à la définition stricte de la correspondance d'expression rationnelle telle qu'elle est implémentée par les autres fonctions regexp, mais c'est généralement le comportement le plus intéressant en pratique. Les autres logiciels tels que Perl utilisent des définitions similaires.

La fonction `regexp_substr` renvoie la sous-chaîne correspondant à un motif d'expression rationnelle POSIX, ou `NULL` s'il n'y a pas de correspondances. Voici sa syntaxe : `regexp_substr(`*`string`*`, `*`pattern`*` [, `*`start`*` [, `*`N`*` [, `*`flags`*` [, `*`subexpr`*` ]]]])`. *`pattern`* est recherché dans *`string`*, normalement à partir du début de la chaîne mais si le paramètre *`start`* est fourni, alors la recherche commence à partir de cette position. Si *`N`* est indiqué, alors la *`N`*-ième correspondance du motif est renvoyé, sinon ce sera la première correspondance. Le paramètre *`flags`* est une chaîne de texte optionnelle contenant zéro ou plusieurs options, indiquées chacune par une lettre indivuelle, changeant le comportement de la fonction. Les options acceptées sont décrites dans Tableau 9.24. Pour un motif contenant des sous-expressions entre parenthèses, *`subexpr`* est un entier désignant la sous-expression d'intérêt : le résultat est la sous-chaîne correspondant à cette sous-expression. Les sous-expressions sont numérotées dans l'ordre de leur première parenthèse. Quand *`subexpr`* est omis ou vaut zéro, le résultat est la correspondance complète quelque soit les sous-expressions entre parenthèses.

Voici quelques exemples :

```
regexp_substr('number of your street, town zip, FR', '[^,]+', 1, 2)
                                    town zip
regexp_substr('ABCDEFGHI', '(c..)(...)', 1, 1, 'i', 2)
                                   FGH
```

9.7.3.1. Détails sur les expressions rationnelles

Les expressions rationnelles de PostgreSQL sont implémentées en utilisant un paquet logiciel écrit par Henry Spencer. La plupart des descriptions d'expressions rationnelles ci-dessous est copiée verbatim de son manuel.

Les expressions rationnelles (connus sous l'acronyme RE), tel qu'elles sont définies dans POSIX 1003.2, viennent sous deux formes : les RE *étendues* ou ERE (en gros celles de `egrep`), et les RE *basiques* ou BRE (en gros celles de `ed`). PostgreSQL accepte les deux formes, et implémente aussi certaines extensions qui ne font pas partie du standard POSIC, mais sont devenues largement utilisées grâce à leur disponibilité dans les langages de programmation tels que Perl et Tcl. Les RE utilisant ces extensions de POSIX sont appelées des RE *avancées* ou ARE dans cette documentation. Les ARE sont pratiquement un surensemble des ERE, mais les BRE ont des incompatibilités de notation (et sont bien plus limitées). Nous décrivons tout d'abord les formes ARE et ERE, en notant les fonctionnalités qui s'appliquent uniquement aux ARE, puis nous décrivons comment les BRE diffèrent.

> **Note**
>
> PostgreSQL présume toujours au départ qu'une expression rationnelle suit les règles ARE. Néanmoins, les règles ERE plus limitées ou les règles BRE peuvent être choisies en ajoutant une *option intégrée* au motif RE, comme décrit dans Section 9.7.3.4. Ceci peut être utile pour la compatibilité avec les applications qui s'attendent à un respect strict des règles POSIX 1003.2.

Une expression rationnelle est définie comme une ou plusieurs *branches*, séparées par `|`. Elle correspond à tout ce qui correspond à une des branches.

Une branche est zéro ou plus *atomes quantifiés* ou *contraintes*, concaténés. Elle établie une correspondance pour le premier, puis pour le second, etc ; une branche vide établit une correspondance à une chaîne vide.

Un atome quantifié est un *atome* potentiellement suivi par un *quantifieur* simple. Sans quantifieur, il peut établir une correspondance pour l'atome. Avec un quantifieur, il peut établir un certain nombre de correspondances à l'atome Un *atom* peut être une des possibilités indiquées dans Tableau 9.17. Les quantifieurs possibles et leur signification sont montrés dans Tableau 9.18.

Une *contrainte* correspond à une chaîne vide, mais correspond seulement quand certaines conditions spécifiques sont rencontrées. Une contrainte peut être utilisée là où un atome peut être utilisé, sauf qu'il ne peut pas être suivi d'un quantifieur. Les contraintes simples sont montrées dans Tableau 9.19 ; d'autres contraintes sont décrites après.

Tableau 9.17. Atomes d'expression rationnelle

Atome	Description
(*re*)	(où *re* est toute expression rationnelle) établit une correspondance pour *re*, avec une correspondance notée pour une capture possible
(?:*re*)	comme ci-dessus, mais la correspondance n'est pas notée pour une capture (un ensemble de parenthèses « sans capture ») (ARE seulement)
.	correspond à tout caractère individuel
[*chars*]	une *expression entre crochets*, correspondant à un des caractères parmi *chars* (voir Section 9.7.3.2 pour plus de détails)
k	(où *k* est un caractère non alpha-numérique) correspond à ce caractère pris comme un caractère ordinaire, par exemple \\ correspond à un caractère antislash
c	où *c* est un caractère alpha-numérique (potentiellement suivi par d'autres caractères) est un *échappement*, voir Section 9.7.3.3 (ARE seulement ; pour les ERE et BRE, ceci correspond à *c*)
{	quand suivi par un caractère autre qu'un chiffre, correspond au caractère { ; quand suivi d'un chiffre, c'est le début d'une *limite* (voir ci-dessous)
x	où *x* est un simple caractère sans signification particulière, correspond à ce caractère

Un RE ne peut pas se terminer avec un antislash (\).

> **Note**
>
> Si vous avez désactivé standard_conforming_strings, tout antislash que vous écrivez dans des constantes de chaînes de caractères devra être doublé. Voir Section 4.1.2.1 pour plus d'informations.

Tableau 9.18. Quantifieur pour expression rationnelle

Quantifieur	Correspondance
*	une séquence de 0 ou plus correspondances de l'atome

Quantifieur	Correspondance
+	une séquence de 1 ou plus correspondances de l'atome
?	une séquence de 0 ou 1 correspondances de l'atome
{*m*}	une séquence de exactement *m* correspondances de l'atome
{*m*,}	une séquence de *m* ou plus correspondances de l'atome
{*m*,*n*}	une séquence de *m* à *n* correspondances (nombre inclus) de l'atome ; *m* ne peut pas dépasser *n*
*?	version non gourmande de *
+?	version non gourmande de +
??	version non gourmande de ?
{*m*}?	version non gourmande de {*m*}
{*m*,}?	version non gourmande de {*m*,}
{*m*,*n*}?	version non gourmande de {*m*,*n*}

Les formes utilisant {...} sont connues comme des *limites*. Les nombres *m* et *n* d'une limite sont des entiers décimaux non signés avec des valeurs autorisées allant de 0 à 255, valeurs comprises.

Des quantifieurs non gourmands (*Non-greedy* en VO), disponibles uniquement dans les ARE, correspondent aux mêmes capacités que leur version gourmande correspondante (*greedy*), mais préfèrent le plus petit nombre plutôt que le plus grand nombre de correspondantes. Voir Section 9.7.3.5 pour plus de détails.

> **Note**
>
> Un quantifieur ne peut pas suivre immédiatement un autre quantifieur, par exemple ** est invalide. Un quantifieur ne peut pas commencer une expression ou une sous-expression, ou suivre ^ ou |.

Tableau 9.19. Contraintes d'expression rationnelle

Contrainte	Description
^	correspond au début de la chaîne
$	correspond à la fin de la chaîne
(?=*re*)	*recherche en avance positive* correspond à tout point où une sous-chaîne correspondant à *re* commence (ARE seulement)
(?!*re*)	*recherche en avance négative* correspond à tout point où aucune sous-chaîne correspondant à *re* commence (ARE seulement)
(?<=*re*)	*recherche en arrière positive* correspond à tout point où une sous-chaîne correspondant à *re* termine (ARE seulement)
(?<!*re*)	*recherche en arrière négative* correspond à tout point où aucune sous-chaîne correspondant à *re* termine (ARE seulement)

Les contraintes de recherche en avant et de recherche en arrière ne peuvent pas contenir des *références arrières* (voir Section 9.7.3.3), et toutes les parenthèses comprises en elles sont considérées comme non capturantes.

9.7.3.2. Expressions entre crochets

Une *expression entre crochets* est une liste de caractères compris dans `[]`. Cela établit habituellement une correspondance avec tout caractère seul à partir de la liste (mais voir ci-dessous). Si la liste commence avec `^`, cela établit une correspondance avec tout caractère seul ne faisant *pas* parti de la liste. Si deux caractères de la liste sont séparés par un caractère `-`, c'est un raccourci pour l'intervalle complet de caractères entre ces deux caractères (en les incluant) dans la séquence, par exemple `[0-9]` en ASCII correspond à tout chiffre décimal. Il est illégal pour deux intervalles de partager la même limite finale, par exemple `a-c-e`. Les intervalles sont particulièrement dépendants de la collation, donc les programmes portables devraient éviter de se baser sur eux.

Pour inclure un caractère `]` littéral dans la liste, faites-en le premier caractère (après `^`, si nécessaire). Pour inclure un `-` littéral, faites-en le premier ou dernier caractère, ou le deuxième point final d'un intervalle. Pour utiliser un `-` littéral comme premier point final d'un intervalle, placez le entre `[.` et `.]` pour en faire un élément d'assemblage (voir ci-dessous). À l'exception de ces caractères, certaines combinaisons utilisant `[` (voir les prochains paragraphes), et des échappements (ARE seulement), tous les autres caractères spéciaux perdent leur signification à l'intérieur d'une expression entre crochets. En particulier, `\` n'est pas spécial quand il suit les règles ERE et BRE, bien qu'il soit spécial (pour introduire un échappement) dans les ARE.

À l'intérieur d'une expression entre crochets, un élément d'assemblage (un caractère, une séquence multi-caractères qui s'assemblent comme s'il s'agissait d'un seul caractère, ou un nom de séquence d'assemblage) englobé dans `[.` and `.]` signifie la séquence de caractères de cet élément d'assemblage. La séquence est traitée comme un seul élément de liste d'expression entre crochets. Ceci permet à une expression entre crochets contenant un élément d'assemblage multi-caractères de correspondre à plus d'un caractère. Par exemple, si la séquence d'assemblage inclut un élément d'assemblage `ch`, alors le RE `[[.ch.]]*c` correspond aux cinq premiers caractères de `chchcc`.

> **Note**
>
> PostgreSQL n'accepte pas actuellement les éléments d'assemblage multi-caractères. Cette information décrit un comportement futur possible.

À l'intérieur d'une expression entre crochets, un élément d'assemblage compris entre `[=` et `=]` est une *classe d'équivalence*, représentant les séquences de caractères de tous les éléments d'assemblage équivalent à cette classe, incluant lui-même. (S'il n'existe pas d'autres éléments d'assemblage équivalents, le traitement est identique, comme si les délimiteurs étaient `[.` et `.]`.) Par exemple, si `o` et `^` sont les membres d'une classe d'équivalence, alors `[[=o=]]`, `[[=^=]]` et `[o^]` sont tous synonymes. Une classe d'équivalent ne peut pas être un point final d'un intervalle.

À l'intérieur d'une expression entre crochets, le nom d'une classe de caractères entouré entre `[:` et `:]` signifie la liste de tous les caractères appartenant à cette classe. Une classe de caractères ne peut pas être utilisée comme point final d'un intervalle. Le standard POSIX définit les noms de ces classes de caractères : `alnum` (lettres et chiffres), `alpha` (lettres), `blank` (espace et tabulation), `cntrl` (caractères de contrôle), `digit` (chiffres), `graph` (caractères affichages, sauf l'espace), `lower` (lettres minuscules), `print` (caractères affichages, incluant l'espace), `punct` (ponctuation), `space` (tout espace blanc), `upper` (lettres majuscules), et `xdigit` (chiffres hexadécimaux). Le comportement de ces classes de caractères standards est généralement cohérent sur les plateformes pour les caractères de l'ensemble ASCII 7 bits. Qu'un caractère non ASCII donné appartienne ou non à une de ces classes dépend de la *collation* utilisée par la fonction ou l'opérateur de l'expression rationnelle, (voir Section 24.2), ou par défaut de la locale indiquée par `LC_CTYPE` pour cette base (voir Section 24.1). La classification de caractères non ASCII peut varier entre les plateformes même pour des locales de nom similaire. (Mais la locale `C` ne considère jamais tout caractère non ASCII

comme appartenant à une de ces classes.) En plus de ces classes de caractères standard, PostgreSQL définit la classe de caractères `mot`, ce qui revient au même que `alnum` plus le caractère underscore (_), et la classe de caractères `ascii`, qui contient l'ensemble ASCII 7 bits exactement.

Il existe deux cas spéciaux d'expressions entre crochets : les expressions entre crochets `[[:<:]]` et `[[:>:]]` sont des contraintes, correspondant à des chaînes vides au début et à la fin d'un mot caractère respectivement. Un mot est défini comme une séquence de mots qui n'est ni précédée ni suivie de mots. Un mot caractère correspond à n'importe quel caractère appartenant à la classe de caractères `word`, c'est-à-dire n'importe quel lettre, nombre ou l'underscore. C'est une extension, compatible avec, mais non spécifiée par POSIX 1003.2, et elle devrait être utilisée avec précaution dans les logiciels portables sur d'autres systèmes. Les échappements de contrainte décrits ci-dessous sont généralement préférables ; ils ne sont pas plus standards, mais plus simples à écrire.

9.7.3.3. Échappements d'expression rationnelle

Les *échappements* sont des séquences spéciales commençant avec \ et suivies par un caractère alphanumérique. Les échappements sont de différents types : entrée de caractère, raccourci de classe, échappements de contrainte, et références. Un \ suivi d'un caractère alphanumérique mais ne constituant pas un échappement valide est illégal dans les ARE. Dans le ERE, il n'y a pas d'échappement : en dehors d'une expression entre crochets, un \ suivi d'un caractère alpha-numérique signifie ce caractère alors qu'à l'intérieur d'une expression entre crochets, \ est un caractère standard. (Ce dernier est la seule incompatibilité entre ERE et ARE.)

Les *échappements d'entrée de caractère* existent pour rendre plus simple l'ajout de caractères non affichables, ainsi que d'autres caractères désagréables à saisir dans les RE. Ils sont affichés dans Tableau 9.20.

Les *échappements de raccourci de classe* fournissent des raccourcis pour certaines classes de caractères fréquemment utilisées. Ils sont affichés dans Tableau 9.21.

Un *échappement de contrainte* est une contrainte, correspondant à la chaîne vide si des conditions spécifiques sont rencontrées, écrites comme un échappement. Ils sont affichés dans Tableau 9.22.

Une *référence* (*n*) établit une correspondance de la même chaîne avec la sous-expression entre parenthèses précédente spécifiée par le numéro *n* (voir Tableau 9.23). Par exemple, `([bc])\1` établit une correspondance avec `bb` et `cc`, mais pas `bc` et `cb`. La sous-expression doit précéder entièrement la référence dans le RE. Les sous-expressions sont numérotées dans l'ordre de leur parenthèses de début. Les parenthèses sans capture ne définissent pas de sous-expressions. La référence considère seulement la chaîne de caractères correspondant à la sous-expression référencée, pas les contraintes qu'elle contient. Par exemple, `(^\d)\1` établira une correspondance avec `22`.

Tableau 9.20. Échappements d'entrée de caractère pour les expressions rationnelles

Échappement	Description
`\a`	caractère alerte (cloche), comme en C
`\b`	suppression, comme en C
`\B`	synonyme pour antislash (\) pour aider à réduire le doublement d'antislash
`\c`*X*	(où *X* est tout caractère) le caractère dont les 5 bits de poids faible sont les mêmes que *X*, et dont tous les autres bits valent zéro
`\e`	le caractère donc le nom de séquence d'assemblage est `ESC`, ou, en dehors de cela, le caractère de valeur octale `033`
`\f`	form feed, comme en C

Échappement	Description
`\n`	nouvelle ligne, comme en C
`\r`	retour chariot, comme en C
`\t`	tabulation horizontale, comme en C
`\u`*wxyz*	(où *wxyz* est exactement quatre chiffres hexadécimaux) le caractère dont la valeur hexadécimale est `0x`*wxyz*
`\U`*stuvwxyz*	(où *stuvwxyz* est exactement huit chiffres hexadécimaux) le caractère dont la valeur hexadécimale est `0x`*stuvwxyz*
`\v`	tabulation verticale, comme en C
`\x`*hhh*	(où *hhh* est toute séquence de chiffres hexadécimaux) le caractère dont la valeur hexadécimale est `0x`*hhh* (un seul caractère, peu importe le nombre de chiffres hexadécimaux utilisés)
`\0`	le caractère dont la valeur est `0` (l'octet nul)
`\`*xy*	(où *xy* est exactement deux chiffres octals, et n'est pas une *référence*) le caractère dont la valeur octale est `0`*xy*
`\`*xyz*	(où *xyz* est exactement trois chiffres octales, et n'est pas une *référence*) le caractère dont la valeur octale est `0`*xyz*

Les chiffres hexadécimaux sont `0-9`, `a-f` et `A-F`. Les chiffres octaux sont `0-7`.

Les échappements d'entrés de caractères numériques spécifiant des valeurs en dehors de l'intervalle ASCII (0–127) ont des significations dépendantes de l'encodage de la base de données. Quand l'encodage est UTF-8, les valeurs d'échappement sont équivalents à des points code Unicode. Par exemple, `\u1234` signifie le caractère `U+1234`. Pour les autres encodages multi-octets, les échappements d'entrée de caractères spécifient habituellement juste la concaténation des valeurs d'octet pour le caractère. Si la valeur d'échappement ne correspond pas à tout caractère légale dans l'encodage de la base de données, aucune erreur ne sera levée, mais elle ne correspondra à aucune donnée.

Les échappements d'entrée de classe sont toujours pris comme des caractères ordinaires. Par exemple, `\135` est `]` en ASCII, mais `\135` ne termine pas une expression entre crochets.

Tableau 9.21. Échappements de raccourci de classe pour les expressions rationnelles

Échappement	Description
`\d`	correspond à n'importe quel chiffre, comme `[[:digit:]]`
`\s`	correspond à tout espace blanc, comme `[[:space:]]`
`\w`	correspond à n'importe quel mot, comme `[[:word:]_]`
`\D`	correspond à n'importe quel caractère non-chiffre, comme `[^[:digit:]]`
`\S`	correspond à n'importe quel caractère non-espace blanc, comme `[^[:space:]]`

Échappement	Description
`\W`	correspond à n'import quel caractère non-mot, comme `[^[:alnum:]_]`

Les échappements de raccourci de classe fonctionnent également dans des expressions entre crochets, bien que les définitions ci-dessus ne soient pas tout-à-fait syntaxiquement valides dans ce contexte. Par exemple, `[a-c\d]` est équivalent à `[a-c[:digit:]]`.

Tableau 9.22. Échappements de contraintes pour les expressions rationnelles

Échappement	Description
`\A`	correspond seulement au début de la chaîne (voir Section 9.7.3.5 sur comment cela diffère de `^`)
`\m`	correspond seulement au début d'un mot
`\M`	correspond seulement à la fin d'un mot
`\y`	correspond seulement au début ou à la fin d'un mot
`\Y`	correspond seulement à un point qui n'est pas le début ou la fin d'un mot
`\Z`	correspond seulement à la fin de la chaîne (voir Section 9.7.3.5 sur comment cela diffère de `$`)

Un mot est défini comme dans la spécification de `[[:<:]]` et `[[:>:]]` ci-dessus. Les échappements de contrainte sont illégales à l'intérieur d'une expression entre crochets.

Tableau 9.23. Références d'expressions rationnelles

Échappement	Description
`\`*m*	(où *m* est un chiffre différent de zéro) une référence à la *m*-ième sous-expression
`\`*mnn*	(où *m* est un chiffre différent de zéro, et *nn* est quelques chiffres, et la valeur décimale de *mnn* n'est pas supérieure au nombre de parenthèses fermantes capturantes vues jusqu'à maintenant) une référence à la *mnn*-ième sous-expression

> **Note**
>
> Il existe une ambiguité inhérente entre les échappements d'entrée de caractères en octal et les références. Cette ambiguité est résolue avec les heuristiques suivantes, comme indiqué ci-dessus. Un zéro en début indique toujours un échappement octal. Un chiffre seul différent de zéro, qui n'est pas suivi par un autre chiffre est toujours pris pour une référence. Une séquence de plusieurs chiffres ne commençant pas par un zéro est pris pour une référence si elle survient après une sous-expression acceptable (c'est-à-dire que le numéro est dans l'intervalle légal pour une référence), et sinon est pris pour un octal.

9.7.3.4. Méta-syntaxe des expressions rationnelles

En plus de la syntaxe principale décrite ci-dessus, il existe des formes spéciales et des fonctionnalités syntaxiques diverses disponibles.

Une RE peut commencer avec un des deux préfixes *director* spéciaux. Si une RE commence avec `***:`, le reste de la RE est prise pour une ARE. (Ceci n'a normalement aucun effet dans PostgreSQL,

car les RE sont supposées être des ARE ; mais cela a un effet si le mode ERE ou le mode BRE a été spécifié avec le paramètre *flags* dans une fonction d'expression rationnelle.) Si une RE commence avec ***=, le reste de la RE est prise comme une chaîne littérale, tous les caractères étant considérés comme des caractères ordinaires.

Une ARE peut commander avec des *options intégrées* : une séquence (?*xyz*) (où *xyz* est un ou plusieurs caractères alphabétiques) indique des options affectant le reste de la RE. Ces options surchargent toutes les options préalablement déterminées -- en particulier, elles peuvent surcharger le comportement de sensibilité à la base impliqué par un opérateur d'expression rationnelle ou le paramètre *flags* d'une fonction d'expression rationnelle. Les lettres disponibles pour les options sont indiquées dans Tableau 9.24. Notez que ces mêmes lettres sont utilisées dans les paramètres *flags* des fonctions d'expression rationnelle.

Tableau 9.24. Lettres pour options intégrées des ARE

Option	Description
b	le reste de la RE est une BRE
c	correspondance sensible à la casse (surcharge le type d'opérateur)
e	le reste de la RE est une ERE
i	correspondance insensible à la casse (voir Section 9.7.3.5) (surcharge le type d'opérateur)
m	synonyme historique pour n
n	correspondance sensible à la nouvelle ligne (voir Section 9.7.3.5)
p	correspondance partielle sensible à la nouvelle ligne (voir Section 9.7.3.5)
q	le reste de la RE est une chaîne littérale (« entre guillemets »), avec tous les caractères ordinaires
s	correspondance non sensible à la nouvelle ligne (par défaut)
t	syntaxe serrée (par défaut ; voir ci-dessous)
w	correspondance partielle inverse à la nouvelle ligne (voir Section 9.7.3.5)
x	syntaxe étendue (voir ci-dessous)

Les options intégrées prennent effet au) terminant la séquence. Elles peuvent apparaître seulement au début d'une ARE (après le directeur ***: le cas échéant).

En plus de l'habituelle syntaxe RE (*serrée*), dans laquelle tous les caractères sont signifiants, il existe une syntaxe *étendue*, disponible en spécifiant l'option étendue x. Dans la syntaxe étendue, les caractères d'espace blanc de la RE sont ignorés, comme le sont tous les caractères entre un # et la nouvelle ligne suivante (ou la fin de la RE). Ceci permet les paragraphes et les commentaires dans une RE complexe. Il existe trois exceptions à cette règle basique :

- un caractère espace blanc ou un # précédé par \ est retenu
- un espace blanc ou un # dans une expression entre crochets est retenu
- un espace blanc et les commentaires ne peuvent pas apparaître avec les symboles multi caractères, tels que (?:

Pour cela, les caractères d'espace blanc sont un blanc, une tabulation, un retour à la ligne et tout caractère qui appartient à la classe de caractères *space*.

Enfin, dans une ARE, les expressions en dehors des crochets, la séquence `(?#ttt)` (où *`ttt`* est tout texte ne contenant pas `)`) est un commentaire, complètement ignoré. Encore une fois, ceci n'est pas autorisé entre les caractères des symboles multi-caractères, comme `(?:`. De tels commentaires sont plus un artéfact historique qu'une fonctionnalité utile, et leur utilisation est déconseillée ; utilisez la syntaxe étendue à la place.

Aucune de ces extensions de métasyntaxe n'est disponible si un directeur `***=` initial n'a spécifié que l'entrée de l'utilisateur doit être traitée comme une chaîne littérale plutôt que comme une RE.

9.7.3.5. Règles de correspondance des expressions rationnelles

Dans le cas où une RE correspond à plus d'une sous-chaîne d'une chaîne donnée, la RE établit une correspondance avec la sous-chaîne la plus proche du début de la chaîne. Si la RE correspond à plus d'une sous-chaîne commençant à ce point, la correspondance conservée sera soit la plus longue, soir, la plus courte, suivant que la RE est gourmande (*greedy*) ou non (*non-greedy*).

Le fait qu'une RE est gourmande ou non se détermine avec les règles suivantes :

- La plupart des atomes, et toutes les contraintes, n'ont pas d'attribut de gourmandise (parce qu'ils ne peuvent pas correspondre à des quantités variables de texte).
- Ajouter des parenthèses autour d'une RE ne modifie pas sa gourmandise.
- Un atome quantifié avec un quantifieur de répétition fixe (`{`*`m`*`}` ou `{`*`m`*`}?`) a la même gourmandise (potentiellement aucune) que l'atome lui-même.
- Un atome quantifié avec d'autres quantifieurs standards (incluant `{`*`m`*`,`*`n`*`}` avec *`m`* égal à *`n`*) est gourmand (il préfère la correspondance la plus longue).
- Un atome quantifié avec un quantifieur non gourmand (incluant `{`*`m`*`,`*`n`*`}?` avec *`m`* égal à *`n`*) est non gourmand (il préfère la correspondance la plus courte).
- Une branche -- autrement dit, une RE qui n'a pas d'opérateur `|` hhaut niveau -- a la même gourmandise que le premier atome quantifié qui a un attribut de gourmandise.
- Une RE consistant en deux ou plusieurs branches connectés par l'opérateur `|` est toujours gourmand.

Les règles ci-dessus associent les attributs de gourmandise non seulement aux atomes quantifiés individuels mais aussi aux branches et aux RE complets qui contiennent des atomes quantifiés. Cela signifie que la correspondance est faite de tel façon que la branche ou la RE complète correspondant à la sous-chaîne la plus longue ou la plus courte *au complet*. Une fois que la longueur de la correspondance entière est déterminée, la partie de celle-ci correspondant à une sous-expression particulière est déterminée sur la base de l'attribut de gourmandise de cette sous-expression, avec les sous-expressions commençant plus tôt dans la RE prenant priorité sur les autres.

Un exemple de ce que cela signifie :

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Résultat : 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Résultat : 1
```

Dans le premier cas, la RE complète est gourmande parce que `Y*` est gourmand. Elle peut correspondance en commençant au `Y`, et elle correspond à la chaîne la plus longue commençant là, donc `Y123`. La sortie est la partie entre parenthèse, ou `123`. Dans le deuxième cas, la RE complète est non gourmande parce que `Y*?` est non gourmande. Elle peut correspondre en commençant à `Y`, et elle correspond à la chaîne laplus courte possible commençant là, donc `Y1`. La sous-expression `[0-9]`

`{1,3}` est gourmande mais elle ne peut pas modifier la décision sur la longue de la correspondance ; donc elle est forcée à une correspondance sur simplement `1`.

En court, quand une RE contient des sous-expressions gourmandes et non gourmandes, la longueur de correspondance totale est soit aussi longue ou aussi courte que possible, suivant l'attribut affecté à la RE complète. Les attributs affectés aux sous-expressions affectent seulement la quantité de correspondance qu'elles sont autorisées de « manger » par rapport aux autres.

Les quantifieurs `{1,1}` et `{1,1}?` peuvent être utilisés pour forcer la gourmandise ou la non gourmandise, respectivement, d'une sous-expression ou de la RE complète. Ceci est utile quand vous avez que la RE complète contienne un attribut de gourmandise différent de celui déduit de ces éléments. Comme exemple, supposez que nous essayons de séparer une chaîne contenant quelques chiffres dans les chiffres et les parties avant et après eux. Nous pourrions essayer de le faire ainsi :

```
SELECT regexp_match('abc01234xyz', '(.*)(\d+)(.*)');
Résultat : {abc0123,4,xyz}
```

Cela n'a pas fonctionné : le premier `.*` est gourmand donc il « mange » autant qu'il peut, laissant le `\d+` correspondre à la dernière place possible, le dernier chiffre. Nous pourrions essayer de corriger cela en le rendant non gourmand :

```
SELECT regexp_match('abc01234xyz', '(.*?)(\d+)(.*)');
Résultat : {abc,0,""}
```

Ceci n'a pas fonctionné non plus parce que, maintenant, la RE entière est non gourmande et donc, elle termine la correspondance globale aussi tôt que possible. Nous pouvons obtenir ce que vous voulons en forçant la RE entière à être gourmande :

```
SELECT regexp_match('abc01234xyz', '(?:(.*?)(\d+)(.*)){1,1}');
Résultat : {abc,01234,xyz}
```

Contrôler la gourmandise globale de la RE séparément de la gourmandise de ses composants donne plus de flexibilité dans la gestion des motifs de longueur variable.

Lors de la décision sur ce qui est une correspondance longue ou courte, les longueurs des correspondances sont mesurés en caractères, pas en éléments. Toute chaîne vide est considérée plus longue qu'aucune correspondance. Par exemple : `bb*` correspond au trois caractères du milieu de `abbbc` ; `(week|wee)(night|knights)` correspond aux dix caractères de `weeknights` ; quand `(.*).*` est comparé à `abc`, la sous-expression entre parenthèses correspond aux trois caractères ; et quand `(a*)*` est comparé à `bc`, la RE complète et la sous-expression entre parenthèses correspond à une chaîne vide.

Si une correspondance indépendante à la casse est demandée, l'effet est comme si toutes les distinctions de casse disparaissaient de l'alphabet. Quand un caractère alphabétique qui existe dans plusieurs casses apparaît comme un caractère ordinaire en dehors d'une expression entre crochets, il est en fait transformé en une expression entre crochets contenant les deux casses, par exemple `x` devient `[xX]`. Quand il apparaît dans une expression entre crochets, toutes les casses sont ajoutées dans l'expression entre crochets, par exemple `[x]` devient `[xX]` et `[^x]` devient `[^xX]`.

Si une correspondance sensible aux nouvelles lignes est spécifiée, `.` et les expressions entre crochets utilisant `^` ne correspondront jamais au caractère de nouvelle ligne (pour que les correspondances ne croisent pas les nouvelles lignes sauf si la RE inclut explicitement un saut de ligne) et `^` et `$` correspondront à la chaîne vide respectivement après et avant une nouvelle ligne, en plus d'une correspondance en début et en fin de chaîne respectivement. Cependant, les échappements d'ARE

\A et \Z continuent de correspondre *seulement* au début et à la fin de chaine. Également, les raccourcis de classes de caractères \D et \W correspondront à une nouvelle ligne indépendamment de ce mode. (Avant PostgreSQL 14, ils ne correspondaient pas à de nouvelles lignes dans le mode sensible aux nouvelles lignes. Écrire [^[:digit:]] ou [^[:word:]] pour retrouver cet ancien comportement.)

Si la correspondance partielle sensible aux nouvelles lignes est spécifiée, ceci affecte . et les expressions entre crochets comme avec la correspondance sensible à la nouvelle ligne, mais pas à ^ et $.

Si la correspondance partielle inverse sensible aux nouvelles lignes est spécifiée, ceci affecte ^ et $ comme avec la correspondance sensible aux nouvelles lignes, mais pas à . et aux expressions entre crochets. Ceci n'est pas très utile mais est fourni pour la symétrie.

9.7.3.6. Limites et Compatibilités

Aucune limite particulière n'est imposée sur la longueur des RE dans cette implémentation. Néanmoins, les programmes hautement portablees ne devraient pas employer des RE plus longs que 256 octets, car une implémentation compatible POSIX peut refuser de telles RE.

La seule fonctionnalité des ARE actuellement incompatible avec les ERE POSIX est que \ ne perd pas sa signification spéciale à l'intérieur des expressions entre crochets. Toutes les autres fonctionnalités ARE utilisent une syntaxe illégale ou a des effets non définis ou spécifiés dans les ERE POSIX ; la syntaxe *** des directeurs est en dehors de la syntaxe POSIX pour les BRE et les ERE.

Beaucoup d'extensions ARE sont empruntés de Perl, mais certaines ont été modifiées pour les nettoyer, et quelques extensions Perl ne sont pas présentes. Les incompatibilités de note incluent \b, \B, le manque de traitement spécial pour une nouvelle ligne en fin, l'addition des expressions entre crochets complémentées, les restrictions sur les parenthèses et des références inverses dans les contraintes lookahead/lookbehind et les sémantiques de correspondance pour la correspondance la plus longue/courte (plutôt que la première correspondance).

9.7.3.7. Expressions rationnelles basiques

Les BRE diffèrent des ERE dans différents aspects. Dans les BRE, |, + et ? sont des caractères ordinaires et il n'existe pas d'équivalent à leur fonctionnalité. Les BRE diffèrent des ERE dans différents aspects. Dans les BRE, |, + et ? sont des caractères ordinaires et il n'existe pas d'équivalent à leur fonctionnalité. Les délimiteurs des limites sont \{ et \}. Les parenthèses pour les sous-expressions imbriquées sont \(and \), with (et). ^ est un caractère ordinaire, sauf en début de la RE ou au commencement de la sous-expression entre parenthèses, $ est un caractère standard sauf en fin de RE ou en fin d'une sous-expression entre parenthèses, et * est un caractère standard s'il apparaît au début de la RE ou au début de la sous-expression entre parenthèses (après potentiellement un ^). Enfin, les références inverses d'un simple chiffre sont disponibles, et \< et \> sont des synonymes pour, respectivement, [[:<:]] and [[:>:]] aucun autre échappement n'est dispn,ible dans le BRE.

9.7.3.8. Differences avec le standard SQL et XQuery

Depuis SQL:2008, le standard SQL inclut des fonctions et opérateurs pour les expressions rationnelles qui réalise de la correspondance de motif suivant le standard d'expression rationnelle XQuery.

- LIKE_REGEX
- OCCURRENCES_REGEX
- POSITION_REGEX
- SUBSTRING_REGEX
- TRANSLATE_REGEX

PostgreSQL n'implémente pas actuellement ces opérateurs et fonctions. Vous pouvez obtenir approximativement des fonctionnalités équivalentes dans chaque cas affichés dans Tableau 9.25. (Différentes clauses optionnelles des deux côtés ont été omises dans cette table.)

Tableau 9.25. Équivalences des fonctions sur les expressions rationnelles

Standard SQL	PostgreSQL
string LIKE_REGEX *pattern*	regexp_like(*string*, *pattern*) ou *string* ~ *pattern*
OCCURRENCES_REGEX(*pattern* IN *string*)	regexp_count(*string*, *pattern*)
POSITION_REGEX(*pattern* IN *string*)	regexp_instr(*string*, *pattern*)
SUBSTRING_REGEX(*pattern* IN *string*)	regexp_substr(*string*, *pattern*)
TRANSLATE_REGEX(*pattern* IN *string* WITH *replacement*)	regexp_replace(*string*, *pattern*, *replacement*)

Des fonctions d'expression rationnelles similaires à celles fournies par PostgreSQL sont aussi disponibles dans un certain nombre d'autres implémentations SQL alors que les fonctions du standard SQL ne sont pas aussi largement implémentées. Certains détails de la syntaxe des expressions rationnelles pourraient différer dans chaque implémentation.

Les opérateurs et fonctions du standard SQL utilisent les expressions rationnelles XQuery, qui sont très proches de la syntaxe ERA décrite ci-dessus. Les différences notables entre la fonctionnalité existante d'expressions rationnelles POSIX et les expressions rationnelles XQuery incluent :

- La soustraction de classe d'opérateurs XQuery n'est pas supportée. Un exemple de cette fonctionnalité est l'utilisation de la syntaxe suivante pour établir une correspondance avec uniquement des consonnes anglaises : `[a-z-[aeiou]]`.
- Les raccourcis de classe de caractères XQuery `\c`, `\C`, `\i` et `\I` ne sont pas supportés.
- Les éléments de classe de caractères XQuery utilisant `\p{UnicodeProperty}` ou l'inverse `\P{UnicodeProperty}` ne sont pas supportés.
- POSIX interprète les classes de caractères telles que `\w` (voir Tableau 9.21) suivant la locale prévalente (que vous pouvez contrôler en attachant une clause `COLLATE` à l'opérateur ou la fonction). XQuery spécifie ces classes par référence aux propriétés de caractères Unicode, donc le comportement équivalent est obtenu seulement avec une locale qui suit les règles Unicode.
- Le standard SQL (pas XQuery lui-même) tente de répondre à plus de variantes de « newline » que POSIX. Les options de correspondance pour la sensibilité aux nouvelles lignes décrites ci-dessus considèrent seulement le code ASCII NL (`\n`) cmme étant une nouvelle ligne, mais SQL souhaiterait que nous traitions CR (`\r`), CRLF (`\r\n`) (une nouvelle ligne pour Windows), et certains caractères uniquement Unicode comme LINE SEPARATOR (U+2028) en tant que nouvelles lignes. Par exemple, `.` et `\s` peut compter `\r\n` comme un seul caractère, et non pas deux, suivant le SQL.
- Des échappements d'entrée de caractères décrits dans Tableau 9.20, XQuery supporte seulement `\n`, `\r` et `\t`.
- XQuery ne supporte pas la syntaxe `[:name:]` pour les classes de caractères dans des expressions entre crochets.
- XQuery n'a pas de contraintes lookahead et lookbehind, ni aucun des échappements de contraintes décrits dans Tableau 9.22.

- Les formes de métasyntaxe décrites dans Section 9.7.3.4 n'existe pas dans XQuery.
- Les lettres drapeaux des expressions rationnelles définis par XQuery sont en relation mais pas identique aux lettres options pour POSIX (Tableau 9.24). Bien que les options `i` and `q` se comportent de la même façon, les autres ne le font pas :
  - Les drapeaux `s` (permet à un point de correspondre à une nouvelle ligne) et `m` (autorise `^` et `$` à correspondre à de nouvelles lignes) de XQuery donnent accès aux mêmes comportements que les drapeaux `n`, `p` et `w` de POSIX, mais ils ne correspondent *pas* au comportement des drapeaux `s` et `m` de POSIX. Notez en particulier que le point-correspondant-nouvelle-ligne est le comportement par défaut de POSIX mais pas de XQuery.
  - Le drapeau `x` de XQuery (ignore les espaces blancs dans le motif) est bien différent du drapeau mode étendu de POSIX. Le drapeau `x` de POSIX permet en plus à # de commencer un commentaire dans le motif, et POSIX n'ignorera pas un caractère espace blanc après un antislash.

9.8. Fonctions de formatage de type de données

Les fonctions de formatage de PostgreSQL forment un ensemble puissant d'outils pour convertir différents types de données (date/heure, entier, nombre à virgule flottante, numérique) vers des chaînes formatées et pour convertir de chaînes formatées vers des types de données spécifiques. Tableau 9.26 les liste. Ces fonctions suivent toutes une même convention de nommage : le premier argument est la valeur à formater et le second argument est un modèle qui définit le format de sortie ou d'entrée.

Tableau 9.26. Fonctions de formatage

Fonction **Description** **Exemple(s)**
`to_char ( timestamp, text ) → text` `to_char ( timestamp with time zone, text ) → text` Convertit une donnée de type timestamp en une chaîne suivant le format indiqué. `to_char(timestamp '2002-04-20 17:31:12.66', 'HH12:MI:SS') → 05:31:12`
`to_char ( interval, text ) → text` Convertit un intervalle en une chaîne suivant le format indiqué. `to_char(interval '15h 2m 12s', 'HH24:MI:SS') → 15:02:12`
`to_char ( numeric_type, text ) → text` Convertit un nombre en une chaîne suivant le format indiqué. Disponible pour les types `integer`, `bigint`, `numeric`, `real`, `double precision`. `to_char(125, '999') → 125` `to_char(125.8::real, '999D9') → 125.8` `to_char(-125.8, '999D99S') → 125.80-`
`to_date ( text, text ) → date` Convertit une chaîne en date suivant le format indiqué. `to_date('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05`
`to_number ( text, text ) → numeric` Convertit une chaîne en type numeric suivant le format indiqué. `to_number('12,454.8-', '99G999D9S') → -12454.8`

Fonction **Description** **Exemple(s)**
`to_timestamp ( text, text ) → timestamp with time zone` Convertit une chaîne en type timestamp suivant le format indiqué. (Voir aussi `to_timestamp(double precision)` dans Tableau 9.33.) `to_timestamp('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05 00:00:00-05`

> **Astuce**
>
> `to_timestamp` et `to_date` existent pour gérer les formats en entrée qui ne peuvent être convertis par une simple conversion. Pour les formats plus standards de date/heure, convertir simplement la chaîne source dans le type de données requis fonctionne, et est plus simple. De façon similaire, `to_number` n'est pas nécessaire pour les représentations standards en type numeric.

Dans une chaîne modèle de sortie pour `to_char`, il existe certains motifs reconnus et remplacés avec les données formatées de façon appropriée basées sur la valeur donnée. Tout texte qui n'est pas un motif modèle est simplement copié directement. De façon similaire, dans une chaîne modèle en entrée (pour les autres fonctions), les motifs modèles identifient les valeurs à fournir par la chaîne de données en entrée. S'il existe des caractères dans la chaîne modèle qui ne sont pas des motifs modèles, les caractères correspondants dans la chaîne de données en entrée sont simplement ignorés (qu'ils soient égaux ou non aux caractères de la chaîne modèle).

Tableau 9.27 affiche les motifs modèles disponibles pour le formatage des valeurs date et heure.

Tableau 9.27. Motifs modèles pour le formatage date/heure

Motif	**Description**
`HH`	heure du jour (01–12)
`HH12`	heure du jour (01–12)
`HH24`	heure du jour (00–23)
`MI`	minute (00–59)
`SS`	seconde (00–59)
`MS`	milliseconde (000–999)
`US`	microseconde (000000–999999)
`FF1`	dixième de seconde (0–9)
`FF2`	centième de seconde (00–99)
`FF3`	milliseconde (000–999)
`FF4`	dixième de milliseconde (0000–9999)
`FF5`	centième de milliseconde (00000–99999)
`FF6`	microseconde (000000–999999)
`SSSS`, `SSSSS`	secondes après minuit (0–86399)
`AM`, `am`, `PM` ou `pm`	indicateur de méridien (sans points)
`A.M.`, `a.m.`, `P.M.` or `p.m.`	indicateur de méridien (avec points)
`Y,YYY`	année (4 chiffres ou plus) avec virgule
`YYYY`	année (4 chiffres ou plus)

Motif	Description
YYY	3 derniers chiffres de l'année
YY	2 derniers chiffres de l'année
Y	dernier chiffre de l'année
IYYY	année, numérotation ISO 8601 (4 chiffres ou plus)
IYY	3 derniers chiffres de l'année, numérotation ISO 8601
IY	2 derniers chiffres de l'année, numérotation ISO 8601
I	dernier chiffre de l'année, numérotation ISO 8601
BC, bc, AD ou ad	indicateur d'ère (sans points)
B.C., b.c., A.D. ou a.d.	indicateur d'ère (avec points)
MONTH	nom du mois en majuscule (sur 9 caractères)
Month	nom du mois capitalisé (sur 9 caractères)
month	nom du mois en minuscule (sur 9 caractères)
MON	abréviation du nom du mois en majuscule (3 caractères en anglais, peut varier ailleurs)
Mon	abréviation du nom du mois capitalisé (3 caractères en anglais, peut varier ailleurs)
mon	abréviation du nom du mois en minuscule (3 caractères en anglais, peut varier ailleurs)
MM	numéro du mois (01–12)
DAY	nom du jour en majuscule (sur 9 caractères)
Day	nom du jour capitalisé (sur 9 caractères)
day	nom du jour en minuscule (sur 9 caractères)
DY	abréviation du nom du jour en majuscule (3 caractères en anglais, peut varier ailleurs)
Dy	abréviation du nom du jour capitalisé (3 caractères en anglais, peut varier ailleurs)
dy	abréviation du nom du jour en minuscule (3 caractères en anglais, peut varier ailleurs)
DDD	jour de l'année (001–366)
IDDD	jour de l'année, d'après le standard ISO 8601 (001–371 ; la journée 1 de l'année est le lundi de la première semaine ISO)
DD	jour du mois (01–31)
D	jour de la semaine, de dimanche (1) à samedi (7)
ID	jour de la semaine d'après le standard ISO 8601, de lundi (1) à dimanche (7)
W	numéro de semaine du mois (1–5) (la première semaine commence le premier jour du mois)
WW	numéro de semaine de l'année (1–53) (la première semaine commence le premier jour de l'année)

Motif	Description
IW	numéro de semaine de l'année, d'après le standard ISO 8601 (01–53 ; le premier jeudi de l'année est en semaine 1)
CC	siècle (2 chiffres) (le 21è siècle commence le 1er janvier 2001)
J	date Julien (nombre de jours depuis le 24 novembre, 4714 avant JC à minuit heure locale ; voir Section B.7)
Q	quartier
RM	mois en majuscules suivant la convention romaine (I–XII; I=janvier)
rm	mois en minuscules suivant le convention romaine (i–xii; i=janvier)
TZ	abréviation du fuseau horaire, en majuscule (supporté uniquement avec to_char)
tz	abréviation du fuseau horaire, en minuscule (supporté uniquement avec to_char)
TZH	heures du fuseau horaire
TZM	minutes du fuseau horaire
OF	décalage du fuseau horaire à partir d'UTC (supporté uniquement avec to_char)

Les modificateurs peuvent être appliqués à tous les motifs modèles pour modifier leurs comportements. Par exemple, FMMonth est le motif Month avec le modificateur FM. Tableau 9.28 affiche les motifs de modificateur pour le formatage date/heure.

Tableau 9.28. Modificateurs des motifs modèles pour le formatage date/heure

Modificateur	Description	Exemple
préfixe FM	mode de remplissage (supprime les zéros avant et les blancs de remplissage)	FMMonth
Suffixe TH	suffixe du numéro ordinal en majuscule	DDTH, e.g., 12TH
Suffixe th	suffixe du numéro ordinal en minuscule	DDth, e.g., 12th
préfixe FX	option globale pour le format fixe (voir les notes d'utilisation)	FX Month DD Day
préfixe TM	mode de traduction (utilise les noms localisés du jour et du mois basés sur lc_time)	TMMonth
Suffixe SP	spell mode (non implémenté)	DDSP

Notes d'utilisation du formatage date/heure :

- FM supprime les zéros en début et les blancs à la fin qui pourraient avoir été ajoutés pour s'assurer d'une taille fix de la sortie. Dans PostgreSQL, FM modifie seulement la spécification suivante alors que dans Oracle, FM affecte toutes les spécifications suivantes, et des modificateurs FM répétés vont basculer l'activation et la désactivation du mode de remplissage.

- TM supprime les espaces en fin de chaîne que FM soit spécifié ou non.

- `to_timestamp` et `to_date` ignorent la casse des lettres en entrée ; donc par exemple `MON`, `Mon`, et `mon` acceptent toutes les mêmes chaînes. Lors de l'utilisation du modificateur `TM`, le changement de casse se fait suivant les règles de la collation pour les données en entrée de la fonction (voir Section 24.2).

- `to_timestamp` et `to_date` ignorent les nombreux espaces blancs au début de la chaîne en entrée et autour des valeurs de date et d'heure sauf si l'option `FX` est utilisée. Par exemple, `to_timestamp(' 2000 JUN', 'YYYY MON')` et `to_timestamp('2000 - JUN', 'YYYY-MON')` fonctionnent, mais `to_timestamp('2000 JUN', 'FXYYYY MON')` renvoie une erreur parce que `to_timestamp` s'attend seulement à un seul espace. `FX` doit être spécifié comme premier élément dans le modèle.

- Un séparateur (un espace ou un caractère qui n'est ni une lettre ni un chiffre) dans la chaîne modèle de `to_timestamp` et `to_date` correspond à tout séparateur dans la chaîne en entrée ou est ignoré, sauf si l'option `FX` est utilisée. Par exemple, `to_timestamp('2000JUN', 'YYYY///MON')` et `to_timestamp('2000/JUN', 'YYYY MON')` fonctionnent mais `to_timestamp('2000//JUN', 'YYYY/MON')` renvoie une erreur parce que le nombre de séparateurs dans la chaîne en entrée dépasse le nombre de séparateurs dans le modèle.

  Si `FX` est indiqué, un séparateur dans la chaîne modèle correspond exactement à un caractère dans la chaîne en entrée. Mais notez que le caractère de la chaîne en entrée n'a pas besoin d'être identique au séparateur de la chaîne modèle. Par exemple, `to_timestamp('2000/JUN', 'FXYYYY MON')` fonctionne mais `to_timestamp('2000/JUN', 'FXYYYY MON')` renvoie une erreur parce que le deuxième espace dans la chaîne modèle consomme la lettre `J` de la chaîne en entrée.

- Un motif de modèle `TZH` peut correspondre à un nombre signé. Sans l'option `FX`, les signes moins pourraient être ambigus, et pourraient être interprétés comme un séparateur. Cette ambiguité est résolue ainsi ; si le nombre de séparateurs avant `TZH` dans la chaîne modèle est inférieur au nombre de séparateur avant le signe moins dans la chaîne en entrée, le signe moins est interprété comme faisant partie de `TZH`. Sinon le signe moins est considéré être un séparateur entre les valeurs. Par exemple, `to_timestamp('2000 -10', 'YYYY TZH')` fait correspondre `-10` à `TZH`, mais `to_timestamp('2000 -10', 'YYYY TZH')` fait correspondre `10` à `TZH`.

- Un texte ordinaire est autorisé dans les modèles `to_char` et sera affiché littéralement. Vous pouvez placer une sous-chaîne entre des guillemets doubles pour forcer son interprétation comme un texte littéral, même s'il contient des motifs de modèle. Par exemple, dans `'"Hello Year "YYYY'`, le texte `YYYY` sera remplacé par l'année, mais le simple `Y` dans `Year` ne le sera pas. Dans `to_date`, `to_number` et `to_timestamp`, le texte littéral et les chaînes entre guillemets doubles finissent en ignorant le nombre de caractères contenus dans la chaîne ; par exemple, `"XX"` ignore les deux caractères en entrée (qu'ils soient ou non `XX`).

  > **Astuce**
  >
  > Avant PostgreSQL 12, il était possible d'ignorer du texte arbitraire dans la chaîne en entrée en utilisant des caractères qui ne sont ni des lettres ni des chiffres. Par exemple, `to_timestamp('2000y6m1d', 'yyyy-MM-DD')` fonctionnait. Maintenant, vous pouvez seulement utiliser des caractères lettres pour cela. Par exemple, `to_timestamp('2000y6m1d', 'yyyytMMtDDt')` et `to_timestamp('2000y6m1d', 'yyyy"y"MM"m"DD"d"')` ignorent `y`, `m` et `d`.

- Si vous voulez avoir un guillemet double dans la sortie, vous devez la précéder avec un antislash, par exemple `'\"YYYY Month\"'`. En dehors de cela, les antislashs ne sont pas particulièrement spéciaux en dehors des chaînes entre guillemets doubles. À l'intérieur d'une chaîne entre guillemets doubles, un antislash peut faire que le prochain caractère soit considéré littéralement, quel qu'il soit (mais ceci n'a pas d'effet spécial sauf si le caractère suivant est un guillemet double ou un autre antislash).

- Dans `to_timestamp` et `to_date`, si la spécification du format de l'année fait moins de quatre chiffres, donc `YYY`, et que l'année indiquée fait moins de quatre chiffres, l'année sera ajustée au plus proche de l'année 2020, donc `95` devient 1995.

- Dans `to_timestamp` et `to_date`, les années négatives sont traitées comme signifiant avant Jésus Christ. Si vous écrivez à la fois une année négative et un champ `BC` explicite, vous obtenez après Jésus Christ de nouveau. Une entrée année zéro est traitée comme l'année 1 après Jésus Christ.

- Dans `to_timestamp` et `to_date`, la conversion `YYYY` a une restriction lors du traitement d'années avec plus de 4 chiffres. Vous devez utiliser certains caractères qui ne sont pas des chiffres ou un modèle après `YYYY`, sinon l'année est toujours interprétée avec 4 chiffres. Par exemple (avec l'année 20000) : `to_date('200001130', 'YYYYMMDD')` sera interprété comme une année sur 4 chiffres ; à la placez, utilisez un séparateur qui n'est pas un chiffre après l'année, comme par exemple `to_date('20000-1130', 'YYYY-MMDD')` ou `to_date('20000Nov30', 'YYYYMonDD')`.

- Dans `to_timestamp` et `to_date`, le champ `CC` (siècle) est accepté mais ignoré s'il existe un champ `YYY`, `YYYY` ou `Y,YYY`. Si `CC` est utilisé avec `YY` ou `Y`, alors le résultat est calculé comme l'année dans le siècle spécifié. Si le siècle est spécifié, mais pas l'année, la première année du siècle est présumée.

- Dans `to_timestamp` et `to_date`, les noms ou numéros des jours de la semaine (`DAY`, `D`, et les champs relatifs) sont acceptés mais ignorés pour calculer le résultat. Ceci est vrai aussi pour le champ `Q`.

- Dans `to_timestamp` et `to_date`, une date d'après la numérotation ISO 8601 (dans le sens distinct d'une date grégorienne) peut être indiqué de deux façons :

  - Année, numéro de semaine et jour de la semaine : par exemple `to_date('2006-42-4', 'IYYY-IW-ID')` renvoie la date date `2006-10-19`. Si vous omettez le jour de la semaine, il est supposé valoir 1 (lundi).

  - Année et jour de l'année : par exemple `to_date('2006-291', 'IYYY-IDDD')` renvoie aussi `2006-10-19`.

  Tenter de saisir une date en utilisant un mix de champs ISO 8601 et de champs grégoriens n'a pas de sens, et résultera en une erreur. Dans le contexte année-numéro de semaine sous l'ISO 8601, le concept d'un « mois » ou du « jour d'un mois » n'a pas de signification. Dans le contexte d'une année grégorienne, la semaine ISO n'a pas de signification.

  > **Attention**
  >
  > Alors que `to_date` rejettera un mix entre champs pour date grégorienne et pour date ISO, `to_char` ne le fera pas parce que les spécifications de format de sortie comme `YYYY-MM-DD (IYYY-IDDD)` peuvent se révéler utile. Mais évitez d'écrire quelque chose comme `IYYY-MM-DD` ; cela pourrait amener des résultats surprenant près du début de l'année. (Voir Section 9.9.1 pour plus d'informations.)

- Dans `to_timestamp`, les champs milliseconde (`MS`) et microseconde (`US`) sont utilisés comme les chiffres secondes après le point décimal. Par exemple, `to_timestamp('12.3', 'SS.MS')` ne vaut pas 3 millisecondes, mais 300 parce que la conversion la traite comme 12 + 0,3 secondes. Donc, pour le format `SS.MS`, les valeurs en entrée `12.3`, `12.30` et `12.300` spécifient le même nombre de millisecondes. Pour obtenir trois millisecondes, il faut écrire `12.003`, que la conversion traite comme 12 + 0,003 = 12,003 secondes.

  Voici un exemple plus complexe : `to_timestamp('15:12:02.020.001230', 'HH24:MI:SS.MS.US')` est 15 heures, 12 minutes, et 2 secondes + 20 millisecondes + 1230 microsecondes = 2,021230 secondes.

- la numérotation du jour de la semaine à partir de `to_char(..., 'ID')` correspond à la fonction `extract(isodow from ...)` mais `to_char(..., 'D')` ne correspond pas à la numérotation des jours de `extract(dow from ...)`.

- `to_char(interval)` formate `HH` et `HH12` comme indiqué sur une horloge 12-heures, par exemple; l'heure zéro et l'heure 36 s'affichent toutes les deux comme `12`, alors que `HH24` affiche la valeur complète de l'heure qui peut dépasser 23 dans une valeur de type `interval`.

Tableau 9.29 montre les motifs de modèle disponibles pour le formatage des valeurs numériques.

Tableau 9.29. Motifs de modèle pour le formatage numérique

Motif	Description
`9`	position d'un chiffre (peut être supprimé si non significatif)
`0`	position d'un chiffre (ne sera pas supprimé même si non significatif)
`.` (point)	point décimal
`,` (virgule)	séparateur de groupe (milliers)
`PR`	valeur négative entre crochets
`S`	signe accroché au nombre (utilise la locale)
`L`	symbôle de monnaie (utilise la locale)
`D`	point décimal (utilise la locale)
`G`	séparateur de groupe (utilise la locale)
`MI`	signe moins dans la position spécifiée (si nombre < 0)
`PL`	signe plus dans la position spécifiée (si nombre > 0)
`SG`	signe plus/moins dans la position spécifiée
`RN`	nombre romain (entrée entre 1 et 3999)
`TH` or `th`	suffixe du numéro ordinal
`V`	décalage spécifié en nombre de chiffres (voir notes)
`EEEE`	exposant pour la notation scientifique

Notes d'utilisation pour le formatage numérique :

- `0` spécifie une position de chiffre qui sera toujours affiché, même s'il contient un zéro en début ou en fin. `9` spécifie aussi une position de chiffre mais, s'il s'agit un zéro en début, alors il sera remplacé par un espace, alors que s'il s'agit d'un zéro en fin et que le mode de remplissage est activé, alors il sera supprimé. (Pour `to_number()`, ces deux modèles sont équivalents.)

- si le format fournit moins de chiffres décimaux que le nombre en cours de formatage, `to_char()` arrondira le nombre du nombre indiqué de chiffres décimaux.

- Les caractères de modèle `S`, `L`, `D` et `G` représentent le signe, le symbôle de monnaie, le point décimal et le séparateur de milliers définis par la locale actuelle (voir lc_monetary et lc_numeric). Le point et la virgule représentent ces caractères avec la signification du point décimal et du séparateur de milliers, quelque soit la locale.

- Si aucune provision explicite n'est faite pour un signe dans le motif de `to_char()`, une colonne sera réservée pour le signe, et il sera ancré au chiffre (il apparaîtra à gauche). Si `S` apparaît juste à gauche de `9`, il sera quand même ancré au nombre.

- Un signe formaté utilisant `SG`, `PL` ou `MI` n'est pas ancré au nombre ; par exemple, `to_char(-12, 'MI9999')` produit `'-  12'` mais `to_char(-12, 'S9999')` produit `'  -12'`. (L'implémentation Oracle ne permet pas l'utilisation de `MI` avant `9`, mais nécessite plutôt que `9` précède `MI`.)

- `TH` ne convertit pas les valeurs inférieures à zéro et ne convertit pas les nombres fractionnés.

- `PL`, `SG` et `TH` sont des extensions de PostgreSQL.

- Dans `to_number`, si des motifs de modèle non-données tels que `L` ou `TH` sont utilisés, le nombre correspondant de caractères en entrée est ignoré qu'ils correspondent ou non au motif du modèle, sauf si ce sont des caractères de données (c'est-à-dire chiffres, signe, point décimal ou virgule). Par exemple, `TH` ignorerait les deux caractères qui ne sont pas des données.

- `V` avec `to_char` multiplie les valeurs en entrée par `10^`*n*, où *n* est le nombre de chiffres suivant `V`. `V` avec `to_number` divise d'une façon similaire. `to_char` et `to_number` ne supportent pas l'utilisation de `V` en combinaison avec un point décimal (de ce fait, `99.9V99` n'est pas autorisé).

- `EEEE` (notation scientifique) ne peut pas être utilisé en combinaison avec tout autre motif de formatage ou modificateur autre que les motifs de chiffres et de point décimal, et doit être à la fin de la chaîne de format (autrement dit, `9.99EEEE` est un motif valide).

Certains modificateurs peuvent être appliqués à tout motif de modèle pour modifier son comportement. Par exemple, `FM99.99` est le motif `99.99` avec le modificateur `FM`. Tableau 9.30 montre les motifs de modificateur pour le formatage numérique.

Tableau 9.30. Modificateurs de motifs de modèle pour le formatage numérique

Modificateur	Description	Exemple
préfixe `FM`	mode de remplissage (supprime les zéros en fin et les espaces d'alignement)	`FM99.99`
suffixe `TH`	suffixe du nombre ordinal en majuscule	`999TH`
suffixe `th`	suffixe du nombre ordinal en minuscule	`999th`

Tableau 9.31 montre quelques exemples de l'utilisation de la fonction `to_char`.

Tableau 9.31. `to_char` Examples

Expression	Résultat
`to_char(current_timestamp, 'Day, DD  HH12:MI:SS')`	`'Tuesday  , 06  05:39:18'`
`to_char(current_timestamp, 'FMDay, FMDD  HH12:MI:SS')`	`'Tuesday, 6  05:39:18'`
`to_char(-0.1, '99.99')`	`'  -.10'`
`to_char(-0.1, 'FM9.99')`	`'-.1'`
`to_char(-0.1, 'FM90.99')`	`'-0.1'`
`to_char(0.1, '0.9')`	`' 0.1'`
`to_char(12, '9990999.9')`	`'    0012.0'`
`to_char(12, 'FM9990999.9')`	`'0012.'`
`to_char(485, '999')`	`' 485'`

Expression	Résultat
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485'
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'
to_char(0.0004859, '9.99EEEE')	' 4.86e-04'

9.9. Fonctions et opérateurs pour date/heure

Tableau 9.33 affiche les fonctions disponibles pour le traitement de valeurs date/heure, avec des détails dans les sections suivantes. Tableau 9.32 illustre le comportement des opérateurs arithmétiques de base (+, *, etc.). Pour les fonctions de formatage, référez-vous à Section 9.8. Vous devez être déjà familier avec les types de données date/heure (voir Section 8.5).

De plus, les opérateurs de comparaison habituels affichés dans Tableau 9.1 sont disponibles pour les types date/heure. Les dates et timestamps (avec ou sans fuseau horaire) sont tous comparables, alors que les heures (avec et sans fuseau horaire) et les intervalles peuvent seulement être comparés aux autres valeurs du même type de données. Lors de la comparaison d'un timestamp sans fuseau horaire à un timestamp avec fuseau horaire, la première valeur est supposée être donnée dans le fuseau horaire indiqué par le paramètre de configuration TimeZone, et est transformée en UTC pour comparaison

avec la deuxième valeur (qui est déjà en UTC). De façon similaire, une valeur date est supposée représenter minuit dans la zone `TimeZone` lors de la comparaison avec un timestamp.

Toutes les fonctions et tous les opérateurs décrits ci-dessous, prenant en entrée des arguments de type `time` ou `timestamp` viennent en deux variantes : une qui prend le type `time with time zone` ou `timestamp with time zone`, et une qui prend le type `time without time zone` ou `timestamp without time zone`. Pour ne pas trop grossir la documentation, ces variantes ne sont pas affichées séparément. De plus, les opérateurs `+` et `*` viennent sous la forme de paires commutatives (par exemple, à la fois `date + integer` et `integer + date`) ; nous ne présentons qu'une seule des deux.

Tableau 9.32. Opérateurs Date/Heure

Opérateur **Description** **Exemple(s)**
`date + integer → date` Ajoute un nombre de jours à une date `date '2001-09-28' + 7 → 2001-10-05`
`date + interval → timestamp` Ajouter un interval à une date `date '2001-09-28' + interval '1 hour' → 2001-09-28 01:00:00`
`date + time → timestamp` Ajoute une heure du jour à une date `date '2001-09-28' + time '03:00' → 2001-09-28 03:00:00`
`interval + interval → interval` Ajoute des intervalles `interval '1 day' + interval '1 hour' → 1 day 01:00:00`
`timestamp + interval → timestamp` Ajoute un interval à un timestamp `timestamp '2001-09-28 01:00' + interval '23 hours' → 2001-09-29 00:00:00`
`time + interval → time` Ajoute un interval à une heure `time '01:00' + interval '3 hours' → 04:00:00`
`- interval → interval` Passe un interval en négatif `- interval '23 hours' → -23:00:00`
`date - date → integer` Soustrait des dates, produisant le nombre de jours passés `date '2001-10-01' - date '2001-09-28' → 3`
`date - integer → date` Soustrait un nombre de jours d'une date `date '2001-10-01' - 7 → 2001-09-24`
`date - interval → timestamp` Soustrait un interval d'une date `date '2001-09-28' - interval '1 hour' → 2001-09-27 23:00:00`

Opérateur **Description** **Exemple(s)**
`time` - `time` → `interval` Soustrait des heures `time '05:00' - time '03:00'` → `02:00:00`
`time` - `interval` → `time` Soustrait un interval d'une heure `time '05:00' - interval '2 hours'` → `03:00:00`
`timestamp` - `interval` → `timestamp` Soustrait un interval d'un timestamp `timestamp '2001-09-28 23:00' - interval '23 hours'` → `2001-09-28 00:00:00`
`interval` - `interval` → `interval` Soustrait des intervalles `interval '1 day' - interval '1 hour'` → `1 day -01:00:00`
`timestamp` - `timestamp` → `interval` Soustrait des timestamps (convertissant des intervalles sur 24 heures en jours, de façon similaire à `justify_hours()`) `timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'` → `1 day 15:00:00`
`interval` * `double precision` → `interval` Multiplie un interval à une valeur scalaire `interval '1 second' * 900` → `00:15:00` `interval '1 day' * 21` → `21 days` `interval '1 hour' * 3.5` → `03:30:00`
`interval` / `double precision` → `interval` Divise un interval par une valeur scalaire `interval '1 hour' / 1.5` → `00:40:00`

Tableau 9.33. Fonctions Date/Heure

Fonction **Description** **Exemple(s)**
`age` (`xid`) → `integer` Renvoie le nombre de transactions entre l'identifiant de transaction fourni et le compteur actuel de transaction.
`mxid_age` (`xid`) → `integer` Renvoie le nombre d'identifiant MultiXact entre l'identifiant MultiXact fourni et le compteur actuel de MultiXact.
`age` (`timestamp`, `timestamp`) → `interval` Soustrait les arguments, produisant un résultat « symbolique » qui utilise des années et des mois, plutôt que simplement des jours `age(timestamp '2001-04-10', timestamp '1957-06-13')` → `43 years 9 mons 27 days`
`age` (`timestamp`) → `interval`

Fonction **Description** **Exemple(s)**
Soustrait l'argument de `current_date` (à minuit) `age(timestamp '1957-06-13')` → `62 years 6 mons 10 days`
`clock_timestamp ( )` → `timestamp with time zone` Date et heure actuelles (change lors de l'exécution d'une requête) ; voir Section 9.9.5 `clock_timestamp()` → `2019-12-23 14:39:53.662522-05`
`current_date` → `date` Date actuelle ; voir Section 9.9.5 `current_date` → `2019-12-23`
`current_time` → `time with time zone` Heure actuelle du jour ; voir Section 9.9.5 `current_time` → `14:39:53.662522-05`
`current_time ( integer )` → `time with time zone` Haure actuelle du jour avec une précision limitée ; voir Section 9.9.5 `current_time(2)` → `14:39:53.66-05`
`current_timestamp` → `timestamp with time zone` Date et heure actuelles (début de la transaction en cours) ; voir Section 9.9.5 `current_timestamp` → `2019-12-23 14:39:53.662522-05`
`current_timestamp ( integer )` → `timestamp with time zone` Date et heure actuelles (début de la transaction en cours) avec une précision limitée ; voir Section 9.9.5 `current_timestamp(0)` → `2019-12-23 14:39:53-05`
`date_bin ( interval, timestamp, timestamp )` → `timestamp` Place une date dans l'intervalle spécifié, alignée avec une origine donnée ; voir Section 9.9.3 `date_bin('15 minutes', timestamp '2001-02-16 20:38:40', timestamp '2001-02-16 20:05:00')` → `2001-02-16 20:35:00`
`date_part ( text, timestamp )` → `double precision` Obtenir un champ du timestamp (équivalent à `extract`) ; voir Section 9.9.1 `date_part('hour', timestamp '2001-02-16 20:38:40')` → `20`
`date_part ( text, interval )` → `double precision` Obtenir un champ interval (équivalent à `extract`) ; voir Section 9.9.1 `date_part('month', interval '2 years 3 months')` → `3`
`date_trunc ( text, timestamp )` → `timestamp` Tronque à la précision spécifiée ; voir Section 9.9.2 `date_trunc('hour', timestamp '2001-02-16 20:38:40')` → `2001-02-16 20:00:00`
`date_trunc ( text, timestamp with time zone, text )` → `timestamp with time zone` Tronque à une précision spécifiée dans le fuseau horaire indiqué ; voir Section 9.9.2 `date_trunc('day', timestamptz '2001-02-16 20:38:40+00', 'Australia/Sydney')` → `2001-02-16 13:00:00+00`
`date_trunc ( text, interval )` → `interval`

Fonction **Description** **Exemple(s)**
Tronque à la précision spécifiée ; voir Section 9.9.2 `date_trunc('hour', interval '2 days 3 hours 40 minutes')` → `2 days 03:00:00`
`extract (` *`field`* `from timestamp )` → `numeric` Obtenir un champ du timestamp ; voir Section 9.9.1 `extract(hour from timestamp '2001-02-16 20:38:40')` → `20`
`extract (` *`field`* `from interval )` → `numeric` Obtenir un champ interval ; voir Section 9.9.1 `extract(month from interval '2 years 3 months')` → `3`
`isfinite ( date )` → `boolean` Test pour une date finie (différent de +/-infinity) `isfinite(date '2001-02-16')` → `true`
`isfinite ( timestamp )` → `boolean` Test pour un timestamp fini (différent de +/-infinity) `isfinite(timestamp 'infinity')` → `false`
`isfinite ( interval )` → `boolean` Test pour un interval fini (actuellement toujours vrai) `isfinite(interval '4 hours')` → `true`
`justify_days ( interval )` → `interval` Ajuste un interval en convertissant les périodes de 30 jours en mois `justify_days(interval '1 year 65 days')` → `1 year 2 mons 5 days`
`justify_hours ( interval )` → `interval` Ajuste un interval en convertissant les périodes de 24 heures en jour `justify_hours(interval '50 hours 10 minutes')` → `2 days 02:10:00`
`justify_interval ( interval )` → `interval` Adjuste un interval en utilisant `justify_days` et `justify_hours`, avec des ajustements de signe supplémentaire `justify_interval(interval '1 mon -1 hour')` → `29 days 23:00:00`
`localtime` → `time` Heure actuelle du jour ; voir Section 9.9.5 `localtime` → `14:39:53.662522`
`localtime ( integer )` → `time` Heure actuelle du jour avec une précision limitée ; voir Section 9.9.5 `localtime(0)` → `14:39:53`
`localtimestamp` → `timestamp` Date et heure actuelles (début de la transaction en cours) ; voir Section 9.9.5 `localtimestamp` → `2019-12-23 14:39:53.662522`
`localtimestamp ( integer )` → `timestamp`

Fonction **Description** **Exemple(s)**
Date et heure actuelles (début de la transaction en cours), avec une précision limitée ; voir Section 9.9.5 `localtimestamp(2)` → `2019-12-23 14:39:53.66`
`make_date` (*`year`* `int`, *`month`* `int`, *`day`* `int`) → `date` Crée une date à partir des champs année, mois et jour (negative years signify BC) `make_date(2013, 7, 15)` → `2013-07-15`
`make_interval` ([*`years`* `int` [, *`months`* `int` [, *`weeks`* `int` [, *`days`* `int` [, *`hours`* `int` [, *`mins`* `int` [, *`secs`* `double precision`]]]]]]) → `interval` Crée un interval à partir des champs année, mois, semaine, jour, heure, minute et seconde, chacun ayant zéro comme valeur par défaut `make_interval(days => 10)` → `10 days`
`make_time` (*`hour`* `int`, *`min`* `int`, *`sec`* `double precision`) → `time` Crée une heure à partir des champs heure, minute et seconde `make_time(8, 15, 23.5)` → `08:15:23.5`
`make_timestamp` (*`year`* `int`, *`month`* `int`, *`day`* `int`, *`hour`* `int`, *`min`* `int`, *`sec`* `double precision`) → `timestamp` Crée un timestamp à partir des champs année, mois, jour, heure, minute et seconde (negative years signify BC) `make_timestamp(2013, 7, 15, 8, 15, 23.5)` → `2013-07-15 08:15:23.5`
`make_timestamptz` (*`year`* `int`, *`month`* `int`, *`day`* `int`, *`hour`* `int`, *`min`* `int`, *`sec`* `double precision` [, *`timezone`* `text`]) → `timestamp with time zone` Crée un timestamp avec fuseau horaire à partir des champs année, mois, jour, heure, minute et seconde (negative years signify BC). Si *`timezone`* n'est pas spécifié, le fuseau horaire actuel est utilisé; les exemples supposent que le fuseau horaire de la session est `Europe/London` `make_timestamptz(2013, 7, 15, 8, 15, 23.5)` → `2013-07-15 08:15:23.5+01` `make_timestamptz(2013, 7, 15, 8, 15, 23.5, 'America/New_York')` → `2013-07-15 13:15:23.5+01`
`now` ( ) → `timestamp with time zone` Date et heure actuelles (début de la transaction en cours) ; voir Section 9.9.5 `now()` → `2019-12-23 14:39:53.662522-05`
`statement_timestamp` ( ) → `timestamp with time zone` Date et heure actuelles (début de la requête en cours) ; voir Section 9.9.5 `statement_timestamp()` → `2019-12-23 14:39:53.662522-05`
`timeofday` ( ) → `text` Date et heure actuelles (comme `clock_timestamp`, mais sous la forme d'une chaîne de type `text`) ; voir Section 9.9.5 `timeofday()` → `Mon Dec 23 14:39:53.662522 2019 EST`
`transaction_timestamp` ( ) → `timestamp with time zone` Date et heure actuelles (début de la transaction en cours) ; voir Section 9.9.5 `transaction_timestamp()` → `2019-12-23 14:39:53.662522-05`

Fonction **Description** **Exemple(s)**
`to_timestamp ( double precision ) → timestamp with time zone` Convertit un epoch Unix (secondes depuis 1970-01-01 00:00:00+00) en timestamp avec fuseau horaire `to_timestamp(1284352323) → 2010-09-13 04:32:03+00`

En plus de ces fonctions, l'opérateur SQL `OVERLAPS` est géré :

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

Cette expression renvoie true quand deux périodes de temps (définies par leur point final) se chevauchent et false dans le cas contraire. Les points finaux peuvent être indiqués sous la forme d'une paires de dates, heures ou dates et heures, ou à une date, heure ou date et heure suivie par un intervalle. Quand une paire de valeurs est fournie, soit le début soit la fin peuvent être écrit en premier ; `OVERLAPS` prend automatiquement la valeur la plus ancienne de la paire comme valeur de départ. Chaque période de temps est considérée comme représentant l'intervalle à moitié ouvert *`start`* <= *`time`* < *`end`*, sauf si *`start`* et *`end`* sont égaux, auquel cas elle représente uniquement cet instant. Ceci signifie que deux périodes de temps avec seulement le point final en commun ne se surchargent pas.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Résultat : true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Résultat : false
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Résultat : false
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Résultat : true
```

Lors de l'ajout (ou de la soustraction) d'une valeur `interval` à une valeur `timestamp with time zone`, le nombre de jours augmente ou décrémente la date de `timestamp with time zone` par le nombre de jours indiqué, conservant à l'identique l'heure du jour. Si cela intervient à une période de changement d'heure pour le fuseau horaire utilisé, cela signifie que `interval '1 day'` n'est pas forcément équivalent à `interval '24 hours'`. Par exemple, avec le fuseau horaire `America/Denver` :

```
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval
 '1 day';
Résultat : 2005-04-03 12:00:00-06
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval
 '24 hours';
Résultat : 2005-04-03 13:00:00-06
```

Ceci survient parce qu'une heure a été ignorée à cause d'un changement d'heure à `2005-04-03 02:00:00` pour le fuseau horaire `America/Denver`.

Notez qu'il peut avoir une ambiguïté dans le champ `months` renvoyé par la fonction `age` parce que des mois différents ont des numéros de jour différents. L'approche de PostgreSQL utilise le mois de la date la plus ancienne lors du calcul de mois partiels. Par exemple, `age('2004-06-01', '2004-04-30')` utilise avril pour renvoyer `1 mon 1 day`, alors que mai renverrait `1 mon 2 days` parce que mai a 31 jours alors qu'avril n'en a que 30.

La soustraction de dates et de timestamps peut aussi être complexe. Une façon simple conceptuellement de réaliser une soustraction est de convertir chaque valeur en un nombre de secondes en utilisant `EXTRACT(EPOCH FROM ...)`, puis de soustraire les résultats ; ceci donne un nombre de *secondes* entre les deux valeurs. Ceci provoquera un ajustement suivant le nombre de jours pour chaque mois, les changements de fuseaux horaires et les ajustements pour les changements d'heure. La soustraction de valeurs date ou timestamp avec l'opérateur « `-` » renvoie le nombre de jours (24 heures) et les heures/minutes/secondes entre les valeurs, en faisant les mêmes ajustements. La fonction `age` renvoie les années, mois, jours et heures/minutes/secondes, soustrayant champ par champ, puis ajustant les valeurs négatives. Les requêtes suivantes illustrent les différences dans ces approches. Les résultats correspondent au fuseau horaire `timezone = 'US/Eastern'` ; il y a un changement d'heure entre les deux dates utilisées :

```
SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
       EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');
Résultat : 10537200.000000
SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
        EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))
        / 60 / 60 / 24;
Résultat : 121.9583333333333333
SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01
 12:00:00';
Résultat : 121 days 23:00:00
SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz
 '2013-03-01 12:00:00');
Résultat : 4 mons
```

9.9.1. EXTRACT, `date_part`

```
EXTRACT(field FROM source)
```

La fonction `extract` récupère les champs tels que l'année ou l'heure de valeurs date/heure. *`source`* doit être l'expression d'une valeur de type `timestamp`, `date`, `time` ou `interval`. (Les données de type `timestamp` ou `time` peuvent être avec ou sans fuseau horaire.) *`field`* est un identifiant ou une chaîne qui sélectionne le champ à extraire à partir de la valeur source. Les champs ne sont pas tous valides pour chaque type de donnée en entrée ; par exemple, les champs plus petit qu'un jour ne peuvent pas être extraits à partir d'une donnée de type `date`, alors que les champs d'un jour ou plus ne peuvent pas être extraits d'un type `time`. La fonction `extract` renvoie des valeurs de type `numeric`. Voici les noms de champ valides :

`century`

Le siècle ; pour les valeurs de type `interval`, le champ year divisé par 100

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Résultat : 20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 21
SELECT EXTRACT(CENTURY FROM DATE '0001-01-01 AD');
```

```
Result: 1
SELECT EXTRACT(CENTURY FROM DATE '0001-12-31 BC');
Result: -1
SELECT EXTRACT(CENTURY FROM INTERVAL '2001 years');
Result: 20
```

`day`

Le jour du mois (1–31) ; pour les valeurs `interval`, le nombre de jours

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 16
SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
Résultat : 40
```

`decade`

Le champ année divisé par 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 200
```

`dow`

Le jour de la semaine du dimanche (`0`) au samedi (`6`)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 5
```

Notez que la numérotation du jour de la semaine d'après `extract` diffère de celle de la fonction `to_char(..., 'D')`.

`doy`

Le jour de l'année (1–365/366)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 47
```

`epoch`

Pour les valeurs de type `timestamp with time zone`, le nombre de secondes depuis le 1er janvier 1970 à minuit UTC. Ce nombre est négatif pour les timestamps avant cette valeur. Pour les valeurs de type `date` et `timestamp`, le nombre de secondes depuis le 1er janvier 1970 à minuit, heure locale, sans regard au fuseau horaire ou aux règles de changement d'heure. Pour les valeurs de type `interval`, le nombre total de secondes dans l'intervalle

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16
 20:38:40.12-08');
Résultat : 982384720.120000
SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40.12');
Résultat : 982355920.120000
```

```
SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Résultat : 4428000000
```

Vous pouvez convertir une valeur epoch en une valeur de type `timestamp with time zone` avec la fonction `to_timestamp` :

```
SELECT to_timestamp(982384720.12);
Résultat : 2001-02-17 04:38:40.12+00
```

Attention que l'application de `to_timestamp` à un epoch extrait d'une valeur `date` ou `timestamp` pourrait produire un résultat trompeur : le résultat supposera en fait que la valeur originale a été donnée en UTC, ce qui pourrait ne pas être le cas.

`hour`

Le champ heure (0–23 pour les données de type `timestamp`, sans contrainte pour les intervals)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 20
```

`isodow`

Le jour de la semaine du lundi (`1`) au dimanche (`7`)

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
Résultat : 7
```

Ceci est identique à `dow` sauf pour les dimanches. Ceci correspond à la numérotation du jour de la semaine d'après ISO 8601.

`isoyear`

L'année suivant la numérotation semaine ISO 8601 dans laquelle la date tombe

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
Résultat : 2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
Résultat : 2006
```

Chaque année suivant la numérotation semaine ISO 8601 commence le lundi de la semaine contenant le 4 janvier, donc un janvier débutant tôt ou un décembre finissant tard pourrait faire que l'année ISO soit différente de l'année grégorienne. Voir le champ `week` pour plus d'informations.

`julian`

La *Date Julien* correspondant à la date ou à l'horodatage. Les horodatages qui ne sont pas à minuit heure locale résultent en une valeur fractionnelle. Voir Section B.7 pour plus d'informations.

```
SELECT EXTRACT(JULIAN FROM DATE '2006-01-01');
Result: 2453737
SELECT EXTRACT(JULIAN FROM TIMESTAMP '2006-01-01 12:00');
Result: 2453737.50000000000000000000
```

`microseconds`

Le champ des secondes, incluant la partie fractionnelle, multiplié par 1 000 000 ; notez que ceci inclut des secondes complètes

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Résultat : 28500000
```

`millennium`

Le millénaire ; pour les valeurs de type `interval`, le champ year divisé par 1000

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 3
SELECT EXTRACT(MILLENNIUM FROM INTERVAL '2001 years');
Result: 2
```

Les années 1900 sont dans le deuxième millénaire. Le troisième millénaire commence le 1er janvier 2001.

`milliseconds`

Le champ des secondes, incluant la partie fractionnelle, multipliée par 1000. Notez que ceci inclut des secondes complètes.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Résultat : 28500.000
```

`minute`

Le champ des minutes (0–59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 38
```

`month`

Le numéro du mois dans l'année (1–12) ; pour les valeurs `interval`, le numéro du mois, modulo 12 (0–11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 2
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Résultat : 3
SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Résultat : 1
```

`quarter`

Le trimestre de l'année (1–4) dans laquelle la date se trouve

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
```

```
Résultat : 1
```

`second`

Le champs secondes, incluant toutes secondes fractionnelles

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 40.000000
SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Résultat : 28.500000
```

`timezone`

Le décalage du fuseau horaire, à partir d'UTC, mesuré en secondes. Les valeurs positives correspondent aux fuseaux horaires à l'est d'UTC, les valeurs négatives aux zones à l'ouest d'UTC. (Techniquement, PostgreSQL n'utilise pas UTC parce que les secondes perdues ne sont pas gérées.)

`timezone_hour`

Le composant heure du décalage du fuseau horaire

`timezone_minute`

Le composant minute du décalage du fuseau horaire

`week`

Le numéro de la semaine de cette année d'après la numérotation de semaine de l'ISO 8601. Par définition, les semaines ISO commencent les lundis et la première semaine de l'année contient le 4 janvier de l'année. Autrement dit, le premier jeudi d'une année est dans la semaine 1 de cette année.

Dans le système ISO de numérotation des semaines, il est possible que les dates de début janvier soient dans la semaine 52 ou 53 de l'année précédente, et pour les dates de fin décembre de faire partie de la première semaine de l'année suivante. Par exemple, le 1er janvier 2005 fait partie de la semaine 53 de l'année 2004, et le 1er janvier 2006 fait partie de la semaine 52 de l'année 2005, alors que le 31 décembre 2012 fait partie de la première semaine de 2013. Il est recommandé d'utiliser le champ `isoyear` avec `week` pour obtenir des résultats cohérents.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 7
```

`year`

Le champ année. Gardez en tête qu'il n'existe pas d'année `0 AD`, donc soustraire `BC` années de `AD` années doit se faire avec prudence.

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Résultat : 2001
```

Lors du traitement d'une valeur `interval`, la fonction `extract` produit des valeurs de champ qui correspondent à l'interprétation utilisée par la fonction interval en sortie. Ceci peut produire des résultats surprenants si l'une commence avec une représentation d'interval non normalisé. Par exemple :

```
SELECT INTERVAL '80 minutes';
Résultat : 01:20:00
SELECT EXTRACT(MINUTES FROM INTERVAL '80 minutes');
Résultat : 20
```

> **Note**
>
> Quand la valeur en entrée est +/-Infinity, `extract` renvoie +/-Infinity pour les champs à augmentation monotonique (`epoch`, `julian`, `year`, `isoyear`, `decade`, `century` et `millennium`). Pour les autres champs, NULL est renvoyé. Les versions de PostgreSQL avant la 9.6 renvoyaient zéro pour tous les cas si l'entrée est infinie.

La fonction `extract` a principalement pour intérêt un traitement calculé. Pour le formatage de valeurs date/heure en vue d'un affichage, voir Section 9.8.

La fonction `date_part` est modelée sur l'équivalent traditionnel d'Ingres pour la fonction `extract` du standard SQL :

```
date_part('field', source)
```

Notez qu'ici le paramètre *`field`* doit être une chaîne, et non pas un nom. Les noms de champs valides pour `date_part` sont les mêmes que pour `extract`. Pour des raisons historiques, la fonction `date_part` retourne des valeurs de type `double precision`. Cela peut entraîner une perte de précision dans certains cas. Il est recommandé d'utiliser `extract` à la place.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Résultat : 16
SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Résultat : 4
```

9.9.2. `date_trunc`

La fonction `date_trunc` est conceptuellement similaire à la fonction `trunc` pour les nombres.

```
date_trunc(field, source [, time_zone ])
```

`source` est une expression de valeur de type `timestamp`, `timestamp with time zone` ou `interval`. (Les valeurs de type `date` et `time` sont converties automatiquement, et respectivement, en `timestamp` ou `interval`) *`field`* sélectionne la précision pour le troncage de la valeur en entrée. La valeur de retour est de type `timestamp`, `timestamp with time zone` ou `interval`, et elle contient tous les champs qui sont moins significatifs que le champ sélectionné, qui est initialisé à zéro (ou un pour le jour et le mois).

Les valeurs valides pour *`field`* sont :

```
microseconds
milliseconds
second
minute
hour
day
week
month
```

```
quarter
year
decade
century
millennium
```

Quand la valeur en entrée est de type `timestamp with time zone`, la troncature est réalisée en respectant un fuseau horaire particulier ; par exemple, la troncature à `day` crée une valeur à minuit de ce fuseau. Par défaut, une troncature se fait en respectant le paramètre TimeZone mais l'argument optionnel *`time_zone`* peut être fourni pour indiquer un fuseau horaire différent. Le nom du fuseau horaire peut être indiqué parmi toutes les façons décrites dans Section 8.5.3.

Un fuseau horaire ne peut pas être indiqué lors du traitement de données `timestamp without time zone` ou `interval`. Ce sont toujours des valeurs prises directement, sans interprétation.

Exemples (en supposant que le fuseau horaire local est `America/New_York`) :

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Résultat : 2001-02-16 20:00:00
SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Résultat : 2001-01-01 00:00:00
SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16
 20:38:40+00');
Résultat : 2001-02-16 00:00:00-05
SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16
 20:38:40+00', 'Australia/Sydney');
Résultat : 2001-02-16 08:00:00-05
SELECT date_trunc('hour', INTERVAL '3 days 02:47:33');
Résultat : 3 days 02:00:00
```

9.9.3. `date_bin`

La fonction `date_bin` place l'instant en entrée au sein de l'intervalle spécifié (le *pas*), aligné avec une origine donnée.

```
date_bin(pas, source, origine)
```

`source` est une expression de type `timestamp` ou `timestamp with time zone`. (Les valeurs de type `date` sont automatiquement converties en `timestamp`.) *`pas`* est une expression de type `interval`. La valeur retournée est aussi de type `timestamp` ou `timestamp with time zone`, et marque le début de l'intervalle dans lequel *`source`* est placée.

Exemples :

```
SELECT date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17',
 TIMESTAMP '2001-01-01');
Result: 2020-02-11 15:30:00
SELECT date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17',
 TIMESTAMP '2001-01-01 00:02:30');
Result: 2020-02-11 15:32:30
```

Dans le cas d'unités rondes (1 minute, 1 heure, etc.), la fonction retourne le même résultat que l'analogue `date_trunc`, la différence étant que `date_bin` est capable de tronquer selon un intervalle arbitraire.

L'intervalle *`pas`* doit être plus grand que zéro et ne peut contenir ni des mois, ni des unités plus grandes.

9.9.4. AT TIME ZONE

L'opérateur `AT TIME ZONE` convertit l'horodatage *sans* sans fuseau horaire en horodatage *avec* fuseau horaire, et des valeurs `time with time zone` dans différents fuseaux horaires. Tableau 9.34 montre les variantes.

Tableau 9.34. Variantes `AT TIME ZONE`

Opérateur Description Exemple(s)
`timestamp without time zone` `AT TIME ZONE` *`zone`* → `timestamp with time zone` Convertit l'horodatage donné *sans* fuseau horaire en horodatage *avec* fuseau horaire en supposant que la valeur indiquée est dans le fuseau horaire nommé. `timestamp '2001-02-16 20:38:40' at time zone 'America/Denver'` → `2001-02-17 03:38:40+00`
`timestamp with time zone` `AT TIME ZONE` *`zone`* → `timestamp without time zone` Convertit l'horodatage donné *avec* fuseau horaire en horodatage *sans* sans fuseau horaire, comme si l'heure apparaissait dans ce fuseau. `timestamp with time zone '2001-02-16 20:38:40-05' at time zone 'America/Denver'` → `2001-02-16 18:38:40`
`time with time zone` `AT TIME ZONE` *`zone`* → `time with time zone` Convertit l'heure donnée *avec* fuseau horaire dans un nouveau fuseau horaire. Comme aucune date n'est fournie, cela utilise le décalage UTC actuellement actif pour le fuseau horaire de destination. `time with time zone '05:34:17-05' at time zone 'UTC'` → `10:34:17+00`

Dans ces expressions, le fuseau horaire désiré *`zone`* peut être spécifié soit sous la forme d'une valeur de type texte (par exemple `'America/Los_Angeles'`) soit comme un intervalle (par exemple `INTERVAL '-08:00'`). Dans le cas du texte, le nom du fuseau horaire peut être indiqué d'une des façons décrites dans Section 8.5.3. Dans le cas de l'intervalle, il est uniquement utile pour les fuseaux qui ont des décalages fixes d'UTC, donc ce n'est pas très courant.

Exemples (en supposant que le paramètre TimeZone a comme valeur actuelle `America/Los_Angeles`) :

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/
Denver';
Résultat : 2001-02-16 19:38:40-08
SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME
 ZONE 'America/Denver';
Résultat : 2001-02-16 18:38:40
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'Asia/Tokyo' AT
 TIME ZONE 'America/Chicago';
Résultat : 2001-02-16 05:38:40
```

Le premier exemple ajoute un fuseau horaire à une valeur qui en manque, et affiche la valeur en utilisant le paramétrage actuel de `TimeZone`. Le deuxième exemple décale l'horodatage avec fuseau horaire dans le fuseau horaire indiqué, et envoie la valeur sans fuseau horaire. Ceci permet le stockage et l'affichage de valeurs différentes de la configuration actuelle de `TimeZone`. Le troisième exemple convertit l'heure de Tokyo en heure de Chicago.

La fonction `timezone(`*`zone`*`, `*`timestamp`*`)` est équivalente à la construction conforme au standard SQL *`timestamp`* `AT TIME ZONE` *`zone`*.

9.9.5. Date/Heure actuelle

PostgreSQL fournit un certain nombre de fonctions qui renvoient des valeurs relatives à la date et l'heure actuelles. Ces fonctions SQL renvoient des valeurs basées sur l'heure de début de la transaction :

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

`CURRENT_TIME` et `CURRENT_TIMESTAMP` renvoient des valeurs avec fuseau horaire ; `LOCALTIME` et `LOCALTIMESTAMP` renvoient des valeurs sans fuseau horaire.

`CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME` et `LOCALTIMESTAMP` peuvent prendre en option un paramètre de précision qui impose que le résultat soit arrondi à ce nombre de chiffres dans le champ des secondes. Sans paramètre de précision, le résultat est donné avec toute la précision disponible.

Quelques exemples :

```
SELECT CURRENT_TIME;
Résultat : 14:39:53.662522-05
SELECT CURRENT_DATE;
Résultat : 2019-12-23
SELECT CURRENT_TIMESTAMP;
Résultat : 2019-12-23 14:39:53.662522-05
SELECT CURRENT_TIMESTAMP(2);
Résultat : 2019-12-23 14:39:53.66-05
SELECT LOCALTIMESTAMP;
Résultat : 2019-12-23 14:39:53.662522
```

Comme ces fonctions renvoient l'heure du début de la transaction en cours, leurs valeurs ne changent pas lors de la transaction courante. Ceci est considéré comme une fonctionnalité : le but est de permettre à une même transaction d'avoir une notion cohérente de l'heure « actuelle », pour que plusieurs modifications au sein de la même transactions arrivent au même moment grâce à un même horodatage.

> **Note**
>
> D'autres systèmes de bases de données pourraient mettre à jour ces valeurs plus fréquemment.

PostgreSQL fournit aussi des fonctions qui renvoient l'heure de début de la requête en cours, ainsi que l'heure actuelle au moment où la fonction est appelée. La liste complète des fonctions d'heure, ne faisant pas parti du standard SQL, est la suivante :

```
transaction_timestamp()
```

```
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

`transaction_timestamp()` est équivalent à `CURRENT_TIMESTAMP`, mais est nommée ainsi pour refléter clairement ce qu'il renvoie. `statement_timestamp()` renvoie l'heure de début de la requête en cours (plus spécifiquement, l'heure de réception du dernier message de commande du client). `statement_timestamp()` et `transaction_timestamp()` renvoient la même valeur lors de la première requête d'une transaction, mais pourrait différer lors des requêtes suivantes. `clock_timestamp()` renvoie l'heure actuelle, et de ce fait, sa valeur changera y compris au sein d'une même requête SQL. `timeofday()` est une fonction historique de PostgreSQL. Tout comme `clock_timestamp()`, elle renvoie l'heure actuelle, mais formatée sous la forme d'une chaîne de type `text` plutôt que sous la forme d'une valeur de type `timestamp with time zone`. `now()` est un équivalent historique de PostgreSQL pour `transaction_timestamp()`.

Tous les types de données date/heure acceptent aussi la valeur littérale spéciale `now` pour indiquer la date et l'heure actuelle (encore une fois, interprété comme l'heure de début de la transaction). De ce fait, les trois suivants renvoient tous le même résultat :

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now'; -- mais voir l'astuce ci-dessous
```

> **Astuce**
>
> Ne pas utiliser la troisième forme lors de la spécification d'une valeur à évaluer ultérieurement, par exemple dans une clause `DEFAULT` de la colonne d'une table. Le système convertirait `now` en valeur de type `timestamp` dès l'analyse de la constante. À chaque fois que la valeur par défaut est nécessaire, c'est l'heure de création de la table qui est alors utilisée. Les deux premières formes ne sont pas évaluées avant l'utilisation de la valeur par défaut, il s'agit d'appels de fonctions. C'est donc bien le comportement attendu, l'heure d'insertion comme valeur par défaut, qui est obtenu. (Voir aussi Section 8.5.1.4.)

9.9.6. Retarder l'exécution

Les fonctions suivants sont disponibles pour retarder l'exécution du traitement du serveur :

```
pg_sleep ( double precision )
pg_sleep_for ( interval )
pg_sleep_until ( timestamp with time zone )
```

`pg_sleep` cause la pause du processus de la session en cours pour le nombre indiqué de secondes. Un délai peut être spécifié avec une partie fractionnelle pour les secondes `pg_sleep_for` est une fonction pratique pour indiquer le temps de pause sous la forme d'un `interval`. `pg_sleep_until` est une autre fonction pratique pour indiquer une heure de réveil, à la place d'une durée de pause. Par exemple :

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

> **Note**
>
> La résolution réelle de l'intervalle de pause est spécifique à la plateforme ; 0,01 secondes est une valeur assez générale. La durée de pause sera au moins aussi longue que celle indiquée. Elle pourra être plus longue suivant des facteurs comme la charge du serveur. En particulier, `pg_sleep_until` ne garantie pas de se réveiller précisément à l'heure indiquée, mais elle ne se réveillera pas avant.

> **Avertissement**
>
> Assurez-vous que votre session ne détient pas plus de verrous que nécessaire lors de l'appel de `pg_sleep` ou une de ses variantes. Sinon, d'autres sessions pourraient avoir à attendre le processus en attente, ralentissant le système entier.

9.10. Fonctions de support d'enum

Pour les types enum (décrites dans Section 8.7), il existe différentes fonctions permettant une programmation propre sans codage en dur de valeurs particulières d'un type enum. Elles sont listées dans Tableau 9.35. Les exemples supposent un type enum créé ainsi :

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green',
 'blue', 'purple');
```

Tableau 9.35. Fonctions de support des enum

Fonction **Description** **Exemple(s)**
`enum_first ( anyenum ) → anyenum` Renvoie la première valeur d'un type enum en entrée. `enum_first(null::rainbow) → red`
`enum_last ( anyenum ) → anyenum` Renvoie la dernière valeur d'un type enum en entrée. `enum_last(null::rainbow) → purple`
`enum_range ( anyenum ) → anyarray` Renvoie toutes les valeurs du type enum en entrée dans un tableau trié. `enum_range(null::rainbow) → {red,orange,yellow, green,blue,purple}`
`enum_range ( anyenum, anyenum ) → anyarray` Renvoie l'intervalle entre deux valeurs enum données, comme un tableau trié. Les valeurs doivent être dans le même type enum. Si le premier paramètre est NULL, le résultat commencera avec la première valeur du type enum. Si le second paramètre est NULL, le résultat se terminera avec la dernière valeur du type enum. `enum_range('orange'::rainbow, 'green'::rainbow) → {orange,yellow,green}` `enum_range(NULL, 'green'::rainbow) → {red,orange, yellow,green}`

Fonction Description Exemple(s)
`enum_range('orange'::rainbow, NULL)` → `{orange,yellow,green,blue,purple}`

Notez que, sauf pour la forme de `enum_range` à deux arguments, ces fonctions ignorent la valeur spécifique fournie. Seul importe le type de donnée déclaré. Peut être fourni soit NULL soit une valeur spécifique du type de données déclaré, avec le même résultat. Il est plus fréquent d'appliquer ces fonctions à une colonne d'une table ou à un argument d'une fonction qu'un nom de type codé en dur, comme utilisé dans les exemples.

9.11. Fonctions et opérateurs de géométrie

Les types géométriques `point`, `box`, `lseg`, `line`, `path`, `polygon` et `circle` ont un large ensemble des fonctions et opérateurs de support natifs, affichés dans Tableau 9.36, Tableau 9.37 et Tableau 9.38.

Tableau 9.36. Opérateurs géométriques

Opérateur Description Exemple(s)
geometric_type + `point` → *geometric_type* Ajoute les coordonnées du deuxième `point` à celles du premier argument, réalisant ainsi la translation. Disponible pour les types `point`, `box`, `path`, `circle`. `box '(1,1),(0,0)' + point '(2,0)'` → `(3,1),(2,0)`
`path` + `path` → `path` Concatène deux chemins ouverts (renvoie NULL si un des chemins est fermé). `path '[(0,0),(1,1)]' + path '[(2,2),(3,3),(4,4)]'` → `[(0,0),(1,1),(2,2),(3,3),(4,4)]`
geometric_type - `point` → *geometric_type* Soustrait les coordonnées du deuxième `point` de celle de chaque point du premier argument, réalisant ainsi la translation. Disponible pour les types `point`, `box`, `path`, `circle`. `box '(1,1),(0,0)' - point '(2,0)'` → `(-1,1),(-2,0)`
geometric_type * `point` → *geometric_type* Multiplie chaque point du premier argument avec le second `point` (traitant un point comme étant un nombre complexe représenté par une partie réelle et une partie imaginaire, et réalisant ainsi une multiplication standard complexe). Si on interprète le deuxième `point` comme un vecteur, ceci est équivalent à mettre à l'échelle la taille et la distance de l'objet de l'origine par la longueur de la vecteur, et en faisant une rotation en sens inverse de l'horloge autour de l'origine par l'angle du vecteur sur l'axe *x*. Disponible pour les types `point`, `box`,[a] `path`, `circle`. `path '((0,0),(1,0),(1,1))' * point '(3.0,0)'` → `((0,0),(3,0),(3,3))` `path '((0,0),(1,0),(1,1))' * point(cosd(45), sind(45))` → `((0,0),(0.7071067811865475,0.7071067811865475),(0,1.414213562373095))`
geometric_type / `point` → *geometric_type* Divise chaque point du premier argument par le deuxième `point` (traitant un point comme étant un nombre complexe représenté par les parties réelle et imaginaire, et

Opérateur **Description** **Exemple(s)**
réalisant la division complexe standard). Si on interprète le deuxième `point` comme un vecteur, ceci est équivalent à mettre à l'échelle la taille et la distance de l'objet à partir de l'origine sur la longueur du vecteur et en faisant une rotation en sens inverse de l'horloge autour de l'origine par l'angle du vecteur sur l'axe *x*. Disponible pour les types `point`, `box`,[a] `path`, `circle`. `path '((0,0),(1,0),(1,1))' / point '(2.0,0)'` → `((0,0),(0.5,0),(0.5,0.5))` `path '((0,0),(1,0),(1,1))' / point(cosd(45), sind(45))` → `((0,0),(0.7071067811865476,-0.7071067811865476),(1.4142135623730951,0))`
`@-@` *`geometric_type`* → `double precision` Calcule la longueur totale. Disponible pour les types `lseg`, `path`. `@-@ path '[(0,0),(1,0),(1,1)]'` → `2`
`@@` *`geometric_type`* → `point` Calcule le point central. Disponible pour les types `box`, `lseg`, `path`, `polygon`, `circle`. `@@ box '(2,2),(0,0)'` → `(1,1)`
`#` *`geometric_type`* → `integer` Renvoie le nombre de points. Disponible pour les types `path`, `polygon`. `# path '((1,0),(0,1),(-1,0))'` → `3`
`geometric_type` `#` *`geometric_type`* → `point` Calcule le point d'intersection, ou NULL s'il n'y en a pas. Disponible pour les types `lseg`, `line`. `lseg '[(0,0),(1,1)]' # lseg '[(1,0),(0,1)]'` → `(0.5,0.5)`
`box` `#` `box` → `box` Calcule l'intersection de deux boîtes, ou NULL s'il n'y en a pas. `box '(2,2),(-1,-1)' # box '(1,1),(-2,-2)'` → `(1,1),(-1,-1)`
`geometric_type` `##` *`geometric_type`* → `point` Calcule le point le plus proche du premier objet à partir du deuxième objet. Disponible pour ces paires de types : (`point`, `box`), (`point`, `lseg`), (`point`, `line`), (`lseg`, `box`), (`lseg`, `lseg`), (`line`, `lseg`). `point '(0,0)' ## lseg '[(2,0),(0,2)]'` → `(1,1)`
`geometric_type` `<->` *`geometric_type`* → `double precision` Calcule la distance entre les objets. Disponible pour les sept types géométriques, pour toutes les combinaisons du `point` avec un autre type géométrique, et pour ces paires supplémentaires de types : (`box`, `lseg`), (`lseg`, `line`), (`polygon`, `circle`) (et les cas de commutation). `circle '<(0,0),1>' <-> circle '<(5,0),1>'` → `3`
`geometric_type` `@>` *`geometric_type`* → `boolean` Est-ce que le premier objet contient le second ? Disponible pour ces paires de types : (`box`, `point`), (`box`, `box`), (`path`, `point`), (`polygon`, `point`), (`polygon`, `polygon`), (`circle`, `point`), (`circle`, `circle`). `circle '<(0,0),2>' @> point '(1,1)'` → `t`
`geometric_type` `<@` *`geometric_type`* → `boolean`

Opérateur **Description** **Exemple(s)**
Est-ce que le premier objet est contenu dans le second ? Disponibles pour ces paires de types : (`point`, `box`), (`point`, `lseg`), (`point`, `line`), (`point`, `path`), (`point`, `polygon`), (`point`, `circle`), (`box`, `box`), (`lseg`, `box`), (`lseg`, `line`), (`polygon`, `polygon`), (`circle`, `circle`). `point '(1,1)' <@ circle '<(0,0),2>'` → `t`
`geometric_type` `&&` *`geometric_type`* → `boolean` Est-ce que ces objets se surchargent ? (Un point en commun rend ceci vrai.) Disponible pour `box`, `polygon`, `circle`. `box '(1,1),(0,0)' && box '(2,2),(0,0)'` → `t`
`geometric_type` `<<` *`geometric_type`* → `boolean` Est-ce que le premier objet est strictement à gauche du second ? Disponible pour `point`, `box`, `polygon`, `circle`. `circle '<(0,0),1>' << circle '<(5,0),1>'` → `t`
`geometric_type` `>>` *`geometric_type`* → `boolean` Est-ce que le premier objet est strictement à droite du second ? Disponible pour `point`, `box`, `polygon`, `circle`. `circle '<(5,0),1>' >> circle '<(0,0),1>'` → `t`
`geometric_type` `&<` *`geometric_type`* → `boolean` Est-ce que le premier objet ne s'étend pas à droite du second ? Disponible pour `box`, `polygon`, `circle`. `box '(1,1),(0,0)' &< box '(2,2),(0,0)'` → `t`
`geometric_type` `&>` *`geometric_type`* → `boolean` Est-ce que le premier objet ne s'étend pas à gauche du second ? Disponible pour `box`, `polygon`, `circle`. `box '(3,3),(0,0)' &> box '(2,2),(0,0)'` → `t`
`geometric_type` `<<|` *`geometric_type`* → `boolean` Est-ce que le premier objet se trouve strictement en dessous du second ? Disponible pour `point`, `box`, `polygon`, `circle`. `box '(3,3),(0,0)' <<| box '(5,5),(3,4)'` → `t`
`geometric_type` `|>>` *`geometric_type`* → `boolean` Est-ce que le premier objet se trouve strictement au-dessus du second ? Disponible pour `point`, `box`, `polygon`, `circle`. `box '(5,5),(3,4)' |>> box '(3,3),(0,0)'` → `t`
`geometric_type` `&<|` *`geometric_type`* → `boolean` Est-ce que le premier objet ne s'étend pas au-dessus du second ? Disponible pour `box`, `polygon`, `circle`. `box '(1,1),(0,0)' &<| box '(2,2),(0,0)'` → `t`
`geometric_type` `|&>` *`geometric_type`* → `boolean` Est-ce que le premier objet ne s'étend pas en dessous du second ? Disponible pour `box`, `polygon`, `circle`. `box '(3,3),(0,0)' |&> box '(2,2),(0,0)'` → `t`
`box <^ box` → `boolean` Est-ce que le premier objet est en-dessous du second (les coins peuvent se toucher) ?

Opérateur **Description** **Exemple(s)**
`box '((1,1),(0,0))' <^ box '((2,2),(1,1))'` → `t`
`box >^ box` → `boolean` Est-ce que le premier objet se trouve au dessus du second (les coins peuvent se toucher) ? `box '((2,2),(1,1))' >^ box '((1,1),(0,0))'` → `t`
`geometric_type` `?#` *`geometric_type`* → `boolean` Est-ce que ces objets s'intersectent ? Disponible pour ces paires de types : (`box`, `box`), (`lseg`, `box`), (`lseg`, `lseg`), (`lseg`, `line`), (`line`, `box`), (`line`, `line`), (`path`, `path`). `lseg '[(-1,0),(1,0)]' ?# box '(2,2),(-2,-2)'` → `t`
`?-` `line` → `boolean` `?-` `lseg` → `boolean` La lign est-elle horizontale ? `?- lseg '[(-1,0),(1,0)]'` → `t`
`point` `?-` `point` → `boolean` Les points sont-ils alignés horizontalement (autrement dit, même coordonnée y) ? `point '(1,0)' ?- point '(0,0)'` → `t`
`?|` `line` → `boolean` `?|` `lseg` → `boolean` La ligne est-elle verticale ? `?| lseg '[(-1,0),(1,0)]'` → `f`
`point` `?|` `point` → `boolean` Les points sont-ils alignés verticalement (autrement dit, même coordonnée x) ? `point '(0,1)' ?| point '(0,0)'` → `t`
`line` `?-|` `line` → `boolean` `lseg` `?-|` `lseg` → `boolean` Les lignes sont-elles perpendiculaires ? `lseg '[(0,0),(0,1)]' ?-| lseg '[(0,0),(1,0)]'` → `t`
`line` `?||` `line` → `boolean` `lseg` `?||` `lseg` → `boolean` Les lignes sont-elles parallèles ? `lseg '[(-1,0),(1,0)]' ?|| lseg '[(-1,2),(1,2)]'` → `t`
`geometric_type` `~=` *`geometric_type`* → `boolean` Ces objets sont-ils les mêmes ? Disponible pour `point`, `box`, `polygon`, `circle`. `polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'` → `t`

[a]Faire une « Rotation » d'une boîte avec ces opérateurs déplace seulement ses coins : la boîte est toujours considérée avoir des côtés parallèles sur les axes. De ce fait, la taille de la boîte n'est pas préservée, bien qu'une vraie rotation le ferait.

Attention

Notez que l'opérateur « same as », `~=`, représente la notion habituelle d'égalité pour les types `point`, `box`, `polygon` et `circle`. Certains types géométriques ont aussi un opérateur `=`

mais = compare seulement des *aires* égales. Les autres opérateurs de comparaison scalaires (<= et ainsi de suite), si disponibles pour ces types, comparent aussi des aires.

Note

Avant PostgreSQL 14, les opérateurs de comparaison au-dessus / en dessous `point <<| point` et `point |>> point` étaient respectivement appelés `<^` et `>^`. Ces noms sont toujours disponibles, mais sont dépréciés et seront éventuellement supprimés.

Tableau 9.37. Fonctions géométriques

Fonction Description Exemple(s)
`area ( geometric_type ) → double precision` Calcule l'aire. Disponible pour `box`, `path`, `circle`. Une entrée `path` doit être terminée, sinon NULL est renvoyé. De plus, si `path` s'intersecte lui-même, le résultat pourrait ne pas avoir de signification. `area(box '(2,2),(0,0)') → 4`
`center ( geometric_type ) → point` Calcule le point central. Disponible pour `box`, `lseg`, `polygon`, `circle`. `center(box '(1,2),(0,0)') → (0.5,1)`
`diagonal ( box ) → lseg` Extrait la diagonale de la boîte sous la forme d'un segment de ligne (identique à `lseg(box)`). `diagonal(box '(1,2),(0,0)') → [(1,2),(0,0)]`
`diameter ( circle ) → double precision` Calcule le diamètre d'un cercle. `diameter(circle '<(0,0),2>') → 4`
`height ( box ) → double precision` Calcule la taille verticale d'une boîte. `height(box '(1,2),(0,0)') → 2`
`isclosed ( path ) → boolean` Le chemin est-il fermé ? `isclosed(path '((0,0),(1,1),(2,0))') → t`
`isopen ( path ) → boolean` Le chemin est-il ouvert ? `isopen(path '[(0,0),(1,1),(2,0)]') → t`
`length ( geometric_type ) → double precision` Calcule la longueur totale. Disponible pour `lseg`, `path`. `length(path '((-1,0),(1,0))') → 4`
`npoints ( geometric_type ) → integer` Renvoie le nombre de points. Disponible pour `path`, `polygon`. `npoints(path '[(0,0),(1,1),(2,0)]') → 3`
`pclose ( path ) → path`

Fonction Description Exemple(s)
Convertit le chemin en un chemin fermé. `pclose(path '[(0,0),(1,1),(2,0)]')` → `((0,0),(1,1),(2,0))`
`popen` (`path`) → `path` Convertit le chemin en un chemin ouvert. `popen(path '((0,0),(1,1),(2,0))')` → `[(0,0),(1,1),(2,0)]`
`radius` (`circle`) → `double precision` Calcule le rayon du cercle. `radius(circle '<(0,0),2>')` → `2`
`slope` (`point`, `point`) → `double precision` Calcule la pente d'une ligne tracée entre les deux points. `slope(point '(0,0)', point '(2,1)')` → `0.5`
`width` (`box`) → `double precision` Calcule la taille horizontale d'une boîte. `width(box '(1,2),(0,0)')` → `1`

Tableau 9.38. Fonctions de conversion de type géométrique

Fonction Description Exemple(s)
`box` (`circle`) → `box` Calcule la boîte inscrite dans un cercle. `box(circle '<(0,0),2>')` → `(1.414213562373095,1.414213562373095), (-1.414213562373095,-1.414213562373095)`
`box` (`point`) → `box` Convertit le point en boîte vide. `box(point '(1,0)')` → `(1,0),(1,0)`
`box` (`point`, `point`) → `box` Convertit deux points (utilisés comme coins) en une boîte. `box(point '(0,1)', point '(1,0)')` → `(1,1),(0,0)`
`box` (`polygon`) → `box` Calcule la boîte englobante d'un polygone. `box(polygon '((0,0),(1,1),(2,0))')` → `(2,1),(0,0)`
`bound_box` (`box`, `box`) → `box` Calcule la boîte englobante de deux boîtes. `bound_box(box '(1,1),(0,0)', box '(4,4),(3,3)')` → `(4,4), (0,0)`
`circle` (`box`) → `circle` Calcule le plus petit cercle englobant une boîte. `circle(box '(1,1),(0,0)')` → `<(0.5,0.5),0.7071067811865476>`
`circle` (`point`, `double precision`) → `circle` Calcule le cercle à partir de son centre et de son rayon.

Fonction
Description
Exemple(s)

`circle(point '(0,0)', 2.0)` → `<(0,0),2>`

`circle ( polygon )` → `circle`
Convertit un poligône en cercle. Le centre du cercle est la moyenne des positions des points du polygone, et le rayon est la distance moyenne des points du polygone à partir de ce centre.
`circle(polygon '((0,0),(1,3),(2,0))')` → `<(1,1),1.6094757082487299>`

`line ( point, point )` → `line`
Convertit deux points en une ligne qui les traverse.
`line(point '(-1,0)', point '(1,0)')` → `{0,-1,0}`

`lseg ( box )` → `lseg`
Extrait la diagonale de la boîte en un segment de ligne.
`lseg(box '(1,0),(-1,0)')` → `[(1,0),(-1,0)]`

`lseg ( point, point )` → `lseg`
Construit le segment de ligne à partir de deux points finaux.
`lseg(point '(-1,0)', point '(1,0)')` → `[(-1,0),(1,0)]`

`path ( polygon )` → `path`
Convertit un polygone en un chemin fermé avec la même liste de points.
`path(polygon '((0,0),(1,1),(2,0))')` → `((0,0),(1,1),(2,0))`

`point ( double precision, double precision )` → `point`
Calcule un point à partir de ses coordonnées.
`point(23.4, -44.5)` → `(23.4,-44.5)`

`point ( box )` → `point`
Calcule le centre d'une boîte.
`point(box '(1,0),(-1,0)')` → `(0,0)`

`point ( circle )` → `point`
Calcule le centre d'un cercle.
`point(circle '<(0,0),2>')` → `(0,0)`

`point ( lseg )` → `point`
Calcule le centre d'un segment de ligne.
`point(lseg '[(-1,0),(1,0)]')` → `(0,0)`

`point ( polygon )` → `point`
Calcule le centre d'un polygone (la moyenne des positions des points du polygone).
`point(polygon '((0,0),(1,1),(2,0))')` → `(1,0.3333333333333333)`

`polygon ( box )` → `polygon`
Convertit une boîte en un polygone à 4 points.
`polygon(box '(1,1),(0,0)')` → `((0,0),(0,1),(1,1),(1,0))`

`polygon ( circle )` → `polygon`
Convertit un cercle en un polygone à 12 points.

Fonction **Description** **Exemple(s)**
`polygon(circle '<(0,0),2>')` → `((-2,0), (-1.7320508075688774,0.9999999999999999), (-1.0000000000000002,1.7320508075688772), (-1.2246063538223773e-16,2), (0.9999999999999996,1.7320508075688774), (1.732050807568877,1.0000000000000007), (2,2.4492127076447545e-16), (1.7320508075688776,-0.9999999999999994), (1.0000000000000009,-1.7320508075688767), (3.673819061467132e-16,-2), (-0.9999999999999987,-1.732050807568878), (-1.7320508075688767,-1.0000000000000009))`
`polygon ( integer, circle )` → `polygon` Convertit un cercle en un polygone à *n* points. `polygon(4, circle '<(3,0),1>')` → `((2,0),(3,1), (4,1.2246063538223773e-16),(3,-1))`
`polygon ( path )` → `polygon` Convertit un chemin fermé en un polygone avec la même liste de points. `polygon(path '((0,0),(1,1),(2,0))')` → `((0,0),(1,1),(2,0))`

Il est possible d'accéder aux deux nombres composants d'un `point` comme si le point était un tableau avec les index 0 et 1. Par exemple, si `t.p` est une colonne `point`, alors `SELECT p[0] FROM t` récupère la coordonnée X et `UPDATE t SET p[1] = ...` modifie la coordonnée Y. De la même façon, une valeur de type `box` ou `lseg` peut être traitée comme un tableau de deux valeurs `point`.

9.12. Fonctions et opérateurs pour les adresses réseau

Les types pour les adresses réseau IP, `cidr` et `inet`, supportent les opérateurs de comparaison habituels indiqués dans Tableau 9.1 ainsi que les opérateurs et fonctions spécialisés indiqués dans Tableau 9.39 et Tableau 9.40.

Toute valeur `cidr` peut être convertie implicitement en `inet` ; de ce fait, les opérateurs et fonctions indiqués ici comme travaillant sur des valeurs `inet` fonctionnent aussi sur des valeurs `cidr`. (Quand il y a des fonctions séparées pour `inet` et `cidr`, c'est dû au comportement différent pour les deux cas.) De plus, il est permis de convertir une valeur `inet` en une valeur `cidr`. Quand cela se fait, tout bit à droite du masque réseau est mis à zéro silencieusement pour créer une valeur `cidr` valide.

Tableau 9.39. Opérateurs d'adresse IP

Opérateur **Description** **Exemple(s)**
`inet <<` `inet` → `boolean` Le sous-réseau est-il strictement contenu dans l'autre sous-réseau ? Cet opérateur, et les quatre suivants, testent l'inclusion du sous-réseau. Ils considèrent seulement les parties réseau des deux adresses (ignorant tout bit à droite des masques réseau), et déterminent si un réseau est identique ou un sous-réseau de l'autre. `inet '192.168.1.5' << inet '192.168.1/24'` → `t` `inet '192.168.0.5' << inet '192.168.1/24'` → `f`

Opérateur **Description** **Exemple(s)**
`inet '192.168.1/24' << inet '192.168.1/24'` → `f`
`inet <<= inet` → `boolean` Est-ce que le sous-réseau est contenu dans ou égal au sous-réseau ? `inet '192.168.1/24' <<= inet '192.168.1/24'` → `t`
`inet >> inet` → `boolean` Est-ce que ce sous-réseau est contenu strictement dans le sous-réseau ? `inet '192.168.1/24' >> inet '192.168.1.5'` → `t`
`inet >>= inet` → `boolean` Est-ce que le sous-réseau contient ou est égal au sous-réseau ? `inet '192.168.1/24' >>= inet '192.168.1/24'` → `t`
`inet && inet` → `boolean` Est-ce qu'un des sous-réseaux contient ou est égale à l'autre ? `inet '192.168.1/24' && inet '192.168.1.80/28'` → `t` `inet '192.168.1/24' && inet '192.168.2.0/28'` → `f`
`~ inet` → `inet` Calcule un NOT bit à bit. `~ inet '192.168.1.6'` → `63.87.254.249`
`inet & inet` → `inet` Calcule un AND bit à bit. `inet '192.168.1.6' & inet '0.0.0.255'` → `0.0.0.6`
`inet | inet` → `inet` Calcule un OR bit à bit. `inet '192.168.1.6' | inet '0.0.0.255'` → `192.168.1.255`
`inet + bigint` → `inet` Ajoute un décalage à une adresse. `inet '192.168.1.6' + 25` → `192.168.1.31`
`bigint + inet` → `inet` Ajoute un décalage à une adresse. `200 + inet '::ffff:fff0:1'` → `::ffff:255.240.0.201`
`inet - bigint` → `inet` Soustrait un décalage à une adresse. `inet '192.168.1.43' - 36` → `192.168.1.7`
`inet - inet` → `bigint` Calcule la différence de deux adresses. `inet '192.168.1.43' - inet '192.168.1.19'` → `24` `inet '::1' - inet '::ffff:1'` → `-4294901760`

Tableau 9.40. Fonctions des adresses IP

Fonction **Description** **Exemple(s)**
`abbrev ( inet ) → text` Crée un format d'affichage abrévié sous la forme d'un texte. (Le résultat est le même que ce que produit la fonction de sortie du type `inet` ; c'est « abrévié » seulement en comparaison du résultat d'une conversion explicite vers du `text` qui, pour des raisons historiques, ne supprimera jamais la partie du masque réseau.) `abbrev(inet '10.1.0.0/32') → 10.1.0.0`
`abbrev ( cidr ) → text` Crée un format d'affichage abrévié sous la forme d'un texte. (L'abréviation consiste en la suppression des octets zéro à droite du masque réseau ; il y a plus d'exemples dans Tableau 8.22.) `abbrev(cidr '10.1.0.0/16') → 10.1/16`
`broadcast ( inet ) → inet` Calcule l'adresse de broadcast pour le réseau de l'adresse. `broadcast(inet '192.168.1.5/24') → 192.168.1.255/24`
`family ( inet ) → integer` Renvoie la famille de l'adresse : `4` pour IPv4, `6` pour IPv6. `family(inet '::1') → 6`
`host ( inet ) → text` Renvoie l'adresse IP sous forme de texte, en ignorant le masque réseau. `host(inet '192.168.1.0/24') → 192.168.1.0`
`hostmask ( inet ) → inet` Calcule le masque de l'hôte pour le réseau de l'adresse. `hostmask(inet '192.168.23.20/30') → 0.0.0.3`
`inet_merge ( inet, inet ) → cidr` Calcule le plus petit réseau qui inclut les deux réseaux donnés. `inet_merge(inet '192.168.1.5/24', inet '192.168.2.5/24') → 192.168.0.0/22`
`inet_same_family ( inet, inet ) → boolean` Teste si les adresses appartiennent à la même famille IP. `inet_same_family(inet '192.168.1.5/24', inet '::1') → f`
`masklen ( inet ) → integer` Renvoie la longueur du masque réseau en bits. `masklen(inet '192.168.1.5/24') → 24`
`netmask ( inet ) → inet` Calcule le masque réseau pour le réseau de l'adresse. `netmask(inet '192.168.1.5/24') → 255.255.255.0`
`network ( inet ) → cidr` Renvoie la partie réseau de l'adresse, mettant à zéro tout ce qui se trouve à droit du masque réseau. (Ceci est équivalent à convertir la valeur en `cidr`.) `network(inet '192.168.1.5/24') → 192.168.1.0/24`
`set_masklen ( inet, integer ) → inet`

Fonction Description Exemple(s)
Configure la longueur du masque réseau pour une valeur `inet`. La partie adresse ne change pas. `set_masklen(inet '192.168.1.5/24', 16)` → `192.168.1.5/16`
`set_masklen ( cidr, integer ) → cidr` Configure la longueur du masque réseau pour une valeur `cidr`. Les bits d'adresse à droit du nouveau masque réseau sont configurés à zéro. `set_masklen(cidr '192.168.1.0/24', 16)` → `192.168.0.0/16`
`text ( inet ) → text` Renvoie l'adresse IP non abrévié et la longueur du masque réseau en texte. (Ceci a le même résultat qu'une conversion explicite vers `text`.) `text(inet '192.168.1.5')` → `192.168.1.5/32`

> **Astuce**
>
> Les fonctions `abbrev`, `host` et `text` ont pour but principal d'offrir des formats d'affichage alternatifs pour les adresses IP.

Les types d'adresse MAC, `macaddr` et `macaddr8`, supportent les opérateurs de comparaison habituels indiqués dans Tableau 9.1 ainsi que les fonctions spécialisées indiquées dans Tableau 9.41. De plus, ils supportent les opérateurs logiques bit à bit `~`, `&` et `|` (NOT, AND et OR), comme indiqué ci-dessus pour les adresses IP.

Tableau 9.41. Fonctions pour les adresses MAC

Fonction Description Exemple(s)
`trunc ( macaddr ) → macaddr` Configure les trois derniers octets de l'adresse à zéro. Le préfixe restant peut être associé avec un manufacturier particulier (en utilisant les données non inclus dans PostgreSQL). `trunc(macaddr '12:34:56:78:90:ab')` → `12:34:56:00:00:00`
`trunc ( macaddr8 ) → macaddr8` Configure les 5 derniers octets de l'adresse à zéro. Le préfixe restant peut être associé à un manufacturier particulier (en utilisant des données non inclus dans PostgreSQL). `trunc(macaddr8 '12:34:56:78:90:ab:cd:ef')` → `12:34:56:00:00:00:00:00`
`macaddr8_set7bit ( macaddr8 ) → macaddr8` Configure le 7è bit de l'adresse à un, créant ce qui est connu comme un EUI-64 modifié, pour inclusion dans une adresse IPv6. `macaddr8_set7bit(macaddr8 '00:34:56:ab:cd:ef')` → `02:34:56:ff:fe:ab:cd:ef`

9.13. Fonctions et opérateurs de recherche plein texte

Tableau 9.42, Tableau 9.43 et Tableau 9.44 résument les fonctions et opérateurs fournis par la recherche plein texte. Voir Chapitre 12 pour une explication détaillée de la fonctionnalité de recherche de texte de PostgreSQL.

Tableau 9.42. Opérateurs de recherche plein texte

Opérateur **Description** **Exemple(s)**
`tsvector @@ tsquery → boolean` `tsquery @@ tsvector → boolean` Est-ce que le `tsvector` correspond au `tsquery` ? (Les arguments peuvent être donnés dans n'importe quel ordre.) `to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat') → t`
`text @@ tsquery → boolean` Est-ce que la chaîne de texte, après appel implicite à `to_tsvector()`, correspond au `tsquery` ? `'fat cats ate rats' @@ to_tsquery('cat & rat') → t`
`tsvector @@@ tsquery → boolean` `tsquery @@@ tsvector → boolean` Synonyme obsolète pour `@@`. `to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat') → t`
`tsvector || tsvector → tsvector` Concatène deux `tsvector`. Si les deux entrées contiennent des positions de lexème, les positions de la deuxième entrée sont ajustées. `'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector → 'a':1 'b':2,5 'c':3 'd':4`
`tsquery && tsquery → tsquery` Calcule un AND des deux `tsquery`, produisant une requête qui établit une correspondance avec les documents correspondant aux deux requêtes. `'fat | rat'::tsquery && 'cat'::tsquery → ( 'fat' | 'rat' ) & 'cat'`
`tsquery || tsquery → tsquery` Calcule un OR des deux `tsquery`, produisant une requête qui établit une correspondance avec les documents correspondant à au moins une des deux requêtes. `'fat | rat'::tsquery || 'cat'::tsquery → 'fat' | 'rat' | 'cat'`
`!! tsquery → tsquery` Inverse la `tsquery`, produisant une requête qui établit une correspondance avec les documents qui ne correspondent pas à la requête en entrée. `!! 'cat'::tsquery → !'cat'`
`tsquery <-> tsquery → tsquery` Construit une requête de phrase, qui établit une correspondance si les deux requêtes en entrée correspondent aux lexèmes successifs. `to_tsquery('fat') <-> to_tsquery('rat') → 'fat' <-> 'rat'`
`tsquery @> tsquery → boolean`

Opérateur **Description** **Exemple(s)**
Est-ce que le premier `tsquery` contient le second ? (Ceci considère seulement si tous les lexèmes apparaissant dans une requêtes apparaissent dans l'autre, ignorant les opérateurs combinés.) `'cat'::tsquery @> 'cat & rat'::tsquery → f`
`tsquery <@ tsquery → boolean` Est-ce que le premier `tsquery` est contenu dans le second ? (Ceci considère seulement si tous les lexèmes apparaissant dans une requête apparaissent aussi dans l'autre, ignorant les opérateurs combinés.) `'cat'::tsquery <@ 'cat & rat'::tsquery → t` `'cat'::tsquery <@ '!cat & rat'::tsquery → t`

En plus de ces opérateurs spécialisés, les opérateurs de comparaison habituels indiqués dans Tableau 9.1 sont disponibles pour les types `tsvector` et `tsquery`. Ils ne sont pas très utiles pour la recherche de texte mais permettent, par exemple, de construire des index d'unicité sur des colonnes de ce type.

Tableau 9.43. Fonctions de recherche plein texte

Fonction **Description** **Exemple(s)**
`array_to_tsvector ( text[] ) → tsvector` Convertit un tableau de chaînes de texte vers un `tsvector`. Les chaînes données sont utilisées comme lexemes, sans autre traitement. Les éléments du tableau ne doivent pas être des chaînes vides ou or `NULL`. `array_to_tsvector('{fat,cat,rat}'::text[]) → 'cat' 'fat' 'rat'`
`get_current_ts_config ( ) → regconfig` Renvoie l'OID de la configuration actuelle de recherche plein texte (comme elle est configurée par default_text_search_config). `get_current_ts_config() → english`
`length ( tsvector ) → integer` Renvoie le nombre de lexèmes dans le `tsvector`. `length('fat:2,4 cat:3 rat:5A'::tsvector) → 3`
`numnode ( tsquery ) → integer` Renvoie le nombre de lexèmes plus les opérateurs dans le `tsquery`. `numnode('(fat & rat) | cat'::tsquery) → 5`
`plainto_tsquery ( [ config regconfig, ] query text ) → tsquery` Convertit le texte en un `tsquery`, normalisant les mots suivant la configuration spécifiée ou par défaut. Toute ponctuation dans la chaîne est ignorée (elle ne détermine pas les opérateurs de la requête). La requête résultante correspond aux documents contenant tous les mots du texte qui ne sont pas des mots à ignorer. `plainto_tsquery('english', 'The Fat Rats') → 'fat' & 'rat'`
`phraseto_tsquery ( [ config regconfig, ] query text ) → tsquery` Convertit du texte en `tsquery`, normalisant les mots suivant la configuration spécifiée ou par défaut. Toute ponctuation dans la chaîne est ignorée (elle ne détermine pas les

Fonction **Description** **Exemple(s)**
opérateurs de requête). La requête résultante correspond aux phrases contenant tous les mots du texte qui ne sont pas des mots à ignorer. `phraseto_tsquery('english', 'The Fat Rats')` → `'fat' <-> 'rat'` `phraseto_tsquery('english', 'The Cat and Rats')` → `'cat' <2> 'rat'`
`websearch_to_tsquery` ([*`config`* `regconfig`,] *`query`* `text`) → `tsquery` Convertit du texte en un `tsquery`, normalisant des mots suivant la configuration spécifiée ou par défaut. Les séquences de mots entre guillemets sont converties pour phraser des tests. Le mot « or » est compris comme produisant un opérateur OR, et un tiret produit un opérateur NOT ; les autres ponctuations sont ignorées. Ceci ressemble au comportement des outils de recherche web. `websearch_to_tsquery('english', '"fat rat" or cat dog')` → `'fat' <-> 'rat' | 'cat' & 'dog'`
`querytree` (`tsquery`) → `text` Produit une représentation de portion indexable d'un `tsquery`. Un résultat qui est vide ou juste `T` indique une requête non indexable. `querytree('foo & ! bar'::tsquery)` → `'foo'`
`setweight` (*`vector`* `tsvector`, *`weight`* `"char"`) → `tsvector` Assigne un poids (*`weight`*) spécifié pour chaque élément d'un *`vector`*. `setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')` → `'cat':3A 'fat':2A,4A 'rat':5A`
`setweight` (*`vector`* `tsvector`, *`weight`* `"char"`, *`lexemes`* `text[]`) → `tsvector` Assigne le poids (*`weight`*) spécifié aux éléments des *`vector`* qui sont listés dans les *`lexemes`*. Les chaînes dans *`lexemes`* sont prises ainsi, sans autre traitement. Les chaînes qui ne correspondent à aucun lexeme dans *`vector`* sont ignorées. `setweight('fat:2,4 cat:3 rat:5,6B'::tsvector, 'A', '{cat,rat}')` → `'cat':3A 'fat':2,4 'rat':5A,6A`
`strip` (`tsvector`) → `tsvector` Supprime les positions et les poids à partir d'un `tsvector`. `strip('fat:2,4 cat:3 rat:5A'::tsvector)` → `'cat' 'fat' 'rat'`
`to_tsquery` ([*`config`* `regconfig`,] *`query`* `text`) → `tsquery` Convertit le texte en `tsquery`, normalisant les mots suivant la configuration spécifiée ou par défaut. Les mots doivent être combinés par des opérateurs `tsquery` valides. `to_tsquery('english', 'The & Fat & Rats')` → `'fat' & 'rat'`
`to_tsvector` ([*`config`* `regconfig`,] *`document`* `text`) → `tsvector` Convertit du texte en un `tsvector`, normalisant les mots suivant la configuration spécifiée ou par défaut. L'information de position est inclus dans le résultat. `to_tsvector('english', 'The Fat Rats')` → `'fat':2 'rat':3`
`to_tsvector` ([*`config`* `regconfig`,] *`document`* `json`) → `tsvector` `to_tsvector` ([*`config`* `regconfig`,] *`document`* `jsonb`) → `tsvector` Convertit chaque valeur chaîne du document JSON en un `tsvector`, normalisant les mots suivant la configuration spécifiée ou par défaut. Les résultats sont ensuite concaténées dans l'ordre du document pour produire la sortie. L'information de position est générée comme si un mot d'arrêt existe chaque paire de valeurs chaînes.

Fonction **Description** **Exemple(s)**
(Attention que l'« ordre du document » des champs d'un objet JSON est dépendant de l'implémentation quand l'entrée est de type `jsonb` ; observez la difference dans les exemples.) `to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::json)` → `'dog':5 'fat':2 'rat':3` `to_tsvector('english', '{"aa": "The Fat Rats", "b": "dog"}'::jsonb)` → `'dog':1 'fat':4 'rat':5`
`json_to_tsvector` ([*`config`* `regconfig`,] *`document`* `json`, *`filter`* `jsonb`) → `tsvector` `jsonb_to_tsvector` ([*`config`* `regconfig`,] *`document`* `jsonb`, *`filter`* `jsonb`) → `tsvector` Sélectionne chaque élément du document JSON qui est réclamé par le *`filter`* et convertit chacun en un `tsvector`, normalisant les mots suivant la configuration spécifiée ou par défaut. Les résultats sont ensuite concaténées dans l'ordre du document pour produire la sortie. L'information de position est générée comme si un mot d'arrêt existe entre chaque paire d'éléments sélectionnés. (Attention que l'« ordre du document » des champs d'un objet JSON est dépendant de l'implémentation quand l'entrée est de type `jsonb`.) Le *`filter`* doit être un tableau `jsonb` contenant zéro ou plusieurs de ces mots-clés : `"string"` (pour inclure toutes les valeurs chaînes), `"numeric"` (pour inclure toutes les valeurs numériques), `"boolean"` (pour inclure toutes les valeurs booléennes), `"key"` (pour inclure toutes les clés) ou `"all"` (pour tous ces mots-clés). Comme exception, *`filter`* peut aussi être une simple valeur JSON qui est un de ces mots-clés. `json_to_tsvector('english', '{"a": "The Fat Rats", "b": 123}'::json, '["string", "numeric"]')` → `'123':5 'fat':2 'rat':3` `json_to_tsvector('english', '{"cat": "The Fat Rats", "dog": 123}'::json, '"all"')` → `'123':9 'cat':1 'dog':7 'fat':4 'rat':5`
`ts_delete` (*`vector`* `tsvector`, *`lexeme`* `text`) → `tsvector` Supprime toute occurrence du *`lexeme`* donné à partir de *`vector`*. La chaîne dans *`lexeme`* est prise ainsi, sans autre traitement. `ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat')` → `'cat':3 'rat':5A`
`ts_delete` (*`vector`* `tsvector`, *`lexemes`* `text[]`) → `tsvector` Supprime toute occurrence des lexèmes dans *`lexemes`* à partir de *`vector`*. Les chaînes dans *`lexemes`* sont prises ainsi, sans autre traitement. Les chaînes qui ne correspondent à aucun lexeme dans *`vector`* sont ignorées. `ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat'])` → `'cat':3`
`ts_filter` (*`vector`* `tsvector`, *`weights`* `"char"[]`) → `tsvector` Sélectionne seulement les éléments avec le *`weights`* donné à partir de *`vector`*. `ts_filter('fat:2,4 cat:3b,7c rat:5A'::tsvector, '{a,b}')` → `'cat':3B 'rat':5A`
`ts_headline` ([*`config`* `regconfig`,] *`document`* `text`, *`query`* `tsquery` [, *`options`* `text`]) → `text` Affiche, dans sa forme abréviée, les correspondances de la *`query`* dans le *`document`*, qui doit être un texte brut, et non pas un `tsvector`. Les mots dans le document

Fonction **Description** **Exemple(s)**
sont normalisés suivant la configuration spécifiée ou par défaut avant d'établir les correspondances avec la requête. L'utilisation de cette fonction est discutée dans Section 12.3.4, quid écrit aussi les *options* disponibles. `ts_headline('The fat cat ate the rat.', 'cat')` → `The fat <b>cat</b> ate the rat.`
`ts_headline` ([*config* `regconfig`,] *document* `json`, *query* `tsquery` [, *options* `text`]) → `text` `ts_headline` ([*config* `regconfig`,] *document* `jsonb`, *query* `tsquery` [, *options* `text`]) → `text` Affiche, dans une forme abréviée, les correspondances pour la *query* qui surviennent dans les valeurs de type chaîne dans le *document* JSON. Voir Section 12.3.4 pour plus de détails. `ts_headline('{"cat":"raining cats and dogs"}'::jsonb, 'cat')` → `{"cat": "raining <b>cats</b> and dogs"}`
`ts_rank` ([*weights* `real[]`,] *vector* `tsvector`, *query* `tsquery` [, *normalization* `integer`]) → `real` Calcule un score indiquant le niveau de correspondance du *vector* avec la *query*. Voir Section 12.3.3 pour plus de détails. `ts_rank(to_tsvector('raining cats and dogs'), 'cat')` → `0.06079271`
`ts_rank_cd` ([*weights* `real[]`,] *vector* `tsvector`, *query* `tsquery` [, *normalization* `integer`]) → `real` Calcule un score indiquant le niveau de correspondance du *vector* avec la *query*, en utilisant un algorithme de densité de couverture. Voir Section 12.3.3 pour les détails. `ts_rank_cd(to_tsvector('raining cats and dogs'), 'cat')` → `0.1`
`ts_rewrite` (*query* `tsquery`, *target* `tsquery`, *substitute* `tsquery`) → `tsquery` Remplace les occurrences de *target* avec *substitute* dans *query*. Voir Section 12.4.2.1 pour les détails. `ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery)` → `'b' & ( 'foo' | 'bar' )`
`ts_rewrite` (*query* `tsquery`, *select* `text`) → `tsquery` Remplace les portions de la *query* suivant les cibles et substituts obtenus en exécutant une commande `SELECT`. Voir Section 12.4.2.1 pour les détails. `SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')` → `'b' & ( 'foo' | 'bar' )`
`tsquery_phrase` (*query1* `tsquery`, *query2* `tsquery`) → `tsquery` Construit une requête de phrases pour rechercher les correspondances de *query1* et *query2* sur des lexèmes successifs (identique à l'opérateur `<->`). `tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'))` → `'fat' <-> 'cat'`
`tsquery_phrase` (*query1* `tsquery`, *query2* `tsquery`, *distance* `integer`) → `tsquery` Construit une requête de phrases pour rechercher les correspondances de *query1* et *query2* qui surviennent exactement entre *distance* lexèmes.

Fonction **Description** **Exemple(s)**
`tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10)` → `'fat' <10> 'cat'`
`tsvector_to_array(tsvector)` → `text[]` Convertit un `tsvector` en un tableau de lexèmes. `tsvector_to_array('fat:2,4 cat:3 rat:5A'::tsvector)` → `{cat,fat,rat}`
`unnest(tsvector)` → `setof record(`*`lexeme`* `text,`*`positions`* `smallint[],`*`weights`* `text)` Étend un `tsvector` en un ensemble de lignes, une par lexème. `select * from unnest('cat:3 fat:2,4 rat:5A'::tsvector)` → `lexeme | positions | weights` `--------+-----------+---------` `cat | {3} | {D}` `fat | {2,4} | {D,D}` `rat | {5} | {A}`

> **Note**
>
> Toutes les fonctions de recherche plein texte qui acceptent un argument `regconfig` optionnel utiliseront la configuration indiquée par default_text_search_config quand cet argument est omis.

Les fonctions de Tableau 9.44 sont listées séparément parce qu'elles ne sont habituellement pas utilisées dans les opérations standard de recherche de texte. Elles sont principalement utiles pour le développement et le débogage des nouvelles configurations de recherche plein texte.

Tableau 9.44. Fonctions de débogage de recherche plein texte

Fonction **Description** **Exemple(s)**
`ts_debug([`*`config`* `regconfig,]`*`document`* `text)` → `setof record(`*`alias`* `text,`*`description`* `text,`*`token`* `text,`*`dictionaries`* `regdictionary[],`*`dictionary`* `regdictionary,`*`lexemes`* `text[])` Extrait et normalise les jetons d'un *`document`* suivant la configuration de recherche plein texte spécifié ou par défaut, et renvoie l'information sur la façon dont chaque jeton a été traité. Voir Section 12.8.1 pour plus de détails. `ts_debug('english', 'The Brightest supernovaes')` → `(asciiword,"Word, all ASCII",The,` `{english_stem},english_stem,{}) ...`
`ts_lexize(`*`dict`* `regdictionary,`*`token`* `text)` → `text[]` Renvoie un tableau de lexèmes de remplacement si le jeton en entrée est connu par le dictionnaire ou un tableau suivant si le jeton est connu du dictionnaire mais est un mot vide, ou NULL si ce n'est pas un mot connu. Voir Section 12.8.3 pour les détails. `ts_lexize('english_stem', 'stars')` → `{star}`

Fonction **Description** **Exemple(s)**
`ts_parse(`*`parser_name`* `text,` *`document`* `text) → setof record(`*`tokid`* `integer,` *`token`* `text)` Extrait les jetons du *`document`* en utilisant l'analyseur nommé. Voir Section 12.8.2 pour les détails. `ts_parse('default', 'foo - bar') → (1,foo) ...`
`ts_parse(`*`parser_oid`* `oid,` *`document`* `text) → setof record(`*`tokid`* `integer,` *`token`* `text)` Extrait les jetons du *`document`* en utilisant un analyseur spécifié par OID. Voir Section 12.8.2 pour les détails. `ts_parse(3722, 'foo - bar') → (1,foo) ...`
`ts_token_type(`*`parser_name`* `text) → setof record(`*`tokid`* `integer,` *`alias`* `text,` *`description`* `text)` Renvoie une table qui décrit chaque type de jeton que l'analyseur nommé peut reconnaître. Voir Section 12.8.2 pour les détails. `ts_token_type('default') → (1,asciiword,"Word, all ASCII") ...`
`ts_token_type(`*`parser_oid`* `oid) → setof record(`*`tokid`* `integer,` *`alias`* `text,` *`description`* `text)` Renvoie une table qui décrit chaque type de jeton qu'un analyseur spécifié par OID peut reconnaître. Voir Section 12.8.2 pour les détails. `ts_token_type(3722) → (1,asciiword,"Word, all ASCII") ...`
`ts_stat(`*`sqlquery`* `text [,` *`weights`* `text ]) → setof record(`*`word`* `text,` *`ndoc`* `integer,` *`nentry`* `integer)` Exécute la *`sqlquery`*, qui doit renvoyer une seule colonne `tsvector`, et renvoyer des statistiques sur chaque lexème distinct contenu dans la donnée. Voir Section 12.4.4 pour les détails. `ts_stat('SELECT vector FROM apod') → (foo,10,15) ...`

9.14. Fonctions UUID

PostgreSQL inclut une fonction pour générer un UUID :

```
gen_random_uuid () → uuid
```

Cette fonction renvoie un UUID version 4 (aléatoire). Ceci est le type le plus communément utilisé d'UUID et est approprié pour la plupart des applications.

Le module uuid-ossp fournit des fonctions supplémentaires implémentant d'autres algorithmes standards de génération d'UUID.

PostgreSQL fournit aussi les opérateurs de comparaison habituels affichés dans Tableau 9.1 pour les UUID.

9.15. Fonctions XML

Les fonctions et les expressions décrites dans cette section opèrent sur des valeurs de type `xml`. Voir Section 8.13 pour des informations sur le type `xml`. Les expressions `xmlparse` et `xmlserialize` pour convertir vers ou à partir d'un type `xml` sont documentées ici, et non pas dans cette section.

L'utilisation de la plupart de ces fonctions requiert que PostgreSQL soit construit avec `configure --with-libxml`.

9.15.1. Produire du contenu XML

Un ensemble de fonctions et d'expressions est disponible pour produire du contenu XML à partir de données SQL. Elles sont donc tout particulièrement intéressantes pour formater le résultat de requêtes en des documents XML pour un traitement dans des applications clientes.

9.15.1.1. `xmlcomment`

```
xmlcomment ( text ) → xml
```

La fonction `xmlcomment` crée une valeur XML contenant un commentaire XML avec le texte spécifié comme contenu. Le texte ne peut pas contenir « `--` » ou se terminer avec un « `-` », sinon la construction résultante serait un commentaire XML invalide. Si l'argument est NULL, le résultat est NULL.

Example:

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

9.15.1.2. `xmlconcat`

```
xmlconcat ( xml [, ...] ) → xml
```

La fonction `xmlconcat` concatène une liste de valeurs XML individuelles pour créer une seule valeur contenant un fragment de contenu XML. Les valeurs NULL sont omises. Le résultat est NULL si tous les arguments sont NULL.

Example:

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

Les déclarations XML sont combinées ainsi. Si toutes les valeurs des arguments ont la même déclaration de version XML, cette version est utilisée comme résultat. Sinon, aucune version n'est utilisée. Si toutes les valeurs des arguments ont une valeur de déclarationl standalone à « yes », alors cette valeur est utilisée dans le résultat. Si toutes les valeurs d'agument ont une valeur de déclaration standalone et qu'au moins une de ces valeurs est « no », alors cette valeur est utilisée dans le résultat. Dans tous les autres cas, le résultat n'aura pas de déclaration standalone. Si le résultat nécessite

une déclaration standalone, mais par de déclaration version, une déclaration version 1.0 sera utilisée car XML requiert qu'une déclaration XML contiennent une déclaration de version. Les déclarations d'encodage sont ignorés et supprimés dans tous les cas.

Example:

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml
 version="1.1" standalone="no"?><bar/>');

            xmlconcat
----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

9.15.1.3. `xmlelement`

```
xmlelement ( NAME name [, XMLATTRIBUTES ( attvalue [ AS attname ]
 [, ...] ) ] [, content [, ...]] ) → xml
```

L'expression `xmlelement` produit un élément XML avec le nom, les attributs et le contenu donnés. Les éléments *`name`* et *`attname`* affichés dans la syntaxe sont de simples identifiants, et non pas des valeurs. Les éléments *`attvalue`* et *`content`* sont des expressions, qui peuvent être de n'importe quel type de données PostgreSQL. Les arguments dans `XMLATTRIBUTES` génèrent les attributs de l'élément XML ; les valeurs *`content`* sont concaténées pour former son contenu.

Examples:

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

    xmlelement
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar),
 'cont', 'ent');

             xmlelement
-------------------------------------
 <foo bar="2007-01-26">content</foo>
```

Les noms d'éléments et d'attributs qui ne sont pas des noms XML valides sont échappés en remplaçant les caractères problématiques par la séquence _x*HHHH*_, où *HHHH* est le point code Unicode du caractère dans sa notation hexadécimale. Par exemple :

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

            xmlelement
----------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

Un nom d'attribut explicite n'a pas besoin d'être spécifié si la valeur de l'attribut est une référence de colonne, auquel cas le nom de la colonne sera utilisé comme nom d'attribut par défaut. Dans les autres cas, l'attribut doit se voir donner un nom explicite. Donc cet exemple est valide :

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

Mais ces exemples ne le sont pas :

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM
 test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

Le contenu d'un élément, si indiqué, sera formaté suivant ton type de données. Si le contenu est lui-même de type `xml`, des documents XML complexes peuvent être construits. Par exemple :

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                            xmlelement(name abc),
                            xmlcomment('test'),
                            xmlelement(name xyz));

                  xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

Le contenu des autres types sera formaté en une donnée caractère XML valide. Ceci signifie en particulier que les caractères <, >, et & seront convertis en entités. Les données binaires (type de données `bytea`) seront représentées dans un encodage base64 ou hex, suivant la configuration du paramètre xmlbinary. Le comportement particulier pour les types de données individuels devrait évoluer pour aligner les correspondances PostgreSQL avec ceux spécifiés dans SQL:2006 et ultérieurs, comme discuté dans Section D.3.1.3.

9.15.1.4. `xmlforest`

```
xmlforest ( content [ AS name ] [, ...] ) → xml
```

L'expression `xmlforest` produit une forêt (séquence) XML d'éléments en utilisant les noms et le contenu donnés. Quant à `xmlelement`, chaque *`name`* doit être un simple identifiant, alors que les expressions *`content`* peuvent avoir n'importe quel type de données.

Exemples :

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

          xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>


SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
```

```
WHERE table_schema = 'pg_catalog';

                                  xmlforest
-----------------------------------
-----------------------------------
 <table_name>pg_authid</table_name><column_name>rolname</
column_name>
 <table_name>pg_authid</table_name><column_name>rolsuper</
column_name>
 ...
```

Comme indiqué dans le deuxième exemple, le nom de l'élément peut être omis si la valeur contenue est une référence de colonne, auquel cas le nom de l'élément est utilisé par défaut. Sinon un nom doit être spécifié.

Les noms d'élément qui ne sont pas des noms XML valides sont échappés comme indiqué pour `xmlelement` ci-dessus. De façon similaire, les données du contenu sont échappées pour rendre le contenu XML valide, sauf s'il est déjà de type `xml`.

Notez que les forêts XML ne sont pas des documents XML valides si elles consistent de plus d'un élément, donc il pourrait être utile d'envelopper les expressions `xmlforest` dans `xmlelement`.

9.15.1.5. `xmlpi`

```
xmlpi ( NAME name [, content ] ) → xml
```

L'instruction `xmlpi` crée une instruction de traitement XML. Quant à `xmlelement`, *`name`* doit être un simple identifiant, alors que l'expression *`content`* peut contenir tout type de données. Si présent, *`content`* ne doit pas contenir la séquence de caractères `?>`.

Exemple :

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

9.15.1.6. `xmlroot`

```
xmlroot ( xml, VERSION {text|NO VALUE} [, STANDALONE {YES|NO|NO
 VALUE} ] ) → xml
```

L'expression `xmlroot` altère les propriétés du nœud racine d'une valeur XML. Si une version est spécifiée, elle remplace la valeur dans la déclaration de version du nœud racine ; si un paramètre standalone est spécifié, il remplace la valeur dans la déclaration standalone du nœud racine.

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?
><content>abc</content>'),
               version '1.0', standalone yes);

                xmlroot
```

```
---------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

9.15.1.7. xmlagg

```
xmlagg ( xml ) → xml
```

La fonction xmlagg est, contrairement aux autres fonctions décrites ici, une fonction d'agrégat. Elle concatène les valeurs en entrée à l'appel de fonction d'agrégat, tout comme le fait xmlconcat, sauf que la concaténation survient au travers des lignes plutôt qu'au travers des expressions dans une seule ligne. Voir Section 9.21 pour l'information supplémentaire sur les fonctions d'agrégat.

Exemple :

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
        xmlagg
----------------------
 <foo>abc</foo><bar/>
```

Pour déterminer l'ordre de la concaténation, une clause ORDER BY peut être ajouté à l'appel de l'agrégat comme décrit dans Section 4.2.7. Par exemple :

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

L'approche non standard suivante était recommendée dans les versions précédentes et pourrait toujours être utile dans des cas spécifiques :

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

9.15.2. Prédicats XML

Les expressions décrites dans cette section vérifient les propriétés des valeurs xml.

9.15.2.1. IS DOCUMENT

```
xml IS DOCUMENT → boolean
```

L'expression IS DOCUMENT renvoie true si la valeur de l'argument XML est un document XML propre, false si elle ne l'est pas (c'est-à-dire un fragment de contenu), ou null si l'argument est null. Voir Section 8.13 sur la différence entre les documents et les fragments de contenu.

9.15.2.2. IS NOT DOCUMENT

```
xml IS NOT DOCUMENT → boolean
```

L'expression `IS NOT DOCUMENT` renvoie false si la valeur de l'argument XML est un document XML propre, true si elle ne l'est pas (c'est-à-dire un fragment de contenu), ou null si l'argument est null.

9.15.2.3. `XMLEXISTS`

```
XMLEXISTS ( text PASSING [BY {REF|VALUE}] xml [BY
 {REF|VALUE}] ) → boolean
```

La fonction `xmlexists` évalue une expression XPath 1.0 (le premier argument), avec la valeur XML passée comme son élément de contexte. La fonction renvoie false si le résultat de cette évaluation renvoie un ensemble de nœud vide, true si elle renvoie toute autre valeur. La fonction renvoie null si un argument vaut null. Une valeur non null passée comme élément de contexte doit être un document XML, et non pas un fragment de contenu ou une valeur non XML.

Example:

```
SELECT xmlexists('//town[text() = ''Toronto'']' PASSING BY VALUE
 '<towns><town>Toronto</town><town>Ottawa</town></towns>');

 xmlexists
------------
 t
(1 row)
```

Les clauses `BY REF` et `BY VALUE` sont acceptées dans PostgreSQL, mais sont ignorées comme indiqué dans Section D.3.2.

Dans le standard SQL, la fonction `xmlexists` évalue une expression dans le langage XML Query, mais PostgreSQL autorise seulement une expression XPath 1.0, comme discuté dans Section D.3.1.

9.15.2.4. `xml_is_well_formed`

```
xml_is_well_formed ( text ) → boolean
xml_is_well_formed_document ( text ) → boolean
xml_is_well_formed_content ( text ) → boolean
```

Ces fonctions vérifient si une chaîne de type `text` représente un XML bien formé, renvoyant un résultat booléen. `xml_is_well_formed_document` vérifie si le document est bien formé alors que `xml_is_well_formed_content` vérifie que c'est une contenu bien formé. `xml_is_well_formed` fait le premier si le paramètre de configuration xmloption est configuré à `DOCUMENT`, et le deuxième s'il est configuré à `CONTENT`. Ceci signifie que `xml_is_well_formed` est utile pour voir si une simple conversion vers le type `xml` réussira, alors que les deux autres fonctions sont utiles pour voir si les variantes correspondantes de `XMLPARSE` réussira.

Exemples :

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://
postgresql.org/stuff">bar</pg:foo>');
 xml_is_well_formed_document
-----------------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://
postgresql.org/stuff">bar</my:foo>');
 xml_is_well_formed_document
-----------------------------
 f
(1 row)
```

Le dernier exemple montre que les vérifications incluent une vérification de correspondance sur les espaces de nom.

9.15.3. Traiter du XML

Pour traiter les valeurs de type `xml`, PostgreSQL offre les fonctions `xpath` et `xpath_exists`, qui évaluent les expressions XPath 1.0 et la fonction de table `XMLTABLE`.

9.15.3.1. `xpath`

```
xpath ( xpath text, xml xml [, nsarray text[] ] ) → xml[]
```

La fonction `xpath` évalue l'expression XPath 1.0 indiquée dans *`xpath`* (sous la forme d'une donnée de type text) avec la valeur XML de type *`xml`*. Elle renvoie des valeurs XML correspondant à l'ensemble de nœuds produits par l'expression XPath. Si l'expression XPath renvoie une valeur scalaire plutôt qu'un ensemble de nœuds, un tableau d'un seul élément est renvoyé.

Le deuxième argument doit être un document XML bien formé. En particulier, il doit avoir un seul élément de nœud racine.

Le troisième argument optionnel de la fonction est un tableau de correspondances d'espaces de nom. Ce tableau devrait être un tableau de `text` à deux dimensions avec la longueur du deuxième axe égale

à 2 (cela devrait être un tableau de tableaux, chacun des deux contenant exactement deux éléments). Le premier élément de chaque entrée de tableau est le nom de l'espace de noms (alias), le second est l'URI de l'espace de noms. Il n'est pas requis que les alias fournis dans ce tableau soient identiques à ceux utilisés dans le document XML lui-même (en d'autres termes, dans le document XML et dans le contexte de la fonction `xpath` function context, les alias sont *local*).

Exemple :

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://
example.com">test</my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

Pour gérer les espaces de noms par défaut (anonymes), faire ceci :

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://
example.com"><b>test</b></a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

9.15.3.2. `xpath_exists`

```
xpath_exists ( xpath text, xml xml [, nsarray text[] ] ) → boolean
```

La fonction `xpath_exists` est une forme spécialisée de la fonction `xpath`. Au lieu de renvoyer les valeurs XML individuelles satisfaisant l'expression XPath 1.0, cette fonction renvoie un booléen indiquant si la requête était satisfaite ou non (spécifiquement s'il produit une valeur autre qu'un ensemble vide de nœuds). Cette fonction est équivalente au prédicat `XMLEXISTS`, sauf qu'il offre aussi du support pour l'argument de correspondance d'espaces de noms.

Example:

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://
example.com">test</my:a>',
                    ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
--------------
 t
(1 row)
```

9.15.3.3. `xmltable`

```
XMLTABLE (
    [ XMLNAMESPACES ( namespace_uri AS namespace_name [, ...] ), ]
```

```
    row_expression PASSING [BY {REF|VALUE}] document_expression [BY
 {REF|VALUE}]
    COLUMNS name { type [PATH column_expression]
 [DEFAULT default_expression] [NOT NULL | NULL]
                  | FOR ORDINALITY }
            [, ...]
) → setof record
```

L'expression `xmltable` produit une table basée sur une valeur XML, un filtre XPath pour extraire les lignes et un ensemble de définition de colonnes. Bien que cela ressemble syntaxiquement à une fonction, il peut aussi apparaître comme une table dans la clause `FROM` de la requête.

La clause `XMLNAMESPACES` optionnelle donne une liste, séparée par des virgules, des définitions d'espaces de nom, où chaque *`namespace_uri`* est une expression `text` et chaque *`namespace_name`* est un simple identifiant. Elle spécifie les espaces de nom XML utilisés dans le document et leur alias. La spécification d'un espace de nom par défaut n'est actuellement pas supportée.

L'argument requis *`row_expression`* est une expression XPath 1.0 (donné sous la forme d'un `text`) qui est évaluée, passant la valeur XML *`document_expression`* sous la forme d'un élément de contexte, pour obtenir un ensemble de nœuds XML. Ces nœuds sont ce que `xmltable` transforme des lignes en sortie. Aucune ligne ne sera produite si *`document_expression`* est NULL, ou si *`row_expression`* produit un ensemble vide de nœuds ou tout autre valeur qu'un ensemble de nœuds.

`document_expression` fournit l'élément de contexte pour *`row_expression`*. Cela doit être un document XML bien formé ; les fragments/forêts ne sont pas acceptés. Les clauses `BY REF` et `BY VALUE` sont acceptées mais ignorées, comme discuté dans Section D.3.2.

Dans le standard SQL, la fonction `xmltable` évalue des expressions dans le langage XML Query, mais PostgreSQL autorise seulement les expressions XPath 1.0, comme discutées dans Section D.3.1.

La clause requise `COLUMNS` spécifie les colonnes produites dans la table en sortie. Voir le résumé de la syntaxe ci-dessus pour le format. Un nom est requis pour chaque colonne, comme l'est un type de données (sauf si `FOR ORDINALITY` est spécifié, auquel cas le type `integer` est implicite). Les clauses de chemin, de valeur par défaut et de nullabilité sont optionnelles.

Une colonne marquée `FOR ORDINALITY` sera peuplée avec les numéros de lignes, commençant par 1, dans l'ordre des nœuds récupérés à partir de l'ensemble de nœuds résultats de *`row_expression`*. Au plus une colonne sera marquée `FOR ORDINALITY`.

> **Note**
>
> XPath 1.0 ne spécifie pas d'ordre pour les nœuds dans un ensemble de nœuds, donc le code qui relie un ordre particulier de résultats sera dépendant de l'implémentation. Les détails sont disponibles dans Section D.3.1.2.

Le *`column_expression`* pour un colonne est une expression XPath 1.0 qui est évaluée pour chaque ligne, avec le nœud courant provenant du résultat de *`row_expression`* comme son élément de contexte, pour trouver la valeur de la colonne. Si aucune *`column_expression`* n'est donnée, alors le nom de la colonne est utilisé comme un chemin implicite.

Si une expression XPath de colonne renvoie une valeur non XML (qui est limitée à une chaîne, un booléen ou un double en XPath 1.0) et la colonne a un type type autre que `xml`, la colonne sera configurée ainsi en affectant la représentation textuelle de la valeur vers le type PostgreSQL. (Si la valeur est un booléen, sa représentation textuelle est comprise comme `1` ou `0` si la catégorie du type de la colonne en sortie est numérique, sinon `true` ou `false`.)

Si l'expression XPath d'une colonne renvoie un ensemble non vide de nœuds XML et si le type PostgreSQL de la colonne est `xml`, la colonne sera affectée au résultat exact de l'expression s'il est de forme document ou contenu. [2]

Un résultat non XML affecté à une colonne `xml` en sortie produit du contenu, un nœud texte simple avec la valeur chaîne du résultat. Un résultat XML affecté à une colonne de tout autre type pourrait ne pas avoir plus d'un nœud, sinon une erreur est levée. S'il y a exactement un nœud, la colonne sera configurée d'après son affectation de la valeur textuelle du nœud (comme défini pour la fonction `string` XPath 1.0) du type PostgreSQL.

La valeur de chaîne d'un élément XML est la concaténation, dans l'ordre du document, de tous les nœuds texte contenus dans cet élément et ses descendants. La valeur de chaîne d'un élément sans nœuds texte descendants est une chaîne vide (non `NULL`). Tous les attributs `xsi:nil` sont ignorés. Notez que le nœud `text()` avec uniquement des espaces blancs entre deux éléments non-texte est préservé et que l'espace blanc au début du nœud `text()` n'est pas aplati. La fonction XPath 1.0 `string` peut être consultée pour les règles définissant la valeur de chaîne des autres types de nœuds XML et des valeurs non XML.

Les règles de conversion présentées ici ne sont pas exactement ceux du standard SQL, comme discuté dans Section D.3.1.3.

Si l'expression de chemin renvoie un ensemble vide de nœuds (typiquement quand cela ne correspond pas) pour une ligne donnée, la colonne sera configurée à `NULL`, sauf si une *`default_expression`* est indiquée ; alors la valeur résultant de l'évaluation d'une expression est utilisée.

Une *`default_expression`*, plutôt qu'être évaluée immédiatement quand `xmltable` est appelée, est évaluée chaque fois qu'une valeur par défaut est nécessaire pour la colonne. Si l'expression qualifie comme stable ou immuable, l'évaluation de répétition peut être ignorée. Ceci signifie que vous pouvez utiliser utilement les fonctions volatiles comme `nextval` dans *`default_expression`*.

Les colonnes peuvent être marquées `NOT NULL`. Si *`column_expression`* pour une colonne `NOT NULL` ne correspond à rien et qu'il n'y a ni valeur par défaut (`DEFAULT`) ni *`default_expression`*, elle s'évalue à NULL, une erreur est rapportée.

Exemples :

```
CREATE TABLE xmldata AS SELECT
xml $$
<ROWS>
  <ROW id="1">
    <COUNTRY_ID>AU</COUNTRY_ID>
    <COUNTRY_NAME>Australia</COUNTRY_NAME>
  </ROW>
  <ROW id="5">
    <COUNTRY_ID>JP</COUNTRY_ID>
    <COUNTRY_NAME>Japan</COUNTRY_NAME>
    <PREMIER_NAME>Shinzo Abe</PREMIER_NAME>
    <SIZE unit="sq_mi">145935</SIZE>
  </ROW>
  <ROW id="6">
    <COUNTRY_ID>SG</COUNTRY_ID>
    <COUNTRY_NAME>Singapore</COUNTRY_NAME>
    <SIZE unit="sq_km">697</SIZE>
  </ROW>
</ROWS>
```

[2] Un résultat contenant plus d'un nœud élément au niveau haut ou du texte sans espace blanc à l'extérieur d'un élément, est un exemple de forme contenu. Un résultat XPath peut être de toute forme, par exemple s'il renvoie un nœud attribut sélectionné à partir de l'élément qui le contient. Un tel résultat sera placé dans sa forme contenu avec chacun des nœuds non autorisés replacés par sa valeur de chaîne de texte, comme définis pour la fonction `string` XPath 1.0.

```
$$ AS data;

SELECT xmltable.*
  FROM xmldata,
       XMLTABLE('//ROWS/ROW'
                PASSING data
                COLUMNS id int PATH '@id',
                        ordinality FOR ORDINALITY,
                        "COUNTRY_NAME" text,
                        country_id text PATH 'COUNTRY_ID',
                        size_sq_km float PATH 'SIZE[@unit =
 "sq_km"]',
                        size_other text PATH
                             'concat(SIZE[@unit!="sq_km"], " ",
 SIZE[@unit!="sq_km"]/@unit)',
                        premier_name text PATH 'PREMIER_NAME'
 DEFAULT 'not specified');

 id | ordinality | COUNTRY_NAME | country_id | size_sq_km |
 size_other  | premier_name
----+------------+--------------+------------+------------
+--------------+--------------
  1 |          1 | Australia    | AU         |            |
      | not specified
  5 |          2 | Japan        | JP         |            | 145935
 sq_mi | Shinzo Abe
  6 |          3 | Singapore    | SG         |        697 |
      | not specified
```

L'exemple suivant montre une concaténation de plusieurs nœuds text(), une utilisation du nom de colonne en tant que filtre XPath, et le traitement des espaces blancs, des commentaires XML et le traitement des instructions :

```
CREATE TABLE xmlelements AS SELECT
xml $$
  <root>
   <element>  Hello<!-- xyxxz -->2a2<?aaaaa?> <!--x-->  bbb<x>xxx</
x>CC  </element>
        </root>
        $$ AS data;

        SELECT xmltable.*
        FROM xmlelements, XMLTABLE('/root' PASSING data COLUMNS
 element text);
        element
        ------------------------
        Hello2a2   bbbxxxCC
```

L'exemple suivant illustre comment la clause `XMLNAMESPACES` peut être utilisé pour spécifier une liste d'espaces de noms utilisée dans le document XML ainsi que dans les expressions XPath :

```
WITH xmldata(data) AS (VALUES ('
<example xmlns="http://example.com/myns" xmlns:B="http://
example.com/b">
 <item foo="1" B:bar="2"/>
 <item foo="3" B:bar="4"/>
```

```
 <item foo="4" B:bar="5"/>
</example>'::xml)
)
SELECT xmltable.*
  FROM XMLTABLE(XMLNAMESPACES('http://example.com/myns' AS x,
                              'http://example.com/b' AS "B"),
                '/x:example/x:item'
                   PASSING (SELECT data FROM xmldata)
                   COLUMNS foo int PATH '@foo',
                     bar int PATH '@B:bar');
 foo | bar
-----+-----
   1 |   2
   3 |   4
   4 |   5
(3 rows)
```

9.15.4. Correspondances tables et XML

Les fonctions suivantes font correspondre le contenu des tables relationnelles aux valeurs XML. Elles peuvent être vues comme une fonctionnalité d'export du XML :

```
table_to_xml ( table regclass, nulls boolean,
               tableforest boolean, targetns text ) → xml
query_to_xml ( query text, nulls boolean,
               tableforest boolean, targetns text ) → xml
cursor_to_xml ( cursor refcursor, count integer, nulls boolean,
                tableforest boolean, targetns text ) → xml
```

`table_to_xml` fait correspondre le contenu de la table nommée, passée en paramètre sous le nom de *`table`*. Le type `regclass` accepte les chaînes identifiant les tables en utilisant la notation habituelle incluant une qualification optionnelle avec le nom du schéma et les guillemets doubles (voir Section 8.19 pour les détails). `query_to_xml` exécute la requête dont le texte est passé avec le paramètre *`query`* et fait la correspondance avec le résultat. `cursor_to_xml` récupère le nombre indiqué de lignes à partir du curseur indiqué par le paramètre *`cursor`*. Cette variante est recommandée si des tables volumineuses sont utilisées car la valeur de résultat est construire en mémoire par chaque fonction.

Si *`tableforest`* vaut false, alors le document XML résultant ressemble à ceci :

```
<tablename>
  <row>
    <columnname1>data</columnname1>
    <columnname2>data</columnname2>
  </row>

  <row>
    ...
  </row>

  ...
</tablename>
```

Si *`tableforest`* vaut true, le résultat est un fragment de contenu XML qui ressemble à ceci :

```
<tablename>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
</tablename>

<tablename>
  ...
</tablename>

...
```

Si aucun nom de table n'est disponible, par exemple pour une requête ou un curseur, la chaîne indiquée pour `table` est utilisée pour le premier format, et `row` pour le deuxième format.

Le choix entre ces formats dépend de l'utilisateur. Le premier format est un document XML propre, qui sera important pour de nombreuses applications. Le deuxième format tend à être plus utile dans la fonction `cursor_to_xml` si les valeurs du résultat sont à réassembler en un seul document plus tard. Les fonctions pour produire du contenu XML discutées ci-dessus, en particulier `xmlelement`, peuvent être utilisées pour modifier les résultats.

Les valeurs des données sont projetées de la même façon que décrit pour la fonction `xmlelement` ci-dessus.

Le paramètre *`nulls`* détermine si les valeurs NULL doivent être incluses dans la sortie. Si true, les valeurs NULL dans les colonnes sont représentées sous cette forme :

```
<columnname xsi:nil="true"/>
```

où `xsi` est le préfixe de l'espace de nom XML pour XML Schema Instance. Une déclaration appropriée d'espace de nom sera ajoutée à la valeur résultat. Si false, les colonnes contenant des valeurs NULL seront simplement omises dans la sortie.

Le paramètre *`targetns`* spécifie l'espace de nom XML désiré pour le résultat. Si aucun espace de nom n'est souhaité, une chaîne vide doit être passée.

Les fonctions suivantes renvoient les documents XML Schema décrivant les correspondances réalisées par les fonctions ci-dessus :

```
table_to_xmlschema ( table regclass, nulls boolean,
                     tableforest boolean, targetns text ) → xml
query_to_xmlschema ( query text, nulls boolean,
                     tableforest boolean, targetns text ) → xml
cursor_to_xmlschema ( cursor refcursor, nulls boolean,
                      tableforest boolean, targetns text ) → xml
```

Il est essentiel que les mêmes paramètres soient passés dans l'ordre pour obtenir les projections de données XML correspondantes et les documents XML Schema.

Les fonctions suivantes produisent les projections de données XML correspondantes et les documents XML Schema dans un document (ou forêt), liés ensemble. Elles peuvent être utiles lorsque des résultats auto contenus et auto descriptifs sont désirés :

```
table_to_xml_and_xmlschema ( table regclass, nulls boolean,
                             tableforest boolean, targetns text
 ) → xml
```

```
query_to_xml_and_xmlschema ( query text, nulls boolean,
                             tableforest boolean, targetns text
 ) → xml
```

De plus, les fonctions suivantes sont disponibles pour produire des projections analogues de schémas entiers ou de la base de donnée courante entière :

```
schema_to_xml ( schema name, nulls boolean,
                tableforest boolean, targetns text ) → xml
schema_to_xmlschema ( schema name, nulls boolean,
                      tableforest boolean, targetns text ) → xml
schema_to_xml_and_xmlschema ( schema name, nulls boolean,
                              tableforest boolean, targetns text
 ) → xml

database_to_xml ( nulls boolean,
                  tableforest boolean, targetns text ) → xml
database_to_xmlschema ( nulls boolean,
                        tableforest boolean, targetns text ) → xml
database_to_xml_and_xmlschema ( nulls boolean,
                                tableforest boolean, targetns text
 ) → xml
```

Ces fonctions ignorent les tables qui ne sont pas lisibles par l'utilisateur courant. Les fonctions sur la base entière ignorent en plus les schémas pour lesquels l'utilisateur actuel ne dispose pas du droit USAGE.

Notez qu'elles peuvent produire beaucoup de données, entièrement stockées en mémoire. Lors de demandes de création de contenu à partir de gros schémas ou de base, il peut être intéressant de réfléchir à traiter les tables séparément, potentiellement même via un curseur.

Le résultat d'un contenu de schéma ressemble à ceci :

```
<schemaname>

table1-mapping

table2-mapping

...

</schemaname>
```

où le format de correspondance de table dépend du paramètre *tableforest* comme expliqué ci-dessus.

Le résultat d'un contenu de base ressemble à ceci :

```
<dbname>

<schema1name>
  ...
```

```
</schema1name>

<schema2name>
  ...
</schema2name>

...

</dbname>
```

où la correspondance de schéma est indiquée ci-dessus.

Comme exemple d'utilisation de la sortie de ces fonctions, Exemple 9.1 montre une feuille de style XSLT qui convertit la sortie de `table_to_xml_and_xmlschema` en un document HTML contenant une représentation en tableau des données de la table. D'une façon similaire, le résultat de ces fonctions peut être converti en d'autres formats basés sur le XML.

Exemple 9.1. Feuille de style XSLT pour convertir un résultat SQL/XML en HTML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/
xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
                  select="$schema/xsd:complexType[@name=
$tabletypename]/xsd:sequence/xsd:element[@name='row']/@type"/>

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=
$rowtypename]/xsd:sequence/xsd:element/@name">
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
```

```
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

9.16. Fonctions et opérateurs JSON

Cette section décrit :

- les fonctions et opérateurs pour traiter et créer des données JSON
- le langage de chemin SQL/JSON

Pour en savoir plus sur le standard SQL/JSON, voir [sqltr-19075-6]. Pour des détails sur les types JSON supportés dans PostgreSQL, voir Section 8.14.

9.16.1. Traiter et créer des données JSON

Tableau 9.45 affiche les opérateurs disponibles pour les types de données JSON (voir Section 8.14). De plus, les opérateurs de comparaison habituels indiqués dans Tableau 9.1 sont disponibles pour le type `jsonb`, mais pas pour le type `json`. Les opérateurs de comparaison suivent les règles de tri des opérateurs B-tree indiqués dans Section 8.14.4. Voir aussi Section 9.21 pour la fonction d'agrégat `json_agg` qui agrège les valeurs d'enregistrements sous la forme d'un JSON, pour la fonction d'agrégat `json_object_agg` qui agrège des paires de valeurs dans un objet JSON, et leurs équivalents `jsonb`, à savoir `jsonb_agg` et `jsonb_object_agg`.

Tableau 9.45. Opérateurs `json` et `jsonb`

Opérateur **Description** **Exemple(s)**
`json -> integer → json` `jsonb -> integer → jsonb` Extrait le *n*-ième élément d'un tableau JSON (les éléments du tableau sont indexés à partir de zéro, mais les nombres négatifs sont pris en compte à partir de la fin). `'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> 2 → {"c":"baz"}` `'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json -> -3 → {"a":"foo"}`
`json -> text → json` `jsonb -> text → jsonb` Extrait le champ objet JSON avec la clé donnée. `'{"a": {"b":"foo"}}'::json -> 'a' → {"b":"foo"}`
`json ->> integer → text` `jsonb ->> integer → text` Extrait le *n*-ième élément d'un tableau JSON, comme le `text`. `'[1,2,3]'::json ->> 2 → 3`

Opérateur **Description** **Exemple(s)**
`json ->> text → text` `jsonb ->> text → text` Extrait le champ objet JSON d'après la clé donnée, comme `text`. `'{"a":1,"b":2}'::json ->> 'b' → 2`
`json #> text[] → json` `jsonb #> text[] → jsonb` Extrait le sous-objet JSON d'un chemin spécifié, où les éléments du chemin peuvent être soit les clés d'un champ, soit les indices d'un tableau. `'{"a": {"b": ["foo","bar"]}}'::json #> '{a,b,1}' → "bar"`
`json #>> text[] → text` `jsonb #>> text[] → text` Extrait le sous-objet JSON au chemin spécifié avec `text`. `'{"a": {"b": ["foo","bar"]}}'::json #>> '{a,b,1}' → bar`

> **Note**
>
> Les opérateurs d'extraction champ/élément/chemin renvoient NULL, plutôt qu'une erreur, si l'entrée JSON n'a pas la bonne structure correspondant à la requête ; par exemple si la clé ou l'élément du tableau n'existe pas.

Quelques opérateurs supplémentaires existent seulement pour le type `jsonb`, comme indiqué dans Tableau 9.46. Section 8.14.4 décrit comment ces opérateurs peuvent être utilisés pour rechercher efficacement les données `jsonb` indexées.

Tableau 9.46. Opérateurs `jsonb` supplémentaires

Opérateur **Description** **Exemple(s)**
`jsonb @> jsonb → boolean` Est-ce que la première valeur JSON contient la seconde ? (Voir Section 8.14.3 pour les détails sur la notion de contenu.) `'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb → t`
`jsonb <@ jsonb → boolean` Est-ce que la première valeur JSON est contenue dans la seconde ? `'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb → t`
`jsonb ? text → boolean` Est-ce que la chaîne de caractères existe comme clé de haut niveau ou élément de tableau dans la valeur JSON ? `'{"a":1, "b":2}'::jsonb ? 'b' → t` `'["a", "b", "c"]'::jsonb ? 'b' → t`
`jsonb ?| text[] → boolean` Est-ce qu'une des chaînes du tableau de texte existe comme clé de haut niveau ou comme élément de tableau ? `'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'd'] → t`

Opérateur **Description** **Exemple(s)**
`jsonb ?& text[] → boolean` Est-ce que toutes les chaînes du tableau de texte existent comme clés de haut niveau ou comme éléments de tableau ? `'["a", "b", "c"]'::jsonb ?& array['a', 'b'] → t`
`jsonb || jsonb → jsonb` Concatène deux valeurs `jsonb`. Concaténer deux tableaux génère un tableau contenant tous les éléments de chaque entrée. Concaténer deux objets génère un objet contenant l'union de leurs clés, en prenant la valeur du deuxième objet quand il existe deux clés dupliquées. Tous les autres cas sont traités en convertissant une entrée non tableau en un tableau à un seul élément, puis en le traitant comme pour deux tableaux. N'est pas récursif : seul le tableau ou la structure objet de haut niveau est assemblé. `'["a", "b"]'::jsonb || '["a", "d"]'::jsonb → ["a", "b", "a", "d"]` `'{"a": "b"}'::jsonb || '{"c": "d"}'::jsonb → {"a": "b", "c": "d"}` `'[1, 2]'::jsonb || '3'::jsonb → [1, 2, 3]` `'{"a": "b"}'::jsonb || '42'::jsonb → [{"a": "b"}, 42]` Pour concaténer un tableau à un autre tableau dans une entrée unique, il faut l'envelopper dans une couche supplémentaire de tableau, par exemple : `'[1, 2]'::jsonb || jsonb_build_array('[3, 4]'::jsonb) → [1, 2, [3, 4]]`
`jsonb - text → jsonb` Supprime une clé (et sa valeur) à partir d'un objet JSON, ou les valeurs correspondantes de chaînes à partir d'un tableau JSON. `'{"a": "b", "c": "d"}'::jsonb - 'a' → {"c": "d"}` `'["a", "b", "c", "b"]'::jsonb - 'b' → ["a", "c"]`
`jsonb - text[] → jsonb` Supprime toutes les clés ou tous les éléments de tableau correspondant à partir de l'opérande gauche. `'{"a": "b", "c": "d"}'::jsonb - '{a,c}'::text[] → {}`
`jsonb - integer → jsonb` Supprime l'élément de tableau pour l'index spécifié (les nombres négatifs se décomptent à partir de la fin). Renvoie une erreur si la valeur JSON n'est pas un tableau. `'["a", "b"]'::jsonb - 1 → ["a"]`
`jsonb #- text[] → jsonb` Supprime le champ ou l'élément de tableau sur le chemin indiqué, les éléments du chemin pouvant être soit des clés de champ, soit des indices de tableau. `'["a", {"b":1}]'::jsonb #- '{1,b}' → ["a", {}]`
`jsonb @? jsonpath → boolean` Est-ce que le chemin JSON renvoie tout élément pour la valeur JSON spécifiée ? `'{"a":[1,2,3,4,5]}'::jsonb @? '$.a[*] ? (@ > 2)' → t`
`jsonb @@ jsonpath → boolean` Renvoie le résultat d'une vérification de prédicat du chemin JSON pour la valeur JSON indiquée. Seul le premier élément du résultat est pris en compte. Si le résultat n'est pas un booléen, alors `NULL` est renvoyé.

Opérateur Description Exemple(s)
`'{"a":[1,2,3,4,5]}'::jsonb @@ '$.a[*] > 2'` → `t`

> **Note**
>
> Les opérateurs `@?` et `@@` du type `jsonpath` suppriment les erreurs suivantes : champ objet ou élément de tableau manquant, type d'élément JSON inattendu, erreurs de date/heure ou de numérique. Les fonctions relatives à `jsonpath` décrites ci-dessous peuvent aussi supprimer ce type d'erreurs à la demande. Ce comportement pourrait être utile lors de la recherche de collections de documents JSON de différentes structures.

Tableau 9.47 montre les fonctions disponibles pour construire des valeurs `json` et `jsonb`.

Tableau 9.47. Fonctions de création de JSON

Fonction Description Exemple(s)
`to_json` (`anyelement`) → `json` `to_jsonb` (`anyelement`) → `jsonb` Convertit toute valeur SQL en `json` ou `jsonb`. Les tableaux et valeurs composites sont convertis récursivement en tableaux et en objets (les tableaux multidimensionnels deviennent des tableaux de tableaux en JSON). Sinon, s'il existe une conversion entre le type de données SQL et `json`, la fonction de conversion sera utilisée pour réaliser la conversion ;[a] sinon, une valeur JSON scalaire est produite. Pour tout scalaire autre qu'un nombre, un booléen ou une valeur NULL, la représentation textuelle sera utilisée avec les échappements nécessaires pour la transformer en valeur JSON valide. `to_json('Fred said "Hi."'::text)` → `"Fred said \"Hi.\""` `to_jsonb(row(42, 'Fred said "Hi."'::text))` → `{"f1": 42, "f2": "Fred said \"Hi.\""}`
`array_to_json` (`anyarray` [, `boolean`]) → `json` Convertit un tableau SQL en tableau JSON. Le comportement est le même que `to_json` sauf que des sauts de ligne seront ajoutés entre les éléments de tableau de haut niveau si le paramètre booléen optionnel vaut true. `array_to_json('{{1,5},{99,100}}'::int[])` → `[[1,5],[99,100]]`
`row_to_json` (`record` [, `boolean`]) → `json` Convertit une valeur composite SQL en objet JSON. Le comportement est le même que `to_json` sauf que des sauts de ligne seront ajoutés entre les éléments de tableau de haut niveau si le paramètre booléen optionnel vaut true. `row_to_json(row(1,'foo'))` → `{"f1":1,"f2":"foo"}`
`json_build_array` (`VARIADIC "any"`) → `json` `jsonb_build_array` (`VARIADIC "any"`) → `jsonb` Construit un tableau JSON avec des types potentiellement hétérogènes à partir d'une liste d'arguments de longueur variable. Chaque argument est converti avec la fonction `to_json` ou `to_jsonb`. `json_build_array(1, 2, 'foo',4,5)` → `[1, 2, "foo", 4, 5]`
`json_build_object` (`VARIADIC "any"`) → `json` `jsonb_build_object` (`VARIADIC "any"`) → `jsonb`

Fonction **Description** **Exemple(s)**
Construit un objet JSON à partir d'une liste d'arguments de longueur indéfinie. Par convention, la liste d'arguments consiste en des clés et valeurs en alternance. Les arguments clés sont convertis en texte ; les arguments valeurs sont convertis via les fonctions `to_json` ou `to_jsonb`. `json_build_object('foo', 1, 2, row(3, 'bar'))` → `{"foo" : 1, "2" : {"f1":3,"f2":"bar"}}`
`json_object ( text[] ) → json` `jsonb_object ( text[] ) → jsonb` Construit un objet JSON à partir d'un tableau de texte. Le tableau doit avoir soit exactement une dimension avec un nombre pair de membres, auquel cas ils sont pris comme des paires alternantes clé/valeur, ou deux dimensions de telle façon que le tableau interne a exactement deux éléments, pris comme une paire clé/valeur. Toutes les valeurs sont converties en chaîne JSON. `json_object('{a, 1, b, "def", c, 3.5}')` → `{"a" : "1", "b" : "def", "c" : "3.5"}` `json_object('{{a, 1}, {b, "def"}, {c, 3.5}}')` → `{"a" : "1", "b" : "def", "c" : "3.5"}`
`json_object ( keys text[], values text[] ) → json` `jsonb_object ( keys text[], values text[] ) → jsonb` Cette forme de `json_object` prend les clés et valeurs à partir de tableaux de texte séparés. Pour le reste, elle est identique à la forme à un argument. `json_object('{a, b}', '{1,2}')` → `{"a": "1", "b": "2"}`

[a] Par exemple, l'extension hstore contient une conversion de `hstore` vers `json`, pour que les valeurs `hstore` converties par les fonctions de création JSON soient représentées en tant qu'objets JSON, et non pas comme des valeurs de type chaîne de caractères.

Tableau 9.48 montre les fonctions disponibles pour le traitement de valeurs `json` et `jsonb`.

Tableau 9.48. Fonctions de traitement JSON

Fonction **Description** **Exemple(s)**
`json_array_elements ( json ) → setof json` `jsonb_array_elements ( jsonb ) → setof jsonb` Étend le tableau JSON de haut niveau en un ensemble de valeurs JSON. `select * from json_array_elements('[1,true, [2,false]]') →` `   value` `-----------` ` 1` ` true` ` [2,false]`
`json_array_elements_text ( json ) → setof text` `jsonb_array_elements_text ( jsonb ) → setof text` Étend le tableau JSON de haut niveau en un ensemble de valeurs de type `text`. `select * from json_array_elements_text('["foo", "bar"]') →`

Fonction
Description
Exemple(s)

```
   value
-----------
 foo
 bar
```

`json_array_length(json)` → `integer`

`jsonb_array_length(jsonb)` → `integer`

Renvoie le nombre d'éléments dans le tableau JSON de haut niveau.

`json_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]')` → `5`

`jsonb_array_length('[]')` → `0`

`json_each(json)` → `setof record(key text, value json)`

`jsonb_each(jsonb)` → `setof record(key text, value jsonb)`

Étend l'objet JSON de haut niveau en un ensemble de paires clé/valeur.

`select * from json_each('{"a":"foo", "b":"bar"}')` →

```
 key | value
-----+-------
 a   | "foo"
 b   | "bar"
```

`json_each_text(json)` → `setof record(key text, value text)`

`jsonb_each_text(jsonb)` → `setof record(key text, value text)`

Étend l'objet JSON de haut niveau en un ensemble de paires clé/valeur. Les *value* renvoyées seront de type `text`.

`select * from json_each_text('{"a":"foo", "b":"bar"}')` →

```
 key | value
-----+-------
 a   | foo
 b   | bar
```

`json_extract_path(from_json json, VARIADIC path_elems text[])` → `json`

`jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])` → `jsonb`

Extrait un sous-objet JSON au chemin spécifié. (Ceci est fonctionnellement équivalent à l'opérateur #>, mais écrire le chemin sous la forme d'une liste peut être plus agréable dans certains cas.)

`json_extract_path('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}', 'f4', 'f6')` → `"foo"`

`json_extract_path_text(from_json json, VARIADIC path_elems text[])` → `text`

`jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])` → `text`

Extrait le sous-objet JSON au chemin spécifié sous la forme d'un `text`. (Ceci est équivalent fonctionnement à l'opérateur #>>.)

Fonction
Description
Exemple(s)

`json_extract_path_text('{"f2":{"f3":1},"f4":{"f5":99,"f6":"foo"}}', 'f4', 'f6')` → `foo`

`json_object_keys(json)` → `setof text`

`jsonb_object_keys(jsonb)` → `setof text`

Renvoie l'ensemble de clés dans l'objet JSON de haut niveau.

```
select * from json_object_keys('{"f1":"abc","f2":{"f3":"a", "f4":"b"}}') →

 json_object_keys
------------------
 f1
 f2
```

`json_populate_record(`*`base`* `anyelement, `*`from_json`* `json)` → `anyelement`

`jsonb_populate_record(`*`base`* `anyelement, `*`from_json`* `jsonb)` → `anyelement`

Étend l'objet JSON de haut niveau en une ligne ayant le type composite de l'argument *`base`*. L'objet JSON est parcouru pour y trouver les champs dont le nom correspond aux noms de colonnes du type de la ligne en sortie, et leurs valeurs sont insérées dans ces colonnes en sortie. (Les champs qui ne correspondent à aucun nom de colonne en sortie sont ignorés.) Dans le cas d'une utilisation typique, la valeur de *`base`* est simplement `NULL`, ce qui signifie que toute colonne en sortie qui ne correspond pas à un champ d'objet sera remplie avec un NULL. Néanmoins, si *`base`* ne vaut pas `NULL`, alors les valeurs qu'il contient seront utilisées pour les colonnes sans correspondance.
Pour convertir une valeur JSON au type SQL d'une colonne en sortie, les règles suivantes sont appliquées sur cette séquence :

- Une valeur JSON null est convertie en NULL SQL dans tous les cas.
- Si la colonne en sortie est de type `json` ou `jsonb`, la valeur JSON est reproduite exactement.
- Si la colonne en sortie est un type (de ligne) composite, et que la valeur JSON est un objet JSON, les champs de l'objet sont convertis en colonnes du type de ligne par application récursive de ces règles.
- De la même façon, si la colonne en sortie est un type tableau et que la valeur JSON est un tableau JSON, les éléments du tableau JSON sont convertis en éléments du tableau en sortie par application récursive de ces règles.
- Sinon, si la valeur JSON est une chaîne, le contenu de la chaîne est donné à la fonction de conversion en entrée pour le type de données de la colonne.
- Sinon, la représentation textuelle habituelle de la valeur JSON est envoyée à la fonction de conversion en entrée pour le type de données de la colonne.

Bien que l'exemple ci-dessous utilise une valeur constante JSON, une utilisation typique serait de référencer une colonne `json` ou `jsonb` latéralement d'une autre table dans la clause `FROM` de la requête. Écrire `json_populate_record` dans la clause `FROM` est une bonne pratique, car toutes les colonnes extraites sont utilisables sans avoir à faire des appels dupliqués à la fonction.

```
create type subrowtype as (d int, e text); create type myrowtype as (a int, b text[], c subrowtype);
select * from json_populate_record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}, "x": "foo"}') →
```

Fonction
Description
Exemple(s)

```
 a |   b       |      c
---+-----------+-------------
 1 | {2,"a b"} | (4,"a b c")
```

`json_populate_recordset` (*`base`* `anyelement`, *`from_json`* `json`) → `setof anyelement`

`jsonb_populate_recordset` (*`base`* `anyelement`, *`from_json`* `jsonb`) → `setof anyelement`

Étend le tableau JSON de haut niveau composé d'objets en un ensemble de lignes ayant le type composite de l'argument *`base`*. Chaque élément du tableau JSON est traité comme décrit ci-dessus pour `json[b]_populate_record`.

```
create type twoints as (a int, b int);
select * from json_populate_recordset(null::twoints,
'[{"a":1,"b":2}, {"a":3,"b":4}]') →


 a | b
---+---
 1 | 2
 3 | 4
```

`json_to_record` (`json`) → `record`

`jsonb_to_record` (`jsonb`) → `record`

Étend l'objet JSON de haut niveau en une ligne ayant le type composite défini par une clause AS. (Comme avec toutes les fonctions renvoyant `record`, la requête appelante doit explicitement définir la structure de l'enregistrement avec une clause AS.) L'enregistrement en sortie est rempli avec les champs de l'objet JSON, de la même façon que décrite ci-dessus pour `json[b]_populate_record`. Comme il n'existe aucune valeur record en entrée, les colonnes sans correspondance sont remplies avec des NULL.

```
create type myrowtype as (a int, b text);
select * from json_to_record('{"a":1, "b":[1,2,3], "c":
[1,2,3], "e":"bar", "r": {"a": 123, "b": "a b c"}}') as x(a
int, b text, c int[], d text, r myrowtype) →


 a |    b    |    c    | d |       r
---+---------+---------+---+---------------
 1 | [1,2,3] | {1,2,3} |   | (123,"a b c")
```

`json_to_recordset` (`json`) → `setof record`

`jsonb_to_recordset` (`jsonb`) → `setof record`

Étend le tableau JSON de haut niveau composé d'objets en un ensemble de lignes ayant le type composite défini par une clause AS. (Comme avec toutes les fonctions renvoyant `record`, la requête appelante doit explicitement définir la structure de l'enregistrement avec une clause AS.) Chaque élément du tableau JSON est traité comme décrit ci-dessus pour `json[b]_populate_record`.

```
select * from json_to_recordset('[{"a":1,"b":"foo"},
{"a":"2","c":"bar"}]') as x(a int, b text) →
```

Fonction **Description** **Exemple(s)**
`` a | b ---+----- 1 | foo 2 | ```
`jsonb_set` (*`target`* `jsonb`, *`path`* `text[]`, *`new_value`* `jsonb` [, *`create_if_missing`* `boolean`]) → `jsonb` Renvoie *`target`* avec l'élément désigné par *`path`* remplacé par *`new_value`* ou avec *`new_value`* ajouté si *`create_if_missing`* vaut true (ce qui est la valeur par défaut) et si l'élément désigné par *`path`* n'existe pas. Toutes les étapes précédentes dans le chemin doivent exister, sinon le *`target`* est renvoyé intact. Comme avec les opérateurs orientés chemin, les entiers négatifs qui apparaissent dans *`path`* se décomptent à partir de la fin des tableaux JSON. Si l'étape du dernier chemin est un index inexistant (hors limite) du tableau et si *`create_if_missing`* vaut true, la nouvelle valeur est ajoutée au début du tableau si l'index est négatif et à la fin du tableau s'il est positif. `jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}','[2,3,4]', false)` → `[{"f1": [2, 3, 4], "f2": null}, 2, null, 3]` `jsonb_set('[{"f1":1,"f2":null},2]', '{0,f3}', '[2,3,4]')` → `[{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]`
`jsonb_set_lax` (*`target`* `jsonb`, *`path`* `text[]`, *`new_value`* `jsonb` [, *`create_if_missing`* `boolean` [, *`null_value_treatment`* `text`]]) → `jsonb` Si *`new_value`* n'est pas `NULL`, se comporte de façon identique à `jsonb_set`. Sinon se comporte suivant la valeur de *`null_value_treatment`* qui doit être parmi `'raise_exception'`, `'use_json_null'`, `'delete_key'` ou `'return_target'`. Par défaut, il s'agit de `'use_json_null'`. `jsonb_set_lax('[{"f1":1,"f2":null},2,null,3]', '{0,f1}', null)` → `[{"f1": null, "f2": null}, 2, null, 3]` `jsonb_set_lax('[{"f1":99,"f2":null},2]', '{0,f3}', null, true, 'return_target')` → `[{"f1": 99, "f2": null}, 2]`
`jsonb_insert` (*`target`* `jsonb`, *`path`* `text[]`, *`new_value`* `jsonb` [, *`insert_after`* `boolean`]) → `jsonb` Renvoie *`target`* avec *`new_value`* inséré. Si l'élément désigné par *`path`* est un élément de tableau, *`new_value`* sera inséré avant cet élément si *`insert_after`* vaut false (ce qui est la valeur par défaut) ou après si *`insert_after`* vaut true. Si l'élément désigné par *`path`* est un champ objet, *`new_value`* sera inséré seulement si l'objet ne contient pas déjà cette clé. Toutes les étapes précédentes dans le chemin doivent exister, sinon *`target`* est renvoyé non modifié. Comme avec les opérateurs orientés chemin, les nombres négatifs qui apparaissent dans *`path`* sont décomptés à partir de la fin des tableaux JSON. Si la dernière étape du chemin est un index hors limite de tableau, la nouvelle valeur est ajoutée au début du tableau si l'index est négatif ou à la fin du tableau s'il est positif. `jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"')` → `{"a": [0, "new_value", 1, 2]}` `jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"', true)` → `{"a": [0, 1, "new_value", 2]}`
`json_strip_nulls` (`json`) → `json`

Fonction
Description
Exemple(s)

`jsonb_strip_nulls` (`jsonb`) → `jsonb`

Supprime tous les champs objets ayant des valeurs NULL à partir de la valeur JSON donnée, de façon récurvise. Les valeurs NULL qui ne sont pas des champs objets ne sont pas touchées.

`json_strip_nulls('[{"f1":1, "f2":null}, 2, null, 3]')` → `[{"f1":1},2,null,3]`

`jsonb_path_exists` (*`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean`]]) → `boolean`

Vérifie si le chemin JSON renvoie des éléments de la valeur JSON spécifiée. Si l'argument *`vars`* est indiqué, il doit correspondre à un objet JSON, et ses champs fournissent des valeurs nommées à substituer dans l'expression `jsonpath`. Si l'argument *`silent`* est indiqué et vaut `true`, la fonction supprime les mêmes erreurs que les opérateurs `@?` et `@@`.

`jsonb_path_exists('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2,"max":4}')` → `t`

`jsonb_path_match` (*`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean`]]) → `boolean`

Renvoie le résultat de la vérification d'un prédicat de chemin JSON pour la valeur JSON spécifiée. Seul le premier élément du résultat est pris en compte. Si le résultat n'est pas un booléen, alors `NULL` est renvoyé. Les arguments optionnels *`vars`* et *`silent`* agissent de la même façon que pour `jsonb_path_exists`.

`jsonb_path_match('{"a":[1,2,3,4,5]}', 'exists($.a[*] ? (@ >= $min && @ <= $max))', '{"min":2,"max":4}')` → `t`

`jsonb_path_query` (*`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean`]]) → `setof jsonb`

Renvoie tous les éléments JSON renvoyés par le chemin JSON pour la valeur JSON spécifiée. Les arguments optionnels *`vars`* et *`silent`* agissent de la même façon que pour `jsonb_path_exists`.

```
select * from jsonb_path_query('{"a":[1,2,3,4,5]}',
'$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2,"max":4}') →


 jsonb_path_query
------------------
 2
 3
 4
```

`jsonb_path_query_array` (*`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean`]]) → `jsonb`

Renvoie tous les éléments JSON renvoyés par le chemin JSON pour la valeur JSON spécifiée, sous la forme d'un tableau JSON. Les arguments optionnels *`vars`* et *`silent`* agissent de la même façon que pour `jsonb_path_exists`.

`jsonb_path_query_array('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` → `[2, 3, 4]`

`jsonb_path_query_first` (*`target`* `jsonb`, *`path`* `jsonpath` [, *`vars`* `jsonb` [, *`silent`* `boolean`]]) → `jsonb`

Fonction
Description
Exemple(s)

Renvoie le premier élément JSON renvoyé par le chemin JSON pour la valeur JSON spécifiée. Renvoie `NULL` s'il n'y a pas de résultat. Les arguments optionnels *`vars`* et *`silent`* agissent de la même façon que pour `jsonb_path_exists`.
`jsonb_path_query_first('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2, "max":4}')` → `2`

`jsonb_path_exists_tz ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ]] )` → `boolean`
`jsonb_path_match_tz ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ]] )` → `boolean`
`jsonb_path_query_tz ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ]] )` → `setof jsonb`
`jsonb_path_query_array_tz ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ]] )` → `jsonb`
`jsonb_path_query_first_tz ( target jsonb, path jsonpath [, vars jsonb [, silent boolean ]] )` → `jsonb`
Ces fonctions agissent comme leurs homologues décrits ci-dessus sans le suffixe `_tz`, sauf que ces fonctions acceptent la comparaison de valeurs date/heure qui nécessitent des conversions adaptées suivant le fuseau horaire. L'exemple ci-dessous requiert l'interprétation de la valeur date seule `2015-08-02` comme un horodatage avec fuseau horaire, pour que le résultat dépende du paramétrage actuel de TimeZone. Du fait de cette dépendance, ces fonctions sont marquées stables, ce qui signifie que ces fonctions ne peuvent pas être utilisées dans les index. Leurs homologues sont immuables, et donc peuvent être utilisés dans les index ; cependant, elles renverront des erreurs si on leur demande ce type de comparaison.
`jsonb_path_exists_tz('["2015-08-01 12:00:00-05"]', '$[*] ? (@.datetime() < "2015-08-02".datetime())')` → `t`

`jsonb_pretty ( jsonb )` → `text`
Convertit la valeur JSON donnée en un texte proprement indenté.
`jsonb_pretty('[{"f1":1, "f2":null}, 2]')` →

```
[
    {
        "f1": 1,
        "f2": null
    },
    2
]
```

`json_typeof ( json )` → `text`
`jsonb_typeof ( jsonb )` → `text`
Renvoie le type de la valeur JSON de haut niveau sous la forme d'une chaîne de caractères. Les types possibles sont `object`, `array`, `string`, `number`, `boolean` et `null`. (Le résultat `null` ne doit pas être pris pour un NULL SQL ; voir les exemples.)
`json_typeof('-123.4')` → `number`
`json_typeof('null'::json)` → `null`
`json_typeof(NULL::json) IS NULL` → `t`

9.16.2. Le langage de chemin SQL/JSON

Les expressions de chemin SQL/JSON indiquent les éléments à récupérer à partir de données JSON, similaires aux expressions XPath utilisées pour l'accès SQL au XML. Dans PostgreSQL, les expressions de chemin sont implémentées sous la forme du type de données `jsonpath` et peuvent utiliser tout élément décrit dans Section 8.14.7.

Les fonctions et opérateurs de requêtes JSON passent l'expression de chemin fournie au *moteur de chemin* pour évaluation. Si l'expression correspond à la donnée JSON requêtée, l'élément JSON correspondant ou l'ensemble d'éléments est renvoyé. Les expressions de chemin sont écrites dans le langage de chemin SQL/JSON et peuvent inclure les expressions et fonctions arithmétiques.

Une expression de chemin consiste en une séquence d'éléments autorisés par le type de données `jsonpath`. L'expression de chemin est habituellement évaluée de gauche à droite, mais vous pouvez utiliser les parenthèses pour modifier l'ordre des opérations. Si l'évaluation réussit, une séquence d'éléments JSON est produite et le résultat de l'évaluation est renvoyé à la fonction de requête JSON qui termine le traitement indiqué.

Pour faire référence à la valeur JSON en cours de requêtage (l'*élément de contexte*), utilisez la variable `$` dans l'expression de chemin. Elle peut être suivie par un ou plusieurs opérateurs d'accesseurs, qui descendent dans la structure JSON, étape par étape, pour récupérer les sous-éléments de l'élément de contexte. Chaque opérateur qui suit traite le résultat de l'étape d'évaluation précédente.

Par exemple, supposez que vous ayez certaines données JSON à partir d'un traqueur GPS que vous voulez analyser, tel que :

```
{
  "track": {
    "segments": [
      {
        "location":   [ 47.763, 13.4034 ],
        "start time": "2018-10-14 10:05:14",
        "HR": 73
      },
      {
        "location":   [ 47.706, 13.2635 ],
        "start time": "2018-10-14 10:39:21",
        "HR": 135
      }
    ]
  }
}
```

Pour récupérer les segments de piste disponibles, vous avez besoin d'utiliser l'opérateur d'accesseur *`.key`* pour descendre à travers les objets JSON tout autour :

```
$.track.segments
```

Pour récupérer le contenu d'un tableau, vous utilisez typiquement l'opérateur `[*]`. Par exemple, le chemin suivant renverra les coordonnées d'emplacement pour tous les segments de piste disponibles :

```
$.track.segments[*].location
```

Pour renvoyer les coordonnées du premier segment seulement, vous pouvez indiquer l'indice correspondant dans l'opérateur `[]`. Rappellez-vous que les indices de tableau JSON commencent à 0 :

```
$.track.segments[0].location
```

Le résultat de chaque étape d'évaluation de chemin peut être traité par un ou plusieurs opérateurs et méthodes `jsonpath`, listés dans Section 9.16.2.2. Chaque nom de méthode doit être précédé d'un point. Par exemple, vous pouvez obtenir la taille d'un tableau :

```
$.track.segments.size()
```

D'autres exemples d'utilisation d'opérateurs et méthodes `jsonpath` à l'intérieur d'expressions de chemins apparaissent ci-dessous dans Section 9.16.2.2.

Lors de la définition d'un chemin, vous pouvez aussi utiliser une ou plusieurs *expressions de filtre* qui fonctionnent de façon similaire à la clause `WHERE` en SQL. Une expression de filtre commence avec un point d'interrogation et fournit une condition entre parenthèses :

```
? (condition)
```

Les expressions de filtre doivent être écrites juste après l'étape d'évaluation du chemin auquel elles s'appliquent. Le résultat de cette étape est filtré pour inclure seulement les éléments qui satisfont la condition fournie. SQL/JSON définit une logique à trois valeurs, donc la condition peut valoir `true`, `false` ou `unknown`. La valeur `unknown` joue le même rôle que le `NULL` SQL et peut être testée avec le prédicat `is unknown`. Les étapes suivantes d'évaluation du chemin utilisent seulement les éléments pour lesquels l'expression de filtre renvoie `true`.

Les fonctions et opérateurs pouvant être utilisés dans des expressions de filtre sont listés dans Tableau 9.50. À l'intérieur d'une expression de filtre, la variable `@` dénote la valeur en cours de filtrage (un résultat de l'étape de chemin précédente). Vous pouvez écrire les opérateurs d'accesseurs après `@` pour récupérer les éléments du composant.

Par exemple, supposez que vous vouliez récupérer toutes les valeurs de fréquence cardiaque supérieures à 130. Vous pouvez le faire en utilisant l'expression suivante :

```
$.track.segments[*].HR ? (@ > 130)
```

Pour obtenir les heures de début des segments ayant une telle valeur, vous devez filtrer les segments incompatiables avant de renvoyer les heures de début, donc l'expression de filtre est appliquée à l'étape précédente, et le chemin utilisé dans la condition est différent :

```
$.track.segments[*] ? (@.HR > 130)."start time"
```

Vous pouvez utiliser plusieurs expressions de filtre en séquence, si nécessaire. Par exemple, l'expression suivante sélectionne les heures de début de tous les segments qui contiennent des emplacements avec les bonnes coordonnées et les valeurs hautes de fréquence cardiaque :

```
$.track.segments[*] ? (@.location[1] < 13.4) ? (@.HR > 130)."start
 time"
```

Il est aussi autorisé d'utiliser des expressions de filtre à différents niveaux de profondeur. L'exemple suivant filtre tout d'abord tous les segments par emplacement, puis renvoie les valeurs hautes de fréquence cardiaque sur ces segments, si disponibles :

```
$.track.segments[*] ? (@.location[1] < 13.4).HR ? (@ > 130)
```

Vous pouvez aussi imbriquer les expressions de filtre :

```
$.track ? (exists(@.segments[*] ? (@.HR > 130))).segments.size()
```

Cette expression renvoie la taille de la trace si elle contient des segments avec des valeurs hautes de fréquence cardiaque ou, sinon, une séquence vide.

L'implémentation PostgreSQL du langage de chemin SQL/JSON présente les différences suivantes sur le standard SQL/JSON :

- Une expression de chemin peut être un prédicat booléen, bien que le standard SQL/JSON autorise les prédicats uniquement dans les filtres. Ceci est nécessaire pour l'implémentation de l'opérateur @@. Par exemple, l'expression `jsonpath` suivante est valide dans PostgreSQL :

  ```
  $.track.segments[*].HR < 70
  ```

- Il existe des différences mineures dans l'interprétation des motifs d'expression rationnelle utilisés dans les filtres `like_regex`, comme décrit dans Section 9.16.2.3.

9.16.2.1. Modes strict et permissif

Quand vous requêtez des données JSON, l'expression de chemin pourrait ne pas correspondre à la structure de données JSON actuelle. Une tentative d'accéder à un membre inexistant d'un objet ou d'un élément d'un tableau résulterait en une erreur de structure. Les expressions de chemin SQL/JSON ont deux modes pour la gestion des erreurs de structure :

- lax (permissif, par défaut) -- le moteur de chemin adapte implicitement la donnée requêtée au chemin spécifié. Toute erreur de structure restante est supprimée et convertie en des séquences SQL/JSON vides.
- strict -- si une erreur de structure survient, une erreur est levée.

Le mode permissif facilite la correspondance de la structure d'un document JSON et celle d'une expression de chemin si la donnée JSON n'est pas conforme au schéma attendu. Si un opérande ne correspond pas aux demandes d'une opération particulière, il peut être automatiquement intégré à un tableau SQL/JSON ou déballé en convertissant ses éléments en une séquence SQL/JSON avant de réaliser cette opération. De plus, les opérateurs de comparaison déballent automatiquement leurs opérandes dans le mode permissif, donc vous pouvez directement comparer les tableaux SQL/JSON. Un tableau de taille 1 est considéré égal à son seul élément. Le déballage automatique n'est pas réaliser si :

- L'expression de chemin contient les méthodes `type()` ou `size()` qui renvoient respectivement le type et le nombre d'éléments dans le tableau.
- Les données JSON requêtées contiennent des tableaux imbriqués. Dans ce cas, seul le tableau externe est déballé alors que les tableaux internes restent inchangés. De ce fait, le déballage implicite peut seulement descendre d'un niveau à chaque étape d'évaluation du chemin.

Par exemple, lors du requêtage de données GPS ci-dessus, vous pouvez faire abstraction du fait qu'il stocke un tableau de segments en utilisant le mode permissif :

```
lax $.track.segments.location
```

Dans le mode strict, le chemin spécifié doit correspondre exactement à la structure du document JSON requêté pour renvoyer un élément SQL/JSON, donc utiliser cette expression de chemin causera une erreur. Pour obtenir le même résultat que dans le mode permissif, vous devez explicitement déballer le tableau `segments` :

```
strict $.track.segments[*].location
```

L'accesseur `.**` peut apporter des résultats surprenants lors de l'utilisation du mode non strict. Par exemple, la requête suivante sélectionne chaque valeur `HR` deux fois :

```
lax $.**.HR
```

Ceci survient parce que l'accesseur `.**` sélectionne à la fois le tableau de `segments` et chacun de ses éléments, alors que l'accesseur `.HR` déballe automatiquement les tableaux lors de l'utilisation du mode non strict. Pour éviter des résultats surprenants, nous recommandons d'utiliser l'accesseur `.**` uniquement dans le mode strict. la requête suivant sélectionne chaque valeur `HR` une seule fois :

```
strict $.**.HR
```

9.16.2.2. Opérateurs et méthodes de chemin SQL/JSON

Tableau 9.49 montre les opérateurs et méthodes disponibles pour `jsonpath`. Notez que, bien que les opérateurs et méthodes unaires puissent être appliqués à de nombreuses valeurs résultant d'une étape de chemin précédente, les opérateurs binaires (addition etc.) peuvent seulement être appliqués à des valeurs seules.

Tableau 9.49. Opérateurs et méthodes `jsonpath`

Opérateur/Méthode **Description** **Exemple(s)**
number + *number* → *number* Addition `jsonb_path_query('[2]', '$[0] + 3')` → `5`
+ *number* → *number* Plus unaire (pas d'opération) ; contrairement à l'addition, ceci peut itérer sur plusieurs valeurs `jsonb_path_query_array('{"x": [2,3,4]}', '+ $.x')` → `[2, 3, 4]`
number - *number* → *number* Soustraction `jsonb_path_query('[2]', '7 - $[0]')` → `5`
- *number* → *number* Négation ; contrairement à la soustraction, ceci peut itérer sur plusieurs valeurs `jsonb_path_query_array('{"x": [2,3,4]}', '- $.x')` → `[-2, -3, -4]`
number * *number* → *number* Multiplication `jsonb_path_query('[4]', '2 * $[0]')` → `8`
number / *number* → *number*

Opérateur/Méthode **Description** **Exemple(s)**
Division `jsonb_path_query('[8.5]', '$[0] / 2')` → `4.2500000000000000`
number `%` *number* → *number* Modulo (reste) `jsonb_path_query('[32]', '$[0] % 10')` → `2`
value `.type()` → *string* Type de l'élément JSON (voir `json_typeof`) `jsonb_path_query_array('[1, "2", {}]', '$[*].type()')` → `["number", "string", "object"]`
value `.size()` → *number* Taille de l'élément JSON (nombre d'éléments d'un tableau, ou 1 si ce n'est pas un tableau) `jsonb_path_query('{"m": [11, 15]}', '$.m.size()')` → `2`
value `.double()` → *number* Nombre flottant approximatif converti en nombre JSON ou en chaîne `jsonb_path_query('{"len": "1.9"}', '$.len.double() * 2')` → `3.8`
number `.ceiling()` → *number* Entier le plus proche, plus grand ou égal au nombre donné `jsonb_path_query('{"h": 1.3}', '$.h.ceiling()')` → `2`
number `.floor()` → *number* Entier le plus proche, plus petit ou égal au nombre donné `jsonb_path_query('{"h": 1.7}', '$.h.floor()')` → `1`
number `.abs()` → *number* Valeur absolue du nombre donné `jsonb_path_query('{"z": -0.3}', '$.z.abs()')` → `0.3`
string `.datetime()` → *datetime_type* (voir note) Valeur date/heure convertie en chaîne `jsonb_path_query('["2015-8-1", "2015-08-12"]', '$[*] ? (@.datetime() < "2015-08-2".datetime())')` → `"2015-8-1"`
string `.datetime(`*template*`)` → *datetime_type* (voir note) Valeur date/heure convertie en une chaîne en utilisant le modèle `to_timestamp` indiqué `jsonb_path_query_array('["12:30", "18:40"]', '$[*].datetime("HH24:MI")')` → `["12:30:00", "18:40:00"]`
object `.keyvalue()` → *array* Les paires clé-valeur de l'objet, représentées sous la forme d'un tableau d'objets contenant trois champs : `"key"`, `"value"` et `"id"` ; `"id"` est un identifiant unique de l'objet auquel la paire clé-valeur appartient `jsonb_path_query_array('{"x": "20", "y": 32}', '$.keyvalue()')` → `[{"id": 0, "key": "x", "value": "20"}, {"id": 0, "key": "y", "value": 32}]`

Note

Le type de résultat des méthodes `datetime()` et `datetime(*template*)` peut être `date`, `timetz`, `time`, `timestamptz` ou `timestamp`. Les deux méthodes déterminent dynamiquement le type du résultat.

La méthode `datetime()` trie en séquence pour faire correspondre sa chaîne en entrée aux formats ISO pour les types `date`, `timetz`, `time`, `timestamptz` et `timestamp`. Elle s'arrête au premier format correspondant et émet le type de données correspondant.

La méthode `datetime(*template*)` détermine le type de résultat suivant les champs utilisés dans la chaîne modèle fournie.

Les méthodes `datetime()` et `datetime(*template*)` utilisent les mêmes règles d'analyse que la fonction SQL `to_timestamp` (voir Section 9.8), avec trois exceptions. Tout d'abord, ces méthodes n'autorisent pas les motifs de modèle sans correspondance. Ensuite, seuls les séparateurs suivants sont autorisés dans la chaîne modèle : signe moins, point, barre oblique, virgule, apostrophe, point-virgule, deux-points et espace. Enfin, les séparateurs dans la chaîne modèle doivent correspondre exactement à la chaîne en entrée.

Si différents types date/heure doivent être comparés, une conversion implicite est appliquée. Une valeur `date` peut être convertie en `timestamp` ou `timestamptz`, une valeur `timestamp` peut être convertie en `timestamptz`, et une valeur `time` en `timetz`. Néanmoins, toutes sauf la première de ces conversions dépendent du paramétrage actuel de TimeZone, et de ce fait peuvent seulement être réalisées à travers les fonctions `jsonpath` sensibles au fuseau horaire.

Tableau 9.50 montre les éléments d'expression de filtre disponibles.

Tableau 9.50. Éléments d'expression de filtre `jsonpath`

Prédicat/valeur **Description** **Exemple(s)**
value == *value* → `boolean` Comparaison d'égalité (ceci, et les autres opérateurs de comparaison, fonctionnent sur toutes les valeurs scalaires JSON) `jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == 1)')` → `[1, 1]` `jsonb_path_query_array('[1, "a", 1, 3]', '$[*] ? (@ == "a")')` → `["a"]`
value != *value* → `boolean` *value* <> *value* → `boolean` Comparaison de non égalité `jsonb_path_query_array('[1, 2, 1, 3]', '$[*] ? (@ != 1)')` → `[2, 3]` `jsonb_path_query_array('["a", "b", "c"]', '$[*] ? (@ <> "b")')` → `["a", "c"]`
value < *value* → `boolean` Comparaison inférieur `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ < 2)')` → `[1]`
value <= *value* → `boolean` Comparaison inférieur ou égal

Prédicat/valeur **Description** **Exemple(s)**
`jsonb_path_query_array('["a", "b", "c"]', '$[*] ? (@ <= "b")')` → `["a", "b"]`
value `>` *value* → `boolean` Comparaison supérieur `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ > 2)')` → `[3]`
value `>=` *value* → `boolean` Comparaison supérieur ou égal `jsonb_path_query_array('[1, 2, 3]', '$[*] ? (@ >= 2)')` → `[2, 3]`
`true` → `boolean` Constante JSON `true` `jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '$[*] ? (@.parent == true)')` → `{"name": "Chris", "parent": true}`
`false` → `boolean` Constante JSON `false` `jsonb_path_query('[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]', '$[*] ? (@.parent == false)')` → `{"name": "John", "parent": false}`
`null` → *value* Constante JSON `null` (notez que, contrairement au SQL, la comparaison avec `null` fonctionne normalement) `jsonb_path_query('[{"name": "Mary", "job": null}, {"name": "Michael", "job": "driver"}]', '$[*] ? (@.job == null) .name')` → `"Mary"`
boolean `&&` *boolean* → `boolean` AND booléen `jsonb_path_query('[1, 3, 7]', '$[*] ? (@ > 1 && @ < 5)')` → `3`
boolean `||` *boolean* → `boolean` OR booléen `jsonb_path_query('[1, 3, 7]', '$[*] ? (@ < 1 || @ > 5)')` → `7`
`!` *boolean* → `boolean` NOT booléen `jsonb_path_query('[1, 3, 7]', '$[*] ? (!(@ < 5))')` → `7`
boolean `is unknown` → `boolean` Teste si une condition booléenne est `unknown`. `jsonb_path_query('[-1, 2, 7, "foo"]', '$[*] ? ((@ > 0) is unknown)')` → `"foo"`
string `like_regex` *string* [`flag` *string*] → `boolean` Teste si le premier opérande correspond à l'expression rationnelle donnée par le deuxième opérande, optionnellement avec les modifications décrites par une chaîne avec les caractères de `flag` (voir Section 9.16.2.3).

Prédicat/valeur **Description** **Exemple(s)**
`jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c")')` → `["abc", "abdacb"]` `jsonb_path_query_array('["abc", "abd", "aBdC", "abdacb", "babc"]', '$[*] ? (@ like_regex "^ab.*c" flag "i")')` → `["abc", "aBdC", "abdacb"]`
string `starts with` *string* → `boolean` Teste si le deuxième opérande est une sous-chaîne initiale du premier opérande. `jsonb_path_query('["John Smith", "Mary Stone", "Bob Johnson"]', '$[*] ? (@ starts with "John")')` → `"John Smith"`
`exists (` *path_expression* `)` → `boolean` Teste si une expression de chemin correspond à au moins un élément SQL/JSON. Renvoie `unknown` si l'expression de chemin retourne une erreur ; le deuxième exemple utilise ceci pour éviter une erreur de clé inexistante dans le mode strict. `jsonb_path_query('{"x": [1, 2], "y": [2, 4]}', 'strict $.* ? (exists (@ ? (@[*] > 2)))')` → `[2, 4]` `jsonb_path_query_array('{"value": 41}', 'strict $ ? (exists (@.name)) .name')` → `[]`

9.16.2.3. Expressions rationnelles SQL/JSON

Les expressions de chemin SQL/JSON permettent la correspondance de texte selon une expression rationnelle avec le filtre `like_regex`. Par exemple, la requête de chemin SQL/JSON suivante correspondrait, sans sensibilité à la casse, à toutes les chaînes d'un tableau commençant avec une voyelle anglaise :

```
$[*] ? (@ like_regex "^[aeiou]" flag "i")
```

La chaîne optionnelle `flag` pourrait inclure un ou plusieurs caractères : `i` pour une recherche insensible à la casse, `m` pour autoriser `^` et `$` à correspondre aux nouvelles lignes, `s` pour autoriser `.` à correspondre à une nouvelle ligne, et `q` pour englober le motif complet (réduisant le comportement à une simple correspondance de sous-chaîne).

Le standard SQL/JSON emprunte sa définition d'expressions rationnelles de l'opérateur `LIKE_REGEX` qui, à son tour, utilise le standard XQuery. PostgreSQL n'accepte pas actuellement l'opérateur `LIKE_REGEX`. De ce fait, le filtre `like_regex` est implémenté en utilisant le moteur d'expression rationnelle POSIX décrit dans Section 9.7.3. Ceci amène différentes variations mineures du comportement du standard SQL/JSON, listées dans Section 9.7.3.8. Notez, néanmoins, que les incompatibilités au niveau des lettres du drapeau décrites ici ne s'appliquent pas à SQL/JSON, car elles traduisent les lettres drapeau de XQuery pour correspondre à ce que le moteur POSIX attend.

Gardez en tête que l'argument motif de `like_regex` est une chaîne littérale de chemin JSON, écrite suivant les règles données dans Section 8.14.7. Ceci signifie en particulier que tout antislash utilisé dans l'expression rationnelle doit être doublé. Par exemple, pour établir une correspondance aux valeurs de type chaîne de caractères du document racine ne contenant que des chiffres :

```
$.* ? (@ like_regex "^\\d+$")
```

9.17. Fonctions de manipulation de séquence

Cette section décrit les fonctions pour opérer sur des *objets séquence*, aussi appelés des générateurs de séquence ou plus simplement séquences. Les objets séquence sont des tables spéciales à une ligne créées avec l'ordre CREATE SEQUENCE. Les objets séquence sont habituellement utilisés pour générer des identifiants uniques pour les lignes d'une table. Les fonctions pour séquence, listées dans Tableau 9.51, fournissent des méthodes simples, sûres pour la concurrence d'utilisateur, permettent d'obtenir des valeurs successives de séquence à partir d'objets séquence.

Tableau 9.51. Fonctions pour séquence

Fonction Description
`nextval ( regclass ) → bigint` Avance l'objet séquence à sa prochaine valeur et renvoie cette valeur. Ceci est fait de façon atomique : même si plusieurs sessions exécutent `nextval` en même temps, chacune recevra une valeur distincte de séquence. Si l'objet séquence a été créé avec les paramètres par défaut, les appels successifs à `nextval` renverront les valeurs successives en partant de 1. D'autres comportements peuvent être obtenus en utilisant les paramètres appropriés dans la commande CREATE SEQUENCE. Cette fonction nécessite le droit USAGE ou UPDATE sur la séquence.
`setval ( regclass, bigint [, boolean ] ) → bigint` Configure la valeur actuelle de l'objet séquence, et en option son drapeau `is_called`. La syntaxe à deux paramètres configure le champ `last_value` à la valeur indiquée et configure le champ `is_called` à la valeur `true`, signifiant que le prochain appel à `nextval` avancera la séquence avant de renvoyer la valeur. La valeur qui sera alors rapportée par `currval` est aussi configurée à la valeur indiquée. Dans la syntaxe à trois paramètres, `is_called` peut être configuré à `true` ou à `false`. `true` a le même effet que dans la syntaxe à deux paramètres. S'il est configuré à `false`, le prochain appel à `nextval` renverra exactement la valeur indiqué, et l'avancement de la séquence commence avec le `nextval` suivant. De plus, la valeur renvoyée par `currval` n'est pas modifiée dans ce cas. Par exemple, `SELECT setval('myseq', 42);  Le prochain nextval renverra 43` `SELECT setval('myseq', 42, true);  Identique` `SELECT setval('myseq', 42, false);  Le prochain nextval renverra 42` Le résultat renvoyé par `setval` est simplement la valeur de son deuxième argument. Cette fonction nécessite le droit UPDATE sur la séquence.
`currval ( regclass ) → bigint` Renvoie la valeur la plus récemment obtenue par `nextval` pour cette séquence dans le session courante. (Une erreur est renvoyée si `nextval` n'a jamais été appelé par cette séquence dans cette session.) Comme cela renvoie une valeur locale à la session, la réponse est prévisible que d'autres sessions aient `nextval` ou pas depuis que la session courant l'a fait. Cette fonction nécessite le droit USAGE ou le droit SELECT sur la séquence.
`lastval () → bigint` Renvoie la valeur la plus récemment renvoyée par la fonction `nextval` dans la session courante. Cette fonction est identique à `currval`, sauf qu'au lieu de prendre le nom de la séquence en argument, elle se réfère à la dernière séquence utilisée avec `nextval`

Fonction Description
dans la session courante. Une erreur est renvoyé si `lastval` a été appelée alors que `nextval` ne l'a pas été dans la session courante. Cette fonction nécessite le droit `USAGE` ou le droit `SELECT` sur la dernière séquence utilisée.

> **Attention**
>
> Pour éviter le blocage de transactions concurrentes pour l'obtention de nombres provenant de la même séquence, la valeur obtenue par `nextval` n'est pas réclamée pour une réutilisation si la transaction appelante s'annule après coup. Ceci signifie que des annulations de transaction ou des crashs de bases de données peuvent avoir pour conséquence des trous dans la séquence des valeurs affectées. Ceci peut aussi survenir sans annulation de transaction. Par exemple, un `INSERT` avec une clause `ON CONFLICT` calculera la ligne à insérer, incluant tout appel nécessaire à `nextval`, avant de détecter un conflit qui causera la poursuite sur la règle `ON CONFLICT`. De ce fait, les objets séquences de PostgreSQL *ne peuvent pas être utilisés pour obtenir des séquences « sans trou »* .
>
> De la même façon, les changements d'état de séquence réalisés par `setval` sont immédiatement visibles aux autres transactions, et ne sont pas annulées si la transaction appelante annule ses modifications.
>
> Si l'instance s'arrête brutalement avant de valider une transaction ayant exécuté un appel à `nextval` ou `setval`, le changement d'état de la séquence pourrait ne pas arriver jusqu'au stockage permanent, donc l'état de la séquence n'est pas certain, qu'il soit à sa valeur d'origine ou à sa nouvelle valeur après le redémarrage de l'instance. Ceci n'est pas grave pour l'utilisation de la séquence dans la base car les autres effets des transactions non valides ne seront pas visibles. Néanmoins, si vous souhaitez utiliser une valeur de séquence pour un stockage persistant hors de la base, assurez-vous que l'appel à `nextval` soit validé avant de l'utiliser hors base.

La séquence à traiter par une fonction de séquence est indiquée par un argument `regclass`, qui est simplement l'OID de la séquence dans le catalogue système `pg_class`. Vous n'avez pas besoin de rechercher l'OID manuellement car le convertisseur en entrée du type de donnée `regclass` fera ce travail pour vous. Voir Section 8.19 pour les détails.

9.18. Expressions conditionnelles

Cette section décrit les expressions conditionnelles compatibles SQL disponibles dans PostgreSQL.

> **Astuce**
>
> Si vos besoins dépassent les possibilités de ces expressions conditionnelles, il faudrait considérer l'écriture d'une fonction serveur dans un langage de programmation plus expressif.

> **Note**
>
> Bien que `COALESCE`, `GREATEST` et `LEAST` sont syntaxiquement similaires aux fonctions, ce ne sont pas des fonctions ordinaires et, de ce fait, ne peuvent pas être utilisées avec des arguments de tableau `VARIADIC`.

9.18.1. CASE

L'expression SQL CASE est une expression conditionnelle générique, similaire à des instructions if/else dans les autres langages de programmation :

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

Les clauses CASE peuvent être utilisées partout où une expression est valide. Chaque *condition* est une expression qui renvoie un résultat boolean. Si le résultat de la condition est true, la valeur de l'expression CASE est le résultat *result* qui suit la condition, et le reste de l'expression CASE n'est pas traité. Si le résultat de la condition ne vaut pas true, toutes les clauses WHEN suivants sont examinées de la même manière. Si aucun WHEN *condition* ne renvoie true, la valeur de l'expression CASE est le *result* de la clause ELSE. S'il n'y a pas de clause ELSE et qu'aucune condition ne vaut true, le résultat est NULL.

An example:

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

Les types de données de toutes les expressions *result* doivent être convertibles en un seul type de sortie. Voir Section 10.5 pour plus de détails.

Il existe une forme « simple » d'expression CASE qui est une variante de la forme générale ci-dessus :

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

La première *expression* est calculée, puis comparée à chacune des expressions *value* dans les clauses WHEN jusqu'à ce qu'une valeur égale soit trouvée. Si aucune égalité n'est trouvée, l'expression *result* de la clause ELSE (ou une valeur NULL) est renvoyée. Ceci est similaire à l'instruction switch en C.

L'exemple ci-dessus peut être écrit en utilisant la syntaxe simple CASE :

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

Une expression CASE n'évalue pas les sous-expressions qui ne sont pas nécessaires pour déterminer le résultat. Par exemple, voici une façon possible d'éviter une erreur de division par zéro :

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

> **Note**
>
> Comme décrit dans Section 4.2.14, il existe différentes situations pour lesquelles les sous-expressions d'une expression sont évaluées à différents moments, donc le principe pour lequel « CASE évalue seulement les sous-expressions nécessaires » n'est pas sûr. Par exemple, une sous-expression constante 1/0 pourrait donner une erreur de division par zéro lors de l'optimisation de la requête, même si elle est dans une partie du CASE qui n'aurait pas été exécutée au moment de l'exécution.

9.18.2. COALESCE

```
COALESCE(value [, ...])
```

La fonction COALESCE renvoie le premier de ses arguments non NULL. NULL est renvoyé seulement si tous les arguments sont NULL. Elle est souvent utilisée pour substituer une valeur par défaut pour les valeurs NULL quand les données sont récupérées pour affichage. Par exemple :

```
SELECT COALESCE(description, short_description, '(none)') ...
```

Ceci renvoie description si ce champ n'est pas NULL, sinon short_description si ce champ n'est pas NULL, et sinon (none).

Les arguments doivent tous être convertibles vers un type de données commun, qui* sera le type du résultat (voir Section 10.5 pour les détails).

Comme une expression CASE, COALESCE évalue seulement les arguments qui sont nécessaires pour déterminer le résultat ; c'est-à-dire les arguments à droite du premier argument non NULL ne sont pas évalués. Cette fonction du standard SQL fournit des possibilités similaires à NVL et IFNULL, qui sont utilisées dans d'autres systèmes de bases de données.

9.18.3. NULLIF

```
NULLIF(value1, value2)
```

La fonction NULLIF renvoie une valeur NULL si *value1* est égal à *value2* ; sinon, elle renvoie *value1*. Elle peut être utilisée pour réaliser l'opération inverse de l'exemple COALESCE donné ci-dessus :

```
SELECT NULLIF(value, '(none)') ...
```

Dans cet exemple, si value vaut (none), NULL est renvoyé, sinon la valeur de value est renvoyée.

Les deux arguments doit être de types comparables. Pour être spécifique, ils sont comparés exactement comme si vous aviez écrit *value1 = value2*, donc il doit y avoir un opérateur = convenable de disponible.

Le résultat a le même type que le premier argument -- mais il existe une subtilité. Ce qui est réellement renvoyé est le premier argument de l'opérateur = impliqué et, dans certains cas, aura été promu pour correspondre au type du deuxième argument. Par exemple, NULLIF(1, 2.2) renvoie numeric parce qu'il n'y a pas d'opérateur integer = numeric, seulement un numeric = numeric.

9.18.4. GREATEST et LEAST

```
GREATEST(value [, ...])
```

```
LEAST(value [, ...])
```

Les fonctions GREATEST et LEAST sélectionnent la valeur la plus grande ou la plus petite d'une liste d'expressions. Les expressions doivent toutes être convertibles en un type de donnée commun, qui sera le type du résultat (voir Section 10.5 pour les détails). Les valeurs NULL de la liste sont ignorées. Le résultat sera NULL seulement si toutes les expressions sont évaluées à NULL.

Notez que GREATEST and LEAST ne font pas partie du standard SQL mais sont une extension courante. Certaines autres bases de données les font renvoyer NULL si un argument est NULL, plutôt que quand toutes sont NULL.

9.19. Fonctions et opérateurs tableau

Tableau 9.52 montre les opérateurs spécialisés disponibles pour les types tableau. En plus de ces derniers, les opérateurs habituels de comparaison montrés dans Tableau 9.1 sont disponibles pour les tableaux. Les opérateurs de comparaison comparent le contenu des tableaux éléments par éléments, en utilisant la fonction de comparaison B-tree par défaut pour le type de données de l'élément, et trient en se basant sur la première différence rencontrée. Dans les tableaux multi-dimensionnels, les éléments sont visités dans l'ordre des lignes (le dernier indice varie plus rapidement). Si les contenus

de deux tableaux sont identiques mais que leur dimension est différente, la première différence dans l'information de dimension détermine l'ordre de tri.

Tableau 9.52. Opérateurs de tableau

Opérateur **Description** **Exemple(s)**
`anyarray @> anyarray → boolean` Est-ce que le premier tableau contient le second, autrement dit, est-ce que chaque élément apparaissant dans le deuxième tableau est égal à un élément du premier tableau ? (les duplicats ne sont pas traités spécialement, donc `ARRAY[1]` et `ARRAY[1,1]` sont tous les deux considérés comme contenant l'autre.) `ARRAY[1,4,3] @> ARRAY[3,1,3] → t`
`anyarray <@ anyarray → boolean` Est-ce que le premier tableau est contenu par le second ? `ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6] → t`
`anyarray && anyarray → boolean` Est-ce que les tableaux se surchargent, autrement dit ont des éléments en commun ? `ARRAY[1,4,3] && ARRAY[2,1] → t`
`anycompatiblearray || anycompatiblearray → anycompatiblearray` Concatène les deux tableaux. Concaténer un tableau NULL ou un tableau vide est une opération vide ; sinon les tableaux doivent avoir le même nombre de dimensions (comme illustré dans le premier exemple) ou différer de 1 dans le nombre de dimensions (comme illustré dans le second exemple). Si les tableaux n'ont pas des éléments de types identiques, ils seront convertis dans un type commun (voir Section 10.5). `ARRAY[1,2,3] || ARRAY[4,5,6,7] → {1,2,3,4,5,6,7}` `ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9.9]] → {{1,2,3},{4,5,6},{7,8,9.9}}`
`anycompatible || anycompatiblearray → anycompatiblearray` Concatène un élément au début d'un tableau (qui doit être vide ou à une dimension). `3 || ARRAY[4,5,6] → {3,4,5,6}`
`anycompatiblearray || anycompatible → anycompatiblearray` Concatène un élément à la fin d'un tableau (qui doit être vide ou à une dimension). `ARRAY[4,5,6] || 7 → {4,5,6,7}`

Voir Section 8.15 pour plus de détails sur le comportement des opérateurs pour tableau. Voir Section 11.2 pour plus de détails sur les opérateurs supportant des opérations indexées.

Tableau 9.53 montre les fonctions disponibles pour utilisation avec des types tableau. Voir Section 8.15 pour plus d'informations et d'exemples sur l'utilisation de ces fonctions.

Tableau 9.53. Fonctions tableau

Fonction **Description** **Exemple(s)**
`array_append ( anycompatiblearray, anycompatible ) → anycompatiblearray` Ajoute un élément à la fin d'un tableau (identique à l'opérateur `anycompatiblearray || anycompatible`).

Fonction **Description** **Exemple(s)**
`array_append(ARRAY[1,2], 3)` → `{1,2,3}`
`array_cat ( anycompatiblearray, anycompatiblearray ) → anycompatiblearray` Concatène deux tableaux (identique à l'opérateur `anycompatiblearray || anycompatiblearray`). `array_cat(ARRAY[1,2,3], ARRAY[4,5])` → `{1,2,3,4,5}`
`array_dims ( anyarray ) → text` Renvoie une représentation textuelle des dimensions du tableau. `array_dims(ARRAY[[1,2,3], [4,5,6]])` → `[1:2][1:3]`
`array_fill ( anyelement, integer[] [, integer[] ] ) → anyarray` Renvoie un tableau rempli de copies de la valeur donnée, ayant les dimensions de longueurs spécifiés par le deuxième argument. Le troisième argument optionnel fournit des valeurs de limite basse pour chaque dimension (qui vaut par défaut `1`). `array_fill(11, ARRAY[2,3])` → `{{11,11,11},{11,11,11}}` `array_fill(7, ARRAY[3], ARRAY[2])` → `[2:4]={7,7,7}`
`array_length ( anyarray, integer ) → integer` Renvoie la longueur de la dimension réclamée du tableau. (Renvoie NULL à la place de 0 pour des dimensions vides ou manquantes.) `array_length(array[1,2,3], 1)` → `3` `array_length(array[]::int[], 1)` → `NULL` `array_length(array['text'], 2)` → `NULL`
`array_lower ( anyarray, integer ) → integer` Renvoie la limite basse de la dimension réclamée du tableau. `array_lower('[0:2]={1,2,3}'::integer[], 1)` → `0`
`array_ndims ( anyarray ) → integer` Renvoie le nombre de dimensions du tableau. `array_ndims(ARRAY[[1,2,3], [4,5,6]])` → `2`
`array_position ( anycompatiblearray, anycompatible [, integer ] ) → integer` Renvoie l'indice de la première occurrence du second argument dans le tableau, ou `NULL` s'il n'est pas présent. Si le troisième argument est donné, la recherche commence à cet indice. Le tableau doit être d'une dimension. Les comparaisons se font en utilisant les sémantiques de `IS NOT DISTINCT FROM`, donc il est possible de rechercher un `NULL`. `array_position(ARRAY['sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat'], 'mon')` → `2`
`array_positions ( anycompatiblearray, anycompatible ) → integer[]` Renvoie un tableau des indices de toutes les occurrences du deuxième argument dans le tableau donné comme premier argument. Le tableau doit être à une dimension. Les comparaisons sont effectuées en utilisant les sémantiques de `IS NOT DISTINCT FROM`, il est donc possible de rechercher un `NULL`. `NULL` n'est renvoyé que si le tableau vaut `NULL` ; si la valeur n'est pas trouvée dans le tableau, un tableau vide est renvoyé. `array_positions(ARRAY['A','A','B','A'], 'A')` → `{1,2,4}`

Fonction **Description** **Exemple(s)**
`array_prepend` (`anycompatible`, `anycompatiblearray`) → `anycompatiblearray` Ajoute un élément au début d'un tableau (identique à l'opérateur `anyelement || anyarray`). `array_prepend(1, ARRAY[2,3])` → `{1,2,3}`
`array_remove` (`anycompatiblearray`, `anycompatible`) → `anycompatiblearray` Supprime tous les éléments identiques à la valeur donnée à partir du tableau. Le tableau doit être à une dimension. Les comparaisons sont faites en utilisant les sémantiques de `IS NOT DISTINCT FROM`, dont il est possible de supprimer un `NULL`. `array_remove(ARRAY[1,2,3,2], 2)` → `{1,3}`
`array_replace` (`anycompatiblearray`, `anycompatible`, `anyelement`) → `anycompatiblearray` Remplace chaque élément de tableau égal au second argument avec le troisième argument. `array_replace(ARRAY[1,2,5,4], 5, 3)` → `{1,2,3,4}`
`array_to_string` (*`array`* `anyarray`, *`delimiter`* `text` [, *`null_string`* `text`]) → `text` Convertit chaque élément de tableau en sa représentation textuelle, et les concatène en les séparant par la chaîne *`delimiter`*. Si *`null_string`* est indiqué et ne vaut pas `NULL`, alors les entrées de tableau `NULL` sont représentées par cette chaîne ; sinon elles sont omises. Voir aussi `string_to_array`. `array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')` → `1,2,3,*,5`
`array_upper` (`anyarray`, `integer`) → `integer` Renvoie la limite haute de la dimension demandée du tableau. `array_upper(ARRAY[1,8,3,7], 1)` → `4`
`cardinality` (`anyarray`) → `integer` Renvoie le nombre total d'éléments dans le tableau, ou 0 si le tableau est vide. `cardinality(ARRAY[[1,2],[3,4]])` → `4`
`trim_array` (*`array`* `anyarray`, *`n`* `integer`) → `anyarray` Coupe un tableau en supprimant les *`n`* derniers éléments. Si le tableau est multidimensionnel, seule la première dimension est coupée. `trim_array(ARRAY[1,2,3,4,5,6], 2)` → `{1,2,3,4}`
`unnest` (`anyarray`) → `setof anyelement` Étend un tableau en un ensemble de lignes. Les éléments du tableau sont lus dans l'ordre de stockage. `unnest(ARRAY[1,2])` → `1` `2` `unnest(ARRAY[['foo','bar'],['baz','quux']])` →

Fonction **Description** **Exemple(s)**
foo bar baz quux
`unnest ( anyarray, anyarray [, ... ] ) → setof anyelement, anyelement [, ... ]` Étend plusieurs tableaux (potentiellement de types de données différents) en un ensemble de lignes. Si les tableaux ne sont pas de la même longueur, alors les plus courts sont alignés avec des `NULL`. Cette forme est seulement autorisée dans la clause FROM d'une requête ; voir Section 7.2.1.4. `select * from unnest(ARRAY[1,2], ARRAY['foo','bar','baz']) as x(a,b) →` a | b ---+----- 1 | foo 2 | bar | baz

Voir aussi Section 9.21 sur la fonction d'agrégat `array_agg` pour une utilisation avec les tableaux.

9.20. Fonctions et opérateurs sur les intervalles simple/multiple

Voir Section 8.17 pour un aperçu les types d'intervalle.

Tableau 9.54 montre les opérateurs spécialisés disponibles pour les types d'intervalle simple. Tableau 9.55 montre les opérateurs spécialisés disponibles pour les types d'intervalle multiple. En plus de ces derniers, les opérateurs de comparaison habituels montrés dans Tableau 9.1 sont disponibles pour les types d'intervalle simple et multiple. Les opérateurs de comparaison trient d'abord par les limites basses de l'intervalle simple et seulement si ces dernières sont identiques, par les limites hautes. Les opérateurs de comparaison multiple comparent chaque intervalle jusqu'à ce qu'au moins un ne corresponde pas. Ceci ne donne généralement pas de résultat utile pour un tri, mais les opérateurs sont fournis pour autoriser la création d'index d'unicité sur des intervalles.

Tableau 9.54. Opérateurs sur les intervalles simples

Opérateur **Description** **Exemple(s)**
`anyrange @> anyrange → boolean` Est-ce que le premier intervalle contient le second ? `int4range(2,4) @> int4range(2,3) → t`
`anyrange @> anyelement → boolean` Est-ce que l'intervalle contient l'élément ? `'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp → t`

Opérateur **Description** **Exemple(s)**
`anyrange <@ anyrange → boolean` Est-ce que le premier intervalle est contenu par le second ? `int4range(2,4) <@ int4range(1,7) → t`
`anyelement <@ anyrange → boolean` Est-ce que l'élément est contenu dans l'intervalle ? `42 <@ int4range(1,7) → f`
`anyrange && anyrange → boolean` Est-ce que les intervalles se surchargent, c'est-à-dire ont-ils des éléments en commun ? `int8range(3,7) && int8range(4,12) → t`
`anyrange << anyrange → boolean` Est-ce que le premier intervalle se trouve strictement à gauche du second ? `int8range(1,10) << int8range(100,110) → t`
`anyrange >> anyrange → boolean` Est-ce que le premier intervalle est strictement à droite du second ? `int8range(50,60) >> int8range(20,30) → t`
`anyrange &< anyrange → boolean` Est-ce que le premier intervalle ne s'étend pas à droite du second ? `int8range(1,20) &< int8range(18,20) → t`
`anyrange &> anyrange → boolean` Est-ce que le premier intervalle ne s'étend pas à gauche du second ? `int8range(7,20) &> int8range(5,10) → t`
`anyrange -|- anyrange → boolean` Est-ce que les intervalles sont adjacents ? `numrange(1.1,2.2) -|- numrange(2.2,3.3) → t`
`anyrange + anyrange → anyrange` Calcule l'union des intervalles. Les intervalles doivent se superposer ou être adjacents, pour que l'union résultante soit un intervalle simple (mais voir `range_merge()`). `numrange(5,15) + numrange(10,20) → [5,20)`
`anyrange * anyrange → anyrange` Calcule l'intersection des intervalles. `int8range(5,15) * int8range(10,20) → [10,15)`
`anyrange - anyrange → anyrange` Calcule la différence des intervalles. Le deuxième intervalle ne doit pas être contenu dans le premier de façon à ce que la différence ne soit pas un intervalle simple. `int8range(5,15) - int8range(10,20) → [5,10)`

Tableau 9.55. Opérateurs sur les intervalles multiples

Opérateur **Description** **Exemple(s)**
`anymultirange @> anymultirange → boolean`

Opérateur **Description** **Exemple(s)**
Est-ce que le premier intervalle multiple contient le second ? `'{[2,4)}'::int4multirange @> '{[2,3)}'::int4multirange → t`
`anymultirange @> anyrange → boolean` Est-ce que l'intervalle multiple contient l'intervalle simple ? `'{[2,4)}'::int4multirange @> int4range(2,3) → t`
`anymultirange @> anyelement → boolean` Est-ce que l'intervalle multiple contient l'élément ? `'{[2011-01-01,2011-03-01)}'::tsmultirange @> '2011-01-10'::timestamp → t`
`anyrange @> anymultirange → boolean` Est-ce que l'intervalle simple contient l'intervalle multiple ? `'[2,4)'::int4range @> '{[2,3)}'::int4multirange → t`
`anymultirange <@ anymultirange → boolean` Est-ce que le premier intervalle multiple est contenu par le second ? `'{[2,4)}'::int4multirange <@ '{[1,7)}'::int4multirange → t`
`anymultirange <@ anyrange → boolean` Est-ce que l'intervalle multiple est contenu par l'intervalle simple ? `'{[2,4)}'::int4multirange <@ int4range(1,7) → t`
`anyrange <@ anymultirange → boolean` Est-ce que l'intervalle simple est contenu par l'intervalle multiple ? `int4range(2,4) <@ '{[1,7)}'::int4multirange → t`
`anyelement <@ anymultirange → boolean` Est-ce que l'élément est contenu dans l'intervalle multiple ? `4 <@ '{[1,7)}'::int4multirange → t`
`anymultirange && anymultirange → boolean` Est-ce que les intervalles multiples se recouvrent, c'est-à-dire, contiennent des éléments communs ? `'{[3,7)}'::int8multirange && '{[4,12)}'::int8multirange → t`
`anymultirange && anyrange → boolean` Est-ce que l'intervalle multiple recouvre l'intervalle simple ? `'{[3,7)}'::int8multirange && int8range(4,12) → t`
`anyrange && anymultirange → boolean` Est-ce que l'intervalle simple recouvre l'intervalle multiple ? `int8range(3,7) && '{[4,12)}'::int8multirange → t`
`anymultirange << anymultirange → boolean` Est-ce que le premier intervalle multiple est strictement à gauche du second ? `'{[1,10)}'::int8multirange << '{[100,110)}'::int8multirange → t`
`anymultirange << anyrange → boolean` Est-ce que l'intervalle multiple est strictement à gauche de l'intervalle simple ? `'{[1,10)}'::int8multirange << int8range(100,110) → t`

Opérateur **Description** **Exemple(s)**
`anyrange << anymultirange → boolean` Est-ce que l'intervalle simple est strictement à gauche de l'intervalle multiple ? `int8range(1,10) << '{[100,110)}'::int8multirange → t`
`anymultirange >> anymultirange → boolean` Est-ce que le premier intervalle multiple est strictement à droite du second ? Is the first multirange strictly right of the second? `'{[50,60)}'::int8multirange >> '{[20,30)}'::int8multirange → t`
`anymultirange >> anyrange → boolean` Est-ce que l'intervalle multiple est strictement à droite de l'intervalle simple ? `'{[50,60)}'::int8multirange >> int8range(20,30) → t`
`anyrange >> anymultirange → boolean` Est-ce que l'intervalle simple est strictement à droite de l'intervalle multiple ? `int8range(50,60) >> '{[20,30)}'::int8multirange → t`
`anymultirange &< anymultirange → boolean` Est-ce que le premier intervalle multiple ne s'étend pas à droite du second ? `'{[1,20)}'::int8multirange &< '{[18,20)}'::int8multirange → t`
`anymultirange &< anyrange → boolean` Est-ce que l'intervalle multiple ne s'étend pas à droite de l'intervalle simple ? `'{[1,20)}'::int8multirange &< int8range(18,20) → t`
`anyrange &< anymultirange → boolean` Est-ce que l'intervalle simple ne s'étend pas à droite de l'intervalle multiple ? `int8range(1,20) &< '{[18,20)}'::int8multirange → t`
`anymultirange &> anymultirange → boolean` Est-ce que l'intervalle multiple ne s'étend pas à gauche du second ? `'{[7,20)}'::int8multirange &> '{[5,10)}'::int8multirange → t`
`anymultirange &> anyrange → boolean` Est-ce que l'intervalle multiple ne s'étend pas à gauche de l'intervalle simple ? `'{[7,20)}'::int8multirange &> int8range(5,10) → t`
`anyrange &> anymultirange → boolean` Est-ce que l'intervalle simple ne s'étend pas à gauche de l'intervalle multiple ? `int8range(7,20) &> '{[5,10)}'::int8multirange → t`
`anymultirange -|- anymultirange → boolean` Est-ce que les intervalles multiples sont adjacents ? `'{[1.1,2.2)}'::nummultirange -|- '{[2.2,3.3)}'::nummultirange → t`
`anymultirange -|- anyrange → boolean` Est-ce que l'intervalle multiple est adjacent à l'intervalle simple ? `'{[1.1,2.2)}'::nummultirange -|- numrange(2.2,3.3) → t`
`anyrange -|- anymultirange → boolean`

Opérateur **Description** **Exemple(s)**
Est-ce que l'intervalle simple est adjacent à l'intervalle multiple ? `numrange(1.1,2.2) -|- '{[2.2,3.3)}'::nummultirange` → `t`
`anymultirange + anymultirange → anymultirange` Calcule l'union des intervalles multiples. Les intervalles multiples ne doivent pas se recouvrir, ni être adjacents. `'{[5,10)}'::nummultirange + '{[15,20)}'::nummultirange → {[5,10), [15,20)}`
`anymultirange * anymultirange → anymultirange` Calcule l'intersection des intervalles multiples. `'{[5,15)}'::int8multirange * '{[10,20)}'::int8multirange → {[10,15)}`
`anymultirange - anymultirange → anymultirange` Calcule la différence des intervalles multiples. `'{[5,20)}'::int8multirange - '{[10,15)}'::int8multirange → {[5,10), [15,20)}`

Les opérateurs à gauche/à droite/adjacent retournent toujours false quand un intervalle vide simple ou multiple est impliqué ; c'est-à-dire qu'un intervalle vide n'est pas considéré comme étant avant ou après n'importe quel autre intervalle.

D'autre part, les intervalles vides simples et multiples sont traités comme l'identité additive : tout ce qui est uni à une valeur vide reste identique. Tout ce à quoi est soustrait une valeur vide reste identique. Un intervalle multiple vide a exactement les mêmes effets qu'un intervalle simple vide. Tout intervalle simple contient l'intervalle vide. Tout intervalle multiple contient autant d'intervalles vides que souhaité.

Les opérateurs d'union et de différence d'intervalle échoueront si la plage résultante doit contenir deux sous-plages disjointes, car un tel intervalle ne peut pas être représenté. Il y a des opérateurs séparés pour union et différence qui prennent des paramètres intervalles multiples et retournent un intervalle multiple, et qui n'échouent pas même si leurs arguments sont disjoints. Donc, si vous avez besoin d'une opération d'union ou de différence pour des intervalles qui peuvent être disjoints, vous pouvez éviter les erreurs en transformant d'abord vos intervalles simples en multiples.

Tableau 9.56 montre les fonctions utilisables avec des intervalles simples. Tableau 9.57 montre les fonctions utilisables avec des intervalles multiples.

Tableau 9.56. Fonctions sur intervalles simples

Fonction **Description** **Exemple(s)**
`lower ( anyrange ) → anyelement` Extrait la limite inférieure de l'intervalle (`NULL` si l'intervalle est vide ou que la limite inférieure est l'infini). `lower(numrange(1.1,2.2)) → 1.1`
`upper ( anyrange ) → anyelement` Extrait la limite supérieure de l'intervalle (`NULL` si l'intervalle est vide ou que la limite supérieure est l'infini). `upper(numrange(1.1,2.2)) → 2.2`

Fonction **Description** **Exemple(s)**
`isempty ( anyrange ) → boolean` Est-ce que l'intervalle est vide ? `isempty(numrange(1.1,2.2)) → f`
`lower_inc ( anyrange ) → boolean` Est-ce que la limite inférieure de l'intervalle est inclusive ? `lower_inc(numrange(1.1,2.2)) → t`
`upper_inc ( anyrange ) → boolean` Est-ce que la limite supérieure de l'intervalle est inclusive ? `upper_inc(numrange(1.1,2.2)) → f`
`lower_inf ( anyrange ) → boolean` Est-ce que la limite inférieure de l'intervalle est infinie ? `lower_inf('(,)'::daterange) → t`
`upper_inf ( anyrange ) → boolean` Est-ce que la limite supérieure de l'intervalle est infinie ? `upper_inf('(,)'::daterange) → t`
`range_merge ( anyrange, anyrange ) → anyrange` Calcule le plus petit intervalle qui inclut les deux intervalles fournis. `range_merge('[1,2)'::int4range, '[3,4)'::int4range) → [1,4)`

Tableau 9.57. Fonctions sur intervalles multiples

Fonction **Description** **Exemple(s)**
`lower ( anymultirange ) → anyelement` Extrait la limite inférieure de l'intervalle multiple (`NULL` si l'intervalle multiple est vide ou que la limite inférieure est l'infini). `lower('{[1.1,2.2)}'::nummultirange) → 1.1`
`upper ( anymultirange ) → anyelement` Extrait la limite supérieur de l'intervalle multiple (`NULL` si l'intervalle multiple est vide ou que la limite supérieure est l'infini). `upper('{[1.1,2.2)}'::nummultirange) → 2.2`
`isempty ( anymultirange ) → boolean` Est-ce que l'intervalle multiple est vide ? `isempty('{[1.1,2.2)}'::nummultirange) → f`
`lower_inc ( anymultirange ) → boolean` Est-ce que la limite inférieure de l'intervalle multiple est inclusive ? `lower_inc('{[1.1,2.2)}'::nummultirange) → t`
`upper_inc ( anymultirange ) → boolean` Est-ce que la limite supérieure de l'intervalle multiple est inclusive ? `upper_inc('{[1.1,2.2)}'::nummultirange) → f`
`lower_inf ( anymultirange ) → boolean`

Fonction **Description** **Exemple(s)**
Est-ce que la limite inférieure de l'intervalle multiple est infinie ? `lower_inf('{(,)}'::datemultirange)` → `t`
`upper_inf` (`anymultirange`) → `boolean` Est-ce que la limite supérieure de l'intervalle multiple est infinie ? `upper_inf('{(,)}'::datemultirange)` → `t`
`range_merge` (`anymultirange`) → `anyrange` Calcule le plus petit intervalle simple qui inclut entièrement l'intervalle multiple. `range_merge('{[1,2), [3,4)}'::int4multirange)` → `[1,4)`
`multirange` (`anyrange`) → `anymultirange` Retourne un intervalle multiple contenant seulement l'intervalle simple fourni. `multirange('[1,2)'::int4range)` → `{[1,2)}`
`unnest` (`anymultirange`) → `setof anyrange` Étend un intervalle multiple en un ensemble d'intervalles simples. Les intervalles simples sont lus dans l'ordre de stockage (croissant). `unnest('{[1,2), [3,4)}'::int4multirange)` → `[1,2)` `[3,4)`

Les fonctions `lower_inc`, `upper_inc`, `lower_inf` et `upper_inf` renvoient toutes false pour un intervalle simple ou multiple vide.

9.21. Fonctions d'agrégat

Les *fonctions d'agrégat* calculent un seul résultat à partir d'un ensemble de valeurs en entrée. Les fonctions d'agrégat natives à but général sont listées dans Tableau 9.58 alors que les agrégats statistiques sont dans Tableau 9.59. Les fonctions d'agrégat natives à ensemble trié dans des groupes sont listées dans Tableau 9.60 alors que les fonctions à ensemble hypothétique dans des groupes sont dans Tableau 9.61. Les opérations de regroupement, qui sont fortement liées aux fonctions d'agrégat, sont listées dans Tableau 9.62. Les considérations spéciales de syntaxe pour les fonctions d'agrégat sont expliquées dans Section 4.2.7. Consultez Section 2.7 pour une introduction supplémentaire.

Les fonctions d'agrégat qui supportent le *Mode Partiel* sont éligibles à participer à différentes optimisations, telles que l'agrégation parallélisée.

Tableau 9.58. Fonctions d'agrégat à but général

Fonction **Description**	**Mode Partiel**
`array_agg` (`anynonarray`) → `anyarray` Récupère toutes les valeurs en entrée, y compris les NULL, et les place dans un tableau.	No
`array_agg` (`anyarray`) → `anyarray` Concatène tous les tableaux en entrée dans un tableau d'une dimension supérieure. (Les entrées doivent toutes avoir la même dimension, et ne peuvent être ni vides ni NULL.)	No
`avg` (`smallint`) → `numeric`	Yes

Fonction **Description**	**Mode Partiel**
`avg ( integer ) → numeric` `avg ( bigint ) → numeric` `avg ( numeric ) → numeric` `avg ( real ) → double precision` `avg ( double precision ) → double precision` `avg ( interval ) → interval` Calcule la moyenne (arithmétique) de toutes les valeurs en entrée, non NULL.	
`bit_and ( smallint ) → smallint` `bit_and ( integer ) → integer` `bit_and ( bigint ) → bigint` `bit_and ( bit ) → bit` Calcule un AND bit à bit de toutes les valeurs en entrée non NULL.	Yes
`bit_or ( smallint ) → smallint` `bit_or ( integer ) → integer` `bit_or ( bigint ) → bigint` `bit_or ( bit ) → bit` Calcule un OR bit à bit de toutes les valeurs en entrée non NULL.	Yes
`bit_xor ( smallint ) → smallint` `bit_xor ( integer ) → integer` `bit_xor ( bigint ) → bigint` `bit_xor ( bit ) → bit` Calcule un OR exclusif bit à bit de toutes les valeurs en entrée non NULL. Peut être utile comme somme de contrôle pour un ensemble de valeurs non ordonnées.	Yes
`bool_and ( boolean ) → boolean` Renvoie true si toutes les valeurs en entrée non NULL valent true, sinon false.	Yes
`bool_or ( boolean ) → boolean` Renvoie true si au moins une des valeurs en entrée non NULL vaut true, sinon false.	Yes
`count ( * ) → bigint` Calcule le nombre de lignes en entrée.	Yes
`count ( "any" ) → bigint` Calcule le nombre de lignes en entrée pour lesquelles la valeur n'est pas NULL.	Yes
`every ( boolean ) → boolean` Ceci est l'équivalent de `bool_and` pour le standard SQL.	Yes
`json_agg ( anyelement ) → json` `jsonb_agg ( anyelement ) → jsonb` Récupère toutes les valeurs en entrée, y compris les NULL, et les place dans un tableau JSON. Les valeurs sont converties en JSON avec `to_json` ou `to_jsonb`.	No
`json_object_agg ( key "any", value "any" ) → json` `jsonb_object_agg ( key "any", value "any" ) → jsonb`	No

Fonction Description	Mode Partiel
Récupère toutes les paires clé/valeur et les place dans un objet JSON. Les arguments clé sont convertis en texte ; les arguments valeur sont convertis avec `to_json` ou `to_jsonb`. Les valeurs peuvent être NULL, mais pas les clés.	
`max` (*see text*) → *same as input type* Calcule la valeur maximale de toutes les valeurs en entrée non NULL. Disponible pour les types numeric, string, date/time ou enum type, ainsi que `inet`, `interval`, `money`, `oid`, `pg_lsn`, `tid`, `xid8` et les tableaux de chacun de ces types.	Yes
`min` (*see text*) → *same as input type* Calcule la valeur minimale de toutes les valeurs en entrée non NULL. Disponible pour les types numeric, string, date/time ou enum type, ainsi que `inet`, `interval`, `money`, `oid`, `pg_lsn`, `tid`, `xid8` et les tableaux de chacun de ces types.	Yes
`range_agg` (*value* `anyrange`) → `anymultirange` `range_agg` (*value* `anymultirange`) → `anymultirange` Calcule l'union des valeurs non NULL en entrée.	No
`range_intersect_agg` (*value* `anyrange`) → `anyrange` `range_intersect_agg` (*value* `anymultirange`) → `anymultirange` Calcule l'intersection des valeurs non NULL en entrée.	No
`string_agg` (*value* `text`, *delimiter* `text`) → `text` `string_agg` (*value* `bytea`, *delimiter* `bytea`) → `bytea` Concatène les valeurs en entrée non NULL dans une chaîne. Chaque valeur après la première est précédée par le *delimiter* correspondant (s'il n'est pas NULL).	No
`sum` (`smallint`) → `bigint` `sum` (`integer`) → `bigint` `sum` (`bigint`) → `numeric` `sum` (`numeric`) → `numeric` `sum` (`real`) → `real` `sum` (`double precision`) → `double precision` `sum` (`interval`) → `interval` `sum` (`money`) → `money` Calcule la somme de toutes les valeurs en entrée non NULL.	Yes
`xmlagg` (`xml`) → `xml` Concatène toutes les valeurs XML en entrée non NULL (voir Section 9.15.1.7).	No

Il doit être noté que, sauf pour `count`, ces fonctions renvoient NULL quand aucune ligne n'est sélectionnée. En particulier, le `sum` d'aucune ligne renvoie NULL, et non pas zéro comme certains s'y attendraient, et `array_agg` renvoie NULL plutôt qu'un tableau vide s'il n'y a pas de lignes en entrée. La fonction `coalesce` peut être utilisée pour substituer une valeur NULL à zéro ou un tableau vide quand cela s'avère nécessaire.

Les fonctions d'agrégat `array_agg`, `json_agg`, `jsonb_agg`, `json_object_agg`, `jsonb_object_agg` `string_agg` et `xmlagg`, ainsi que les fonctions d'agrégat utilisateurs similaires, produisent des valeurs de résultat différentes suivant l'ordre des valeurs en entrée. Cet ordre n'est pas spécifié par défaut, mais peut être contrôlé en écrivant une clause `ORDER BY` dans l'appel

à l'agrégat, comme indiqué dans Section 4.2.7. Autrement, fournir les valeurs en entrée triées à partir d'une sous-requête fonctionne généralement. Par exemple :

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

Attention que cette approche peut échouer si le niveau de la requête externe contient un traitement supplémentaire, tel qu'une jointure, parce que cela pourrait remettre en cause la sortie de la sous-requête et notamment un nouveau tri avant le calcul de l'agrégat.

> **Note**
>
> Les agrégats booléens `bool_and` et `bool_or` correspondent aux agrégats du standard SQL `every` et `any` ou `some`. PostgreSQL accepte `every`, mais pas `any` et `some`, car il y a une ambiguité dans la syntaxe standard :
>
> ```
> SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
> ```
>
> Ici, `ANY` peut être considéré soit comme introduisant une sous-requête, soit comme étant une fonction d'agrégat, si la sous-requête renvoie une ligne avec une valeur booléenne. De ce fait, le nom standard ne peut pas être donné à ces agrégats.

> **Note**
>
> Les utilisateurs habitués à travailler avec d'autres systèmes de gestion de bases de données pourraient être déçus par les performances de l'agrégat `count` quand il est appliqué à la table entière. Une requête comme :
>
> ```
> SELECT count(*) FROM sometable;
> ```
>
> nécessite un effort proportionnel à la taille de la table : PostgreSQL aura besoin de parcourir soit toute la table soit tout un index qui inclut toutes les lignes de la table.

Tableau 9.59 montre les fonctions d'agrégat typiquement utilisées dans les analyses statistiques. (Elles sont séparées principalement pour éviter d'encombrer la liste des agrégats plus fréquemment utilisés.) Les fonctions acceptant *`numeric_type`* sont disponibles pour les types `smallint`, `integer`, `bigint`, `numeric`, `real` et `double precision`. Quand la description mentionne *`N`*, cela signifie le nombre de lignes en entrée pour lesquelles les expressions en entrée ne sont pas NULL. Dans tous les cas, NULL est renvoyé si le calcul n'a pas de sens, par exemple quand *`N`* vaut zéro.

Tableau 9.59. Fonctions d'agrégat pour les statistiques

Fonction Description	Mode Partiel
`corr(`*`Y`* `double precision,` *`X`* `double precision)` → `double precision` Calcule le coefficient de corrélation.	Yes
`covar_pop(`*`Y`* `double precision,` *`X`* `double precision)` → `double precision` Calcule la covariance de population.	Yes

Fonction Description	Mode Partiel
`covar_samp` (*Y* `double precision`, *X* `double precision`) → `double precision` Calcule la covariance d'échantillon.	Yes
`regr_avgx` (*Y* `double precision`, *X* `double precision`) → `double precision` Calcule la moyenne de la variable indépendante, `sum(`*X*`)/`*N*.	Yes
`regr_avgy` (*Y* `double precision`, *X* `double precision`) → `double precision` Calcule la moyenne de la variable dépendante, `sum(`*Y*`)/`*N*.	Yes
`regr_count` (*Y* `double precision`, *X* `double precision`) → `bigint` Calcule le nombre de lignes pour lesquelles les deux entrées sont non NULL.	Yes
`regr_intercept` (*Y* `double precision`, *X* `double precision`) → `double precision` Calcule l'ordonnée à l'origine de l'équation d'ajustement linéaire par les moindres carrés déterminée par les paires de valeurs (*X*, *Y*).	Yes
`regr_r2` (*Y* `double precision`, *X* `double precision`) → `double precision` Calcule le carré du coefficient de corrélation.	Yes
`regr_slope` (*Y* `double precision`, *X* `double precision`) → `double precision` Calcule la pente de la droite de l'équation d'ajustement linéaire par les moindres carrés déterminée par les paires de valeurs (*X*, *Y*).	Yes
`regr_sxx` (*Y* `double precision`, *X* `double precision`) → `double precision` Calcule la « somme des carrés » de la variable indépendante, `sum(`*X*`^2) - sum(`*X*`)^2/`*N*.	Yes
`regr_sxy` (*Y* `double precision`, *X* `double precision`) → `double precision` Calcule la « somme des produits » des variables dépendantes heures indépendantes, `sum(`*X*`*`*Y*`) - sum(`*X*`) * sum(`*Y*`)/`*N*.	Yes
`regr_syy` (*Y* `double precision`, *X* `double precision`) → `double precision` Calcule la « somme des carrés » de la variable dépendante, `sum(`*Y*`^2) - sum(`*Y*`)^2/`*N*.	Yes
`stddev` (*`numeric_type`*) → `double precision` pour `real` ou `double precision`, sinon `numeric` Alias historique pour `stddev_samp`.	Yes
`stddev_pop` (*`numeric_type`*) → `double precision` pour `real` ou `double precision`, sinon `numeric` Calcule la déviation standard de la population pour les valeurs en entrée.	Yes
`stddev_samp` (*`numeric_type`*) → `double precision` pour `real` ou `double precision`, sinon `numeric` Calcule la déviation standard d'échantillon des valeurs en entrée.	Yes
`variance` (*`numeric_type`*) → `double precision` pour `real` ou `double precision`, sinon `numeric`	Yes

Fonction Description	Mode Partiel
Alias historique pour `var_samp`.	
`var_pop` (*`numeric_type`*) → `double precision` pour `real` ou `double precision`, sinon `numeric` Calcule la variance de la population pour les valeurs en entrée (carré de la déviation standard de la population).	Yes
`var_samp` (*`numeric_type`*) → `double precision` for `real` or `double precision`, otherwise `numeric` Calcule la variance d'échantillon des valeurs en entrée (carré de la déviation standard d'échantillon).	Yes

Tableau 9.60 montre certaines fonctions d'agrégat utilisant la syntaxe d'*agrégat à ensemble ordonné*. Ces fonctions sont parfois nommées fonctions à « distribution inverse ». Leur entrée est introduite par `ORDER BY`, et elles peuvent aussi prendre un *argument direct* qui n'est pas agrégé mais calculé une seule fois. Toutes ces fonctions ignorent les valeurs NULL dans leur entrée agrégée. Pour celles qui prennent un paramètre *`fraction`*, la valeur de la fraction est comprise entre 0 et 1 ; une erreur est renvoyée dans le cas contraire. Néanmoins une valeur de *`fraction`* NULL donne simplement un résultat NULL.

Tableau 9.60. Fonctions d'agrégat à ensemble trié

Fonction Description	Mode Partiel
`mode` () `WITHIN GROUP` (`ORDER BY` `anyelement`) → `anyelement` Calcule le *mode*, la valeur la plus fréquente d'un argument agrégé (en choisissant arbitrairement le premier s'il y a plusieurs valeurs de fréquence égale). L'argument agrégé doit être d'un type triable.	No
`percentile_cont` (*`fraction`* `double precision`) `WITHIN GROUP` (`ORDER BY` `double precision`) → `double precision` `percentile_cont` (*`fraction`* `double precision`) `WITHIN GROUP` (`ORDER BY` `interval`) → `interval` Calcule le *pourcentage continu*, une valeur correspondant à la *`fraction`* à l'intérieur d'un ensemble ordonné de valeurs d'argument agrégées. Ceci va interpoler entre les éléments en entrée adjacents si nécessaire.	No
`percentile_cont` (*`fractions`* `double precision[]`) `WITHIN GROUP` (`ORDER BY` `double precision`) → `double precision[]` `percentile_cont` (*`fractions`* `double precision[]`) `WITHIN GROUP` (`ORDER BY` `interval`) → `interval[]` Calcule plusieurs pourcentages continues. Le résultat est un tableau de mêmes dimensions que le paramètre *`fractions`*, avec chaque élément non NULL remplacé par la valeur (potentiellement interpolée) correspondance à ce pourcentage.	No
`percentile_disc` (*`fraction`* `double precision`) `WITHIN GROUP` (`ORDER BY` `anyelement`) → `anyelement` Calcule le *pourcentage discret*, la première valeur avec l'ensemble de tri des valeurs d'argument agrégé dont la position dans le tri est égale ou dépasse la *`fraction`* indiquée. L'argument agrégé doit être d'un type triable.	No
`percentile_disc` (*`fractions`* `double precision[]`) `WITHIN GROUP` (`ORDER BY` `anyelement`) → `anyarray` Calcule plusieurs pourcentages discrets. Le résultat est un tableau de mêmes dimensions que le paramètre *`fractions`*, avec chaque élément non NULL	No

Fonction Description	Mode Partiel
remplacé par la valeur en entrée correspondant à ce pourcentage. L'argument agrégé doit être d'un type triable.	

Chacun des agrégats d'« ensemble hypothétique » listés dans Tableau 9.61 est associé avec une fonction de fenêtrage du même nom définie dans Section 9.22. Dans chaque cas, le résultat de l'agrégat est la valeur que la fonction de fenêtrage associée aurait renvoyée pour la ligne « hypothétique » construite à partir de *args*, si une telle ligne a été ajoutée au groupe trié de lignes représenté par *sorted_args*. Pour chacune de ces fonctions, la liste des arguments directs donnée dans *args* doit correspondre au nombre et types des arguments agrégés donnés dans *sorted_args*. Contrairement à la plupart des agrégats natifs, ces agrégats ne sont pas stricts, c'est-à-dire qu'ils ne suppriment pas les lignes en entrée contenant des NULL. Les valeurs NULL se trient suivant la règle indiquée dans la clause ORDER BY.

Tableau 9.61. Fonctions d'agrégat à ensemble hypothétique

Fonction Description	Mode Partiel
rank (*args*) WITHIN GROUP (ORDER BY *sorted_args*) → bigint Calcule le rang de la ligne hypothétique avec des trous ; c'est-à-dire le numéro de la première ligne dans son groupe.	No
dense_rank (*args*) WITHIN GROUP (ORDER BY *sorted_args*) → bigint Calcule le rang de la ligne hypothétique sans trous ; cette fonction compte réellement le nombre de groupes.	No
percent_rank (*args*) WITHIN GROUP (ORDER BY *sorted_args*) → double precision Calcule le rang relatif de la ligne hypothétique, c'est-à-dire (rank - 1) / (lignes totales - 1). La valeur est donc comprise entre 0 et 1, les deux étant inclus.	No
cume_dist (*args*) WITHIN GROUP (ORDER BY *sorted_args*) → double precision Calcule la distribution cumulative, c'est-à-dire (nombre de lignes précédentes ou proches de la ligne hypothétique) / (total de lignes). La valeur est donc comprise entre 1/*N* et 1.	No

Tableau 9.62. Opérations de regroupement

Fonction Description
GROUPING (*group_by_expression(s)*) → integer Renvoie un masque de bits indiquant les expressions GROUP BY non incluses dans l'ensemble de regroupement actuel. Les bits sont affectés avec l'argument le plus à droit correspondant au bit de plus faible poids ; chaque bit vaut 0 si l'expression correspondante est inclus dans le critère de regroupement de l'ensemble de regroupement générant la ligne résultat actuelle, et 1 dans le cas contraire.

Les opérations de regroupement affichées dans Tableau 9.62 sont utilisées en conjonction avec les ensembles de regroupement (voir Section 7.2.4) pour distinguer les lignes résultats. Les arguments à la fonction GROUPING ne sont pas réellement évalués car ils doivent correspondre exactement aux expressions données dans la clause GROUP BY du niveau de requête associé. Par exemple :

```
=> SELECT * FROM items_sold;
 make  | model | sales
-------+-------+-------
```

```
 Foo   | GT    |  10
 Foo   | Tour  |  20
 Bar   | City  |  15
 Bar   | Sport |  5
(4 rows)

=> SELECT make, model, GROUPING(make,model), sum(sales) FROM
 items_sold GROUP BY ROLLUP(make,model);
 make  | model | grouping | sum
-------+-------+----------+-----
 Foo   | GT    |        0 | 10
 Foo   | Tour  |        0 | 20
 Bar   | City  |        0 | 15
 Bar   | Sport |        0 | 5
 Foo   |       |        1 | 30
 Bar   |       |        1 | 20
       |       |        3 | 50
(7 rows)
```

Ici, la valeur `0` dans la colonne `grouping` des quatre premières lignes montre qu'elles ont été regroupées normalement par rapport aux colonnes de regroupement. La valeur `1` indique que la colonne `model` n'a pas été groupé dans les deux lignes suivantes, et la valeur `3` indique que ni la colonne `make` ni la colonne `model` n'ont été regroupées dans la dernière ligne (qui, de ce fait, est un agrégat sur tous les lignes en entrée).

9.22. Fonctions de fenêtrage

Les *fonctions de fenêtrage* fournissent des moyens pour réaliser des calculs sur des ensembles de lignes relatives à la ligne actuelle de la requête. Voir Section 3.5 pour une introduction à cette fonctionnalité, et Section 4.2.8 pour les détails sur la syntaxe.

Les fonctions de fenêtrage natives sont montrées dans Tableau 9.63. Notez que ces fonctions *doivent* être appelées en utilisant la syntaxe des fonctions de fénêtrage, c'est-à-dire en utilisant une clause `OVER`.

En plus de ces fonctions, toute fonction d'agrégat standard native ou définie par un utilisateur (donc pas les agrégats à ensemble ordonné ou à ensemble hypothétique) peut être utilisée comme une fonction de fenêtrage ; voir Section 9.21 pour une liste des agrégats natifs. Les fonctions d'agrégat agissent comme des fonctions de fenêtrage quand une clause `OVER` est utilisée pour l'appel ; sinon elles agissent comme des agrégats standards et renvoient une seule ligne pour un ensemble complet.

Tableau 9.63. Fonctions de fenêtrage à usage général

Fonction Description
`row_number` () → `bigint` Renvoie le nombre de la ligne courante dans sa partition, en commençant à 1.
`rank` () → `bigint` Renvoie le range de la ligne courante, avec des trous ; c'est-à-dire le `row_number` de la première ligne dans son groupe.
`dense_rank` () → `bigint` Renvoie le range de la ligne courante, sans trous ; cette fonction compte réellement les groupes.
`percent_rank` () → `double precision` Renvoie le rang relatif de la ligne courante, c'est-à-dire (`rank` - 1) / (total de lignes dans la partition - 1). La valeur est donc comprise entre 0 et 1, les deux inclus.

Fonction Description
`cume_dist` () → `double precision` Renvoie la distribution cumulative, c'est-à-dire (nombre de lignes dans la partition précédente ou nombre de groupes à partir de la ligne courante) / (total de lignes dans la partition). La valeur est donc comprise entre 1/*N* et 1.
`ntile` (*num_buckets* `integer`) → `integer` Renvoie un entier entre 1 et la valeur argument, divisant la partition aussi également que possible.
`lag` (*value* `anycompatible` [, *offset* `integer` [, *default* `anycompatible`]]) → `anycompatible` Renvoie *value* évaluée à la ligne qui se trouve à *offset* lignes avant la ligne actuelle dans la partition ; si une telle ligne n'existe pas, renvoie *default* à la place (qui doit être d'un type compatible avec *value*). *offset* et *default* sont évalués suivant la ligne actuelle. Par défaut, *offset* vaut 1 et *default* vaut `NULL`.
`lead` (*value* `anycompatible` [, *offset* `integer` [, *default* `anycompatible`]]) → `anycompatible` Renvoie *value* évaluée à la ligne qui se trouve à *offset* lignes après la ligne actuelle dans la partition ; si une telle ligne n'existe pas, renvoie *default* à la place (qui doit être d'un type compatible que *value*). *offset* et *default* sont évalués suivant la ligne actuelle. Par défaut, *offset* vaut 1 et *default* vaut `NULL`.
`first_value` (*value* `anyelement`) → `anyelement` Renvoie *value* évaluée à la première ligne de la fenêtre.
`last_value` (*value* `anyelement`) → `anyelement` Renvoie *value* évaluée à la dernière ligne de la fenêtre.
`nth_value` (*value* `anyelement`, *n* `integer`) → `anyelement` Renvoie *value* évaluée à la ligne qui se trouve à la *n*-ième ligne de la fenêtre (en comptant à partir de 1) ; renvoie `NULL` si cette ligne n'existe pas.

Toutes les fonctions listées dans Tableau 9.63 dépendent de l'ordre de tri indiqué par la clause `ORDER BY` sur la définition de la fenêtre associée. Les lignes qui ne sont pas distinctes lors de la seule considération des colonnes `ORDER BY` sont des *peers*. Les quatre fonctions de rang (y compris `cume_dist`) sont définies pour qu'elles donnent la même réponse pour toutes les lignes d'un groupe peer.

Notez que les fonctions `first_value`, `last_value` et `nth_value` ne prennent en compte que les lignes à l'intérieur d'une « fenêtre », qui, par défaut, contient les lignes du début de la partition jusqu'au dernier peer de la ligne actuelle. Ceci risque de donner des résultats sans intérêt pour `last_value` et quelques fois aussi pour `nth_value`. Vous pouvez redéfinir la fenêtre en ajoutant une spécification convenable (`RANGE`, `ROWS` ou `GROUPS`) à la clause `OVER`. Voir Section 4.2.8 pour plus d'informations.

Quand une fonction d'agrégat est utilisée comme fonction de fenêtrage, elle agrège les lignes dans la fenêtre de la ligne courante. Un agrégat utilisé avec `ORDER BY` et la définition par défaut de la fenêtre produit un type de comportement du style « somme mouvante », qui pourrait être ou pas ce qui est souhaité. Pour obtenir l'agrégat sur la partition complète, n'utilisez pas `ORDER BY` ou utilisez `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`. Les autres spécifications de fenêtre peuvent être utilisées pour obtenir d'autres effets.

> **Note**
>
> Le standard SQL définit une option `RESPECT NULLS` ou `IGNORE NULLS` pour les fonctions `lead`, `lag`, `first_value`, `last_value` et `nth_value`. Ceci n'est pas couvert dans

PostgreSQL : le comportement est toujours identique au comportement par défaut du standard, autrement dit `RESPECT NULLS`. De même, les options du standard `FROM FIRST` et `FROM LAST` pour `nth_value` ne sont pas supportées : seul le comportement par défaut, `FROM FIRST`, l'est. (Vous pouvez obtenir le résultat de `FROM LAST` en inversant le tri `ORDER BY`.)

9.23. Expressions de sous-requêtes

Cette section décrit les expressions de sous-requêtes, compatible SQL, et disponible dans PostgreSQL. Toutes les formes d'expression documentées dans cette section renvoient des résultats booléens (true/false).

9.23.1. EXISTS

`EXISTS (`*`subquery`*`)`

L'argument de `EXISTS` est une requête `SELECT` arbitraire ou une *sous-requête*. La sous-requête est évaluée pour déterminer si elle renvoie des lignes. Si elle renvoie au moins une ligne, le résultat de `EXISTS` est « true » ; si la sous-requête ne renvoie aucune ligne, le résultat de `EXISTS` est « false ».

La sous-requête peut faire référence à des variables provenant de la requête englobante, qui agira comme constantes lors de toute évaluation de la sous-requête.

La sous-requête s'exécutera seulement suffisamment longtemps pour déterminer qu'au moins une ligne sera renvoyée. Elle ne sera donc pas exécuté entièrement. Il est déconseillé d'écrire une sous-requête qui a des effets secondaires (tels qu'appeler des fonctions de séquence) le fait que les effets secondaires surviennent n'est pas prévisible.

Comme le résultat dépend seulement du fait que des lignes sont renvoyées ou pas, et non pas sur le contenu de ces lignes, la liste en sortie de la sous-requête n'a pas d'importance. Une convention commune de codage est d'écrire tous les tests `EXISTS` sous la forme `EXISTS(SELECT 1 WHERE ...)`. Il existe néanmoins des exceptions à cette règle, comme les sous-requêtes qui utilisent `INTERSECT`.

Cet exemple simple est comme une jointure interne sur `col2`, mais il produit au plus une ligne pour chaque ligne de `tab1`, même s'il existe plus lignes correspondantes dans `tab2` :

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.23.2. IN

`expression` `IN (`*`subquery`*`)`

Le côté droit est une sous-requête entre parenthèses, qui doit renvoyer exactement une ligne. L'expression côté gauche est évaluée et comparée à chaque ligne du résultat de la sous-requête. Le résultat de `IN` vaut « true » si au moins une ligne identique est trouvée dans la sous-requête. Le résultat est « false » si aucune ligne identique n'a été trouvée (y compris dans le cas où la sous-requête ne renvoie aucune ligne).

Notez que si l'expression côté gauche renvoie NULL ou s'il y a aucune valeur identique côté droit et au moins une ligne renvoyant NULL côté droit, le résultat de la construction `IN` sera NULL, et non pas false. Ceci est en accord avec les règles normales du SQL pour les combinaisons booléennes de valeurs NULL.

Comme avec EXISTS, il est déconseillé de supposer que la sous-requête sera complètement évaluée.

row_constructor IN (*subquery*)

Le côté gauche de cette syntaxe de IN est un constructeur de ligne, comme décrit dans Section 4.2.13. Le côté droit est une sous-requête entre parenthèses, qui doit renvoyer exactement autant de colonnes qu'il y a d'expressions côté gauche. Les expressions du côté gauche sont évaluées et comparées, ligne par ligne, à chaque ligne du résultat de la sous-requête. Le résultat de IN vaut « true » si au moins une ligne identique est trouvée dans la sous-requête. Le résultat est « false » si aucune ligne identique n'est trouvée (y compris le cas où la sous-requête ne renvoie aucune ligne).

Comme d'habitude, les valeurs NULL dans les lignes sont combinées par les règles habituelles des expressions SQL booléennes. Deux lignes sont considérées égales si tous leurs membres correspondance sont non NULL et égaux ; les lignes ne sont pas égales si au moins un membre correspondant est non NULL et différent ; sinon le résultat de cette comparaison de ligne est inconnue (NULL). Si tous les résultats ligne par ligne sont soit différents soit NULL, avec au moins un NULL, alors le résultat de IN est NULL.

9.23.3. NOT IN

expression NOT IN (*subquery*)

Le côté droit est une sous-requête entre parenthèses, qui doit renvoyer exactement une colonne. L'expression côté gauche est évaluée et comparée à chaque ligne du résultat de la sous-requête. Le résultat de NOT IN vaut « true » si seulement des lignes différentes sont trouvées dans la sous-requête (y compris dans le cas où la sous-requête ne renvoie pas de lignes). Le résultat vaut « false » si au moins une ligne identique est trouvée.

Notez que si l'expression côté gauche renvoie NULL ou s'il n'existe pas de valeurs identiques côté droit et qu'au moins une ligne côté droit renvoie NULL, le résultat de la construction NOT IN sera NULL, et non pas true. Ceci est en accord avec les règles standards du SQL pour les combinaisons booléennes de valeurs NULL.

Tout comme EXISTS, il est déconseillé de supposer que la sous-requête sera totalement évaluée.

row_constructor NOT IN (*subquery*)

Le côté gauche de cette syntaxe pour NOT IN est un constructeur de lignes, comme décrit dans Section 4.2.13. Le côté droit est une sous-requête entre parenthèses, qui doit renvoyer exactement autant de colonnes qu'il y a d'expressions du côté gauche. Les expressions du côté gauche sont évaluées et comparées ligne par ligne pour chaque ligne du résultat de la sous-requête. Le résultat de NOT IN vaut « true » si seules des lignes différentes sont trouvées dans la sous-requête (y compris dans le cas où la sous-requête ne renvoie aucune ligne). Le résultat vaut « false » si aucune ligne identique n'est trouvée.

Comme d'habitude, les valeurs NULL dans les lignes sont combinées d'après les règles standards en SQL pour les expressions booléennes. Deux lignes sont considérées identiques si tous leurs membres correspondant sont non NULL et égaux ; les lignes sont différentes si au moins un membre correspondant est non NULL et différent ; sinon le résultat de cette comparaison de lignes est inconnu (NULL). Si tous les résultats ligne par ligne sont soit différentes soit NULL, avec au moins un NULL, alors le résultat de NOT IN is NULL.

9.23.4. ANY/SOME

expression operator ANY (*subquery*)
expression operator SOME (*subquery*)

Le côté droit est une sous-requête entre parenthèses, qui doit renvoyer exactement une colonne. L'expression côté gauche est évaluée et comparée à chaque ligne du résultat de la sous-requête en utilisant l'opérateur *operator* indiquée, qui doit renvoyer un résultat booléen. Le résultat de ANY vaut « true » si au moins une valeur true est obtenue dans le résultat. Le résultat vaut « false » si aucun résultat true n'est trouvé (y compris dans le cas où la sous-requête ne renvoie aucune ligne).

SOME est un synonyme pour ANY. IN est équivalent à = ANY.

Notez qe s'il n'y a pas de succès et qu'au moins une ligne côté droit renvoie NULL comme résultat de l'opérateur, le résultat de la construction ANY sera NULL, et non pas false. Ceci est en accord avec les règles SQL habituelles sur les combinaisons booléennes de valeurs NULL.

Comme avec EXISTS, il est déconseillé de supposer que la sous-requête sera totalement évaluée.

row_constructor operator ANY (*subquery*)
row_constructor operator SOME (*subquery*)

Le côté gauche de cette syntaxe de ANY est un constructeur de lignes, comme décrit dans Section 4.2.13. Le côté droit est une sous-requête entre parenthèses, qui doit renvoyer autant de colonnes qu'il y a d'expressions sur la ligne côté gauche. Les expressions côté gauche sont évaluées et comparées ligne par ligne pour chaque ligne du résultat de la sous-requête, en utilisant l'opérateur *operator* indiqué. Le résultat de ANY vaut « true » si la comparaison renvoie true pour au moins une ligne de la sous-requête. Le résultat vaut « false » si la comparaison renvoie false pour chaque ligne de la sous-requête (y compris dans le cas où la sous-requête ne renvoie aucune ligne). Le résultat est NULL si aucune comparaison avec une ligne de la sous-requête ne renvoie true, et qu'au mpoins une comparaison renvoie NULL.

Voir Section 9.24.5 pour les détails sur la signification de la comparaison d'un constructeur de ligne.

9.23.5. ALL

expression operator ALL (*subquery*)

Le côté droit est une sous-requête entre parenthèses, qui doit renvoyer exactement une colonne. L'expression côté gauche est évaluée, et comparée à chaque ligne du résultat de la sous-requête en utilisant l'opérateur *operator* indiqué, qui doit renvoyer un résultat booléen. Le résultat de ALL vaut « true » si toutes les lignes renvoient true (y compris dans le cas où la sous-requête ne renvoie aucune ligne). Le résultat vaut « false » si un résultat false est trouvé. Le résultat est NULL si aucune comparaison avec une ligne de la sous-requête renvoie false et au moins une comparaison renvoie NULL.

NOT IN est équivalent à <> ALL.

Comme avec EXISTS, il est déconseillé de supposer que la sous-requête sera totalement évaluée.

row_constructor operator ALL (*subquery*)

Le côté gauche de cette forme de ALL est un constructeur de lignes, comme décrit dans Section 4.2.13. Le côté droit est une sous-requête entre parenthèses, qui doit renvoyer exactement autant de colonnes qu'il y a d'expressions dans la ligne côté gauche. Les expressions côté gauche sont évaluées et comparées ligne par ligne à chaque ligne du résultat de la sous-requête, en utilisant l'opérateur *operator* donné. Le résultat de ALL vaut « true » si la comparaison renvoie true pour toutes les lignes de la sous-requête (y compris le cas où la sous-requête ne renvoie aucune ligne). Le résultat est « false » si la comparaison renvoie false pour une ligne de la sous-requête. Le résultat est NULL si aucune comparaison avec une ligne de la sous-requête renvoie false et au moins une comparaison renvoie NULL.

Voir Section 9.24.5 pour les détails sur la signification d'une comparaison de constructeur de ligne.

9.23.6. Comparaison de ligne simple

row_constructor `operator (`*subquery*`)`

Le côté gauche est un constructeur de lignes, comme décrit dans Section 4.2.13. Le côté droit est une sous-requête entre parenthèses, qui doit renvoyer exactement autant de colonnes qu'il y a d'expressions dans la ligne côté gauche. De plus, la sous-requête ne peut pas renvoyer plus d'une ligne. (Si elle renvoie zéro ligne, le résultat est pris pour NULL.) Le côté gauche est évalué et comparé ligne par ligne à la ligne seule du résultat de la sous-requête.

Voir Section 9.24.5 pour plus de détails sur la signification d'une comparaison de constructeur de lignes.

9.24. Comparaisons de ligne et de tableau

Cette section décrit plusieurs constructions spécialisées pour réaliser plusieurs comparaisons entre des groupes de valeurs. Ces formes sont syntaxiquement en relation avec les formes de sous-requêtes de la section précédente, mais n'impliquent pas de sous-requêtes. Les formes qui impliquent des sous-expressions de tableau sont des extensions PostgreSQL ; le reste est compatible avec la norme SQL. Toutes les formes d'expression documentées dans cette section renvoient des résultats booléens (true/false).

9.24.1. `IN`

expression `IN (`*value* `[, ...])`

Le côté droit est une liste entre parenthèses d'expressions. Le résultat est « true » si le résultat de l'expression côté gauche est égal à une des expressions côté droit. Cette notation est un raccourci pour

```
expression = value1
OR
expression = value2
OR
...
```

Notez que si l'expression côté gauche renvoie NULL ou s'il n'existe pas de valeurs identiques côté droit et qu'au moins une expression côté droit renvoie NULL, le résultat de la construction `IN` sera NULL, et non pas false. Ceci est en accord avec les règles SQL habituelles pour les combinaisons booléennes de valeurs NULL.

9.24.2. `NOT IN`

expression `NOT IN (`*value* `[, ...])`

Le côté droit est une liste entre parenthèses d'expressions. Le résultat est « true » si le résultat de l'expression côté gauche est différent à toutes les expressions côté droit. Cette notation est un raccourci pour

```
expression <> value1
AND
expression <> value2
AND
...
```

Notez que si l'expression côté gauche renvoie NULL ou s'il n'existe pas de valeurs identiques côté droit et qu'au moins une expression côté droit renvoie NULL, le résultat de la construction `NOT IN` sera NULL, et non pas true comme on pourrait s'y attendre. Ceci est en accord avec les règles SQL habituelles pour les combinaisons booléennes de valeurs NULL.

> **Astuce**
>
> `x NOT IN y` est équivalent à `NOT (x IN y)` dans tous les cas. Néanmoins, les valeurs NULL risquent bien plus souvent de tromper le novice lorsqu'il travaille avec `NOT IN` que quand il travaille avec `IN`. Il est préférable d'exprimer votre condition de façon positive si possible.

9.24.3. ANY/SOME (array)

```
expression operator ANY (array expression)
expression operator SOME (array expression)
```

Le côté droit est une expression entre parenthèses, qui doit renvoyer une valeur de type tableau. L'expression côté gauche est évaluée et comparée à chaque élément du tableau en utilisant l'opérateur *`operator`* indiqué, qui doit renvoyer un résultat booléen. Le résultat de `ANY` vaut « true » si aucun résultat true n'est obtenu. Le résultat vaut « false » si aucun résultat true n'est trouvé (y compris dans le cas où le tableau a zéro élément).

Si l'expression de tableau renvoie un tableau NULL, le résultat de `ANY` sera NULL. Si l'expression côté gauche renvoie NULL, le résultat de `ANY` est habituellement NULL (bien qu'un opérateur de comparaison non strict pourrait renvoyer un résultat différent). De plus, si le tableau côté droit contient des éléments NULL et qu'aucun résultat de la comparaison ne renvoie true, le résultat d' `ANY` sera NULL, et non pas false (encore une fois, en supposant un opérateur de comparaison strict). Ceci est en accord avec les règles SQL habituelles pour la combinaison booléenne de valeurs NULL.

`SOME` est un synonyme pour `ANY`.

9.24.4. ALL (array)

```
expression operator ALL (array expression)
```

Le côté droit est une expression entre parenthèses, qui doit renvoyer une valeur de type tableau. L'expression côté gauche est évaluée et comparée à chaque élément du tableau en utilisant l'opérateur *`operator`* indiqué, qui doit renvoyer un résultat booléen. Le résultat de `ALL` vaut « true » si toutes les comparaisons ramènent true (y compris dans le cas où le tableau a zéro élément). Le résultat vaut « false » si au moins un résultat false est trouvé.

Si l'expression de tableau renvoie un tableau NULL, le résultat de `ALL` sera NULL. Si l'expression côté gauche renvoie NULL, le résultat de `ALL` est habituellement NULL (bien qu'un opérateur de comparaison non strict pourrait renvoyer un résultat différent). De plus, si le tableau côté droit contient au moins un élément NULL et qu'aucun résultat de la comparaison ne renvoie false, le résultat de `ALL` sera NULL, et non pas true (encore une fois, en supposant un opérateur de comparaison strict). Ceci est en accord avec les règles SQL habituelles pour la combinaison booléenne de valeurs NULL.

9.24.5. Comparaison de constructeur de lignes

```
row_constructor operator row_constructor
```

Chaque côté est un constructeur de lignes, comme décrit dans Section 4.2.13. Les deux constructeurs de lignes doivent avoir le même nombre de champs. L'*opérateur* indiqué est appliqué à chaque paire de champs correspondant. (Comme les champs pourraient être de types différents, ceci signifie qu'un opérateur spécifique différent pourrait être sélectionné pour chaque paire.) Tous les opérateurs sélectionnés doivent être les membres d'une classe d'opérateur B-tree ou être l'inverse d'un membre = d'une classe d'opérateur B-tree, ceci signifiant que la comparaison de constructeur de lignes est seulement possible quand l'opérateur *operator* est =, <>, <, <=, > ou >=, ou a une sémantique similaire à l'une d'entre elles.

Les cas = et <> fonctionnent légèrement différemment des autres. Les deux lignes sont considérées égales si tous leurs membres correspondant sont non NULL et égaux ; les lignes sont différentes si un des membres correspondant est non NULL et différent ; sinon le résultat de la comparaison est inconnu (NULL).

Pour les cas <, <=, > et >=, les éléments de ligne sont comparés de gauche à droite, en s'arrêtant dès qu'une paire d'éléments différents ou NULL est trouvée. Si un des éléments est NULL, le résultat de la comparaison est inconnu (NULL) ,; sinon la comparaison de cette paire d'éléments détermine le résultat. Par exemple, `ROW(1,2,NULL) < ROW(1,3,0)` renvoie true, et non pas NULL, parce que la troisième paire d'éléments n'est pas considérée.

> **Note**
>
> Avant PostgreSQL 8.2, les cas <, <=, > et >= n'étaient pas gérés par la spécification SQL. Une comparaison comme `ROW(a,b) < ROW(c,d)` était implémentée comme `a < c AND b < d` alors que le comportement correct est équivalent à `a < c OR (a = c AND b < d)`.

row_constructor `IS DISTINCT FROM` *row_constructor*

Cette construction est similaire à une comparaison de ligne <> mais elle ne renvoie pas NULL pour des entrées NULL. À la place, toute valeur NULL est considérée différente (distincte de) toute valeur non NULL, et deux valeurs NULL sont considérées égales (non distinctes). De ce fait, le résultat sera soit true, soit false, mais jamais NULL.

row_constructor `IS NOT DISTINCT FROM` *row_constructor*

Cette construction est similaire à une comparaison de ligne = mais elle ne renvoie pas NULL pour des entrées NULL. À la place, toute valeur NULL est considérée différente (distincte de) toute valeur non NULL, et deux valeurs NULL sont considérés égales (non distinctes). De ce fait, le résultat sera toujours soit true soit false, mais jamais NULL.

9.24.6. Comparaison de type composite

record operator record

La spécification SQL requiert que la comparaison de lignes renvoie NULL si le résultat depend de la comparaison de deux valeurs NULL, ou d'un NULL et d'un non NULL. PostgreSQL le fait seulement lors de la comparaison des résultats de deux constructeurs de ligne (comme dans Section 9.24.5) ou lors de la comparaison d'un constructeur de ligne et de la sortie d'une sous-requête (comme dans Section 9.23). Dans les autres contextes où les valeurs de deux types composites sont comparées, deux valeurs NULL sont considérées égales et un NULL est considéré plus grand qu'un non NULL. Ceci est nécessaire pour avoir un tri et une indexation cohérentes pour les types composites.

Chaque côté est évalué, puis comparé ligne par ligne. Les comparaisons de type composite sont autorisées quand l'opérateur *operator* vaut =, <>, <, <=, > ou >=, ou a une sémantique similaire à ces derniers. (Pour être précis, un opérateur peut être un opérateur de comparaison de ligne s'il est

membre d'une classe d'opérateurs B-tree, ou s'il est la négation du membre = d'une classe d'opérateurs B-tree.) Le comportement par défaut des opérateurs ci-dessus est identique à celui de `IS [ NOT ] DISTINCT FROM` pour les constructeurs de lignes (voir Section 9.24.5).

Pour supporter la comparaison des lignes qui incluent des éléments sans une classe d'opérateurs B-tree par défaut, les opérateurs suivants sont définis pour la comparaison de type composite : `*=`, `*<>`, `*<`, `*<=`, `*>` et `*>=`. Ces opérateurs comparent la représentation binaire interne de deux lignes. Les deux lignes peuvent avoir une représentation binaire différente même si la comparaison des deux lignes avec l'opérateur d'égalité est vraie. L'ordre des lignes sous ces opérateurs de comparaison est déterministique, mais sans sens. Ces opérateurs sont utilisés en interne pour les vues matérialisées et peuvent se révéler utile pour d'autres buts spécialisés tels que la réplication et la déduplication de B-Tree (voir Section 67.4.3). Ils n'ont pas pour but d'être utilisés dans l'écriture de requêtes.

9.25. Fonctions renvoyant des ensembles

Cette section décrit les fonctions qui renvoient potentiellement plus d'une ligne. Les fonctions les plus fréquemment utilisées de ce type sont celles générant des séries, comme détaillées dans Tableau 9.64 et Tableau 9.65. D'autres, plus spécialisées, sont décrites ailleurs dans ce manuel. Voir Section 7.2.1.4 pour des façons de combiner plusieurs fonctions renvoyant des ensembles.

Tableau 9.64. Fonctions générant des séries

Fonction **Description**
`generate_series ( start integer, stop integer [, step integer ] ) → setof integer` `generate_series ( start bigint, stop bigint [, step bigint ] ) → setof bigint` `generate_series ( start numeric, stop numeric [, step numeric ] ) → setof numeric` Génère une série de valeurs à partir de *start* jusqu'à *stop*, avec un pas de *step*. *step* a 1 pour valeur par défaut.
`generate_series ( start timestamp, stop timestamp, step interval ) → setof timestamp` `generate_series ( start timestamp with time zone, stop timestamp with time zone, step interval ) → setof timestamp with time zone` Génère une série de valeurs à partir de *start* jusqu'à *stop*, avec un pas de *step*.

Quand *step* est positif, aucune ligne n'est renvoyée si *start* est supérieure à *stop*. Par contre, quand *step* est négatif negative, aucune ligne n'est renvoyée si *start* est inférieur à *stop*. Aucune ligne n'est renvoyé si au moins une entrée est `NULL`. Si *step* vaut zéro, c'est considéré comme une erreur. Voici quelques exemples :

```
SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
```

```
              3
              1
(3 rows)

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

SELECT generate_series(1.1, 4, 1.3);
 generate_series
-----------------
             1.1
             2.4
             3.7
(3 rows)

-- cet exemple se base sur l'opérateur date-plus-integer :
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS
 s(a);
   dates
------------
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
   generate_series
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)
```

Tableau 9.65. Fonctions générant des indices

Fonction Description
`generate_subscripts` (*`array`* `anyarray`, *`dim`* `integer`) → `setof integer` Génère une série comprenant les indices valides de la *`dim`*-ème dimension du tableau fourni.
`generate_subscripts` (*`array`* `anyarray`, *`dim`* `integer`, *`reverse`* `boolean`) → `setof integer` Génère une série comprenant les indices valides de la *`dim`*-ième dimension du tableau fourni. Quand *`reverse`* vaut true, renvoie la série dans l'ordre inverse.

`generate_subscripts` est une fonction de facilité qui génère l'ensemble d'indices valides pour la dimension donnée pour le tableau. Aucune ligne n'est renvoyée pour les tableaux qui n'ont pas la dimension demandée ou si une des lignes vaut `NULL`. Voici quelques exemples :

```
-- utilisation basique :
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
 3
 4
(4 rows)

-- à partir d'un tableau, l'indice et la valeur indicée
-- nécessite une sous-requête :
SELECT * FROM arrays;
         a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
     array     | subscript | value
---------------+-----------+-------
 {-1,-2}       |         1 |    -1
 {-1,-2}       |         2 |    -2
 {100,200,300} |         1 |   100
 {100,200,300} |         2 |   200
 {100,200,300} |         3 |   300
(5 rows)

-- déballer un tableau 2D :
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
   from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
       3
       4
(4 rows)
```

Quand une fonction dans la clause `FROM` a pour suffixe `WITH ORDINALITY`, une colonne `bigint` est ajoutée aux colonnes en sortie de la fonction, commençant à 1 et s'incrémentant de 1 pour chaque ligne de la sortie de la fonction. Ceci est plus utile dans le cas de fonctions renvoyant des ensembles comme `unnest()`.

```
-- fonction renvoyant un ensemble WITH ORDINALITY:
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls,n);
       ls        | n
-----------------+----
```

```
 pg_serial       |  1
 pg_twophase     |  2
 postmaster.opts |  3
 pg_notify       |  4
 postgresql.conf |  5
 pg_tblspc       |  6
 logfile         |  7
 base            |  8
 postmaster.pid  |  9
 pg_ident.conf   | 10
 global          | 11
 pg_xact         | 12
 pg_snapshots    | 13
 pg_multixact    | 14
 PG_VERSION      | 15
 pg_wal          | 16
 pg_hba.conf     | 17
 pg_stat_tmp     | 18
 pg_subtrans     | 19
(19 rows)
```

9.26. Fonctions et opérateurs d'information sur le système

Tableau 9.66 montre différentes fonctions pour extraire des informations sur les sessions et le système.

En plus des fonctions listées dans cette section, il existe plusieurs fonctions relatives au système de statistiques qui fournissent aussi des informations sur le système. Voir Section 28.2.24 pour plus d'informations.

Tableau 9.66. Fonctions d'information sur la session

Fonction **Description**
`current_catalog` → `name` `current_database` () → `name` Renvoie le nom de la base actuelle. (Les bases sont appelées des « catalogues » dans le standard SQL, donc `current_catalog` est le terme du standard.)
`current_query` () → `text` Renvoie le texte de la requête en cours d'exécution telle qu'elle a été envoyée par le client (elle pourrait donc contenir plus d'une requête).
`current_role` → `name` Équivalent à `current_user`.
`current_schema` → `name` `current_schema` () → `name` Renvoie le nom du premier schéma dans le chemin de recherche (ou une valeur NULL si le chemin de recherche est vide). C'est le schéma qui sera utilisé par toute création de tables ou d'autres objets nommés si un schéma cible n'est pas précisé.
`current_schemas` (*`include_implicit`* `boolean`) → `name[]` Renvoie un tableau de noms pour tous les schémas présents dans le chemin de recherche, dans leur ordre de priorité. (Les éléments dans le paramètre search_path ne correspondant pas à des objets existants et cherchables sont omis.) Si l'argument booléen

Fonction **Description**
vaut `true`, alors les schémas systèmes implicites, tels que `pg_catalog` sont inclus dans le résultat.
`current_user` → `name` Renvoie le nom de l'utilisateur dans le contexte d'exécution actuel.
`inet_client_addr` () → `inet` Renvoie l'adresse IP du client actuel ou `NULL` si la connexion actuelle a été réalisée via une socket de domaine Unix.
`inet_client_port` () → `integer` Renvoie le numéro de port IP du client actuel, ou `NULL` si la connexion actuelle a été réalisée via une socket de domaine Unix.
`inet_server_addr` () → `inet` Renvoie l'adresse IP par laquelle le serveur a accepté la connexion actuelle, ou `NULL` si la connexion actuelle a été réalisée via une socket de domaine Unix.
`inet_server_port` () → `integer` Renvoie le numéro de port IP par lequel le serveur a accepté la connexion actuelle, ou `NULL` si la connexion actuelle a été réalisée via une socket de domaine Unix.
`pg_backend_pid` () → `integer` Renvoie l'identifiant de processus du processus serveur attaché à la session actuelle.
`pg_blocking_pids` (`integer`) → `integer[]` Renvoie un tableau d'identifiants de processus des sessions bloquant l'accès à un verrou pour le processus serveur dont l'identifiant a été donné. Renvoie un tableau vide si aucun processus ne le bloque. Un processus serveur bloque un autre processus s'il détient un verrou qui entre en conflit avec la demande de verrou du processus bloqué (blocage dur) ou s'il est en attente d'un verrou qui rentrerait en conflit avec la demande de verrou du processus bloqué et est devant lui dans la queue d'attente (verrou léger). Lors de l'utilisation de requêtes parallélisées, le résultat liste toujours les identifiants des processus visibles du client (autrement dit, le résultat de `pg_backend_pid`) même si le verrou réel est détenu ou attendu par un processus worker. Ceci résulte potentiellement en des PID dupliqués dans le résultat. De plus, notez que quand une transaction préparée détient un verrou en conflit, il sera représenté par un identifiant zéro. Les appels fréquents à cette fonction ont des impacts sur les performances de la base car il est nécessaire d'avoir un accès exclusif à l'état partagé du gestionnaire de verrous pour un court moment.
`pg_conf_load_time` () → `timestamp with time zone` Renvoie l'horodatage du dernier chargement des fichiers de configuration. Si la session actuelle était vivante à ce moment, cela sera l'heure quand la session a elle-même relu les fichiers de configuration (donc le résultat peut légèrement varier sur les différentes sessions). Sinon c'est le moment où le processus postmater relit les fichiers de configuration.
`pg_current_logfile` ([`text`]) → `text` Renvoie le chemin du fichier de traces actuellement utilisé par le collecteur de traces. Le chemin inclut le répertoire log_directory et le nom du fichier de traces individuel. Le résultat est `NULL` si le collecteur de traces est désactivé. Quand plusieurs fichiers de trace existent, chacun dans un format différent, `pg_current_logfile` sans argument renvoie le fichier vers le fichier ayant le premier format trouvé dans la liste ordonnée : `stderr`, `csvlog`, `jsonlog`. `NULL` est renvoyé si aucun fichier de trace n'a un de ces formats. Pour réclamer des informations sur un format spécifique de fichier de trace, indiquez soit `csvlog`, soit `jsonlog`, soit `stderr` comme valeur au paramètre

Fonction Description
optionnel. Le résultat est NULL si le format de trace demandé n'est pas configuré dans log_destination. Le résultat réflète le contenu du fichier current_logfiles.
pg_my_temp_schema () → oid Renvoie l'OID du schéma temporaire de la session en cours, ou zéro si ce schéma n'existe pas (parce que la session n'a pas créé de tables temporaires).
pg_is_other_temp_schema (oid) → boolean Renvoie true si l'OID donné est l'OID du schéma temporaire d'une autre session. (Ceci peut être utile, par exemple, pour exclure les tables temporaires des autres sessions d'un affichage du catalogue.)
pg_jit_available () → boolean Renvoie true si l'extension de compilation JIT est disponible (voir Chapitre 32) et si le paramètre de configuration jit est configuré à on.
pg_listening_channels () → setof text Renvoie l'ensemble de noms des canaux de notification asynchrone sur lesquels la session actuelle écoute.
pg_notification_queue_usage () → double precision Renvoie la fraction (0–1) de la taille maximale de la queue de notification asynchrone actuellement occupée par des notifications en attente de traitement. Voir LISTEN et NOTIFY pour plus d'informations.
pg_postmaster_start_time () → timestamp with time zone Renvoie l'heure de démarrage du serveur.
pg_safe_snapshot_blocking_pids (integer) → integer[] Renvoie un tableau d'identifiants de processus des sessions bloquant le processus serveur avec l'identifiant de processus spécifié pour acquérir une image propre. Renvoie un tableau vide si ce processus n'existe pas ou s'il n'est pas bloqué. Une session exécutant une transaction SERIALIZABLE bloque une transaction SERIALIZABLE READ ONLY DEFERRABLE sur l'acquisition d'une image jusqu'à ce que ce dernier détermine s'il est sûr d'éviter de prendre des verrous de prédicat. Voir Section 13.2.3 pour plus d'informations sur les transactions sérialisables et déferrables. Les appels fréquents à cette fonction pourraient avoir des impacts sur la performance des bases car elle a besoin d'accéder à l'état partagé du gestionnaire de verrous de prédicats pour un court moment.
pg_trigger_depth () → integer Renvoie le niveau de profondeur actuel des triggers PostgreSQL (0 si non appelé, directement ou indirectement, à partir d'un trigger).
session_user → name Renvoie le nom d'utilisateur de la session.
user → name Équivalent à current_user.
version () → text Renvoie une chaîne décrivant la version du serveur PostgreSQL. Vous pouvez aussi obtenir cette information à partir de server_version ou, pour une version lisible par un logiciel, vous pouvez utiliser server_version_num. Les développeurs de logiciels devraient utiliser server_version_num (disponible depuis la version 8.2) ou PQserverVersion au lieu d'analyser la version texte.

> **Note**
>
> `current_catalog`, `current_role`, `current_schema`, `current_user`, `session_user` et `user` ont un statut syntaxique spécial en SQL : elles doivent être appelées sans les parenthèses. Dans PostgreSQL, les parenthèses peuvent être utilisées en option avec `current_schema`, mais pas avec les autres.

`session_user` est habituellement l'utilisateur qui a initié la connexion à la base de données ; les superutilisateurs peuvent modifier ce paramètre avec SET SESSION AUTHORIZATION. `current_user` est l'identifiant de l'utilisateur applicable pour les vérifications de droits. Habituellement, il est égal à l'utilisateur de la session mais il peut être changé avec SET ROLE. Il peut aussi changer durant l'exécution d'une fonction ayant l'attribut `SECURITY DEFINER`. Dans la parlance Unix, l'utilisateur de session est le « real user » et l'utilisateur courant est le « effective user ». `current_role` et `user` sont des synonymes pour `current_user`. (Le standard SQL fait une distinction entre `current_role` et `current_user`, mais PostgreSQL ne la fait pas car il unifie les utilisateurs et les rôles en une seule entité.)

Tableau 9.67 liste les fonctions qui permettent d'accéder programmatiquement aux droits d'accès aux objets. (Voir Section 5.7 pour plus d'informations sur les droits.) Dans ces fonctions, l'utilisateur dont les droits sont interrogés peut être indiqué par son nom ou son OID (`pg_authid.oid`), ou si le nom donné est `public`, alors la vérification se fait sur les droits du pseudo-rôle PUBLIC. De plus, l'argument *`user`* peut être complètement omis, auquel cas la vérification se fera sur `current_user`. L'objet vérifié peut être indiqué par son nom ou son OID. Dans le cas du nom, le nom du schéma peut être inclus si nécessaire. Le droit d'accès d'intérêt est indiqué par une chaîne de caractères qui doit correspondre à un des mots clés approprié pour le type d'objet (par exemple `SELECT`). En option, `WITH GRANT OPTION` peut être ajouté au type de droit pour tester si le droit contient cette option. De plus, plusieurs types de droits peuvent être listés en les séparant par des virgules, auquel cas le résultat sera vrai si un des droits est détenu (la casse n'est pas significative et les espaces blancs supplémentaires sont autorisés entre les noms des droits, mais pas à l'intérieur des noms.) Quelques exemples :

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH
 GRANT OPTION');
```

Tableau 9.67. Fonctions d'accès aux droits

Fonction Description
`has_any_column_privilege` ([*`user`* `name` or `oid`,] *`table`* `text` or `oid`, *`privilege`* `text`) → `boolean` Est-ce que l'utilisateur a le droit pour une colonne de la table ? Ceci réussit si le droit est détenu pour la table entière ou s'il est détenu par au moins une colonne de la table. Les droits autorisés sont `SELECT`, `INSERT`, `UPDATE` et `REFERENCES`.
`has_column_privilege` ([*`user`* `name` or `oid`,] *`table`* `text` or `oid`, *`column`* `text` or `smallint`, *`privilege`* `text`) → `boolean` Est-ce que l'utilisateur a le droit pour la colonne spécifié de la table ? Ceci réussit si le droit est détenu pour la table entière ou s'il est détenu par cette colonne de la table. La colonne peut être spécifiée par son nom ou son numéro d'attribut (`pg_attribute.attnum`). Les droits autorisés sont `SELECT`, `INSERT`, `UPDATE` et `REFERENCES`.
`has_database_privilege` ([*`user`* `name` or `oid`,] *`database`* `text` or `oid`, *`privilege`* `text`) → `boolean`

Fonction **Description**
Est-ce que l'utilisateur a le droit pour la base ? Les droits autorisés sont CREATE, CONNECT, TEMPORARY et TEMP (qui est équivalent à TEMPORARY).
has_foreign_data_wrapper_privilege ([*user* name or oid,] *fdw* text or oid, *privilege* text) → boolean Est-ce que l'utilisateur a le droit pour le foreign data wrapper ? Le seul droit autorisé est USAGE.
has_function_privilege ([*user* name or oid,] *function* text or oid, *privilege* text) → boolean Est-ce que l'utilisateur a le droit pour la fonction ? Le seul droit autorisé est EXECUTE. Lors de la spécification d'une fonction par nom, plutôt que par OID, l'entrée autorisée est la même que pour le type de données regprocedure (voir Section 8.19). En voici un exemple : `SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');`
has_language_privilege ([*user* name or oid,] *language* text or oid, *privilege* text) → boolean Est-ce que l'utilisateur a le droit pour le langage ? Le seul droit autorisé est USAGE.
has_parameter_privilege ([*user* name or oid,] *parameter* text, *privilege* text) → boolean Est-ce que l'utilisateur a le droit pour configurer ce paramètre ? Le nom du paramètre est insensible à la casse. Les droits autorisés sont SET et ALTER SYSTEM.
has_schema_privilege ([*user* name or oid,] *schema* text or oid, *privilege* text) → boolean Est-ce que l'utilisateur a le droit pour le schéma ? Les droits autorisés sont CREATE et USAGE.
has_sequence_privilege ([*user* name or oid,] *sequence* text or oid, *privilege* text) → boolean Est-ce que l'utilisateur a le droit pour la séquence ? Les droits autorisés sont USAGE, SELECT et UPDATE.
has_server_privilege ([*user* name or oid,] *server* text or oid, *privilege* text) → boolean Est-ce que l'utilisateur a le droit pour le serveur distant ? Le seul droit autorisé est USAGE.
has_table_privilege ([*user* name or oid,] *table* text or oid, *privilege* text) → boolean Est-ce que l'utilisateur a le droit pour la table ? Les droits autorisés sont SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES et TRIGGER.
has_tablespace_privilege ([*user* name or oid,] *tablespace* text or oid, *privilege* text) → boolean Est-ce que l'utilisateur a le droit pour le tablespace ? Le seul droit autorisé est CREATE.
has_type_privilege ([*user* name or oid,] *type* text or oid, *privilege* text) → boolean Est-ce que l'utilisateur a le droit pour le type de données ? Le seul droit autorisé est USAGE. Lors de la spécification d'un type par son nom plutôt que par son OID, l'entrée autorisée est la même que pour le type de données regtype (voir Section 8.19).

Fonction Description
pg_has_role ([*user* name or oid,] *role* text or oid, *privilege* text) → boolean Est-ce que l'utilisateur a le droit pour le rôle ? Les droits autorisés sont MEMBER et USAGE. MEMBER dénote une appartenance de groupe directe ou indirecte dans le rôle (autrement dit, le droit d'exécuter SET ROLE), alors que USAGE dénote si les droits de ce rôle sont immédiatement disponibles sans exécuter SET ROLE. WITH ADMIN OPTION ou WITH GRANT OPTION peuvent être ajoutés à un de ces types de droits si le droit ADMIN est détenu (les quatre syntaxes testent la même chose). Cette fonction n'autorise pas le cas spécial de configurer *user* à public, car le pseudo-rôle PUBLIC ne peut jamais être un membre d'un vrai rôle.
row_security_active (*table* text or oid) → boolean Est-ce que la sécurité niveau ligne est active pour la table dans le contexte de l'utilisateur et de l'environnement actuels ?

Tableau 9.68 montre les opérateurs disponibles pour le type aclitem, qui est la représentation catalogue pour les droits d'accès. Voir Section 5.7 pour des informations sur la lecture des droits d'accès.

Tableau 9.68. Opérateurs aclitem

Opérateur Description Exemple(s)
aclitem = aclitem → boolean Est-ce que les aclitem sont identiques ? (Notez que le type aclitem manque de l'ensemble habituel d'opérateurs de comparaison ; il n'a que l'égalité. Cependant, les tableaux aclitem peuvent seulement être comparés par égalité.) `'calvin=r*w/hobbes'::aclitem = 'calvin=r*w*/hobbes'::aclitem` → f
aclitem[] @> aclitem → boolean Est-ce que le tableau contient les droits indiqués ? (Ceci vaut true s'il existe une entrée de tableau qui correspond à l'aclitem et a au moins l'ensemble indiqué de droits.) `'{calvin=r*w/hobbes,hobbes=r*w*/postgres}'::aclitem[] @> 'calvin=r*/hobbes'::aclitem` → t

Tableau 9.68 montre quelques fonctions supplémentaires pour la gestion du type aclitem.

Tableau 9.69. Fonctions aclitem

Fonction Description
acldefault (*type* "char", *ownerId* oid) → aclitem[] Construit un tableau d'aclitem détenant les droits d'accès par défaut d'un objet de type *type* appartenant au rôle d'OID *ownerId*. Ceci représente les droits d'accès supposés quand une entrée ACL d'un objet est NULL. (Les droits d'accès par défaut sont décrits dans Section 5.7.) Le paramètre *type* doit faire partie de 'c' pour COLUMN, 'r' pour TABLE ainsi que les objets de type table, 's' pour SEQUENCE, 'd' pour DATABASE, 'f' pour FUNCTION ou PROCEDURE, 'l' pour LANGUAGE, 'L' pour LARGE OBJECT, 'n' pour SCHEMA, 'p' pour PARAMETER, 't' pour TABLESPACE, 'F' pour FOREIGN DATA WRAPPER, 'S' pour FOREIGN SERVER, ou 'T' pour TYPE ou DOMAIN.
aclexplode (aclitem[]) → setof record (*grantor* oid, *grantee* oid, *privilege_type* text, *is_grantable* boolean)

Fonction Description
Renvoie le tableau the `aclitem` sous la forme d'un ensemble de lignes. Si le bénéficiaire est le pseudo-rôle PUBLIC, il est représenté par zéro dans la colonne *`grantee`*. Chaque droit donné est représenté sous la forme `SELECT`, `INSERT`, etc. Notez que chaque droit est séparé sur sa propre ligne, donc un seul mot clé apparaît dans la colonne *`privilege_type`*.
`makeaclitem` (*`grantee`* `oid`, *`grantor`* `oid`, *`privileges`* `text`, *`is_grantable`* `boolean`) → `aclitem` Construit un `aclitem` avec les propriétés données.

Tableau 9.70 montre les fonctions qui déterminent si un certain objet est *visible* dans le chelub de recherche de schémas courant. Par exemple, une table est dite visible si le schéma qui la contient est dans le chemin de recherche. C'est équivalent au fait que la table doit être référencée par nom sans qualification explicite par d'un schéma. De ce fait, pour lister les noms des tables visibles :

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

Pour les fonctions et opérateurs, un objet dans le chemin de recherche est dit visible si aucun objet du même nom *et du même type* se trouve avant dans le chemin. Pour les classes et familles d'opérateur, le nom et la méthode d'accès associé de l'index sont considérés.

Tableau 9.70. Fonctions de vérification de la visibilité

Fonction Description
`pg_collation_is_visible` (*`collation`* `oid`) → `boolean` La collation est-elle visible dans le chemin de recherche ?
`pg_get_partition_constraintdef` (*`table`* `oid`) → `text` Reconstruit la définition d'une contrainte de partition. (C'est une reconstruction décompilée, pas le texte original de la commande.)
`pg_conversion_is_visible` (*`conversion`* `oid`) → `boolean` La conversion est-elle visible dans le chemin de recherche ?
`pg_function_is_visible` (*`function`* `oid`) → `boolean` La fonction est-elle visible dans le chemin de recherche ? (Ceci fonctionne aussi pour les procédures et les agrégats.)
`pg_opclass_is_visible` (*`opclass`* `oid`) → `boolean` La classe d'opérateurs est-elle visible dans le chemin de recherche ?
`pg_operator_is_visible` (*`operator`* `oid`) → `boolean` L'opérateur est-il visible dans le chemin de recherche ?
`pg_opfamily_is_visible` (*`opclass`* `oid`) → `boolean` La famille d'opérateur est-elle visible dans le chemin de recherche ?
`pg_statistics_obj_is_visible` (*`stat`* `oid`) → `boolean` L'objet statistique est-il visible dans le chemin de recherche ?
`pg_table_is_visible` (*`table`* `oid`) → `boolean` La table est-elle visible dans le chemin de recherche ? (Cela fonctionne aussi pour tous les types de relations, ceci incluant les vues, les vues matérialisées, les séquences et les tables externes.)
`pg_ts_config_is_visible` (*`config`* `oid`) → `boolean`

Fonction Description
La configuration de recherche plein texte est-elle visible dans le chemin de recherche ?
`pg_ts_dict_is_visible` (*`dict`* `oid`) → `boolean` Le dictionnaire de recherche plein texte est-il visible dans le chemin de recherche ?
`pg_ts_parser_is_visible` (*`parser`* `oid`) → `boolean` L'analyseur de recherche plein texte est-il visible dans le chemin de recherche ?
`pg_ts_template_is_visible` (*`template`* `oid`) → `boolean` Le modèle de recherche plein texte est-il visible dans le chemin de recherche ?
`pg_type_is_visible` (*`type`* `oid`) → `boolean` Le type (ou domaine) est-il visible dans le chemin de recherche ?

Toutes ces fonctions requièrent les OID des objets pour identifier l'objet à vérifier. Si vous voulez tester un objet par son nom, il est préférable d'utiliser les types d'alias de l'OID (`regclass`, `regtype`, `regprocedure`, `regoperator`, `regconfig` ou `regdictionary`), par exemple :

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

Notez qu'il n'y aurait pas de sens à tester un nom de type non qualifié par son schéma de cette façon -- si le nom peut être reconnu, c'est qu'il est visible.

Tableau 9.71 liste les fonctions qui extraient des informations des catalogues systèmes.

Tableau 9.71. Fonctions d'information sur les catalogues systèmes

Fonction Description
`format_type` (*`type`* `oid`, *`typemod`* `integer`) → `text` Renvoie le nom SQL pour un type de données identifié par l'OID type et potentiellement par un modifieur de type. Envoyez NULL pour le modifieur si aucun modifieur spécifique n'est connu.
`pg_char_to_encoding` (*`encoding`* `name`) → `integer` Convertit le nom d'encodage fourni en un entier représentant l'identifiant interne utilisé dans certains catalogues systèmes. Renvoie `-1` si le nom d'encodage fourni est inconnu.
`pg_encoding_to_char` (*`encoding`* `integer`) → `name` Convertit l'entier utilisé comme identifiant interne d'encodage pour certains catalogues systèmes en une chaîne lisible par un humain. Renvoie une chaîne vide si le numéro d'encodage fourni est invalide.
`pg_get_catalog_foreign_keys` () → `setof record` (*`fktable`* `regclass`, *`fkcols`* `text[]`, *`pktable`* `regclass`, *`pkcols`* `text[]`, *`is_array`* `boolean`, *`is_opt`* `boolean`) Retourne un ensemble de lignes décrivant les clés étrangères qui existent dans le catalogue système de PostgreSQL. La colonne *`fktable`* contient le nom du catalogue référent, et la colonne *`fkcols`* contient le(s) nom(s) de(s) colonne(s) référente(s). De même, la colonne *`pktable`* contient le nom du catalogue référencé, et la colonne *`pkcols`* contient le(s) nom(s) de(s) colonne(s) référencée(s). Si *`is_array`* vaut true, la dernière colonne référente est un tableau, dont chaque élément doit correspondre à une entrée du catalogue référencé. Si *`is_opt`* vaut true, la(les) colonne(s) référente(s) est(sont) autorisée(s) à contenir des zéros au lieu d'une référence valide.
`pg_get_constraintdef` (*`constraint`* `oid` [, *`pretty`* `boolean`]) → `text`

Fonction Description
Reconstruit la commande de création pour une contrainte. (Ceci est une reconstruction décompilée, pas le texte original de la commande.)
`pg_get_expr` (*`expr`* `pg_node_tree`, *`relation`* `oid` [, *`pretty`* `boolean`]) → `text` Décompile la forme interne de l'expression stockée dans les catalogues systèmes, telle que la valeur par défaut pour une colonne. Si l'expression peut contenir Vars, indiquez l'OID de la relation à laquelle elles font référence dans le deuxième paramètre ; si aucun Var n'est attendu, indiquer 0 est suffisant.
`pg_get_functiondef` (*`func`* `oid`) → `text` Reconstruit la commande de création pour une fonction ou une procédure. (C'est une reconstruction décompilée, pas le texte original de la commande.) Le résultat est une instruction complète, `CREATE OR REPLACE FUNCTION` ou `CREATE OR REPLACE PROCEDURE`.
`pg_get_function_arguments` (*`func`* `oid`) → `text` Reconstruit la liste d'argument d'une fonction ou procédure, dans la forme où elle devrait apparaître dans un `CREATE FUNCTION` (en incluant les valeurs par défaut).
`pg_get_function_identity_arguments` (*`func`* `oid`) → `text` Reconstruit la liste d'arguments nécessaire pour identifier une fonction ou une procédure, sous la forme nécessaire pour intégrer une commande telle que `ALTER FUNCTION`. Cette forme omet les valeurs par défaut.
`pg_get_function_result` (*`func`* `oid`) → `text` Reconstruit la clause `RETURNS` d'une function, dans la forme nécessaire pour apparaître dans `CREATE FUNCTION`. Renvoie `NULL` pour une procédure.
`pg_get_indexdef` (*`index`* `oid` [, *`column`* `integer`, *`pretty`* `boolean`]) → `text` Reconstruit la commande de création pour un index (C'est une reconstruction décompilée, et non pas le texte original de la commande.) Si *`column`* est fourni et est différent de zéro, seule la définition de cette colonne est reconstruite.
`pg_get_keywords` () → `setof record` (*`word`* `text`, *`catcode`* `"char"`, *`barelabel`* `boolean`, *`catdesc`* `text`, *`baredesc`* `text`) Renvoie un ensemble d'enregistrements décrivant les mots clés SQL reconnus par le serveur. La colonne *`word`* contient le mot clé. La colonne *`catcode`* contient un code de catégorie : `U` pour un mot clé non réservé, `C` pour un mot clé qui peut être un nom de colonne, `T` pour un mot clé qui peut être un nom de type ou de fonction, `R` pour un mot totalement réservé. La colonne *`barelabel`* contient `true` si le mot clé peut être utilisé comme un nom de colonne strict dans les listes de `SELECT`, ou `false` si il ne peut être utilisé qu'après `AS`. La colonne *`catdesc`* contient une chaîne potentiellement localisée décrivant la catégorie du mot clé. La colonne *`baredesc`* contient une chaîne potentiellement localisée décrivant l'état de l'étiquette de la colonne du mot clé.
`pg_get_ruledef` (*`rule`* `oid` [, *`pretty`* `boolean`]) → `text` Reconstruit la commande de création pour une règle. (C'est une reconstruction décompilée, et non pas le texte original de la commande.)
`pg_get_serial_sequence` (*`table`* `text`, *`column`* `text`) → `text` Renvoie le nom de la séquence associée à une colonne ou NULL si aucune séquence n'est associée à la colonne. Si la colonne est une colonne identité, la séquence associée est la séquence créée en interne pour cette colonne. Pour les colonnes créées en utilisant un des types serial (`serial`, `smallserial`, `bigserial`), il s'agit de la séquence crée pour la définition de la colonne serial. Dans ce dernier cas, l'association peut être modifiée ou supprimée avec `ALTER SEQUENCE OWNED BY`. (Cette fonction aurait due s'appeler `pg_get_owned_sequence` ; son nom actuel reflète le fait qu'elle était historiquement utilisée avec des colonnes serial.) Le premier paramètre est un nom de

Fonction Description
tableau avec le schéma en option, et le deuxième paramètre est un nom de colonne. Comme le premier paramètre contient potentiellement un nom de schéma et un nom de table, il est analysé suivant les règles SQL habituelles, signifiant qu'il est mis en minuscule par défaut. Le second paramètre, étant juste un nom de colonne, est traité littéralement et conserve donc sa casse. Le résultat est formaté de façon convenable pour être donné à des fonctions de séquence (voir Section 9.17). Une utilisation typique est la lecture de la valeur actuelle d'une séquence pour une colonne identité ou serial. Par exemple : `SELECT currval(pg_get_serial_sequence('sometable', 'id'));`
`pg_get_statisticsobjdef` (*`statobj`* `oid`) → `text` Reconstruit le commande de création d'un objet statistique étendu. (Ceci est une reconstruction décompilée, par le texte de la commande originale.)
`pg_get_triggerdef` (*`trigger`* `oid` [, *`pretty`* `boolean`]) → `text` Reconstruit la commande de création d'un trigger. (Ceci est une reconstruction décompilée, par le texte de la commande originale.)
`pg_get_userbyid` (*`role`* `oid`) → `name` Renvoie le nom d'un rôle d'après son OID.
`pg_get_viewdef` (*`view`* `oid` [, *`pretty`* `boolean`]) → `text` Reconstruit la commande `SELECT` sous-jacente pour une vue ou une vue matérialisée. (Ceci est une reconstruction décompilée, par le texte de la commande originale.)
`pg_get_viewdef` (*`view`* `oid`, *`wrap_column`* `integer`) → `text` Reconstruit la commande `SELECT` sous-jacente pour une vue ou une vue matérialisée. (Ceci est une reconstruction décompilée, par le texte de la commande originale.) Dans cette forme de la fonction, l'affichage joli est toujours activé, et les longues lignes sont coupées pour essayer de les conserver plus petites que le nombre indiqué de colonnes.
`pg_get_viewdef` (*`view`* `text` [, *`pretty`* `boolean`]) → `text` Reconstruit la commande `SELECT` sous-jacente pour une vue ou une vue matérialisée, en travaillant à partir d'un nom pour la vue plutôt que de son OID. (Ceci est obsolète ; utilisez la variante OID à la place.)
`pg_index_column_has_property` (*`index`* `regclass`, *`column`* `integer`, *`property`* `text`) → `boolean` Teste si une colonne d'index a la propriété nommée. Les propriétés communes de colonne d'index sont listées dans Tableau 9.72. (Notez que les méthodes d'accès d'extension peuvent définir des noms de propriété supplémentaires pour leurs index.) `NULL` est renvoyé si le nom de la propriété n'est pas connu ou ne s'applique pas à l'objet particulier ou si l'OID ou le numéro de colonne n'identifie pas un objet valide.
`pg_index_has_property` (*`index`* `regclass`, *`property`* `text`) → `boolean` Teste si un index a la propriété nommée. Les propriétés communes d'index sont listées dans Tableau 9.73. (Notez que les méthodes d'accès d'extension peuvent définir des noms de propriété supplémentaires pour leurs index.) `NULL` est renvoyé si le nom de la propriété n'est pas connu ou ne s'applique pas à l'objet particulier ou si l'OID n'identifie pas un objet valide.
`pg_indexam_has_property` (*`am`* `oid`, *`property`* `text`) → `boolean` Teste si une méthode d'accès aux index a la propriété nommée. Les propriétés communes des méthodes d'accès aux index sont listées dans Tableau 9.74. `NULL` est renvoyé si le

Fonction **Description**
nom de la propriété n'est pas connu ou ne s'applique pas à l'objet particulier ou si l'OID n'identifie pas un objet valide.
`pg_options_to_table` (*`options_array`* `text[]`) → `setof record` (*`option_name`* `text`, *`option_value`* `text`) Renvoie l'ensemble d'options de stockage représenté par une valeur à partir de `pg_class.reloptions` ou `pg_attribute.attoptions`.
`pg_settings_get_flags` (*`guc`* `text`) → `text[]` Renvoie un tableau de drapeaux associés avec le paramètre donné ou `NULL` s'il n'existe pas. Le résultat est un tableau vide si le paramètre existe mais qu'aucun drapeau n'existe pour lui. Seuls les drapeaux les plus utiles listés dans Tableau 9.75 sont exposés.
`pg_tablespace_databases` (*`tablespace`* `oid`) → `setof oid` Renvoie l'ensemble d'OID des bases ayant des objets dans le tablespace indiqué. Si cette fonction renvoie des lignes, le tablespace n'est pas vide et ne peut pas être supprimé. Pour identifier les objets spécifiques peuplant le tablespace, vous aurez besoin de vous connecter aux bases identifiées par `pg_tablespace_databases` et de requêter leurs catalogues `pg_class`.
`pg_tablespace_location` (*`tablespace`* `oid`) → `text` Renvoie le chemin du système de fichiers où ce tablespace est situé.
`pg_typeof` (`"any"`) → `regtype` Renvoie l'OID du type de données de la valeur qui lui est fournie. Ceci peut être utile pour débogguer ou pour construire dynamiquement des requêtes SQL. Cette fonction est déclarée renvoyer le type `regtype`, qui est un alias d'OID pour les objets types de données (voir Section 8.19) ; ceci signifie qu'il est identique à un OID pour des comparaisons mais qu'il s'affiche sous la forme du nom du type. Par exemple : `SELECT pg_typeof(33);` ` pg_typeof` `-----------` ` integer` `SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);` ` typlen` `--------` `      4`
`COLLATION FOR` (`"any"`) → `text` Renvoie le nom de la collation de la valeur qui lui est fournie. La valeur est entre guillemet et qualifiée du schéma si nécessaire. Si aucune collation n'a été dérivée de l'expression de l'argument, alors `NULL` est renvoyée. Si l'argument n'est pas un type de données collationnable, alors une erreur est levée. Par exemple : `SELECT collation for (description) FROM pg_description` ` LIMIT 1;` ` pg_collation_for` `------------------` ` "default"`

Fonction Description
SELECT collation for ('foo' COLLATE "de_DE"); pg_collation_for ------------------ "de_DE"
`to_regclass ( text ) → regclass` Traduit un nom de relation en son OID. Un résultat similaire est obtenu en convertissant la chaîne dans le type `regclass` (voir Section 8.19) ; néanmoins, cette fonction renverra `NULL` plutôt que de lever une erreur si le nom n'est pas trouvé. De plus, contrairement à la conversion, ceci n'accepte pas un OID numérique en entrée.
`to_regcollation ( text ) → regcollation` Traduit un nom de collation en son OID. Un résultat similaire est obtenu en convertissant la chaîne dans le type `regcollation` (voir Section 8.19) ; néanmoins, cette fonction renverra `NULL` plutôt que de lever une erreur si le nom n'est pas trouvé. De plus, contrairement à la conversion, ceci n'accepte pas un OID numérique en entrée.
`to_regnamespace ( text ) → regnamespace` Traduit un nom de schéma en son OID. Un résultat similaire est obtenu en convertissant la chaîne dans le type `regnamespace` (voir Section 8.19) ; néanmoins, cette fonction renverra `NULL` plutôt que de lever une erreur si le nom n'est pas trouvé. De plus, contrairement à la conversion, ceci n'accepte pas un OID numérique en entrée.
`to_regoper ( text ) → regoper` Traduit un nom d'opérateur en son OID. Un résultat similaire est obtenu en convertissant la chaîne dans le type `regoper` (voir Section 8.19) ; néanmoins, cette fonction renverra `NULL` plutôt que de lever une erreur si le nom n'est pas trouvé. De plus, contrairement à la conversion, ceci n'accepte pas un OID numérique en entrée.
`to_regoperator ( text ) → regoperator` Traduit un nom d'opérateur en son OID. Un résultat similaire est obtenu en convertissant la chaîne dans le type `regoperator` (voir Section 8.19) ; néanmoins, cette fonction renverra `NULL` plutôt que de lever une erreur si le nom n'est pas trouvé. De plus, contrairement à la conversion, ceci n'accepte pas un OID numérique en entrée.
`to_regproc ( text ) → regproc` Traduit un nom de fonction ou procédure en son OID. Un résultat similaire est obtenu en convertissant la chaîne dans le type `regproc` (voir Section 8.19) ; néanmoins, cette fonction renverra `NULL` plutôt que de lever une erreur si le nom n'est pas trouvé. De plus, contrairement à la conversion, ceci n'accepte pas un OID numérique en entrée.
`to_regprocedure ( text ) → regprocedure` Traduit un nom de fonction ou procédure en son OID. Un résultat similaire est obtenu en convertissant la chaîne dans le type `regprocedure` (voir Section 8.19) ; néanmoins, cette fonction renverra `NULL` plutôt que de lever une erreur si le nom n'est pas trouvé. De plus, contrairement à la conversion, ceci n'accepte pas un OID numérique en entrée.
`to_regrole ( text ) → regrole` Traduit un nom de rôle en son OID. Un résultat similaire est obtenu en convertissant la chaîne dans le type `regrole` (voir Section 8.19) ; néanmoins, cette fonction renverra `NULL` plutôt que de lever une erreur si le nom n'est pas trouvé. De plus, contrairement à la conversion, ceci n'accepte pas un OID numérique en entrée.
`to_regtype ( text ) → regtype` Traduit un nom de type en son OID. Un résultat similaire est obtenu en convertissant la chaîne dans le type `regtype` (voir Section 8.19) ; néanmoins, cette fonction renverra

Fonction Description	
	`NULL` plutôt que de lever une erreur si le nom n'est pas trouvé. De plus, contrairement à la conversion, ceci n'accepte pas un OID numérique en entrée.

La plupart des fonctions qui reconstruisent (décompilent) des objets de base ont un drapeau *`pretty`*, qui à `true`, cause que le résultat soit « joliment affiché ». Un bel affichage supprime les parenthèses inutiles et ajoute des espaces blancs pour la lisibilité. Ce format est plus lisible mais le format par défaut a plus de chance d'être interprété de la même façon par les versions futures de PostgreSQL ; pour éviter d'utiliser une sortie joliment affichée pour les sauvegardes. Configurer `false` pour le paramètre *`pretty`* donne le même résultat que d'omettre le paramètre.

Tableau 9.72. Propriétés des colonnes d'un index

Nom	Description
`asc`	Est-ce que la colonne trie en ordre ascendant sur un parcours en avant ?
`desc`	Est-ce que la colonne trie en ordre descendant sur un parcours en avant ?
`nulls_first`	Est-ce que la colonne trie les valeurs NULL en premier sur un parcours en avant ?
`nulls_last`	Est-ce que la colonne trie les valeurs NULL en dernier sur un parcours en avant ?
`orderable`	Est-ce que la colonne possède un ordre de tri défini ?
`distance_orderable`	Est-ce que la colonne peut être parcourue en ordre par un opérateur « distance », par exemple `ORDER BY col <-> constant` ?
`returnable`	Est-ce que la valeur de la colonne peut être renvoyée par un parcours d'index seul ?
`search_array`	Est-ce que la colonne supporte nativement les recherches `col = ANY(array)` ?
`search_nulls`	Est-ce que la colonne supporte les recherches `IS NULL` et `IS NOT NULL` ?

Tableau 9.73. Propriétés des index

Nom	Description
`clusterable`	L'index peut-il être utilisé dans une commande `CLUSTER` ?
`index_scan`	L'index supporte-t-il les parcours simples (non bitmap) ?
`bitmap_scan`	L'index supporte-t-il les parcours bitmap ?
`backward_scan`	Est-ce que la direction du parcours peut être changée en milieu de parcours (pour accepter `FETCH BACKWARD` sur un curseur sans nécessiter de matérialisation) ?

Tableau 9.74. Propriétés des méthodes d'accès aux index

Nom	Description
`can_order`	Est-ce que la méthode d'accès supporte `ASC`, `DESC` et les autres mots-clés dans `CREATE INDEX` ?
`can_unique`	Est-ce que la méthode d'accès supporte les index d'unicité ?
`can_multi_col`	Est-ce que la méthode d'accès supporte les index multi-colonnes ?
`can_exclude`	Est-ce que la méthode d'accès supporte les contraintes d'exclusion ?
`can_include`	Est-ce que la méthode d'accès supporte la clause `INCLUDE` de `CREATE INDEX` ?

Tableau 9.75. Drapeaux des GUC

Drapeau	Description
`EXPLAIN`	Les paramètres disposant de ce drapeau sont inclus dans les commandes `EXPLAIN (SETTINGS)`.
`NO_SHOW_ALL`	Les paramètres disposant de ce drapeau sont exclus des commandes `SHOW ALL`.
`NO_RESET_ALL`	Les paramètres disposant de ce drapeau sont exclus des commandes `RESET ALL`.
`NOT_IN_SAMPLE`	Les paramètres disposant de ce drapeau ne sont pas inclus par défaut dans `postgresql.conf`.
`RUNTIME_COMPUTED`	Les paramètres disposant de ce drapeau sont calculés à l'exécution.

Tableau 9.76 liste les fonctions relatives à l'identification et l'adressage des objets de la base.

Tableau 9.76. Fonctions d'information et d'adressage des objets

Fonction Description
`pg_describe_object` (*`classid`* `oid`, *`objid`* `oid`, *`objsubid`* `integer`) → `text` Renvoie une description textuelle d'un objet de base identifié par un OID de catalogue, un OID d'objet, et un identifiant de sous-objet (tel que le numéro de colonne dans une table ; cet identifiant vaut zéro lorsqu'il référence un objet complet). Cette description a pour but d'être lisible par un humain, et pourrait être traduite, suivant la configuration du serveur. C'est particulièrement utile pour déterminer l'identité d'un objet référencé dans le catalogue `pg_depend`. Cette fontion retourne des valeurs `NULL` pour les objets non définis.
`pg_identify_object` (*`classid`* `oid`, *`objid`* `oid`, *`objsubid`* `integer`) → `record` (*`type`* `text`, *`schema`* `text`, *`name`* `text`, *`identity`* `text`) Renvoie une ligne contenant suffisamment d'informations pour identifier de façon unique l'objet de la base spécifié par un OID de catalogue, un OID d'objet et un identifiant de sous-objet. Cette information a pour but d'être lisible par un logiciel et n'est jamais traduite. *`type`* identifie le type d'objet de base ; *`schema`* est le nom du schéma auquel l'objet appartient ou `NULL` pour les types d'objet n'appartenant pas à des schémas ; *`name`* est le nom de l'objet, entre guillemets si nécessaire, si le nom (avec le nom du schéma

Fonction Description
le cas échéant) est suffisant pour identifier de façon unique l'objet, et NULL sinon ; *identity* est l'identité complète de l'objet, avec le format précis dépendant du type d'objet, et chaque nom dans le format habituel (qualification avec le schéma et guillemets doubles si nécessaire). Les objets non définis sont identifiés par des valeurs NULL.
pg_identify_object_as_address (*classid* oid, *objid* oid, *objsubid* integer) → record (*type* text, *object_names* text[], *object_args* text[]) Renvoie une ligne contenant suffisamment d'informations pour identifier de façon unique l'objet de la base spécifié par un OID de catalogue, un OID d'objet et un identifiant de sous-objet. L'information renvoyé est indépendante du serveur, autrement dit il pourrait être utilisé pour idenfier un objet pareillement nommé sur un autre serveur. *type* identifie le type d'objet de base ; *object_names* et *object_args* sont des tableaux de texte qui forment ensemble une référence à l'objet. Ces trois valeurs peuvent être passées à pg_get_object_address pour obtenir l'adresse interne de l'objet.
pg_get_object_address (*type* text, *object_names* text[], *object_args* text[]) → record (*classid* oid, *objid* oid, *objsubid* integer) Renvoie une ligne contenant suffisamment d'informations pour identifier de façon unique l'objet de base spécifié par le code type et le nom d'objet et les tableaux d'arguments. Les valeurs renvoyées sont celles qui seraient utilisées dans les catalogues systèmes tels que pg_depend ; elles peuvent être fournies à d'autres fonctions systèmes telles que pg_describe_object ou pg_identify_object. *classid* est l'OID du catalogue système contenant l'objet ; *objid* est l'OID de l'objet lui-même et *objsubid* est l'identifiant du sous-objet ou 0 si aucun. Cette fonction est l'inverse de pg_identify_object_as_address. Les objets non définis sont identifiés par des valeurs NULL.

Les fonctions listées dans Tableau 9.77 extraient les commentaires préalablement enregistrés avec la commande COMMENT. Une valeur NULL est renvoyée si aucun commentaire n'a été trouvé pour les paramètres donnés.

Tableau 9.77. Fonctions d'informations sur les commentaires

Fonction Description
col_description (*table* oid, *column* integer) → text Renvoie le commentaire pour une colonne de table, qui est spécifié par l'OID de la table et le numéro de colonne. (obj_description ne peut être utilisé pour les colonnes d'une table car les colonnes n'ont pas d'OID propre.)
obj_description (*object* oid, *catalog* name) → text Renvoie le commentaire pour un objet de la base qui est spécifié par son OID et le nom du catalogue système le contenant. Par exemple, obj_description(123456, 'pg_class') récupérera le commentaire de la table d'OID 123456.
obj_description (*object* oid) → text Renvoie le commentaire pour un objet de la base spécifié par son seul OID. Cette fonction est *obsolète* cat il n'existe aucune garantie que les OID soient uniques entre différents catalogues systèmes ; de ce fait, un mauvais commentaire pourrait être renvoyé.
shobj_description (*object* oid, *catalog* name) → text Renvoie le commentaire pour un objet de base partagé, spécifié par son OID et le nom du catalogue système le contenant. C'est comme obj_description, sauf que c'est utilisé

Fonction Description
pour récupérer les commentaires sur les objets partagés (donc les bases, les rôles et les tablespaces). Certains catalogues systèmes sont globaux à toutes les bases d'une instance, et les descriptions de ces objects sont stockées là-aussi globalement.

Les fonctions listées dans Tableau 9.78 fournissent des informations sur la transaction serveur dans un format exportable. L'utilisation principale de ces fonctions est de déterminer les transactions validées entre deux images de base.

Tableau 9.78. Fonctions d'identifiants de transaction et d'informations sur les images de base

Fonction Description
`pg_current_xact_id` () → `xid8` Renvoie l'identifiant actuel de la transaction. Il sera affecté si la transaction n'en a pas déjà un (ceci arrive si elle n'a pas encore réalisé d'écritures sur la base).
`pg_current_xact_id_if_assigned` () → `xid8` Renvoie l'identifiant actuel de la transaction ou `NULL` si aucun identifiant n'a encore été assigné. (Il est préférable d'utiliser cette variante si la transaction est habituellement en lecture seule pour éviter de consommer inutilement un identifiant de transaction.)
`pg_xact_status` (`xid8`) → `text` Renvoie le statut de validation d'une transaction récente. Le résultat fait partie de `in progress`, `committed` ou `aborted`, en supposant que la transaction est suffisamment récente pour que le système dispose du statut de validation de cette transaction. S'il est assez ancien pour qu'aucune référence de la transaction n'est survécu au système et que le statut de validation ait disparu, le résultat est `NULL`. Les applications peuvent utiliser cette fonction pour déterminer, par exemple, si leur transaction a été validée ou annulée si l'application a été déconnectée alors qu'un `COMMIT` était en cours. Notez que les transactions préparées sont rapportés `en cours` ; les applications doivent vérifier `pg_prepared_xacts` si elles ont besoin de savoir si un identifiant de transaction appartient à une transaction préparée.
`pg_current_snapshot` () → `pg_snapshot` Renvoie un *snapshot* (image) actuel, une structure de données indiquant les identifiants de transaction actuellement en cours.
`pg_snapshot_xip` (`pg_snapshot`) → `setof xid8` Renvoie l'ensemble des identifiants de transactions en cours contenu dans un snapshot (image de base).
`pg_snapshot_xmax` (`pg_snapshot`) → `xid8` Renvoie le `xmax` d'une image de base.
`pg_snapshot_xmin` (`pg_snapshot`) → `xid8` Renvoie le `xmin` d'une image de base
`pg_visible_in_snapshot` (`xid8`, `pg_snapshot`) → `boolean` Est-ce que l'identifiant de transaction donné est *visible* d'après cette image de base (autrement dit, est-il terminé avant que l'image ne soit réalisée) ? Notez que cette fonction ne renverra pas la bonne réponse pour un identifiant de sous-transaction.
`pg_get_multixact_members` (*`multixid`* `xid`) → `setof record` (*`xid`* `xid`, *`mode`* `text`) Renvoie l'identifiant de transaction et le mode de verrou pour chaque membre de l'identifiant MultiXact indiqué. Les modes de verrou `forupd`, `fornokeyupd`,

Fonction Description
sh et keysh correspondent, respectivement, aux verrous niveau ligne de FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE et FOR KEY SHARE, comme décrit dans Section 13.3.2. Deux modes supplémentaires sont spécifiques aux MultiXact : nokeyupd, utilisé par les mises à jour qui ne modifient pas de colonnes clés, et upd, utilisé par les mises à jour des colonnes clés.

Le type interne des identifiants de transaction, `xid`, est sur 32 bits et boucle tous les 4 milliards de transactions. Néanmoins, les fonctions listées dans Tableau 9.78, sauf `age`, `mxid_age` et `pg_get_multixact_members`, utilisent un type `xid8` sur 64 bits qui ne boucle pas sur toute la durée de vie d'une instance, et qui peut être converti en `xid` par conversion si nécessaire. Le type de données `pg_snapshot` enregistre des informations sur la visibilité des identifiants de transaction à un moment particulier. Ses composants sont décrits dans Tableau 9.79. La représentation textuelle de `pg_snapshot` est *`xmin`*`:`*`xmax`*`:`*`xip_list`*. Par exemple, `10:20:10,14,15` signifie `xmin=10, xmax=20, xip_list=10, 14, 15`.

Tableau 9.79. Composants d'image de base

Nom	Description
`xmin`	Plus petit identifiant de transaction toujours actif. Tous les identifiants de transaction inférieurs à `xmin` sont soit validés et visibles, soit annulés et morts.
`xmax`	L'identifiant de transaction après le dernier identifiant terminé. Tous les identifiants de transaction supérieurs ou égals à `xmax` ne sont pas encore terminés au moment de la création de l'image, et sont de ce fait invisibles.
`xip_list`	Transactions en cours au moment de la création de l'image de base. Un identifiant de transaction qui est `xmin <= `*`X`*` < xmax` et qui n'est pas dans cette liste a déjà été terminé au moins de la création de l'image et, de ce fait, est soit visible soit mort suivant son statut de validation. Cette liste n'inclut pas les identifiants de transaction des sous-transactions.

Dans les versions de PostgreSQL antérieures à la 13, le type `xid8` n'existait pas, donc des variantes de ces fonctions étaient fournies. Elles utilisaient le type `bigint` pour représenter un identifiant sur 64 bits, avec un type de données pour l'image nommé `txid_snapshot`. Ces anciennes fonctions avaient le mot `txid` dans leur nom. Elles sont toujours acceptées pour des raisons de compatibilité ascendante, mais pourraient être supprimées dans une version future. Voir Tableau 9.80.

Tableau 9.80. Fonctions obsolètes sur les identifiants de transactions et les informations sur les images de base

Fonction Description
`txid_current` () → `bigint` Voir `pg_current_xact_id()`.
`txid_current_if_assigned` () → `bigint` Voir `pg_current_xact_id_if_assigned()`.
`txid_current_snapshot` () → `txid_snapshot`

Fonction Description
Voir `pg_current_snapshot()`.
`txid_snapshot_xip ( txid_snapshot ) → setof bigint` Voir `pg_snapshot_xip()`.
`txid_snapshot_xmax ( txid_snapshot ) → bigint` Voir `pg_snapshot_xmax()`.
`txid_snapshot_xmin ( txid_snapshot ) → bigint` Voir `pg_snapshot_xmin()`.
`txid_visible_in_snapshot ( bigint, txid_snapshot ) → boolean` Voir `pg_visible_in_snapshot()`.
`txid_status ( bigint ) → text` Voir `pg_xact_status()`.

Les fonctions montrées dans Tableau 9.81 fournissent des informations sur le moment où d'anciennes transactions ont été validées. Elles fournissent des informations utiles seulement quand le paramètre track_commit_timestamp est activé, et seulement pour les transactions qui ont été validées après l'activation du paramètre.

Tableau 9.81. Fonctions d'informations sur les transactions validées

Fonction Description
`pg_xact_commit_timestamp ( xid ) → timestamp with time zone` Renvoie l'horodatage de validation d'une transaction.
`pg_xact_commit_timestamp_origin ( xid ) → record ( *timestamp* timestamp with time zone, *roident* oid)` Retourne l'horodatage de validation et l'origine de réplication d'une transaction.
`pg_last_committed_xact () → record ( *xid* xid, *timestamp* timestamp with time zone, *roident* oid )` Renvoie l'identifiant de transaction, l'horodatage de sa validation et l'origine de la réplication pour la dernière transaction validée.

Les fonctions listées dans Tableau 9.82 affichent des informations initialisées lors de l'exécution de la commande `initdb`, telle que la version du catalogue. Elles affichent aussi des informations sur les écritures dans les journaux de transactions et le traitement des checkpoints. Ces informations sont globales à l'instance, et non pas spécifique à une base. Ces fonctions fournissent la plupart des informations en utilisant la mêle source que l'application pg_controldata.

Tableau 9.82. Fonctions des données de contrôle

Fonction Description
`pg_control_checkpoint () → record` Renvoie des informations sur l'état du checkpoint en cours, comme indiqué dans Tableau 9.83.
`pg_control_system () → record` Renvoie des informations sur l'état du fichier de contrôle en cours, comme indiqué dans Tableau 9.84.
`pg_control_init () → record`

Fonction Description
Renvoie des informations sur l'état d'initialisation de l'instance, comme indiqué dans Tableau 9.85.
`pg_control_recovery` () → `record` Renvoie des informations sur l'état de restauration, comme indiqué dans Tableau 9.86.

Tableau 9.83. Colonnes en sortie de `pg_control_checkpoint`

Nom de la colonne	Type de données
checkpoint_lsn	pg_lsn
redo_lsn	pg_lsn
redo_wal_file	text
timeline_id	integer
prev_timeline_id	integer
full_page_writes	boolean
next_xid	text
next_oid	oid
next_multixact_id	xid
next_multi_offset	xid
oldest_xid	xid
oldest_xid_dbid	oid
oldest_active_xid	xid
oldest_multi_xid	xid
oldest_multi_dbid	oid
oldest_commit_ts_xid	xid
newest_commit_ts_xid	xid
checkpoint_time	timestamp with time zone

Tableau 9.84. Colonnes en sortie de `pg_control_system`

Nom de colonne	Type de données
pg_control_version	integer
catalog_version_no	integer
system_identifier	bigint
pg_control_last_modified	timestamp with time zone

Tableau 9.85. Colonnes en sortie de `pg_control_init`

Nom de colonne	Type de données
max_data_alignment	integer
database_block_size	integer
blocks_per_segment	integer
wal_block_size	integer
bytes_per_wal_segment	integer
max_identifier_length	integer

Nom de colonne	Type de données
max_index_columns	integer
max_toast_chunk_size	integer
large_object_chunk_size	integer
float8_pass_by_value	boolean
data_page_checksum_version	integer

Tableau 9.86. Colonnes en sortie de `pg_control_recovery`

Nom de colonne	Type de données
min_recovery_end_lsn	pg_lsn
min_recovery_end_timeline	integer
backup_start_lsn	pg_lsn
backup_end_lsn	pg_lsn
end_of_backup_record_required	boolean

9.27. Fonctions d'administration système

Les fonctions décrites dans cette section sont utilisées pour contrôler et superviser une installation PostgreSQL.

9.27.1. Fonctions de configuration

Tableau 9.87 liste les fonctions disponibles pour requêter et modifier les paramètres de configuration.

Tableau 9.87. Fonctions de configuration

Fonction **Description** **Exemple(s)**
`current_setting ( *setting_name* text [, *missing_ok* boolean ] ) → text` Renvoie la valeur actuelle du paramètre *setting_name*. Si ce paramètre n'existe pas, current_setting renvoie une erreur sauf si *missing_ok* est renseigné et vaut true (auquel cas NULL est renvoyé). Cette fonction correspond à la commande SQL SHOW. `current_setting('datestyle') → ISO, MDY`
`set_config ( *setting_name* text, *new_value* text, *is_local* boolean ) → text` Configure le paramètre *setting_name* à *new_value*, et renvoie cette valeur. Si *is_local* vaut true, la nouvelle valeur s'appliquera uniquement durant la transaction en cours. Si vous voulez que la nouvelle valeur s'applique pour le reste de la session courante, utilisez false à la place. Cette fonction correspond à la commande SQL SET. `set_config('log_statement_stats', 'off', false) → off`

9.27.2. Fonctions d'envoi de signaux

Les fonctions listées dans Tableau 9.88 envoient des signaux de contrôle aux autres processus serveur. L'utilisation de ces fonctions est restreinte aux superutilisateurs par défaut, mais un accès peut être fourni aux utilisateurs en utilisant la commande GRANT, avec quelques exceptions notables.

Chacune de ces fonctions renvoient `true` si le signal a bien été envoyé et `false` si l'envoi du signal a échoué.

Tableau 9.88. Fonctions d'envoi de signaux

Fonction Description
`pg_cancel_backend` (*`pid`* `integer`) → `boolean` Annule la requête en cours de la session du processus serveur dont l'identifiant de processus a été donné. Ceci est aussi autorisé si le rôle appelant est un membre de `pg_signal_backend`. Néanmoins, seuls les superutilisateurs peuvent annuler les processus des superutilisateurs.
`pg_log_backend_memory_contexts` (*`pid`* `integer`) → `boolean` Demande le traçage des contextes mémoire du processus dont l'identifiant est spécifié. Cette fonction doit envoyer la demande aux processus serveur sauf le logger. Ces contextes mémoire seront tracés au niveau `LOG`. Ils apparaîtront dans les traces du serveur selon la configuration (voir Section 20.8 pour plus d'informations), mais ne seront pas envoyés au client, comme indiqué dans client_min_messages. Seul un superutilisateur peut demander le traçage des contextes mémoire.
`pg_reload_conf` () → `boolean` Cause le rechargement des fichiers de configurations par tous les processus du serveur PostgreSQL. (Ceci se fait en envoyant un signal SIGHUP au processus postmaster qui, à son tour, envoie un signal SIGHUP à chacun de ses processus fils.) Vous pouvez utiliser les vues `pg_file_settings` et `pg_hba_file_rules` `pg_ident_file_mappings` pour vérifier d'éventuelles erreurs dans les fichiers de configuration, avant leur rechargement.
`pg_rotate_logfile` () → `boolean` Signale au collecteur de traces qu'il doit changer de fichier de trace immédiatement. Ceci fonctionne seulement quand le collecteur de trace natif est en cours d'exécution.
`pg_terminate_backend` (*`pid`* `integer`, *`timeout`* `bigint` `DEFAULT` `0`) → `boolean` Termine la session pour le processus serveur dont l'identifiant de processus a été donné. Ceci est aussi autorisé si le rôle appelant est un membre du rôle dont le processus est en cours d'arrêt ou si le rôle appelant est membre de `pg_signal_backend`. Néanmoins, seuls les superutilisateurs peuvent terminer les processus des superutilisateurs. Si *`timeout`* n'est pas précisé ou vaut zéro, cette fonction renvoie `true`, que le processus actuel se termine ou pas, confirmant simplement l'envoi correct du signal. Si *`timeout`* est précisé (en millisecondes) et que sa valeur est supérieure à zéro, la fonction attend que le processus se termine effectivement ou que le temps soit dépassé. Si le processus est terminé, la fonction retourne `true`. Si le temps est dépassé, un avertissement est émis et la fonction retourne `false`.

`pg_cancel_backend` et `pg_terminate_backend` envoient des signaux (respectivement SIGINT et SIGTERM) aux processus serveurs identifiés par leur PID. Le PID d'un processus serveur actif est disponible dans la colonne `pid` de la vue `pg_stat_activity` ou en listant les processus `postgres` sur le serveur (en utilisant ps sur Unix ou le Task Manager sur Windows). Le rôle d'un processus serveur actif est disponible dans la colonne `usename` de la vue `pg_stat_activity`.

`pg_log_backend_memory_contexts` peut être utilisée pour tracer les contextes mémoires de processus en arrière-plan. Par exemple :

```
postgres=# SELECT pg_log_backend_memory_contexts(pg_backend_pid());
 pg_log_backend_memory_contexts
--------------------------------
```

```
 t
(1 row)
```

Un message pour chaque contexte mémoire sera tracé. Par exemple :

```
LOG:  logging memory contexts of PID 10377
STATEMENT:  SELECT
 pg_log_backend_memory_contexts(pg_backend_pid());
LOG:  level: 0; TopMemoryContext: 80800 total in 6 blocks; 14432
 free (5 chunks); 66368 used
LOG:  level: 1; pgstat TabStatusArray lookup hash table: 8192 total
 in 1 blocks; 1408 free (0 chunks); 6784 used
LOG:  level: 1; TopTransactionContext: 8192 total in 1 blocks; 7720
 free (1 chunks); 472 used
LOG:  level: 1; RowDescriptionContext: 8192 total in 1 blocks; 6880
 free (0 chunks); 1312 used
LOG:  level: 1; MessageContext: 16384 total in 2 blocks; 5152 free
 (0 chunks); 11232 used
LOG:  level: 1; Operator class cache: 8192 total in 1 blocks; 512
 free (0 chunks); 7680 used
LOG:  level: 1; smgr relation table: 16384 total in 2 blocks; 4544
 free (3 chunks); 11840 used
LOG:  level: 1; TransactionAbortContext: 32768 total in 1 blocks;
 32504 free (0 chunks); 264 used
...
LOG:  level: 1; ErrorContext: 8192 total in 1 blocks; 7928 free (3
 chunks); 264 used
LOG:  Grand total: 1651920 bytes in 201 blocks; 622360 free (88
 chunks); 1029560 used
```

S'il y a plus de 100 contextes fils sous le même parent, les 100 premiers contextes fils sont tracés, le reste des contextes étant résumé. Notez que des appels fréquents à cette fonction peut produire une surchage significative, parce qu'elle génère un grand nombre de messages dans les traces.

9.27.3. Fonctions de contrôle de la sauvegarde

Les fonctions listées dans Tableau 9.89 aident à la réalisation de sauvegardes en ligne. Ces fonctions ne peuvent pas être exécutées lors de la restauration (sauf pour `pg_backup_start`, `pg_backup_stop`, `pg_wal_lsn_diff`).

Pour les détails sur l'utilisation correcte de ces fonctions, voir Section 26.3.

Tableau 9.89. Fonctions de contrôle de la sauvegarde

Fonction **Description**
`pg_create_restore_point` (*name* `text`) → `pg_lsn` Crée un enregistrement marqueur nommé dans le journal de transactions. Ce marqueur peut ensuite être utilisé comme cible de restauration et renvoie l'emplacement correspondant dans le journal de transactions. Le nom donné peut être utilisé avec recovery_target_name pour indiquer le point final de restauration. Évitez de créer plusieurs points de restauration de même nom car la restauration s'arrêtera au premier marqueur rencontré dont le nom correspond à la cible de restauration. Cette fonction est par défaut restreinte aux superutilisateurs mais les autres utilisateurs peuvent se voir donner le droit EXECUTE pour exécuter cette fonction.
`pg_current_wal_flush_lsn` () → `pg_lsn`

Fonction Description
Renvoie l'emplacement actuel de vidage du journal de transaction (voir les notes ci-dessous).
`pg_current_wal_insert_lsn` () → `pg_lsn` Renvoie l'emplacement actuel d'insertion du journal de transaction (voir les notes ci-dessous).
`pg_current_wal_lsn` () → `pg_lsn` Renvoie l'emplacement actuel d'écriture du journal de transaction (voir les notes ci-dessous).
`pg_backup_start` (*`label`* `text` [, *`fast`* `boolean`]) → `pg_lsn` Prépare le serveur à commencer une sauvegarde en ligne. Le seul paramètre requis est un label arbitraire défini par l'utilisateur. (Typiquement, ce serait nom sous lequel le fichier de sauvegarde sera stocké.) Si le deuxième paramètre optionnel est donné à `true`, il demande l'exécution aussi rapide que possible de `pg_backup_start`. Ceci force un checkpoint immédiat ce qui va cause un pic dans les opérations disques, ralentissant les requêtes en cours d'exécution. Cette fonction est par défaut restreinte aux superutilisateurs mais les autres utilisateurs peuvent se voir donner le droit EXECUTE pour exécuter cette fonction.
`pg_backup_stop` ([*`wait_for_archive`* `boolean`]) → `record` (*`lsn`* `pg_lsn`, *`labelfile`* `text`, *`spcmapfile`* `text`) Termine l'exécution d'une sauvegarde en ligne. Le contenu désiré du fichier de label et du fichier de correspondance des tablespaces est renvoyé comme résultat de la fonction, et doit être écrit dans des fichiers dans l'aire de sauvegarde. Ces fichiers ne doivent pas être écrits dans le répertoire actuel des données (le faire pour causer l'échec de redémarrage de PostgreSQL dans le cas d'un crash). Il existe un paramètre optionnel de type `boolean`. À false, la fonction se terminera immédiatement après la fin de sauvegarde, sans attendre l'archivage des journaux de transactions. Ce comportement est seulement utile pour les logiciels de sauvegarde qui surveillent indépendamment l'archivage des journaux de transactions. Sinon, les journaux requis pour rendre la sauvegarde cohérente pourraient manquer et rendre la sauvegarde inutilisable. Par défaut ou quand ce paramètre vaut true, `pg_backup_stop` attendra l'archivage des journaux de transactions si l'archivage est activé. (Sur un serveur secondaire, cela signifie qu'elle attendra que si `archive_mode` est configuré à `always`. Si l'activité en écriture est faible sur le serveur primaire, il pourrait être utile d'exécuter un `pg_switch_wal` sur le serveur primaire pour forcer un changement de journal.) Lors d'une exécution sur un serveur primaire, cette option crée aussi un fichier d'historique de sauvegarde dans la zone d'archivage des journaux de transactions. Le fichier d'historique inclut un label donné par `pg_backup_start`, les emplacements de début et de fin de la sauvegarde, et les dates et heures de début et de fin de la sauvegarde. Après l'enregistrement de l'emplacement de fin, le point d'insertion actuel dans les journaux de transactions est automatiquement avancé au prochain journal, pour que le journal de fin de sauvegarde soit immédiatement archivé pour terminer la sauvegarde. Le résultat de la fonction est un simple enregistrement. La colonne *`lsn`* détient l'emplacement de la fin de la sauvegarde dans les journaux de transactions (qui peut de nouveau être ignoré). La deuxième colonne renvoie le contenu du fichier de label, et la troisième colonne le contenu du fichier de correspondance des tablespaces. Elles doivent être enregistrées comme faisant partie de la sauvegarde et sont requises pour le traitement de la restauration. Cette fonction est par défaut restreinte aux superutilisateurs mais les autres utilisateurs peuvent se voir donner le droit EXECUTE pour exécuter cette fonction.
`pg_switch_wal` () → `pg_lsn`

Fonction Description
Force le serveur à utiliser un nouveau journal de transactions, ce qui permet au journal actuel d'être archivé (en supposant que vous utilisez l'archivage continu). Le résultat est l'emplacement de fin du journal plus 1 dans le journal tout juste terminé. S'il n'y a pas eu d'activité d'écriture depuis le dernier changement de journal, `pg_switch_wal` ne fait rien et renvoie l'emplacement de début du journal actuellement utilisé. Cette fonction est par défaut restreinte aux superutilisateurs mais les autres utilisateurs peuvent se voir donner le droit EXECUTE pour exécuter cette fonction.
`pg_walfile_name` (*`lsn`* `pg_lsn`) → `text` Convertit un emplacement d'un journal de transaction par le nom du fichier du journal de transaction détenant cet emplacement.
`pg_walfile_name_offset` (*`lsn`* `pg_lsn`) → `record` (*`file_name`* `text`, *`file_offset`* `integer`) Convertit un emplacement d'un journal de transaction par le nom du fichier du journal de transaction détenant cet emplacement et le décalage en octets pour aller sur cet emplacement.
`pg_wal_lsn_diff` (*`lsn1`* `pg_lsn`, *`lsn`* `pg_lsn2`) → `numeric` Calcule la différence en octets (*`lsn1`* - *`lsn2`*) entre deux emplacements dans les journaux de transactions. Cette fonction peut être utilisée avec `pg_stat_replication` ou certaines des fonctions listées dans Tableau 9.89 pour obtenir le retard de réplication.

`pg_current_wal_lsn` affiche l'emplacement actuel d'écriture dans le journal de transaction en cours, dans le même format que celui utilisé par les fonctions ci-dessus. De même, `pg_current_wal_insert_lsn` affiche l'emplacement d'insertion actuel et `pg_current_wal_flush_lsn` affiche l'emplacement de vidage actuel. L'emplacement d'insertion est la fin « logiqie » du journal de transaction à n'importe quel moment, alors que l'emplacement d'écriture est la fin de ce qui a été réellement écrit à partir des caches internes du serveur, et l'emplacement de vidage est le dernier emplacement connu pour avoir été réellement écrit sur disque. L'emplacement d'écriture est la fin de ce qui doit être examiné en dehors du serveur et est habituellement ce que vous voulez si vous êtes intéressé par l'archivage des journaux partiellement complets. Les emplacements d'insertion et de vidage sont rendus disponibles principalement dans un but de débogage. Ce sont des opérations de lecture seule et ne nécessite pas l'attribut superutilisateur.

Vous pouvez utiliser `pg_walfile_name_offset` pour extrait le nom du fichier et le décalage en octet du journal de transaction correspondant à partir d'une valeur `pg_lsn`. Par exemple :

```
postgres=# SELECT * FROM
 pg_walfile_name_offset((pg_backup_stop()).lsn);
        file_name         | file_offset
--------------------------+-------------
 00000001000000000000000D |     4039624
(1 row)
```

De la même façon, `pg_walfile_name` extrait juste le nom du fichier du journal de transactions. Quand l'emplacement donné est exactement à la limite du fichier, ces deux fonctions renvoient le nom du journal précédent. Ceci est généralement le comportement souhaité pour gérer l'archivage car le fichier précédent est le dernier qui doit être archivé.

9.27.4. Fonctions de contrôle de la restauration

Les fonctions listées dans Tableau 9.90 fournissent des informations sur le statut actuel d'un serveur secondaire. Ces fonctions peuvent être exécutées pendant une restauration et pendant une exécution normale.

Tableau 9.90. Fonctions d'information sur la restauration

Fonction Description
`pg_is_in_recovery` () → `boolean` Renvoie true si la restauration est toujours en cours.
`pg_last_wal_receive_lsn` () → `pg_lsn` Renvoie le dernier emplacement des journaux de transactions, reçu et synchronisé sur disque par la réplication en flux. Tant que la réplication en flux est en cours, ceci augmentera de façon monotique. Si la restauration s'est terminée, alors cela restera statique à l'emplacement du dernier enregistrement reçu et synchronisé sur disque lors de la restauration. Si la réplication en flux est désactivée ou qu'elle n'a pas commencé, la fonction renvoie `NULL`.
`pg_last_wal_replay_lsn` () → `pg_lsn` Renvoie le dernier emplacement des journaux de transactions, rejoué lors de la restauration. Si la restauration est toujours en cours, ceci augmentera de façon monotique. Si la restauration s'est terminée, alors cela restera statique à l'emplacement du dernier enregistrement appliqué lors de la restauration. Quand le serveur a été démarré normalement sans restauration, la fonction renvoie `NULL`.
`pg_last_xact_replay_timestamp` () → `timestamp with time zone` Renvoie l'horodatage de la dernière transaction rejouée pendant la restauration. C'est le moment où l'enregistrement de validation ou d'annulation a été généré sur le serveur promaire. Si aucune transaction n'a été rejouée pendant la restauration, la fonction renvoie `NULL`. Sinon, si la restauration est toujours en cours, ceci augmentera de façon monotonique. Si la restauration s'est terminée, alors cela restera statique à partir du moment de la dernière transaction appliquée à la restauration. Quand le serveur est démarré normalement sans restauration, la fonction renvoie `NULL`.
`pg_get_wal_resource_managers` () → `setof record` (*`rm_id`* `integer`, *`rm_name`* `text`, *`rm_builtin`* `boolean`) Renvoie les gestionnaires de ressource WAL actuellement chargés dans le système. La colonne *`rm_builtin`* indique s'il s'agit d'un gestionnaire de ressources interne ou d'un gestionnaire de ressources personnalisé chargé par une extension.

Les fonctions listées dans Tableau 9.91 contrôlent la progression de la restauration. Ces fonctions peuvent seulement être exécutées lors d'une restauration.

Tableau 9.91. Fonctions de contrôle de la restauration

Fonction Description
`pg_is_wal_replay_paused` () → `boolean` Renvoie true si la pause de la restauration est demandée.
`pg_get_wal_replay_pause_state` () → `text` Renvoie l'état de la pause de restauration. Les valeurs retournées sont `not paused` si aucune pause n'est requise, `pause requested` si une pause est requise, mais que la restauration n'est pas en pause, et `paused` si la restauration est effectivement en pause.

Fonction Description
`pg_promote` (*`wait`* `boolean` `DEFAULT` `true`, *`wait_seconds`* `integer` `DEFAULT` `60`) → `boolean` Promeut un serveur secondaire au statut primaire. Avec *`wait`* configuré à `true` (la valeur par défaut), la fonction attend jusqu'à ce que la promotion soit terminée ou jusqu'à ce que *`wait_seconds`* se soient écoulées, et renvoie `true` si la promotion a réussi. Elle renvoie `false` dans les autres cas. Si *`wait`* est configuré à `false`, la fonction renvoie `true` immédiatement après avoir envoyé le signal `SIGUSR1` au postmaster pour déclencher sa promotion. Cette fonction est par défaut restreinte aux superutilisateurs mais les autres utilisateurs peuvent se voir donner le droit EXECUTE pour exécuter cette fonction.
`pg_wal_replay_pause` () → `void` Demande la mise en pause de la restauration. Une demande ne signifie pas que la restauration va effectivement s'arrêter. Si vous voulez vérifier que la restauration s'est effectivement arrêtée, vous devez vérifier l'état de la pause de restauration retournée par `pg_get_wal_replay_pause_state()`. Notez que `pg_is_wal_replay_paused()` retourne si oui ou non une demande est faite. Pendant que la restauration est en pause, plus aucune modification ne peut être faite sur la base de données. Si aucun hot standby n'est actif, toutes les requêtes verront le même instantané consistant de la base de données, et plus aucun conflit de requête ne sera généré jusqu'à ce que la restauration redémarre. Cette fonction est par défaut restreinte aux superutilisateurs, mais les autres utilisateurs peuvent se voir donner le droit EXECUTE pour exécuter cette fonction.
`pg_wal_replay_resume` () → `void` Relance la restauration si elle était en pause. Cette fonction est par défaut restreinte aux superutilisateurs mais les autres utilisateurs peuvent se voir donner le droit EXECUTE pour exécuter cette fonction.

`pg_wal_replay_pause` et `pg_wal_replay_resume` ne peuvent pas être exécutés quand une promotion est en cours. Si une promotion est déclenchée alors que la restauration est en pause, l'état de pause se termine et la promotion continue.

Si la réplication en flux est désactivé, l'état de pause peut continuer indéfiniment sans problème. Si la réplication en flux est en cours, alors les enregistrements des journaux de transactions continuent à être rçus, ce qui finira par remplir l'espace disque disponible, suivant la durée de la pause, le taux de génération des journaux de transactions et l'espace disque disponible.

9.27.5. Fonctions de synchronisation d'image

PostgreSQL permet aux sessions des bases de données de synchroniser leur image de base. Une image de base (ou *snapshot*) détermine les données visibles pour la transaction utilisant l'image. Les images synchronisées sont nécessaires quand deux sessions ou plus ont besoin de voir un contenu identique de la base de données. Si deux sessions démarrent leur transaction indépendamment, il existe toujours une possibilité qu'une autre transaction valid entre l'exécution des deux commandes `START TRANSACTION`, de telle façon qu'une session voit l'effet de cette transaction alors que l'autre ne la voit pas.

Pour résoudre ce problème, PostgreSQL autorise une transaction à *exporter* l'image qu'elle utilise. Tant que la transaction exportée reste ouverte, les autres transactions peuvent *importer* son image et, de ce fait, garantir qu'elles partagent exactement la même vue de la base que la première transaction. Mais notez que tout changement effectué par une de ces transactions reste invisible aux autres transactions, comme d'habitude pour les modifications réalisées par des transactions non validées. Donc les transactions sont synchronisées suivant les données pré-existantes, mais agissent normalement pour les modifications qu'elles réalisent.

Les images sont exportées avec la fonction `pg_export_snapshot`, affichée dans Tableau 9.92, et importées avec la commande SET TRANSACTION.

Tableau 9.92. Fonctions de synchronisation d'image

Fonction Description
`pg_export_snapshot` () → `text` Sauvegarde l'image actuelle de la transaction et renvoie une donnée de type `text` identifiant l'image. Cette chaîne doit être fournie (en dehors de la base) aux clients qui veulent importer l'image. L'image est disponible à l'import seulement jusqu'à la fin de la transaction qui l'a exporté. Une transaction peut exporter plus d'une image, si nécessaire. Notez que faire cela n'est utile que pour les transactions en niveau `READ COMMITTED` car, dans les niveaux d'isolation `REPEATABLE READ` et supérieurs, les transactions utilisent la même image tout au long de leur vie. Une fois qu'une transaction a export une image, elle ne peut pas être préparée avec PREPARE TRANSACTION.

9.27.6. Fonctions de gestion de la réplication

Les fonctions listées dans Tableau 9.93 sont pour le contrôle et l'interaction avec les fonctionnalités de réplication. Voir Section 27.2.5, Section 27.2.6 et Chapitre 50 pour des informations sur les fonctionnalités sous-jacentes. L'utilisation de fonctions pour l'origine de réplication est seulement autorisée aux superutilisateurs par défaut, mais peut être autorisée aux autres utilisateurs en utilisant la commande `GRANT`. L'utilisation des fonctions pour les slots de réplication est restreinte aux superutilisateurs et aux utilisateurs ayant l'attribut `REPLICATION`.

La plupart de ces fonctions ont des commandes équivalentes dans le protocole de réplication ; voir Section 55.4.

Les fonctions décrites dans Section 9.27.3, Section 9.27.4 et Section 9.27.5 sont aussi adéquates pour la réplication.

Tableau 9.93. Fonctions de gestion de la réplication

Fonction Description
`pg_create_physical_replication_slot` (*`slot_name`* `name` [, *`immediately_reserve`* `boolean`, *`temporary`* `boolean`, *`twophase`* `boolean`]) → `record` (*`slot_name`* `name`, *`lsn`* `pg_lsn`) Crée un nouveau slot de réplication physique nommé *`slot_name`*. Le deuxième paramètre, optionnel, indique, quand il vaut `true`, que le LSN pour ce slot de réplication doit être réservé immédiatement. Dans le cas contraire, le LSN est réservé lors de la première connexion à partir d'un client de réplication de flux. Les changements en flux d'un slot de réplication sont seulement possible avec le protocole de réplication de flux -- voir Section 55.4. Le troisième argument, optionnel, nommé *`temporary`*, indique, quand il vaut true, que le slot ne doit pas être enregistré de façon permanente sur disque et qu'il a seulement pour but d'être utilisé dans la session en cours. Les slots temporaires sont aussi supprimés dès qu'une erreur survient. Le quatrième paramètre optionnel, *`twophase`*, quand il vaut true, indique que le décodage des transactions préparées est autorisé pour ce slot. Cette fonction correspond à la commande `CREATE_REPLICATION_SLOT ... PHYSICAL` du protocole de réplication.
`pg_drop_replication_slot` (*`slot_name`* `name`) → `void` Supprime le slot de réplication physique ou logique nommé *`slot_name`*. Elle est identique à la commande `DROP_REPLICATION_SLOT` du protocole de réplication.

Fonction Description
pg_create_logical_replication_slot (*slot_name* name, *plugin* name [, *temporary* boolean, *twophase* boolean]) → record (*slot_name* name, *lsn* pg_lsn) Crée un nouveau slot de réplication logique (décodage) nommé *slot_name* en utilisant le plugin de sortie *plugin*. Le troisième paramètre, optionnel, nommé *temporary* indique, quand il vaut true, que le slot ne devrait pas être stockée de façon permanente sur disque et a seulement pour but d'être utilisé par la session en cours. Les slots temporaires sont aussi supprimés en cas d'erreur. Un appel à cette fonction a le même effet que la commande CREATE_REPLICATION_SLOT ... LOGICAL du protocole de réplication.
pg_copy_physical_replication_slot (*src_slot_name* name, *dst_slot_name* name [, *temporary* boolean]) → record (*slot_name* name, *lsn* pg_lsn) Copie un slot de réplication physique existant, nommé *src_slot_name*, en un slot de réplication physique nommé *dst_slot_name*. Le slot physique copié commencera à réserver les journaux de transactions à partir du même LSN que celui du slot copié. Le paramètre *temporary* est en option. Si *temporary* est omis, la valeur du slot source est utilisée.
pg_copy_logical_replication_slot (*src_slot_name* name, *dst_slot_name* name [, *temporary* boolean [, *plugin* name]]) → record (*slot_name* name, *lsn* pg_lsn) Copie un slot existant de réplication logique nommé named *src_slot_name* en un slot de réplication logique nommé *dst_slot_name*, en changeant en option son plugin de sortie et sa persistence. Le slot logique copié commence à partir du même LSN que celui du slot logique source. Les paramètres *temporary* et *plugin* sont optionnels ; s'ils sont omis, leur valeur sur le slot source est utilisée.
pg_logical_slot_get_changes (*slot_name* name, *upto_lsn* pg_lsn, *upto_nchanges* integer, VARIADIC *options* text[]) → setof record (*lsn* pg_lsn, *xid* xid, *data* text) Renvoie les changements dans le slot *slot_name*, en commençant à partir du point où les changements ont été consommés en dernier. Si *upto_lsn* et *upto_nchanges* sont NULL, le décodage logique continuera jusqu'à la fin des journaux de transactions. Si *upto_lsn* n'est pas NULL, le décodage incluera seulement les transactions dont la validation date d'avant le LSN indiqué. Si *upto_nchanges* n'est pas NULL, le décodage stoppera quand le nombre de lignes produites par le décodage dépasse la valeur indiquée. Cependant, notez que le nombre réel de lignes renvoyées pourrait être plus important car cette limite est seulement vérifiée après l'ajout des lignes produites lors du décodage de chaque nouvelle validation de transaction.
pg_logical_slot_peek_changes (*slot_name* name, *upto_lsn* pg_lsn, *upto_nchanges* integer, VARIADIC *options* text[]) → setof record (*lsn* pg_lsn, *xid* xid, *data* text) Se comporte comme la fonction pg_logical_slot_get_changes(), sauf que les changements ne sont pas consommés ; autrement dit, ils sont de nouveau renvoyés lors des prochains appels.
pg_logical_slot_get_binary_changes (*slot_name* name, *upto_lsn* pg_lsn, *upto_nchanges* integer, VARIADIC *options* text[]) → setof record (*lsn* pg_lsn, *xid* xid, *data* bytea) Se comporte comme la fonction pg_logical_slot_get_changes(), sauf que les changements sont renvoyés comme des bytea.

Fonction Description
`pg_logical_slot_peek_binary_changes` (*`slot_name`* `name`, *`upto_lsn`* `pg_lsn`, *`upto_nchanges`* `integer`, `VARIADIC` *`options`* `text[]`) → `setof record` (*`lsn`* `pg_lsn`, *`xid`* `xid`, *`data`* `bytea`) Se comporte comme la fonction `pg_logical_slot_peek_changes()`, sauf que les modifications sont renvoyées sous la forme de `bytea`.
`pg_replication_slot_advance` (*`slot_name`* `name`, *`upto_lsn`* `pg_lsn`) → `record` (*`slot_name`* `name`, *`end_lsn`* `pg_lsn`) Avance la position confirmée actuelle pour le slot de réplication nommé *`slot_name`*. Le slot ne sera pas déplacé en arrière, et il ne sera pas déplacé après l'emplacement d'insertion actuel. Renvoie le nom du slot et la position où il a été réellement avancé. L'information mise à jour de position du slot est écrite dans le prochain checkpoint si l'avance a été possible. Dans le cas d'un crash, le slot pourrait retourner à une position précédente.
`pg_replication_origin_create` (*`node_name`* `text`) → `oid` Crée une origine de réplication avec le nom externe donné, et renvoie l'identifiant interne qui lui a été affecté.
`pg_replication_origin_drop` (*`node_name`* `text`) → `void` Supprime une origine de réplication précédemment créée, incluant toute progression de rejeu associée.
`pg_replication_origin_oid` (*`node_name`* `text`) → `oid` Recherche une origine de réplication par son nom et renvoie son identifiant interne. Si cette origine de réplication n'existe pas, `NULL` est renvoyée.
`pg_replication_origin_session_setup` (*`node_name`* `text`) → `void` Marque la session en cours comme rejouant à partir de l'origine donnée, en autorisant le traçage de la progression du rejeu. Peut seulement être utilisée si aucune origine n'est actuellement sélectionnée. Utilisez `pg_replication_origin_session_reset` pour annuler.
`pg_replication_origin_session_reset` () → `void` Annule les effets de `pg_replication_origin_session_setup()`.
`pg_replication_origin_session_is_setup` () → `boolean` Renvoie true si une origine de réplication a été sélectionnée dans la session en cours.
`pg_replication_origin_session_progress` (*`flush`* `boolean`) → `pg_lsn` Renvoie l'emplacement de rejeu pour l'origine de réplication sélectionnée dans la session en cours. Le paramètre *`flush`* détermine si la transaction locale correspondance sera garantie d'être vidée sur disque ou pas.
`pg_replication_origin_xact_setup` (*`origin_lsn`* `pg_lsn`, *`origin_timestamp`* `timestamp with time zone`) → `void` Marque la transaction en cours comme rejouant une transaction qui a été validée au LSN et à l'horodatage donnés. Peut être appelé quand une origine de réplication a été sélectionnée en utilisant `pg_replication_origin_session_setup`.
`pg_replication_origin_xact_reset` () → `void` Annule les effets de `pg_replication_origin_xact_setup()`.
`pg_replication_origin_advance` (*`node_name`* `text`, *`lsn`* `pg_lsn`) → `void` Configure la progression de la réplication pour le nœud donné à l'emplacement donné. Ceci est principalement utile pour configurer l'emplacement initial ou pour configurer un nouvel emplacement après des changements de configuration ou similaires. Faites

Fonction Description
attention qu'une mauvaise utilisation de cette fonction peut amener à des données répliquées de façon incohérente.
`pg_replication_origin_progress ( node_name text, flush boolean ) → pg_lsn` Renvoie l'emplacement de rejeu pour l'origine de réplication donnée. Le paramètre *flush* détermine si la transaction locale correspondante sera garantie d'être vidée sur disque ou pas.
`pg_logical_emit_message ( transactional boolean, prefix text, content text ) → pg_lsn` `pg_logical_emit_message ( transactional boolean, prefix text, content bytea ) → pg_lsn` Émet un message de décodage logique. Cela peut être utilisé pour passer des messages génériques aux plugins de décodage logique via les journaux de transactions. Le paramètre *transactional* indique si le message devrait faire partie de la transaction en cours ou s'il devrait être écrit immédiatement et décodé dès que le décodeur logique lit l'enregistrement. Le paramètre *prefix* est un préfixe textuel qui peut être utilisé par les plugins de décodage logique pour reconnaître facilement les messages qui les intéressent. Le paramètre *content* est le contenu du message, donné soit au format texte soit au format binaire.

9.27.7. Fonctions de gestion des objets de la base

Les fonctions listées dans Tableau 9.94 calculent l'utilisation de l'espace disque des objets de la base, ou assistent dans la présentation ou la compréhension de ces résultats. Les résultats du type `bigint` correspondent à la taille mesurée en octets. `NULL` est renvoyé si un OID qui ne représente pas un objet existant est fourni à une de ces fonctions.

Tableau 9.94. Fonctions de taille des objets

Fonction Description
`pg_column_size ( "any" ) → integer` Affiche le nombre d'octets utilisés pour enregistrer une valeur individuelle. Si elle est appliquée directement à une valeur d'une colonne de la table, cela représente la compression effectuée.
`pg_column_compression ( "any" ) → text` Montre l'algorithme de compression utilisé pour compresser une valeur individuelle. Retourne `NULL` si la valeur n'est pas compressée.
`pg_database_size ( name ) → bigint` `pg_database_size ( oid ) → bigint` Calcule l'espace disque total utilisé par la base dont le nom ou l'OID est indiqué. Pour utiliser cette fonction, vous devez avoir le droit `CONNECT` sur la base (qui est donné par défaut) ou être un membre du rôle `pg_read_all_stats`.
`pg_indexes_size ( regclass ) → bigint` Calcule l'espace disque total utilisé par les index attachés à une table spécifique.
`pg_relation_size ( relation regclass [, fork text ] ) → bigint` Calcule l'espace disque utilisé par un « élément » de la relation indiquée. (Notez que, dans la plupart des cas, il est préférable d'utiliser les fonctions `pg_total_relation_size` et `pg_table_size`, qui additionnent les tailles de

Fonction Description
tous les éléments d'une relation.) Avec un argument, ceci renvoie la taille de l'élément principal de la relation. Le second argument permet de préciser l'élément à examiner : • `main` renvoie la taille de l'élément principal de la relation. • `fsm` renvoie la taille de la carte des espaces libres (voir Section 73.3) associée à cette relation. • `vm` renvoie la taille de la carte de visibilité (voir Section 73.4) associée à cette relation. • `init` renvoie la taille de l'élément d'initialisation, si elle existe, associée à cette relation.
`pg_size_bytes ( text ) → bigint` Convertit la taille indiquée dans un format lisible par un humain (tel qu'il est donné par `pg_size_pretty`) en un nombre d'octets.
`pg_size_pretty ( bigint ) → text` `pg_size_pretty ( numeric ) → text` Convertit une taille indiquée en octets dans un format plus facilement lisible par un humain avec des unités (bytes, kB, MB, GB, TB ou PB suivant le cas). Notez que les unités sont des puissances de 2, plutôt que des puissances de 10, donc 1kB vaut 1024 octets, 1MB vaut 1024^2 = 1048576 octets, et ainsi de suite.
`pg_table_size ( regclass ) → bigint` Calcule l'espace disque utilisé par la table indiquée, en excluant les index (mais en incluant la table TOAST si elle existe, la carte des espaces libres et la carte de visibilité).
`pg_tablespace_size ( name ) → bigint` `pg_tablespace_size ( oid ) → bigint` Calcule l'espace disque total utilisé dans le tablespace indiqué par son nom ou son OID. Pour utiliser cette fonction, vous devez avoir le droit `CREATE` sur le tablespace ou être un membre du rôle `pg_read_all_stats`, sauf s'il s'agit du tablespace par défaut de la base actuelle.
`pg_total_relation_size ( regclass ) → bigint` Calcule l'espace disque utilisé par la table indiquée, en incluant tous les index et les données TOAST. Le résultat est équivalent à `pg_table_size` + `pg_indexes_size`.

Les fonctions ci-dessus qui opèrent sur les tables et sur les index acceptent un argument `regclass`, qui est simplement l'OID de la table ou de l'index dans le catalogue système `pg_class`. Néanmoins, vous n'avez pas besoin de rechercher manuellement l'OID comme le convertisseur en entrée du type de données `regclass` fera ce travail pour vous. Voir Section 8.19 pour les détails.

Les fonctions listées dans Tableau 9.95 assistent à l'identification des fichiers spécifiques associés aux objets de la base.

Tableau 9.95. Fonctions d'emplacement des objets de la base

Fonction Description
`pg_relation_filenode ( relation regclass ) → oid` Renvoie le numéro « filenode » actuellement affecté à la relation indiquée. Ce numéro est le composant de base du nom du fichier utilisé pour la relation (voir Section 73.1 pour plus d'informations). Pour la plupart des relations, le résultat est identique à `pg_class.relfilenode`, mais pour certains catalogues systèmes, le `relfilenode` vaut zéro et cette fonction doit être utilisée pour obtenir la valeur

Fonction Description
correcte. La fonction renvoie NULL si on lui passe une relation qui n'a pas de stockage, comme une vue.
`pg_relation_filepath` (*`relation`* `regclass`) → `text` Renvoie le chemin complet avec le nom du fichier pour cette relation (chemin relatif au répertoire principal des données de l'instance, `PGDATA`).
`pg_filenode_relation` (*`tablespace`* `oid`, *`filenode`* `oid`) → `regclass` Renvoie l'OID d'une relation d'après l'OID de son tablespace et son filenode. Ceci est la correspondance inverse de `pg_relation_filepath`. Pour une relation dans le tablespace par défaut de la base, le tablespace peut être indiqué par un zéro. Renvoie `NULL` si aucune relation n'est associée aux valeurs données dans la base en cours ou s'il s'agit d'une relation temporaire.

Tableau 9.96 liste les fonctions utilisées pour gérer les collations.

Tableau 9.96. Fonctions de gestion des collations

Fonction Description
`pg_collation_actual_version` (`oid`) → `text` Renvoie la version actuelle de l'objet collation tel qu'il est installé sur le système d'exploitation. S'il est différent de la valeur dans `pg_collation.collversion`, alors les objets dépendant de la collation doivent être reconstruits. Voir aussi ALTER COLLATION.
`pg_database_collation_actual_version` (`oid`) → `text` Renvoie la version actuelle de la collation de la base de données tel qu'elle est actuellement installée dans le système d'exploitation. Si c'est différent de la valeur dans `pg_database.datcollversion`, alors les objets dépendants de la collation pourraient avoir besoin d'être reconstruits. Voir aussi ALTER DATABASE.
`pg_import_system_collations` (*`schema`* `regnamespace`) → `integer` Ajoute des collations dans le catalogue système `pg_collation` basé sur toutes les locales trouvées dans le système d'exploitation. Ceci est ce que `initdb` utilise ; voir Section 24.2.2 pour plus de détails. Si les locales supplémentaires sont installées plus tard dans le système d'exploitation, cette fonction doit être de nouveau exécutée pour ajouter des collations pour les nouvelles locales. Les locales correspondant à des entrées existantes dans `pg_collation` seront ignorées. (Mais les objets de collation basés sur des locales qui ne sont plus présentes au niveau du système d'exploitation ne sont pas supprimées par cette fonction.) Le paramètre *`schema`* sera typiquement `pg_catalog`, mais ce n'est pas obligatoire ; les collations peuvent être installés dans d'autres schémas. La fonction renvoie le nombre des nouveaux objets collations créés. L'utilisation de cette fonction est restreinte aux superutilisateurs.

Tableau 9.97 liste les fonctions qui fournissent des informations sur la structure des tables partitionnées.

Tableau 9.97. Fonctions d'information sur le partitionnement

Fonction Description
`pg_partition_tree` (`regclass`) → `setof record` (*`relid`* `regclass`, *`parentrelid`* `regclass`, *`isleaf`* `boolean`, *`level`* `integer`) Liste les tables ou index dans l'arbre de partitionnement de la table ou de l'index partitionné indiqué, avec une ligne pour chaque partition. Les informations fournies

Fonction Description
incluent l'OID de la partition, l'OID de son parent immédiat, une valeur booléenne indiquant si la partition est une feuille et un entier indiquant son niveau dans la hiérarchie. Le niveau 0 correspond à la table ou l'index en entrée, 1 pour ses partitions immédiates, 2, pour leurs partitions, et ainsi de suite. Ne renvoie aucune ligne si la relation n'existe pas ou s'il ne s'agit ni d'une table partitionnée ni d'une partition.
`pg_partition_ancestors ( regclass ) → setof regclass` Liste les relations ancêtres d'une partition donnée, y compris cette relation. Ne renvoie aucune ligne si la relation n'existe pas ou s'il ne s'agit ni d'une table partitionnée ni d'une partition.
`pg_partition_root ( regclass ) → regclass` Renvoie le parent de plus haut niveau de l'arbre de partition à laquelle appartient la relation donnée. Renvoie `NULL` si la relation n'existe pas ou s'il ne s'agit ni d'une table partitionnée ni d'une partition.

Par exemple, pour vérifier la taille totale de la donnée contenue dans une table partitionnée `measurement`, il est possible d'utiliser la requête suivante :

```
SELECT pg_size_pretty(sum(pg_relation_size(relid))) AS total_size
  FROM pg_partition_tree('measurement');
```

9.27.8. Fonctions de maintenance des index

Tableau 9.98 liste les fonctions disponibles pour les tâches de maintenance d'index. (Notez que ces tâches de maintenance sont normalement réalisés par l'autovacuum ; l'utilisation de ces fonctions est seulement requise dans des cas particuliers.) Ces fonctions ne peuvent pas être exécutées lors de la restauration. L'utilisation de ces fonctions est restreinte aux superutilisateurs et au propriétaire de l'index donné.

Tableau 9.98. Fonctions de maintenance des index

Fonction Description
`brin_summarize_new_values ( index regclass ) → integer` Parcourt l'index BRIN spécifié pour trouver les intervalles de blocs da la table qui ne sont pas actuellement résumés par l'index ; pour chaque intervalle, il crée un nouvel enregistrement de résumé dans l'index en parcourant ces blocs dans la table. Renvoie le nombre de nouveaux résumés d'intervalle de blocs insérés dans l'index.
`brin_summarize_range ( index regclass, blockNumber bigint ) → integer` Résume l'intervalle de blocs couvrant le bloc donné, s'il n'est pas déjà résumé. Cela ressemble à `brin_summarize_new_values`, sauf qu'il traite seulement l'intervalle de blocs qui couvre le numéro de bloc de la table donnée.
`brin_desummarize_range ( index regclass, blockNumber bigint ) → void` Supprime l'enregistement de l'index BRIN qui résume l'intervalle de blocs couvrant le bloc de la table donnée, s'il y en a un.
`gin_clean_pending_list ( index regclass ) → bigint` Supprime la liste « pending » de l'index GIN spécifié en déplaçant les enregistrements à l'intérieur, en masse, vers la structure principale de données GIN. Renvoie le nombre de blocs supprimés dans la liste d'attente. Si l'argument est un index GIN construit avec l'option `fastupdate` désactivée, aucun nettoyage ne survient et le résultat vaut zéro

Fonction Description
car l'index n'a pas de liste d'attente. Voir Section 70.4.1 et Section 70.5 pour les détails sur la liste d'attente et l'option `fastupdate`.

9.27.9. Fonctions génériques d'accès aux fichiers

Les fonctions listées dans Tableau 9.99 fournissent des accès natifs aux fichiers du serveur gérant le serveur. Seuls les fichiers à l'intérieur du répertoire d'instance de la base et ceux ciblés par le paramètre `log_directory` peuvent être accédés, que l'utilisateur est un superutilisateur ou s'est vu donné le rôle `pg_read_server_files`. Utilisez un chemin relatif pour les fichiers du répertoire principal des données, et un chemin correspondant à la configuration du paramètre `log_directory` pour les fichiers de trace.

Notez que donner aux utilisateurs le droit EXECUTE sur `pg_read_file()`, ou les fonctions relatives, les autorise à lire tout fichier du serveur que le processus serveur peut lire. Ces fonctions contournent toutes les vérifications de droit d'accès à la base. Ceci signifie que, pour un exemple, un utilisateur avec un tel accès est capable de lire le contenu de la table `pg_authid` où l'information d'authentification est stockée dans les données de la table de la base. De ce fait, donner l'accès à ces fonctions devrait être considéré avec attention.

Certaines de ces fonctions prennent un paramètre *`missing_ok`* qui modifie le comportement quand le fichier ou le répertoire n'existe pas. Si `true`, la fonction renvoie `NULL` ou un ensemble vide de résultats, comme approprié. Si `false`, une erreur est levée. La valeur par défaut est `false`.

Tableau 9.99. Fonctions génériques d'accès aux fichiers

Fonction Description
`pg_ls_dir` (*`dirname`* `text` [, *`missing_ok`* `boolean`, *`include_dot_dirs`* `boolean`]) → `setof text` Renvoie le nom de tous les fichiers (et répertoires et autres fichiers spéciaux) dans le répertoire donné. Le paramètre *`include_dot_dirs`* indique si les pseudo-répertoires « . » et « .. » sont à inclure dans le résultat ; par défaut, ils sont exclus. Les inclure peut être utile dans *`missing_ok`* vaut `true` pour distinguer un répertoire vide d'un répertoire inexistant. Cette fonction est restreinte par défaut aux superutilisateurs, mais d'autres utilisateurs peuvent se voir donner le droit EXECUTE pour cette fonction.
`pg_ls_logdir` () → `setof record` (*`name`* `text`, *`size`* `bigint`, *`modification`* `timestamp with time zone`) Renvoie le nom, la taille et l'horodatage de la dernière modification de chaque fichier ordinaire dans le répertoire des traces du serveur. Les fichiers dont le nom commence avec un point, les répertoires et les autres fichiers spéciaux sont exclus. Cette fonction est restreinte par défaut aux superutilisateurs et aux utilisateurs membres du rôle `pg_monitor`, mais d'autres utilisateurs peuvent se voir donner le droit EXECUTE pour cette fonction.
`pg_ls_waldir` () → `setof record` (*`name`* `text`, *`size`* `bigint`, *`modification`* `timestamp with time zone`) Renvoie le nom, la taille et l'horodatage de la dernière modification de chaque fichier ordinaire dans le répertoire des journaux de transactions. Les fichiers dont le nom commence avec un point, les répertoires et les autres fichiers spéciaux sont exclus. Cette fonction est restreinte aux superutilisateurs et aux rôles ayant les droits du rôle `pg_monitor` par défaut, mais les autres utilisateurs peuvent se voir donner le droit EXECUTE pour exécuter la fonction.

Fonction Description
`pg_ls_logicalmapdir` () → `setof record` (*`name`* `text`, *`size`* `bigint`, *`modification`* `timestamp with time zone`) Renvoie le nom, la taille et l'horodatage de la dernière modification de chaque fichier ordinaire dans le répertoire `pg_logical/mappings`. Les fichiers dont le nom commence avec un point, les répertoires et les autres fichiers spéciaux sont exclus. Cette fonction est restreinte aux superutilisateurs et aux rôles ayant les droits du rôle `pg_monitor` par défaut, mais les autres utilisateurs peuvent se voir donner le droit EXECUTE pour exécuter la fonction.
`pg_ls_logicalsnapdir` () → `setof record` (*`name`* `text`, *`size`* `bigint`, *`modification`* `timestamp with time zone`) Renvoie le nom, la taille et l'horodatage de la dernière modification de chaque fichier ordinaire dans le répertoire `pg_logical/snapshots`. Les fichiers dont le nom commence avec un point, les répertoires et les autres fichiers spéciaux sont exclus. Cette fonction est restreinte aux superutilisateurs et aux rôles ayant les droits du rôle `pg_monitor` par défaut, mais les autres utilisateurs peuvent se voir donner le droit EXECUTE pour exécuter la fonction.
`pg_ls_replslotdir` (*`slot_name`* `text`) → `setof record` (*`name`* `text`, *`size`* `bigint`, *`modification`* `timestamp with time zone`) Renvoie le nom, la taille et l'horodatage de la dernière modification de chaque fichier ordinaire dans le répertoire `pg_replslot/slot_name`. Les fichiers dont le nom commence avec un point, les répertoires et les autres fichiers spéciaux sont exclus. Cette fonction est restreinte aux superutilisateurs et aux rôles ayant les droits du rôle `pg_monitor` par défaut, mais les autres utilisateurs peuvent se voir donner le droit EXECUTE pour exécuter la fonction.
`pg_ls_archive_statusdir` () → `setof record` (*`name`* `text`, *`size`* `bigint`, *`modification`* `timestamp with time zone`) Renvoie le nom, la taille et l'horodatage de la dernière modification de chaque fichier ordinaire dans le répertoire des statuts d'archivage. Les fichiers dont le nom commence avec un point, les répertoires et les autres fichiers spéciaux sont exclus. Cette fonction est restreinte par défaut aux superutilisateurs et aux utilisateurs membres du rôle `pg_monitor`, mais d'autres utilisateurs peuvent se voir donner le droit EXECUTE pour cette fonction.
`pg_ls_tmpdir` ([*`tablespace`* `oid`]) → `setof record` (*`name`* `text`, *`size`* `bigint`, *`modification`* `timestamp with time zone`) Renvoie le nom, la taille et l'horodatage de la dernière modification de chaque fichier ordinaire dans le répertoire des fichiers temporaires pour le *`tablespace`* indiqué. Si *`tablespace`* est omis, le tablespace `pg_default` est examiné. Les fichiers dont le nom commence avec un point, les répertoires et les autres fichiers spéciaux sont exclus. Cette fonction est restreinte par défaut aux superutilisateurs et aux utilisateurs membres du rôle `pg_monitor`, mais d'autres utilisateurs peuvent se voir donner le droit EXECUTE pour cette fonction.
`pg_read_file` (*`filename`* `text` [, *`offset`* `bigint`, *`length`* `bigint` [, *`missing_ok`* `boolean`]]) → `text` Renvoie tout ou partie d'un fichier texte, en commençant à l'octet indiqué par *`offset`*, en renvoyant au plus *`length`* octets (moins si la fin du fichier est atteinte avant). Si *`offset`* est négatif, il est relatif à la fin du fichier. Si *`offset`* et *`length`* sont omis, le fichier entier est renvoyé. Les octets lus dans le fichier sont interprétés comme une chaîne dans l'encodage de la base ; une erreur est renvoyée s'ils ne sont pas valides pour cet encodage. Cette fonction est restreinte par défaut aux superutilisateurs, mais d'autres utilisateurs peuvent se voir donner le droit EXECUTE pour cette fonction.

Fonction Description
`pg_read_binary_file` (*`filename`* `text` [, *`offset`* `bigint`, *`length`* `bigint` [, *`missing_ok`* `boolean`]]) → `bytea` Renvoie tout ou partie d'un fichier. Cette fonction est identique à `pg_read_file` sauf qu'elle peut lire toute donnée binaire, en renvoyant le résultat sous la forme d'un `bytea`, et non pas d'un `text` ; de ce fait, aucune vérification d'encodage n'est réalisée. Cette fonction est restreinte par défaut aux superutilisateurs, mais d'autres utilisateurs peuvent se voir donner le droit EXECUTE pour cette fonction. En combinant avec la fonction `convert_from`, cette fonction peut être utilisée pour lire un fichier texte dans un encodage spécial et le convertir dans l'encodage de la base : `SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');`
`pg_stat_file` (*`filename`* `text` [, *`missing_ok`* `boolean`]) → `record` (*`size`* `bigint`, *`access`* `timestamp with time zone`, *`modification`* `timestamp with time zone`, *`change`* `timestamp with time zone`, *`creation`* `timestamp with time zone`, *`isdir`* `boolean`) Renvoie un enregistement contenant la taille du fichier, son horodatage de dernier accès, celui de dernière modification, celui de dernier changement de statut (plateformes Unix uniquement), l'horodatage de création (Windows uniquement), et un drapeau indiquant si c'est un répertoire. Cette fonction est restreinte par défaut aux superutilisateurs, mais d'autres utilisateurs peuvent se voir donner le droit EXECUTE pour cette fonction.

9.27.10. Fonctions sur les verrous consultatifs

Les fonctions listées dans Tableau 9.100 gèrent les verrous consultatifs. Pour les détails sur une utilisation correcte de ces fonctions, voir Section 13.3.5.

Toutes ces fonctions ont pour but d'être utilisées pour verrouiller des ressources définies par l'application, qui peuvent être identifiées soit par une valeur de clé sur 64 bits, soit par deux valeurs de clés sur 32 bits (notez que ces deux espaces de clés ne se surchargent pas). Si une autre session détient déjà un verrou conflictuel sur le même identifiant de ressource, les fonctions devront soit attendre que la ressource devienne disponible, soit renvoyer un résultat `false`, comme approprié pour la fonction. Les verrous peuvent être soit partagés soit exclusifs ; un verrou partagé n'entre pas en conflit avec d'autres verrous partagés sur la même ressource, mais entre en conflit avec les verrous exclusifs. Les verrous peuvent se prendre au niveau de la session (pour qu'ils soient détenus jusqu'à leur suppression ou jusqu'à la fin de la session) ou au niveau de la transaction (pour qu'ils soient détenus jusqu'à la fin de la transaction ; il n'existe pas de moyen pour les supprimer manuellement). Les demandes multiples de verrou au niveau session s'empilent, pour que, si le même identifiant de ressource est verrouillé trois fois, alors il doit y avoir trois demandes de déverrouillage pour relâcher la ressource avant que la session se termine.

Tableau 9.100. Fonctions pour les verrous informatifs

Fonction Description
`pg_advisory_lock` (*`key`* `bigint`) → `void` `pg_advisory_lock` (*`key1`* `integer`, *`key2`* `integer`) → `void` Obtient un verrou informatif exclusif niveau session, en attendant si nécessaire.
`pg_advisory_lock_shared` (*`key`* `bigint`) → `void`

Fonction **Description**
`pg_advisory_lock_shared` (*`key1`* `integer`, *`key2`* `integer`) → `void` Obtient un verrou informatif partagé niveau session, en attendant si nécessaire.
`pg_advisory_unlock` (*`key`* `bigint`) → `boolean` `pg_advisory_unlock` (*`key1`* `integer`, *`key2`* `integer`) → `boolean` Relâche un verrou informatif exclusif niveau session précédemment acquis. Renvoie `true` si le verrou a été relâché avec succès. Si le verrou n'était pas détenu, `false` est renvoyé et, en plus, un message d'avertissement SQL est reporté au serveur.
`pg_advisory_unlock_all` () → `void` Relâche tous les verrous informatifs niveau session détenus par la session en cours. (Cette fonction est appelée implicitement à la fin d'une session, même si le client s'est mal déconnecté.)
`pg_advisory_unlock_shared` (*`key`* `bigint`) → `boolean` `pg_advisory_unlock_shared` (*`key1`* `integer`, *`key2`* `integer`) → `boolean` Relâche un verrou informatif partagé niveau session précédemment acquis. Renvoie `true` si le verrou a été relâché avec succès. Si le verrou n'était pas détenu, `false` est renvoyé et, en plus, un message d'avertissement SQL est reporté au serveur.
`pg_advisory_xact_lock` (*`key`* `bigint`) → `void` `pg_advisory_xact_lock` (*`key1`* `integer`, *`key2`* `integer`) → `void` Obtient un verrou informatif exclusif niveau transaction, en attendant si nécessaire.
`pg_advisory_xact_lock_shared` (*`key`* `bigint`) → `void` `pg_advisory_xact_lock_shared` (*`key1`* `integer`, *`key2`* `integer`) → `void` Obtient un verrou informatif partagé niveau transaction, en attendant si nécessaire.
`pg_try_advisory_lock` (*`key`* `bigint`) → `boolean` `pg_try_advisory_lock` (*`key1`* `integer`, *`key2`* `integer`) → `boolean` Obtient un verrou informatif exclusif niveau session si disponible. Cela va soit obtenir immédiatement le verrou et renvoyer `true`, soit renvoyer `false` sans attendre si le verrou ne peut pas être acquis immédiatement.
`pg_try_advisory_lock_shared` (*`key`* `bigint`) → `boolean` `pg_try_advisory_lock_shared` (*`key1`* `integer`, *`key2`* `integer`) → `boolean` Obtient un verrou informatif partagé niveau session si disponible. Cela va soit obtenir immédiatement le verrou et renvoyer `true`, soit renvoyer `false` sans attendre si le verrou ne peut pas être acquis immédiatement.
`pg_try_advisory_xact_lock` (*`key`* `bigint`) → `boolean` `pg_try_advisory_xact_lock` (*`key1`* `integer`, *`key2`* `integer`) → `boolean` Obtient un verrou informatif exclusif niveau transaction si disponible. Cela va soit obtenir immédiatement le verrou et renvoyer `true`, soit renvoyer `false` sans attendre si le verrou ne peut pas être acquis immédiatement.
`pg_try_advisory_xact_lock_shared` (*`key`* `bigint`) → `boolean` `pg_try_advisory_xact_lock_shared` (*`key1`* `integer`, *`key2`* `integer`) → `boolean` Obtient un verrou informatif partagé niveau transaction si disponible. Cela va soit obtenir immédiatement le verrou et renvoyer `true`, soit renvoyer `false` sans attendre si le verrou ne peut pas être acquis immédiatement.

9.28. Fonctions trigger

Bien que la très grande majorité des triggers implique des fonctions triggers écrites par les utilisateurs, PostgreSQL fournit quelques fonctions triggers natives, pouvant être utilisées directement dans des triggers définis par un utilisateur. Elles sont résumées dans Tableau 9.101. (Des fonctions triggers natives supplémentaires existent pour implémenter des contraintes de clés étrangères et des contraintes d'index différés. Ce ne sont pas celles documentées ici car les utilisateurs ne les utilisent pas directement.)

Pour plus d'informations sur la création de triggers, voir CREATE TRIGGER.

Tableau 9.101. Fonctions triggers natives

Fonction **Description** **Exemple d'utilisation**
`suppress_redundant_updates_trigger ( ) → trigger` Supprime les opérations de mise à jour à vide. Voir ci-dessous pour les détails. `CREATE TRIGGER ... suppress_redundant_updates_trigger()`
`tsvector_update_trigger ( ) → trigger` Met à jour automatiquement une colonne `tsvector` à partir de la ou les colonnes documents associées. La configuration de recherche plein texte à utiliser est indiquée par son nom comme argument du trigger. Voir Section 12.4.3 pour les détails. `CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)`
`tsvector_update_trigger_column ( ) → trigger` Met à jour automatiquement une colonne `tsvector` à partir d'une ou plusieurs colonnes textes associées. La configuration de recherche plein texte à utiliser est prise à partir d'une colonne de type `regconfig` dans la table. Voir Section 12.4.3 pour les détails. `CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, tsconfigcol, title, body)`

La fonction `suppress_redundant_updates_trigger`, quand appliqué sur un trigger `BEFORE UPDATE` niveau ligne, empêchera toute mise à jour qui ne change pas réellement les données dans la ligne. Ceci surcharge le comportement habituel qui est de toujours réaliser une mise à jour physique de la ligne, que les données changent ou pas. (Ce comportement habituel rend les mises à jour plus rapides vu qu'il n'est pas besoin de vérifier, et est aussi plus utile dans certains cas.)

Idéalement, vous devriez éviter d'exécuter des mises à jour qui ne changent pas vraiment les données de l'enregistrement. Des mises à jour redondantes peuvent prendre un temps considérable, tout spécialement si beaucoup d'index sont à mettre à jour, et l'espace des lignes mortes sera à nettoyer plus tard. Cependant, détecter ce genre de cas dans le code client n'est pas toujours simple, voire possible, et écrire des expressions pour les détecter peut être source d'erreurs. Une alternative revient à utiliser `suppress_redundant_updates_trigger`, qui ignorera les mises à jour qui ne modifient pas les données. Vous devez faire attention en l'utilisant. Le trigger prend un peu de temps pour chaque enregistrement, donc si la majorité des enregistrements affectés par les mises à jour fait un changement, l'utilisation de ce trigger rendra les mises à jour plus lentes en moyenne.

La fonction `suppress_redundant_updates_trigger` peut être ajoutée ainsi à une table :

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE FUNCTION suppress_redundant_updates_trigger();
```

Dans la plupart des cas, vous avez besoin d'exécuter ce trigger à la fin pour chaque ligne, pour qu'il n'écrase pas l'exécution d'autres triggers qui voudraient modifier la ligne. Gardez en tête que les triggers sont exécutés dans l'ordre alphabétique de leur nom, vous devez de ce fait choisir un nom de trigger qui le fait arriver en dernier parmi les triggers de cette table. (D'où le préfixe « z » dans l'exemple.)

9.29. Fonctions de trigger d'événement

PostgreSQL fournit des fonctions d'aide pour récupérer des informations sur les triggers d'événement.

Pour plus d'informations sur les triggers d'événement, voir Chapitre 40.

9.29.1. Capturer les modifications à la fin d'une commande

`pg_event_trigger_ddl_commands () → setof record`

`pg_event_trigger_ddl_commands` renvoie une liste de commandes DDL exécutées par chaque action utilisateur, lorsqu'elles sont appelées dans une fonction attachée à un trigger d'événement `ddl_command_end`. Si appelé d'un autre contexte, une erreur est levée. `pg_event_trigger_ddl_commands` renvoie une ligne pour chaque commande de base exécutée ; certaines commandes qui sont une simple phrase SQL pourraient renvoyer plus d'une ligne. Cette fonction renvoie les colonnes suivantes :

Nom	Type	Description
`classid`	`oid`	OID du catalogue d'appartenance de l'objet
`objid`	`oid`	OID de l'objet lui-même
`objsubid`	`integer`	identifiant du sous-objet (par exemple, le numéro d'attribut pour une colonne)
`command_tag`	`text`	Balise de commande
`object_type`	`text`	Type de l'objet
`schema_name`	`text`	Nom du schéma d'appartenance de l'objet, s'il existe ; sinon `NULL`. Aucun guillemet n'est utilisé.
`object_identity`	`text`	Texte indiquant l'identité de l'objet, qualifié du schéma. Chaque identifiant inclus dans l'identifié est entre guillemets si nécessaire.
`in_extension`	`boolean`	True si la commande fait partie d'un script d'extension
`command`	`pg_ddl_command`	Une représentation complète de la commande dans le format interne. Ceci ne peut être affiché directement mais peut être passé à d'autres fonctions pour obtenir différentes pièces d'information sur la commande.

9.29.2. Traitement des objets supprimés par une commande DDL

`pg_event_trigger_dropped_objects () → setof record`

`pg_event_trigger_dropped_objects` renvoie une liste de tous les objets supprimés par la commande pour laquelle l'événement `sql_drop` a été appelé. Si elle est appelée dans un autre contexte, une erreur est levée. Cette fonction renvoie les colonnes suivantes :

Nom	Type	Description
`classid`	`oid`	OID du catalogue d'appartenance de l'objet
`objid`	`oid`	OID de l'objet lui-même
`objsubid`	`integer`	Identifiant du sous-objet (par exemple le numéro d'attribut d'une colonne)
`original`	`boolean`	True s'il s'agit d'un des objets racines de la suppression
`normal`	`boolean`	True s'il y avait une relation de dépendance normale dans le graphe de dépendance amenant à cet objet
`is_temporary`	`boolean`	True si c'était un objet temporaire
`object_type`	`text`	Type de l'objet
`schema_name`	`text`	Nom du schéma d'appartenance de l'objet, s'il existe ; sinon `NULL`. Aucun guillemet n'est utilisé.
`object_name`	`text`	Nom de l'objet, si la combinaison du schéma et du nom peut être utilisée comme un identifiant unique pour l'objet ; sinon `NULL`. Aucun guillemet n'est utilisé, et le nom n'est jamais qualifié du schéma.
`object_identity`	`text`	Texte indiquant l'identité de l'objet, qualifié du schéma. Chaque identifiant inclus dans l'identifié est entre guillemets si nécessaire.
`address_names`	`text[]`	Un tableau qui, avec `object_type` et `address_args`, peut être utilisé dans la fonction `pg_get_object_address` pour recréer l'adresse de l'objet dans un serveur distant contenant un objet nommé de façon identique et de même type.

Nom	Type	Description
address_args	text[]	Complément pour address_names

La fonction pg_event_trigger_dropped_objects peut être utilisée dans un trigger d'événement tel que :

```
CREATE FUNCTION test_event_trigger_for_drops()
        RETURNS event_trigger LANGUAGE plpgsql AS $$
DECLARE
    obj record;
BEGIN
    FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
    LOOP
        RAISE NOTICE '% dropped object: % %.% %',
                     tg_tag,
                     obj.object_type,
                     obj.schema_name,
                     obj.object_name,
                     obj.object_identity;
    END LOOP;
END
$$;
CREATE EVENT TRIGGER test_event_trigger_for_drops
   ON sql_drop
   EXECUTE FUNCTION test_event_trigger_for_drops();
```

9.29.3. Gérer un événement de réécriture de table

Les fonctions listées dans Tableau 9.102 fournissent des informations sur une table pour laquelle l'événement table_rewrite a été justement appelé. Si appelée dans un autre contexte, une erreur est levée.

Tableau 9.102. Fonctions d'informations sur la réécriture de table

Fonction Description
pg_event_trigger_table_rewrite_oid () → oid Renvoie l'OID de la table à réécrire.
pg_event_trigger_table_rewrite_reason () → integer Renvoie un code expliquant les raisons de la réécriture. La valeur est un champ de bits construit à partir des valeurs suivantes : 1 (la persistence de la table a été modifiée), 2 (la valeur par défaut d'une colonne a été modifiée), 4 (le type de données de la colonne a été modifié) et 8 (la méthode d'accès à la table a été modifiée).

Ces fonctions peuvent être utilisées dans un trigger d'événement ainsi :

```
CREATE FUNCTION test_event_trigger_table_rewrite_oid()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
BEGIN
  RAISE NOTICE 'rewriting table % for reason %',
```

```
                pg_event_trigger_table_rewrite_oid()::regclass,
                pg_event_trigger_table_rewrite_reason();
END;
$$;

CREATE EVENT TRIGGER test_table_rewrite_oid
                  ON table_rewrite
   EXECUTE FUNCTION test_event_trigger_table_rewrite_oid();
```

9.30. Fonctions d'informations statistiques

PostgreSQL fournit une fonction pour inspecter les statistiques complexes définies en utilisant la commande `CREATE STATISTICS`.

9.30.1. Inspecter les listes MCV

`pg_mcv_list_items ( pg_mcv_list ) → setof record`

`pg_mcv_list_items` renvoie un ensemble d'enregistrements décrivant tous les éléments enregistrés dans une liste MCV multi-colonnes. Elle renvoie les colonnes suivantes :

Nom	Type	Description
`index`	`integer`	index de l'élément dans la liste MCV
`values`	`text[]`	valeurs enregistrées dans l'élément MCV
`nulls`	`boolean[]`	drapeaux identifiant les valeurs `NULL`
`frequency`	`double precision`	fréquence de cet élément MCV
`base_frequency`	`double precision`	fréquence de base de cet élément MCV

La fonction `pg_mcv_list_items` peut être utilisée comme ceci :

```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid
 = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname =
 'stts';
```

Les valeurs du type `pg_mcv_list` peut être obtenues seulement à partir de la colonne `pg_statistic_ext_data.stxdmcv`.

Chapitre 10. Conversion de types

Le mélange de différents types de données dans la même expression peut être requis, intentionnellement ou pas, par les instructions SQL. PostgreSQL possède des fonctionnalités étendues pour évaluer les expressions de type mixte.

Dans la plupart des cas, un utilisateur n'aura pas besoin de comprendre les détails du mécanisme de conversion des types. Cependant, les conversions implicites faites par PostgreSQL peuvent affecter le résultat d'une requête. Quand cela est nécessaire, ces résultats peuvent être atteints directement en utilisant la conversion *explicite* de types.

Ce chapitre introduit les mécanismes et les conventions sur les conversions de types dans PostgreSQL. Référez-vous aux sections appropriées du Chapitre 8 et du Chapitre 9 pour plus d'informations sur les types de données spécifiques, les fonctions et les opérateurs autorisés.

10.1. Aperçu

SQL est un langage fortement typé. C'est-à-dire que chaque élément de données est associé à un type de données qui détermine son comportement et son utilisation permise. PostgreSQL a un système de types extensible qui est beaucoup plus général et flexible que les autres implémentations de SQL. Par conséquent, la plupart des comportements de conversion de types dans PostgreSQL sont régis par des règles générales plutôt que par une heuristique *ad hoc*. Cela permet aux expressions de types mixtes d'être significatives, même avec des types définis par l'utilisateur.

L'analyseur de PostgreSQL divise les éléments lexicaux en cinq catégories fondamentales : les entiers, les nombres non entiers, les chaînes de caractères, les identifiants et les mots-clés. Les constantes de la plupart des types non numériques sont d'abord classifiées comme chaînes de caractères. La définition du langage SQL permet de spécifier le nom des types avec une chaîne et ce mécanisme peut être utilisé dans PostgreSQL pour lancer l'analyseur sur le bon chemin. Par exemple, la requête :

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

a deux constantes littérales, de types `text` et `point`. Si un type n'est pas spécifié pour une chaîne littérale, alors le type `unknown` est assigné initialement pour être résolu dans les étapes ultérieures comme décrit plus bas.

Il y a quatre constructions SQL fondamentales qui exigent des règles distinctes de conversion de types dans l'analyseur de PostgreSQL :

Les appels de fonctions

> Une grande partie du système de types de PostgreSQL est construite autour d'un riche ensemble de fonctions. Les fonctions peuvent avoir un ou plusieurs arguments. Puisque PostgreSQL permet la surcharge des fonctions, le nom seul de la fonction n'identifie pas de manière unique la fonction à appeler ; l'analyseur doit sélectionner la bonne fonction par rapport aux types des arguments fournis.

Les opérateurs

> PostgreSQL autorise les expressions avec des opérateurs de préfixe (un argument) aussi bien qu'infixes (deux arguments). Comme les fonctions, les opérateurs peuvent être surchargés. Du coup, le même problème existe pour sélectionner le bon opérateur.

Le stockage des valeurs

Les instructions SQL `INSERT` et `UPDATE` placent le résultat des expressions dans une table. Les expressions dans une instruction doivent être en accord avec le type des colonnes cibles et peuvent être converties vers celles-ci.

Les constructions `UNION`, `CASE` et des constructions relatives

Comme toutes les requêtes issues d'une instruction `SELECT` utilisant une union doivent apparaître dans un ensemble unique de colonnes, les types de résultats de chaque instruction `SELECT` doivent être assortis et convertis en un ensemble uniforme. De façon similaire, les expressions de résultats d'une construction `CASE` doivent être converties vers un type commun de façon à ce que l'ensemble de l'expression `CASE` ait un type de sortie connu. Quelques autres constructions, telles que `ARRAY[]` et les fonctions `GREATEST` et `LEAST`, nécessitent de la même façon la détermination d'un type commun aux différentes sous-expressions.

Les catalogues systèmes stockent les informations concernant l'existence de conversions entre certains types de données et la façon d'exécuter ces conversions. Les conversions sont appelées *casts* en anglais. Des conversions de types supplémentaires peuvent être ajoutées par l'utilisateur avec la commande CREATE CAST (c'est habituellement réalisé en conjonction avec la définition de nouveaux types de données. L'ensemble des conversions entre les types prédéfinis a été soigneusement choisi et le mieux est de ne pas le modifier).

Une heuristique supplémentaire est fournie dans l'analyseur pour permettre de meilleures estimations sur la bonne conversion de type parmi un groupe de types qui ont des conversions implicites. Les types de données sont divisés en plusieurs *catégories de type* basiques, incluant `boolean`, `numeric`, `string`, `bitstring`, `datetime`, `timespan`, `geometric`, `network` et définis par l'utilisateur. (Pour une liste, voir Tableau 53.65 ; mais notez qu'il est aussi possible de créer des catégories de type personnalisées.) À l'intérieur de chaque catégorie, il peut y avoir un ou plusieurs *types préférés*, qui sont sélectionnés quand il y a un choix possible de types. Avec une sélection attentive des types préférés et des conversions implicites disponibles, il est possible de s'assurer que les expressions ambiguës (celles avec plusieurs solutions candidates) peuvent être résolues d'une façon utile.

Toutes les règles de conversion de types sont écrites en gardant à l'esprit plusieurs principes :

- Les conversions implicites ne doivent jamais avoir de résultats surprenants ou imprévisibles.
- Il n'y aura pas de surcharge depuis l'analyseur ou l'exécuteur si une requête n'a pas besoin d'une conversion implicite de types. C'est-à-dire que si une requête est bien formulée et si les types sont déjà bien distinguables, alors la requête devra s'exécuter sans perte de temps supplémentaire et sans introduire à l'intérieur de celle-ci des appels à des conversions implicites non nécessaires.
- De plus, si une requête nécessite habituellement une conversion implicite pour une fonction et si l'utilisateur définit une nouvelle fonction avec les types des arguments corrects, l'analyseur devrait utiliser cette nouvelle fonction et ne fera plus de conversions implicites en utilisant l'ancienne fonction.

10.2. Opérateurs

L'opérateur spécifique qui est référencé par une expression d'opérateur est déterminé par la procédure ci-dessous. Notez que cette procédure est indirectement affectée par l'ordre d'insertion des opérateurs, car cela va déterminer les sous-expressions prises en entrée des opérateurs. Voir la Section 4.1.6 pour plus d'informations.

Résolution de types pour les opérateurs

1. Sélectionner les opérateurs à examiner depuis le catalogue système `pg_operator`. Si un nom non qualifié d'opérateur est utilisé (le cas habituel), les opérateurs examinés sont ceux avec un

nom et un nombre d'arguments corrects et qui sont visibles dans le chemin de recherche courant (voir la Section 5.9.3). Si un nom qualifié d'opérateur a été donné, seuls les opérateurs dans le schéma spécifié sont examinés.

- (Optional) Si un chemin de recherche trouve de nombreux opérateurs avec des types d'arguments identiques, seul sera examiné celui apparaissant le plus tôt dans le chemin. Mais les opérateurs avec des types d'arguments différents sont examinés sur une base d'égalité indépendamment de leur position dans le chemin de recherche.

2. Vérifier que l'opérateur accepte le type exact des arguments en entrée. Si un opérateur existe (il peut y en avoir uniquement un qui corresponde exactement dans l'ensemble des opérateurs considérés), utiliser cet opérateur. Le manque de correspondance exacte crée un risque de sécurité lors de l'appel, via un nom qualifié [1] (inhabituel), tout opérateur trouvé dans un schéma permettant à des utilisateurs sans confiance de créer des objets. Dans de telles situations, convertir les arguments pour forcer une correspondance exacte.

   a. (Optional) Si un argument lors d'une invocation d'opérateur binaire est de type `unknown` (NdT : inconnu), alors considérer pour ce contrôle que c'est le même type que l'autre argument. Les invocations impliquant deux entrées de type `unknown`, ou un opérateur préfixe avec en entrée une donnée de type `unknown` ne trouveront jamais une correspondance à ce niveau.

   b. (Optional) Si un argument d'un opérateur binaire est de type `unknown` et que l'autre est un domaine, vérifier ensuite s'il existe un opérateur qui accepte le type de base du domaine des deux côtés ; si c'est le cas, l'utiliser.

3. Rechercher la meilleure correspondance.

   a. Se débarrasser des opérateurs candidats pour lesquels les types en entrée ne correspondent pas et qui ne peuvent pas être convertis (en utilisant une conversion implicite) dans le type correspondant. Le type `unknown` est supposé être convertible vers tout. Si un candidat reste, l'utiliser, sinon aller à la prochaine étape.

   b. Si l'argument en entrée est d'un type de domaine, le traiter comme étant le type de base du domaine pour les étapes suivantes. Ceci nous assure que les domaines se comportent comme leur type de base pour la résolution d'opérateurs ambigus.

   c. Parcourir tous les candidats et garder ceux avec la correspondance la plus exacte par rapport aux types en entrée. Garder tous les candidats si aucun n'a de correspondance exacte. Si un seul candidat reste, l'utiliser ; sinon, aller à la prochaine étape.

   d. Parcourir tous les candidats et garder ceux qui acceptent les types préférés (de la catégorie des types de données en entrée) aux positions où la conversion de types aurait été requise. Garder tous les candidats si aucun n'accepte les types préférés. Si seulement un candidat reste, l'utiliser ; sinon aller à la prochaine étape.

   e. Si des arguments en entrée sont `unknown`, vérifier la catégorie des types acceptés à la position de ces arguments par les candidats restants. À chaque position, sélectionner la catégorie `chaîne de caractères` si un des candidats accepte cette catégorie (cette préférence vers les chaînes de caractères est appropriée, car le terme type-inconnu ressemble à une chaîne de caractères). Dans le cas contraire, si tous les candidats restants acceptent la même catégorie de types, sélectionner cette catégorie. Dans le cas contraire, échouer, car le choix correct ne peut pas être déduit sans plus d'indices. Se débarrasser maintenant des candidats qui n'acceptent pas la catégorie sélectionnée. De plus, si des candidats acceptent un type préféré de cette catégorie, se débarrasser des candidats qui acceptent, pour cet argument, les types qui ne sont pas préférés. Conserver tous les candidats si aucun ne survit à ces tests. Si un candidat survit, utilisez-le ; sinon continuer avec l'étape suivante.

[1] Le risque ne vient pas d'un nom sans qualification de schéma, parce qu'un chemin de recherche contenant des schémas permettant à des utilisateurs sans confiance de créer des objets n'est pas un modèle d'utilisation sécurisée des schémas.

f. S'il y a des arguments à la fois `unknown` et connus, et que tous les arguments de type connu ont le même type, supposer que les arguments `unknown` sont de ce même type, et vérifier les candidats qui acceptent ce type aux positions des arguments de type `unknown`. Si un seul candidat réussit ce test, l'utiliser. Sinon, échec.

Quelques exemples suivent.

Exemple 10.1. Résolution du type d'opérateur racine carrée

Il n'existe qu'un seul opérateur racine carrée (préfixe `|/`) défini dans le catalogue standard. Il prend un argument de type `double precision`. Le scanner affecte au début le type `integer` à l'argument dans cette expression :

```
SELECT |/ 40 AS "square root of 40";
 square root of 40
-------------------
 6.324555320336759
```

L'analyseur fait donc une conversion de types sur l'opérande et la requête est équivalente à

```
SELECT |/ CAST(40 AS double precision) AS "square root of 40"
```

Exemple 10.2. Résolution de types pour les opérateurs de concaténation de chaînes

La syntaxe d'une chaîne de caractères est utilisée pour travailler avec les types chaînes, mais aussi avec les types d'extensions complexes. Les chaînes de caractères avec un type non spécifié sont comparées avec les opérateurs candidats probables.

Un exemple avec un argument non spécifié :

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

Dans ce cas, l'analyseur cherche à voir s'il existe un opérateur prenant `text` pour ses deux arguments. Comme il y en a, il suppose que le second argument devra être interprété comme un type `text`.

Voici une concaténation sur des valeurs de type non spécifié :

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

Dans ce cas, il n'y a aucune allusion initiale sur le type à utiliser puisqu'aucun type n'est spécifié dans la requête. Donc, l'analyseur regarde tous les opérateurs candidats et trouve qu'il existe des candidats acceptant en entrée la catégorie chaîne de caractères (string) et la catégorie morceaux de chaînes (bit-string). Puisque la catégorie chaînes de caractères est préférée quand elle est disponible, cette catégorie est sélectionnée. Le type préféré pour la catégorie chaînes étant `text`, ce type est utilisé comme le type spécifique pour résoudre les types inconnus.

Exemple 10.3. Résolution de types pour les opérateurs de valeur absolue et de négation

Le catalogue d'opérateurs de PostgreSQL a plusieurs entrées pour l'opérateur de préfixe @. Ces entrées implémentent toutes des opérations de valeur absolue pour des types variés de données numériques. Une de ces entrées est pour le type `float8` (réel) qui est le type préféré dans la catégorie des numériques. Par conséquent, PostgreSQL utilisera cette entrée quand il sera en face d'un argument de type `unknown` :

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

Le système a compris implicitement que le littéral de type `unknown` est de type `float8` (réel) avant d'appliquer l'opérateur choisi. Nous pouvons vérifier que `float8`, et pas un autre type, a été utilisé :

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

D'un autre côté, l'opérateur préfixe ~ (négation bit par bit) est défini seulement pour les types entiers et non pas pour `float8` (réel). Ainsi, si nous essayons un cas similaire avec ~, nous obtenons :

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You might need
 to add explicit
type casts.
```

Ceci se produit parce que le système ne peut pas décider quel opérateur doit être préféré parmi les différents opérateurs ~ possibles. Nous pouvons l'aider avec une conversion explicite :

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

Exemple 10.4. Résolution du type d'opérateur avec des inclusions de tableaux

Voici un autre exemple de résolution d'un opérateur avec une entrée de type connu et une entrée de type inconnu :

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";

 is subset
-----------
 t
(1 row)
```

Le catalogue d'opérateurs pour PostgreSQL dispose de plusieurs entrées pour un opérateur <@, mais les deux seuls qui peuvent accepter un tableau d'entiers en argument gauche sont ceux d'inclusion de tableaux (`anyarray` <@ `anyarray`) et d'inclusion d'intervalles (`anyelement` <@ `anyrange`).

Comme aucun de ces pseudo-types polymorphiques (voir Section 8.21) n'est considéré comme préféré, l'analyseur ne peut pas résoudre l'ambiguïté sur cette base. Néanmoins, Étape 3.f dit de supposer que le littéral de type inconnu est du même type que l'autre entrée, c'est-à-dire dans cet exemple le tableau d'entiers. Maintenant, seul un des deux opérateurs peut correspondre, donc l'inclusion de tableaux est sélectionnée. (Si l'inclusion d'intervalles avait été sélectionnée, nous aurions obtenu une erreur, car la chaîne n'a pas le bon format pour un intervalle.)

Exemple 10.5. Opérateur personnalisé sur un domaine

Les utilisateurs essaient parfois de déclarer des opérateurs s'appliquant juste à un domaine. Ceci est possible, mais pas aussi intéressant que cela paraît, car les règles de résolution des opérateurs sont conçues pour sélectionner des opérateurs s'appliquant au type de base du domaine. Voici un exemple :

```
CREATE DOMAIN mon_texte AS text CHECK(...);
CREATE FUNCTION mon_texte_eq_text (mon_texte, text) RETURNS boolean
 AS ...;
CREATE OPERATOR = (procedure=mon_texte_eq_text, leftarg=mon_texte,
 rightarg=text);
CREATE TABLE ma_table (val mon_texte);

SELECT * FROM ma_table WHERE val = 'foo';
```

Cette dernière requête n'utilisera pas l'opérateur personnalisé. L'analyseur verra tout d'abord s'il existe un opérateur `mon_texte = mon_texte` (Étape 2.a) ; puis il considérera le type de base du domaine et verra s'il existe un opérateur `text = text` (Étape 2.b), ce qui est vrai ; donc il résout le littéral de type `unknown` comme un type `text` et utilise l'opérateur `text = text`. La seule façon d'obtenir l'utilisation de l'opérateur personnalisé est de convertir explicitement la valeur littérale :

```
SELECT * FROM ma_table WHERE val = text 'foo';
```

de façon à ce que l'opérateur `mon_texte = text` soit immédiatement trouvé suivant la règle de correspondance exacte. Si les règles de meilleure correspondance sont atteintes, elles discriminent complètement les opérateurs sur les domaines. Dans le cas contraire, un tel opérateur créerait trop d'échecs sur des opérateurs ambigus, car les règles de conversion considèrent en permanence un domaine comme réductible à son type de base, et de ce fait, l'opérateur du domaine serait considéré comme utilisable dans les mêmes cas qu'un opérateur de même nom sur le type de base.

10.3. Fonctions

La fonction spécifique référencée par un appel de fonction est déterminée selon les étapes suivantes.

Résolution de types pour les fonctions

1. Sélectionner les fonctions à examiner depuis le catalogue système `pg_proc`. Si un nom non qualifié de fonction est utilisé, les fonctions examinées sont celles avec un nom et un nombre d'arguments corrects et qui sont visibles dans le chemin de recherche courant (voir la Section 5.9.3). Si un nom qualifié de fonction a été donné, seules les fonctions dans le schéma spécifique sont examinées.

    a. (Optional) Si un chemin de recherche trouve de nombreuses fonctions avec des types d'arguments identiques, seule celle apparaissant le plus tôt dans le chemin sera examinée. Mais les fonctions avec des types d'arguments différents sont examinées sur une base d'égalité indépendamment de leur position dans le chemin de recherche.

b. (Optional) Si une fonction est déclarée avec un paramètre `VARIADIC` et que l'appel n'utilise pas le mot-clé `VARIADIC`, alors la fonction est traitée comme si le paramètre tableau était remplacé par une ou plusieurs occurrences de son type élémentaire, autant que nécessaire pour correspondre à l'appel. Après cette expansion, la fonction pourrait avoir des types d'arguments identiques à certaines fonctions non variadic. Dans ce cas, la fonction apparaissant plus tôt dans le chemin de recherche est utilisée ou, si les deux fonctions sont dans le même schéma, celle qui n'est pas VARIADIC est préférée.

   Le manque de correspondance exacte crée un risque de sécurité lors de l'appel, via un nom qualifié [2], d'une fonction avec un nombre variable d'arguments trouvée dans un schéma qui permet à des utilisateurs sans confiance de créer des objets. Un utilisateur mal intentionné peut prendre contrôle et exécuter des fonctions SQL arbitraires comme si vous les aviez exécutées. Remplacez un appel utilisant le mot-clé `VARIADIC` qui contourne ce risque. Les appels utilisant des paramètres `VARIADIC "any"` n'ont généralement pas de formulation équivalente contenant le mot-clé `VARIADIC`. Pour réaliser ces appels en toute sécurité, le schéma de la fonction doit permettre de créer des objets uniquement à des utilisateurs de confiance.

c. (Optional) Les fonctions qui ont des valeurs par défaut pour les paramètres sont considérées comme correspondant à un appel qui omet zéro ou plusieurs des paramètres ayant des valeurs par défaut. Si plus d'une fonction de ce type correspondent à un appel, celui apparaissant en premier dans le chemin des schémas est utilisé. S'il existe deux ou plus de ces fonctions dans le même schéma avec les mêmes types de paramètres pour les paramètres sans valeur par défaut (ce qui est possible s'ils ont des ensembles différents de paramètres par défaut), le système ne sera pas capable de déterminer laquelle sélectionner, ce dont il résultera une erreur « ambiguous function call ».

   Ceci crée un risque de disponibilité lors de l'appel, via un nom qualifié[2], de toute fonction trouvée dans un schéma permettant à des utilisateurs sans confiance de créer des objets. Un utilisateur mal intentionné peut créer une fonction avec le nom d'une fonction existante, répliquant les paramètres de la fonction et ajouter des nouveaux paramètres avec des valeurs par défaut. Ceci empêche les nouveaux appels à la fonction originale. Pour supprimer ce risque, placez les fonctions dans des schémas permettant uniquement aux utilisateurs de confiance de créer des objets.

2. Vérifier que la fonction accepte le type exact des arguments en entrée. Si une fonction existe (il peut y en avoir uniquement une qui corresponde exactement dans tout l'ensemble des fonctions considérées), utiliser cette fonction. Le manque d'une correspondance exacte crée un risque de sécurité lors de l'appel, via un nom qualifié[2], d'une fonction trouvée dans un schéma permettant à des utilisateurs sans confiance de créer des objets. Dans de telles situations, convertir les arguments pour forcer une correspondance exacte. (Les cas impliquant le type `unknown` ne trouveront jamais de correspondance à cette étape).

3. Si aucune correspondance n'est trouvée, vérifier si l'appel à la fonction apparaît être une requête spéciale de conversion de types. Cela arrive si l'appel à la fonction a juste un argument et si le nom de la fonction est le même que le nom (interne) de certains types de données. De plus, l'argument de la fonction doit être soit un type inconnu soit, un type qui a une compatibilité binaire avec le type de données nommé, soit un type qui peut être converti dans le type de données indiqué en appliquant les fonctions d'entrées/sorties du type (c'est-à-dire que la conversion est vers ou à partir d'un type standard de chaîne). Quand ces conditions sont rencontrées, l'appel de la fonction est traité sous la forme d'une spécification `CAST`. [3]

4. Regarder pour la meilleure correspondance.

[2] Le risque ne vient pas d'un nom sans qualification de schéma parce qu'un chemin de recherche contenant des schémas permettant à des utilisateurs sans confiance de créer des objets n'est pas un modèle d'utilisation sécurisée des schémas.

[3] La raison de cette étape est le support des spécifications de conversion au format fonction pour les cas où la vraie fonction de conversion n'existe pas. S'il existe une fonction de conversion, elle est habituellement nommée suivant le nom du type en sortie et donc il n'est pas nécessaire d'avoir un cas spécial. Pour plus d'informations, voir CREATE CAST.

a. Se débarrasser des fonctions candidates pour lesquelles les types en entrée ne correspondent pas et qui ne peuvent pas être convertis (en utilisant une conversion implicite) pour correspondre. Le type `unknown` est supposé être convertible vers n'importe quoi. Si un seul candidat reste, l'utiliser ; sinon, aller à la prochaine étape.

b. Si tout argument en entrée est un type domaine, le traiter comme son type de base pour toutes les étapes suivantes. Ceci nous assure que les domaines agissent comme leurs types de base pour la résolution des fonctions ambiguës.

c. Parcourir tous les candidats et garder ceux avec la correspondance la plus exacte par rapport aux types en entrée. Garder tous les candidats si aucun n'a de correspondance exacte. Si un seul candidat reste, l'utiliser ; sinon, aller à la prochaine étape.

d. Parcourir tous les candidats et garder ceux qui acceptent les types préférés (de la catégorie des types de données en entrée) aux positions où la conversion de types aurait été requise. Garder tous les candidats si aucun n'accepte les types préférés. Si un seul candidat reste, l'utiliser ; sinon, aller à la prochaine étape.

e. Si des arguments en entrée sont `unknown`, vérifier les catégories de types acceptées à la position de ces arguments par les candidats restants. À chaque position, sélectionner la catégorie `chaîne de caractères` si un des candidats accepte cette catégorie (cette préférence envers les chaînes de caractères est appropriée depuis que le terme type-inconnu ressemble à une chaîne de caractères). Dans le cas contraire, si tous les candidats restants acceptent la même catégorie de types, sélectionner cette catégorie. Dans le cas contraire, échouer, car le choix correct ne peut pas être déduit sans plus d'indices. Se débarrasser maintenant des candidats qui n'acceptent pas la catégorie sélectionnée. De plus, si des candidats acceptent un type préféré dans cette catégorie, se débarrasser des candidats qui acceptent, pour cet argument, les types qui ne sont pas préférés. Garder tous les candidats si aucun ne survit à ces tests. Si un seul candidat reste, l'utiliser. Sinon, continuer avec l'étape suivante.

f. S'il y a des arguments à la fois `unknown` et connus, et que tous les arguments de type connu ont le même type, supposer que les arguments `unknown` sont de ce même type, et vérifier les candidats qui acceptent ce type aux positions des arguments de type `unknown`. Si un seul candidat réussit ce test, l'utiliser. Sinon, échec.

Notez que les règles de « correspondance optimale » sont identiques pour la résolution de types concernant les opérateurs et les fonctions. Quelques exemples suivent.

Exemple 10.6. Résolution de types pour les arguments de la fonction arrondi

Il n'existe qu'une seule fonction `round` avec deux arguments (le premier est de type `numeric`, le second est de type `integer`). Ainsi, la requête suivante convertit automatiquement le type du premier argument de `integer` vers `numeric`.

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

La requête est en fait transformée par l'analyseur en

```
SELECT round(CAST (4 AS numeric), 4);
```

Puisque le type `numeric` est initialement assigné aux constantes numériques avec un point décimal, la requête suivante ne requièrera pas une conversion de types et pourra par conséquent être un peu plus efficace :

```
SELECT round(4.0, 4);
```

Exemple 10.7. Résolution de fonction à arguments variables

```
CREATE FUNCTION public.variadic_example(VARIADIC numeric[]) RETURNS
 int
  LANGUAGE sql AS 'SELECT 1';
CREATE FUNCTION
```

Cette fonction accepte, mais ne requiert pas, le mot-clé VARIADIC. Elle tolère des arguments à la fois entier et numérique :

```
SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                1 |                1 |                1
(1 row)
```

Néanmoins, le premier et le deuxième appel préféreront des fonctions plus spécifiques si elles sont disponibles :

```
CREATE FUNCTION public.variadic_example(numeric) RETURNS int
  LANGUAGE sql AS 'SELECT 2';
CREATE FUNCTION

CREATE FUNCTION public.variadic_example(int) RETURNS int
  LANGUAGE sql AS 'SELECT 3';
CREATE FUNCTION

SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                3 |                2 |                1
(1 row)
```

Étant donné la configuration par défaut et si seule la première fonction existe, le premier et le deuxième appel ne sont pas sécurisés. Tout utilisateur peut les intercepter en créant la deuxième et la troisième fonction. En utilisant une correspondance exacte du type d'argument et en utilisant le mot-clé `VARIADIC`, le troisième appel est sécurisé.

Exemple 10.8. Résolution de types pour les fonctions retournant un segment de chaîne

Il existe plusieurs fonctions `substr`, une d'entre elles prend les types `text` et `integer`. Si cette fonction est appelée avec une constante de chaînes d'un type inconnu, le système choisit la fonction candidate qui accepte un argument issu de la catégorie préférée `string` (c'est-à-dire de type `text`).

```
SELECT substr('1234', 3);
```

```
 substr
--------
     34
(1 row)
```

Si la chaîne de caractères est déclarée comme étant du type `varchar` (chaîne de caractères de longueur variable), ce qui peut être le cas si elle vient d'une table, alors l'analyseur essaiera de la convertir en `text` :

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

Ceci est transformé par l'analyseur en

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **Note**
>
> L'analyseur apprend depuis le catalogue `pg_cast` que les types `text` et `varchar` ont une compatibilité binaire, ce qui veut dire que l'un peut être passé à une fonction qui accepte l'autre sans avoir à faire de conversion physique. Par conséquent, aucun appel de conversion de types n'est réellement inséré dans ce cas.

Et si la fonction est appelée avec un argument de type `integer`, l'analyseur essaie de le convertir en `text` :

```
SELECT substr(1234, 3);
ERROR:  function substr(integer, integer) does not exist
HINT:  No function matches the given name and argument types. You
 might need
to add explicit type casts.
```

Ceci ne fonctionne pas, car `integer` n'a pas de conversion implicite vers `text`. Néanmoins, une conversion explicite fonctionnera :

```
SELECT substr(CAST (1234 AS text), 3);

 substr
--------
     34
(1 row)
```

10.4. Stockage de valeurs

Les valeurs qui doivent être insérées dans une table sont converties vers le type de données de la colonne de destination selon les règles suivantes.

Conversion de types pour le stockage de valeurs

1. Vérifier qu'il y a une correspondance exacte avec la cible.

2. Dans le cas contraire, essayer de convertir l'expression vers le type cible. Cela réussira s'il y a une conversion d'affectation (cast) enregistrée entre ces deux types dans le catalogue `pg_cast` (voir CREATE CAST). Si une expression est de type inconnu, le contenu de la chaîne littérale sera fourni à l'entrée de la routine de conversion pour le type cible.

3. Vérifier s'il y a une conversion de taille pour le type cible. Une conversion de taille est une conversion d'un type vers lui-même. Si elle est trouvée dans le catalogue `pg_cast`, l'appliquer à l'expression avant de la stocker dans la colonne de destination. La fonction d'implémentation pour une telle conversion prend toujours un paramètre supplémentaire de type `integer`, qui reçoit la valeur `atttypmod` de la colonne de destination (en fait, sa valeur déclarée ; l'interprétation de `atttypmod` varie pour les différents types de données). La fonction de conversion est responsable de l'application de toute sémantique dépendante de la longueur comme la vérification de la taille ou une troncature.

Exemple 10.9. Conversion de types pour le stockage de `character`

Pour une colonne cible déclarée comme `character(20)`, la déclaration suivante montre que la valeur stockée a la taille correcte :

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v           | octet_length
----------------------+--------------
 abcdef               |           20
(1 row)
```

Voici ce qui s'est réellement passé ici : les deux types inconnus sont résolus en `text` par défaut, permettant à l'opérateur `||` de les résoudre comme une concaténation de `text`. Ensuite, le résultat `text` de l'opérateur est converti en `bpchar` (« blank-padded char », le nom interne du type de données `character` (caractère)) pour correspondre au type de la colonne cible (comme la conversion de `text` à `bpchar` est compatible binairement, cette conversion n'insère aucun appel réel à une fonction). Enfin, la fonction de taille `bpchar(bpchar, integer, boolean)` est trouvée dans le catalogue système et appliquée au résultat de l'opérateur et à la longueur de la colonne stockée. Cette fonction de type spécifique effectue le contrôle de la longueur requise et ajoute des espaces pour combler la chaîne.

10.5. Constructions `UNION`, `CASE` et constructions relatives

Les constructions SQL avec des `UNION` doivent potentiellement faire correspondre des types différents pour avoir un ensemble unique dans le résultat. L'algorithme de résolution est appliqué séparément à chaque colonne de sortie d'une requête d'union. Les constructions `INTERSECT` et `EXCEPT` résolvent des types différents de la même manière qu'`UNION`. Quelques autres constructions `CASE`, `ARRAY`, `VALUES`, et les fonctions `GREATEST` et `LEAST` utilisent le même algorithme pour faire correspondre les expressions qui les composent et sélectionner un type de résultat.

Résolution des types pour `UNION`, `CASE` et les constructions relatives

1. Si toutes les entrées sont du même type et qu'il ne s'agit pas du type `unknown`, résoudre comme étant de ce type.

2. Si un type en entrée est un domaine, le traiter comme le type de base du domaine pour toutes les étapes suivantes. [4]

3. Si toutes les entrées sont du type `unknown`, résoudre comme étant du type `text` (le type préféré de la catégorie chaîne). Dans le cas contraire, les entrées `unknown` seront ignorées pour les règles restantes.

4. Si toutes les entrées non inconnues ne sont pas toutes de la même catégorie, échouer.

5. Sélectionne le premier type en entrée connu comme le type candidat, puis considère chaque autre type en entrée connu, de gauche à droite. [5] Si le type candidat peut être converti implicitement dans l'autre type, mais pas vice versa, sélectionne l'autre type comme nouveau type candidat. Puis continue en considérant les entrées restantes. Si, à tout moment de ce traitement, un type préféré est sélectionné, s'arrête de considérer les entrées supplémentaires.

6. Convertir toutes les entrées vers le type candidat final. Échoue s'il n'y a pas de conversion implicite à partir de l'entrée donnée vers le type candidat.

Quelques exemples suivent.

Exemple 10.10. Résolution de types avec des types sous-spécifiés dans une union

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

Ici, la chaîne de type inconnu `'b'` sera convertie vers le type `text`.

Exemple 10.11. Résolution de types dans une union simple

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

Le littéral `1.2` est du type `numeric` et la valeur `1`, de type `integer`, peut être convertie implicitement vers un type `numeric`, donc ce type est utilisé.

Exemple 10.12. Résolution de types dans une union transposée

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

[4] Un peu comme le traitement des arguments de type domaine pour les opérateurs et les fonctions, ce comportement permet à un type domaine d'être préservé par un `UNION` ou toute construction similaire, tant que l'utilisateur veille à ce que toutes les entrées soient explicitement ou implicitement du type exact. Dans le cas contraire, le type de base du domaine sera utilisé.

[5] Pour des raisons historiques, `CASE` traite sa clause `ELSE` (s'il y en a) comme la « première » entrée, et les clauses `THEN` sont considérés après ça. Dans tous les autres cas, « de gauche à droite » signifie l'ordre dans lequel les expressions apparaissent dans le texte de la requête.

Dans cet exemple, le type `real` (réel) ne peut pas être implicitement converti en `integer` (entier), mais un `integer` peut être implicitement converti en `real` ; le résultat de l'union est résolu comme étant un `real`.

Exemple 10.13. Résolution de type dans une union imbriquée

```
SELECT NULL UNION SELECT NULL UNION SELECT 1;

ERROR:  UNION types text and integer cannot be matched
```

Cet échec survient parce que PostgreSQL traite plusieurs `UNION` comme une imbrication d'opérations sous forme de paires ; c'est-à-dire que cette entrée est identique à

```
(SELECT NULL UNION SELECT NULL) UNION SELECT 1;
```

Le `UNION` interne est résolu en émettant le type `text`, suivant les règles données ci-dessus. Puis le `UNION` externe a en entrée les types `text` et `integer`, amenant l'erreur observée. Le problème peut être corrigé en s'assurant que le `UNION` le plus à gauche dispose au moins d'une entrée du type résultant désiré.

Les opérations `INTERSECT` et `EXCEPT` procèdent de la même façon. Néanmoins, les autres constructions décrites dans cette section considèrent toutes leurs entrées en une seule étape de résolution.

10.6. Colonnes de sortie du SELECT

Les règles des sections précédentes auront pour résultat d'assigner les types de données autres que `unknown` à toutes les expressions dans la requête SQL, à l'exception des littéraux de type non spécifié apparaissant en tant que colonne de sortie dans une commande `SELECT`. Par exemple, dans :

```
SELECT 'Hello World';
```

rien ne permet d'identifier quel devrait être le type du litteral chaîne de caractères. Dans ce cas, PostgreSQL résoudra le type du littéral en type `text`.

Lorsque le `SELECT` fait partie d'une construction `UNION` (ou `INTERSECT EXCEPT`) ou lorsqu'il apparaît dans un `INSERT ... SELECT`, cette règle n'est pas appliquée, car les règles des sections précédentes ont priorité. Le type d'un littéral au type non spécifié peut être choisi d'après l'autre partie d'un `UNION` dans le premier cas, ou d'après la colonne de destination dans le second cas.

Dans ce but, les listes `RETURNING` sont traitées de la même manière que les retours de `SELECT`.

> **Note**
>
> Avant la version 10 de PostgreSQL, cette règle n'existait pas, et les littéraux aux types non spécifiés dans la sortie d'un `SELECT` étaient de type `unknown`. Suite à plusieurs conséquences néfastes, ce comportement a été modifié.

Chapitre 11. Index

L'utilisation d'index est une façon habituelle d'améliorer les performances d'une base de données. Un index permet au serveur de bases de données de retrouver une ligne spécifique bien plus rapidement. Mais les index ajoutent aussi une surcharge au système de base de données dans son ensemble, si bien qu'ils doivent être utilisés avec discernement.

11.1. Introduction

Soit une table définie ainsi :

```
CREATE TABLE test1 (
    id integer,
    contenu varchar
);
```

et une application qui utilise beaucoup de requêtes de la forme :

```
SELECT contenu FROM test1 WHERE id = constante;
```

Sans préparation, le système doit lire la table `test1` dans son intégralité, ligne par ligne, pour trouver toutes les lignes qui correspondent. S'il y a beaucoup de lignes dans `test1`, et que seules quelques lignes correspondent à la requête (peut-être même zéro ou une seule), alors, clairement, la méthode n'est pas efficace. Mais si le système doit maintenir un index sur la colonne `id`, alors il peut utiliser une manière beaucoup plus efficace pour trouver les lignes recherchées. Il se peut qu'il n'ait ainsi qu'à parcourir quelques niveaux d'un arbre de recherche.

Une approche similaire est utilisée dans la plupart des livres autres que ceux de fiction : les termes et concepts fréquemment recherchés par les lecteurs sont listés par ordre alphabétique à la fin du livre. Le lecteur qui recherche un mot particulier peut facilement parcourir l'index, puis aller directement à la page (ou aux pages) indiquée(s). De la même façon que l'auteur doit anticiper les sujets que les lecteurs risquent de rechercher, il est de la responsabilité du programmeur de prévoir les index qui sont utiles.

La commande suivante peut être utilisée pour créer un index sur la colonne `id` :

```
CREATE INDEX test1_id_index ON test1 (id);
```

Le nom `test1_id_index` peut être choisi librement, mais il est conseillé de choisir un nom qui rappelle le but de l'index.

Pour supprimer l'index, on utilise la commande `DROP INDEX`. Les index peuvent être ajoutés et retirés des tables à tout moment.

Une fois un index créé, aucune intervention supplémentaire n'est nécessaire : le système met à jour l'index lorsque la table est modifiée et utilise l'index dans les requêtes lorsqu'il pense que c'est plus efficace qu'une lecture complète de la table. Il faut néanmoins lancer la commande `ANALYZE` régulièrement pour permettre à l'optimiseur de requêtes de prendre les bonnes décisions. Voir le Chapitre 14 pour comprendre quand et pourquoi l'optimiseur décide d'utiliser ou de ne *pas* utiliser un index.

Les index peuvent aussi bénéficier aux commandes `UPDATE` et `DELETE` à conditions de recherche. De plus, les index peuvent être utilisés dans les jointures. Ainsi, un index défini sur une colonne qui fait partie d'une condition de jointure peut aussi accélérer significativement les requêtes avec jointures.

En général, les index PostgreSQL peuvent être utilisées pour optimiser les requêtes qui contiennent une ou plusieurs clauses `WHERE` ou `JOIN` de la forme

colonne-indexee operateur-indexable valeur-comparaison

Ici, la *colonne-indexee* est n'importe quel colonne ou expression de la définition de l'index. L'*operateur-indexable* est un opérateur qui est un membre de la *classe d'opérateurs* de l'index pour la colonne indexée. (Plus de détails là-dessus ci-dessous.) et la *valeur-comparaison* peut être toute expression qui n'est pas volatile et ne fait pas référence à la table de l'index.

Dans certains cas, l'optimiseur de requêtes peut extraire une clause indexable de cette forme à partir d'une autre construction SQL. Un exemple simple qui, si la clause originale est

```
valeur-comparaison operateur colonne-indexee
```

alors elle peut être inversée dans une forme indexable si l'*operateur* original a un opérateur de commutation qui est un membre de la classe d'opérateurs de l'index.

Créer un index sur une grosse table peut prendre beaucoup de temps. Par défaut, PostgreSQL autorise la lecture (`SELECT`) sur la table pendant la création d'un index sur celle-ci, mais interdit les écritures (`INSERT`, `UPDATE`, `DELETE`). Elles sont bloquées jusqu'à la fin de la construction de l'index. Dans des environnements de production, c'est souvent inacceptable. Il est possible d'autoriser les écritures en parallèle de la création d'un index, mais quelques précautions sont à prendre. Pour plus d'informations, voir Construire des index en parallèle.

Après la création d'un index, le système doit le maintenir synchronisé avec la table. Cela rend plus lourdes les opérations de manipulation de données. Les index peuvent aussi empêcher la création de heap-only tuples. C'est pourquoi les index qui sont peu, voire jamais, utilisés doivent être supprimés.

11.2. Types d'index

PostgreSQL propose plusieurs types d'index : B-tree, Hash, GiST, SP-GiST, GIN, BRIN et l'extension bloom. Chaque type d'index utilise un algorithme différent qui convient à un type particulier de clauses indexables. Par défaut, la commande `CREATE INDEX` crée un index B-tree, ce qui convient dans la plupart des situations. Les autres types d'index sont sélectionnés en écrivant le mot clé `USING` suivi du nom du type d'index. Par exemple, pour créer un index hash :

```
CREATE INDEX name ON table USING HASH (column);
```

11.2.1. B-Tree

Les index B-tree savent traiter les requêtes d'égalité et par tranches sur des données qu'il est possible de trier. En particulier, l'optimiseur de requêtes de PostgreSQL considère l'utilisation d'un index B-tree lorsqu'une colonne indexée est utilisée dans une comparaison qui utilise un de ces opérateurs :

```
<   <=   =   >=   >
```

Les constructions équivalentes à des combinaisons de ces opérateurs, comme `BETWEEN` et `IN`, peuvent aussi être implémentées avec une recherche par index B-tree. Une condition `IS NULL` ou `IS NOT NULL` sur une colonne indexée peut aussi être utilisée avec un index B-tree.

L'optimiseur peut aussi utiliser un index B-tree pour des requêtes qui utilisent les opérateurs de recherche de motif `LIKE` et `~` *si* le motif est une constante et se trouve au début de la chaîne à rechercher -- par exemple, `col LIKE 'foo%'` ou `col ~ '^foo'`, mais pas `col LIKE '%bar'`. Toutefois, si la base de données n'utilise pas la locale C, il est nécessaire de créer l'index avec une classe d'opérateurs spéciale pour supporter l'indexation à correspondance de modèles. Voir la Section 11.10 ci-dessous. Il est aussi possible d'utiliser des index B-tree pour `ILIKE` et `~*`, mais seulement si le modèle débute par des caractères non alphabétiques, c'est-à-dire des caractères non affectés par les conversions majuscules/minuscules.

Les index B-tree peuvent aussi être utilisés pour récupérer des données triées. Ce n'est pas toujours aussi rapide qu'un simple parcours séquentiel suivi d'un tri, mais c'est souvent utile.

11.2.2. Hash

Les index Hash enregistrent un code de hachage sur 32 bits dérivé de la valeur d'une colonne indexée. De ce fait, de tels index peuvent seulement gérer des comparaisons simples d'égalité. Le planificateur de requêtes considère l'utilisation d'un index hash quand une colonne indexée est impliquée dans une comparaison avec l'opérateur d'égalité :

=

11.2.3. GiST

Les index GiST ne constituent pas un unique type d'index, mais plutôt une infrastructure à l'intérieur de laquelle plusieurs stratégies d'indexage peuvent être implémentées. De cette façon, les opérateurs particuliers avec lesquels un index GiST peut être utilisé varient en fonction de la stratégie d'indexage (la *classe d'opérateur*). Par exemple, la distribution standard de PostgreSQL inclut des classes d'opérateurs GiST pour plusieurs types de données géométriques à deux dimensions, qui supportent des requêtes indexées utilisant ces opérateurs :

<< &< &> >> <<| &<| |&> |>> @> <@ ~= &&

Voir la Section 9.11 pour connaître la signification de ces opérateurs. Les classes d'opérateurs GiST incluses dans la distribution standard sont documentées dans Tableau 68.1. Beaucoup de classes d'opérateurs GiST sont disponibles dans l'ensemble des `contrib` ou comme projet séparé. Pour plus d'informations, voir Chapitre 68.

Les index GiST sont aussi capables d'optimiser des recherches du type « voisin-le-plus-proche » (*nearest-neighbor*), comme :

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT
 10;
```

qui trouve les dix places les plus proches d'une cible donnée. Cette fonctionnalité dépend de nouveau de la classe d'opérateurs utilisée. Dans Tableau 68.1, les opérateurs pouvant être utilisés de cette façon sont listés dans la colonne « Opérateurs de tri ».

11.2.4. SP-GiST

Les index SP-GiST, tout comme les index GiST, offrent une infrastructure qui supporte différents types de recherches. SP-GiST permet l'implémentation d'une grande étendue de structures de données non balancées, stockées sur disque comme les *quadtrees*, les arbres k-d, et les arbres *radix*. PostgreSQL inclut les classes d'opérateurs SP-GiST pour les points à deux dimensions, qui supportent les requêtes indexées en utilisant les opérateurs suivants :

<< >> ~= <@ <<| |>>

(Voir Section 9.11 pour la signification de ces opérateurs.) Les classes d'opérateurs SP-GiST incluses dans la distribution standard sont documentées dans Tableau 69.1. Pour plus d'informations, voir Chapitre 69.

Comme GiST, SP-GiST supporte les recherches de type « voisin-le-plus-proche ». Pour les classes d'opérateurs SP-GiST qui supportent le tri par distance, l'opérateur correspondant est spécifié dans la colonne « Opérateurs d'ordre » dans Tableau 69.1.

11.2.5. GIN

Les index GIN sont des « index inversés » qui sont appropriés quand les valeurs à index contiennent plusieurs valeurs composantes, comme les tableaux. Un index inversé contient une entrée séparée pour chaque valeur composante, et peut gérer efficacement les requêtes testant la présence de valeurs composantes spécifiques.

Comme GiST et SP-GiST, GIN supporte différentes stratégies d'indexation utilisateur. Les opérateurs particuliers avec lesquels un index GIN peut être utilisé varient selon la stratégie d'indexation. Par exemple, la distribution standard de PostgreSQL inclut une classe d'opérateurs GIN pour des tableaux à une dimension qui supportent les requêtes indexées utilisant ces opérateurs :

```
<@    @>    =    &&
```

Voir Section 9.19 pour la signification de ces opérateurs. Les classes d'opérateurs GIN incluses dans la distribution standard sont documentées dans Tableau 70.1. Beaucoup d'autres classes d'opérateurs GIN sont disponibles dans les modules `contrib` ou dans des projets séparés. Pour plus d'informations, voir Chapitre 70.

11.2.6. BRIN

Les index BRIN (raccourci pour *Block Range INdexes*) stockent des résumés sur les valeurs enregistrées dans des blocs physiques successifs de la table. De ce fait, ils sont plus efficaces pour les colonnes dont les valeurs sont corrélées avec l'ordre physique des lignes dans la table. Comme GiST, SP-GiST et GIN, BRIN supporte plusieurs stratégies d'indexation, et les opérateurs compatibles avec un index BRIN varient suivant la stratégie d'indexation. Pour les types de données qui ont un ordre de tri linéaire, la donnée indexée correspond aux valeurs minimale et maximale des valeurs de la colonne pour chaque ensemble de blocs. Cela supporte les requêtes indexées utilisant ces opérateurs :

```
<    <=    =    >=    >
```

Les classes d'opérateurs BRIN incluses dans la distribution standard sont documentées dans Tableau 71.1. Pour plus d'informations, voir Chapitre 71.

11.3. Index multicolonnes

Un index peut porter sur plusieurs colonnes d'une table. Soit, par exemple, une table de la forme :

```
CREATE TABLE test2 (
  majeur int,
  mineur int,
  nom varchar
);
```

(cas d'un utilisateur gardant son répertoire `/dev` dans une base de données...) et que des requêtes comme :

```
SELECT nom FROM test2 WHERE majeur = constante AND mineur
 = constante;
```

sont fréquemment exécutées. Il peut alors être souhaitable de définir un index qui porte sur les deux colonnes `majeur` et `mineur`. Ainsi :

```
CREATE INDEX test2_mm_idx ON test2 (majeur, mineur);
```

Actuellement, seuls les types d'index B-tree, GiST, GIN et BRIN supportent les index multicolonnes. Qu'il puisse y avoir plusieurs colonnes clés est indépendant de si les colonnes `INCLUDE` peuvent être ajoutées à l'index. Les index peuvent avoir jusqu'à 32 colonnes en incluant les colonnes

INCLUDE. Cette limite peut être modifiée à la compilation de PostgreSQL. Voir le fichier pg_config_manual.h.

Un index B-tree multicolonne peut être utilisé avec des conditions de requêtes impliquant un sous-ensemble quelconque de colonnes de l'index. L'index est toutefois plus efficace lorsqu'il y a des contraintes sur les premières colonnes (celles de gauche). La règle exacte est la suivante : les contraintes d'égalité sur les premières colonnes, et toute contrainte d'inégalité sur la première colonne qui ne possède pas de contrainte d'égalité sont utilisées pour limiter la partie parcourue de l'index. Les contraintes sur les colonnes à droite de ces colonnes sont vérifiées dans l'index, et limitent ainsi les visites de la table, mais elles ne réduisent pas la partie de l'index à parcourir.

Par exemple, avec un index sur (a, b, c) et une condition de requête WHERE a = 5 AND b >= 42 AND c < 77, l'index est parcouru à partir de la première entrée pour laquelle a = 5 et b = 42 jusqu'à la dernière entrée pour laquelle a = 5. Les entrées de l'index avec c >= 77 sont sautées, mais elles sont toujours parcourues. En principe, cet index peut être utilisé pour les requêtes qui ont des contraintes sur b et/ou c sans contrainte sur a -- mais l'index entier doit être parcouru, donc, dans la plupart des cas, le planificateur préfère un parcours séquentiel de la table à l'utilisation de l'index.

Un index GiST multicolonne peut être utilisé avec des conditions de requête qui impliquent un sous-ensemble quelconque de colonnes de l'index. Les conditions sur des colonnes supplémentaires restreignent les entrées renvoyées par l'index, mais la condition sur la première colonne est la plus importante pour déterminer la part de l'index parcourue. Un index GiST est relativement inefficace si sa première colonne n'a que quelques valeurs distinctes, même s'il y a beaucoup de valeurs distinctes dans les colonnes supplémentaires.

Un index multicolonne GIN peut être utilisé avec des conditions de requête qui impliquent tout sous-ensemble des colonnes de l'index. Contrairement à B-tree ou GiST, la qualité de la recherche dans l'index est identique quelles que soient les colonnes de l'index que la requête utilise.

Un index BRIN multicolonne peut être utilisé avec des conditions dans la requête qui impliquent tout sous-ensemble de colonnes dans l'index. Comme GIN et contrairement à B-tree ou GiST, l'efficacité de la recherche par l'index est la même quelles que soient les colonnes utilisées dans les conditions de la requête. La seule raison d'avoir plusieurs index BRIN au lieu d'un index BRIN multicolonne sur une table est d'avoir un paramétrage de stockage pages_per_range différent.

Chaque colonne doit évidemment être utilisée avec des opérateurs appropriés au type de l'index ; les clauses qui impliquent d'autres opérateurs ne sont pas prises en compte.

Il est préférable d'utiliser les index multicolonnes avec parcimonie. Dans la plupart des cas, un index sur une seule colonne est suffisant et préserve espace et temps. Les index de plus de trois colonnes risquent fort d'être inefficaces, sauf si l'utilisation de cette table est extrêmement stylisée. Voir aussi la Section 11.5 et Section 11.9 pour les discussions sur les mérites des différentes configurations d'index.

11.4. Index et ORDER BY

Au-delà du simple fait de trouver les lignes à renvoyer à une requête, un index peut les renvoyer dans un ordre spécifique. Cela permet de résoudre une clause ORDER BY sans étape de tri séparée. De tous les types d'index actuellement supportés par PostgreSQL, seuls les B-tree peuvent produire une sortie triée -- les autres types d'index renvoient les lignes correspondantes dans un ordre imprécis, dépendant de l'implantation.

Le planificateur répond à une clause ORDER BY soit en parcourant un index disponible qui correspond à la clause, soit en parcourant la table dans l'ordre physique et en réalisant un tri explicite. Pour une requête qui nécessite de parcourir une fraction importante de la table, le tri explicite est probablement plus rapide que le parcours d'un index, car il nécessite moins d'entrées/sorties disque, du fait de son accès séquentiel. Les index sont plus utiles lorsqu'il s'agit de ne récupérer que quelques lignes. ORDER BY combiné à LIMIT *n* est un cas spécial très important : un tri explicite doit traiter toutes les données pour identifier les *n* premières lignes, mais s'il y a un index qui correspond à l'ORDER BY, alors les *n* premières lignes peuvent être récupérées directement sans qu'il soit nécessaire de parcourir les autres.

Par défaut, les index B-tree stockent leurs entrées dans l'ordre ascendant, valeurs NULL en dernier (le TID de la table est traité comme une colonne de départage pour les entrées qui, sans quoi, seraient identiques). Cela signifie que le parcours avant d'un index sur une colonne `x` produit une sortie satisfaisant `ORDER BY x` (ou en plus verbeux `ORDER BY x ASC NULLS LAST`). L'index peut aussi être parcouru en arrière, produisant ainsi une sortie satisfaisant un `ORDER BY x DESC` (ou en plus verbeux `ORDER BY x DESC NULLS FIRST`, car `NULLS FIRST` est la valeur par défaut pour un `ORDER BY DESC`).

L'ordre d'un index B-tree peut être défini à la création par l'inclusion des options `ASC`, `DESC`, `NULLS FIRST`, et/ou `NULLS LAST` lors de la création de l'index ; par exemple :

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

Un index stocké en ordre ascendant avec les valeurs NULL en premier peut satisfaire soit `ORDER BY x ASC NULLS FIRST`, soit `ORDER BY x DESC NULLS LAST` selon la direction du parcours.

On peut s'interroger sur l'intérêt de proposer quatre options, alors que deux options associées à la possibilité d'un parcours inverse semblent suffire à couvrir toutes les variantes d'`ORDER BY`. Dans les index monocolonnes, les options sont en effet redondantes, mais pour un index à plusieurs colonnes, elles sont utiles. Si l'on considère un index à deux colonnes `(x, y)`, il peut satisfaire une clause `ORDER BY x, y` sur un parcours avant, ou `ORDER BY x DESC, y DESC` sur un parcours inverse. Mais il se peut que l'application utilise fréquemment `ORDER BY x ASC, y DESC`. Il n'y a pas moyen d'obtenir cet ordre à partir d'un index plus simple, mais c'est possible si l'index est défini comme `(x ASC, y DESC)` or `(x DESC, y ASC)`.

Les index d'ordre différent de celui par défaut sont visiblement une fonctionnalité très spécialisée, mais ils peuvent parfois être à l'origine d'accélérations spectaculaires des performances sur certaines requêtes. L'intérêt de maintenir un tel index dépend de la fréquence des requêtes qui nécessitent un tri particulier.

11.5. Combiner des index multiples

Un parcours unique d'index ne peut utiliser que les clauses de la requête qui utilisent les colonnes de l'index avec les opérateurs de sa classe d'opérateurs et qui sont jointes avec `AND`. Par exemple, étant donné un index sur `(a, b)`, une condition de requête `WHERE a = 5 AND b = 6` peut utiliser l'index, mais une requête `WHERE a = 5 OR b = 6` ne peut pas l'utiliser directement.

Heureusement, PostgreSQL peut combiner plusieurs index (y compris plusieurs utilisations du même index) pour gérer les cas qui ne peuvent pas être résolus par des parcours d'index simples. Le système peut former des conditions `AND` et `OR` sur plusieurs parcours d'index. Par exemple, une requête comme `WHERE x = 42 OR x = 47 OR x = 53 OR x = 99` peut être divisée en quatre parcours distincts d'un index sur `x`, chaque parcours utilisant une des clauses de la requête. Les résultats de ces parcours sont alors assemblés par OR pour produire le résultat. Autre exemple, s'il existe des index séparés sur `x` et `y`, une résolution possible d'une requête comme `WHERE x = 5 AND y = 6` consiste à utiliser chaque index avec la clause de la requête appropriée et d'assembler les différents résultats avec un AND pour identifier les lignes résultantes.

Pour combiner plusieurs index, le système parcourt chaque index nécessaire et prépare un *bitmap* en mémoire qui donne l'emplacement des lignes de table qui correspondent aux conditions de l'index. Les bitmaps sont ensuite assemblés avec des opérateurs AND ou OR selon les besoins de la requête. Enfin, les lignes réelles de la table sont visitées et renvoyées. Elles sont visitées dans l'ordre physique parce c'est ainsi que le bitmap est créé ; cela signifie que l'ordre des index originaux est perdu et que, du coup, une étape de tri séparée est nécessaire si la requête comprend une clause `ORDER BY`. Pour cette raison, et parce que chaque parcours d'index supplémentaire ajoute un temps additionnel, le planificateur choisit quelques fois d'utiliser un parcours d'index simple même si des index supplémentaires sont disponibles et peuvent être utilisés.

Le nombre de combinaisons d'index possibles croît parallèlement à la complexité des applications. Il est alors de la responsabilité du développeur de la base de décider des index à fournir. Il est quelques fois préférable de créer des index multicolonnes, mais il est parfois préférable de créer des index séparés et de s'appuyer sur la fonctionnalité de combinaison des index.

Par exemple, si la charge inclut un mélange de requêtes qui impliquent parfois uniquement la colonne `x`, parfois uniquement la colonne `y` et quelques fois les deux colonnes, on peut choisir deux index séparés sur `x` et `y` et s'appuyer sur la combinaison d'index pour traiter les requêtes qui utilisent les deux colonnes. On peut aussi créer un index multicolonne sur `(x, y)`. Cet index est typiquement plus efficace que la combinaison d'index pour les requêtes impliquant les deux colonnes, mais, comme discuté dans la Section 11.3, il est pratiquement inutile pour les requêtes n'impliquant que `y`. Il ne peut donc pas être le seul index. Une combinaison de l'index multicolonne et d'un index séparé sur `y` est une solution raisonnable. Pour les requêtes qui n'impliquent que `x`, l'index multicolonne peut être utilisé, bien qu'il soit plus large et donc plus lent qu'un index sur `x` seul. La dernière alternative consiste à créer les trois index, mais cette solution n'est raisonnable que si la table est lue bien plus fréquemment qu'elle n'est mise à jour et que les trois types de requêtes sont communs. Si un des types de requêtes est bien moins courant que les autres, il est préférable de ne créer que les deux index qui correspondent le mieux aux types communs.

11.6. Index d'unicité

Les index peuvent aussi être utilisés pour garantir l'unicité des valeurs d'une colonne, ou l'unicité des valeurs combinées de plusieurs colonnes.

```
CREATE UNIQUE INDEX nom ON table (colonne [, ...]) [ NULLS
 [ NOT ]optional> DISTINCT ];
```

À ce jour, seuls les index B-tree peuvent être déclarés uniques.

Lorsqu'un index est déclaré unique, il ne peut exister plusieurs lignes d'une table qui possèdent la même valeur indexée. Par défaut, les valeurs NULL dans une colonne d'unicité ne sont pas considérées comme égales, permettant ainsi plusieurs valeurs NULL dans la colonne. L'option `NULLS NOT DISTINCT` modifie ceci et fait que l'index traite les valeurs NULL comme identiques. Un index d'unicité multicolonne ne rejette que les cas où toutes les colonnes indexées sont égales sur plusieurs lignes.

PostgreSQL crée automatiquement un index d'unicité à la déclaration d'une contrainte d'unicité ou d'une clé primaire sur une table. L'index porte sur les colonnes qui composent la clé primaire ou la contrainte d'unicité (au besoin, il s'agit d'un index multicolonne). C'est cet index qui assure le mécanisme de vérification de la contrainte.

> **Note**
>
> Il n'est pas nécessaire de créer manuellement un index sur les colonnes uniques. Cela duplique l'index créé automatiquement.

11.7. Index d'expressions

Une colonne d'index ne correspond pas nécessairement exactement à une colonne de la table associée, mais peut être une fonction ou une expression scalaire calculée à partir d'une ou plusieurs colonnes de la table. Cette fonctionnalité est utile pour obtenir un accès rapide aux tables en utilisant les résultats de calculs.

Par exemple, une façon classique de faire des comparaisons indépendantes de la casse est d'utiliser la fonction `lower` :

```
SELECT * FROM test1 WHERE lower(col1) = 'valeur';
```

Si un index a été défini sur le résultat de `lower(col1)`, cette requête peut l'utiliser. Un tel index est créé avec la commande :

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

Si l'index est déclaré `UNIQUE`, il empêche la création de lignes dont les valeurs de la colonne `col1` ne diffèrent que par la casse, ainsi que celle de lignes dont les valeurs de la colonne `col1` sont identiques. Ainsi, les index d'expressions peuvent être utilisés pour appliquer des contraintes qui ne peuvent être définies avec une simple contrainte d'unicité.

Autre exemple. Lorsque des requêtes comme :

```
SELECT * FROM personnes WHERE (prenom || ' ' || nom) = 'Jean
 Dupont';
```

sont fréquentes, alors il peut être utile de créer un index comme :

```
CREATE INDEX personnes_noms ON personnes ((prenom || ' ' || nom));
```

La syntaxe de la commande `CREATE INDEX` nécessite normalement de mettre des parenthèses autour de l'expression indexée, comme dans l'exemple précédent. Les parenthèses peuvent être omises quand l'expression est un simple appel de fonction, comme dans le premier exemple.

Les expressions d'index sont relativement coûteuses à calculer, car l'expression doit être recalculée à chaque insertion ou mise à jour non-HOT de ligne. Néanmoins, les expressions d'index ne sont *pas* recalculées lors d'une recherche par index, car elles sont déjà stockées dans l'index. Dans les deux exemples ci-dessus, le système voit la requête comme un `WHERE colonne_indexée = 'constante'`. De ce fait, la recherche est aussi rapide que toute autre requête d'index. Ainsi, les index d'expressions sont utiles lorsque la rapidité de recherche est plus importante que la rapidité d'insertion et de mise à jour.

11.8. Index partiels

Un *index partiel* est un index construit sur un sous-ensemble d'une table ; le sous-ensemble est défini par une expression conditionnelle (appelée *prédicat* de l'index partiel). L'index ne contient des entrées que pour les lignes de la table qui satisfont au prédicat. Les index partiels sont une fonctionnalité spécialisée, mais ils trouvent leur utilité dans de nombreuses situations.

Une raison majeure à l'utilisation d'index partiels est d'éviter d'indexer les valeurs courantes. Puisqu'une requête qui recherche une valeur courante (qui correspond à plus de quelques pour cents de toutes les lignes) n'utilise, de toute façon, pas cet index, il ne sert à rien de garder ces lignes dans l'index. Cela réduit la taille de l'index, ce qui accélérera les requêtes qui l'utilisent. Cela accélère aussi nombre d'opérations de mise à jour de la table, car l'index n'a pas à être mis à jour à chaque fois. L'Exemple 11.1 montre une application possible de cette idée.

Exemple 11.1. Mettre en place un index partiel pour exclure des valeurs courantes

Soit l'enregistrement d'un journal d'accès à un serveur web dans une base de données. La plupart des accès proviennent de classes d'adresses IP internes à l'organisation, mais certaines proviennent de l'extérieur (des employés connectés par modem, par exemple). Si les recherches par adresses IP concernent essentiellement les accès extérieurs, il est inutile d'indexer les classes d'adresses IP qui correspondent au sous-réseau de l'organisation.

Si la table ressemble à :

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
```

```
    ...
);
```

Pour créer un index partiel qui corresponde à l'exemple, il faut utiliser une commande comme celle-ci :

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND
               client_ip < inet '192.168.100.255');
```

Une requête typique qui peut utiliser cet index est :

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

Ici l'adresse IP de la requête est couverte par un index partiel. La requête suivante ne peut pas utiliser l'index partiel, puisqu'elle utilise une adresse IP qui est exclue de l'index :

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '192.168.100.23';
```

Ce type d'index partiel nécessite que les valeurs courantes soient prédéterminées, de façon à ce que ce type d'index soit mieux utilisé avec une distribution des données qui ne change pas. Les index peuvent être recréés occasionnellement pour s'adapter aux nouvelles distributions de données, mais cela ajoute de la maintenance.

Une autre utilisation possible d'index partiel revient à exclure des valeurs de l'index qui ne correspondent pas aux requêtes courantes ; ceci est montré dans l'Exemple 11.2. Cette méthode donne les mêmes avantages que la précédente, mais empêche l'accès par l'index aux valeurs « sans intérêt ». Évidemment, mettre en place des index partiels pour ce genre de scénarios nécessite beaucoup de soin et d'expérimentation.

Exemple 11.2. Mettre en place un index partiel pour exclure les valeurs inintéressantes

Soit une table qui contient des commandes facturées et des commandes non facturées, avec les commandes non facturées qui ne prennent qu'une petite fraction de l'espace dans la table, et qui sont les plus accédées. Il est possible d'améliorer les performances en créant un index limité aux lignes non facturées. La commande pour créer l'index ressemble à :

```
CREATE INDEX index_commandes_nonfacturees ON commandes
 (no_commande)
    WHERE facturee is not true;
```

La requête suivante utilise cet index :

```
SELECT * FROM commandes WHERE facturee is not true AND no_commande
 < 10000;
```

Néanmoins, l'index peut aussi être utilisé dans des requêtes qui n'utilisent pas `no_commande`, comme :

```
SELECT * FROM commandes WHERE facturee is not true AND montant >
 5000.00;
```

Ceci n'est pas aussi efficace qu'un index partiel sur la colonne `montant`, car le système doit lire l'index en entier. Néanmoins, s'il y a assez peu de commandes non facturées, l'utilisation de cet index partiel pour trouver les commandes non facturées peut être plus efficace.

La requête suivante ne peut pas utiliser cet index :

```
SELECT * FROM commandes WHERE no_commande = 3501;
```

La commande 3501 peut faire partie des commandes facturées ou non facturées.

L'Exemple 11.2 illustre aussi le fait que la colonne indexée et la colonne utilisée dans le prédicat ne sont pas nécessairement les mêmes. PostgreSQL supporte tous les prédicats sur les index partiels, tant que ceux-ci ne portent que sur des champs de la table indexée. Néanmoins, il faut se rappeler que le prédicat doit correspondre aux conditions utilisées dans les requêtes qui sont supposées profiter de l'index. Pour être précis, un index partiel ne peut être utilisé pour une requête que si le système peut reconnaître que la clause `WHERE` de la requête implique mathématiquement le prédicat de l'index. PostgreSQL n'a pas de méthode sophistiquée de démonstration de théorème pour reconnaître que des expressions apparemment différentes sont mathématiquement équivalentes. (Non seulement une telle méthode générale de démonstration serait extrêmement complexe à créer, mais, en plus, elle serait probablement trop lente pour être d'une quelconque utilité). Le système peut reconnaître des implications d'inégalités simples, par exemple « x < 1 » implique « x < 2 » ; dans les autres cas, la condition du prédicat doit correspondre exactement à une partie de la clause `WHERE` de la requête, sans quoi l'index ne peut pas être considéré comme utilisable. La correspondance prend place lors de l'exécution de la planification de la requête, pas lors de l'exécution. À ce titre, les clauses de requêtes à paramètres ne fonctionnent pas avec un index partiel. Par exemple, une requête préparée avec un paramètre peut indiquer « x < ? » qui n'implique jamais « x < 2 » pour toutes les valeurs possibles du paramètre.

Un troisième usage possible des index partiels ne nécessite pas que l'index soit utilisé dans des requêtes. L'idée ici est de créer un index d'unicité sur un sous-ensemble de la table, comme dans l'Exemple 11.3. Cela permet de mettre en place une unicité parmi le sous-ensemble des lignes de la table qui satisfont au prédicat, sans contraindre les lignes qui n'y satisfont pas.

Exemple 11.3. Mettre en place un index d'unicité partielle

Soit une table qui décrit des résultats de tests. On souhaite s'assurer qu'il n'y a qu'une seule entrée « succès » (succes) pour chaque combinaison de sujet et de résultat, alors qu'il peut y avoir un nombre quelconque d'entrées « echec ». Une façon de procéder :

```
CREATE TABLE tests (
    sujet text,
    resultat text,
    succes boolean,
    ...
);

CREATE UNIQUE INDEX contrainte_tests_reussis ON tests (sujet,
 resultat)
    WHERE succes;
```

C'est une méthode très efficace quand il y a peu de tests réussis et beaucoup de tests en échec. Il est aussi possible de permettre une seule valeur NULL dans une colonne en créant un index partiel unique avec une restriction `IS NULL`.

Enfin, un index partiel peut aussi être utilisé pour surcharger les choix de plan d'exécution de requêtes du système. De plus, des jeux de données à distribution particulière peuvent inciter le système à utiliser un index alors qu'il ne le devrait pas. Dans ce cas, on peut mettre en place l'index de telle façon qu'il ne soit pas utilisé pour la requête qui pose problème. Normalement, PostgreSQL fait des choix d'usage d'index raisonnables. Par exemple, il les évite pour rechercher les valeurs communes, si bien que l'exemple précédent n'économise que la taille de l'index, il n'est pas nécessaire pour éviter l'utilisation de l'index. En fait, les choix de plan d'exécution incorrects doivent être traités comme des bogues, et être transmis à l'équipe de développement.

Mettre en place un index partiel indique une connaissance au moins aussi étendue que celle de l'analyseur de requêtes, en particulier, savoir quand un index peut être profitable. Une telle

connaissance nécessite de l'expérience et une bonne compréhension du fonctionnement des index de PostgreSQL. Dans la plupart des cas, les index partiels ne représentent pas un gros gain par rapport aux index classiques. Dans certains cas, ils sont même contre-productifs, comme dans Exemple 11.4.

Exemple 11.4. Ne pas utiliser les index partiels comme substitut au partitionnement

Vous pourriez être tenté de créer un gros ensemble d'index partiels qui ne se recouvrent pas, par exemple :

```
CREATE INDEX mytable_cat_1 ON mytable (data) WHERE category = 1;
CREATE INDEX mytable_cat_2 ON mytable (data) WHERE category = 2;
CREATE INDEX mytable_cat_3 ON mytable (data) WHERE category = 3;
...
CREATE INDEX mytable_cat_N ON mytable (data) WHERE category = N;
```

C'est une mauvaise idée ! Pratiquement à coup sûr, vous seriez mieux avec un seul index complet, déclaré ainsi :

```
CREATE INDEX mytable_cat_data ON mytable (category, data);
```

(Placez la colonne category en premier, pour les raisons décrites dans Section 11.3.) Bien qu'une recherche dans cet index plus gros pourrait avoir à descendre quelques niveaux de plus dans l'arbre que ce que ferait une recherche dans un index plus petit, cela sera certainement moins cher que ce que va coûter l'effort du planificateur pour sélectionner le bon index parmi tous les index. Le cœur du problème est que le système ne comprend pas la relation entre les index partiels et va laborieusement tester chaque index pour vois s'il est applicable à la requête courante.

Si votre table est suffisamment volumineuse pour qu'un index seul soit réellement une mauvaise idée, vous devriez plutôt regarder du côté du partitionnement (voir Section 5.11). Avec ce mécanisme, le système comprend que les tables et les index ne se croisent pas, et donc de meilleures performances sont possibles.

Plus d'informations sur les index partiels sont disponibles dans [ston89b], [olson93] et [seshadri95].

11.9. Parcours d'index seul et index couvrants

Tous les index dans PostgreSQL sont des index *secondaires*, ceci signifiant que chaque index est stocké séparément des données de la table (ce qui est appelé le *heap* dans la terminologie PostgreSQL). Ceci signifie que, dans un parcours d'index habituel, chaque récupération de ligne nécessite de récupérer les données de l'index et du heap. De plus, bien que les entrées d'un index correspondant à une condition `WHERE` indexable soient habituellement proches dans l'index, les lignes de la table qu'elles référencent peuvent se trouver n'importe où dans le heap. La portion accédée du heap pendant un parcours d'index implique donc beaucoup d'accès aléatoires au heap, ce qui peut être lent, tout particulièrement sur les disques magnétiques traditionnels. (Comme décrit dans Section 11.5, les parcours de bitmap essaient de diminuer ce coût en réalisant les accès au heap de façon ordonnée, mais cette méthode a ses limites.)

Pour résoudre ce problème de performances, PostgreSQL supporte les *parcours d'index seul*, qui peuvent répondre aux requêtes à partir d'un index seul sans aucun accès au heap. L'idée de base est de renvoyer les valeurs directement à partir de chaque entrée dans l'index au lieu de consulter l'entrée associée dans le heap. Il existe deux restrictions fondamentales pour l'utilisation de cette méthode :

1. Le type d'index doit supporter les parcours d'index seul. Les index B-tree peuvent toujours le faire. Les index GiST et SP-GiST supportent les parcours d'index seuls uniquement pour certaines classes d'opérateurs, mais pas pour les autres. D'autres types d'index n'ont aucun support. Le prérequis sous-jacent est que l'index doit enregistrer physiquement, ou être capable de reconstruire, les données originales pour chaque entrée d'index. En contre-exemple, les index GIN ne supportent pas les parcours d'index seul, car chaque entrée d'index contient typiquement seulement une partie de la valeur originale.

2. La requête doit référencer seulement les colonnes enregistrées dans l'index. Par exemple, avec un index sur les colonnes `x` et `y` d'une table qui a aussi une colonne `z`, ces requêtes peuvent utiliser des parcours d'index seul :

   ```
   SELECT x, y FROM tab WHERE x = 'key';
   SELECT x FROM tab WHERE x = 'key' AND y < 42;
   ```

   alors que ces requêtes ne le peuvent pas :

   ```
   SELECT x, z FROM tab WHERE x = 'key';
   SELECT x FROM tab WHERE x = 'key' AND z < 42;
   ```

   (Les index fonctionnels et les index partiels compliquent cette règle, comme expliquée ci-dessous.)

Si ces deux prérequis fondamentaux sont rencontrés, alors toutes les valeurs requises par la requête sont disponibles dans l'index, donc un parcours d'index seul est physiquement possible. Mais il existe un prérequis supplémentaire pour tout parcours de table dans PostgreSQL : il doit vérifier que chaque ligne récupérée est « visible » dans le cadre du snapshot MVCC de la requête, comme indiqué dans Chapitre 13. Les informations de visibilité ne sont pas enregistrées dans les entrées de l'index, uniquement dans les entrées de la table. Donc a priori, cela voudrait dire que chaque récupération de ligne nécessite un accès au heap de la table. Et c'est bien le cas si la ligne de la table a été modifiée récemment. Néanmoins, pour les données changeant peu, il y a toujours un moyen de contourner ce problème. PostgreSQL trace pour chaque page dans le heap de la table, si toutes les lignes enregistrées dans cette page sont suffisamment anciennes pour être visibles par toutes les transactions en cours et futures. Cette information est enregistrée dans un bit de la *carte de visibilité* de la table. Un parcours d'index seul, pour trouver une entrée d'index candidate, vérifie le bit de la carte de visibilité pour la page correspondante du heap. Si ce bit est vrai, la ligne est connue comme étant visible et donc la donnée peut être renvoyée sans plus de tests. Dans le cas contraire, l'entrée heap doit être visitée pour trouver si elle est visible, donc aucune amélioration des performances n'est obtenue par rapport à un parcours d'index standard. Même dans le cas d'une réussite, cette approche remplace des accès au heap par des accès à la carte de visibilité. Comme la carte de visibilité est quatre fois plus petite que le heap qu'elle décrit, moins d'accès IO sont nécessaires pour accéder à l'information. Dans la plupart des cas, la carte de visibilité reste en mémoire tout le temps.

En bref, quand un parcours d'index seul est possible d'après les deux prérequis fondamentaux, son utilisation ne sera réellement intéressante que si une faction significative des blocs du heap de la table ont leur bit all-visible configuré. Mais les tables dont une large fraction des lignes ne changent pas sont habituellement suffisantes pour que ce type de parcours se révèle très utile en pratique.

Pour une utilisation efficace de la fonctionnalité du parcours d'index seul, vous pourriez choisir de créer un *index couvrant*, qui est un index conçu spécifiquement pour inclure les colonnes nécessaires pour un type particulier de requêtes que vous exécutez fréquemment. Comme les requêtes ont typiquement besoin de récupérer plus de colonnes que de colonnes incluses dans la recherche, PostgreSQL vous permet de créer un index pour lequel certaines colonnes ne sont qu'une « charge » et ne peuvent pas faire partie de la recherche. Ceci se fait en ajoutant la clause `INCLUDE` avec la liste des colonnes supplémentaires. Par exemple, si vous exécutez fréquemment des requêtes comme :

```
SELECT y FROM tab WHERE x = 'key';
```

l'approche habituelle pour accélérer de telles requêtes est de créer un index uniquement sur `x`. Néanmoins, un index défini comme

```
CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

peut gérer ces requêtes sous la forme de parcours d'index seul, car les valeurs de `y` peuvent être obtenues de l'index sans visiter la table.

Comme la colonne `y` ne fait pas partie des clés de recherche de l'index, elle n'a pas besoin d'être d'un type de donnée que l'index peut gérer ; la valeur est simplement enregistrée dans l'index et n'est pas interprétée par la machinerie de l'index. De plus, si l'index est un index unique, autrement dit

```
CREATE UNIQUE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

la condition d'unicité s'applique uniquement à la colonne `x`, et non pas à la combinaison `x` et `y`. (Une clause `INCLUDE` peut aussi être écrite dans les contraintes `UNIQUE` et `PRIMARY KEY`, fournissant une syntaxe alternative pour configurer ce type d'index.)

Il est conseillé d'être conservateur sur l'ajout de colonnes non-clés dans un index, tout spécialement les colonnes volumineuses. Si un enregistrement d'index dépasse la taille maximale autorisée pour ce type d'index, l'insertion de données échouera. Dans tous les cas, les colonnes non-clés dupliquent les données de la table et augmentent la taille de l'index, ce qui peut ralentir les recherches. Et rappelez-vous qu'il y a peu d'intérêt d'inclure des colonnes non-clés dans un index, sauf si la table change très doucement pour qu'un parcours d'index seul n'ait pas besoin d'accéder à la table. Si la ligne de la table doit être visitée, cela ne coûte rien de récupérer la valeur de la colonne dans la table. Les autres restrictions sont que les expressions ne sont actuellement pas supportées dans les colonnes incluses, et que seuls les index B-tree, GiST et SP-GiST supportent actuellement les colonnes incluses.

Avant que PostgreSQL ne dispose de la fonctionnalité `INCLUDE`, les utilisateurs créaient parfois des index couvrants en ajoutant les colonnes non-clés comme des colonnes d'index habituels, par exemple

```
CREATE INDEX tab_x_y ON tab(x, y);
```

même s'ils n'avaient jamais l'intention d'utiliser `y` comme partie de la clause `WHERE`. Ceci fonctionne bien tant que les colonnes supplémentaires sont les dernières colonnes ; il est déconseillé de les ajouter comme premières colonnes pour les raisons expliquées dans Section 11.3. Néanmoins, cette méthode ne supporte pas le cas où vous voulez que l'index assure l'unicité des colonnes clés.

La *troncature de suffixe* supprime toujours les colonnes non-clé des niveaux supérieurs du B-Tree. En tant que colonnes de charge utile, elles ne sont jamais utilisées pour guider des parcours d'index. Le processus de troncature supprime également une ou plusieurs colonne(s) clé quand le reste du préfixe de colonnes(s) se montre suffisant pour décrire les tuples du plus bas niveau de B-Tree. Dans la pratique, les index couvrants sans clause `INCLUDE` évitent souvent de stocker les colonnes qui sont de la charge utile effective dans les niveaux supérieurs. Cependant, définir explicitement les colonnes de charge utile comme colonnes non-clé permet de conserver des tuples de petite taille dans les niveaux supérieurs *de manière fiable*.

En principe, les parcours d'index seul peuvent être utilisés avec des index fonctionnels. Par exemple, avec un index sur `f(x)` où `x` est une colonne de table, il est possible de l'utiliser avec la requête suivante :

```
SELECT f(x) FROM tab WHERE f(x) < 1;
```

pour un parcours d'index seul ; et c'est très intéressant si `f()` est une fonction coûteuse à l'exécution. Néanmoins, l'optimiseur de PostgreSQL n'est pas très intelligent actuellement avec de tels cas. Il considère qu'une requête est réalisable avec un parcours d'index seul uniquement quand toutes les *colonnes* nécessaires pour la requête sont disponibles à partir de l'index. Dans cet exemple, `x` n'est pas nécessaire, sauf dans le contexte `f(x)`, mais le planificateur ne le remarque pas et conclut qu'un parcours d'index seul n'est pas possible. Si un parcours d'index seul semble suffisamment intéressant, ceci peut être contourné en ajoutant `x` comme colonne incluse, par exemple

```
CREATE INDEX tab_f_x ON tab (f(x)) INCLUDE (x);
```

Si le but est d'éviter de recalculer `f(x)`, une autre astuce est que l'optimiseur ne fera pas forcément une correspondance entre les utilisations de `f(x)` qui ne sont pas dans les clauses `WHERE` indexables et la colonne de l'index. Généralement, le test sera efficace pour les requêtes simples comme indiquées ci-dessus, mais pas pour les requêtes qui impliquent des jointures. Ces déficiences pourraient être corrigées dans les versions futures de PostgreSQL.

Les index partiels ont aussi des interactions intéressantes avec les parcours d'index seul. Considérez l'index partiel indiqué dans Exemple 11.3 :

```
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject,
 target)
    WHERE success;
```

En principe, nous pouvons faire un parcours d'index seul sur cet index pour satisfaire une requête du type :

```
SELECT target FROM tests WHERE subject = 'some-subject' AND
 success;
```

Mais il reste un problème : la clause `WHERE` fait référence à `success` qui n'est pas disponible comme colonne de résultat de l'index. Néanmoins, un parcours d'index seul est possible parce que le plan n'a pas besoin de vérifier de nouveau cette partie de la clause `WHERE` à l'exécution : toutes les entrées trouvées dans l'index ont obligatoirement `success = true`, donc il n'est pas nécessaire de le vérifier explicitement dans le plan. Les versions 9.6 et ultérieures de PostgreSQL reconnaîtront de tels cas et permettront aux parcours d'index seul d'être générés, mais les anciennes versions ne le pourront pas.

11.10. Classes et familles d'opérateurs

Une définition d'index peut indiquer une *classe d'opérateurs* pour chaque colonne de l'index.

```
CREATE INDEX nom ON table (colonne classe_operateur [ ( options
 classe opérateur ) ] [options de tri] [, ...]);
```

La classe d'opérateurs identifie les opérateurs que l'index doit utiliser sur cette colonne. Par exemple, un index B-tree sur une colonne de type `int4` utilise la classe `int4_ops`. Cette classe d'opérateurs comprend des fonctions de comparaison pour les valeurs de type `int4`. En pratique, la classe d'opérateurs par défaut pour le type de données de la colonne est généralement suffisante. Les classes d'opérateurs sont utiles pour certains types de données, pour lesquels il peut y avoir plus d'un comportement utile de l'index. Par exemple, une donnée de type nombre complexe peut être classée par sa valeur absolue, ou par sa partie entière. Cela peut s'obtenir en définissant deux classes

d'opérateurs pour ce type de données et en sélectionnant la bonne classe à la création de l'index. La classe d'opérateurs détermine l'ordre de tri basique (qui peut ensuite être modifié en ajoutant des options de tri comme COLLATE, ASC/DESC et/ou NULLS FIRST/NULLS LAST).

Il y a quelques classes d'opérateurs en plus des classes par défaut :

- Les classes d'opérateurs text_pattern_ops, varchar_pattern_ops et bpchar_pattern_ops supportent les index B-tree sur les types text, varchar et char, respectivement. À la différence des classes d'opérateurs par défaut, les valeurs sont comparées strictement caractère par caractère plutôt que suivant les règles de tri spécifiques à la localisation. Cela rend ces index utilisables pour des requêtes qui effectuent des recherches sur des motifs (LIKE ou des expressions régulières POSIX) quand la base de données n'utilise pas la locale standard « C ». Par exemple, on peut indexer une colonne varchar comme ceci :

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  Il faut créer un index avec la classe d'opérateurs par défaut pour que les requêtes qui utilisent une comparaison <, <=, > ou >= ordinaire utilisent un index. De telles requêtes ne peuvent pas utiliser les classes d'opérateurs *xxx*_pattern_ops. Néanmoins, des comparaisons d'égalité ordinaires peuvent utiliser ces classes d'opérateurs. Il est possible de créer plusieurs index sur la même colonne avec différentes classes d'opérateurs. Si la locale C est utilisée, les classes d'opérateurs *xxx*_pattern_ops ne sont pas nécessaires, car un index avec une classe d'opérateurs par défaut est utilisable pour les requêtes de correspondance de modèles dans la locale C.

Les requêtes suivantes montrent les classes d'opérateurs prédéfinies :

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;
```

Une classe d'opérateurs n'est qu'un sous-ensemble d'une structure plus large appelée *famille d'opérateurs*. Dans les cas où plusieurs types de données ont des comportements similaires, il est fréquemment utile de définir des opérateurs identiques pour plusieurs types de données et d'autoriser leur utilisation avec des index. Pour cela, les classes d'opérateurs de chacun de ces types doivent être groupés dans la même famille d'opérateurs. Les opérateurs intertypes sont membres de la famille, mais ne sont pas associés avec une seule classe de la famille.

Cette version étendue de la requête précédente montre la famille d'opérateurs à laquelle appartient chaque classe d'opérateurs :

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opf.opfname AS opfamily_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc, pg_opfamily opf
    WHERE opc.opcmethod = am.oid AND
          opc.opcfamily = opf.oid
    ORDER BY index_method, opclass_name;
```

Cette requête affiche toutes les familles d'opérateurs définies et tous les opérateurs inclus dans chaque famille :

```
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid AND
          amop.amopfamily = opf.oid
    ORDER BY index_method, opfamily_name, opfamily_operator;
```

> **Astuce**
>
> psql dispose des méta-commandes `\dAc`, `\dAf` et `\dAo`, qui fournissent des versions un peu plus sophistiquées de ces requêtes.

11.11. Index et collationnements

Un index peut supporter seulement un collationnement par colonne d'index. Si plusieurs collationnements ont un intérêt, plusieurs index pourraient être nécessaires.

Regardez ces requêtes :

```
CREATE TABLE test1c (
    id integer,
    content varchar COLLATE "x"
);

CREATE INDEX test1c_content_index ON test1c (content);
```

L'index utilise automatiquement le collationnement de la colonne sous-jacente. Donc une requête de la forme

```
SELECT * FROM test1c WHERE content > constant;
```

peut utiliser l'index, car la comparaison utilisera par défaut le collationnement de la colonne. Néanmoins, cet index ne peut pas accélérer les requêtes qui impliquent d'autres collationnements. Donc, pour des requêtes de cette forme

```
SELECT * FROM test1c WHERE content > constant COLLATE "y";
```

un index supplémentaire, supportant le collationnement `"y"` peut être ajouté ainsi :

```
CREATE INDEX test1c_content_y_index ON test1c (content COLLATE
 "y");
```

11.12. Examiner l'utilisation des index

Bien que les index de PostgreSQL n'aient pas besoin de maintenance ou d'optimisation, il est important de s'assurer que les index sont effectivement utilisés sur un système en production. On vérifie

l'utilisation d'un index pour une requête particulière avec la commande EXPLAIN. Son utilisation dans notre cas est expliquée dans la Section 14.1. Il est aussi possible de rassembler des statistiques globales sur l'utilisation des index sur un serveur en cours de fonctionnement, comme décrit dans la Section 28.2.

Il est difficile de donner une procédure générale pour déterminer les index à créer. Plusieurs cas typiques ont été cités dans les exemples précédents. Une bonne dose d'expérimentation est souvent nécessaire dans de nombreux cas. Le reste de cette section donne quelques pistes.

- La première chose à faire est de lancer ANALYZE. Cette commande collecte les informations sur la distribution des valeurs dans la table. Cette information est nécessaire pour estimer le nombre de lignes retournées par une requête. L'optimiseur de requêtes en a besoin pour donner des coûts réalistes aux différents plans de requêtes possibles. En l'absence de statistiques réelles, le système utilise quelques valeurs par défaut, qui ont toutes les chances d'être inadaptées. Examiner l'utilisation des index par une application sans avoir lancé `ANALYZE` au préalable est, de ce fait, peine perdue. Voir Section 25.1.3 et Section 25.1.6 pour plus d'informations.

- Utiliser des données réelles pour l'expérimentation. Utiliser des données de test pour mettre en place des index permet de trouver les index utiles pour les données de test, mais c'est tout.

  Il est particulièrement néfaste d'utiliser des jeux de données très réduits. Alors qu'une requête sélectionnant 1000 lignes parmi 100000 peut utiliser un index, il est peu probable qu'une requête sélectionnant 1 ligne dans une table de 100 le fasse, parce que les 100 lignes tiennent probablement dans une seule page sur le disque, et qu'il n'y a aucun plan d'exécution qui puisse aller plus vite que la lecture d'une seule page.

  Être vigilant en créant des données de test. C'est souvent inévitable quand l'application n'est pas encore en production. Des valeurs très similaires, complètement aléatoires, ou insérées déjà triées peuvent modifier la distribution des données et fausser les statistiques.

- Quand les index ne sont pas utilisés, il peut être utile pour les tests de forcer leur utilisation. Certains paramètres d'exécution du serveur peuvent interdire certains types de plans (voir la Section 20.7.1). Par exemple, en interdisant les lectures séquentielles de tables `enable_seqscan`) et les jointures à boucles imbriquées (`enable_nestloop`), qui sont les deux plans les plus basiques, on force le système à utiliser un plan différent. Si le système continue néanmoins à choisir une lecture séquentielle ou une jointure à boucles imbriquées, alors il y a probablement une raison plus fondamentale qui empêche l'utilisation de l'index ; la condition peut, par exemple, ne pas correspondre à l'index. (Les sections précédentes expliquent quelles sortes de requêtes peuvent utiliser quelles sortes d'index.)

- Si l'index est effectivement utilisé en forçant son utilisation, alors il y a deux possibilités : soit le système a raison et l'utilisation de l'index est effectivement inappropriée, soit les coûts estimés des plans de requêtes ne reflètent pas la réalité. Il faut alors comparer la durée de la requête avec et sans index. La commande `EXPLAIN ANALYZE` peut être utile pour cela.

- S'il apparaît que les estimations de coûts sont fausses, il y a de nouveau deux possibilités. Le coût total est calculé à partir du coût par ligne de chaque nœud du plan, multiplié par l'estimation de sélectivité du nœud de plan. Le coût estimé des nœuds de plan peut être ajusté avec des paramètres d'exécution (décrits dans la Section 20.7.2). Une estimation de sélectivité inadaptée est due à des statistiques insuffisantes. Il peut être possible de les améliorer en optimisant les paramètres de collecte de statistiques. Voir ALTER TABLE.

  Si les coûts ne peuvent pas être ajustés à une meilleure représentation de la réalité, alors il faut peut-être forcer l'utilisation de l'index explicitement. Il peut aussi s'avérer utile de contacter les développeurs de PostgreSQL afin qu'ils examinent le problème.

Chapitre 12. Recherche plein texte

12.1. Introduction

La recherche plein texte (ou plus simplement la *recherche de texte*) permet de sélectionner des *documents* en langage naturel qui satisfont une *requête* et, en option, de les trier par intérêt suivant cette requête. Le type le plus fréquent de recherche concerne la récupération de tous les documents contenant les *termes de recherche* indiqués et de les renvoyer dans un ordre dépendant de leur *similarité* par rapport à la requête. Les notions de `requête` et de `similarité` peuvent beaucoup varier et dépendent de l'application réelle. La recherche la plus simple considère une `requête` comme un ensemble de mots et la `similarité` comme la fréquence des mots de la requête dans le document.

Les opérateurs de recherche plein texte existent depuis longtemps dans les bases de données. PostgreSQL dispose des opérateurs `~`, `~*`, `LIKE` et `ILIKE` pour les types de données texte, mais il lui manque un grand nombre de propriétés essentielles requises par les systèmes d'information modernes :

- Aucun support linguistique, même pour l'anglais. Les expressions rationnelles ne sont pas suffisantes, car elles ne peuvent pas gérer facilement les mots dérivés, par exemple `satisfait` et `satisfaire`. Vous pouvez laisser passer des documents qui contiennent `satisfait` bien que vous souhaitiez quand même les trouver avec une recherche sur `satisfaire`. Il est possible d'utiliser `OR` pour rechercher plusieurs formes dérivées, mais cela devient complexe et augmente le risque d'erreur (certains mots peuvent avoir des centaines de variantes).
- Ils ne fournissent aucun classement (score) des résultats de la recherche, ce qui les rend inefficaces quand des centaines de documents correspondants sont trouvés.
- Ils ont tendance à être lents, car les index sont peu supportés, donc ils doivent traiter tous les documents à chaque recherche.

L'indexage pour la recherche plein texte permet au document d'être *prétraité* et qu'un index de ce prétraitement soit sauvegardé pour une recherche ultérieure plus rapide. Le prétraitement inclut :

Analyse des documents en jetons. Il est utile d'identifier les différentes classes de jetons, c'est-à-dire nombres, mots, mots complexes, adresses email, pour qu'ils puissent être traités différemment. En principe, les classes de jetons dépendent de l'application, mais, dans la plupart des cas, utiliser un ensemble prédéfini de classes est adéquat. PostgreSQL utilise un *analyseur* pour réaliser cette étape. Un analyseur standard est fourni, mais des analyseurs personnalisés peuvent être écrits pour des besoins spécifiques.

Conversion des jetons en lexèmes. Un lexème est une chaîne, identique à un jeton, mais elle a été *normalisée* pour que différentes formes du même mot soient découvertes. Par exemple, la normalisation inclut pratiquement toujours le remplacement des majuscules par des minuscules, ainsi que la suppression des suffixes (comme `s` ou `es` en anglais). Ceci permet aux recherches de trouver les variantes du même mot, sans avoir besoin de saisir toutes les variantes possibles. De plus, cette étape élimine typiquement les *termes courants*, qui sont des mots si courants qu'il est inutile de les rechercher. Donc, les jetons sont des fragments bruts du document alors que les lexèmes sont des mots supposés utiles pour l'indexage et la recherche. PostgreSQL utilise des *dictionnaires* pour réaliser cette étape. Différents dictionnaires standards sont fournis et des dictionnaires personnalisés peuvent être créés pour des besoins spécifiques.

Stockage des documents prétraités pour optimiser la recherche . Chaque document peut être représenté comme un tableau trié de lexèmes normalisés. Avec ces lexèmes, il est souvent souhaitable de stocker des informations de position à utiliser pour obtenir un *score de proximité*, pour qu'un document qui contient une région plus « dense » des mots de la requête se voit affecter un score plus important qu'un document qui en a moins.

Les dictionnaires autorisent un contrôle fin de la normalisation des jetons. Avec des dictionnaires appropriés, vous pouvez :

- Définir les termes courants qui ne doivent pas être indexés.
- Établir une liste des synonymes pour un simple mot en utilisant Ispell.
- Établir une correspondance entre des phrases et un simple mot en utilisant un thésaurus.
- Établir une correspondance entre différentes variations d'un mot et une forme canonique en utilisant un dictionnaire Ispell.
- Établir une correspondance entre différentes variations d'un mot et une forme canonique en utilisant les règles du « stemmer » Snowball.

Un type de données `tsvector` est fourni pour stocker les documents prétraités, avec un type `tsquery` pour représenter les requêtes traitées (Section 8.11). Il existe beaucoup de fonctions et d'opérateurs disponibles pour ces types de données (Section 9.13), le plus important étant l'opérateur de correspondance `@@`, dont nous parlons dans la Section 12.1.2. Les recherches plein texte peuvent être accélérées en utilisant des index (Section 12.9).

12.1.1. Qu'est-ce qu'un document ?

Un *document* est l'unité de recherche dans un système de recherche plein texte, par exemple un article de magazine ou un message email. Le moteur de recherche plein texte doit être capable d'analyser des documents et de stocker les associations de lexèmes (mots-clés) avec les documents parents. Ensuite, ces associations seront utilisées pour rechercher les documents contenant des mots de la requête.

Pour les recherches dans PostgreSQL, un document est habituellement un champ texte à l'intérieur d'une ligne d'une table de la base ou une combinaison (concaténation) de champs, parfois stockés dans différentes tables ou obtenus dynamiquement. En d'autres termes, un document peut être construit à partir de différentes parties pour l'indexage et il peut ne pas être stocké quelque part. Par exemple :

```
SELECT titre || ' ' ||  auteur || ' ' ||  resume || ' ' || corps AS
 document
FROM messages
WHERE mid = 12;

SELECT m.titre || ' ' || m.auteur || ' ' || m.resume || ' ' ||
 d.corps AS document
FROM messages m, docs d
WHERE m.mid = did AND mid = 12;
```

> **Note**
>
> En fait, dans ces exemples de requêtes, `coalesce` devrait être utilisée pour empêcher un résultat `NULL` pour le document entier à cause d'une seule colonne `NULL`.

Une autre possibilité est de stocker les documents dans de simples fichiers texte du système de fichiers. Dans ce cas, la base est utilisée pour stocker l'index de recherche plein texte et pour exécuter les recherches, et un identifiant unique est utilisé pour retrouver le document sur le système de fichiers. Néanmoins, retrouver les fichiers en dehors de la base demande les droits d'un superutilisateur ou le support de fonctions spéciales, donc c'est habituellement moins facile que de conserver les données dans PostgreSQL. De plus, tout conserver dans la base permet un accès simple aux métadonnées du document pour aider l'indexage et l'affichage.

Dans le but de la recherche plein texte, chaque document doit être réduit au format de prétraitement, `tsvector`. La recherche et le calcul du score sont réalisés entièrement à partir de la représentation `tsvector` d'un document -- le texte original n'a besoin d'être retrouvé que lorsque le document a été sélectionné pour être montré à l'utilisateur. Nous utilisons souvent `tsvector` pour le document, mais, bien sûr, il ne s'agit que d'une représentation compacte du document complet.

12.1.2. Correspondance de base d'un texte

La recherche plein texte dans PostgreSQL est basée sur l'opérateur de correspondance @@, qui renvoie `true` si un `tsvector` (document) correspond à un `tsquery` (requête). Peu importe le type de données indiqué en premier :

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat
 & rat'::tsquery;
 ?column?
----------
 t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a
 fat rat'::tsvector;
 ?column?
----------
 f
```

Comme le suggère l'exemple ci-dessus, un `tsquery` n'est pas un simple texte brut, pas plus qu'un `tsvector` ne l'est. Un `tsquery` contient des termes de recherche qui doivent déjà être des lexèmes normalisés, et peut combiner plusieurs termes en utilisant les opérateurs AND, OR, NOT et FOLLOWED BY. (Pour les détails sur la syntaxe, voir la Section 8.11.2.) Les fonctions `to_tsquery`, `plainto_tsquery` et `phraseto_tsquery` sont utiles pour convertir un texte écrit par un utilisateur dans un `tsquery` correct, principalement en normalisant les mots apparaissant dans le texte. De façon similaire, `to_tsvector` est utilisée pour analyser et normaliser un document. Donc, en pratique, une correspondance de recherche ressemblerait plutôt à ceci :

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat &
 rat');
 ?column?
----------
 t
```

Observez que cette correspondance ne réussit pas si elle est écrite ainsi :

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat &
 rat');
 ?column?
----------
 f
```

car ici, aucune normalisation du mot `rats` n'interviendra. Les éléments d'un `tsvector` sont des lexèmes, qui sont supposés déjà normalisés, donc `rats` ne correspond pas à `rat`.

L'opérateur @@ supporte aussi une entrée de type `text`, permettant l'oubli de conversions explicites de text vers `tsvector` ou `tsquery` dans les cas simples. Les variantes disponibles sont :

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

Nous avons déjà vu les deux premières. La forme `text @@ tsquery` est équivalente à `to_tsvector(x) @@ y`. La forme `text @@ text` est équivalente à `to_tsvector(x) @@ plainto_tsquery(y)`.

Dans un `tsquery`, l'opérateur & (AND) spécifie que ses deux arguments doivent être présents dans le document pour qu'il y ait correspondance. De même, l'opérateur | (OR) spécifie qu'au moins un de ses arguments doit être présent dans le document, alors que l'opérateur ! (NOT) spécifie que son argument ne doit *pas* être présent pour qu'il y ait une correspondance. Par exemple, la requête `fat & ! rat` correspond aux documents contenant `fat`, mais pas `rat`.

Chercher des phrases est possible à l'aide de l'opérateur <-> (FOLLOWED BY) `tsquery`, qui établit la correspondance seulement si tous ses arguments sont adjacents et dans l'ordre indiqué. Par exemple :

```
SELECT to_tsvector('fatal error') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 t

SELECT to_tsvector('error is not fatal') @@ to_tsquery('fatal <->
 error');
 ?column?
----------
 f
```

Il existe une version plus générale de l'opérateur FOLLOWED BY qui s'écrit <*N*>, où *N* est un entier représentant la différence entre les positions des lexèmes correspondants. L'opérateur <1> est identique à <->, tandis que l'opérateur <2> n'établit la correspondance que si exactement un lexème différent apparaît entre les deux lexèmes en argument, et ainsi de suite. La fonction `phraseto_tsquery` exploite cet opérateur pour construire un `tsquery` permettant de reconnaître une phrase quand certains des mots sont des termes courants. Par exemple :

```
SELECT phraseto_tsquery('cats ate rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <-> 'rat'

SELECT phraseto_tsquery('the cats ate the rats');
       phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <2> 'rat'
```

Un cas particulier potentiellement utile est <0> qui peut être utilisé pour vérifier que deux motifs correspondent à un même mot.

On peut utiliser des parenthèses pour contrôler l'imbrication des opérateurs `tsquery`. En l'absence de parenthèses, l'opérateur | a une priorité moindre que &, puis <->, et finalement !.

Il est important de noter que les opérateurs AND/OR/NOT ont une signification légèrement différente quand ils sont les arguments d'un opérateur FOLLOWED BY que quand ils ne le sont pas. La raison en est que, dans un FOLLOWED BY, la position exacte de la correspondance a une importance. Par exemple, habituellement, `!x` ne fait une correspondance qu'avec les documents qui ne contiennent pas `x` quelque part. Mais `!x <-> y` correspond à `y` s'il n'est pas immédiatement après un `x` ; une occurrence de `x` quelque part dans le document n'empêche pas une correspondance. Un autre exemple est que `x & y` nécessite seulement que `x` et `y` apparaissent quelque part dans le document, mais `(x & y) <-> z` nécessite que `x` et `y` réalisent une correspondance immédiatement avant un `z`. De ce fait, cette requête se comporte différemment de `x <-> z & y <-> z`, qui correspondra

à un document contenant deux séquences séparées `x z` et `y z`. (Cette requête spécifique est inutile quand elle est écrite ainsi, car `x` et `y` ne peuvent pas être exactement à la même place ; mais avec des situations plus complexes comme les motifs de correspondance avec préfixe, une requête de cette forme pourrait être utile.)

12.1.3. Configurations

Les exemples ci-dessus ne sont que des exemples simples de recherche plein texte. Comme mentionné précédemment, la recherche plein texte permet de faire beaucoup plus : ignorer l'indexation de certains mots (termes courants), traiter les synonymes et utiliser une analyse sophistiquée, c'est-à-dire une analyse basée sur plus qu'un espace blanc. Ces fonctionnalités sont contrôlées par les *configurations de recherche plein texte*. PostgreSQL arrive avec des configurations prédéfinies pour de nombreux langages et vous pouvez facilement créer vos propres configurations (la commande `\dF` de psql affiche toutes les configurations disponibles).

Lors de l'installation, une configuration appropriée est sélectionnée et default_text_search_config est configuré dans `postgresql.conf` pour qu'elle soit utilisée par défaut. Si vous utilisez la même configuration de recherche plein texte pour le cluster entier, vous pouvez utiliser la valeur de `postgresql.conf`. Pour utiliser différentes configurations dans le cluster, mais avec la même configuration pour une base, utilisez `ALTER DATABASE ... SET`. Sinon, vous pouvez configurer `default_text_search_config` dans chaque session.

Chaque fonction de recherche plein texte qui dépend d'une configuration à un argument `regconfig` en option, pour que la configuration utilisée puisse être précisée explicitement. `default_text_search_config` est seulement utilisé quand cet argument est omis.

Pour rendre plus facile la construction de configurations de recherche plein texte, une configuration est construite à partir d'objets de la base de données. La recherche plein texte de PostgreSQL fournit quatre types d'objets relatifs à la configuration :

- Les *analyseurs de recherche plein texte* cassent les documents en jetons et classifient chaque jeton (par exemple, un mot ou un nombre).
- Les *dictionnaires de recherche plein texte* convertissent les jetons en une forme normalisée et rejettent les termes courants.
- Les *modèles de recherche plein texte* fournissent les fonctions nécessaires aux dictionnaires. (Un dictionnaire spécifie uniquement un modèle et un ensemble de paramètres pour ce modèle.)
- Les *configurations de recherche plein texte* sélectionnent un analyseur et un ensemble de dictionnaires à utiliser pour normaliser les jetons produits par l'analyseur.

Les analyseurs de recherche plein texte et les modèles sont construits à partir de fonctions bas niveau écrites en C ; du coup, le développement de nouveaux analyseurs ou modèles nécessite des connaissances en langage C, et les droits superutilisateur pour les installer dans une base de données. (Il y a des exemples d'analyseurs et de modèles en addon dans la partie `contrib/` de la distribution PostgreSQL.) Comme les dictionnaires et les configurations utilisent des paramètres et se connectent aux analyseurs et modèles, aucun droit spécial n'est nécessaire pour créer un nouveau dictionnaire ou une nouvelle configuration. Les exemples personnalisés de création de dictionnaires et de configurations seront présentés plus tard dans ce chapitre.

12.2. Tables et index

Les exemples de la section précédente illustrent la correspondance plein texte en utilisant des chaînes simples. Cette section montre comment rechercher les données de la table, parfois en utilisant des index.

12.2.1. Rechercher dans une table

Il est possible de faire des recherches plein texte sans index. Une requête qui ne fait qu'afficher le champ `title` de chaque ligne contenant le mot `friend` dans son champ `body` ressemble à ceci :

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english',
 'friend');
```

Cela trouve aussi les mots relatifs comme `friends` et `friendly`, car ils ont tous la même racine, le même lexème normalisé.

La requête ci-dessus spécifie que la configuration `english` doit être utilisée pour analyser et normaliser les chaînes. Nous pouvons aussi omettre les paramètres de configuration :

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

Cette requête utilisera l'ensemble de configuration indiqué par default_text_search_config.

Un exemple plus complexe est de sélectionner les dix documents les plus récents qui contiennent les mots `create` et `table` dans les champs `title` ou `body` :

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create &
 table')
ORDER BY last_mod_date DESC LIMIT 10;
```

Pour plus de clarté, nous omettons les appels à la fonction `coalesce` qui est nécessaire pour rechercher les lignes contenant `NULL` dans un des deux champs.

Bien que ces requêtes fonctionnent sans index, la plupart des applications trouvent cette approche trop lente, sauf peut-être pour des recherches occasionnelles. Une utilisation pratique de la recherche plein texte réclame habituellement la création d'un index.

12.2.2. Créer des index

Nous pouvons créer un index GIN (Section 12.9) pour accélérer les recherches plein texte :

```
CREATE INDEX pgweb_idx ON pgweb USING GIN(to_tsvector('english',
 body));
```

Notez que la version à deux arguments de `to_tsvector` est utilisée. Seules les fonctions de recherche plein texte qui spécifient un nom de configuration peuvent être utilisées dans les index sur des expressions (Section 11.7). Ceci est dû au fait que le contenu de l'index ne doit pas être affecté par default_text_search_config. Dans le cas contraire, le contenu de l'index peut devenir incohérent parce que différentes entrées pourraient contenir des `tsvector` créés avec différentes configurations de recherche plein texte et qu'il ne serait plus possible de deviner à quelle configuration fait référence une entrée. Il serait impossible de sauvegarder et restaurer correctement un tel index.

Comme la version à deux arguments de `to_tsvector` a été utilisée dans l'index ci-dessus, seule une référence de la requête qui utilise la version à deux arguments de `to_tsvector` avec le même nom

de configuration utilise cet index. C'est-à-dire que `WHERE to_tsvector('english', body) @@ 'a & b'` peut utiliser l'index, mais `WHERE to_tsvector(body) @@ 'a & b'` ne le peut pas. Ceci nous assure qu'un index est seulement utilisé avec la même configuration que celle utilisée pour créer les entrées de l'index.

Il est possible de configurer des index avec des expressions plus complexes où le nom de la configuration est indiqué dans une autre colonne. Par exemple :

```
CREATE INDEX pgweb_idx ON pgweb USING GIN(to_tsvector(config_name,
 body));
```

où `config_name` est une colonne de la table `pgweb`. Ceci permet l'utilisation de configurations mixtes dans le même index tout en enregistrant la configuration utilisée pour chaque entrée d'index. Ceci est utile dans le cas d'une bibliothèque de documents dans différentes langues. Encore une fois, les requêtes voulant utiliser l'index doivent être écrites pour correspondre à l'index, donc `WHERE to_tsvector(config_name, body) @@ 'a & b'`.

Les index peuvent même concaténer des colonnes :

```
CREATE INDEX pgweb_idx ON pgweb USING GIN(to_tsvector('english',
 title || ' ' || body));
```

Une autre approche revient à créer une colonne `tsvector` séparée pour contenir le résultat de `to_tsvector`. Pour garder cette colonne automatiquement à jour par rapport à la source, on peut utiliser une colonne générée stockée. Cet exemple est une concaténation de `title` et `body`, en utilisant `coalesce` pour s'assurer qu'un champ est toujours indexé même si l'autre vaut `NULL` :

```
ALTER TABLE pgweb
    ADD COLUMN textsearchable_index_col tsvector
               GENERATED ALWAYS AS (to_tsvector('english',
 coalesce(title, '') || ' ' || coalesce(body, ''))) STORED;
```

Puis nous créons un index GIN pour accélérer la recherche :

```
CREATE INDEX textsearch_idx ON pgweb USING
 GIN(textsearchable_index_col);
```

Maintenant, nous sommes prêts pour des recherches plein texte rapides :

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

Un avantage de l'approche de la colonne séparée sur un index par expression est qu'il n'est pas nécessaire de spécifier explicitement la configuration de recherche plein texte dans les requêtes pour utiliser l'index. Comme indiqué dans l'exemple ci-dessus, la requête peut dépendre de `default_text_search_config`. Un autre avantage est que les recherches seront plus rapides,

car il n'est plus nécessaire de refaire des appels à `to_tsvector` pour vérifier la correspondance de l'index. (Ceci est plus important lors de l'utilisation d'un index GiST par rapport à un index GIN ; voir la Section 12.9.) Néanmoins, l'approche de l'index par expression est plus simple à configurer et elle réclame moins d'espace disque, car la représentation `tsvector` n'est pas réellement stockée.

12.3. Contrôler la recherche plein texte

Pour implémenter la recherche plein texte, une fonction doit permettre la création d'un `tsvector` à partir d'un document et la création d'un `tsquery` à partir de la requête d'un utilisateur. De plus, nous avons besoin de renvoyer les résultats dans un ordre utile, donc nous avons besoin d'une fonction de comparaison des documents suivant leur adéquation à la recherche. Il est aussi important de pouvoir afficher joliment les résultats. PostgreSQL fournit un support pour toutes ces fonctions.

12.3.1. Analyser des documents

PostgreSQL fournit la fonction `to_tsvector` pour convertir un document vers le type de données `tsvector`.

```
    to_tsvector([ config regconfig, ] document text)
 returns tsvector
```

`to_tsvector` analyse un document texte et le convertit en jetons, réduit les jetons en des lexèmes et renvoie un `tsvector` qui liste les lexèmes avec leur position dans le document. Ce dernier est traité suivant la configuration de recherche plein texte spécifiée ou celle par défaut. Voici un exemple simple :

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a
 fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

Dans l'exemple ci-dessus, nous voyons que le `tsvector` résultant ne contient pas les mots `a`, `on` et `it`, le mot `rats` est devenu `rat` et le signe de ponctuation `-` a été ignoré.

En interne, la fonction `to_tsvector` appelle un analyseur qui casse le texte en jetons et affecte un type à chaque jeton. Pour chaque jeton, une liste de dictionnaires (Section 12.6) est consultée, liste pouvant varier suivant le type de jeton. Le premier dictionnaire qui *reconnaît* le jeton émet un ou plusieurs *lexèmes* pour représenter le jeton. Par exemple, `rats` devient `rat`, car un des dictionnaires sait que le mot `rats` est la forme plurielle de `rat`. Certains mots sont reconnus comme des *termes courants* (Section 12.6.1), ce qui fait qu'ils sont ignorés, car ils surviennent trop fréquemment pour être utile dans une recherche. Dans notre exemple, il s'agissait de `a`, `on` et `it`. Si aucun dictionnaire de la liste ne reconnaît le jeton, il est aussi ignoré. Dans cet exemple, il s'agit du signe de ponctuation `-`, car il n'existe aucun dictionnaire affecté à ce type de jeton (`Space symbols`), ce qui signifie que les jetons espace ne seront jamais indexés. Le choix de l'analyseur, des dictionnaires et des types de jetons à indexer est déterminé par la configuration de recherche plein texte sélectionnée (Section 12.7). Il est possible d'avoir plusieurs configurations pour la même base, et des configurations prédéfinies sont disponibles pour différentes langues. Dans notre exemple, nous avons utilisé la configuration par défaut, à savoir `english` pour l'anglais.

La fonction `setweight` peut être utilisée pour ajouter un label aux entrées d'un `tsvector` avec un *poids* donné. Ce poids consiste en une lettre : `A`, `B`, `C` ou `D`. Elle est utilisée typiquement pour marquer

les entrées provenant de différentes parties d'un document, comme le titre et le corps. Plus tard, cette information peut être utilisée pour modifier le score des résultats.

Comme `to_tsvector(NULL)` renvoie `NULL`, il est recommandé d'utiliser `coalesce` quand un champ peut être NULL. Voici la méthode recommandée pour créer un `tsvector` à partir d'un document structuré :

```
UPDATE tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

Ici, nous avons utilisé `setweight` pour ajouter un label à la source de chaque lexème dans le `tsvector` final, puis assemblé les valeurs `tsvector` en utilisant l'opérateur de concaténation des `tsvector`, `||`. (La Section 12.4.1 donne des détails sur ces opérations.)

12.3.2. Analyser des requêtes

PostgreSQL fournit les fonctions `to_tsquery`, `plainto_tsquery`, `phraseto_tsquery` et `websearch_to_tsquery` pour convertir une requête dans le type de données `tsquery`. `to_tsquery` offre un accès à d'autres fonctionnalités que `plainto_tsquery` et `phraseto_tsquery`, mais est moins indulgent sur ses arguments. `websearch_to_tsquery` est une version simplifiée de `to_tsquery` avec une syntaxe différente, similaire à celle utilisée par les moteurs web de recherche.

```
    to_tsquery([ config regconfig, ] querytext text)
 returns tsquery
```

`to_tsquery` crée une valeur `tsquery` à partir de *`querytext`* qui doit contenir un ensemble de jetons individuels séparés par les opérateurs `tsquery` & (AND), | (OR) et ! (NOT), et l'opérateur de recherche de phrase `<->` (FOLLOWED BY), possiblement groupés en utilisant des parenthèses. En d'autres termes, les arguments de `to_tsquery` doivent déjà suivre les règles générales pour un `tsquery` comme décrit dans la Section 8.11.2. La différence est que, alors qu'un `tsquery` basique prend les jetons bruts, `to_tsquery` normalise chaque jeton en un lexème en utilisant la configuration spécifiée ou par défaut, et annule tout jeton qui est un terme courant d'après la configuration. Par exemple :

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

Comme une entrée `tsquery` basique, des poids peuvent être attachés à chaque lexème à restreindre pour établir une correspondance avec seulement des lexèmes `tsvector` de ces poids. Par exemple :

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
 'fat' | 'rat':AB
```

De plus, * peut être attaché à un lexème pour demander la correspondance d'un préfixe :

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
--------------------------
 'supern':*A & 'star':*AB
```

Un tel lexème correspondra à tout mot dans un `tsvector` qui commence par la chaîne indiquée.

`to_tsquery` peut aussi accepter des phrases avec des guillemets simples. C'est utile quand la configuration inclut un dictionnaire thésaurus qui peut se déclencher sur de telles phrases. Dans l'exemple ci-dessous, un thésaurus contient la règle `supernovae stars : sn` :

```
SELECT to_tsquery('''supernovae stars'' & !crab');
  to_tsquery
---------------
 'sn' & !'crab'
```

Sans guillemets, `to_tsquery` génère une erreur de syntaxe pour les jetons qui ne sont pas séparés par un opérateur AND, ou FOLLOWED BY.

```
    plainto_tsquery([ config regconfig, ] querytext text)
 returns tsquery
```

`plainto_tsquery` transforme le texte non formaté *`querytext`* en `tsquery`. Le texte est analysé et normalisé un peu comme pour `to_tsvector`, ensuite l'opérateur `tsquery` & (AND) est inséré entre les mots restants.

Exemple :

```
 SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

Notez que `plainto_tsquery` ne reconnaîtra pas un opérateur `tsquery`, des labels de poids en entrée ou des labels de correspondance de préfixe :

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

Ici, tous les symboles de ponctuation ont été annulés.

```
phraseto_tsquery([ config regconfig, ] querytext text)
 returns tsquery
```

`phraseto_tsquery` se comporte largement comme `plainto_tsquery`, sauf qu'elle insère l'opérateur `<->` (FOLLOWED BY) entre les mots restants plutôt que l'opérateur & (AND). De plus, les termes courants ne sont pas simplement écartés, mais sont comptabilisés par l'utilisation d'opérateurs *<N>* plutôt que d'opérateurs `<->`. Cette fonction est utile quand on recherche des séquences exactes de lexèmes, puisque l'opérateur FOLLOWED BY vérifie l'ordre des lexèmes et pas seulement la présence de tous les lexèmes.

Exemple :

```
SELECT phraseto_tsquery('english', 'The Fat Rats');
 phraseto_tsquery
------------------
 'fat' <-> 'rat'
```

Comme `plainto_tsquery`, la fonction `phraseto_tsquery` ne reconnait ni les opérateurs `tsquery`, ni les labels de poids, ni les labels de correspondance de préfixe dans ses arguments :

```
SELECT phraseto_tsquery('english', 'The Fat & Rats:C');
      phraseto_tsquery
-----------------------------
 'fat' <-> 'rat' <-> 'c'
```

```
websearch_to_tsquery([ config regconfig, ] querytext text)
 returns tsquery
```

`websearch_to_tsquery` crée une valeur `tsquery` à partir de *querytext* en utilisant une syntaxe différente dans laquelle un texte simple non formaté est une requête valide. Contrairement à `plainto_tsquery` et `phraseto_tsquery`, elle reconnaît aussi certains opérateurs. De plus, cette fonction ne lèvera jamais d'erreurs de syntaxe, ce qui permet de l'utiliser pour la recherche à partir d'entrées brutes fournies par un utilisateur. La syntaxe suivante est supportée :

- `unquoted text` : du texte en dehors de guillemets sera converti en des termes séparés par des opérateurs &, comme traité par `plainto_tsquery`.
- `"quoted text"` : du texte à l'intérieur de guillemets sera converti en termes séparés par des opérateurs `<->`, comme traité par `phraseto_tsquery`.
- `OR` : the word « or » sera converti vers l'opérateur `|`.
- `-` : un tiret sera converti vers l'opérateur `!`.

Les autres ponctuations sont ignorées. Donc, tout comme `plainto_tsquery` et `phraseto_tsquery`,la fonction `websearch_to_tsquery` ne reconnaîtra pas les opérateurs `tsquery`, les labels de poids ou les labels de correspondance de préfixe en entrée.

Exemples :

```
SELECT websearch_to_tsquery('english', 'The fat rats');
 websearch_to_tsquery
----------------------
```

```
 'fat' & 'rat'
(1 row)

SELECT websearch_to_tsquery('english', '"supernovae stars" -crab');
       websearch_to_tsquery
----------------------------------
 'supernova' <-> 'star' & !'crab'
(1 row)

SELECT websearch_to_tsquery('english', '"sad cat" or "fat rat"');
       websearch_to_tsquery
---------------------------------
 'sad' <-> 'cat' | 'fat' <-> 'rat'
(1 row)

SELECT websearch_to_tsquery('english', 'signal -"segmentation
 fault"');
         websearch_to_tsquery
---------------------------------------
 'signal' & !( 'segment' <-> 'fault' )
(1 row)

SELECT websearch_to_tsquery('english', '""" )( dummy \\ query <-
>');
 websearch_to_tsquery
----------------------
 'dummi' & 'queri'
(1 row)
```

12.3.3. Ajouter un score aux résultats d'une recherche

Les tentatives de score pour mesurer l'adéquation des documents se font par rapport à une certaine requête. Donc, quand il y a beaucoup de correspondances, les meilleurs doivent être montrés en premier. PostgreSQL fournit deux fonctions prédéfinies de score, prenant en compte l'information lexicale, la proximité et la structure ; en fait, elles considèrent le nombre de fois où les termes de la requête apparaissent dans le document, la proximité des termes de la recherche avec ceux de la requête et l'importance du passage du document où se trouvent les termes du document. Néanmoins, le concept d'adéquation pourrait demander plus d'informations pour calculer le score, par exemple la date et l'heure de modification du document. Les fonctions internes de calcul de score sont seulement des exemples. Vous pouvez écrire vos propres fonctions de score et/ou combiner leurs résultats avec des facteurs supplémentaires pour remplir un besoin spécifique.

Les deux fonctions de score actuellement disponibles sont :

`ts_rank([ *weights* float4[], ] *vector* tsvector, *query* tsquery [, *normalization* integer ]) returns float4`

Calcule un score sur les vecteurs en se basant sur la fréquence des lexèmes correspondant à la recherche.

`ts_rank_cd([ *weights* float4[], ] *vector* tsvector, *query* tsquery [, *normalization* integer ]) returns float4`

Cette fonction calcule le score de la *densité de couverture* pour le vecteur du document et la requête donnés, comme décrit dans l'article de Clarke, Cormack et Tudhope, « Relevance Ranking for One to Three Term Queries », article paru dans le journal « Information Processing and Management » en 1999. La densité de couverture est similaire au classement effectué par

`ts_rank`, à la différence que la proximité de correspondance des lexèmes les uns par rapport aux autres est prise en compte.

Cette fonction a besoin d'information sur la position des lexèmes pour effectuer son travail. Par conséquent, elle ignore les lexèmes « stripés » dans le `tsvector`. S'il n'y a aucun lexème « non stripé » en entrée, le résultat sera zéro. (Voir Section 12.4.1 pour plus d'informations sur la fonction `strip` et les informations de position dans les `tsvector`.)

Pour ces deux fonctions, l'argument optionnel des *`poids`* offre la possibilité d'impacter certains mots plus ou moins suivant la façon dont ils sont marqués. Le tableau de poids indique à quel point chaque catégorie de mots est marquée. Dans l'ordre :

```
{poids-D, poids-C, poids-B, poids-A}
```

Si aucun *`poids`* n'est fourni, alors ces valeurs par défaut sont utilisées :

```
{0.1, 0.2, 0.4, 1.0}
```

Typiquement, les poids sont utilisés pour marquer les mots compris dans des aires spéciales du document, comme le titre ou le résumé initial, pour qu'ils puissent être traités avec plus ou moins d'importance que les mots dans le corps du document.

Comme un document plus long a plus de chances de contenir un terme de la requête, il est raisonnable de prendre en compte la taille du document, par exemple un document de cent mots contenant cinq fois un mot de la requête est probablement plus intéressant qu'un document de mille mots contenant lui aussi cinq fois un mot de la requête. Les deux fonctions de score prennent une option *`normalization`*, de type integer, qui précise si la longueur du document doit impacter son score. L'option contrôle plusieurs comportements, donc il s'agit d'un masque de bits : vous pouvez spécifier un ou plusieurs comportements en utilisant `|` (par exemple, `2|4`).

- 0 (valeur par défaut) ignore la longueur du document
- 1 divise le score par 1 + le logarithme de la longueur du document
- 2 divise le score par la longueur du document
- 4 divise le score par la moyenne harmonique de la distance entre les mots (ceci est implémenté seulement par `ts_rank_cd`)
- 8 divise le score par le nombre de mots uniques dans le document
- 16 divise le score par 1 + le logarithme du nombre de mots uniques dans le document
- 32 divise le score par lui-même + 1

Si plus d'un bit de drapeau est indiqué, les transformations sont appliquées dans l'ordre indiqué.

Il est important de noter que les fonctions de score n'utilisent aucune information globale, donc il est impossible de produire une normalisation de 1% ou 100%, comme c'est parfois demandé. L'option de normalisation 32 (`score/(score+1)`) peut s'appliquer pour échelonner tous les scores dans une échelle de zéro à un, mais, bien sûr, c'est une petite modification cosmétique, donc l'ordre des résultats ne changera pas.

Voici un exemple qui sélectionne seulement les dix correspondances de meilleur score :

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
```

```
LIMIT 10;
                     title                     |   rank
-----------------------------------------------+----------
 Neutrinos in the Sun                          |      3.1
 The Sudbury Neutrino Detector                 |      2.4
 A MACHO View of Galactic Dark Matter          |  2.01317
 Hot Gas and Dark Matter                       |  1.91171
 The Virgo Cluster: Hot Plasma and Dark Matter |  1.90953
 Rafting for Solar Neutrinos                   |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter     |  1.85774
 Hot Gas and Dark Matter                       |   1.6123
 Ice Fishing for Cosmic Neutrinos              |      1.6
 Weak Lensing Distorts the Universe            | 0.818218
```

Voici le même exemple en utilisant un score normalisé :

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1)
 */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |        rank
-----------------------------------------------+--------------------
 Neutrinos in the Sun                          | 0.756097569485493
 The Sudbury Neutrino Detector                 | 0.705882361190954
 A MACHO View of Galactic Dark Matter          | 0.668123210574724
 Hot Gas and Dark Matter                       |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
 Rafting for Solar Neutrinos                   | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter     | 0.650072921219637
 Hot Gas and Dark Matter                       | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos              | 0.615384618911517
 Weak Lensing Distorts the Universe            | 0.450010798361481
```

Le calcul du score peut consommer beaucoup de ressources, car il demande de consulter le `tsvector` de chaque document correspondant, ce qui est très consommateur en entrées/sorties et du coup lent. Malheureusement, c'est presque impossible à éviter, car les requêtes intéressantes ont un grand nombre de correspondances.

12.3.4. Surligner les résultats

Pour présenter les résultats d'une recherche, il est préférable d'afficher une partie de chaque document et en quoi cette partie concerne la requête. Habituellement, les moteurs de recherche affichent des fragments du document avec des marques pour les termes recherchés. PostgreSQL fournit une fonction `ts_headline` qui implémente cette fonctionnalité.

```
    ts_headline([ config regconfig, ] document text, query tsquery
 [, options text ]) returns text
```

`ts_headline` accepte un document avec une requête et renvoie un résumé du document. Les termes de la requête sont surlignés dans les extractions. La configuration à utiliser pour

analyser le document peut être précisée par *config* ; si *config* est omis, le paramètre default_text_search_config est utilisé.

Si une chaîne *options* est spécifiée, elle doit consister en une liste d'une ou plusieurs paires *option=valeur* séparées par des virgules. Les options disponibles sont :

- MaxWords, MinWords (entiers) : ces nombres déterminent les limites minimum et maximum des résumés à afficher. Les valeurs par défaut sont respectivement 35 et 15.
- ShortWord (entier) : les mots de cette longueur et les mots plus petits seront supprimés au début et à la fin d'un résumé, sauf si ce sont des mots de la recherche. La valeur par défaut est de trois pour éliminer les articles anglais communs.
- HighlightAll (booléen) : si true, le document complet sera utilisé comme résumé, en ignorant les trois paramètres précédents. La valeur par défaut est false.
- MaxFragments (entier) : nombre maximum de fragments de texte à afficher. La valeur par défaut, 0, sélectionne une méthode de génération de résumés non basés sur des fragments. Une valeur positive sélectionne la génération de résumé basée sur les fragments (voir ci-dessous).
- StartSel, StopSel (chaînes) : les chaînes qui permettent de délimiter les mots de la requête apparaissant dans le document pour les distinguer des autres mots du résumé. Les valeurs par défaut sont « » et « », qui sont convenables pour une sortie HTML (mais voir le message d'avertissement ci-dessous).
- FragmentDelimiter (chaîne) : quand plus d'un fragment est affiché, alors les fragments seront séparés par ce délimiteur. La valeur par défaut est « ... ».

> **Attention : Sécurité des scripts intersites (XSS)**
>
> Il n'est pas garantie que la sortie de ts_headline peut être inclus directement dans les pages web en toute sécurité. Quand HighlightAll est false (la valeur par défaut), certaines balises XML simples sont supprimées du document mais il n'est pas garanti que tout le balisage HTML sera supprimé. De ce fait, cela ne fournit pas une défense efficace contre les attaques de type scripts intersites (XSS), lorsque vous travaillez avec des données non sûres. Pour se défendre contre de telles attaques, tout le balisage HTML doit être supprimé à partir du document en entrée ou il faut utiliser un outil de nettoyage HTML sur la sortie.

Les noms de ces options s'utilisent avec ou sans casse. Vous devez mettre les chaînes de caractères entre guillemets doubles si elles contiennent des espaces ou des virgules.

Pour la génération des résumés qui ne sont pas basés sur des fragments, ts_headline repère les correspondances pour la *requete* donnée et en choisit une seule à afficher, en préférant les correspondances qui ont le plus grand nombre de mots provenant de la requête dans la longueur autorisée pour le résumé. Pour la génération des résumés basés sur les fragments, ts_headline repère les correspondances pour la requête et divise chaque correspondance en « fragments » d'au plus MaxWords mots chacun, en préférant les fragments avec le plus de mots provenant de la requête, et si possible des fragments se prolongeant pour inclure les mots autour. Le mode basé sur les fragments est donc plus utile quand la requête correspond à de grosses sections du document ou quand il est préférable d'afficher plusieurs correspondances. Dans les deux modes, si aucune correspondance n'est identifiée, alors un seul fragment des MinWords premiers mots du document sera affiché.

Par exemple :

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('english', 'query & similarity'));
                        ts_headline
```

```
------------------------------------------------------------
 containing given <b>query</b> terms                        +
 and return them in order of their <b>similarity</b> to the+
 <b>query</b>.

SELECT ts_headline('english',
  'Search terms may occur
many times in a document,
requiring ranking of the search matches to decide which
occurrences to display in the result.',
  to_tsquery('english', 'search & term'),
  'MaxFragments=10, MaxWords=7, MinWords=3, StartSel=<<,
 StopSel=>>');
                        ts_headline
------------------------------------------------------------
 <<Search>> <<terms>> may occur                             +
 many times ... ranking of the <<search>> matches to decide
```

`ts_headline` utilise le document original, pas un résumé `tsvector`, donc elle peut être lente et doit être utilisée avec parcimonie et attention.

12.4. Fonctionnalités supplémentaires

Cette section décrit des fonctions et opérateurs supplémentaires qui sont utiles en relation avec la recherche plein texte.

12.4.1. Manipuler des documents

La Section 12.3.1 a montré comment des documents en texte brut peuvent être convertis en valeurs `tsvector`. PostgreSQL fournit aussi des fonctions et des opérateurs pouvant être utilisés pour manipuler des documents qui sont déjà au format `tsvector`.

`tsvector || tsvector`

L'opérateur de concaténation `tsvector` renvoie un vecteur qui combine les lexèmes et des informations de position pour les deux vecteurs donnés en argument. Les positions et les labels de poids sont conservés lors de la concaténation. Les positions apparaissant dans le vecteur droit sont décalées par la position la plus large mentionnée dans le vecteur gauche, pour que le résultat soit pratiquement équivalent au résultat du traitement de `to_tsvector` sur la concaténation des deux documents originaux. (L'équivalence n'est pas exacte, car tout terme courant supprimé de la fin de l'argument gauche n'affectera pas le résultat alors qu'ils auraient affecté les positions des lexèmes dans l'argument droit si la concaténation de texte avait été utilisée.)

Un avantage de l'utilisation de la concaténation au format vecteur, plutôt que la concaténation de texte avant d'appliquer `to_tsvector`, est que vous pouvez utiliser différentes configurations pour analyser les différentes sections du document. De plus, comme la fonction `setweight` marque tous les lexèmes du secteur donné de la même façon, il est nécessaire d'analyser le texte et de lancer `setweight` avant la concaténation si vous voulez des labels de poids différents sur les différentes parties du document.

`setweight(`*`vector`* `tsvector, `*`weight`* `"char") returns tsvector`

Cette fonction renvoie une copie du vecteur en entrée pour chaque position de poids *`weight`*, soit `A`, soit `B`, soit `C` soit `D`. (`D` est la valeur par défaut pour les nouveaux vecteurs et, du coup, n'est pas affichée en sortie.) Ces labels sont conservés quand les vecteurs sont concaténés, permettant aux mots des différentes parties d'un document de se voir attribuer un poids différent par les fonctions de score.

Notez que les labels de poids s'appliquent seulement aux *positions*, pas aux *lexèmes*. Si le vecteur en entrée se voit supprimer les positions, alors `setweight` ne pourra rien faire.

`length(`*`vector`* `tsvector) returns integer`

Renvoie le nombre de lexèmes enregistrés dans le vecteur.

`strip(`*`vector`* `tsvector) returns tsvector`

Renvoie un vecteur qui liste les mêmes lexèmes que le vecteur donné, mais à qui il manquera les informations de position et de poids. Alors que le vecteur renvoyé est bien moins utile qu'un vecteur normal pour calculer le score, il est habituellement bien plus petit. Le classement par pertinence ne fonctionne pas aussi bien sur les vecteurs stripés que sur les non-stripés. Par ailleurs, l'opérateur `tsquery <->` (FOLLOWED BY) ne trouvera jamais de correspondance pour des entrées stripées, puisqu'il ne peut pas déterminer la distance entre deux occurrences de lexèmes dans ce cas.

Une liste complète des fonctions relatives aux `tsvector` est disponible à Tableau 9.43.

12.4.2. Manipuler des requêtes

La Section 12.3.2 a montré comment des requêtes texte peuvent être converties en valeurs de type `tsquery`. PostgreSQL fournit aussi des fonctions et des opérateurs pouvant être utilisés pour manipuler des requêtes qui sont déjà de la forme `tsquery`.

`tsquery && tsquery`

Renvoie une combinaison AND des deux requêtes données.

`tsquery || tsquery`

Renvoie une combinaison OR des deux requêtes données.

`!! tsquery`

Renvoie la négation (NOT) de la requête donnée.

`tsquery <-> tsquery`

Renvoie une requête qui recherche une correspondance avec la première requête donnée suivie immédiatement par une correspondance avec la seconde requête donnée, en utilisant l'opérateur `tsquery <->` (FOLLOWED BY). Par exemple :

```
SELECT to_tsquery('fat') <-> to_tsquery('cat | rat');
             ?column?
---------------------------------
 'fat' <-> 'cat' | 'fat' <-> 'rat'
```

`tsquery_phrase(`*`query1`* `tsquery,` *`query2`* `tsquery [,` *`distance`* `integer ]) returns tsquery`

Renvoie une requête qui recherche une correspondance avec la première requête donnée suivie par une correspondance avec la seconde requête donnée, à une distance d'au plus *`distance`* lexèmes, en utilisant l'opérateur `tsquery <`*`N`*`>`. Par exemple :

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);
  tsquery_phrase
------------------
```

```
 'fat' <10> 'cat'
```

`numnode(`*`query`* `tsquery) returns integer`

Renvoie le nombre de nœuds (lexèmes et opérateurs) dans un `tsquery`. Cette fonction est utile pour déterminer si la requête (*`query`*) a un sens (auquel cas elle renvoie > 0) ou si elle ne contient que des termes courants (auquel cas elle renvoie 0). Exemples :

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE:  query contains only stopword(s) or doesn't contain
 lexeme(s), ignored
 numnode
---------
       0

SELECT numnode('foo & bar'::tsquery);
 numnode
---------
       3
```

`querytree(`*`query`* `tsquery) returns text`

Renvoie la portion d'un `tsquery` qui peut être utilisée pour rechercher dans un index. Cette fonction est utile pour détecter les requêtes qui ne peuvent pas utiliser un index, par exemple celles qui contiennent des termes courants ou seulement des négations de termes. Par exemple :

```
SELECT querytree(to_tsquery('defined'));
 querytree
-----------
 'defin'

SELECT querytree(to_tsquery('!defined'));
 querytree
-----------
 T
```

12.4.2.1. Ré-écriture des requêtes

La famille de fonctions `ts_rewrite` cherche dans un `tsquery` donné les occurrences d'une sous-requête cible et remplace chaque occurrence avec une autre sous-requête de substitution. En fait, cette opération est une version spécifique à `tsquery` d'un remplacement de sous-chaîne. Une combinaison cible et substitut peut être vue comme une *règle de ré-écriture de la requête*. Un ensemble de règles de ré-écriture peut être une aide puissante à la recherche. Par exemple, vous pouvez étendre la recherche en utilisant des synonymes (`new york`, `big apple`, `nyc`, `gotham`) ou restreindre la recherche pour diriger l'utilisateur vers des thèmes en vogue. Cette fonctionnalité n'est pas sans rapport avec les thésaurus (Section 12.6.4). Néanmoins, vous pouvez modifier un ensemble de règles de ré-écriture directement, sans réindexer, alors que la mise à jour d'un thésaurus nécessite une réindexation pour être pris en compte.

`ts_rewrite (`*`query`* `tsquery,` *`target`* `tsquery,` *`substitute`* `tsquery) returns tsquery`

Cette forme de `ts_rewrite` applique simplement une seule règle de ré-écriture : *`target`* est remplacé par *`substitute`* partout où il apparaît dans *`query`*. Par exemple :

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
------------
 'b' & 'c'
```

ts_rewrite (*query* tsquery, *select* text) returns tsquery

Cette forme de `ts_rewrite` accepte une *`query`* de début et une commande SQL *`select`*, qui est fournie comme une chaîne de caractères. *`select`* doit renvoyer deux colonnes de type `tsquery`. Pour chaque ligne de résultats du *`select`*, les occurrences de la valeur de la première colonne (la cible) sont remplacées par la valeur de la deuxième colonne (le substitut) dans la valeur actuelle de *`query`*. Par exemple :

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
 ts_rewrite
------------
 'b' & 'c'
```

Notez que, quand plusieurs règles de ré-écriture sont appliquées de cette façon, l'ordre d'application peut être important ; donc, en pratique, vous voudrez que la requête source utilise `ORDER BY` avec un ordre précis.

Considérons un exemple réel pour l'astronomie. Nous étendons la requête `supernovae` en utilisant les règles de ré-écriture par la table :

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'),
 to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM
 aliases');
          ts_rewrite
--------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

Nous pouvons modifier les règles de ré-écriture simplement en mettant à jour la table :

```
UPDATE aliases SET s = to_tsquery('supernovae|sn & !nebulae') WHERE
 t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM
 aliases');
                ts_rewrite
-------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

La ré-écriture peut être lente quand il y a beaucoup de règles de ré-écriture, car elle vérifie l'intérêt de chaque règle. Pour filtrer les règles qui ne sont pas candidates de façon évidente, nous pouvons utiliser

les opérateurs de contenant pour le type `tsquery`. Dans l'exemple ci-dessous, nous sélectionnons seulement les règles qui peuvent correspondre avec la requête originale :

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE ''a & b''::tsquery
 @> t');
 ts_rewrite
------------
 'b' & 'c'
```

12.4.3. Triggers pour les mises à jour automatiques

> **Note**
>
> La méthode décrite dans cette section a été rendue obsolète par l'utilisation des colonnes générées stockées, comme décrit dans Section 12.2.2.

Lors de l'utilisation d'une colonne séparée pour stocker la représentation `tsvector` de vos documents, il est nécessaire de créer un trigger pour mettre à jour la colonne `tsvector` quand le contenu des colonnes document change. Deux fonctions trigger intégrées sont disponibles pour cela, mais vous pouvez aussi écrire la vôtre.

```
tsvector_update_trigger(nom_colonne_tsvector,
 nom_config, nom_colonne_texte [, ... ])
tsvector_update_trigger_column(nom_colonne_tsvector,
 nom_colonne_config, nom_colonne_texte [, ... ])
```

Ces fonctions trigger calculent automatiquement une colonne `tsvector` à partir d'une ou plusieurs colonnes texte sous le contrôle des paramètres spécifiés dans la commande `CREATE TRIGGER`. Voici un exemple de leur utilisation :

```
CREATE TABLE messages (
    title       text,
    body        text,
    tsv         tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE FUNCTION
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title    |         body          |            tsv
------------+-----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title &
 body');
   title    |         body
```

```
------------+-----------------------
 title here | the body text is here
```

Après avoir créé ce trigger, toute modification dans `title` ou `body` sera automatiquement reflétée dans `tsv`, sans que l'application ait à s'en soucier.

Le premier argument du trigger doit être le nom de la colonne `tsvector` à mettre à jour. Le deuxième argument spécifie la configuration de recherche plein texte à utiliser pour réaliser la conversion. Pour `tsvector_update_trigger`, le nom de la configuration est donné en deuxième argument du trigger. Il doit être qualifié du nom du schéma comme indiqué ci-dessus pour que le comportement du trigger ne change pas avec les modifications de `search_path`. Pour `tsvector_update_trigger_column`, le deuxième argument du trigger est le nom d'une autre colonne de table qui doit être du type `regconfig`. Ceci permet une sélection par ligne de la configuration à faire. Les arguments restants sont les noms des colonnes texte (de type `text`, `varchar` ou `char`). Elles sont incluses dans le document suivant l'ordre donné. Les valeurs NULL sont ignorées (mais les autres colonnes sont toujours indexées).

Une limitation des triggers internes est qu'ils traitent les colonnes de façon identique. Pour traiter les colonnes différemment -- par exemple pour donner un poids plus important au titre qu'au corps -- il est nécessaire d'écrire un trigger personnalisé. Voici un exemple utilisant PL/pgSQL comme langage du trigger :

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
  new.tsv :=
     setweight(to_tsvector('pg_catalog.english',
 coalesce(new.title,'')), 'A') ||
     setweight(to_tsvector('pg_catalog.english',
 coalesce(new.body,'')), 'D');
  return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE FUNCTION messages_trigger();
```

Gardez en tête qu'il est important de spécifier explicitement le nom de la configuration lors de la création de valeurs `tsvector` dans des triggers, pour que le contenu de la colonne ne soit pas affecté par des modifications de `default_text_search_config`. Dans le cas contraire, des problèmes surviendront comme des résultats de recherche changeant après une sauvegarde/restauration.

12.4.4. Récupérer des statistiques sur les documents

La fonction `ts_stat` est utile pour vérifier votre configuration et pour trouver des candidats pour les termes courants.

```
    ts_stat(sqlquery text, [ weights text, ]
            OUT word text, OUT ndoc integer,
            OUT nentry integer) returns setof record
```

`sqlquery` est une valeur de type texte contenant une requête SQL qui doit renvoyer une seule colonne `tsvector`. `ts_stat` exécute la requête et renvoie des statistiques sur chaque lexème (mot) contenu dans les données `tsvector`. Les colonnes renvoyées sont :

- *word* `text` -- la valeur d'un lexème
- *ndoc* `integer` -- le nombre de documents (`tsvector`) où le mot se trouve
- *nentry* `integer` -- le nombre total d'occurrences du mot

Si *weights* est précisé, seules les occurrences d'un de ces poids sont comptabilisées.

Par exemple, pour trouver les dix mots les plus fréquents dans un ensemble de document :

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

De la même façon, mais en ne comptant que les occurrences de poids `A` ou `B` :

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

12.5. Analyseurs

Les analyseurs de recherche plein texte sont responsables du découpage d'un document brut en *jetons* et d'identifier le type des jetons. L'ensemble des types possibles est défini par l'analyseur lui-même. Notez qu'un analyseur ne modifie pas le texte -- il identifie les limites plausibles des mots. Comme son domaine est limité, il est moins important de pouvoir construire des analyseurs personnalisés pour une application. Actuellement, PostgreSQL fournit un seul analyseur interne qui s'est révélé utile pour un ensemble varié d'applications.

L'analyseur interne est nommé `pg_catalog.default`. Il reconnaît 23 types de jeton, dont la liste est disponible dans Tableau 12.1.

Tableau 12.1. Types de jeton de l'analyseur par défaut

Alias	Description	Exemple
`asciiword`	Mot, toute lettre ASCII	`elephant`
`word`	Mot, toute lettre	`mañana`
`numword`	Mot, lettres et chiffres	`beta1`
`asciihword`	Mot composé, en ASCII	`up-to-date`
`hword`	Mot composé, toutes les lettres	`lógico-matemática`
`numhword`	Mot composé, lettre et chiffre	`postgresql-beta1`
`hword_asciipart`	Partie d'un mot composé, en ASCII	`postgresql` dans le contexte `postgresql-beta1`
`hword_part`	Partie d'un mot composé, toutes les lettres	`lógico` ou `matemática` dans le contexte `lógico-matemática`
`hword_numpart`	Partie d'un mot composé, lettres et chiffres	`beta1` dans le contexte `postgresql-beta1`
`email`	Adresse email	`foo@example.com`
`protocol`	En-tête de protocole	`http://`

Alias	Description	Exemple
`url`	URL	`example.com/stuff/index.html`
`host`	Hôte	`example.com`
`url_path`	Chemin URL	`/stuff/index.html`, dans le contexte d'une URL
`file`	Fichier ou chemin	`/usr/local/foo.txt`, en dehors du contexte d'une URL
`sfloat`	Notation scientifique	`-1.234e56`
`float`	Notation décimale	`-1.234`
`int`	Entier signé	`-1234`
`uint`	Entier non signé	`1234`
`version`	Numéro de version	`8.3.0`
`tag`	Balise XML	`<a href="dictionaries.html">`
`entity`	Entité XML	`&`
`blank`	Symboles espaces	(tout espace blanc, ou signe de ponctuation non reconnu autrement)

> **Note**
>
> La notion de l'analyseur d'une « lettre » est déterminée par la configuration de la locale sur la base de données, spécifiquement par `lc_ctype`. Les mots contenant seulement des lettres ASCII basiques sont reportés comme un type de jeton séparé, car il est parfois utile de les distinguer. Dans la plupart des langues européennes, les types de jeton `word` et `asciiword` doivent toujours être traités de la même façon.

`email` ne supporte pas tous les caractères email valides tels qu'ils sont définis par la RFC 5322[1]. Spécifiquement, les seuls caractères non alphanumériques supportés sont le point, le tiret et le tiret bas.

`tag` ne supporte pas tous les noms de balise valides d'après la définition du W3C Recommendation, XML[2]. Spécifiquement, les seuls noms de balise supportés sont ceux commençant avec une lettre ASCII, tiret bas, ou un deux-points et contenant seulement des lettres, des chiffres, des tirets, des tirets bas et des deux-points. `tag` inclut aussi les commentaires XML commençant avec `<!--` et finissant avec `-->`, et les déclarations XML (mais notez que ceci inclut tout ce qui commence avec `<?x` et se finit avec `>`).

Il est possible que l'analyseur produise des jetons qui coïncident à partir du même texte. Comme exemple, un mot composé peut être reporté à la fois comme un mot entier et pour chaque composante :

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |               description                |
 token
-----------------+------------------------------------------
+---------------
 numhword        | Hyphenated word, letters and digits      | foo-
bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
```

[1] https://datatracker.ietf.org/doc/html/rfc5322
[2] https://www.w3.org/TR/xml/

```
 blank           | Space symbols                               | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

Ce comportement est souhaitable, car il autorise le bon fonctionnement de la recherche sur le mot composé et sur les composants. Voici un autre exemple instructif :

```
SELECT alias, description, token FROM ts_debug('http://example.com/
stuff/index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

12.6. Dictionnaires

Les dictionnaires sont utilisés pour éliminer des mots qui ne devraient pas être considérés dans une recherche (*termes courants*), et pour *normaliser* des mots pour que des formes dérivées de ce même mot établissent une correspondance. Un mot normalisé avec succès est appelé un *lexème*. En dehors du fait d'améliorer la qualité de la recherche, la normalisation et la suppression des termes courants réduisent la taille de la représentation d'un document en `tsvector`, et donc améliorent les performances. La normalisation n'a pas toujours une signification linguistique et dépend habituellement de la sémantique de l'application.

Quelques exemples de normalisation :

- Linguistique -- les dictionnaires ispell tentent de réduire les mots en entrée en une forme normalisée ; les dictionnaires stemmer suppriment la fin des mots
- Les URL peuvent être réduites pour établir certaines correspondances :
  - http://www.pgsql.ru/db/mw/index.html
  - http://www.pgsql.ru/db/mw/
  - http://www.pgsql.ru/db/../db/mw/index.html
- Les noms de couleur peuvent être remplacés par leur valeur hexadécimale, par exemple `red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF`
- En cas d'indexation de nombre, nous pouvons supprimer certains chiffres à fraction pour réduire les nombres possibles, donc par exemple *3.14*159265359, *3.14*15926, *3.14* seront identiques après normalisation si seuls deux chiffres sont conservés après le point décimal.

Un dictionnaire est un programme qui accepte un jeton en entrée et renvoie :

- un tableau de lexèmes si le jeton en entrée est connu dans le dictionnaire (notez qu'un jeton peut produire plusieurs lexèmes)
- un unique lexème avec le drapeau `TSL_FILTER` configuré, pour remplacer le jeton original avec un nouveau jeton à passer aux dictionnaires suivants (un dictionnaire de ce type est appelé un *dictionnaire filtrant*)
- un tableau vide si le dictionnaire connaît le jeton, mais que ce dernier est un terme courant
- `NULL` si le dictionnaire n'a pas reconnu le jeton en entrée

PostgreSQL fournit des dictionnaires prédéfinis pour de nombreuses langues. Il existe aussi plusieurs modèles prédéfinis qui peuvent être utilisés pour créer de nouveaux dictionnaires avec des paramètres personnalisés. Chaque modèle prédéfini de dictionnaire est décrit ci-dessous. Si aucun modèle ne convient, il est possible d'en créer de nouveaux ; voir le répertoire `contrib/` de PostgreSQL pour des exemples.

Une configuration de recherche plein texte lie un analyseur avec un ensemble de dictionnaires pour traiter les jetons en sortie de l'analyseur. Pour chaque type de jeton que l'analyseur peut renvoyer, une liste séparée de dictionnaires est indiquée par la configuration. Quand un jeton de ce type est trouvé par l'analyseur, chaque dictionnaire de la liste est consulté jusqu'à ce qu'un dictionnaire le reconnaisse comme un mot connu. S'il est identifié comme un terme courant ou si aucun dictionnaire ne le reconnaît, il sera ignoré et non indexé. Normalement, le premier dictionnaire qui renvoie une sortie non `NULL` détermine le résultat et tout dictionnaire restant n'est pas consulté ; par contre, un dictionnaire filtrant peut remplacer le mot donné avec un autre mot qui est ensuite passé aux dictionnaires suivants.

La règle générale pour la configuration de la liste des dictionnaires est de placer en premier les dictionnaires les plus précis, les plus spécifiques, puis les dictionnaires généralistes, en finissant avec un dictionnaire le plus général possible, comme un stemmer Snowball ou `simple`, qui reconnaît tout. Par exemple, pour une recherche en astronomie (configuration `astro_en`), vous pouvez lier le type de jeton `asciiword` (mot ASCII) vers un dictionnaire des synonymes des termes de l'astronomie, un dictionnaire anglais généraliste et un stemmer Snowball anglais :

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell,
 english_stem;
```

Un dictionnaire filtrant peut être placé n'importe où dans la liste. Cependant, le placer à la fin n'a aucun intérêt. Les dictionnaires filtrants sont utiles pour normaliser partiellement les mots, ce qui permet de simplifier la tâche aux dictionnaires suivants. Par exemple, un dictionnaire filtrant peut être utilisé pour supprimer les accents des lettres accentuées. C'est ce que fait le module unaccent.

12.6.1. Termes courants

Les termes courants sont des mots très courants, apparaissant dans pratiquement chaque document et n'ont donc pas de valeur discriminatoire. Du coup, ils peuvent être ignorés dans le contexte de la recherche plein texte. Par exemple, tous les textes anglais contiennent des mots comme `a` et `the`, donc il est inutile de les stocker dans un index. Néanmoins, les termes courants n'affectent pas les positions dans `tsvector`, ce qui affecte le score :

```
SELECT to_tsvector('english', 'in the list of stop words');
        to_tsvector
----------------------------
 'list':3 'stop':5 'word':6
```

Les positions 1, 2, 4 manquantes sont dues aux termes courants. Les scores calculés pour les documents avec et sans termes courants sont vraiment différents :

```
SELECT ts_rank_cd (to_tsvector('english', 'in the list of stop
 words'), to_tsquery('list & stop'));
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english', 'list stop words'),
 to_tsquery('list & stop'));
 ts_rank_cd
------------
        0.1
```

C'est au dictionnaire de savoir comment traiter les mots courants. Par exemple, les dictionnaires `Ispell` normalisent tout d'abord les mots puis cherchent les termes courants alors que les stemmers `Snowball` vérifient d'abord leur liste de termes courants. La raison de leur comportement différent est qu'ils tentent de réduire le bruit.

12.6.2. Dictionnaire simple

Le modèle du dictionnaire `simple` opère en convertissant le jeton en entrée en minuscules puis en vérifiant s'il fait partie de la liste des mots courants qu'il a sur fichier. S'il est trouvé dans ce fichier, un tableau vide est renvoyé. Le jeton sera alors ignoré. Dans le cas contraire, la forme minuscule du mot est renvoyée en tant que lexème normalisé. Autrement, le dictionnaire peut être configuré pour rapporter les termes courants comme étant non reconnus, ce qui permet de les passer au prochain dictionnaire de la liste.

Voici un exemple d'une définition de dictionnaire utilisant le modèle `simple` :

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

Dans ce cas, `english` est le nom de base du fichier contenant les termes courants. Le nom complet du fichier sera donc `$SHAREDIR/tsearch_data/english.stop`, où `$SHAREDIR` est le répertoire des données partagées de l'installation de PostgreSQL (souvent `/usr/local/share/postgresql`, mais utilisez `pg_config --sharedir` pour vous en assurer). Le format du fichier est une simple liste de mots, un mot par ligne. Les lignes vides et les espaces en fin de mot sont ignorées. Les mots en majuscules sont basculés en minuscules, mais aucun autre traitement n'est réalisé sur le contenu de ce fichier.

Maintenant, nous pouvons tester notre dictionnaire :

```
SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
 {}
```

Nous pouvons aussi choisir de renvoyer `NULL` à la place du mot en minuscules s'il n'est pas trouvé dans le fichier des termes courants. Ce comportement est sélectionné en configurant le paramètre `Accept` du dictionnaire à `false`. En continuant l'exemple :

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------


SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
```

```
-----------
 {}
```

Avec le paramétrage par défaut d'`Accept` (à savoir, `true`), il est préférable de placer un dictionnaire `simple` à la fin de la liste des dictionnaires. `Accept` = `false` est seulement utile quand il y a au moins un dictionnaire après celui-ci.

> **Attention**
>
> La plupart des types de dictionnaires se basent sur des fichiers de configuration, comme les fichiers de termes courants. Ces fichiers *doivent* être dans l'encodage UTF-8. Ils seront traduits vers l'encodage actuel de la base de données, s'il est différent, quand ils seront lus.

> **Attention**
>
> Habituellement, une session lira un fichier de configuration du dictionnaire une seule fois, lors de la première utilisation. Si vous modifiez un fichier de configuration et que vous voulez forcer la prise en compte des modifications par les sessions en cours, exécutez une commande `ALTER TEXT SEARCH DICTIONARY` sur le dictionnaire. Cela peut être une mise à jour « à vide », donc sans réellement modifier des valeurs.

12.6.3. Dictionnaire des synonymes

Ce modèle de dictionnaire est utilisé pour créer des dictionnaires qui remplacent un mot par un synonyme. Les phrases ne sont pas supportées (utilisez le modèle thésaurus pour cela, Section 12.6.4). Un dictionnaire des synonymes peut être utilisé pour contourner des problèmes linguistiques, par exemple pour empêcher un dictionnaire stemmer anglais de réduire le mot « Paris » en « pari ». Il suffit d'avoir une ligne `Paris paris` dans le dictionnaire des synonymes et de le placer avant le dictionnaire `english_stem`. Par exemple :

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |  dictionaries  |  dictionary
  | lexemes
-----------+-----------------+-------+----------------
+--------------+---------
 asciiword | Word, all ASCII | Paris | {english_stem} |
 english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |       dictionaries        |
 dictionary | lexemes
-----------+-----------------+-------+---------------------------
+------------+---------
```

```
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} |
 my_synonym | {paris}
```

Le seul paramètre requis par le modèle `synonym` est `SYNONYMS`, qui est le nom de base de son fichier de configuration -- `my_synonyms` dans l'exemple ci-dessus. Le nom complet du fichier sera `$SHAREDIR/tsearch_data/my_synonyms.syn` (où `$SHAREDIR` correspond au répertoire des données partagées de l'installation de PostgreSQL). Le format du fichier est une ligne par mot à substituer, avec le mot suivi par son synonyme séparé par une espace blanche. Les lignes vierges et les espaces après les mots sont ignorées, les lettres majuscules sont mises en minuscules.

Le modèle `synonym` a aussi un paramètre optionnel, appelé `CaseSensitive`, qui vaut par défaut `false`. Quand `CaseSensitive` vaut `false`, les mots dans le fichier des synonymes sont mis en minuscules, comme les jetons en entrée. Quand il vaut vrai, les mots et les jetons ne sont plus mis en minuscules, mais comparés tels quels..

Un astérisque (`*`) peut être placé à la fin d'un synonyme dans le fichier de configuration. Ceci indique que le synonyme est un préfixe. L'astérisque est ignoré quand l'entrée est utilisée dans `to_tsvector()`, mais quand il est utilisé dans `to_tsquery()`, le résultat sera un élément de la requête avec le marqueur de correspondance du préfixe (voir Section 12.3.2). Par exemple, supposons que nous ayons ces entrées dans `$SHAREDIR/tsearch_data/synonym_sample.syn` :

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

Alors nous obtiendrons les résultats suivants :

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym,
 synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn', 'indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR
 asciiword WITH syn;
mydb=# SELECT to_tsvector('tst', 'indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst', 'indices');
 to_tsquery
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
            tsvector
--------------------------------
```

```
 'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@
 to_tsquery('tst', 'indices');
 ?column?
----------
 t
(1 row)
```

12.6.4. Dictionnaire thésaurus

Un dictionnaire thésaurus (parfois abrégé en TZ) est un ensemble de mots qui incluent des informations sur les relations des mots et des phrases, par exemple des termes plus lointains (BT), plus proches (NT), des termes préférés, des termes non aimés, des termes en relation, etc.

De façon simple, un dictionnaire thésaurus remplace tous les termes par des termes préférés et, en option, conserve les termes originaux pour l'indexage. L'implémentation actuelle du dictionnaire thésaurus par PostgreSQL est une extension du dictionnaire des synonymes avec un support additionnel des *phrases*. Un dictionnaire thésaurus nécessite un fichier de configuration au format suivant :

```
# ceci est un commentaire
mots(s) : mot(s) indexé(s)
d'autre(s) mot(s) : d'autre(s) mot(s) indexé(s)
...
```

où le deux-points (`:`) agit comme un délimiteur entre une phrase et son remplacement.

Un dictionnaire thésaurus utilise un *sous-dictionnaire* (qui est spécifié dans la configuration du dictionnaire) pour normaliser le texte en entrée avant la vérification des correspondances de phrases. Un seul sous-dictionnaire est sélectionnable. Une erreur est renvoyée si le sous-dictionnaire échoue dans la reconnaissance d'un mot. Dans ce cas, vous devez supprimer l'utilisation du mot ou le faire connaître au sous-dictionnaire. Vous pouvez placer un astérisque (`*`) devant un mot indexé pour ignorer l'utilisation du sous-dictionnaire, mais tous les mots *doivent* être connus du sous-dictionnaire.

Le dictionnaire thésaurus choisit la plus grande correspondance s'il existe plusieurs phrases correspondant à l'entrée.

Les mots spécifiques reconnus par le sous-dictionnaire ne peuvent pas être précisés ; à la place, utilisez `?` pour marquer tout emplacement où un terme courant peut apparaître. Par exemple, en supposant que `a` et `the` soient des termes courants d'après le sous-dictionnaire :

```
? one ? two : swsw
```

correspond à `a one the two` et à `the one a two`. Les deux pourraient être remplacés par `swsw`.

Comme un dictionnaire thésaurus a la possibilité de reconnaître des phrases, il doit se rappeler son état et interagir avec l'analyseur. Un dictionnaire thésaurus utilise ces assignations pour vérifier s'il doit gérer le mot suivant ou arrêter l'accumulation. Le dictionnaire thésaurus doit être configuré avec attention. Par exemple, si le dictionnaire thésaurus s'occupe seulement du type de jeton `asciiword`, alors une définition du dictionnaire thésaurus comme `one 7` ne fonctionnera pas, car le type de jeton `uint` n'est pas affecté au dictionnaire thésaurus.

> **Attention**
>
> Les thésaurus sont utilisés lors des indexages pour que toute modification dans les paramètres du dictionnaire thésaurus *nécessite* un réindexage. Pour la plupart des autres types de dictionnaires, de petites modifications comme l'ajout ou la suppression de termes courants ne demandent pas un réindexage.

12.6.4.1. Configuration du thésaurus

Pour définir un nouveau dictionnaire thésaurus, utilisez le modèle `thesaurus`. Par exemple :

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

Dans ce cas :

- `thesaurus_simple` est le nom du nouveau dictionnaire
- `mythesaurus` est le nom de base du fichier de configuration du thésaurus. (Son nom complet est `$SHAREDIR/tsearch_data/mythesaurus.ths`, où `$SHAREDIR` est remplacé par le répertoire des données partagées de l'installation.)
- `pg_catalog.english_stem` est le sous-dictionnaire (ici un stemmer Snowball anglais) à utiliser pour la normalisation du thésaurus. Notez que le sous-dictionnaire aura sa propre configuration (par exemple, les termes courants) qui n'est pas affichée ici.

Maintenant, il est possible de lier le dictionnaire du thésaurus `thesaurus_simple` aux types de jetons désirés dans une configuration, par exemple :

```
ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart WITH
 thesaurus_simple;
```

12.6.4.2. Exemple de thésaurus

Considérez un thésaurus d'astronomie `thesaurus_astro`, contenant quelques combinaisons de mots d'astronomie :

```
supernovae stars : sn
crab nebulae : crab
```

Ci-dessous, nous créons un dictionnaire et lions certains types de jetons à un thésaurus d'astronomie et à un stemmer anglais :

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
```

```
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart WITH
 thesaurus_astro, english_stem;
```

Maintenant, nous pouvons voir comment cela fonctionne. `ts_lexize` n'est pas très utile pour tester un thésaurus, car elle traite l'entrée en tant que simple jeton. À la place, nous pouvons utiliser `plainto_tsquery` et `to_tsvector` qui cassera les chaînes en entrée en plusieurs jetons :

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

En principe, il est possible d'utiliser `to_tsquery` si vous placez l'argument entre guillemets :

```
SELECT to_tsquery('''supernova star''');
 to_tsquery
------------
 'sn'
```

Notez que `supernova  star` établit une correspondance avec `supernovae  stars` dans `thesaurus_astro`, car nous avions indiqué le stemmer `english_stem` dans la définition du thésaurus. Le stemmer a supprimé `e` et `s`.

Pour indexer la phrase originale ainsi que son substitut, incluez-le dans la partie droite de la définition :

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
       plainto_tsquery
-----------------------------
 'sn' & 'supernova' & 'star'
```

12.6.5. Dictionnaire Ispell

Le modèle de dictionnaire Ispell ajoute le support des *dictionnaires morphologiques* qui peuvent normaliser plusieurs formes linguistiques différentes d'un mot en un même lexème. Par exemple, un dictionnaire Ispell anglais peut établir une correspondance avec toutes les déclinaisons et conjugaisons du terme `bank`, c'est-à-dire `banking`, `banked`, `banks`, `banks'` et `bank's`.

La distribution standard de PostgreSQL n'inclut aucun des fichiers de configuration Ispell. Les dictionnaires sont disponibles pour un grand nombre de langues à partir du site web Ispell[3]. De plus,

[3] https://www.cs.hmc.edu/~geoff/ispell.html

certains formats de fichiers dictionnaires plus modernes sont supportés -- MySpell[4] (OO < 2.0.1) et Hunspell[5] (OO >= 2.0.2). Une large liste de dictionnaires est disponible sur le Wiki d'OpenOffice [6].

Pour créer un dictionnaire Ispell, réalisez les étapes suivantes :

- télécharger les fichiers de configuration de dictionnaires. Ces fichiers OpenOffice portent l'extension `.oxt`. Il est nécessaire d'extraire les fichiers `.aff` et `.dic`, et de changer ces extensions en `.affix` et `.dict`. Pour certains fichiers de dictionnaire, il faut aussi convertir les caractères en encodage UTF-8 avec les commandes suivantes (par exemple, pour le dictionnaire du norvégien) :

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

- copier les fichiers dans le répertoire `$SHAREDIR/tsearch_data`
- charger les fichiers dans PostgreSQL avec la commande suivante :

```
CREATE TEXT SEARCH DICTIONARY english_hunspell (
     TEMPLATE = ispell,
    DictFile = en_us,
    AffFile = en_us,
    Stopwords = english);
```

Ici, `DictFile`, `AffFile` et `StopWords` indiquent les noms de base des fichiers dictionnaire, affixes et termes courants. Le fichier des termes courants a le même format qu'indiqué ci-dessus pour le type de dictionnaire `simple`. Le format des autres fichiers n'est pas indiqué ici, mais est disponible sur les sites web mentionnés ci-dessus.

Les dictionnaires Ispell reconnaissent habituellement un ensemble limité de mots, pour qu'ils puissent être suivis par un dictionnaire encore plus généraliste ; par exemple un dictionnaire Snowball qui reconnaît tout.

Le fichier `.affix` de Ispell suit la structure suivante :

```
prefixes
flag *A:
    .           >   RE      # Comme dans enter > reenter
suffixes
flag T:
    E           >   ST      # Comme dans late > latest
    [^AEIOU]Y   >   -Y,IEST # Comme dans dirty > dirtiest
    [AEIOU]Y    >   EST     # Comme dans gray > grayest
    [^EY]       >   EST     # Comme dans small > smallest
```

Et le fichier `.dict` suit la structure suivante :

```
lapse/ADGRS
lard/DGRS
large/PRTY
lark/MRS
```

[4] https://en.wikipedia.org/wiki/MySpell
[5] https://hunspell.github.io/
[6] https://wiki.openoffice.org/wiki/Dictionaries

Le format du fichier `.dict` est :

```
basic_form/affix_class_name
```

Dans le fichier `.affix`, chaque règle d'affixe est décrite dans le format suivant :

```
condition > [-stripping_letters,] adding_affix
```

Ici, une condition a un format similaire au format des expressions régulières. Elle peut comporter des groupements `[...]` et `[^...]`. Par exemple, `[AEIOU]Y` signifie que la dernière lettre du mot est `"y"` et que l'avant-dernière lettre est `"a"`, `"e"`, `"i"`, `"o"` ou `"u"`. `[^EY]` signifie que la dernière lettre n'est ni `"e"` ni `"y"`.

Les dictionnaires Ispell supportent la séparation des mots composés, une fonctionnalité intéressante. Notez que le fichier d'affixes doit indiquer une option spéciale qui marque les mots du dictionnaire qui peuvent participer à une formation composée :

```
compoundwords  controlled z
```

Voici quelques exemples en norvégien :

```
SELECT ts_lexize('norwegian_ispell',
 'overbuljongterningpakkmesterassistent');
   {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
   {sjokoladefabrikk,sjokolade,fabrikk}
```

Le format MySpell est un sous-ensemble du format Hunspell. Le fichier `.affix` de Hunspell suit la structure suivante :

```
PFX A Y 1
PFX A   0     re         .
SFX T N 4
SFX T   0     st         e
SFX T   y     iest       [^aeiou]y
SFX T   0     est        [aeiou]y
SFX T   0     est        [^ey]
```

La première ligne d'une classe d'affixe est l'en-tête. Les champs des règles d'affixes sont listés après l'en-tête.

- nom du paramètre (PFX ou SFX)
- flag (nom de la classe d'affixe)
- éliminer les caractères au début (au préfixe) ou à la fin (au suffixe) du mot
- ajouter l'affixe
- condition ayant un format similaire à celui des expressions régulières.

Le fichier `.dict` ressemble au fichier `.dict` de Ispell :

```
larder/M
```

```
lardy/RT
large/RSPMYT
largehearted
```

> **Note**
>
> MySpell ne supporte pas les mots composés. Hunspell a un support sophistiqué des mots composés. Actuellement, PostgreSQL implémente seulement les opérations basiques de Hunspell pour les mots composés.

12.6.6. Dictionnaire Snowball

Le modèle de dictionnaire Snowball est basé sur le projet de Martin Porter, inventeur du populaire algorithme stemming de Porter pour l'anglais. Snowball propose maintenant des algorithmes stemming pour un grand nombre de langues (voir le site Snowball[7] pour plus d'informations). Chaque algorithme sait comment réduire les variantes standard d'un mot vers une base, ou stem, en rapport avec la langue. Un dictionnaire Snowball réclame un paramètre `langue` pour identifier le stemmer à utiliser et, en option, un nom de fichier des `termes courants` donnant une liste de mots à éliminer. (Les listes de termes courants au standard PostgreSQL sont aussi fournies par le projet Snowball.) Par exemple, il existe un équivalent de la définition interne en

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

Le format du fichier des termes courants est identique à celui déjà expliqué.

Un dictionnaire Snowball reconnaît tout, qu'il soit ou non capable de simplifier le mot, donc il doit être placé en fin de la liste des dictionnaires. Il est inutile de l'avoir avant tout autre dictionnaire, car un jeton ne passera jamais au prochain dictionnaire.

12.7. Exemple de configuration

Une configuration de recherche plein texte précise toutes les options nécessaires pour transformer un document en un `tsvector` : le planificateur à utiliser pour diviser le texte en jetons, et les dictionnaires à utiliser pour transformer chaque jeton en un lexème. Chaque appel à `to_tsvector` ou `to_tsquery` a besoin d'une configuration de recherche plein texte pour réaliser le traitement. Le paramètre de configuration default_text_search_config indique le nom de la configuration par défaut, celle utilisée par les fonctions de recherche plein texte si un paramètre explicite de configuration est oublié. Il se configure soit dans `postgresql.conf` soit dans une session individuelle en utilisant la commande `SET`.

Plusieurs configurations de recherche plein texte prédéfinies sont disponibles et vous pouvez créer des versions personnalisées facilement. Pour faciliter la gestion des objets de recherche plein texte, un ensemble de commandes SQL est disponible, et il existe plusieurs commandes psql affichant des informations sur les objets de la recherche plein texte (Section 12.10).

Comme exemple, nous allons créer une configuration `pg` en commençant à partir d'une duplication de la configuration `english`.

[7] https://snowballstem.org/

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY =
 pg_catalog.english );
```

Nous allons utiliser une liste de synonymes spécifique à PostgreSQL et nous allons la stocker dans `$SHAREDIR/tsearch_data/pg_dict.syn`. Le contenu du fichier ressemble à ceci :

```
postgres    pg
pgsql       pg
postgresql  pg
```

Nous définissons le dictionnaire des synonymes ainsi :

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

Ensuite, nous enregistrons le dictionnaire Ispell `english_ispell` qui a ses propres fichiers de configuration :

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

Maintenant, nous configurons la correspondance des mots dans la configuration `pg` :

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

Nous choisissons de ne pas indexer certains types de jeton que la configuration par défaut peut gérer :

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

Maintenant, nous pouvons tester notre configuration :

```
SELECT * FROM ts_debug('public.pg', '
PostgreSQL, the highly scalable, SQL compliant, open source object-
relational
database management system, is now undergoing beta testing of the
 next
```

```
version of our software.
');
```

La prochaine étape est d'initialiser la session pour utiliser la nouvelle configuration qui était créée dans le schéma `public` :

```
=> \dF
   List of text search configurations
 Schema  | Name | Description
---------+------+-------------
 public  | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.pg
```

12.8. Tester et déboguer la recherche plein texte

Le comportement d'une configuration personnalisée de recherche plein texte peut facilement devenir confus. Les fonctions décrites dans cette section sont utiles pour tester les objets de recherche plein texte. Vous pouvez tester une configuration complète ou tester séparément analyseurs et dictionnaires.

12.8.1. Test d'une configuration

La fonction `ts_debug` permet un test facile d'une configuration de recherche plein texte.

```
ts_debug([ config regconfig, ] document text,
         OUT alias text,
         OUT description text,
         OUT token text,
         OUT dictionaries regdictionary[],
         OUT dictionary regdictionary,
         OUT lexemes text[])
         returns setof record
```

`ts_debug` affiche des informations sur chaque jeton d'un *`document`* tel qu'il est produit par l'analyseur et traité par les dictionnaires configurés. Elle utilise la configuration indiquée par *`config`*, ou `default_text_search_config` si cet argument est omis.

`ts_debug` renvoie une ligne pour chaque jeton identifié dans le texte par l'analyseur. Les colonnes renvoyées sont :

- *`alias`* `text` -- nom court du type de jeton
- *`description`* `text` -- description du type de jeton
- *`token`* `text` -- texte du jeton

- *dictionaries* `regdictionary[]` -- les dictionnaires sélectionnés par la configuration pour ce type de jeton
- *dictionary* `regdictionary` -- le dictionnaire qui a reconnu le jeton, ou `NULL` dans le cas contraire
- *lexemes* `text[]` -- le ou les lexèmes produits par le dictionnaire qui a reconnu le jeton, ou `NULL` dans le cas contraire ; un tableau vide (`{}`) signifie qu'il a été reconnu comme un terme courant

Voici un exemple simple :

```
SELECT * FROM ts_debug('english', 'a fat  cat sat on a mat - it ate
 a fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary
  | lexemes
-----------+-----------------+-------+----------------
+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} |
 english_stem | {}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | fat   | {english_stem} |
 english_stem | {fat}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | cat   | {english_stem} |
 english_stem | {cat}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | sat   | {english_stem} |
 english_stem | {sat}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | on    | {english_stem} |
 english_stem | {}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | a     | {english_stem} |
 english_stem | {}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | mat   | {english_stem} |
 english_stem | {mat}
 blank     | Space symbols   |       | {}             |
  |
 blank     | Space symbols   | -     | {}             |
  |
 asciiword | Word, all ASCII | it    | {english_stem} |
 english_stem | {}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | ate   | {english_stem} |
 english_stem | {ate}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | a     | {english_stem} |
 english_stem | {}
 blank     | Space symbols   |       | {}             |
  |
```

```
 asciiword | Word, all ASCII | fat   | {english_stem} |
 english_stem | {fat}
 blank     | Space symbols   |       | {}             |
  |
 asciiword | Word, all ASCII | rats  | {english_stem} |
 english_stem | {rat}
```

Pour une démonstration plus importante, nous créons tout d'abord une configuration `public.english` et un dictionnaire ispell pour l'anglais :

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY =
 pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;



SELECT * FROM ts_debug('public.english', 'The Brightest
 supernovaes');
   alias   |   description   |    token    |         dictionaries
        |   dictionary   |   lexemes
-----------+-----------------+-------------
+-------------------------------+----------------+-------------
 asciiword | Word, all ASCII | The         |
 {english_ispell,english_stem} | english_ispell | {}
 blank     | Space symbols   |             | {}
       |                |
 asciiword | Word, all ASCII | Brightest   |
 {english_ispell,english_stem} | english_ispell | {bright}
 blank     | Space symbols   |             | {}
       |                |
 asciiword | Word, all ASCII | supernovaes |
 {english_ispell,english_stem} | english_stem   | {supernova}
```

Dans cet exemple, le mot `Brightest` a été reconnu par l'analyseur comme un `mot ASCII` (alias `asciiword`). Pour ce type de jeton, la liste de dictionnaire est `english_ispell` et `english_stem`. Le mot a été reconnu par `english_ispell`, qui l'a réduit avec le mot `bright`. Le mot `supernovaes` est inconnu dans le dictionnaire `english_ispell` donc il est passé au dictionnaire suivant et, heureusement, est reconnu (en fait, `english_stem` est un dictionnaire Snowball qui reconnaît tout ; c'est pourquoi il est placé en dernier dans la liste des dictionnaires).

Le mot `The` est reconnu par le dictionnaire `english_ispell` comme étant un terme courant (Section 12.6.1) et n'est donc pas indexé. Les espaces sont aussi ignorées, car la configuration ne fournit aucun dictionnaire pour eux.

Vous pouvez réduire le volume en sortie en spécifiant explicitement les colonnes que vous voulez voir :

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias   |    token    |   dictionary   |   lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

12.8.2. Test de l'analyseur

Les fonctions suivantes permettent un test direct d'un analyseur de recherche plein texte.

```
   ts_parse(parser_name text, document text, OUT tokid integer,
 OUT token text) returns setof record
   ts_parse(parser_oid oid, document text, OUT tokid integer,
 OUT token text) returns setof record
```

`ts_parse` analyse le *`document`* indiqué et renvoie une série d'enregistrements, un pour chaque jeton produit par l'analyse. Chaque enregistrement inclut un `tokid` montrant le type de jeton affecté et un jeton (`token`) qui est le texte dudit jeton. Par exemple :

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
   ts_token_type(parser_name text, OUT tokid integer,
 OUT alias text, OUT description text) returns setof record
   ts_token_type(parser_oid oid, OUT tokid integer, OUT alias text,
 OUT description text) returns setof record
```

`ts_token_type` renvoie une table qui décrit chaque type de jeton que l'analyseur indiqué peut reconnaître. Pour chaque type de jeton, la table donne l'entier `tokid` que l'analyseur utilise pour labeliser un jeton de ce type, l'`alias` qui nomme le type de jeton dans les commandes de configuration et une courte `description`. Par exemple :

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------
+------------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
```

```
 3 | numword         | Word, letters and digits
 4 | email           | Email address
 5 | url             | URL
 6 | host            | Host
 7 | sfloat          | Scientific notation
 8 | version         | Version number
 9 | hword_numpart   | Hyphenated word part, letters and digits
10 | hword_part      | Hyphenated word part, all letters
11 | hword_asciipart | Hyphenated word part, all ASCII
12 | blank           | Space symbols
13 | tag             | XML tag
14 | protocol        | Protocol head
15 | numhword        | Hyphenated word, letters and digits
16 | asciihword      | Hyphenated word, all ASCII
17 | hword           | Hyphenated word, all letters
18 | url_path        | URL path
19 | file            | File or path name
20 | float           | Decimal notation
21 | int             | Signed integer
22 | uint            | Unsigned integer
23 | entity          | XML entity
```

12.8.3. Test des dictionnaires

La fonction `ts_lexize` facilite le test des dictionnaires.

```
ts_lexize(dict regdictionary, token text) returns text[]
```

`ts_lexize` renvoie un tableau de lexèmes si le jeton (*`token`*) en entrée est connu du dictionnaire ou un tableau vide si le jeton est connu du dictionnaire en tant que terme courant, ou enfin `NULL` si le mot n'est pas connu.

Exemples :

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

> **Note**
>
> La fonction `ts_lexize` attend un seul jeton, pas du texte. Voici un cas où cela peut devenir source de confusion :
>
> ```
> SELECT ts_lexize('thesaurus_astro', 'supernovae stars') is
>  null;
> ```

```
 ?column?
----------
 t
```

Le dictionnaire thésaurus `thesaurus_astro` connaît la phrase `supernovae stars`, mais `ts_lexize` échoue, car il ne peut pas analyser le texte en entrée, mais le traite bien en tant que simple jeton. Utilisez `plainto_tsquery` ou `to_tsvector` pour tester les dictionnaires thésaurus. Par exemple :

```
SELECT plainto_tsquery('supernovae stars');
 plainto_tsquery
-----------------
 'sn'
```

12.9. Types d'index préférés pour la recherche plein texte

Il existe deux types d'index qui peuvent être utilisés pour accélérer les recherches plein texte : GIN et GiST. Notez que les index ne sont pas obligatoires pour la recherche plein texte, mais, dans les cas où une colonne est utilisée fréquemment dans une recherche, un index sera suffisamment intéressant.

Pour créer un tel index, faites l'un des deux :

```
CREATE INDEX nom ON table USING GIN(colonne);
```

Crée un index GIN (Generalized Inverted Index). La *`colonne`* doit être de type `tsvector`.

```
CREATE INDEX nom ON table USING GIST (colonne [ { DEFAULT | tsvector_ops } (siglen = nombre) ] );
```

Crée un index GiST (Generalized Search Tree). La *`colonne`* peut être de type `tsvector` ou `tsquery`. Le paramètre entier optionnel `siglen` détermine la longueur de la signature en octets (voir ci-dessous pour les détails).

Les index GIN sont le type d'index préféré pour la recherche plein texte. En tant qu'index inversés, ils contiennent une entrée d'index pour chaque mot (lexème), avec une liste compressée des emplacements correspondants. Les recherches multimots peuvent trouver la première correspondance, puis utiliser l'index pour supprimer les lignes qui ne disposent pas des autres mots recherchés. Les index GIN stockent uniquement les mots (lexèmes) des valeurs de type `tsvector`, et non pas les labels de poids. De ce fait, une vérification de la ligne de table est nécessaire quand une recherche implique les poids.

Un index GiST est *à perte*, signifiant que l'index peut produire des faux positifs, et il est nécessaire de vérifier la ligne de la table pour les éliminer. PostgreSQL le fait automatiquement si nécessaire. Les index GiST sont à perte, car chaque document est représenté dans l'index par une signature à longueur fixe. La longueur de la signature en octets est déterminée par la valeur du paramètre optionnel de type entier nommé `siglen`. La longueur par défaut de la signature (quand `siglen` n'est pas indiqué) est 124 octets, la longueur maximale est de 2024 octets. La signature est générée par le hachage de chaque mot en un bit aléatoire dans une chaîne à n bits, tous ces bits étant assemblés dans une opération OR qui produit une signature du document sur n bits. Quand deux hachages de mots sont identiques, nous avons un faux positif. Si tous les mots de la requête ont une correspondance (vraie ou fausse), alors la ligne de la table doit être récupérée pour voir si la correspondance est correcte. Des signatures plus longues amènent à une recherche plus précise (en parcourant une fraction plus réduite de l'index et un plus petit nombre de blocs dans la table), au prix d'un index plus gros.

Un index GiST peut être couvrant, c'est-à-dire utiliser la clause `INCLUDE`. Les colonnes incluses peuvent avoir des types de données sans aucune classe d'opérateurs GiST. Les attributs inclus seront stockés sans compression.

La perte implique une dégradation des performances à cause de récupérations inutiles d'enregistrements de la table qui s'avèrent être de fausses correspondances. Comme les accès aléatoires aux enregistrements de la table sont lents, ceci limite l'utilité des index GiST. La probabilité de faux positifs dépend de plusieurs facteurs, en particulier le nombre de mots uniques, donc l'utilisation de dictionnaires qui réduisent ce nombre est recommandée.

Notez que le temps de construction de l'index GIN peut souvent être amélioré en augmentant maintenance_work_mem alors qu'un index GiST n'est pas sensible à ce paramètre.

Le partitionnement de gros ensembles et l'utilisation intelligente des index GIN et GiST autorise l'implémentation de recherches très rapides avec une mise à jour en ligne. Le partitionnement peut se faire au niveau de la base en utilisant l'héritage, ou en distribuant les documents sur des serveurs et en récupérant les résultats de la recherche, par exemple via un accès Foreign Data. Ce dernier est possible, car les fonctions de score utilisent les informations locales.

12.10. Support de psql

Des informations sur les objets de configuration de la recherche plein texte peuvent être obtenues dans psql en utilisant l'ensemble de commandes :

```
\dF{d,p,t}[+] [MODÈLE]
```

Un + supplémentaire affiche plus de détails.

Le paramètre optionnel *`MODÈLE`* doit être le nom d'un objet de la recherche plein texte, pouvant être qualifié du nom du schéma. Si *`MODÈLE`* est omis, alors l'information sur tous les objets visibles est affichée. *`MODÈLE`* peut être une expression rationnelle et peut fournir des modèles *séparés* pour les noms du schéma et de l'objet. Les exemples suivants illustrent ceci :

```
=> \dF *fulltext*
       List of text search configurations
 Schema |  Name        | Description
--------+--------------+-------------
 public | fulltext_cfg |


=> \dF *.fulltext*
       List of text search configurations
 Schema   |  Name        | Description
----------+----------------------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

Les commandes suivantes sont :

`\dF[+] [MODÈLE]`

Liste les configurations de recherche plein texte (ajouter + pour plus de détails).

```
=> \dF russian
            List of text search configurations
```

```
   Schema   |  Name   |            Description
------------+---------+-----------------------------------
 pg_catalog | russian | configuration for russian language

=> \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | russian_stem
```

`\dFd[+] [MODÈLE]`

Liste les dictionnaires de recherche plein texte (ajouter + pour plus de détails).

```
=> \dFd
                              List of text search dictionaries
   Schema   |      Name       |
 Description
------------+-----------------
+------------------------------------------------------------
 pg_catalog | arabic_stem     | snowball stemmer for arabic
 language
 pg_catalog | armenian_stem   | snowball stemmer for armenian
 language
 pg_catalog | basque_stem     | snowball stemmer for basque
 language
 pg_catalog | catalan_stem    | snowball stemmer for catalan
 language
 pg_catalog | danish_stem     | snowball stemmer for danish
 language
 pg_catalog | dutch_stem      | snowball stemmer for dutch
 language
 pg_catalog | english_stem    | snowball stemmer for english
 language
 pg_catalog | finnish_stem    | snowball stemmer for finnish
 language
 pg_catalog | french_stem     | snowball stemmer for french
 language
```

```
 pg_catalog | german_stem     | snowball stemmer for german
 language
 pg_catalog | greek_stem      | snowball stemmer for greek
 language
 pg_catalog | hindi_stem      | snowball stemmer for hindi
 language
 pg_catalog | hungarian_stem  | snowball stemmer for hungarian
 language
 pg_catalog | indonesian_stem | snowball stemmer for indonesian
 language
 pg_catalog | irish_stem      | snowball stemmer for irish
 language
 pg_catalog | italian_stem    | snowball stemmer for italian
 language
 pg_catalog | lithuanian_stem | snowball stemmer for lithuanian
 language
 pg_catalog | nepali_stem     | snowball stemmer for nepali
 language
 pg_catalog | norwegian_stem  | snowball stemmer for norwegian
 language
 pg_catalog | portuguese_stem | snowball stemmer for portuguese
 language
 pg_catalog | romanian_stem   | snowball stemmer for romanian
 language
 pg_catalog | russian_stem    | snowball stemmer for russian
 language
 pg_catalog | serbian_stem    | snowball stemmer for serbian
 language
 pg_catalog | simple          | simple dictionary: just lower
 case and check for stopword
 pg_catalog | spanish_stem    | snowball stemmer for spanish
 language
 pg_catalog | swedish_stem    | snowball stemmer for swedish
 language
 pg_catalog | tamil_stem      | snowball stemmer for tamil
 language
 pg_catalog | turkish_stem    | snowball stemmer for turkish
 language
 pg_catalog | yiddish_stem    | snowball stemmer for yiddish
 language
```

`\dFp[+] [MODÈLE]`

Liste les analyseurs de recherche plein texte (ajouter + pour plus de détails).

```
=> \dFp
        List of text search parsers
   Schema   |  Name   |     Description
------------+---------+---------------------
 pg_catalog | default | default word parser
=> \dFp+
    Text search parser "pg_catalog.default"
     Method      |    Function    | Description
-----------------+----------------+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
 End parse       | prsd_end       |
```

```
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

        Token types for parser "pg_catalog.default"
   Token name    |               Description
-----------------+------------------------------------------
 asciihword      | Hyphenated word, all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
 file            | File or path name
 float           | Decimal notation
 host            | Host
 hword           | Hyphenated word, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
 int             | Signed integer
 numhword        | Hyphenated word, letters and digits
 numword         | Word, letters and digits
 protocol        | Protocol head
 sfloat          | Scientific notation
 tag             | HTML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
 version         | Version number
 word            | Word, all letters
(23 rows)
```

`\dFt[+] [MODÈLE]`

Liste les modèles de recherche plein texte (ajouter + pour plus de détails).

```
=> \dFt
                            List of text search templates
   Schema   |   Name    |                        Description
------------+-----------
+-----------------------------------------------------------
 pg_catalog | ispell    | ispell dictionary
 pg_catalog | simple    | simple dictionary: just lower case and
 check for stopword
 pg_catalog | snowball  | snowball stemmer
 pg_catalog | synonym   | synonym dictionary: replace word by
 its synonym
 pg_catalog | thesaurus | thesaurus dictionary: phrase by phrase
 substitution
```

12.11. Limites

Les limites actuelles de la recherche plein texte de PostgreSQL sont :

- La longueur de chaque lexème doit être inférieure à 2 ko
- La longueur d'un `tsvector` (lexèmes + positions) doit être inférieure à 1 Mo

- Le nombre de lexèmes doit être inférieur à 2^{64}
- Les valeurs de position dans un `tsvector` doivent être supérieures à 0 et inférieures ou égales à 16383
- La distance de correspondance dans un opérateur `tsquery` <*N*> (FOLLOWED BY) ne peut pas dépasser 16384
- Pas plus de 256 positions par lexème
- Le nombre de nœuds (lexèmes + opérateurs) dans un `tsquery` doit être inférieur à 32768

Pour comparaison, la documentation de PostgreSQL 8.1 contient 10441 mots uniques, un total de 335420 mots, et le mot le plus fréquent, « postgresql », est mentionné 6127 fois dans 655 documents.

Un autre exemple -- les archives de la liste de discussions de PostgreSQL contenaient 910989 mots uniques avec 57491343 lexèmes dans 461020 messages.

Chapitre 13. Contrôle d'accès simultané

Ce chapitre décrit le comportement de PostgreSQL lorsque deux sessions, ou plus essaient d'accéder aux mêmes données au même moment. Le but dans cette situation est de permettre un accès efficace pour toutes les sessions tout en maintenant une intégrité stricte des données. Chaque développeur d'applications utilisant des bases de données doit avoir une bonne compréhension des thèmes couverts dans ce chapitre.

13.1. Introduction

PostgreSQL fournit un ensemble d'outils pour les développeurs qui souhaitent gérer des accès simultanés aux données. En interne, la cohérence des données est obtenue avec l'utilisation d'un modèle multiversion (Multiversion Concurrency Control, MVCC). Cela signifie que chaque requête SQL voit une image des données (une *version de la base de données*) telle qu'elles étaient quelques temps auparavant, quel que soit l'état actuel des données sous-jacentes. Cela évite que les requêtes puissent voir des données non cohérentes produites par des transactions concurrentes effectuant des mises à jour sur les mêmes lignes de données, fournissant ainsi une *isolation des transactions* pour chaque session de la base de données. MVCC, en évitant les méthodes des verrous des systèmes de bases de données traditionnels, minimise la durée des verrous pour permettre des performances raisonnables dans des environnements multiutilisateurs.

Le principal avantage de l'utilisation du modèle MVCC pour le contrôle des accès simultanés, contrairement au verrouillage, est que, dans les verrous acquis par MVCC pour récupérer (en lecture) des données, aucun conflit n'intervient avec les verrous acquis pour écrire des données. Du coup, lire ne bloque jamais l'écriture et écrire ne bloque jamais la lecture. PostgreSQL maintient cette garantie même quand il fournit le niveau d'isolation le plus strict au moyen d'un niveau *Serializable Snapshot Isolation* (SSI) innovant.

Des possibilités de verrouillage des tables ou des lignes sont aussi disponibles dans PostgreSQL pour les applications qui n'ont pas besoin en général d'une isolation complète des transactions et préfèrent gérer explicitement les points de conflits particuliers. Néanmoins, un bon usage de MVCC fournira généralement de meilleures performances que les verrous. De plus, les verrous informatifs définis par l'utilisateur fournissent un mécanisme d'acquisition de verrous qui n'est pas lié à une transaction.

13.2. Isolation des transactions

Le standard SQL définit quatre niveaux d'isolation de transaction. Le plus strict est Serializable, qui est défini par le standard dans un paragraphe qui déclare que toute exécution concurrente d'un jeu de transactions sérialisables doit apporter la garantie de produire le même effet que l'exécution consécutive de chacune d'entre elles dans un certain ordre. Les trois autres niveaux sont définis en termes de phénomènes résultant de l'interaction entre les transactions concurrentes, qui ne doivent pas se produire à chaque niveau. Le standard note qu'en raison de la définition de Serializable, aucun de ces phénomènes n'est possible à ce niveau. (Cela n'a rien de surprenant -- si l'effet des transactions doit être cohérent avec l'exécution consécutive de chacune d'entre elles, comment pourriez-vous voir un phénomène causé par des interactions?).

Les phénomènes qui sont interdits à chaque niveau sont:

lecture sale

Une transaction lit des données écrites par une transaction concurrente non validée (dirty read).

lecture non reproductible

Une transaction relit des données qu'elle a lues précédemment et trouve que les données ont été modifiées par une autre transaction (validée depuis la lecture initiale) (non repeatable read).

lecture fantôme

Une transaction réexécute une requête renvoyant un ensemble de lignes satisfaisant une condition de recherche et trouve que l'ensemble des lignes satisfaisant la condition a changé du fait d'une autre transaction récemment validée (phantom read).

anomalie de sérialisation

Le résultat de la validation réussie d'un groupe de transactions est incohérent avec tous les ordres possibles d'exécutions de ces transactions, une par une.

Les niveaux d'isolation des transactions proposés par le standard SQL et implémentés par PostgreSQL sont décrits dans le Tableau 13.1.

Tableau 13.1. Niveaux d'isolation des transactions

Niveau d'isolation	Lecture sale	Lecture non reproductible	Lecture fantôme	Anomalie de sérialisation
Read Uncommited (en français, « Lecture de données non validées »)	Autorisé, mais pas dans PostgreSQL	Possible	Possible	Possible
Read Commited (en français, « Lecture de données validées »)	Impossible	Possible	Possible	Possible
Repeatable Read (en français, « Lecture répétée »)	Impossible	Impossible	Autorisé, mais pas dans PostgreSQL	Possible
Serializable (en français, « Sérialisable »)	Impossible	Impossible	Impossible	Impossible

Dans PostgreSQL, vous pouvez demander un des quatre niveaux standards d'isolation des transactions, mais seuls trois niveaux distincts sont implémentés (le mode Read Uncommited de PostgreSQL se comporte comme le mode Read Commited). Ceci est dû au fait qu'il s'agit de la seule façon logique de faire correspondre les niveaux d'isolation standards à l'architecture de contrôle de la concurrence de PostgreSQL.

Le tableau montre aussi que l'implémentation Repeatable Read de PostgreSQL n'autorise pas les lectures fantômes. C'est acceptable d'après le standard SQL car le standard SQL indique les anomalies qui ne doivent *pas* survenir à certains niveaux d'isolation ; des garanties plus fortes sont acceptables. Le comportement des niveaux d'isolation disponibles est détaillé dans les sous-sections suivantes.

Pour initialiser le niveau d'isolation d'une transaction, utilisez la commande SET TRANSACTION.

> **Important**
>
> Certains types de données et certaines fonctions de PostgreSQL ont des règles spéciales sur le comportement des transactions. En particulier, les modifications réalisées sur une séquence (et du coup sur le compteur d'une colonne déclarée `serial`) sont immédiatement visibles de toutes les autres transactions et ne sont pas annulées si la transaction qui a fait la modification est annulée. Voir Section 9.17 et Section 8.1.4.

13.2.1. Niveau d'isolation Read committed (lecture uniquement des données validées)

Read Committed est le niveau d'isolation par défaut dans PostgreSQL. Quand une transaction utilise ce niveau d'isolation, une requête SELECT (sans clause FOR UPDATE/SHARE) voit seulement les données validées avant le début de la requête ; il ne voit jamais les données non validées et les modifications validées pendant l'exécution de la requête par des transactions exécutées en parallèle. En effet, une requête SELECT voit une image de la base de données datant du moment où l'exécution de la requête commence. Néanmoins, SELECT voit les effets de mises à jour précédentes exécutées dans sa propre transaction, même si celles-ci n'ont pas encore été validées. De plus, notez que deux commandes SELECT successives peuvent voir des données différentes, même si elles sont exécutées dans la même transaction si d'autres transactions valident des modifications après que le premier SELECT a démarré et avant que le second SELECT ne commence.

Les commandes UPDATE, DELETE, SELECT FOR UPDATE et SELECT FOR SHARE se comportent de la même façon que SELECT en ce qui concerne la recherche des lignes cibles : elles ne trouveront que les lignes cibles qui ont été validées avant le début de la commande. Néanmoins, une telle ligne cible pourrait avoir déjà été mise à jour (ou supprimée ou verrouillée) par une autre transaction concurrente au moment où elle est découverte. Dans ce cas, le processus de mise à jour attendra que la première transaction soit validée ou annulée (si elle est toujours en cours). Si la première mise à jour est annulée, alors ses effets sont niés et le deuxième processus peut exécuter la mise à jour des lignes originellement trouvées. Si la première mise à jour est validée, la deuxième mise à jour ignorera la ligne si la première mise à jour l'a supprimée, sinon elle essaiera d'appliquer son opération à la version mise à jour de la ligne. La condition de la recherche de la commande (la clause WHERE) est réévaluée pour savoir si la version mise à jour de la ligne correspond toujours à la condition de recherche. Dans ce cas, la deuxième mise à jour continue son opération en utilisant la version mise à jour de la ligne. Dans le cas des commandes SELECT FOR UPDATE et SELECT FOR SHARE, cela signifie que la version mise à jour de la ligne est verrouillée et renvoyée au client.

INSERT avec une clause ON CONFLICT DO UPDATE se comporte de la même façon. Dans le mode Read Committed, chaque ligne proposée à l'insertion sera soit insérée soit mise à jour. Sauf s'il y a des erreurs sans rapport, une des deux solutions est garantie. Si un conflit survient d'une autre transaction dont les effets ne sont pas encore visibles à INSERT, la clause UPDATE affectera cette ligne, même s'il est possible qu'il n'existe *pas* de version de cette ligne visible à cette commande.

INSERT avec une clause ON CONFLICT DO NOTHING pourrait avoir une insertion non terminée pour une ligne à cause du résultat d'une autre transaction dont les effets ne sont pas visibles à l'image de base de la commande INSERT. Là encore, c'est seulement le cas du mode Read Committed.

MERGE permet à l'utilisateur d'indiquer plusieurs combinaisons de sous-commandes INSERT, UPDATE et DELETE. Une commande MERGE avec à fois des sous-commandes INSERT et UPDATE ressemble à un INSERT avec une clause ON CONFLICT DO UPDATE mais ne garantie pas que soit un INSERT soit un UPDATE arrivera. Si un MERGE tente un UPDATE ou un DELETE et que la ligne est mise à jour en même temps mais que la condition de jointure passe toujours pour la cible en cours et pour la ligne source en cours, alors MERGE se comportera de la même façon que les commandes UPDATE ou DELETE et réalisera son action sur la version mise à jour de la ligne. Néanmoins, comme MERGE peut indiquer plusieurs actions et qu'elles peuvent être conditionnelles, les conditions de chaque action sont ré-évaluées sur la version mise à jour de la ligne, dès la première action, même si l'action qui a initialement été comparée apparaît plus tard dans la liste des actions. D'un autre côté, si la ligne D'une autre façon, si la ligne est mise à jour ou supprimée en concurrence de façon à ce que la condition de jointure échoue, alors MERGE évaluera les actions NOT MATCHED de la condition ensuite, et exécutera la première qui réussit. Si MERGE tente un INSERT et qu'un index d'unicité est présent, et qu'une ligne dupliquée est insérée en même temps, alors une erreur de violation de l'unicité est levée ; MERGE ne tente pas d'éviter de telles erreurs en recommençant l'évaluation des conditions du MATCHED.

À cause des règles ci-dessus, une commande de mise à jour a la possibilité de voir une image non cohérente : elle peut voir les effets de commandes de mises à jour concurrentes sur les mêmes lignes que celles qu'elle essaie de mettre à jour, mais elle ne voit pas les effets de ces commandes sur les autres lignes de la base de données. Ce comportement rend le mode de lecture validée non convenable pour les commandes qui impliquent des conditions de recherche complexes ; néanmoins, il est intéressant pour

les cas simples. Par exemple, considérons la mise à jour de balances de banque avec des transactions comme :

```
BEGIN;
UPDATE comptes SET balance = balance + 100.00 WHERE no_compte =
 12345;
UPDATE comptes SET balance = balance - 100.00 WHERE no_compte =
 7534;
COMMIT;
```

Si deux transactions comme celle-ci essaient de modifier en même temps la balance du compte 12345, nous voulons clairement que la deuxième transaction commence à partir de la version mise à jour de la ligne du compte. Comme chaque commande n'affecte qu'une ligne prédéterminée, la laisser voir la version mise à jour de la ligne ne crée pas de soucis de cohérence.

Des utilisations plus complexes peuvent produire des résultats non désirés dans le mode Read Committed. Par exemple, considérez une commande `DELETE` opérant sur des données qui sont à la fois ajoutées et supprimées du critère de restriction par une autre commande. Supposons que `website` est une table sur deux lignes avec `website.hits` valant `9` et `10` :

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- exécuté par une autre session :  DELETE FROM website WHERE hits
 = 10;
COMMIT;
```

La commande `DELETE` n'aura pas d'effet même s'il existe une ligne `website.hits = 10` avant et après la commande `UPDATE`. Cela survient parce que la valeur `9` de la ligne avant mise à jour est ignorée et que lorsque l'`UPDATE` termine et que `DELETE` obtient un verrou, la nouvelle valeur de la ligne n'est plus `10`, mais `11`, ce qui ne correspond plus au critère.

Comme le mode Read Committed commence chaque commande avec une nouvelle image qui inclut toutes les transactions validées jusqu'à cet instant, les commandes suivantes dans la même transaction verront les effets de la transaction validée en parallèle dans tous les cas. Le problème en question est de savoir si une *seule* commande voit une vue absolument cohérente ou non de la base de données.

L'isolation partielle des transactions fournie par le mode Read Committed est adéquate pour de nombreuses applications, et ce mode est rapide et simple à utiliser. Néanmoins, il n'est pas suffisant dans tous les cas. Les applications qui exécutent des requêtes et des mises à jour complexes pourraient avoir besoin d'une vue plus rigoureusement cohérente de la base de données, une vue que le mode Read Committed ne fournit pas.

13.2.2. Repeatable Read Isolation Level

Le niveau d'isolation *Repeatable Read* ne voit que les données validées avant que la transaction ait démarré; il ne voit jamais ni les données non validées, ni les données validées par des transactions concurrentes durant son exécution. (Toutefois, la requête voit les effets de mises à jour précédentes effectuées dans sa propre transaction, bien qu'elles ne soient pas encore validées). C'est une garantie plus élevée que celle requise par le standard SQL pour ce niveau d'isolation, et elle évite le phénomène décrit dans Tableau 13.1 sauf pour les anomalies de sérialisation. Comme mentionné plus haut, c'est permis par le standard, qui ne définit que la protection *minimale* que chaque niveau d'isolation doit fournir.

Ce niveau est différent de Read Committed parce qu'une requête dans une transaction repeatable read voit un instantané au début de la *transaction*, et non pas du début de la requête en cours à l'intérieur de la transaction. Du coup, les commandes `SELECT` successives à l'intérieur d'une *seule* transaction voient toujours les mêmes données, c'est-à-dire qu'elles ne voient jamais les modifications faites par les autres transactions qui ont été validées après le début de leur propre transaction.

Les applications utilisant ce niveau d'isolation doivent être préparées à retenter des transactions à cause d'échecs de sérialisation.

Les commandes UPDATE, DELETE, MERGE, SELECT FOR UPDATE et SELECT FOR SHARE se comportent de la même façon que SELECT en ce qui concerne la recherche de lignes cibles : elles trouveront seulement les lignes cibles qui ont été validées avant le début de la transaction. Néanmoins, une telle ligne cible pourrait avoir été mise à jour (ou supprimée ou verrouillée) par une autre transaction concurrente au moment où elle est utilisée. Dans ce cas, la transaction repeatable read attendra que la première transaction de mise à jour soit validée ou annulée (si celle-ci est toujours en cours). Si la première mise à jour est annulée, les effets sont inversés et la transaction repeatable read peut continuer avec la mise à jour de la ligne trouvée à l'origine. Mais si la mise à jour est validée (et que la ligne est mise à jour ou supprimée, pas simplement verrouillée), alors la transaction repeatable read sera annulée avec le message

```
ERROR:  could not serialize access due to concurrent update
```

parce qu'une transaction sérialisable ne peut pas modifier ou verrouiller les lignes changées par d'autres transactions après que la transaction sérialisable a commencé.

Quand une application reçoit ce message d'erreurs, elle devrait annuler la transaction actuelle et réessayer la transaction complète. La seconde fois, la transaction voit les modifications déjà validées comme faisant partie de sa vue initiale de la base de données, donc il n'y a pas de conflit logique en utilisant la nouvelle version de la ligne comme point de départ pour la mise à jour de la nouvelle transaction.

Notez que seules les transactions de modifications ont besoin d'être tentées de nouveau ; les transactions en lecture seule n'auront jamais de conflits de sérialisation.

Le mode Repeatable Read fournit une garantie rigoureuse que chaque transaction voit un état complètement stable de la base de données. Toutefois cette vue ne sera pas nécessairement toujours cohérente avec l'exécution sérielle (un à la fois) de transactions concurrentes du même niveau d'isolation. Par exemple, même une transaction en lecture seule à ce niveau pourrait voir un enregistrement de contrôle mis à jour pour indiquer qu'un traitement par lot a été terminé, mais *ne pas* voir un des enregistrements de détail qui est une partie logique du traitement par lot parce qu'il a lu une ancienne version de l'enregistrement de contrôle. L'implémentation correcte de règles de gestion par des transactions s'exécutant à ce niveau d'isolation risque de ne pas marcher correctement sans une utilisation prudente de verrouillages explicites qui bloquent les transactions en conflit.

Le niveau d'isolation Repeatable Read est implémenté en utilisant une technique connue dans la littérature académique des bases de données et dans certains autres produits de bases de données comme l'*isolation de snapshots* (*Snapshot Isolation*). Les différences en comportement et performance peuvent être observées en comparant avec les systèmes utilisant une technique traditionnelle de verrouillage qui réduit la concurrence. Certains autres systèmes offrent même les niveaux Repeatable Read et Snapshot Isolation comme des niveaux d'isolation distinct avec des comportements différents. Le phénomène qui permet cette distinction des deux techniques n'a pas été formalisé par les chercheurs en base de données jusqu'au développement du standard SQL et sont en dehors du périmètre de ce manuel. Pour un traitement complet, merci de lire [berenson95].

Avant la version 9.1 de PostgreSQL, une demande d'isolation de transaction Serializable fournissait exactement le comportement décrit ici. Pour maintenir l'ancien niveau Serializable, il faudra maintenant demander Repeatable Read.

13.2.3. Niveau d'Isolation Serializable

Le niveau d'isolation *Serializable* fournit le niveau d'isolation le plus strict. Ce niveau émule l'exécution sérielle de transactions pour toutes les transactions validées, comme si les transactions avaient été exécutées les unes après les autres, séquentiellement, plutôt que simultanément. Toutefois, comme pour le niveau Repeatable Read, les applications utilisant ce niveau d'isolation doivent être prêtes à répéter leurs transactions en cas d'échec de sérialisation. En fait, ce niveau d'isolation fonctionne exactement comme Repeatable Read, excepté qu'il surveille aussi les conditions qui

pourraient amener l'exécution d'un jeu de transactions concurrentes à se comporter d'une manière incompatible avec les exécutions séquentielles (une à la fois) de toutes ces transactions. Cette surveillance n'introduit aucun blocage supplémentaire par rapport à repeatable read, mais il y a un coût à cette surveillance, et la détection des conditions pouvant amener une *anomalie de sérialisation* déclenchera un *échec de sérialisation*.

Comme exemple, considérons la table `ma_table`, contenant initialement :

```
 classe | valeur
--------+-------
      1 |    10
      1 |    20
      2 |   100
      2 |   200
```

Supposons que la transaction sérialisable A calcule :

```
SELECT SUM(valeur) FROM ma_table WHERE classe = 1;
```

puis insère le résultat (30) comme `valeur` dans une nouvelle ligne avec `classe = 2`. En même temps, la transaction sérialisable B calcule :

```
SELECT SUM(valeur) FROM ma_table WHERE classe = 2;
```

et obtient le résultat 300, qu'elle insère dans une nouvelle ligne avec `classe = 1`. Puis, les deux transactions essaient de valider. Si l'une des transactions fonctionnait au niveau d'isolation Repeatable Read, les deux seraient autorisées à valider ; mais puisqu'il n'y a pas d'ordre d'exécution séquentiel cohérent avec le résultat, l'utilisation de transactions Serializable permettra à une des deux transactions de valider, et annulera l'autre avec ce message :

```
ERREUR:  n'a pas pu sérialiser un accès à cause d'une mise à jour
 en parallèle"
```

En effet, si A avait été exécuté avant B, B aurait trouvé la somme 330, et non pas 300. De façon similaire, l'autre ordre aurait eu comme résultat une somme différente pour le calcul par A.

Si on se fie aux transactions sérialisées pour empêcher les anomalies, il est important que toute donnée lue depuis une table utilisateur permanente ne soit pas considérée comme valide, et ce jusqu'à ce que la transaction qui l'a lue soit validée avec succès. Ceci est vrai même pour les transactions en lecture, sauf pour les données lues dans une transaction en lecture seule et *déferrable*, dont les données sont considérées valides dès leur lecture. En effet, une telle transaction, avant de lire quoi que ce soit, attend jusqu'à l'obtention d'une image garantie libre de tout problème. Dans tous les autres cas, les applications ne doivent pas dépendre des lectures d'une transaction annulée par la suite. Elles doivent plutôt retenter la transaction jusqu'à ce qu'elle réussisse.

Pour garantir une vraie sérialisation PostgreSQL utilise le *verrouillage de prédicats*, ce qui signifie qu'il conserve des verrous qui lui permettent de déterminer si une écriture aurait eu un impact sur le résultat d'une lecture antérieure par une transaction concurrente, si elle s'était exécutée d'abord. Dans PostgreSQL, ces verrous ne causent pas de blocage, et ne peuvent donc *pas* jouer un rôle dans un verrou mortel (deadlock). Ces verrous sont utilisés pour identifier et marquer les dépendances entre des transactions sérialisables concurrentes qui, dans certaines combinaisons, peuvent entraîner des anomalies de sérialisation. Par contraste, une transaction Read Committed ou Repeatable Read qui voudrait garantir la cohérence des données devra prendre un verrou sur la table entière, ce qui pourrait bloquer d'autres utilisateurs voulant utiliser cette table, ou pourrait utiliser `SELECT FOR UPDATE` ou `SELECT FOR SHARE` qui non seulement peut bloquer d'autres transactions, mais entraîne un accès au disque.

Les verrous de prédicats dans PostgreSQL, comme dans la plupart des autres systèmes de bases de données, s'appuient sur les données réellement accédées par une transaction. Ils seront visibles dans

la vue système pg_locks avec un mode de SIReadLock. Les verrous acquis pendant l'exécution d'une requête dépendront du plan utilisé par la requête, et plusieurs verrous fins (par exemple des verrous d'enregistrement) peuvent être combinés en verrous plus grossiers (comme des verrous de page) pendant le déroulement de la transaction, pour éviter d'épuiser la mémoire utilisée par le suivi des verrous. Une transaction READ ONLY peut libérer ses verrous SIRead avant sa fin, si elle détecte que ne peut plus se produire un conflit qui entraînerait une anomalie de sérialisation. En fait, les transactions READ ONLY seront souvent capables d'établir ce fait dès leur démarrage, et ainsi éviteront de prendre des verrous de prédicat. Si vous demandez explicitement une transaction SERIALIZABLE READ ONLY DEFERRABLE, elle bloquera jusqu'à ce qu'elle puisse établir ce fait. (C'est le *seul* cas où une transaction Serializable bloque, mais pas une transaction Repeatable Read.) D'autre part, les verrous SIRead doivent souvent être gardés après la fin d'une transaction, jusqu'à ce que toutes les lectures-écritures s'étant déroulées simultanément soient terminées.

L'utilisation systématique de transactions Serializable peut simplifier le développement. La garantie que tout ensemble de transactions sérialisées, concurrentes, et validées avec succès, aura le même effet que si elles avaient été exécutées une par une signifie que, si vous pouvez démontrer qu'une transaction exécutée seule est correcte, alors vous pouvez être certain qu'elle le restera dans tout mélange de transactions sérialisées, même sans informations sur ce que font les autres transactions, ou qu'elle ne sera pas validée. Il est important qu'un environnement qui utilise cette technique ait une méthode générale pour traiter les erreurs de sérialisation (qui retournent toujours un SQLSTATE valant '40001'). En effet, il sera très difficile de prédire correctement quelles transactions pourront contribuer à des dépendances lecture/écriture, et auront besoin d'être annulées pour éviter les anomalies de sérialisation. La surveillance des dépendances lecture/écriture a un coût, tout comme la répétition des transactions annulées pour un échec de sérialisation. Mais les transactions sérialisables sont le meilleur choix en termes de performances pour certains environnements, en regard du coût et du blocage de verrous explicites, de SELECT FOR UPDATE ou de SELECT FOR SHARE, .

Bien que le niveau d'isolation Serializable de PostgreSQL ne permette à des transactions parallèles de valider leurs modifications que s'il est prouvé qu'une exécution dans l'ordre produirait le même résultat, il n'empêche pas toujours la levée d'erreurs qui ne surviendraient pas dans une véritable exécution en série. En particulier, il est possible de voir des violations de contraintes uniques suite à des conflits entre transactions Serializable qui se surchargent, même vérification explicite que la clé n'est pas présente avant de tenter de l'insérer. Ceci peut s'éviter en s'assurant que *toutes* les transactions Serializable qui peuvent insérer des clés en conflit vérifient explicitement avant si elles peuvent l'insérer. Par exemple, imaginez une application qui demande à un utilisateur une nouvelle clé, puis vérifie si elle n'existe pas déjà en cherchant à la lire d'abord, ou génère une nouvelle clé en sélectionnant la clé pré-existante la plus grande puis en ajoutant un. Si certaines transactions Serializable insèrent de nouvelles clés directement sans suivre ce protocole, des violations de contraintes uniques peuvent être rapportées, même dans des cas où elles ne pourraient pas survenir dans le cas d'une exécution en série de transactions concurrentes.

Pour une performance optimale quand on s'appuie sur les transactions Serializable pour le contrôle de la concurrence, ces points doivent être pris en considération :

- Déclarer les transactions comme READ ONLY quand c'est possible.
- Contrôler le nombre de connexions actives, au besoin en utilisant un pool de connexions. C'est toujours un point important pour les performances, mais cela peut être particulièrement important pour un système chargé qui utilise des transactions Serializable.
- Ne mettez pas plus dans une transaction seule qu'il n'est nécessaire pour l'intégrité.
- Ne laissez pas des connexions traîner en « idle in transaction » plus longtemps que nécessaire. Le paramètre de configuration idle_in_transaction_session_timeout peut être utilisé pour déconnecter automatiquement les sessions persistantes.
- Supprimez les verrous explicites, SELECT FOR UPDATE, et SELECT FOR SHARE, quand ils ne sont plus nécessaires grâce aux protections fournies automatiquement par les transactions Serializable.

- Quand le système est forcé à combiner plusieurs verrous de prédicat de niveau page en un seul verrou de niveau relation (parce que la table des verrous de prédicat est à court de mémoire), une augmentation du taux d'échecs de sérialisation peut survenir. Vous pouvez éviter ceci en augmentant max_pred_locks_per_transaction, max_pred_locks_per_relation, et/ou max_pred_locks_per_page.

- Un parcours séquentiel nécessitera toujours un verrou de prédicat au niveau relation. Ceci peut résulter en un taux plus important d'échecs de sérialisation. Il peut être utile d'encourager l'utilisation de parcours d'index en diminuant random_page_cost et/ou en augmentant cpu_tuple_cost. Assurez-vous de bien mettre en balance toute diminution du nombre d'annulations et de redémarrages de transactions et l'évolution globale du temps d'exécution des requêtes.

Le niveau d'isolation Serializable est implémenté en utilisant une technique connue sous le nom de *Serializable Snapshot Isolation* dans la littérature académique des bases de données. Elle se base sur l'isolation par snapshot et ajoute des vérifications pour les anomalies de sérialisation. Quelques différences de comportement et de performance peuvent être observées lors de la comparaison avec d'autres systèmes qui utilisent une technique de verrouillage traditionnelle. Merci de lire [ports12] pour des informations détaillées.

13.3. Verrouillage explicite

PostgreSQL fournit de nombreux modes de verrous pour contrôler les accès simultanés aux données des tables. Ces modes peuvent être utilisés pour contrôler le verrouillage par l'application dans des situations où MVCC n'a pas le comportement désiré. De plus, la plupart des commandes PostgreSQL acquièrent automatiquement des verrous avec les modes appropriés pour s'assurer que les tables référencées ne sont pas supprimées ou modifiées de façon incompatible lorsque la commande s'exécute (par exemple, `TRUNCATE` ne peut pas être exécuté de façon sûre en même temps que d'autres opérations sur la même table, donc il obtient un verrou de type `ACCESS EXCLUSIVE` sur la table pour s'assurer d'une bonne exécution).

Pour examiner une liste des verrous en cours, utilisez la vue système `pg_locks`. Pour plus d'informations sur la surveillance du statut du sous-système de gestion des verrous, référez-vous au Chapitre 28.

13.3.1. Verrous de niveau table

La liste ci-dessous affiche les modes de verrous disponibles et les contextes dans lesquels ils sont automatiquement utilisés par PostgreSQL. Vous pouvez aussi acquérir explicitement n'importe lequel de ces verrous avec la commande LOCK. Rappelez-vous que tous ces modes de verrous sont des verrous au niveau table, même si le nom contient le mot « row » (NdT : ligne) ; les noms des modes de verrous sont historiques. Dans une certaine mesure, les noms reflètent l'utilisation typique de chaque mode de verrou -- mais la sémantique est identique. La seule vraie différence entre un mode verrou et un autre est l'ensemble des modes verrous avec lesquels ils rentrent en conflit (voir Tableau 13.2). Deux transactions ne peuvent pas conserver des verrous de modes en conflit sur la même table au même moment (néanmoins, une transaction n'entre jamais en conflit avec elle-même. Par exemple, elle pourrait acquérir un verrou `ACCESS EXCLUSIVE` et acquérir plus tard un verrou `ACCESS SHARE` sur la même table). Des modes de verrou sans conflit peuvent être détenus en même temps par plusieurs transactions. Notez, en particulier, que certains modes de verrous sont en conflit avec eux-mêmes (par exemple, un verrou `ACCESS EXCLUSIVE` ne peut pas être détenu par plus d'une transaction à la fois) alors que d'autres n'entrent pas en conflit avec eux-mêmes (par exemple, un verrou `ACCESS SHARE` peut être détenu par plusieurs transactions).

Modes de verrous au niveau table

`ACCESS SHARE` (`AccessShareLock`)

En conflit avec le mode verrou `ACCESS EXCLUSIVE`.

Les commandes `SELECT` acquièrent un verrou de ce mode avec les tables référencées. En général, toute requête *lisant* seulement une table et ne la modifiant pas obtient ce mode de verrou.

`ROW SHARE` (`RowShareLock`)

En conflit avec les modes de verrous `EXCLUSIVE` et `ACCESS EXCLUSIVE`.

La commande `SELECT` acquiert un verrou de ce mode sur toutes les tables pour lesquelles l'une des options `FOR UPDATE`, `FOR NO KEY UPDATE`, `FOR SHARE`, ou `FOR KEY SHARE` est spécifiée (en plus des verrous `ACCESS SHARE` sur toutes les autres tables référencées sans une option de verrouillage `FOR ...` explicite).

`ROW EXCLUSIVE` (`RowExclusiveLock`)

En conflit avec les modes de verrous `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS EXCLUSIVE`.

Les commandes `UPDATE`, `DELETE`, `INSERT` et `MERGE` acquièrent ce mode de verrou sur la table cible (en plus des verrous `ACCESS SHARE` sur toutes les autres tables référencées). En général, ce mode de verrouillage sera acquis par toute commande *modifiant* des données de la table.

`SHARE UPDATE EXCLUSIVE` (`ShareUpdateExclusiveLock`)

En conflit avec les modes de verrous `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS EXCLUSIVE`. Ce mode protège une table contre les modifications simultanées de schéma et l'exécution d'un `VACUUM`.

Acquis par `VACUUM` (sans `FULL`), `ANALYZE`, `CREATE INDEX CONCURRENTLY`, `CREATE STATISTICS`, `COMMENT ON`, `REINDEX CONCURRENTLY`, et certaines variantes de `ALTER INDEX` et `ALTER TABLE` (pour plus de détails, voir la documentation de ces commandes).

`SHARE` (`ShareLock`)

En conflit avec les modes de verrous `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS EXCLUSIVE`. Ce mode protège une table contre les modifications simultanées des données.

Acquis par `CREATE INDEX` (sans `CONCURRENTLY`).

`SHARE ROW EXCLUSIVE` (`ShareRowExclusiveLock`)

En conflit avec les modes de verrous `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS EXCLUSIVE`. Ce mode protège une table contre les modifications concurrentes de données, et est en conflit avec elle-même, afin qu'une seule session puisse le posséder à un moment donné.

Acquis par `CREATE TRIGGER` et différentes formes de `ALTER TABLE`.

`EXCLUSIVE` (`ExclusiveLock`)

En conflit avec les modes de verrous `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS EXCLUSIVE`. Ce mode autorise uniquement les verrous `ACCESS SHARE` concurrents, c'est-à-dire que seules les lectures à partir de la table peuvent être effectuées en parallèle avec une transaction contenant ce mode de verrouillage.

Acquis par `REFRESH MATERIALIZED VIEW CONCURRENTLY`.

`ACCESS EXCLUSIVE` (`AccessExclusiveLock`)

Entre en conflit avec tous les modes (`ACCESS SHARE`, `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE EXCLUSIVE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE` et `ACCESS`

EXCLUSIVE). Ce mode garantit que le détenteur est la seule transaction à accéder à la table de quelque façon que ce soit.

Acquis par les commandes DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL, REFRESH MATERIALIZED VIEW (sans l'option CONCURRENTLY). De nombreuses formes d'ALTER INDEX et d'ALTER TABLE acquièrent également un verrou de ce niveau. C'est aussi le mode de verrou par défaut des instructions LOCK TABLE qui ne spécifient pas explicitement de mode de verrouillage.

> **Astuce**
>
> Seul un verrou ACCESS EXCLUSIVE bloque une instruction SELECT (sans FOR UPDATE/SHARE).

Une fois acquis, un verrou est normalement détenu jusqu'à la fin de la transaction. Mais si un verrou est acquis après l'établissement d'un point de sauvegarde, le verrou est relâché immédiatement si le point de sauvegarde est annulé. Ceci est cohérent avec le principe du ROLLBACK annulant tous les effets des commandes depuis le dernier point de sauvegarde. Il se passe la même chose pour les verrous acquis à l'intérieur d'un bloc d'exception PL/pgSQL : un échappement d'erreur à partir du bloc lâche les verrous acquis dans le bloc.

Tableau 13.2. Modes de verrou conflictuels

Verrou demandé	Mode verrou existant							
	ACCESS SHARE	ROW SHARE	ROW EXCL.	SHARE UPDATE EXCL.	SHARE	SHARE ROW EXCL.	EXCL.	ACCESS EXCL.
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

13.3.2. Verrous au niveau ligne

En plus des verrous au niveau table, il existe des verrous au niveau ligne. Ils sont listés ci-dessous, avec les contextes de leur utilisation automatique par PostgreSQL. Voir Tableau 13.3 pour une table complète des conflits de verrou niveau ligne. Notez qu'une transaction peut détenir des verrous en conflit sur la même ligne, y compris sur des sous-transactions différentes ; mais en dehors de cela, deux transactions ne peuvent jamais détenir des verrous en conflit pour la même ligne. Les verrous au niveau ligne n'affectent pas les lectures des données ; elles bloquent seulement les *écrivains et*

verrouilleurs sur la même ligne. Les verrous au niveau ligne sont relachés à la fin de la transaction ou lors de l'annulation du savepoint, tout comme les verrous de niveau table.

Modes des verrous au niveau ligne

`FOR UPDATE`

> `FOR UPDATE` verrouille pour modification les lignes récupérées par l'instruction `SELECT`. Cela les empêche d'être modifiées ou supprimées par les autres transactions jusqu'à la fin de la transaction en cours. Les autres transactions qui tentent des `UPDATE`, `DELETE`, `SELECT FOR UPDATE`, `SELECT FOR NO KEY UPDATE`, `SELECT FOR SHARE` ou `SELECT FOR KEY SHARE` sur ces lignes sont bloquées jusqu'à la fin de la transaction courante ; et inversement, `SELECT FOR UPDATE` attendra après une transaction concurrente qui a exécuté une de ces commandes sur la même ligne et qui verrouillera et renverra la ligne mise à jour (ou aucune ligne si elle a été supprimée). Néanmoins, à l'intérieur d'une transaction `REPEATABLE READ` ou `SERIALIZABLE`, une erreur sera renvoyée si une ligne à verrouiller a changé depuis que la transaction a commencé. Pour plus de détails, voir Section 13.4.
>
> Le mode de verrouillage `FOR UPDATE` est aussi acquis par toute commande `DELETE` sur une ligne ainsi que par un `UPDATE` qui modifie les valeurs de certaines colonnes. Actuellement, l'ensemble de colonnes considéré par le cas `UPDATE` est celui qui a un index unique lors de son utilisation par une clé étrangère (donc les index partiels et fonctionnels ne sont pas considérés), mais cela pourra être modifié dans le futur.

`FOR NO KEY UPDATE`

> `FOR NO KEY UPDATE` se comporte de la même façon que `FOR UPDATE` sauf que le verrou acquis est moins fort : ce verrou ne bloquera pas les commandes `SELECT FOR KEY SHARE` qui tenteraient d'acquérir un verrou sur les mêmes lignes. Ce mode de verrou est aussi acquis par tout `UPDATE` qui ne nécessite pas un verrou `FOR UPDATE`.

`FOR SHARE`

> `FOR SHARE` a un comportement similaire à `FOR NO KEY UPDATE`, sauf qu'il obtient un verrou partagé plutôt qu'un verrou exclusif sur chaque ligne récupérée. Un verrou partagé bloque les autres transactions réalisant des `UPDATE`, `DELETE`, `SELECT FOR UPDATE` et `SELECT FOR NO KEY UPDATE` sur ces lignes, mais il n'empêche pas les `SELECT FOR SHARE` et `SELECT FOR KEY SHARE`.

`FOR KEY SHARE`

> `FOR KEY SHARE` a un comportement similaire à `FOR SHARE`, sauf que le verrou est plus faible : `SELECT FOR UPDATE` est bloqué alors que `SELECT FOR NO KEY UPDATE` ne l'est pas. Un verrou à clé partagée bloque les autres transactions lors de l'exécution d'un `DELETE` ou d'un `UPDATE` qui modifie les valeurs clés, mais pas les autres `UPDATE`. Il n'empêche pas non plus les `SELECT FOR NO KEY UPDATE`, `SELECT FOR SHARE` et `SELECT FOR KEY SHARE`.

PostgreSQL ne garde en mémoire aucune information sur les lignes modifiées, il n'y a donc aucune limite sur le nombre de lignes verrouillées à un moment donné. Néanmoins, verrouiller une ligne peut causer une écriture disque ; ainsi, `SELECT FOR UPDATE` modifie les lignes sélectionnées pour les marquer verrouillées et cela aboutit à des écritures disques.

Tableau 13.3. Verrous en conflit au niveau ligne

Verrou demandé	**Verrou en cours**			
	FOR KEY SHARE	**FOR SHARE**	**FOR NO KEY UPDATE**	**FOR UPDATE**
FOR KEY SHARE				X

Verrou demandé	Verrou en cours			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

13.3.3. Verrous au niveau page

En plus des verrous tables et lignes, les verrous partagés/exclusifs sur les pages sont utilisés pour contrôler la lecture et l'écriture des pages de table dans l'ensemble des tampons partagées. Ces verrous sont immédiatement relâchés une fois la ligne récupérée ou mise à jour. Les développeurs d'applications ne sont normalement pas concernés par les verrous au niveau page, mais nous les mentionnons dans un souci d'exhaustivité.

13.3.4. Verrous morts (blocage)

L'utilisation de verrous explicites accroît le risque de *verrous morts* lorsque deux transactions (voire plus) détiennent chacune un verrou que l'autre convoite. Par exemple, si la transaction 1 a acquis un verrou exclusif sur la table A puis essaie d'acquérir un verrou exclusif sur la table B alors que la transaction 2 possède déjà un verrou exclusif sur la table B et souhaite maintenant un verrou exclusif sur la table A, alors aucun des deux ne peut continuer. PostgreSQL détecte automatiquement ces situations de blocage et les résout en annulant une des transactions impliquées, permettant ainsi à l'autre (aux autres) de se terminer (savoir quelle transaction est réellement annulée est difficile à prévoir, mais vous ne devriez pas vous en préoccuper).

Notez que les verrous morts peuvent aussi se produire suite à des verrous de niveau ligne (et du coup, ils peuvent se produire même si le verrouillage explicite n'est pas utilisé). Considérons le cas où il existe deux transactions concurrentes modifiant une table. La première transaction exécute :

```
UPDATE comptes SET balance = balance + 100.00 WHERE no_compte =
 11111;
```

Elle acquiert un verrou au niveau ligne sur la ligne spécifiée par le numéro de compte (no_compte). Ensuite, la deuxième transaction exécute :

```
UPDATE comptes SET balance = balance + 100.00 WHERE no_compte =
 22222;
UPDATE comptes SET balance = balance - 100.00 WHERE no_compte =
 11111;
```

La première instruction `UPDATE` acquiert avec succès un verrou au niveau ligne sur la ligne spécifiée, donc elle réussit à mettre à jour la ligne. Néanmoins, la deuxième instruction `UPDATE` trouve que la ligne qu'elle essaie de mettre à jour a déjà été verrouillée, alors elle attend la fin de la transaction ayant acquis le verrou. Maintenant, la première transaction exécute :

```
UPDATE comptes SET balance = balance - 100.00 WHERE no_compte =
 22222;
```

La première transaction essaie d'acquérir un verrou au niveau ligne sur la ligne spécifiée, mais ne le peut pas : la deuxième transaction détient déjà un verrou. Donc, elle attend la fin de la transaction deux. Du coup, la première transaction est bloquée par la deuxième et la deuxième est bloquée par la première : une condition de blocage, un verrou mort. PostgreSQL détectera cette situation et annulera une des transactions.

Le niveau d'isolation Repeatable Read est implémenté en utilisant une technique connue dans la littérature académique sur les bases de données et dans certains produits de bases de données

sous le nom de *Snapshot Isolation*. Des différences en comportement et en performance peuvent être observées lors de comparaisons avec des systèmes qui utilisent une technique de verrouillage traditionnelle qui réduit la concurrence. Quelques autres systèmes peuvent même proposer Repeatable Read et Snapshot Isolation sous la forme de niveaux d'isolation distincts avec des comportements différents. Les phénomènes qui distinguent les deux techniques n'ont pas été formalisés par les chercheurs en bases de données jusqu'à ce que le standard SQL ne soit écrit. Pour un traitement complet, voir [berenson95].

La meilleure défense contre les verrous morts est généralement de les éviter en s'assurant que toutes les applications utilisant une base de données acquièrent des verrous sur des objets multiples dans un ordre cohérent. Dans l'exemple ci-dessus, si les deux transactions avaient mis à jour les lignes dans le même ordre, aucun blocage n'aurait eu lieu. Vous devez vous assurer que le premier verrou acquis sur un objet dans une transaction est dans le mode le plus restrictif pour cet objet. S'il n'est pas possible de vérifier ceci à l'avance, alors les blocages doivent être gérés à l'exécution en réessayant les transactions annulées à cause du blocage.

Tant qu'aucune situation de blocage n'est détectée, une transaction cherchant soit un verrou de niveau table soit un verrou de niveau ligne attend indéfiniment que les verrous en conflit soient relâchés. Ceci signifie que maintenir des transactions ouvertes sur une longue période de temps (par exemple en attendant une saisie de l'utilisateur) est parfois une mauvaise idée.

13.3.5. Verrous informatifs

PostgreSQL fournit un moyen pour créer des verrous qui ont une signification définie par l'application. Ils sont qualifiés d'*informatifs*, car le système ne force pas leur utilisation -- c'est à l'application de les utiliser correctement. Les verrous informatifs peuvent être utiles pour des manières d'utiliser le verrouillage qui ne sont pas en phase avec le modèle MVCC. Par exemple, une utilisation habituelle des verrous informatifs est l'émulation de stratégie de verrouillage pessimiste typique des systèmes de gestion de données à partir de « fichiers à plat ». Bien qu'un drapeau stocké dans une table puisse être utilisé pour la même raison, les verrous informatifs sont plus rapides, évitent la fragmentation de la table et sont nettoyés automatiquement par le serveur à la fin de la session.

Il existe deux façons pour acquérir un verrou informatif dans PostgreSQL : au niveau de la session ou au niveau de la transaction. Une fois acquis au niveau de la session, un verrou informatif est détenu jusqu'à ce que le verrou soit explicitement relâché ou à la fin de la session. Contrairement aux demandes de verrou standard, les demandes de verrous informatifs au niveau session n'honorent pas la sémantique de la transaction : un verrou acquis lors d'une transaction qui est annulée plus tard sera toujours acquis après le ROLLBACK, et de la même façon, un verrou relâché reste valide même si la transaction appelante a échoué après. Un verrou peut être acquis plusieurs fois par le processus qui le détient ; pour chaque demande de verrou terminée, il doit y avoir une demande de relâche du verrou correspondant avant que ce dernier ne soit réellement relâché. D'un autre côté, les demandes de verrou au niveau transaction se comportent plutôt comme des demandes de verrous standards : les verrous sont automatiquement relâchés à la fin de la transaction, et il n'y a pas d'opération explicite de déverrouillage. Ce comportement est souvent plus intéressant que le comportement au niveau session pour un usage rapide d'un verrou informatif. Les demandes de verrou au niveau session et transaction pour le même identifiant de verrou informatif se bloqueront de la façon attendue. Si une session détient déjà un verrou informatif donné, les demandes supplémentaires par le même processus réussiront toujours, même si d'autres sessions sont en attente ; ceci est vrai, quel que soit le niveau (session ou transaction) du verrou détenu et des verrous demandés.

Comme tous les verrous dans PostgreSQL, une liste complète des verrous informatifs détenus actuellement par toute session est disponible dans la vue système `pg_locks`.

Les verrous informatifs et les verrous standards sont stockés dans une partie de la mémoire partagée, dont la taille est définie par les variables de configuration max_locks_per_transaction et max_connections. Attention à ne pas vider cette mémoire, sinon le serveur ne serait plus capable d'accorder des verrous. Ceci impose une limite supérieure au nombre de verrous informatifs que le serveur peut accorder, typiquement entre des dizaines et des centaines de milliers suivant la façon dont le serveur est configuré.

Dans certains cas utilisant cette méthode, tout spécialement les requêtes impliquant un tri explicite et des clauses `LIMIT`, une grande attention doit être portée au contrôle des verrous acquis, à cause de l'ordre dans lequel les expressions SQL sont évaluées. Par exemple :

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; --
 danger !
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- ok
```

Dans les requêtes ci-dessus, la deuxième forme est dangereuse parce qu'il n'est pas garanti que l'application de `LIMIT` ait lieu avant que la fonction du verrou soit exécutée. Ceci pourrait entraîner l'acquisition de certains verrous que l'application n'attendait pas, donc qu'elle ne pourrait, du coup, pas relâcher (sauf à la fin de la session). Du point de vue de l'application, de tels verrous sont en attente, bien qu'ils soient visibles dans `pg_locks`.

Les fonctions fournies pour manipuler les verrous informatifs sont décrites dans Section 9.27.10.

13.4. Vérification de cohérence des données au niveau de l'application

Il est très difficile d'implémenter des règles de gestion sur l'intégrité des données en utilisant des transactions Read Committed parce que la vue des données est changeante avec chaque ordre, et même un seul ordre peut ne pas se cantonner à son propre instantané si un conflit en écriture se produit.

Bien qu'une transaction Repeatable Read ait une vue stable des données dans toute la durée de son exécution, il y a un problème subtil quand on utilise les instantanés MVCC pour vérifier la cohérence des données, impliquant quelque chose connu sous le nom de *conflits lecture/écriture*. Si une transaction écrit des données et qu'une transaction concurrente essaie de lire la même donnée (que ce soit avant ou après l'écriture), elle ne peut pas voir le travail de l'autre transaction. Le lecteur donne donc l'impression de s'être exécuté le premier, quel que soit celui qui a commencé le premier ou qui a validé le premier. Si on s'en tient là, ce n'est pas un problème, mais si le lecteur écrit aussi des données qui sont lues par une transaction concurrente, il y a maintenant une transaction qui semble s'être exécutée avant les transactions précédemment mentionnées. Si la transaction qui semble s'être exécutée en dernier valide en premier, il est très facile qu'un cycle apparaisse dans l'ordre d'exécution des transactions. Quand un cycle de ce genre apparaît, les contrôles d'intégrité ne fonctionneront pas correctement sans aide.

Comme mentionné dans Section 13.2.3, les transactions Serializable ne sont que des transactions Repeatable Read qui ajoutent une supervision non bloquante de formes dangereuses de conflits lecture/écriture. Quand une de ces formes est détectée qui pourrait entraîner un cycle dans l'ordre apparent d'exécution, une des transactions impliquées est annulée pour casser le cycle.

13.4.1. Garantir la Cohérence avec des Transactions Serializable

Si le niveau d'isolation de transactions Serializable est utilisé pour toutes les écritures et toutes les lectures qui ont besoin d'une vue cohérente des données, aucun autre effort n'est requis pour garantir la cohérence. Un logiciel d'un autre environnement écrit pour utiliser des transactions Serializable pour garantir la cohérence devrait « fonctionner sans modification » de ce point de vue dans PostgreSQL.

L'utilisation de cette technique évitera de créer une charge de travail inutile aux développeurs d'applications si le logiciel utilise un framework qui réessaie automatiquement les transactions

annulées pour échec de sérialisation. Cela pourrait être une bonne idée de positionner `default_transaction_isolation` à `serializable`. Il serait sage, par ailleurs, de vous assurer qu'aucun autre niveau d'isolation n'est utilisé, soit par inadvertance, soit pour contourner les contrôles d'intégrité, en vérifiant les niveaux d'isolations dans les triggers.

Voyez Section 13.2.3 pour des suggestions sur les performances.

> **Avertissement**
>
> Ce niveau de protection de l'intégrité en utilisant des transactions Serializable ne s'étend pour le moment pas jusqu'au mode standby (Section 27.4). Pour cette raison, les utilisateurs du hot standby voudront peut-être utiliser Repeatable Read et un verrouillage explicite sur le primaire.

13.4.2. Garantir la Cohérence avec des Verrous Bloquants Explicites

Quand des écritures non sérialisables sont possibles, pour garantir la validité courante d'un enregistrement et le protéger contre des mises à jour concurrentes, on doit utiliser `SELECT FOR UPDATE`, `SELECT FOR SHARE`, ou un ordre `LOCK TABLE` approprié. (`SELECT FOR UPDATE` et `SELECT FOR SELECT` ne verrouillent que les lignes retournées contre les mises à jour concurrentes, tandis que `LOCK TABLE` verrouille toute la table.) Cela doit être pris en considération quand vous portez des applications PostgreSQL à partir d'autres environnements.

Il est aussi important de noter pour ceux qui convertissent à partir d'autres environnements le fait que `SELECT FOR UPDATE` ne garantit pas qu'une transaction concurrente ne mettra pas à jour ou n'effacera pas l'enregistrement sélectionné. Pour faire cela dans PostgreSQL, vous devez réellement modifier l'enregistrement, même si vous n'avez pas besoin de modifier une valeur. `SELECT FOR UPDATE` *empêche temporairement* les autres transactions d'acquérir le même verrou ou d'exécuter un `UPDATE` ou `DELETE` qui modifierait l'enregistrement verrouillé, mais une fois que la transaction possédant ce verrou valide ou annule, une transaction bloquée pourra continuer avec son opération en conflit, sauf si un réel `UPDATE` de l'enregistrement a été effectué pendant que le verrou était possédé.

Les vérifications globales de validité demandent davantage de réflexion sous un MVCC non sérialisable. Par exemple, une application bancaire pourrait vouloir vérifier que la somme de tous les crédits d'une table est égale à la somme de tous les débits d'une autre, alors que les deux tables sont en cours de mise à jour. La comparaison des résultats de deux `SELECT sum(...)` successifs ne fonctionnera pas correctement en mode Read Committed, puisque la seconde requête inclura probablement les résultats de transactions non prises en compte dans la première. Effectuer les deux sommes dans une seule transaction repeatable read donnera uniquement une image précise des effets des transactions qui ont validé avant le début de la transaction repeatable read — mais on pourrait légitimement se demander si la réponse est toujours valide au moment où elle est fournie. Si la transaction repeatable read a elle-même effectué des modifications avant d'effectuer le test de cohérence, l'utilité de la vérification devient encore plus sujette à caution, puisque maintenant elle inclut des modifications depuis le début de la transaction, mais pas toutes. Dans ce genre de cas, une personne prudente pourra vouloir verrouiller toutes les tables nécessaires à la vérification, afin d'avoir une vision incontestable de la réalité courante. Un mode `SHARE` (ou plus élevé) garantit qu'il n'y a pas de changements non validés dans la table verrouillée, autres que ceux de la transaction courante.

Notez aussi que si on se fie au verrouillage explicite pour empêcher les mises à jour concurrentes, on devrait soit utiliser Read Committed, soit utiliser Repeatable Read et faire attention à obtenir les verrous avant d'effectuer les requêtes. Un verrou obtenu par une transaction repeatable read garantit qu'aucune autre transaction modifiant la table n'est en cours d'exécution, mais si l'instantané vu par la transaction est antérieur à l'obtention du verrou, il pourrait aussi précéder des modifications maintenant validées dans la table. Un instantané de transaction repeatable read est en fait figé à l'exécution de sa première requête ou commande de modification de données (`SELECT`, `INSERT`, `UPDATE`, `DELETE` ou `MERGE`), il est donc possible d'obtenir les verrous explicitement avant que l'instantané ne soit figé.

Le niveau d'isolation Serializable est implémenté en utilisant une technique connue dans la littérature académique sur les bases de données sous le nom de Serializable Snapshot Isolation, qui ajoute à une isolation de snapshot en ajoutant des vérifications sur les anomalies de sérialisation. Quelques différences dans le comportement et les performances sont observables si on le compare à d'autres systèmes qui utilisent une technique de verrouillage traditionnelle. Merci de lire [ports12] pour des informations détaillées.

13.5. Gestion des échecs de sérialisation

Les niveaux d'isolation Repeatable Read et Serializable peuvent produire des erreurs conçues pour empêcher des anomalies de sérialisation. Comme indiqué précédemment, les applications utilisant ces niveaux doivent être préparées pour tenter de nouveau les transactions qui échouent suite à des erreurs de sérialisation. Ce type de message d'erreur variera suivant les circonstances précises, mais il y aura toujours le code SQLSTATE `40001` (`serialization_failure`).

Il pourrait aussi être conseillé de reprendre les échecs de deadlock. Ils ont le code SQLSTATE `40P01` (`deadlock_detected`).

Dans certains cas, il est aussi approprié de tenter de nouveau après des échecs d'unicité, qui ont le code SQLSTATE `23505` (`unique_violation`), et des échecs de contrainte d'exclusion qui ont le code SQLSTATE `23P01` (`exclusion_violation`). Par exemple, si l'application sélectionne une nouvelle valeur pour une clé primaire après l'inspection les clés stockées enregistrées, elle pourrait obtenir un échec d'unicité parce qu'une autre application a sélectionné la même nouvelle clé en même temps. Ceci est en réalité un échec de sérialisation mais le serveur ne le détectera pas ainsi parce qu'il ne peut pas « voir » la connexion entre la valeur insérée et les lectures précédentes. Il existe aussi quelques cas spécifiques pour lesquels le serveur renverra une erreur d'unicité ou de contrainte d'exclusion même si, en principe, il a suffisamment d'informations pour déterminer qu'un problème de sérialisation en est la cause sous-jacente. Alors qu'il est recommendable de seulement tester de nouveau les erreurs `serialization_failure` de façon inconditionnelle, plus d'attention est nécessaire lors de nouvelles tentatives avec ces autres codes d'erreur car ils pourraient représenter des conditions d'erreur persistentes plutôt que des échecs temporaires.

Il est important de ré-essayer la transaction complète, incluant toute la logique qui décide quel SQL envoyer et/ou quelles valeurs utiliser. De ce fait, PostgreSQL n'offre pas une capacité de tentatives automatiques car il ne peut le faire sans garantie d'exactitude.

Une nouvelle tentative ne garantit pas que la transaction nouvellement essayée se termine ; plusieurs essais pourraient se révéler nécessaires. Dans les cas avec une contention très haute, il est possible que la fin d'une transaction demande de nombreuses tentatives. Dans les cas impliquant une transaction préparée en conflit, il pourrait être impossible de progresser jusqu'à ce que la transaction préparée soit validée ou annulée.

13.6. Avertissements

Certaines commandes DDL, actuellement seulement TRUNCATE et les formes d'ALTER TABLE qui réécrivent la table, ne sont pas sûres au niveau MVCC. Ceci signifie que, après la validation d'une troncature ou d'une réécriture, la table apparaîtra vide aux transactions concurrentes si elles utilisaient une image de la base datant d'avant la validation de la commande DDL. Ceci ne sera un problème que pour une transaction qui n'a pas encore accédé à la table en question avant le lancement de la commande DDL -- toute transaction qui a fait cela détiendra au moins un verrou de type ACCESS SHARE sur la table, ce qui bloquera la commande DDL jusqu'à la fin de la transaction. Donc ces commandes ne causeront pas d'incohérence apparente dans le contenu de la table pour des requêtes successives sur la table cible, mais elles seront la cause d'incohérence visible entre le contenu de la table cible et les autres tables de la base.

L'accès interne aux catalogues système n'est pas faite en utilisant le niveau d'isolation de la transaction actuelle. Ceci signifie que les objets de base nouvellement créés, comme des tables, sont visibles aux transactions concurrentes en Repeatable Read et en Serializable, même si les lignes qu'elles

contiennent ne le sont pas. À l'inverse les requêtes qui examinent les catalogues systèmes ne voient pas les lignes représentant les objets créés en concurrence dans les plus haut niveaux d'isolation.

13.7. Verrous et index

Bien que PostgreSQL fournisse un accès en lecture/écriture non bloquant aux données de la table, l'accès en lecture/écriture non bloquant n'est pas proposé pour chaque méthode d'accès aux index implémentés dans PostgreSQL. Les différents types d'index sont gérés ainsi :

Index B-tree, GiST et SP-GiST

> Des verrous partagés/exclusifs au niveau page à court terme sont utilisés pour les accès en lecture/écriture. Les verrous sont relâchés immédiatement après que chaque ligne d'index est lue ou insérée. Ces types d'index fournissent la plus grande concurrence d'accès, sans condition de verrous mortels.

Index hash

> Des verrous partagés/exclusifs au niveau des blocs de hachage sont utilisés pour l'accès en lecture/écriture. Les verrous sont relâchés après qu'un bloc a été traité entièrement. Les verrous au niveau bloc fournissent une meilleur concurrence d'accès que les verrous au niveau index, mais les verrous mortels sont possibles, car les verrous sont détenus plus longtemps que l'opération sur l'index.

Index GIN

> Des verrous partagés/exclusifs au niveau page à court terme sont utilisés pour les accès en lecture/écriture. Les verrous sont relâchés immédiatement après que chaque ligne d'index est lue ou insérée. Cependant, notez que l'insertion d'une valeur indexée par GIN produit généralement plusieurs insertions de clés d'index par ligne, donc GIN peut avoir un travail important à réaliser pour l'insertion d'une seule valeur.

Actuellement, les index B-tree offrent les meilleures performances pour les applications concurrentes. Comme ils ont plus de fonctionnalités que les index hash, ils sont le type d'index recommandé pour les applications concurrentes qui ont besoin d'indexer des données scalaires. Lors du traitement de données non scalaires, les index B-tree ne sont pas utiles. Les index GiST, SP-GiST ou GIN doivent être utilisés à la place.

L'accès interne aux catalogues systèmes n'est pas réalisée en utilisant le niveau d'isolation de la transaction courante. Cela signifie que les objets nouvellement créés d'une base, comme les tables, sont visibles aux transactions Repeatable Read et Serializable, même si les lignes qu'elles contiennent ne le sont pas. À l'inverse, les requêtes qui examinent explicitement les catalogues systèmes ne voient pas les lignes représentant les objets nouvellement créés en concurrence, dans les plus hauts niveaux d'isolation.

Chapitre 14. Conseils sur les performances

La performance des requêtes peut être affectée par un grand nombre d'éléments. Certains peuvent être contrôlés par l'utilisateur, d'autres sont fondamentaux au concept sous-jacent du système. Ce chapitre fournit des conseils sur la compréhension et sur la configuration fine des performances de PostgreSQL.

14.1. Utiliser EXPLAIN

PostgreSQL réalise un *plan de requête* pour chaque requête qu'il reçoit. Choisir le bon plan correspondant à la structure de la requête et aux propriétés des données est absolument critique pour de bonnes performances, donc le système inclut un *planificateur* ou *optimiseur* complexe qui tente de choisir les bons plans. Vous pouvez utiliser la commande EXPLAIN pour voir quel plan de requête le planificateur crée pour une requête particulière. La lecture du plan est un art qui requiert de l'expérience pour le maîtriser, mais cette section essaie de couvrir les bases.

Les exemples dans cette section sont tirés de la base de données pour les tests de régression après avoir effectué un VACUUM ANALYZE, avec les sources de la version de développement 9.3. Vous devriez obtenir des résultats similaires si vous essayez les exemples vous-même, mais vos estimations de coût et de nombre de lignes pourraient légèrement varier parce que les statistiques d'ANALYZE sont basées sur des échantillons aléatoires, et parce que les coûts sont dépendants de la plateforme utilisée.

Les exemples utilisent le format de sortie par défaut (« text ») d'EXPLAIN, qui est compact et pratique pour la lecture. Si vous voulez utiliser la sortie d'EXPLAIN avec un programme pour une analyse ultérieure, vous devriez utiliser un des formats de sortie au format machine (XML, JSON ou YAML) à la place.

14.1.1. Concepts d'EXPLAIN

La structure d'un plan de requête est un arbre de *nœuds de plan*. Les nœuds de bas niveau sont les nœuds de parcours : ils renvoient les lignes brutes d'une table. Il existe différents types de nœuds de parcours pour les différentes méthodes d'accès aux tables : parcours séquentiel, parcours d'index et parcours d'index bitmap. Il y a également des ensembles de lignes qui ne proviennent pas de tables, avec par exemple des clauses VALUES ainsi que des fonctions renvoyant des ensembles dans un FROM, qui ont leurs propres types de nœuds de parcours. Si la requête requiert des jointures, agrégations, tris ou d'autres opérations sur les lignes brutes, ce seront des nœuds supplémentaires au-dessus des nœuds de parcours pour réaliser ces opérations. Encore une fois, il existe plus d'une façon de réaliser ces opérations, donc différents types de nœuds peuvent aussi apparaître ici. La sortie d'EXPLAIN comprend une ligne pour chaque nœud dans l'arbre du plan, montrant le type de nœud basique avec les estimations de coût que le planificateur a faites pour l'exécution de ce nœud du plan. Des lignes supplémentaires peuvent apparaître, indentées par rapport à la ligne de résumé du nœud, pour montrer les propriétés supplémentaires du nœud. La première ligne (le nœud tout en haut) comprend le coût d'exécution total estimé pour le plan ; c'est ce nombre que le planificateur cherche à minimiser.

Voici un exemple trivial, juste pour montrer à quoi ressemble l'affichage.

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Puisque la requête n'a pas de clause WHERE, il faut parcourir toutes les lignes de la table, c'est pourquoi le planificateur a choisi d'utiliser un plan avec un simple parcours séquentiel. Les nombres affichés entre parenthèses sont (de gauche à droite) :

- Coût estimé du lancement. Cela correspond au temps passé avant que l'affichage de la sortie ne commence, par exemple le temps de faire un tri dans un nœud de tri ;
- Coût total estimé. Cela suppose que le nœud du plan d'exécution est exécuté entièrement, c'est-à-dire que toutes les lignes disponibles sont récupérées. En pratique, un nœud parent peut arrêter la récupération de toutes les lignes disponibles avant la fin (voir l'exemple `LIMIT` ci-dessous) ;
- Nombre de lignes estimé en sortie par ce nœud de plan. Encore une fois, on suppose que le nœud est exécuté entièrement.
- Largeur moyenne estimée (en octets) des lignes en sortie par ce nœud du plan.

Les coûts sont mesurés en unités arbitraires déterminées par les paramètres de coût du planificateur (voir Section 20.7.2). La pratique habituelle est de mesurer les coûts en unité de récupération de pages disque ; autrement dit, seq_page_cost est initialisé à `1.0` par convention et les autres paramètres de coût sont relatifs à cette valeur. Les exemples de cette section sont exécutés avec les paramètres de coût par défaut.

Il est important de comprendre que le coût d'un nœud de haut niveau inclut le coût de tous les nœuds fils. Il est aussi important de réaliser que le coût reflète seulement les éléments d'importance pour le planificateur. En particulier, le coût ne considère pas le temps dépensé dans la transmission des lignes de résultat au client, ce qui pourrait être un facteur important dans le temps réel passé ; mais le planificateur l'ignore parce qu'il ne peut pas le changer en modifiant le plan (chaque plan correct sortira le même ensemble de lignes).

La valeur `rows` est un peu difficile car il ne s'agit pas du nombre de lignes traitées ou parcourues par le plan de nœuds, mais plutôt le nombre émis par le nœud. C'est habituellement moins, reflétant la sélectivité estimée des conditions de la clause `WHERE` qui sont appliquées au nœud. Idéalement, les estimations des lignes de haut niveau seront une approximation des nombres de lignes déjà renvoyées, mises à jour, supprimées par la requête.

Quand un `UPDATE`, un `DELETE` ou un `MERGE` affecte une hiérarchie d'héritage, la sortie pourrait ressembler à ceci :

```
EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;

                                                QUERY PLAN
-----------------------------------------------------------------------
-----------------------------------
 Update on parent  (cost=0.00..24.59 rows=0 width=0)
   Update on parent parent_1
   Update on child1 parent_2
   Update on child2 parent_3
   Update on child3 parent_4
   ->  Result  (cost=0.00..24.59 rows=4 width=14)
         ->  Append  (cost=0.00..24.54 rows=4 width=14)
               ->  Seq Scan on parent parent_1  (cost=0.00..0.00
 rows=1 width=14)
                     Filter: (f1 = 101)
               ->  Index Scan using child1_pkey on child1 parent_2
 (cost=0.15..8.17 rows=1 width=14)
                     Index Cond: (f1 = 101)
               ->  Index Scan using child2_pkey on child2 parent_3
 (cost=0.15..8.17 rows=1 width=14)
                     Index Cond: (f1 = 101)
               ->  Index Scan using child3_pkey on child3 parent_4
 (cost=0.15..8.17 rows=1 width=14)
                     Index Cond: (f1 = 101)
```

Dans cet exemple, le nœud Update doit prendre en compte les trois tables filles ainsi que la table parente mentionnée dans la requête. Donc il y a quatre sous-plans de parcours en entrée, un par table.

Pour plus de clarté, le nœud Update est annoté pour afficher les tables cibles spécifiques à mettre à jour, dans le même ordre que les sous-plans correspondants.

Le `Temps de planification (Planning time)` affiché est le temps qu'il a fallu pour générer le plan d'exécution de la requête analysée et pour l'optimiser. Cela n'inclut ni le temps de réécriture ni le temps d'analyse.

Pour revenir à notre exemple :

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Ces nombres sont directement dérivés. Si vous faites :

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

vous trouverez que `tenk1` a 358 blocs disque et 10000 lignes. Le coût estimé est calculé avec (nombre de blocs lus * seq_page_cost) + (lignes parcourues * cpu_tuple_cost). Par défaut, `seq_page_cost` vaut 1,0 et `cpu_tuple_cost` vaut 0,01. Donc le coût estimé est de (358 * 1,0) + (10000 * 0,01), soit 458.

Maintenant, modifions la requête originale pour ajouter une condition `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

Notez que l'affichage d'`EXPLAIN` montre la clause `WHERE` appliquée comme une condition de « filtre » rattachée au nœud de parcours séquentiel ; ceci signifie que le nœud de plan vérifie la condition pour chaque ligne qu'il parcourt et ne conserve que celles qui satisfont la condition. L'estimation des lignes en sortie a baissé à cause de la clause `WHERE`. Néanmoins, le parcours devra toujours visiter les 10000 lignes, donc le coût n'a pas baissé ; en fait, il a un peu augmenté(par 10000 * cpu_operator_cost pour être exact) dans le but de refléter le temps CPU supplémentaire dépensé pour vérifier la condition `WHERE`.

Le nombre réel de lignes que cette requête sélectionnera est 7000, mais l'estimation `rows` est approximative. Si vous tentez de dupliquer cette expérience, vous obtiendrez probablement une estimation légèrement différente ; de plus, elle changera après chaque commande `ANALYZE` parce que les statistiques produites par `ANALYZE` sont prises à partir d'un extrait au hasard de la table.

Maintenant, rendons la condition plus restrictive :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

Ici, le planificateur a décidé d'utiliser un plan en deux étapes : le nœud en bas du plan visite un index pour trouver l'emplacement des lignes correspondant à la condition de l'index, puis le nœud du plan

du dessus récupère réellement ces lignes de la table. Récupérer séparément les lignes est bien plus coûteux que de les lire séquentiellement, mais comme toutes les pages de la table n'ont pas à être visitées, cela revient toujours moins cher qu'un parcours séquentiel (la raison de l'utilisation d'un plan à deux niveaux est que le nœud du plan du dessus trie les emplacements des lignes identifiées par l'index dans l'ordre physique avant de les lire pour minimiser les coûts des récupérations séparées. Le « bitmap » mentionné dans les noms de nœuds est le mécanisme qui s'occupe du tri).

Maintenant, ajoutons une autre condition à la clause `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 =
 'xxx';

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

L'ajout de la condition `stringu1 = 'xxx'` réduit l'estimation du nombre de lignes renvoyées, mais pas son coût, car il faut toujours parcourir le même ensemble de lignes. Notez que la clause `stringu1` ne peut être appliquée comme une condition d'index car l'index ne porte que sur la colonne `unique1`. À la place, un filtre a été appliqué sur les lignes récupérées par l'index. C'est pourquoi le coût a légèrement augmenté pour refléter la vérification supplémentaire.

Dans certains cas, le planificateur préfèrera un plan « simple » d'index :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1
 width=244)
   Index Cond: (unique1 = 42)
```

Dans ce type de plan, les lignes de la table sont récupérées dans l'ordre de l'index, ce qui les rend encore plus coûteuses à récupérer, mais il y en a tellement peu que le coût supplémentaire pour trier l'ordre des lignes n'est pas rentable. Vous verrez principalement ce type de plan pour les requêtes qui ne récupèrent qu'une seule ligne, ou pour les requêtes qui ont une condition `ORDER BY` qui correspond à l'ordre de l'index, car cela ne nécessite aucune étape supplémentaire pour satisfaire l'`ORDER BY`. Dans cet exemple, ajouter `ORDER BY unique1` ferait que l'optimiseur utilise le même plan parce que l'index fournit déjà implicitement le tri requis.

L'optimiseur peut exécuter une clause `ORDER BY` de plusieurs façons. L'exemple ci-dessus montre qu'une clause de tri peut être effectué implicitement. L'optimiseur peut aussi ajouter une étape de tri (`sort`) explicite :

```
EXPLAIN SELECT * FROM tenk1 ORDER BY unique1;
                            QUERY PLAN
-------------------------------------------------------------------
 Sort  (cost=1109.39..1134.39 rows=10000 width=244)
   Sort Key: unique1
   ->  Seq Scan on tenk1  (cost=0.00..445.00 rows=10000 width=244)
```

Si une partie du plan garantit un ordre sur un préfixe des clés de tri requises, alors l'optimiseur peut décider à la place d'utiliser une étape de tri incrémental (`incremental sort`) :

```
EXPLAIN SELECT * FROM tenk1 ORDER BY four, ten LIMIT 100;
```

```
                                                        QUERY PLAN
-----------------------------------------------------------------------------
---------------------------------
 Limit  (cost=521.06..538.05 rows=100 width=244)
   ->  Incremental Sort  (cost=521.06..2220.95 rows=10000
 width=244)
         Sort Key: four, ten
         Presorted Key: four
         ->  Index Scan using index_tenk1_on_four on tenk1
 (cost=0.29..1510.08 rows=10000 width=244)
```

Comparé aux tris habituels, le tri incrémental permet de renvoyer les lignes avant que l'ensemble du résultat ne soit trié, ce qui permet en particulier des optimisations avec les requêtes utilisant la clause `LIMIT`. Il peut aussi réduire l'utilisation de la mémoire et la probabilité d'envoyer des tris sur disque, mais cela a un coût : une surcharge pour répartir l'ensemble de lignes résultats dans plusieurs groupes de tri.

S'il y a des index sur plusieurs colonnes référencées dans la clause `WHERE`, le planificateur pourrait choisir d'utiliser une combinaison binaire (AND et OR) des index :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
 -------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
    Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
 rows=101 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78
 rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

Mais ceci requiert de visiter plusieurs index, donc ce n'est pas nécessairement un gain comparé à l'utilisation d'un seul index et au traitement de l'autre condition par un filtre. Si vous variez les échelles de valeurs impliquées, vous vous apercevrez que le plan change en accord.

Voici un exemple montrant les effets d'un `LIMIT` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000
 LIMIT 2;

                                     QUERY PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27
 rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

C'est la même requête qu'au-dessus, mais avec l'ajout de `LIMIT`, ce qui fait que toutes les lignes ne seront pas récupérées, et donc que le planificateur change sa façon de procéder. Notez que le coût total ainsi que le nombre de lignes du nœud de parcours d'index sont affichés comme si le nœud devait être exécuté entièrement. Cependant, le nœud Limit s'attend à s'arrêter après avoir récupéré seulement un cinquième de ces lignes, c'est pourquoi son coût total n'est qu'un cinquième du coût précédent, ce qui est le vrai coût estimé de la requête. Ce plan est préférable à l'ajout d'un nœud Limit au plan précédent, car le Limit ne pourrait pas empêcher le coût de départ du parcours d'index Bitmap, ce qui augmenterait le coût d'environ 25 unités avec cette approche.

Maintenant, essayons de joindre deux tables, en utilisant les colonnes dont nous avons discuté :

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
 ------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10
 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36
 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91
 rows=1 width=244)
          Index Cond: (unique2 = t1.unique2)
```

Dans ce plan, nous avons un nœud de jointure en boucle imbriquée sur deux parcours de tables en entrée. L'indentation des lignes de sommaire des nœuds reflète la structure en arbre du plan. Le premier nœud, ou nœud « externe », utilise le même parcours de bitmap que celui vu précédemment, et donc ses coût et nombre de lignes sont les mêmes que ce que l'on aurait obtenu avec `SELECT ... WHERE unique1 < 10`, car la même clause `WHERE unique1 < 10` est appliquée à ce nœud. La clause `t1.unique2 = t2.unique2` n'a pas encore d'intérêt, elle n'affecte donc pas le nombre de lignes du parcours externe. Le nœud de jointure en boucle imbriquée s'exécutera sur le deuxième nœud, ou nœud « interne », pour chaque ligne obtenue du nœud externe. Les valeurs de colonne de la ligne externe courante peuvent être utilisées dans le parcours interne ; ici, la valeur `t1.unique2` de la ligne externe est disponible, et on peut obtenir un plan et un coût similaires à ce que l'on a vu plus haut pour le cas simple `SELECT ... WHERE t2.unique2 = constant`.(Le coût estimé est ici un peu plus faible que celui vu précédemment, en prévision de la mise en cache des données durant les parcours d'index répétés sur `t2`.) Les coûts du nœud correspondant à la boucle sont ensuite initialisés sur la base du coût du parcours externe, avec une répétition du parcours interne pour chaque ligne externe (ici 10 * 7,91), plus un petit temps CPU pour traiter la jointure.

Dans cet exemple, le nombre de lignes en sortie de la jointure est identique au nombre de lignes des deux parcours, mais ce n'est pas vrai en règle générale car vous pouvez avoir des clauses `WHERE` mentionnant les deux tables et qui, donc, peuvent seulement être appliquées au point de jointure, et non pas aux parcours d'index. Voici un exemple :

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred <
 t2.hundred;

                                      QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..49.46 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10
 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36
 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2
 (cost=0.29..8.46 rows=10 width=244)
               Index Cond: (unique2 < 10)
```

La condition `t1.hundred < t2.hundred` ne peut être testée dans l'index `tenk2_unique2`, elle est donc appliquée au nœud de jointure. Cela réduit l'estimation du nombre de lignes dans le nœud de jointure, mais ne change aucun parcours d'entrée.

Notez qu'ici le planificateur a choisi de matérialiser la relation interne de la jointure en plaçant un nœud Materialize au-dessus. Cela signifie que le parcours d'index de `t2` ne sera réalisé qu'une seule fois, même si le nœud de jointure par boucle imbriquée va lire dix fois les données, une fois par ligne de la relation externe. Le nœud Materialize conserve les données en mémoire lors de leur première lecture, puis renvoie les données depuis la mémoire à chaque lecture supplémentaire.

Quand vous utilisez des jointures externes, vous pouvez voir des nœuds de plan de jointure avec à la fois des conditions « Join Filter » et « Filter » simples attachées. Les conditions « Join Filter » viennent des clauses de jointures externes `ON`, pour qu'une ligne ne satisfaisant pas la condition « Join Filter » puisse toujours être récupérée comme une ligne NULL. Mais une condition « Filter » simple est appliquée après la règle de jointure externe et supprime donc les lignes de manière inconditionnelle. Dans une jointure interne, il n'y a pas de différence sémantique entre ces types de filtres.

Si nous changeons un peu la sélectivité de la requête, on pourrait obtenir un plan de jointure très différent :

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=230.47..713.98 rows=101 width=488)
    Hash Cond: (t2.unique2 = t1.unique2)
    ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000
 width=244)
    ->  Hash  (cost=229.20..229.20 rows=101 width=244)
          ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20
 rows=101 width=244)
                Recheck Cond: (unique1 < 100)
                ->  Bitmap Index Scan on tenk1_unique1
 (cost=0.00..5.04 rows=101 width=0)
                      Index Cond: (unique1 < 100)
```

Ici, le planificateur a choisi d'utiliser une jointure de hachage, dans laquelle les lignes d'une table sont entrées dans une table de hachage en mémoire, après quoi l'autre table est parcourue et la table de hachage est sondée pour faire correspondre chaque ligne. Notez encore une fois comment l'indentation reflète la structure du plan : le parcours d'index bitmap sur `tenk1` est l'entrée du nœud de hachage, qui construit la table de hachage. C'est alors retourné au nœud de jointure de hachage, qui lit les lignes depuis le plan du fils externe et cherche dans la table de hachage pour chaque ligne.

Un autre type de jointure possible est la jointure d'assemblage, illustrée ici :

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1
 (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.83..200.33 rows=1000 width=244)
```

```
        Sort Key: t2.unique2
        ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000
 width=244)
```

La jointure d'assemblage nécessite que les données en entrée soient triées sur la clé de jointure. Dans ce plan, les données de `tenk1` sont triées grâce à l'utilisation d'un parcours d'index pour visiter les lignes dans le bon ordre, mais un parcours séquentiel suivi d'un tri sont préférables pour `onek`, car il y a beaucoup plus de lignes à visiter dans cette table. (Un parcours séquentiel suivi d'un tri bat fréquemment un parcours d'index pour trier de nombreuses lignes, du fait des accès disques non séquentiels requis par le parcours d'index.)

Une façon de rechercher des plans différents est de forcer le planificateur à oublier certaines stratégies qu'il aurait trouvées moins coûteuses en utilisant les options d'activation (enable)/désactivation (disable) décrites dans la Section 20.7.1 (c'est un outil complexe, mais utile ; voir aussi la Section 14.3). Par exemple, si nous n'étions pas convaincus qu'un parcours séquentiel suivi d'un tri soit la meilleure façon de parcourir la table `onek` dans l'exemple précédent, nous pourrions essayer

```
SET enable_sort = off;
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
 WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
 ------------------------------------------------------------------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1
 (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79
 rows=1000 width=244)
```

ce qui montre que le planificateur pense que le tri de `onek` par un parcours d'index est plus coûteux d'environ 12% par rapport à un parcours séquentiel suivi d'un tri. Bien sûr, la question suivante est de savoir s'il a raison sur ce point. Nous pourrions vérifier cela en utilisant `EXPLAIN ANALYZE`, comme expliqué ci-dessous.

14.1.2. EXPLAIN ANALYZE

Il est possible de vérifier l'exactitude des estimations du planificateur en utilisant l'option `ANALYZE` de `EXPLAIN`. Avec cette option, `EXPLAIN` exécute vraiment la requête, puis affiche le vrai nombre de lignes et les vrais temps passés dans chaque nœud, avec ceux estimés par un simple `EXPLAIN`. Par exemple, nous pourrions avoir un résultat tel que :

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                                  QUERY
 PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488) (actual
 time=0.128..0.377 rows=10 loops=1)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10
 width=244) (actual time=0.057..0.121 rows=10 loops=1)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36
 rows=10 width=0) (actual time=0.024..0.024 rows=10 loops=1)
               Index Cond: (unique1 < 10)
```

```
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91
 rows=1 width=244) (actual time=0.021..0.022 rows=1 loops=10)
          Index Cond: (unique2 = t1.unique2)
 Planning time: 0.181 ms
 Execution time: 0.501 ms
```

Notez que les valeurs « temps réel » sont en millisecondes alors que les estimations de « coût » sont exprimées dans des unités arbitraires ; il y a donc peu de chances qu'elles correspondent. L'information qu'il faut généralement rechercher est si le nombre de lignes estimées est raisonnablement proche de la réalité. Dans cet exemple, les estimations étaient toutes rigoureusement exactes, mais c'est en pratique plutôt inhabituel.

Dans certains plans de requête, il est possible qu'un nœud de sous-plan soit exécuté plus d'une fois. Par exemple, le parcours d'index interne est exécuté une fois par ligne externe dans le plan de boucle imbriquée ci-dessus. Dans de tels cas, la valeur `loops` renvoie le nombre total d'exécutions du nœud, et le temps réel et les valeurs des lignes affichées sont une moyenne par exécution. Ceci est fait pour que les nombres soient comparables avec la façon dont les estimations de coûts sont affichées. Multipliez par la valeur de `loops` pour obtenir le temps total réellement passé dans le nœud. Dans l'exemple précédent, le parcours d'index sur `tenk2` a pris un total de 0,220 milliseconde.

Dans certains cas, `EXPLAIN ANALYZE` affiche des statistiques d'exécution supplémentaires après le temps et le nombre de lignes de l'exécution d'un nœud du plan. Par exemple, les nœuds de tri et de hachage fournissent des informations supplémentaires :

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY
 t1.fivethous;


 QUERY PLAN
--------------------------------------------------------------------------------
 Sort  (cost=717.34..717.59 rows=101 width=488) (actual
 time=7.761..7.774 rows=100 loops=1)
    Sort Key: t1.fivethous
    Sort Method: quicksort  Memory: 77kB
    ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual
 time=0.711..7.427 rows=100 loops=1)
          Hash Cond: (t2.unique2 = t1.unique2)
          ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000
 width=244) (actual time=0.007..2.583 rows=10000 loops=1)
          ->  Hash  (cost=229.20..229.20 rows=101 width=244)
 (actual time=0.659..0.659 rows=100 loops=1)
                Buckets: 1024  Batches: 1  Memory Usage: 28kB
                ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20
 rows=101 width=244) (actual time=0.080..0.526 rows=100 loops=1)
                      Recheck Cond: (unique1 < 100)
                      ->  Bitmap Index Scan on tenk1_unique1
 (cost=0.00..5.04 rows=101 width=0) (actual time=0.049..0.049
 rows=100 loops=1)
                            Index Cond: (unique1 < 100)
 Planning time: 0.194 ms
 Execution time: 8.008 ms
```

Le nœud de tri donne la méthode de tri utilisée (en particulier, si le tri s'est effectué en mémoire ou sur disque) ainsi que la quantité de mémoire ou d'espace disque requis. Le nœud de hachage montre le nombre de paquets de hachage, le nombre de lots ainsi la quantité maximale de mémoire utilisée pour la table de hachage (si le nombre de lots est supérieur à un, il y aura également l'utilisation de l'espace disque impliqué, mais cela n'est pas montré dans cet exemple).

Un autre type d'information supplémentaire est le nombre de lignes supprimées par une condition de filtrage :

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual
 time=0.016..5.107 rows=7000 loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning time: 0.083 ms
 Execution time: 5.905 ms
```

Ces nombres peuvent être particulièrement précieux pour les conditions de filtres appliquées aux nœuds de jointure. La ligne « Rows Removed » n'apparaît que si au moins une ligne parcourue, ou une ligne potentiellement appairée dans le cas d'un nœud de jointure, est rejetée par la condition de filtre.

Un cas similaire aux conditions de filtre apparaît avec des parcours d'index « avec perte ». Par exemple, regardez cette recherche de polygone contenant un point spécifique :

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon
 '(0.5,2.0)';

                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------
 Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32) (actual
 time=0.044..0.044 rows=0 loops=1)
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 4
 Planning time: 0.040 ms
 Execution time: 0.083 ms
```

Le planificateur pense (plutôt correctement) que cette table d'échantillon est trop petite pour s'embêter avec un parcours d'index, et utilise donc un parcours séquentiel dans lequel toutes les lignes sont rejetées par la condition de filtre. Mais si nous forçons l'utilisation d'un parcours d'index, nous voyons :

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon
 '(0.5,2.0)';

                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15
 rows=1 width=32) (actual time=0.062..0.062 rows=0 loops=1)
   Index Cond: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Index Recheck: 1
 Planning time: 0.034 ms
 Execution time: 0.144 ms
```

L'index retourne une ligne candidate, qui est ensuite rejetée par une deuxième vérification de la condition de l'index. Cela arrive car un index GiST est « avec perte » pour les tests de contenance de polygone : il retourne en fait les lignes pour lesquelles les polygones chevauchent la cible, ce qui nécessite après coup un test de contenance exacte sur ces lignes.

`EXPLAIN` a une option `BUFFERS` qui peut être utilisée avec `ANALYZE` pour obtenir encore plus de statistiques d'exécution:

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100
 AND unique2 > 9000;
```

```
                                                                    QUERY
 PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
 (actual time=0.323..0.342 rows=10 loops=1)
    Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
    Buffers: shared hit=15
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0) (actual
 time=0.309..0.309 rows=0 loops=1)
          Buffers: shared hit=7
          ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04
 rows=101 width=0) (actual time=0.043..0.043 rows=100 loops=1)
                Index Cond: (unique1 < 100)
                Buffers: shared hit=2
          ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78
 rows=999 width=0) (actual time=0.227..0.227 rows=999 loops=1)
                Index Cond: (unique2 > 9000)
                Buffers: shared hit=5
 Planning time: 0.088 ms
 Execution time: 0.423 ms
```

Les nombres fournis par BUFFERS aident à identifier les parties de la requête les plus intensives en termes d'entrées sorties.

Il faut garder en tête que comme EXPLAIN ANALYZE exécute vraiment la requête, tous les effets secondaires se produiront comme d'habitude, même si, quel que soit l'affichage de la requête, il est remplacé par la sortie des données d'EXPLAIN. Si vous voulez analyser une requête modifiant les données sans changer les données en table, vous pouvez annuler les modifications après, par exemple :

```
BEGIN;
EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE
 unique1 < 100;

                                                                    QUERY
 PLAN
-------------------------------------------------------------------------------
Update on tenk1  (cost=5.08..230.08 rows=0 width=0) (actual
 time=3.791..3.792 rows=0 loops=1)
  ->  Bitmap Heap Scan on tenk1  (cost=5.08..230.08 rows=102
 width=10) (actual time=0.069..0.513 rows=100 loops=1)
        Recheck Cond: (unique1 < 100)
        Heap Blocks: exact=90
        ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.05
 rows=102 width=0) (actual time=0.036..0.037 rows=300 loops=1)
              Index Cond: (unique1 < 100)
Planning Time: 0.113 ms
Execution Time: 3.850 ms

ROLLBACK;
```

Comme vous pouvez le voir dans cet exemple, quand la requête contient une commande INSERT, UPDATE, DELETE ou MERGE, l'application des changements est faite au niveau du nœud principal Insert, Update, Delete ou Merge du plan. Les nœuds du plan sous celui-ci effectuent le travail de recherche des anciennes lignes et/ou le calcul des nouvelles données. Ainsi, au-dessus, on peut voir les mêmes tris de parcours de bitmap déjà vus précédemment, et leur sortie est envoyée à un nœud de mise à jour qui stocke les lignes modifiées. Il est intéressant de noter que bien que le nœud de modification de données puisse prendre une part considérable sur le temps d'exécution (ici, c'est la partie la plus gourmande), le planificateur n'ajoute rien au coût estimé pour considérer ce travail. C'est

dû au fait que le travail à effectuer est le même pour chaque plan de requête correct, et n'affecte donc pas les décisions du planificateur.

La phrase `Planning time` affichée par `EXPLAIN ANALYZE` correspond au temps pris pour générer et optimiser le plan de requêtes à partir de la requête analysée. Cela n'inclut pas l'analyse syntaxique et la réécriture.

Le `Temps total d'exécution` donné par `EXPLAIN ANALYZE` inclut le temps de démarrage et d'arrêt de l'exécuteur, ainsi que le temps d'exécution de tous les triggers pouvant être déclenchés, mais n'inclut pas les temps d'analyse, de réécriture ou de planification. Le temps passé à exécuter les triggers `BEFORE`, s'il y en a, est inclus dans le temps passé à l'exécution des nœuds Insert, Update ou Delete associés, mais le temps passé à exécuter les triggers `AFTER` n'est pas compté, car les triggers `AFTER` sont déclenchés après l'achèvement du plan entier. Le temps total passé dans chaque trigger (que ce soit `BEFORE` ou `AFTER`) est affiché séparément. Notez que les triggers de contrainte ne seront pas exécutés avant la fin de la transaction et par conséquent ne seront pas affichés du tout par `EXPLAIN ANALYZE`.

14.1.3. Avertissements

Il existe deux raisons importantes pour lesquelles les temps d'exécution mesurés par `EXPLAIN ANALYZE` peuvent dévier de l'exécution normale de la même requête. Tout d'abord, comme aucune ligne n'est réellement envoyée au client, les coûts de conversion réseau et les coûts de formatage des entrées/sorties ne sont pas inclus. Ensuite, le surcoût de mesure induit par `EXPLAIN ANALYZE` peut être significatif, plus particulièrement sur les machines avec un appel système `gettimeofday()` lent. Vous pouvez utiliser l'outil pg_test_timing pour mesurer le surcoût du calcul du temps sur votre système.

Les résultats de `EXPLAIN` ne devraient pas être extrapolés pour des situations autres que celles de vos tests en cours ; par exemple, les résultats sur une petite table ne peuvent être appliqués à des tables bien plus volumineuses. Les estimations de coût du planificateur ne sont pas linéaires et, du coup, il pourrait bien choisir un plan différent pour une table plus petite ou plus grande. Un exemple extrême est celui d'une table occupant une page disque. Vous obtiendrez pratiquement toujours un parcours séquentiel, que des index soient disponibles ou non. Le planificateur réalise que cela va nécessiter la lecture d'un seul bloc disque pour traiter la table dans ce cas, il n'y a donc pas d'intérêt à étendre des lectures de blocs supplémentaires pour un index. (Nous voyons cela arriver dans l'exemple `polygon_tbl` au-dessus.)

Ici, ce sont des cas dans lesquels les valeurs réelles et estimées ne correspondent pas vraiment, mais qui ne sont pas non plus totalement fausses. Un tel cas peut se produire quand un nœud d'exécution d'un plan est arrêté par un `LIMIT` ou autre chose avec un effet similaire. Par exemple, dans la requête `LIMIT` utilisée précédemment,

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2
 > 9000 LIMIT 2;

                                                                  QUERY
 PLAN
-------------------------------------------------------------------------------
 Limit  (cost=0.29..14.71 rows=2 width=244) (actual
 time=0.177..0.249 rows=2 loops=1)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..72.42
 rows=10 width=244) (actual time=0.174..0.244 rows=2 loops=1)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
 Planning time: 0.096 ms
 Execution time: 0.336 ms
```

les estimations de coût et de nombre de lignes pour le nœud de parcours d'index sont affichées comme s'ils devaient s'exécuter jusqu'à la fin. Mais en réalité le nœud Limit arrête la récupération des

lignes après la deuxième ligne récupérée, et donc le vrai nombre de lignes n'est que de 2 et le temps d'exécution est moindre que ne le suggérait le coût estimé. Ce n'est pas une erreur d'estimation, juste une contradiction entre la façon dont l'estimation et les valeurs réelles sont affichées.

Les jointures d'assemblage ont également leurs artefacts de mesure qui peuvent embrouiller une personne non avertie. Une jointure d'assemblage arrêtera la lecture d'une entrée si l'autre entrée est épuisée et que la prochaine valeur clé dans la première entrée est supérieure à la dernière valeur clé de l'autre entrée ; dans un cas comme ça, il ne peut plus y avoir de correspondance et il est donc inutile de parcourir le reste de la première entrée. Cela a donc pour conséquence de ne pas lire entièrement un des fils, avec des résultats similaires à ceux mentionnés pour `LIMIT`. De même, si le fils externe (premier fils) contient des lignes avec des valeurs de clé dupliquées, le fils externe (second fils) est sauvegardé et les lignes correspondant à cette valeur clé sont parcourues de nouveau. `EXPLAIN ANALYZE` compte ces émissions répétées de mêmes lignes internes comme si elles étaient de vraies lignes supplémentaires. Quand il y a de nombreux doublons externes, le nombre réel de lignes affiché pour le nœud de plan du fils interne peut être significativement plus grand que le nombre de lignes qu'il y a vraiment dans la relation interne.

Les nœuds BitmapAnd et BitmapOr affichent toujours un nombre de lignes réelles à 0, du fait des limitations d'implémentation.

Habituellement, la sortie d'`EXPLAIN` affichera chaque nœud de plan généré par le planificateur de requêtes. Néanmoins, il existe des cas où l'exécuteur peut déterminer que certains nœuds n'ont pas besoin d'être exécutés car ils ne produisent aucune ligne. (Actuellement, ceci peut n'arriver qu'aux nœuds enfants du nœud `Append` qui parcourt une table partitionnée.) Quand cela arrive, ces nœuds sont omis de la sortie de la commande `EXPLAIN` et une annotation `Subplans Removed: N` apparaît à la place.

14.2. Statistiques utilisées par le planificateur

14.2.1. Statistiques mono-colonne

Comme nous l'avons vu dans la section précédente, le planificateur de requêtes a besoin d'estimer le nombre de lignes récupérées par une requête pour faire les bons choix dans ses plans de requêtes. Cette section fournit un aperçu sur les statistiques que le système utilise pour ces estimations.

Un élément des statistiques est le nombre total d'entrées dans chaque table et index, ainsi que le nombre de blocs disque occupés par chaque table et index. Cette information est conservée dans la table `pg_class` sur les colonnes `reltuples` et `relpages`. Nous pouvons la regarder avec des requêtes comme celle-ci :

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

Ici, nous pouvons voir que `tenk1` contient 10000 lignes, comme pour ses index, mais que les index sont bien plus petits que la table (ce qui n'est pas surprenant).

Pour des raisons d'efficacité, les valeurs des colonnes `reltuples` et `relpages` ne sont pas mises à jour en temps réel, et contiennent alors souvent des valeurs un peu obsolètes. Elles sont mises à jour

par les commandes VACUUM, ANALYZE et quelques commandes DDL comme CREATE INDEX. Une opération VACUUM ou ANALYZE qui ne parcourt pas la table entièrement (ce qui est le cas le plus fréquent) augmentera de façon incrémentale la valeur de reltuples sur la base de la partie de la table qu'elle a parcourue, résultant en une valeur approximative. Dans tous les cas, le planificateur mettra à l'échelle les valeurs qu'il aura trouvées dans pg_class pour correspondre à la taille physique de la table, obtenant ainsi une approximation plus proche de la réalité.

La plupart des requêtes ne récupèrent qu'une fraction des lignes dans une table à cause de clauses WHERE qui restreignent les lignes à examiner. Du coup, le planificateur a besoin d'une estimation de la *sélectivité* des clauses WHERE, c'est-à-dire la fraction des lignes qui correspondent à chaque condition de la clause WHERE. L'information utilisée pour cette tâche est stockée dans le catalogue système pg_statistic. Les entrées de pg_statistic sont mises à jour par les commandes ANALYZE et VACUUM ANALYZE et sont toujours approximatives même si elles ont été mises à jour récemment.

Plutôt que de regarder directement dans pg_statistic, il vaut mieux voir sa vue pg_stats lors d'un examen manuel des statistiques. La vue pg_stats est conçue pour être plus facilement lisible. De plus, pg_stats est lisible par tous alors que pg_statistic n'est lisible que par un superutilisateur (ceci empêche les utilisateurs sans droits d'apprendre certaines choses sur le contenu des tables appartenant à d'autres personnes à partir des statistiques. La vue pg_stats est restreinte pour n'afficher que les lignes des tables lisibles par l'utilisateur courant). Par exemple, nous pourrions lancer :

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';

 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------
+-----------------------------------
 name    | f         |  -0.363388 | I- 580
 Ramp+
         |           |            | I- 880
 Ramp+
         |           |            | Sp Railroad
   +
         |           |            | I- 580
   +
         |           |            | I- 680
 Ramp
 name    | t         |  -0.284859 | I- 880
 Ramp+
         |           |            | I- 580
 Ramp+
         |           |            | I- 680
 Ramp+
         |           |            | I- 580
   +
         |           |            | State Hwy 13
 Ramp
(2 rows)
```

Notez que deux lignes sont affichées pour la même colonne, une correspondant à la hiérarchie d'héritage complète commençant à la table road (inherited=t), et une autre incluant seulement la table road elle-même (inherited=f).

Les informations stockées dans pg_statistic par ANALYZE, en particulier le nombre maximum d'éléments dans les tableaux most_common_vals et histogram_bounds pour chaque colonne,

peuvent être définies colonne par colonne en utilisant la commande ALTER TABLE SET STATISTICS ou globalement en initialisant la variable de configuration default_statistics_target. La limite par défaut est actuellement de 100 entrées. Augmenter la limite pourrait résulter en des estimations plus précises du planificateur, en particulier pour les colonnes ayant des distributions de données irrégulières, au prix d'un plus grand espace consommé dans pg_statistic et d'un temps plus long pour calculer les estimations. En revanche, une limite plus basse pourrait être suffisante pour des colonnes avec des distributions de données simples.

Le Chapitre 75 donne plus de détails sur l'utilisation des statistiques par le planificateur.

14.2.2. Statistiques étendues

Il est habituel de voir des requêtes lentes tourner avec de mauvais plans d'exécution, car plusieurs colonnes utilisées dans les clauses de la requête sont corrélées. L'optimiseur part normalement du principe que toutes les conditions sont indépendantes les unes des autres, ce qui est faux quand les valeurs des colonnes sont corrélées. Les statistiques classiques, du fait qu'il s'agit par nature de statistiques sur une seule colonne, ne peuvent pas capturer d'information sur la corrélation entre colonnes. Toutefois, PostgreSQL a la possibilité de calculer des *statistiques multivariées*, qui peuvent capturer une telle information.

Comme le nombre de combinaisons de colonnes est très important, il n'est pas possible de calculer les statistiques multivariées automatiquement. À la place, des *objets statistiques étendus*, plus souvent appelés simplement *objets statistiques*, peuvent être créés pour indiquer au serveur qu'il faut obtenir des statistiques sur un ensemble intéressant de colonnes.

Les objets statistiques sont créés en utilisant la commande CREATE STATISTICS. La création de tels objets crée seulement une entrée dans le catalogue pour exprimer l'intérêt dans cette statistique. La vraie récupération de données est effectuée par ANALYZE (soit une commande manuelle, soit une analyse automatique en tâche de fond). Les valeurs collectées peuvent être examinées dans le catalogue pg_statistic_ext_data.

ANALYZE calcule des statistiques étendues basées sur le même ensemble de lignes de la table qu'il utilise pour calculer les statistiques standard sur une seule colonne. Puisque la taille d'échantillon peut être augmentée en augmentant la cible de statistiques de la table ou de n'importe laquelle de ses colonnes (comme décrit dans la section précédente), une plus grande cible de statistiques donnera normalement des statistiques étendues plus précises, mais nécessitera également plus de temps pour les calculer.

La section suivante décrit les types de statistiques étendues qui sont actuellement supportées.

14.2.2.1. Dépendances fonctionnelles

Le type le plus simple de statistiques étendues trace les *dépendances fonctionnelles* , un concept utilisé dans les définitions des formes normales des bases de données. On dit qu'une colonne b est fonctionnellement dépendante d'une colonne a si la connaissance de la valeur de a est suffisante pour déterminer la valeur de b, et donc qu'il n'existe pas deux lignes ayant la même valeur de a avec des valeurs différentes de b. Dans une base de données complètement normalisée, les dépendances fonctionnelles ne devraient exister que sur la clé primaire et les superclés. Toutefois, dans la pratique, beaucoup d'ensembles de données ne sont pas totalement normalisés pour de nombreuses raisons ; une dénormalisation intentionnelle pour des raisons de performances est un exemple courant. Même dans une base de données totalement normalisée, il peut y avoir une corrélation partielle entre des colonnes, qui peuvent être exprimées comme une dépendance fonctionnelle partielle.

L'existence de dépendances fonctionnelles a un impact direct sur la précision de l'estimation pour certaines requêtes. Si une requête contient des conditions à la fois sur des colonnes indépendantes et sur des colonnes dépendantes, les conditions sur les colonnes dépendantes ne réduisent plus la taille du résultat ; mais sans la connaissance de cette dépendance fonctionnelle, l'optimiseur de requêtes supposera que les conditions sont indépendantes, avec pour résultat une taille de résultat sous-estimée.

Pour informer l'optimiseur des dépendances fonctionnelles, ANALYZE peut collecter des mesures sur des dépendances entre colonnes. Évaluer le degré de dépendance entre tous les ensembles de colonnes aurait un coût prohibitif, c'est pourquoi la collecte de données est limitée aux groupes de colonnes apparaissant ensemble dans un objet statistique défini avec l'option dependencies. Il est conseillé de ne créer des dépendences statistiques que pour des groupes de colonnes fortement corrélées, pour éviter un surcoût à la fois dans ANALYZE et plus tard lors de la planification de requête.

Voici un exemple de collecte de statistiques fonctionnellement dépendantes :

```
CREATE STATISTICS stts (dependencies) ON city, zip FROM zipcodes;

ANALYZE zipcodes;

SELECT stxname, stxkeys, stxddependencies
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid =
 stxoid)
  WHERE stxname = 'stts';
 stxname | stxkeys |             stxddependencies
---------+---------+------------------------------------------
 stts    | 1 5     | {"1 => 5": 1.000000, "5 => 1": 0.423130}
(1 row)
```

On peut voir ici que la colonne 1 (zip code) détermine complètement la colonne 5 (city) et que donc le coefficient est 1.0, alors que la ville ne détermine le code postal qu'environ 42% du temps, ce qui veut dire qu'il y a beaucoup de villes (58%) qui sont représentées par plus d'un seul code postal.

Lors du calcul de la sélectivité d'une requête impliquant des colonnes fonctionnellement dépendantes, le planificateur ajoute l'estimation de sélectivité par condition en utilisant les cœfficients de dépendance afin de ne pas produire de résultats sous-estimés.

14.2.2.1.1. Limites des dépendances fonctionnelles

Les dépendances fonctionnelles sont pour le moment uniquement appliquées pour les conditions sur une simple égalité entre une colonne et une valeur constante et des clauses IN contenant des valeurs constantes. Elles ne sont pas utilisées pour améliorer l'estimation sur les conditions d'égalité entre deux colonnes ou la comparaison d'une colonne avec une expression ni pour les clauses d'intervalle, LIKE ou tout autre type de condition.

Lors d'une estimation avec des dépendances fonctionnelles, l'optimiseur part du principe que les conditions sur les colonnes impliquées sont compatibles et donc redondantes. Si elles sont incompatibles, l'estimation correcte devrait être zéro ligne, mais cette possibilité n'est pas envisagée. Par exemple, dans une requête telle que

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip =
 '94105';
```

l'optimiseur négligera la clause city puisqu'elle ne changera pas la sélectivité, ce qui est correct. Par contre, il fera la même supposition pour

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip =
 '90210';
```

bien qu'il n'y ait en réalité aucune ligne satisfaisant cette requête. Toutefois, les statistiques de dépendances fonctionnelles ne fournissent pas suffisamment d'information pour en arriver à cette conclusion.

Pour beaucoup de situations pratiques, cette supposition est généralement correcte ; par exemple, l'application pourrait contenir une interface graphique qui n'autorise que la sélection de villes et codes postaux compatibles pour l'utilisation dans une requête. Mais si ce n'est pas le cas, les dépendances fonctionnelles pourraient ne pas être une solution viable.

14.2.2.2. Nombre N-Distinct multivarié

Les statistiques sur une seule colonne stockent le nombre de valeurs distinctes pour chaque colonne. Les estimations du nombre de valeurs distinctes combinant plus d'une colonne (par exemple, pour `GROUP BY a, b`) sont souvent fausses quand l'optimiseur ne dispose que de données statistiques par colonne, avec pour conséquence le choix de mauvais plans.

Afin d'améliorer de telles estimations, `ANALYZE` peut collecter des statistiques n-distinct pour des groupes de colonne. Comme précédemment, il n'est pas envisageable de le faire pour tous les regroupements possibles, ainsi les données ne sont collectées que pour ceux apparaissant ensemble dans un objet statistique défini avec l'option `ndistinct`. Des données seront collectées pour chaque combinaison possible de deux colonnes ou plus dans l'ensemble de colonnes listées.

En continuant avec l'exemple précédent, le nombre n-distinct dans une table de code postaux pourrait ressembler à ceci :

```
CREATE STATISTICS stts2 (ndistinct) ON city, state, zip FROM
 zipcodes;

ANALYZE zipcodes;

SELECT stxkeys AS k, stxdndistinct AS nd
  FROM pg_statistic_ext join pg_statistic_ext_data on (oid =
 stxoid)
  WHERE stxname = 'stts2';
-[ RECORD
 1 ]-----------------------------------------------------
k  | 1 2 5
nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5":
 33178}
(1 row)
```

Cela indique qu'il y a trois combinaisons de colonnes qui ont 33178 valeurs distinctes : le code postal et l'état ; le code postal et la ville ; et le code postal, la ville et l'état (le fait qu'ils soient tous égaux est attendu puisque le code postal seul est unique dans cette table). D'un autre côté, la combinaison de la ville et de l'état n'a que 27435 valeurs distinctes.

Il est conseillé de créer des objets statistiques `ndistinct` uniquement sur les combinaisons de colonnes réellement utilisées pour des regroupements, et pour lesquelles les mauvaises estimations du nombre de groupe a pour conséquence de mauvais plans. Sinon le temps consommé par `ANALYZE` serait gaspillé.

14.2.2.3. Listes multivariées MCV

Un autre type de statistiques enregistrées pour chaque colonne est les listes des valeurs les plus communes. Ceci permet des estimations très précises pour les colonnes individuelles, mais pourrait résulter en des estimations significativement mauvaises pour les requêtes ayant des filtres sur plusieurs colonnes.

Pour améliorer ces estimations, `ANALYZE` peut récupérer des listes MCV sur des combinaisons de colonnes. De façon similaire aux dépendances fonctionnelles et coefficients de valeurs distinctes, il n'est pas possible de le faire pour chaque regroupement de colonnes. Ceci est encore plus vrai dans ce cas, car la liste MCV (contrairement aux dépendances fonctionnelles et coefficients de valeurs distinctes), enregistre les valeurs les plus communes. Donc les données ne sont récupérées que pour les groupes de colonnes apparaissant dans un objet statistique défini avec l'option `mcv`.

En continuant sur l'exemple précédent, la liste MCV pour une table de codes ZIP pourrait ressembler à ce qui suit (contrairement aux types plus simples de statistiques, une fonction est requise pour inspecter le contenu du MCV) :

```
CREATE STATISTICS stts3 (mcv) ON city, state FROM zipcodes;

ANALYZE zipcodes;

SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid
 = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname =
 'stts3';

 index |         values         | nulls | frequency |
 base_frequency
-------+------------------------+-------+-----------
+----------------
     0 | {Washington, DC}       | {f,f} |  0.003467 |
 2.7e-05
     1 | {Apo, AE}              | {f,f} |  0.003067 |
 1.9e-05
     2 | {Houston, TX}          | {f,f} |  0.002167 |
 0.000133
     3 | {El Paso, TX}          | {f,f} |     0.002 |
 0.000113
     4 | {New York, NY}         | {f,f} |  0.001967 |
 0.000114
     5 | {Atlanta, GA}          | {f,f} |  0.001633 |
 3.3e-05
     6 | {Sacramento, CA}       | {f,f} |  0.001433 |
 7.8e-05
     7 | {Miami, FL}            | {f,f} |    0.0014 |
 6e-05
     8 | {Dallas, TX}           | {f,f} |  0.001367 |
 8.8e-05
     9 | {Chicago, IL}          | {f,f} |  0.001333 |
 5.1e-05
   ...
(99 rows)
```

Ceci indique que la combinaison la plus commune des colonnes city et state est Washington DC, avec la fréquence réelle (dans cet exemple) de 0,35 %. La fréquence de base de la combinaison (telle qu'elle est calculée par les fréquences par mono colonne) est seulement de 0,0027 %, résultant en une sous-estimation très forte.

Il est préférable de créer des objets statistiques MCV uniquement sur les combinaisons de colonnes réellement utilisées ensemble dans des filtres et pour lesquelles la mauvaise estimation du nombre de groupes a pour conséquence de mauvais plans. Dans le cas contraire, le `ANALYZE` et le temps de planification sont juste gâchés.

14.3. Contrôler le planificateur avec des clauses `JOIN` explicites

Il est possible de contrôler le planificateur de requêtes jusqu'à un certain point en utilisant une syntaxe `JOIN` explicite. Pour voir en quoi ceci est important, nous avons besoin d'un peu de théorie.

Dans une simple requête de jointure, telle que :

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

le planificateur est libre de joindre dans n'importe quel ordre les tables indiquées. Par exemple, il peut générer un plan de requête joignant A à B par la condition `WHERE a.id = b.id`, puis joignant C

à cette table jointe par l'autre condition WHERE. Ou il peut joindre B à C, puis le résultat avec A. Ou il peut joindre A à C, puis les joindre avec B, mais cela serait inefficace, puisque le produit cartésien complet de A et C devra être formé alors qu'il n'y a pas de condition applicable dans la clause WHERE pour optimiser cette jointure. (Toutes les jointures dans l'exécuteur PostgreSQL se produisent entre deux tables, il est donc nécessaire de construire le résultat de l'une ou de l'autre de ces façons). Le point important est que ces différentes possibilités de jointures donnent des résultats sémantiquement équivalents, mais peuvent avoir des coûts d'exécution extrêmement différents. Le planificateur va donc toutes les explorer pour trouver le plan de requête le plus efficace.

Quand une requête n'implique que deux ou trois tables, il y a peu d'ordres de jointures à considérer. Mais le nombre d'ordres de jointures possibles grandit de façon exponentielle quand le nombre de tables augmente. Au-delà d'environ dix tables en entrée, il n'est plus possible de faire une recherche exhaustive de toutes les possibilités, et même la planification de six ou sept tables peut prendre une durée gênante. Avec trop de tables en entrée, le planificateur PostgreSQL basculera d'une recherche exhaustive à une recherche *génétique* probabiliste depuis un nombre limité de possibilités (la limite de bascule est définie par le paramètre geqo_threshold). La recherche génétique prend moins de temps, mais elle ne trouvera pas nécessairement le meilleur plan possible.

Quand la requête implique des jointures externes, le planificateur est moins libre qu'avec des jointures internes. Par exemple, considérez :

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id =
 b.id);
```

Bien qu'à première vue les conditions de cette requête semblent similaires à l'exemple précédent, les sémantiques sont différentes, car une ligne doit être émise pour chaque ligne de A qui sans ligne correspondante dans la jointure entre B et C. Le planificateur n'a alors pas le choix dans l'ordre de la jointure : ici, il doit joindre B à C, puis joindre A au résultat. En conséquence, cette requête prend moins de temps à planifier que la requête précédente. Dans d'autres cas, le planificateur peut arriver à déterminer que plus d'un ordre de jointure est sûr. Par exemple, étant donné :

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid
 = c.id);
```

il est valide de joindre A en premier soit à B, soit à C. Actuellement, seul un FULL JOIN contraint complètement l'ordre de jointure. En pratique, la plupart des cas impliquant un LEFT JOIN ou un RIGHT JOIN peuvent être réarrangés jusqu'à un certain degré.

Sémantiquement, la syntaxe d'une jointure interne explicite (INNER JOIN, CROSS JOIN ou JOIN) revient à lister les relations en entrée du FROM, donc sans contraindre l'ordre de la jointure.

Même si la plupart des types de JOIN ne sont pas complètement contraignantes pour l'ordre de jointure, il est possible de forcer le planificateur de requête de PostgreSQL de les considérer comme contraignantes. Par exemple, ces trois requêtes sont logiquement équivalentes :

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND
 b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Mais si nous disons au planificateur d'honorer l'ordre des JOIN, la deuxième et la troisième prendront moins de temps à planifier que la première. Cet effet n'est pas inquiétant pour seulement trois tables, mais cela pourrait bien nous aider avec un nombre important de tables.

Pour forcer le planificateur à suivre l'ordre de jointure des JOIN explicites, passez le paramètre join_collapse_limit à 1 (d'autres valeurs possibles sont discutées plus bas).

Vous n'avez pas besoin de restreindre l'ordre de jointure pour diminuer le temps de recherche car il est correct d'utiliser des opérateurs JOIN parmi les éléments d'une liste FROM. Par exemple, considérez :

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

Avec `join_collapse_limit` = 1, le planificateur est forcé de joindre A à B avant la jointure aux autres tables, mais sans restreindre ses choix par ailleurs. Dans cet exemple, le nombre d'ordres de jointures possibles est réduit par un facteur de cinq.

Cette technique pour restreindre la recherche du planificateur peut servir à réduire les temps de planification et aiguiller le planificateur vers un bon plan de requête. Si le planificateur choisit spontanément un mauvais ordre de jointure, vous pouvez le forcer à choisir un meilleur via la syntaxe `JOIN` -- à supposer que vous connaissiez un meilleur ordre. Il faut tester.

Un problème très proche et affectant le temps de planification est le regroupement de sous-requêtes dans leurs requêtes parentes. Par exemple, considérez :

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE quelquechose) AS ss
WHERE quelquechosedautre;
```

Cette situation peut apparaître lors de l'utilisation d'une vue contenant une jointure ; la règle `SELECT` de la vue sera insérée à la place de la référence de la vue, produisant une requête comme celle ci-dessus. Normalement, le planificateur essaiera de regrouper la sous-requête avec son parent, donnant :

```
SELECT * FROM x, y, a, b, c WHERE quelquechose AND
 quelquechosedautre;
```

Ceci donne généralement un meilleur plan que planifier séparément la sous-requête. (Par exemple, grâce aux clauses `WHERE` externes, joindre X à A peut éliminer d'entrée un bon nombre de lignes de A, évitant ainsi le besoin de générer la totalité de la sous-requête). Mais en même temps, nous avons accru le temps de planification ; ici, nous avons un problème de jointure à cinq tables remplaçant un problème de deux jointures séparées à trois tables. À cause de l'augmentation exponentielle du nombre de possibilités, ceci fait une grande différence. Le planificateur essaie d'éviter de se retrouver coincé dans des problèmes de recherche de grosses jointures en ne regroupant pas une sous-requête si plus de `from_collapse_limit` éléments sont la résultante de la requête parent. Vous pouvez arbitrer entre le temps de planification et la qualité du plan en ajustant ce paramètre à la hausse ou à la baisse.

from_collapse_limit et join_collapse_limit sont nommés de façon similaire car ils font pratiquement la même chose : le premier contrôle le moment où le planificateur « aplatira » les sous-requêtes, et le second contrôle quand aplatir les jointures explicites. Typiquement, vous définirez `join_collapse_limit` à la même valeur que `from_collapse_limit` (de façon à ce que les jointures explicites et les sous-requêtes agissent de la même façon) ou vous affecterez `join_collapse_limit` à 1 (si vous voulez contrôler l'ordre de jointure par des jointures explicites). Mais vous pouvez les définir différemment en essayant de configurer finement la relation entre le temps de planification et le temps d'exécution.

14.4. Remplir une base de données

Vous pourriez avoir besoin d'insérer un grand nombre de données pour remplir une base de données tout au début. Cette section contient quelques suggestions pour réaliser cela de la façon la plus efficace.

14.4.1. Désactivez la validation automatique (autocommit)

Lors d'`INSERT` multiples, désactivez la validation automatique et faites une seule validation à la fin (en SQL, ceci signifie de lancer `BEGIN` au début et `COMMIT` à la fin. Quelques bibliothèques client pourraient le faire derrière votre dos, auquel cas vous devez vous assurer que la bibliothèque le fait quand vous le voulez). Si vous permettez à chaque insertion d'être validée séparément, PostgreSQL fait un gros travail pour chaque ligne ajoutée. Un bénéfice supplémentaire de réaliser toutes les insertions dans une seule transaction est que si l'insertion d'une ligne échoue alors les lignes insérées jusqu'à maintenant seront annulées. Vous ne serez donc pas bloqué avec des données partiellement chargées.

14.4.2. Utilisez COPY

Utilisez l'instruction COPY pour charger toutes les lignes en une seule commande, plutôt que d'utiliser une série de commandes INSERT. La commande COPY est optimisée pour charger un grand nombre de lignes ; elle est moins flexible que INSERT, mais introduit significativement moins de surcharge lors du chargement de grosses quantités de données. Comme COPY est une seule commande, il n'y a pas besoin de désactiver la validation automatique (autocommit) si vous utilisez cette méthode pour remplir une table.

Si vous ne pouvez pas utiliser COPY, utilisez PREPARE pour créer une instruction préparée INSERT, puis utilisez EXECUTE autant de fois que nécessaire. Ceci évite certaines surcharges lors d'une analyse et d'une planification répétées de commandes INSERT. Différentes interfaces fournissent cette fonctionnalité de plusieurs façons ; recherchez « instructions préparées » dans la documentation de l'interface.

Notez que charger un grand nombre de lignes en utilisant COPY est pratiquement toujours plus rapide que d'utiliser INSERT, même si PREPARE ... INSERT est utilisé lorsque de nombreuses insertions sont groupées en une seule transaction.

COPY est plus rapide quand il est utilisé dans la même transaction que la commande CREATE TABLE ou TRUNCATE précédente. Dans ce cas, les journaux de transactions ne sont pas impactés, car, en cas d'erreur, les fichiers contenant les données nouvellement chargées seront supprimés de toute façon. Néanmoins, cette considération ne s'applique que quand wal_level vaut minimal car toutes les commandes doivent écrire dans les journaux de transactions dans les autres cas.

14.4.3. Supprimez les index

Si vous chargez une table tout juste créée, la méthode la plus rapide est de créer la table, de charger en lot les données de cette table en utilisant COPY, puis de créer tous les index nécessaires pour la table. Créer un index sur des données déjà existantes est plus rapide que de mettre à jour de façon incrémentale à chaque ligne ajoutée.

Si vous ajoutez beaucoup de données à une table existante, il pourrait être avantageux de supprimer les index, de charger la table, puis de recréer les index. Bien sûr, les performances de la base de données pour les autres utilisateurs pourraient souffrir tout le temps où les index seront manquants. Vous devez aussi y penser à deux fois avant de supprimer des index uniques, car la vérification d'erreur apportée par la contrainte unique sera perdue tout le temps où l'index est manquant.

14.4.4. Suppression des contraintes de clés étrangères

Comme avec les index, une contrainte de clé étrangère peut être vérifiée « en gros volume » plus efficacement que ligne par ligne. Donc, il pourrait être utile de supprimer les contraintes de clés étrangères, de charger les données et de créer de nouveau les contraintes. De nouveau, il y a un compromis à faire entre la vitesse de chargement des données et la perte de la vérification des erreurs lorsque la contrainte manque.

De plus, quand vous chargez des données dans une table contenant des contraintes de clés étrangères, chaque nouvelle ligne requiert une entrée dans la liste des événements de trigger en attente(puisque c'est le lancement d'un trigger qui vérifie la contrainte de clé étrangère de la ligne). Le chargement de plusieurs millions de lignes peut amener la taille de la file d'attente des triggers à dépasser la mémoire disponible, causant ainsi une mise en mémoire swap intolérable, voire l'échec de la commande. Dans ce cas, il peut être *nécessaire*, pas seulement souhaitable, de supprimer et recréer la clé étrangère lors de chargements de grandes quantités de données. Si la suppression temporaire de la contrainte n'est pas acceptable, le seul recours possible est de découper les opérations de chargement en de plus petites transactions.

14.4.5. Augmentez `maintenance_work_mem`

Augmenter temporairement la variable maintenance_work_mem lors du chargement de grosses quantités de données peut amener une amélioration des performances. Ceci aidera à l'accélération des commandes `CREATE INDEX` et `ALTER TABLE ADD FOREIGN KEY`. Cela ne changera pas grand-chose pour la commande `COPY`. Donc, ce conseil est seulement utile quand vous utilisez une des deux ou les deux techniques ci-dessus.

14.4.6. Augmenter `max_wal_size`

Augmenter temporairement la variable de configuration max_wal_size peut aussi aider à un chargement rapide de grosses quantités de données. Ceci est dû au fait que charger une grosse quantité de données dans PostgreSQL causera la venue très fréquente de checkpoints (la fréquence de ces checkpoints est spécifiée par la variable de configuration `checkpoint_timeout`). Quand survient un checkpoint, toutes les pages modifiées sont écrites sur disque. En augmentant `max_wal_size` temporairement lors du chargement des données, le nombre de checkpoints requis peut être significativement diminué.

14.4.7. Désactiver l'archivage des journaux de transactions et la réplication en flux

Lors du chargement de grosse quantité de données dans une instance qui utilise l'archivage des journaux de transactions ou la réplication en flux, il pourrait être plus rapide de prendre une nouvelle sauvegarde de base après que le chargement ait terminé, plutôt que de traiter une grosse quantité de données incrémentales dans les journaux de transactions. Pour empêcher un accroissement de la journalisation des transactions lors du chargement, vous pouvez désactiver l'archivage et la réplication en flux lors du chargement en configurant wal_level à `minimal`, archive_mode à `off` et max_wal_senders à zéro). Mais notez que le changement de ces paramètres requiert un redémarrage du serveur, et rends toute sauvegarde de base réalisée auparavant inutilisable pour la restauration par archives et pour créer un serveur secondaire, ce qui pourrait amener à des pertes de données.

En dehors d'éviter le temps de traitement des données des journaux de transactions par l'archiveur ou l'émetteur des journaux de transactions, ce paramétrage accélérera certaines commandes où la sous-transaction (ou transaction de plus haut niveau) courante crée ou tronque une table ou un index : elles n'écriront pas du tout dans les journaux de transactions si `wal_level` vaut `minimal`. (Elles peuvent garantir la sûreté des données de façon moins coûteuse en exécutant un `fsync` à la fin plutôt qu'en écrivant les journaux de transactions.)

14.4.8. Lancez `ANALYZE` après

Quand vous avez changé significativement la distribution des données à l'intérieur d'une table, exécuter `ANALYZE` est fortement recommandé. Ceci inclut le chargement de grosses quantités de données dans la table. Exécuter `ANALYZE` (ou `VACUUM ANALYZE`) vous assure que le planificateur dispose de statistiques à jour sur la table. Sans statistiques ou avec des statistiques obsolètes, le planificateur pourrait prendre de mauvaises décisions lors de la planification de la requête, amenant des performances pauvres sur toutes les tables sans statistiques ou avec des statistiques inexactes. Notez que si le démon autovacuum est activé, il pourrait exécuter `ANALYZE` automatiquement ; voir Section 25.1.3 et Section 25.1.6 pour plus d'informations.

14.4.9. Quelques notes sur pg_dump

Les scripts de sauvegarde générés par pg_dump appliquent automatiquement plusieurs des indications ci-dessus, mais pas toutes. Pour recharger une sauvegarde pg_dump aussi rapidement que possible, vous avez besoin de faire quelques étapes supplémentaires manuellement (notez que ces points s'appliquent lors de la *restauration* d'une sauvegarde, et non pas lors de sa *création*. Les mêmes points

s'appliquent soit lors de la restauration d'une sauvegarde texte avec psql soit lors de l'utilisation de pg_restore pour charger un fichier de sauvegarde pg_dump).

Par défaut, pg_dump utilise `COPY` et, lorsqu'il génère une sauvegarde complexe, schéma et données, il est préférable de charger les données avant de créer les index et les clés étrangères. Donc, dans ce cas, plusieurs lignes de conduite sont gérées automatiquement. Ce qui vous reste à faire est de :

- Configurer des valeurs appropriées (c'est-à-dire plus importantes que la normale) pour `maintenance_work_mem` et `max_wal_size`.
- Si vous utilisez l'archivage des journaux de transactions ou la réplication en flux, considérez leur désactivation lors de la restauration. Pour faire cela, configurez `archive_mode` à `off`, `wal_level` à `minimal` et `max_wal_senders` à zéro avant de charger le script de sauvegarde. Après coup, remettez les anciennes valeurs et effectuez une nouvelle sauvegarde de base.
- Tester le mode parallélisé de la sauvegarde et de la restauration des outils pg_dump et pg_restore, et trouver le nombre optimal de tâches parallélisées à utiliser. La sauvegarde et la restauration en parallèle avec l'option `-j` devraient vous donner de meilleures performances.
- Se demander si la sauvegarde complète doit être restaurée dans une seule transaction. Pour cela, passez l'option `-1` ou `--single-transaction` à psql ou pg_restore. Lors de l'utilisation de ce mode, même les erreurs les plus petites annuleront la restauration complète, peut-être en annulant des heures de traitements. Suivant à quel point les données sont en relation, il peut être préférable de faire un nettoyage manuel. Les commandes `COPY` s'exécuteront plus rapidement si vous utilisez une transaction simple et que vous avez désactivé l'archivage des journaux de transaction.
- Si plusieurs processeurs sont disponibles sur le serveur, penser à utiliser l'option `--jobs` de pg_restore. Cela permet la parallélisation du chargement des données et de la création des index.
- Exécuter `ANALYZE` après coup.

Une sauvegarde des données seules utilise toujours `COPY`, mais elle ne supprime ni ne recrée les index et elle ne touche généralement pas les clés étrangères. [1] Donc, lorsque vous chargez une sauvegarde ne contenant que les données, c'est à vous de supprimer et recréer les index et clés étrangères si vous souhaitez utiliser ces techniques. Il est toujours utile d'augmenter `max_wal_size` lors du chargement des données, mais ne vous embêtez pas à augmenter `maintenance_work_mem` ; en fait, vous le ferez lors d'une nouvelle création manuelle des index et des clés étrangères. Et n'oubliez pas `ANALYZE` une fois que vous avez terminé ; voir Section 25.1.3 et Section 25.1.6 pour plus d'informations.

14.5. Configuration avec une perte acceptée

La durabilité est une fonctionnalité des serveurs de bases de données permettant de garantir l'enregistrement des transactions validées même si le serveur s'arrête brutalement, par exemple en cas de coupure électrique. Néanmoins, la durabilité ajoute une surcharge significative. Si votre base de données n'a pas besoin de cette garantie, PostgreSQL peut être configuré pour fonctionner bien plus rapidement. Voici des modifications de configuration que vous pouvez faire pour améliorer les performances dans ce cas. Sauf indication contraire, la durabilité des transactions est garantie dans le cas d'un crash du serveur de bases de données ; seul un arrêt brutal du système d'exploitation crée un risque de perte de données ou de corruption quand ces paramètres sont utilisés.

- Placer le répertoire des données dans un système de fichiers en mémoire (par exemple un disque RAM). Ceci élimine toutes les entrées/sorties disque de la base de données. Cela limite aussi la quantité de mémoire disponible (et peut-être aussi du swap).
- Désactiver fsync ; il n'est pas nécessaire d'écrire les données sur disque.

[1] Vous pouvez obtenir l'effet de désactivation des clés étrangères en utilisant l'option `--disable-triggers` -- mais réalisez que cela élimine, plutôt que repousse, la validation des clés étrangères et qu'il est du coup possible d'insérer des données mauvaises si vous l'utilisez.

- Désactiver synchronous_commit ; il n'est pas forcément nécessaire d'écrire les journaux de transactions WAL à chaque validation de transaction. Ce paramètre engendre un risque de perte de transactions (mais pas de corruption de données) dans le cas d'un arrêt brutal de la *base de données*.
- Désactiver full_page_writes ; il n'est pas nécessaire de se prémunir contre les écritures de pages partielles.
- Augmenter max_wal_size et checkpoint_timeout ; cela réduit les fréquences des checkpoints, mais augmente l'espace disque nécessaire dans `pg_wal`.
- Créer des tables non journalisées pour éviter des écritures dans les journaux de transactions (WAL), bien que cela rende les tables non résistantes à un arrêt brutal.

Chapitre 15. Requêtes parallélisées

PostgreSQL peut préparer des plans de requêtes utilisant plusieurs CPU pour répondre plus rapidement à certaines requêtes. Cette fonctionnalité est connue sous le nom de requêtes parallélisées. Un grand nombre de requêtes ne peuvent pas bénéficier de cette fonctionnalité, soit à cause de la limitation de l'implémentation actuelle, soit parce qu'il n'existe pas de plan imaginable qui soit plus rapide qu'un plan sériel. Néanmoins, pour les requêtes pouvant en bénéficier, l'accélération due à une requête parallélisée est souvent très significative. Beaucoup de ces requêtes peuvent s'exécuter au moins deux fois plus rapidement grâce à la parallélisation, et certaines requêtes quatre fois voire plus. Les requêtes touchant à une grande quantité de données, mais ne retournant que quelques lignes à l'utilisateur sont généralement celles qui bénéficient le plus de cette fonctionnalité. Ce chapitre explique quelques détails sur le fonctionnement des requêtes parallélisées et dans quelles situations elles peuvent être utilisées pour que les utilisateurs intéressés sachent quoi en attendre.

15.1. Comment fonctionne la parallélisation des requêtes

Quand l'optimiseur détermine que la parallélisation est la stratégie la plus rapide pour une requête particulière, il crée un plan d'exécution incluant un *nœud Gather* ou un *nœud Gather Merge*. En voici un exemple simple :

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE filler LIKE '%x%';
                                     QUERY PLAN
------------------------------------------------------------------------------
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..216018.33
 rows=1 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

Dans tous les cas, le nœud `Gather` ou `Gather Merge` aura exactement un nœud enfant, qui est la portion du plan exécutée en parallèle. Si le nœud `Gather` ou `Gather Merge` est à la racine du plan, alors la requête entière est parallélisée. S'il est placé ailleurs dans le plan, alors seule cette portion du plan s'exécutera en parallèle. Dans l'exemple ci-dessus, la requête accède à une seule table, donc il n'existe qu'un seul autre nœud de plan que le nœud `Gather` lui-même ; comme ce nœud est un enfant du nœud `Gather`, il s'exécutera en parallèle.

En utilisant EXPLAIN, vous pouvez voir le nombre de processus auxiliaires (appelés *workers*) choisis par le planificateur. Quand le nœud `Gather` est atteint lors de l'exécution de la requête, le processus en charge de la session demandera un nombre de processus d'arrière-plan (background workers) égal au nombre de workers choisi par le planificateur. Le nombre de *background workers* que le planificateur envisagera est limité à au plus la valeur de max_parallel_workers_per_gather. Le nombre total de *background workers* pouvant exister à un même moment est limité par les paramètres max_worker_processes et max_parallel_workers. De ce fait, il est possible qu'une requête parallélisée s'exécute avec moins de *workers* que prévu, voire sans *worker* du tout. Le plan optimal peut dépendre du nombre de *workers* disponibles, ce qui peut résulter en de médiocres performances des requêtes. Si cela survient fréquemment, envisagez l'augmentation de `max_worker_processes` et de `max_parallel_workers` pour qu'un plus grand nombre de *workers* puissent travailler simultanément ou la diminution de `max_parallel_workers_per_gather` pour que le planificateur réclame moins de *workers*.

Chaque processus *background worker* démarré avec succès dans une requête parallélisée donnée exécutera la portion parallélisée du plan. Le processus principal, appelé *leader*, exécutera aussi cette

portion du plan bien qu'il ait des responsabilités supplémentaires : il doit aussi lire toutes les lignes générées par les *workers*. Quand la portion parallélisée du plan ne génère qu'un petit nombre de lignes, le *leader* se comportera souvent comme un *worker* supplémentaire, accélérant l'exécution de la requête. Par contre, quand la portion parallèle du plan génère un grand nombre de lignes, le *leader* peut être accaparé par la lecture des lignes générées par les *workers* et par le traitement des autres étapes au-dessus du nœud `Gather` ou du nœud `Gather Merge`. Dans de tels cas, le *leader* travaillera très peu sur la portion parallélisée du plan.

Quand le nœud principal de la portion parallélisée est `Gather Merge` au lieu de `Gather`, cela indique que chaque processus exécutant la portion parallélisée du plan produit des lignes triées, et que le processus principal s'occupe de conserver l'ordre des lignes pendant leur assemblage. Par contre, `Gather` lit les lignes de processus d'aide dans n'importe quel ordre, détruisant tout ordre qui aurait pu exister.

15.2. Quand la parallélisation des requêtes peut-elle être utilisée ?

Il existe plusieurs paramètres pouvant empêcher le planificateur de la requête de générer un plan parallélisé quelles que soient les circonstances. Pour faire en sorte que des plans parallélisés puissent être générés, les paramètres suivants doivent être configurés ainsi :

- max_parallel_workers_per_gather doit être configuré à une valeur strictement positive. Ceci est un cas spécial du principe plus général qu'il n'y aura pas plus de *workers* que le nombre configuré via `max_parallel_workers_per_gather`.

De plus, le système ne doit pas fonctionner en mode mono-utilisateur. Comme le système de bases de données entier fonctionne alors avec un seul processus, aucun *background worker* ne sera disponible.

Même quand il est possible, dans l'absolu, de générer des plans pour des requêtes parallélisées, le planificateur n'en générera pas pour une requête donnée si une des conditions suivantes se vérifie :

- La requête écrit des données ou verrouille des lignes de la base. Si une requête contient une opération de modification de données, soit au niveau supérieur, soit dans une CTE, aucun plan parallèle ne peut être généré pour cette requête. Il existe des exceptions : les commandes suivantes qui créent une nouvelle table et la remplissent. Un plan parallélisé peut être utilisé pour le `SELECT` sous-jacent de la requête :
  - `CREATE TABLE ... AS`
  - `SELECT INTO`
  - `CREATE MATERIALIZED VIEW`
  - `REFRESH MATERIALIZED VIEW`
- La requête est susceptible d'être suspendue durant l'exécution. Dans des situations où le système pense qu'une exécution pourrait être partielle ou incrémentale, aucun plan parallèle n'est généré. Par exemple, un curseur créé avec DECLARE CURSOR n'utilisera jamais un plan parallélisé. De façon similaire, une boucle PL/pgSQL de la forme `FOR x IN query LOOP .. END LOOP` n'utilisera jamais un plan parallélisé, car le système est incapable de vérifier que le code dans la boucle peut s'exécuter en toute sécurité avec une requête parallélisée.
- La requête utilise une fonction marquée `PARALLEL UNSAFE` (à parallélisation non sûre). La plupart des fonctions systèmes sont `PARALLEL SAFE` (à parallélisation sûre), mais les fonctions utilisateurs sont marquées `PARALLEL UNSAFE` par défaut. Voir la discussion de Section 15.4.
- La requête est exécutée à l'intérieur d'une autre requête qui est déjà parallélisée. Par exemple, si une fonction appelée par une requête parallélisée exécute elle-même une requête SQL, celle-ci n'utilisera

jamais un plan parallélisé. Ceci est une limitation de l'implémentation actuelle, mais il ne serait pas forcément souhaitable de la supprimer, car cela pourrait mener à ce qu'une seule requête utilise un très grand nombre de processus.

Même quand un plan parallélisé est généré pour une requête donnée, certaines circonstances rendront impossible l'exécution en parallèle. Si cela arrive, le *leader* exécutera tout seul la portion du plan sous le nœud `Gather`, pratiquement comme s'il n'était pas là. Ceci surviendra si une des conditions suivantes est vérifiée :

- Aucun *background worker* ne peut être obtenu à cause de la limitation sur le nombre total de *background workers*, due au paramètre max_worker_processes.

- Aucun background worker ne peut être obtenu à cause de la limitation sur le nombre total de *background workers*, démarrés dans le cadre de requêtes parallèles, qui ne peut pas dépasser max_parallel_workers.

- Le client envoie un message Execute avec un nombre de lignes à récupérer différent de zéro. Voir la discussion sur le protocole de requête étendu. Comme la bibliothèque libpq ne fournit actuellement aucun moyen pour envoyer ce type de message, cela ne peut survenir qu'en utilisant un client qui ne se base pas sur la libpq. Si cela arrive fréquemment, ce pourrait être une bonne idée de configurer max_parallel_workers_per_gather à zéro pour les sessions concernées, pour éviter de générer des plans de requêtes non optimaux s'ils sont exécutés de façon sérialisée.

15.3. Plans parallélisés

Comme chaque *worker* exécute la portion parallélisée du plan jusqu'à la fin, il n'est pas possible de prendre un plan de requête ordinaire et de l'exécuter en utilisant plusieurs *workers*. Chaque *worker* produirait une copie complète du jeu de résultats, donc la requête ne s'exécuterait pas plus rapidement qu'à la normale, et produirait des résultats incorrects. À la place, en interne, l'optimiseur considère la portion parallélisée du plan comme un *plan partiel* ; c'est-à-dire construit pour que chaque processus exécutant le plan ne génère qu'un sous-ensemble des lignes en sortie, et que chacune ait la garantie d'être générée par exactement un des processus participants. Habituellement, cela signifie que le parcours de la table directrice de la requête sera un parcours parallélisable (*parallel-aware*).

15.3.1. Parcours parallélisés

Les types suivants de parcours de table sont actuellement parallélisables :

- Lors d'un *parcours séquentiel parallèle*, les blocs de la table seront divisés en groupe et partagés entre les processus participant au parcours. Chaque processus terminera le parcours de son groupe de blocs avant de demander un groupe supplémentaire.

- Lors d'un *parcours de bitmap parallèle*, un processus est choisi pour être le dirigeant (*leader*). Ce processus effectue le parcours d'un ou plusieurs index et construit un bitmap indiquant quels blocs de la table doivent être visités. Ces blocs sont alors divisés entre les processus participants comme lors d'un parcours d'accès séquentiel. En d'autres termes, le parcours de la table est effectué en parallèle, mais le parcours d'index associé ne l'est pas.

- Lors d'un *parcours d'index parallèle* ou d'un *parcours d'index seul parallèle*, les processus participants lisent les données depuis l'index chacun à leur tour. Actuellement, les parcours d'index parallèles ne sont supportés que pour les index B-tree. Chaque processus réclamera un seul bloc de l'index, et scannera et retournera toutes les lignes référencées par ce bloc ; les autres processus peuvent être en train de retourner des lignes d'un bloc différent de l'index au même moment. Les résultats d'un parcours d'index parallèle sont retournés triés au sein de chaque worker parallèle.

Dans le futur, d'autres types de parcours pourraient supporter la parallélisation, comme les parcours d'index autres que B-tree.

15.3.2. Jointures parallélisées

Tout comme dans les plans non parallélisés, la table conductrice peut être jointe à une ou plusieurs autres tables en utilisant une boucle imbriquée, une jointure par hachage ou une jointure par tri-fusion. Le côté interne de la jointure peut être n'importe quel type de plan non parallélisé par ailleurs supporté par l'optimiseur, pourvu qu'il soit sans danger de le lancer au sein d'un worker parallèle. Suivant le type de jointure, la relation interne peut aussi être un plan parallélisé.

- Dans une *boucle imbriquée*, le côté interne n'est jamais parallèle. Bien qu'il soit exécuté intégralement, c'est efficace si le côté interne est un parcours d'index, car les enregistrements extérieurs sont partagés entre les processus d'aide, et donc aussi les boucles qui recherchent les valeurs dans l'index.

- Dans une *jointure par tri-fusion*, le côté intérieur n'est jamais parallélisé et donc toujours exécuté intégralement. Ce peut être inefficace, surtout s'il faut faire un tri, car le travail et les résultats sont dupliqués dans tous les processus participants.

- Dans une *jointure par hachage* (sans l'adjectif « parallélisée »), le côté intérieur est exécuté intégralement par chaque processus participant pour fabriquer des copies identiques de la table de hachage. Cela peut être inefficace si cette table est grande ou le plan coûteux. Dans une *jointure par hachage parallélisée*, le côté interne est un *hachage parallèle* qui divise le travail sur une table de hachage partagée entre les processus participants.

15.3.3. Agrégations parallélisées

PostgreSQL procède à l'agrégation parallélisée en deux étapes. Tout d'abord, chaque processus de la partie parallélisée de la requête réalise une étape d'agrégation, produisant un résultat partiel pour chaque regroupement qu'il connaît. Ceci se reflète dans le plan par le nœud `PartialAggregate`. Puis les résultats partiels sont transférés au *leader* via le nœud `Gather` ou `Gather Merge`. Enfin, le *leader* réagrège les résultats partiels de tous les *workers* pour produire le résultat final. Ceci apparaît dans le plan sous la forme d'un nœud `Finalize Aggregate`.

Comme le nœud `Finalize Aggregate` s'exécute sur le processus leader, les requêtes produisant un nombre relativement important de groupes en comparaison du nombre de lignes en entrée apparaîtront comme moins favorables au planificateur de requêtes. Par exemple, dans le pire scénario, le nombre de groupes vus par le nœud `Finalize Aggregate` pourrait être aussi grand que le nombre de lignes en entrée traitées par les processus workers à l'étape `Partial Aggregate`. Dans de tels cas, au niveau des performances il n'y a clairement aucun intérêt à utiliser l'agrégation parallélisée. Le planificateur de requêtes prend cela en compte lors du processus de planification et a peu de chances de choisir un agrégat parallélisé sur ce scénario.

L'agrégation parallélisée n'est pas supportée dans toutes les situations. Chaque agrégat doit être à parallélisation sûre et doit avoir une fonction de combinaison. Si l'agrégat a un état de transition de type `internal`, il doit avoir des fonctions de sérialisation et de désérialisation. Voir CREATE AGGREGATE pour plus de détails. L'agrégation parallélisée n'est pas supportée si un appel à la fonction d'agrégat contient une clause `DISTINCT` ou `ORDER BY` ainsi que pour les agrégats d'ensembles triés ou quand la requête contient une clause `GROUPING SETS`. Elle ne peut être utilisée que si toutes les jointures impliquées dans la requête sont dans la partie parallélisée du plan.

15.3.4. Parallel Append

Quand PostgreSQL a besoin de combiner des lignes de plusieurs sources dans un seul ensemble de résultats, il utilise un nœud `Append` ou `MergeAppend`. Cela arrive souvent en implémentant un `UNION ALL` ou en parcourant une table partitionnée. Ces nœuds peuvent être utilisés dans des plans parallélisés aussi bien que dans n'importe quel autre plan. Cependant, dans un plan parallélisé, le planificateur peut utiliser un nœud `Parallel Append` à la place.

Quand un nœud `Append` est utilisé au sein d'un plan parallélisé, chaque processus exécutera les plans enfants dans l'ordre où ils apparaissent, de manière à ce que tous les processus participants coopèrent pour exécuter le premier plan enfant jusqu'à la fin, et passent au plan suivant à peu près au même moment. Quand un `Parallel Append` est utilisé à la place, l'exécuteur va par contre répartir les processus participants aussi uniformément que possible entre ses plans enfants, pour que les multiples plans enfants soient exécutés simultanément. Cela évite la contention, et évite aussi de payer le coût de démarrage d'un plan enfant dans les processus qui ne l'exécutent jamais.

À l'inverse d'un nœud `Append` habituel, qui ne peut avoir des enfants partiels que s'il est utilisé dans un plan parallélisé, un nœud `Parallel Append` peut avoir à la fois des plans enfants partiels et non partiels. Les enfants non partiels seront parcourus par un seul processus, puisque les parcourir plus d'une fois provoquerait une duplication des résultats. Les plans qui impliquent l'ajout de plusieurs ensembles de résultat peuvent alors parvenir à un parallélisme « à gros grains » même si des plans partiels efficaces ne sont pas possibles. Par exemple, soit une requête sur une table partitionnée, qui ne peut être implémentée efficacement qu'en utilisant un index qui ne supporte pas les parcours parallélisés. Le planificateur peut choisir un `Parallel Append` de l'`Index Scan` habituel chaque parcours séparé de l'index devra être exécuté jusqu'à la fin par un seul processus, mais des parcours différents peuvent être exécutés au même moment par des processus différents.

enable_parallel_append peut être utilisé pour désactiver cette fonctionnalité.

15.3.5. Conseils pour les plans parallélisés

Si une requête ne produit pas un plan parallélisé comme attendu, vous pouvez tenter de réduire parallel_setup_cost ou parallel_tuple_cost. Bien sûr, ce plan pourrait bien se révéler plus lent que le plan sériel préféré par le planificateur, mais ce ne sera pas toujours le cas. Si vous n'obtenez pas un plan parallélisé même pour de très petites valeurs de ces paramètres (par exemple après les avoir définis tous les deux à zéro), le planificateur a peut-être une bonne raison pour ne pas le faire pour votre requête. Voir Section 15.2 et Section 15.4 pour des explications sur les causes possibles.

Lors de l'exécution d'un plan parallélisé, vous pouvez utiliser `EXPLAIN (ANALYZE, VERBOSE)` qui affichera des statistiques par *worker* pour chaque nœud du plan. Ce peut être utile pour déterminer si le travail est correctement distribué entre les nœuds du plan et plus généralement pour comprendre les caractéristiques de performance du plan.

15.4. Sécurité de la parallélisation

Le planificateur classifie les opérations impliquées dans une requête comme étant à *parallélisation sûre*, *parallélisation restreinte*, ou *parallélisation non sûre*. Une opération à parallélisation sûre est une opération n'entrant pas en conflit avec une requête parallélisée. Une opération à parallélisation restreinte ne peut pas être exécutée par un *worker* parallélisé, mais peut l'être par le *leader* pendant l'exécution. De ce fait, les opérations à parallélisation restreinte ne peuvent jamais survenir sous un nœud `Gather` ou `Gather Merge`. Une opération à parallélisation non sûre ne peut être exécutée dans une requête parallélisée, y compris au niveau du *leader*. Quand une requête contient quoi que ce soit de non sûr à paralléliser, la parallélisation y est complètement désactivée.

Les opérations suivantes sont toujours à parallélisation restreinte.

- Parcours de CTE (*Common Table Expressions*).
- Parcours de tables temporaires.
- Parcours de tables externes, sauf si le wrapper de données distantes a une API `IsForeignScanParallelSafe` qui indique le contraire.
- Nœuds du plan pour lesquels un `InitPlan` est attaché.
- Nœuds du plan qui référencent un `SubPlan` corrélé.

15.4.1. Marquage de parallélisation pour les fonctions et agrégats

Le planificateur ne peut pas déterminer automatiquement si une fonction ou un agrégat définis par un utilisateur est à parallélisation sûre, restreinte ou non sûre, car cela nécessiterait de pouvoir prédire chaque opération réalisable par la fonction. En général, c'est équivalent au problème de l'arrêt et de ce fait, impossible. Même pour des fonctions simples où cela pourrait se faire, nous n'essayons pas, car ce serait coûteux et sujet à erreurs. À la place, toutes les fonctions définies par des utilisateurs sont supposées à parallélisation non sûre sauf indication contraire. Lors de l'utilisation des instructions CREATE FUNCTION et ALTER FUNCTION, un marquage est possible en spécifiant `PARALLEL SAFE`, `PARALLEL RESTRICTED` ou `PARALLEL UNSAFE` suivant ce qui est approprié. Lors de l'utilisation de CREATE AGGREGATE, l'option `PARALLEL` peut être spécifiée comme `SAFE`, `RESTRICTED` ou `UNSAFE`.

Les fonctions et agrégats doivent être marqués `PARALLEL UNSAFE` s'ils écrivent dans la base, accèdent à des séquences, modifient l'état de la transaction même temporairement (par exemple, une fonction PL/pgSQL qui définit un bloc `EXCEPTION` pour récupérer des erreurs), ou font des modifications persistantes sur les paramètres. De façon similaire, les fonctions doivent être marquées `PARALLEL RESTRICTED` si elles accèdent à des tables temporaires, à l'état de connexion du client, à des curseurs, à des requêtes préparées ou à un quelconque état local du processus serveur que le système ne peut pas synchroniser entre les différents *workers*. Par exemple, `setseed` et `random` sont à parallélisation restreinte pour cette dernière raison.

En général, si une fonction est marquée comme étant sûre alors qu'elle ne l'est pas, ou si elle est marquée restreinte alors que sa parallélisation en fait n'est pas sûre, elle peut être cause d'erreurs ou de réponses fausses lors de l'utilisation dans une requête parallélisée. Les fonctions en langage C peuvent en théorie avoir des comportements indéfinis en cas de mauvais marquage, car le système n'a aucun moyen de se défendre contre du code C arbitraire. Cela étant dit, dans la plupart des cas, le résultat ne sera pas pire qu'avec toute autre fonction. En cas de doute, le mieux est probablement de marquer les fonctions en tant que `UNSAFE`.

Si une fonction exécutée avec un *worker* parallèle acquiert des verrous non détenus par le *leader*, par exemple en exécutant une requête sur une table non référencée dans la requête, ces verrous seront relâchés à la sortie du *worker*, et non pas à la fin de la transaction. Si vous écrivez une fonction qui fait cela et que cette différence de comportement a une importance pour vous, marquez ces fonctions comme `PARALLEL RESTRICTED` pour vous assurer qu'elles ne s'exécutent qu'au sein du *leader*.

Notez que le planificateur de requêtes ne cherche pas à différer l'évaluation des fonctions ou agrégats à parallélisation restreinte impliqués dans la requête pour obtenir un meilleur plan. Donc, par exemple, si une clause `WHERE` appliquée à une table particulière est à parallélisation restreinte, le planificateur ne tentera pas de placer le parcours de cette table dans une portion parallélisée du plan. Dans certains cas, il serait possible (voire efficace) d'inclure le parcours de cette table dans la partie parallélisée de la requête et de différer l'évaluation de la clause `WHERE` afin qu'elle se déroule au-dessus du nœud `Gather`. Néanmoins, le planificateur ne le fait pas.

Partie III. Administration du serveur

Cette partie couvre des sujets intéressant un administrateur de bases de données PostgreSQL, dont l'installation du logiciel, la mise en place et la configuration du serveur, la gestion des utilisateurs et des bases de données et la maintenance. Tout administrateur d'un serveur PostgreSQL, même pour un usage personnel, mais plus particulièrement en production, doit être familier des sujets abordés dans cette partie.

Les informations sont à peu près ordonnées de sorte qu'un nouvel utilisateur puisse les lire linéairement du début à la fin. Cependant, les chapitres sont indépendants et peuvent être lus séparément. L'information est présentée dans un style narratif, regroupée en unités thématiques. Les lecteurs cherchant une description complète d'une commande particulière doivent se référer à la Partie VI.

Les premiers chapitres peuvent être compris sans connaissances préalables. Ainsi, les nouveaux utilisateurs installant leur propre serveur peuvent commencer leur exploration avec cette partie. Le reste du chapitre concerne l'optimisation (*tuning*) et la gestion, et suppose que le lecteur est familier avec l'utilisation du système de bases de données PostgreSQL. Les lecteurs sont encouragés à regarder la Partie I et la Partie II pour des informations complémentaires.

Table des matières

16. Installation depuis les binaires 561
17. Procédure d'installation depuis le code source 562
    17.1. Version courte 562
    17.2. Prérequis 562
    17.3. Obtenir les sources 564
    17.4. Procédure d'installation 564
        17.4.1. Options de `configure` 567
        17.4.2. Variables d'environnement de `configure` 576
    17.5. Initialisation post-installation 579
        17.5.1. Bibliothèques partagées 579
        17.5.2. Variables d'environnement 580
    17.6. Plateformes supportées 581
    17.7. Notes spécifiques à des plateformes 581
        17.7.1. AIX 581
        17.7.2. Cygwin 583
        17.7.3. macOS 583
        17.7.4. MinGW/Windows Natif 584
        17.7.5. Solaris 585
18. Installation à partir du code source sur Windows 587
    18.1. Construire avec Visual C++ ou le Microsoft Windows SDK 587
        18.1.1. Prérequis 588
        18.1.2. Considérations spéciales pour Windows 64 bits 590
        18.1.3. Construction 590
        18.1.4. Nettoyage et installation 591
        18.1.5. Exécuter les tests de régression 591
19. Configuration du serveur et mise en place 593
    19.1. Compte utilisateur PostgreSQL 593
    19.2. Créer un groupe de base de données 593
        19.2.1. Utilisation de systèmes de fichiers secondaires 595
        19.2.2. Systèmes de fichiers 595
    19.3. Lancer le serveur de bases de données 596
        19.3.1. Échecs de lancement 598
        19.3.2. Problèmes de connexion du client 599
    19.4. Gérer les ressources du noyau 599
        19.4.1. Mémoire partagée et sémaphore 599
        19.4.2. systemd RemoveIPC 604
        19.4.3. Limites de ressources 605
        19.4.4. Linux memory overcommit 606
        19.4.5. Pages mémoire de grande taille (*huge pages*) sous Linux 607
    19.5. Arrêter le serveur 608
    19.6. Mise à jour d'une instance PostgreSQL 609
        19.6.1. Mettre à jour les données via pg_dumpall 610
        19.6.2. Mettre à jour les données via pg_upgrade 612
        19.6.3. Mettre à jour les données via la réplication 612
    19.7. Empêcher l'usurpation de serveur (spoofing) 612
    19.8. Options de chiffrement 613
    19.9. Connexions TCP/IP sécurisées avec SSL 614
        19.9.1. Configuration basique 614
        19.9.2. Configuration d'OpenSSL 615
        19.9.3. Utiliser des certificats clients 615
        19.9.4. Utilisation des fichiers serveur SSL 616
        19.9.5. Créer des certificats 616
    19.10. Connexions TCP/IP sécurisées avec le chiffrage GSSAPI 618
        19.10.1. Mise en place de base 618
    19.11. Connexions TCP/IP sécurisées avec des tunnels ssh tunnels 618

19.12. Enregistrer le journal des événements sous Windows 619
20. Configuration du serveur 621
20.1. Paramètres de configuration 621
20.1.1. Noms et valeurs des paramètres 621
20.1.2. Interaction avec les paramètres via le fichier de configuration 622
20.1.3. Interaction avec les paramètres via SQL 623
20.1.4. Interaction avec les paramètre via le shell 623
20.1.5. Gestion du contenu des fichiers de configuration 624
20.2. Emplacement des fichiers 626
20.3. Connexions et authentification 627
20.3.1. Paramètres de connexion 627
20.3.2. Authentification 630
20.3.3. SSL 631
20.4. Consommation des ressources 634
20.4.1. Mémoire 634
20.4.2. Disque 638
20.4.3. Usage des ressources du noyau 639
20.4.4. Report du VACUUM en fonction de son coût 639
20.4.5. Processus d'écriture en arrière-plan 640
20.4.6. Comportement asynchrone 641
20.5. Write Ahead Log 644
20.5.1. Paramètres 644
20.5.2. Points de vérification 649
20.5.3. Archivage 650
20.5.4. Restauration 652
20.5.5. Restauration des archives 652
20.5.6. Cible de restauration 654
20.6. Réplication 655
20.6.1. Serveurs d'envoi 655
20.6.2. Serveur primaire 658
20.6.3. Serveurs secondaire/standby 660
20.6.4. Souscripteurs 663
20.7. Planification des requêtes 664
20.7.1. Configuration de la méthode du planificateur 664
20.7.2. Constantes de coût du planificateur 666
20.7.3. Optimiseur génétique de requêtes 668
20.7.4. Autres options du planificateur 669
20.8. Remonter et tracer les erreurs 671
20.8.1. Où tracer 671
20.8.2. Quand tracer 675
20.8.3. Que tracer 678
20.8.4. Utiliser les journaux au format CSV 683
20.8.5. Utiliser le format de trace JSON 685
20.8.6. Titre des processus 686
20.9. Statistiques d'exécution 686
20.9.1. Statistiques cumulatives sur les requêtes et index 686
20.9.2. Surveillance et statistiques 688
20.10. Nettoyage (`vacuum`) automatique 688
20.11. Valeurs par défaut des connexions client 690
20.11.1. Comportement des instructions 690
20.11.2. Préchargement de bibliothèques partagées 697
20.11.3. Locale et formatage 699
20.11.4. Autres valeurs par défaut 701
20.12. Gestion des verrous 702
20.13. Compatibilité de version et de plateforme 703
20.13.1. Versions précédentes de PostgreSQL 703
20.13.2. Compatibilité entre la plateforme et le client 704
20.14. Gestion des erreurs 705

20.15. Options préconfigurées 706
20.16. Options personnalisées 708
20.17. Options pour les développeurs 708
20.18. Options courtes 713
21. Authentification du client 715
21.1. Le fichier pg_hba.conf 715
21.2. Correspondances d'utilisateurs 724
21.3. Méthodes d'authentification 726
21.4. Authentification trust 726
21.5. Authentification par mot de passe 727
21.6. Authentification GSSAPI 728
21.7. Authentification SSPI 730
21.8. Authentification fondée sur ident 731
21.9. Authentification Peer 731
21.10. Authentification LDAP 732
21.11. Authentification RADIUS 735
21.12. Authentification de certificat 736
21.13. Authentification PAM 736
21.14. Authentification BSD 737
21.15. Problèmes d'authentification 737
22. Rôles de la base de données 739
22.1. Rôles de la base de données 739
22.2. Attributs des rôles 740
22.3. Appartenance d'un rôle 742
22.4. Supprimer des rôles 743
22.5. Rôles prédéfinis 744
22.6. Sécurité des fonctions 746
23. Administration des bases de données 747
23.1. Aperçu 747
23.2. Création d'une base de données 748
23.3. Bases de données modèles 749
23.4. Configuration d'une base de données 750
23.5. Détruire une base de données 750
23.6. Tablespaces 751
24. Localisation 753
24.1. Support des locales 753
24.1.1. Aperçu 753
24.1.2. Comportement 754
24.1.3. Sélectionner les locales 755
24.1.4. Fournisseurs de locale 755
24.1.5. Problèmes 756
24.2. Support des collations 756
24.2.1. Concepts 756
24.2.2. Gestion des collations 758
24.3. Support des jeux de caractères 763
24.3.1. Jeux de caractères supportés 764
24.3.2. Choisir le jeu de caractères 766
24.3.3. Conversion automatique d'encodage entre serveur et client 767
24.3.4. Conversions disponibles entre jeux de caractères 768
24.3.5. Pour aller plus loin 772
25. Planifier les tâches de maintenance 774
25.1. Nettoyages réguliers 774
25.1.1. Bases du VACUUM 774
25.1.2. Récupérer l'espace disque 775
25.1.3. Maintenir les statistiques de l'optimiseur 776
25.1.4. Mettre à jour la carte de visibilité 777
25.1.5. Éviter les cycles des identifiants de transactions 778
25.1.6. Le démon auto-vacuum 783

25.2. Ré-indexation régulière 784
25.3. Maintenance du fichier de traces 785
26. Sauvegardes et restaurations 787
26.1. Sauvegarde SQL 787
26.1.1. Restaurer la sauvegarde 788
26.1.2. Utilisation de pg_dumpall 789
26.1.3. Gérer les grosses bases de données 789
26.2. Sauvegarde de niveau système de fichiers 790
26.3. Archivage continu et récupération d'un instantané (PITR) 791
26.3.1. Configurer l'archivage des journaux de transactions 792
26.3.2. Réaliser une sauvegarde de base 795
26.3.3. Effectuer une sauvegarde de base avec l'API bas niveau 796
26.3.4. Restauration à partir d'un archivage continu 798
26.3.5. Lignes temporelles (*Timelines*) 800
26.3.6. Conseils et exemples 801
26.3.7. Restrictions 802
27. Haute disponibilité, répartition de charge et réplication 803
27.1. Comparaison de différentes solutions 803
27.2. Serveurs secondaires par transfert de journaux 807
27.2.1. Préparatifs 808
27.2.2. Fonctionnement du serveur secondaire 808
27.2.3. Préparer le primaire pour les serveurs secondaires 809
27.2.4. Paramétrer un serveur secondaire 809
27.2.5. Réplication en flux 810
27.2.6. Slots de réplication 812
27.2.7. Réplication en cascade 813
27.2.8. Réplication synchrone 813
27.2.9. Archivage continu côté secondaire 817
27.3. Bascule (*Failover*) 817
27.4. Hot Standby 818
27.4.1. Aperçu pour l'utilisateur 818
27.4.2. Gestion des conflits avec les requêtes 820
27.4.3. Aperçu pour l'administrateur 822
27.4.4. Référence des paramètres de Hot Standby 826
27.4.5. Avertissements 826
28. Superviser l'activité de la base de données 827
28.1. Outils Unix standard 827
28.2. Le système de statistiques cumulatives 828
28.2.1. Configuration de la récupération de statistiques 828
28.2.2. Visualiser les statistiques 829
28.2.3. `pg_stat_activity` 833
28.2.4. `pg_stat_replication` 848
28.2.5. `pg_stat_replication_slots` 850
28.2.6. `pg_stat_wal_receiver` 851
28.2.7. `pg_stat_recovery_prefetch` 852
28.2.8. `pg_stat_subscription` 853
28.2.9. `pg_stat_subscription_stats` 853
28.2.10. `pg_stat_ssl` 853
28.2.11. `pg_stat_gssapi` 854
28.2.12. `pg_stat_archiver` 855
28.2.13. `pg_stat_bgwriter` 855
28.2.14. `pg_stat_wal` 856
28.2.15. `pg_stat_database` 857
28.2.16. `pg_stat_database_conflicts` 859
28.2.17. `pg_stat_all_tables` 859
28.2.18. `pg_stat_all_indexes` 861
28.2.19. `pg_statio_all_tables` 862
28.2.20. `pg_statio_all_indexes` 862

28.2.21. pg_statio_all_sequences 863
28.2.22. pg_stat_user_functions 863
28.2.23. pg_stat_slru 864
28.2.24. Fonctions statistiques 864
28.3. Visualiser les verrous 867
28.4. Rapporter la progression 867
28.4.1. Rapporter la progression d'ANALYZE 867
28.4.2. Rapporter la progression du CREATE INDEX 869
28.4.3. Rapporter la progression du VACUUM 871
28.4.4. Rapporter la progression de CLUSTER 873
28.4.5. Rapporter la progression de la sauvegarde de base 874
28.4.6. COPY Progress Reporting 875
28.5. Traces dynamiques 876
28.5.1. Compiler en activant les traces dynamiques 876
28.5.2. Sondes disponibles 877
28.5.3. Utiliser les sondes 885
28.5.4. Définir de nouvelles sondes 886
29. Surveiller l'utilisation des disques 888
29.1. Déterminer l'utilisation des disques 888
29.2. Panne pour disque saturé 889
30. Fiabilité et journaux de transaction 890
30.1. Fiabilité 890
30.2. Sommes de contrôle pour les données 892
30.2.1. Vérification hors ligne des sommes de contrôle 892
30.3. Write-Ahead Logging (WAL) 892
30.4. Validation asynchrone (Asynchronous Commit) 893
30.5. Configuration des journaux de transaction 895
30.6. Vue interne des journaux de transaction 898
31. Réplication logique 900
31.1. Publication 900
31.2. Abonnement 901
31.2.1. Gestion des slots de réplication 902
31.2.2. Exemples 903
31.3. Filtres de lignes 906
31.3.1. Règles des filtres de lignes 906
31.3.2. Restrictions de l'expression 906
31.3.3. Transformations d'un UPDATE 906
31.3.4. Tables partitionnées 907
31.3.5. Synchronisation initiale des données 907
31.3.6. Combiner plusieurs filtres de lignes 907
31.3.7. Exemples 908
31.4. Listes de colonnes 914
31.4.1. Exemples 915
31.5. Conflits 916
31.6. Restrictions 917
31.7. Architecture 918
31.7.1. Instantané initial 918
31.8. Supervision 919
31.9. Sécurité 919
31.10. Paramètres de configuration 920
31.11. Démarrage rapide 920
32. JIT (compilation à la volée) 922
32.1. Qu'est-ce que le JIT ? 922
32.1.1. JIT Opérations accélérées 922
32.1.2. Inclusion 922
32.1.3. Optimisation 922
32.2. Quand utiliser le JIT ? 922
32.3. Configuration 924

32.4. Extensibilité 924
32.4.1. Support de l'intégration pour les extensions 924
32.4.2. Fournisseur JIT interchangeable 924
33. Tests de régression 926
33.1. Lancer les tests 926
33.1.1. Exécuter les tests sur une installation temporaire 926
33.1.2. Exécuter les tests sur une installation existante 927
33.1.3. Suites supplémentaires de tests 927
33.1.4. Locale et encodage 929
33.1.5. Paramétrages personnalisés du serveur 929
33.1.6. Tests supplémentaires 930
33.2. Évaluation des tests 930
33.2.1. Différences dans les messages d'erreurs 930
33.2.2. Différences au niveau des locales 930
33.2.3. Différences au niveau des dates/heures 931
33.2.4. Différences sur les nombres à virgules flottantes 931
33.2.5. Différences dans l'ordre des lignes 931
33.2.6. Profondeur insuffisante de la pile 932
33.2.7. Test « random » 932
33.2.8. Paramètres de configuration 932
33.3. Fichiers de comparaison de variants 932
33.4. Tests TAP 933
33.5. Examen de la couverture des tests 934

Chapitre 16. Installation depuis les binaires

PostgreSQL est aujourd'hui disponible sous forme de paquets binaires pour la plupart des systèmes d'exploitation courants. Quand ils sont disponibles, ces paquets sont la manière recommandée d'installer PostgreSQL pour les utilisateurs du système. Compiler depuis les sources (voir Chapitre 17) n'est recommandé que pour les développeurs de PostgreSQL ou d'extensions.

Pour une liste à jour des plateformes fournissant des paquets binaires, visitez la section téléchargements du site de PostgreSQL à https://www.postgresql.org/download/, et suivez les instructions pour la plateforme en question.

Chapitre 17. Procédure d'installation depuis le code source

Ce chapitre décrit l'installation de PostgreSQL à partir du code source. Si vous installez un paquet fourni par une distribution, comme un paquet RPM ou Debian, ignorez ce chapitre, et allez lire Chapitre 16 à la place.

Si vous compilez PostgreSQL pour Microsoft Windows, continuez la lecture de ce chapitre si vous avez pour but d'utiliser MinGW ou Cygwin ; par contre, si vous voulez utiliser Visual C++ de Microsoft, lisez à la place Chapitre 18.

17.1. Version courte

```
./configure
make
su
make install
adduser postgres
mkdir -p /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/pg_ctl -D /usr/local/pgsql/data -l logfile
 start
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

Le reste du chapitre est la version longue.

17.2. Prérequis

En général, les plateformes Unix modernes sont capables d'exécuter PostgreSQL. Les plateformes sur lesquelles des tests ont été effectués sont décrites dans la Section 17.6 ci-après.

Les logiciels suivants sont nécessaires pour compiler PostgreSQL :

- GNU make version 3.81 (ou une version plus récente) est nécessaire ; les autres programmes make ou les versions plus anciennes de GNU make *ne* fonctionnent *pas*. (GNU make est parfois installé sous le nom `gmake`). Pour connaître la version utilisée, saisir

  ```
  make --version
  ```

- Il est nécessaire d'avoir un compilateur C ISO/ANSI (au minimum compatible avec C99). Une version récente de GCC est recommandée, mais PostgreSQL est connu pour compiler avec de nombreux compilateurs de différents vendeurs.

- tar est requis pour déballer la distribution des sources, associé à gzip ou bzip2.

- La bibliothèque GNU Readline est utilisée par défaut. Elle permet à psql (l'interpréteur de ligne de commandes SQL de PostgreSQL) de se souvenir de chaque commande saisie, et permet d'utiliser les flèches du clavier pour rappeler et éditer les commandes précédentes. C'est très pratique et fortement recommandé. Si vous n'en voulez pas, vous devrez renseigner l'option `--without-readline` lors de l'appel à la commande `configure`. Une alternative possible est l'utilisation de la bibliothèque `libedit` sous licence BSD, développée au départ sur NetBSD. La bibliothèque

`libedit` est compatible GNU Readline, et est utilisée si cette dernière n'est pas trouvée, ou si l'option `--with-libedit-preferred` est fournie à `configure`. Si vous utilisez une distribution Linux à base de paquets, et que ceux de `readline` et `readline-devel` sont séparés, il faut impérativement installer les deux.

- La bibliothèque de compression zlib est utilisée par défaut. Si vous n'en voulez pas, il faut préciser `--without-zlib` à `configure`. Cela a pour conséquence de désactiver le support des archives compressées dans pg_dump et pg_restore.

Les paquets suivants sont optionnels. S'ils ne sont pas obligatoires lors d'une compilation par défaut de PostgreSQL, ils le deviennent lorsque certaines options sont utilisées, comme cela est expliqué par la suite.

- Pour compiler le langage procédural PL/Perl, une installation complète de Perl, comprenant la bibliothèque `libperl` et les fichiers d'en-tête est nécessaire. La version minimale requise est Perl 5.8.3.

  Comme PL/Perl est une bibliothèque partagée, la bibliothèque `libperl` doit aussi être partagée sur la plupart des plateformes. C'est désormais le choix par défaut dans les versions récentes de Perl, mais ce ne l'était pas dans les versions plus anciennes ; dans tous les cas, c'est du ressort de celui qui a installé Perl chez vous. `configure` échouera si la compilation de PL/Perl est sélectionnée, mais qu'il ne trouve pas une bibliothèque partagée `libperl`. Dans ce cas, vous devrez recompiler et installer Perl manuellement pour être capable de compiler PL/Perl. Lors du processus de configuration pour Perl, demandez une bibliothèque partagée.

  Si vous avez l'intention d'avoir plus qu'une utilisation occasionnelle de PL/Perl, vous devez vous assurer que l'installation de Perl a été faite avec l'option `usemultiplicity` activée (`perl -V` vous indiquera si c'est le cas).

- Pour compiler le langage de programmation serveur PL/Python, il faut que Python soit installé avec les fichiers d'en-tête et le module distutils. La version minimum requise est Python 3.2.

  Puisque PL/Python doit être une bibliothèque partagée, la bibliothèque `libpython` doit l'être aussi sur la plupart des plateformes. Ce n'est pas le cas des installations par défaut de Python compilées à partir des sources, mais une bibliothèque partagée est disponible dans de nombreuses distributions de systèmes d'exploitation. `configure` échouera si la compilation de PL/Python est sélectionnée et qu'il ne peut pas trouver une bibliothèque partagée `libpython`. Cela peut impliquer que vous deviez soit installer des packages supplémentaires, soit recompiler (une partie de) votre installation Python pour fournir cette bibliothèque partagée. Lors de la compilation à partir des sources, lancez le `configure` de Python avec l'option `--enable-shared`.

- Pour compiler le langage procédural PL/Tcl, Tcl doit bien sûr être installé. La version minimale requise est Tcl 8.4.

- Pour activer le support de langage natif (NLS), qui permet d'afficher les messages d'un programme dans une langue autre que l'anglais, une implémentation de l'API Gettext est nécessaire. Certains systèmes d'exploitation l'intègrent (par exemple, Linux, NetBSD, Solaris) ; pour d'autres systèmes, un paquet additionnel peut être téléchargé sur https://www.gnu.org/software/gettext/. Si vous utilisez l'implémentation Gettext des bibliothèques C GNU, certains utilitaires nécessiteront le paquet GNU Gettext. Il n'est pas nécessaire dans les autres implémentations.

- Vous aurez besoin de OpenSSL, si vous voulez utiliser du chiffrement pour vos connexions clientes. OpenSSL est aussi requis pour la génération de nombres aléatoires sur les plateformes qui n'ont pas `/dev/urandom` (sauf Windows). La version minimale requise est la 1.0.1.

- Vous avez besoin de Kerberos, OpenLDAP ou PAM pour bénéficier de l'authentification ou du chiffrement en utilisant ces services.

- Vous avez besoin de LZ4 si vous voulez disposer de la compression de données avec cette méthode ; voir default_toast_compression et wal_compression.

- Vous avez besoin de Zstandard si vous voulez disposer de la compression de données avec cette méthode ; voir wal_compression. La version minimale requise est 1.4.0.

- Pour compiler la documentation PostgreSQL, il existe un ensemble de prérequis séparé ; voir Section J.2.

Si vous compilez depuis une arborescence Git et non d'un paquet des sources publié, ou pour faire du développement au niveau serveur, les paquets suivants seront également nécessaires :

- GNU Flex et Bison sont nécessaires pour compiler à partir d'un export du Git, ou si vous avez changé les fichiers de définition de l'analyseur ou du « scanner ». Au besoin, les versions nécessaires sont Flex 2.5.31 ou ultérieure et Bison 1.875 ou ultérieure. D'autres programmes lex et yacc ne peuvent pas être utilisés.

- Perl 5.8.3 ou ultérieur est aussi nécessaire pour compiler les sources du Git, ou si vous avez changé les fichiers en entrée pour n'importe laquelle des étapes qui utilisent des scripts Perl. Sous Windows, Perl est nécessaire dans tous les cas. Perl est aussi nécessaire pour lancer certains jeux de tests.

Si vous avez besoin de récupérer un paquet GNU, vous le trouverez sur votre site miroir local de GNU (voir https://www.gnu.org/order/ftp.html pour la liste) ou sur ftp://ftp.gnu.org/gnu/.

Il est important de vérifier qu'il y a suffisamment d'espace disque disponible. 350 Mo sont nécessaires pour la compilation et 60 Mo pour le répertoire d'installation. Un groupe de bases de données vide nécessite 40 Mo ; les bases de données prennent environ cinq fois plus d'espace que des fichiers texte contenant les mêmes données. Si des tests de régression sont prévus, 300 Mo supplémentaires sont temporairement nécessaires. Utilisez la commande `df` pour vérifier l'espace disque disponible.

17.3. Obtenir les sources

Le code source de PostgreSQL pour les versions sorties sont disponibles dans la section de téléchargement de notre site web : téléchargement[1]. Téléchargez le fichier `postgresql-`*`version`*`.tar.gz` ou le fichier `postgresql-`*`version`*`.tar.bz2` qui vous intéresse, puis déballez le

```
tar xf
   postgresql-version.tar.bz2
```

Cette commande crée un répertoire `postgresql-`*`version`* contenant les sources de PostgreSQL dans le répertoire courant. Placez-vous dans ce répertoire pour le reste de la procédure d'installation.

Les sources peuvent également être obtenues directement à partir du système de contrôle de version Git. Pour plus d'informations, voir Section I.1 pour plus d'informations.

17.4. Procédure d'installation

1. **Configuration**

   La première étape de la procédure d'installation est de configurer l'arborescence système et de choisir les options qui vous intéressent. Cela se fait en exécutant le script `configure`. Pour une installation par défaut, entrer simplement

   ```
   ./configure
   ```

   Ce script exécutera de nombreux tests afin de déterminer les valeurs de certaines variables dépendantes du système, et de détecter certaines spécificités de votre système d'exploitation. Il créera divers fichiers dans l'arborescence de compilation pour enregistrer ce qui a été trouvé.

[1] https://www.postgresql.org/ftp/source/

Pour garder une arborescence de compilation séparée de celle des sources, `configure` peut être exécuté à partir d'un répertoire hors de l'arborescence des sources, où la compilation s'effectuera. Cette procédure est aussi appelée une compilation de type *VPATH*. Voici comment faire :

```
mkdir build_dir
cd build_dir
/path/to/source/tree/configure [les options vont ici]
make
```

La configuration par défaut compilera le serveur et les outils, ainsi que toutes les applications clientes et les interfaces ne nécessitant qu'un compilateur C. Tous les fichiers seront installés par défaut sous `/usr/local/pgsql`.

Les processus de compilation et d'installation peuvent être personnalisés en fournissant une ou plusieurs options de ligne de commande à `configure`. Généralement, vous allez personnaliser l'endroit de l'installation, ou la liste des fonctionnalités optionnelles à compiler. `configure` a beaucoup d'options, décrites dans Section 17.4.1.

`configure` tient aussi compte de certaines variables d'environnement, comme décrit dans Section 17.4.2. Elle offre d'autres moyens de personnaliser la configuration.

2. **Compilation**

Pour démarrer la compilation, entrez l'un de ces ordres :

```
make
make all
```

(Rappelez-vous qu'il faut GNU make.) La durée de la compilation sera de quelques minutes, et dépend de votre matériel.

Si vous voulez compiler tout ce qui peut l'être, dont la documentation (HTML et pages de manuel) et les modules optionnels (`contrib`), entrez plutôt :

```
make world
```

La dernière ligne affichée doit être :

```
PostgreSQL, contrib, and documentation successfully made. Ready
 to install.
```

Si vous voulez compiler tout ce qui peut être compilé, en incluant les modules supplémentaires (`contrib`), mais sans la documentation, saisissez à la place :

```
make world-bin
```

Si vous voulez lancer la compilation depuis un autre makefile plutôt que manuellement, vous devez désactiver la variable `MAKELEVEL` ou la mettre à zéro, par exemple ainsi :

```
build-postgresql:
```

```
$(MAKE) -C postgresql MAKELEVEL=0 all
```

L'oublier peut mener à d'étranges messages d'erreur, typiquement sur des fichiers d'en-tête manquants.

3. **Tests de régression**

   Si, avant de l'installer, vous voulez tester ce serveur nouvellement compilé, vous devez lancer les tests de régression maintenant. Il s'agit d'une suite de tests pour vérifier que PostgreSQL fonctionne sur votre machine de la manière prévue par ses développeurs. Entrez :

   ```
   make check
   ```

   (Cela ne fonctionnera pas en tant que root ; faites-le en tant qu'utilisateur non privilégié.) Voir Chapitre 33 pour des informations détaillées sur l'interprétation des résultats des tests. Vous pouvez répéter ces tests n'importe quand par la suite en entrant la même commande.

4. **Installer les fichiers**

   > **Note**
   >
   > Si vous mettez à jour un système existant, assurez-vous de lire Section 19.6, qui contient des instructions sur la mise à jour d'une instance.

   Pour installer PostgreSQL, entrez :

   ```
   make install
   ```

   Cela installera les fichiers dans les répertoires spécifiés dans Étape 1. Assurez-vous que vous avez les droits nécessaires pour y écrire. Normalement, vous devez faire cela en tant que root. Une alternative est de créer les répertoires cibles par avance, et de vous arranger pour obtenir les permissions adéquates.

   Si vous voulez tout construire, sauf la documentation, saisissez à la place :

   ```
   make install-world-bin
   ```

   Pour installer la documentation (HTML et pages de manuel), entrez :

   ```
   make install-docs
   ```

   Si vous avez entré `make world` plus haut, entrez plutôt :

   ```
   make install-world
   ```

   Cela va installer aussi la documentation.

   Vous pouvez aussi utiliser `make install-strip` au lieu de `make install` pour débarrasser les fichiers exécutables et les bibliothèques de leurs informations de débogage lors

de l'installation. Cela économisera un peu d'espace disque. Dans une compilation avec le support du débogage, cette purge va supprimer ce support ; ce n'est donc à faire que s'il n'y a plus besoin de débogage. `install-strip` réussit assez bien à économiser de l'espace, mais ne sait pas toujours effacer le moindre octet inutile d'un exécutable ; pour récupérer tout l'espace disque possible, vous devrez donc terminer manuellement.

L'installation standard fournit tous les fichiers d'en-tête nécessaires au développement d'applications clientes, comme pour celui côté serveur, par exemple pour des fonctions spécifiques ou des types de données codés en C.

Installation cliente : Si vous voulez installer uniquement les applications clientes et les bibliothèques d'interface, vous pouvez utiliser ces commandes :

```
make -C src/bin install
make -C src/include install
make -C src/interfaces install
make -C doc install
```

`src/bin` contient quelques binaires utilisables uniquement sur le serveur, mais ils sont petits.

Désinstallation : Pour annuler l'installation, utilisez la commande `make uninstall`. Cependant, cela ne supprimera pas tous les répertoires qui ont été créés.

Nettoyage : Après l'installation, vous pouvez libérer de l'espace disque en supprimant les fichiers compilés de l'arborescence des sources avec la commande `make clean`. Cela préservera les fichiers créés par `configure`, pour que vous puissiez tout recompiler plus tard avec `make`. Pour réinitialiser l'arbre des sources dans l'état où il est distribué, utilisez `make distclean`. Si vous voulez compiler pour plusieurs plateformes au sein de la même arborescence, vous devez le lancer et reconfigurer pour chaque plateforme. (Une alternative est d'utiliser une arborescence pour chaque plateforme, pour qu'elles ne soient pas modifiées.)

Si, après compilation, vous découvrez que vos options à `configure` étaient fausses, ou si vous changez quelque chose que `configure` a pris en compte (par exemple, par une mise à jour logicielle), il est conseillé de faire `make distclean` avant de reconfigurer et recompiler. Sans cela, vos choix de configuration pourraient ne pas se propager à tous les endroits nécessaires.

17.4.1. Options de `configure`

Les paramètres en ligne de commande à `configure` sont expliqués ci-dessous. Cette liste n'est pas exhaustive (utilisez `./configure --help` pour en avoir une qui le soit). Les options non évoquées ici sont destinées à des utilisations avancées, comme la compilation croisée, et figurent dans la documentation standard d'Autoconf.

17.4.1.1. Emplacements de l'installation

Ces options contrôlent où `make install` va poser les fichiers. L'option `--prefix` est suffisante dans la plupart des cas. Pour des besoins spécifiques, vous pouvez personnaliser les sous-répertoires d'installation avec d'autres options décrites dans cette section. Attention : changer les emplacements relatifs des différents sous-répertoires peut rendre l'installation non déplaçable, c'est-à-dire que vous ne pourrez plus la déplacer par la suite. (Les emplacements pour `man` et `doc` ne sont pas concernés par cette restriction.) Pour obtenir des installations déplaçables, vous pouvez utiliser l'option `--disable-rpath` décrite plus bas.

`--prefix=`*`PREFIX`*

Installe tous les fichiers dans le répertoire *`PREFIX`* au lieu du répertoire `/usr/local/pgsql`. Les fichiers eux-mêmes seront installés dans divers sous-répertoires ; aucun fichier ne sera directement installé sous *`PREFIX`*.

`--exec-prefix=`*`EXEC-PREFIX`*

Les fichiers qui dépendent de l'architecture peuvent être installés dans un répertoire avec un préfixe différent, *`EXEC-PREFIX`*, différent de celui donné par *`PREFIX`*. Cela peut être utile pour partager entre plusieurs machines les fichiers indépendants de l'architecture. S'il est omis, *`EXEC-PREFIX`* est égal à *`PREFIX`*, et les fichiers dépendants seront installés sous la même arborescence que les fichiers indépendants de l'architecture, ce qui est probablement le but recherché.

`--bindir=`*`REPERTOIRE`*

Indique le répertoire des exécutables. Par défaut, il s'agit de *`EXEC-PREFIX`*`/bin`, ce qui signifie `/usr/local/pgsql/bin`.

`--sysconfdir=`*`REPERTOIRE`*

Précise le répertoire de divers fichiers de configuration, par défaut *`PREFIX`*`/etc`.

`--libdir=`*`REPERTOIRE`*

Précise le répertoire d'installation des bibliothèques et des modules chargeables dynamiquement. Par défaut, il s'agit de *`EXEC-PREFIX`*`/lib`.

`--includedir=`*`REPERTOIRE`*

Précise le répertoire d'installation des fichiers d'en-tête C et C++. Par défaut, il s'agit de *`PREFIX`*`/include`.

`--datarootdir=`*`REPERTOIRE`*

Indique le répertoire racine de différents types de fichiers de données en lecture seule. Cela ne sert qu'à paramétrer des valeurs par défaut pour certaines des options suivantes. La valeur par défaut est *`PREFIX`*`/share`.

`--datadir=`*`REPERTOIRE`*

Indique le répertoire pour les fichiers de données en lecture seule utilisés par les programmes installés. La valeur par défaut est *`DATAROOTDIR`*. NB : cela n'a aucun rapport avec l'endroit où les fichiers de base de données seront placés.

`--localedir=`*`REPERTOIRE`*

Indique le répertoire pour installer les données de localisation, en particulier les fichiers des catalogues de traduction des messages. La valeur par défaut est *`DATAROOTDIR`*`/locale`.

`--mandir=`*`REPERTOIRE`*

Les pages de manuel fournies avec PostgreSQL seront installées sous ce répertoire, dans leur sous-répertoire `man`*`x`* respectif. Par défaut, il s'agit de *`DATAROOTDIR`*`/man`.

`--docdir=`*`RÉPERTOIRE`*

Configure le répertoire racine pour installer les fichiers de documentation, sauf les pages « man ». Ceci ne positionne la valeur par défaut que pour les options suivantes. La valeur par défaut pour cette option est *`DATAROOTDIR`*`/doc/postgresql`.

`--htmldir=`*`RÉPERTOIRE`*

La documentation de PostgreSQL, formatée en HTML, sera installée dans ce répertoire. La valeur par défaut est *`DATAROOTDIR`*.

Note

Une attention toute particulière a été prise afin de rendre possible l'installation de PostgreSQL dans des répertoires partagés (comme `/usr/local/include`), sans interférer avec l'espace de noms du reste du système. En premier lieu, le mot « `/postgresql` » est automatiquement ajouté aux répertoires `datadir`, `sysconfdir` et `docdir`, à moins que le nom du répertoire à partir de la racine ne contienne déjà le mot « `postgres` » ou « `pgsql` ». Par exemple, si `/usr/local` est choisi comme préfixe, la documentation sera installée dans `/usr/local/doc/postgresql` ; mais si le préfixe est `/opt/postgres`, alors ce sera dans `/opt/postgres/doc`. Les fichiers d'en-tête C publics pour les interfaces clientes sont installés sous `includedir`, et sont indépendants de l'espace de noms du système. Les fichiers d'en-tête internes et ceux du serveur sont installés dans des répertoires privés sous `includedir`. Voir la documentation de chaque interface pour savoir comment obtenir ces fichiers d'en-tête. Enfin, si nécessaire, un répertoire privé sera aussi créé sous `libdir`, pour les modules chargeables dynamiquement.

17.4.1.2. Fonctionnalités de PostgreSQL

Les options décrites dans cette section permettent d'ajouter diverses fonctionnalités de PostgreSQL qui ne sont pas compilées par défaut ; pour la plupart à cause du besoin d'autres logiciels, comme décrit dans Section 17.2.

`--enable-nls[=`*`LANGUES`*`]`

Active le support des langues natives (NLS), c'est-à-dire la capacité d'afficher les messages d'un programme dans une langue autre que l'anglais. *`LANGUES`* est une liste optionnelle des codes de langue que vous voulez supporter, séparés par une espace ; par exemple, `--enable-nls='de fr'` (l'intersection entre la liste et l'ensemble des langues traduites actuellement sera calculée automatiquement). En l'absence de liste, toutes les traductions disponibles seront installées.

Pour utiliser cette option, une implémentation de l'API Gettext est nécessaire.

`--with-perl`

Permet la compilation du langage côté serveur PL/Perl

`--with-python`

Permet la compilation du langage côté serveur PL/Python.

`--with-tcl`

Permet la compilation du langage côté serveur PL/Tcl.

`--with-tclconfig=`*`REPERTOIRE`*

Tcl installe le fichier `tclConfig.sh`, qui contient des informations de configuration nécessaires pour compiler le module d'interfaçage avec Tcl. Ce fichier est normalement trouvé automatiquement à un emplacement connu, mais pour utiliser une version différente de Tcl, il faut indiquer le répertoire où chercher `tclConfig.sh`.

`--with-icu`

Compiler avec le support de la bibliothèque ICU, activant les collations ICU (voir Section 24.2). Le paquet ICU4C doit être installé. Sa version minimum nécessaire est actuellement la 4.2.

Par défaut, pkg-config sera utilisé pour déterminer les options de compilation nécessaires. Cela est supporté pour ICU4C version 4.6 et suivantes. Pour des versions plus anciennes, ou si pkg-

config n'est pas disponible, les variables `ICU_CFLAGS` et `ICU_LIBS` peuvent être fournies à `configure`, comme dans cet exemple :

```
./configure ... --with-icu ICU_CFLAGS='-I/some/where/include'
 ICU_LIBS='-L/some/where/lib -licui18n -licuuc -licudata'
```

(Si ICU4C est dans le chemin par défaut du compilateur, vous aurez besoin de préciser des chaînes non vides pour éviter l'appel à pkg-config, par exemple `ICU_CFLAGS=' '`.)

`--with-llvm`

Compile avec le support de JIT, basé sur LLVM (voir Chapitre 32). Ceci nécessite l'installation de la bibliothèque LLVM. Sa version minimale requise est actuellement la 3.9.

`llvm-config` sera utilisé pour trouver les options de compilation nécessaires. `llvm-config`, puis `llvm-config-$major-$minor` pour toutes les versions supportées, seront recherchés dans votre `PATH`. Au cas où le bon programme n'est pas trouvé, il faut utiliser la variable `LLVM_CONFIG` pour spécifier le chemin du bon `llvm-config`. Par exemple :

```
./configure ... --with-llvm LLVM_CONFIG='/path/to/llvm/bin/llvm-
config'
```

Le support de LLVM nécessite un compilateur `clang` compatible (à spécifier, si nécessaire, avec la variable d'environnement `CLANG`), et un compilateur C++ fonctionnel (à spécifier, si nécessaire, avec la variable d'environnement `CXX`).

`--with-lz4`

Compile avec le support de la compression LZ4.

`--with-zstd`

Compile avec le support de la compression Zstandard.

`--with-ssl=`*`LIBRARY`*

Compile avec le support pour les connexions SSL (avec chiffrement). Le seul *`LIBRARY`* supporté est `openssl`. Ceci nécessite que le paquet OpenSSL soit installé. `configure` vérifiera les fichiers d'en-tête et les bibliothèques pour s'assurer que votre installation d'OpenSSL est suffisante avant de continuer.

`--with-openssl`

Équivalent obsolète de `--with-ssl=openssl`.

`--with-gssapi`

Compile avec le support de l'authentification GSSAPI. Sur beaucoup d'environnements, le système GSSAPI (habituellement une partie de l'installation Kerberos) n'est pas installé dans un endroit recherché par défaut (par exemple `/usr/include` ou `/usr/lib`) ; vous devez donc ajouter aussi les options `--with-includes` et `--with-libraries`. `configure` vérifiera les fichiers d'en-tête et les bibliothèques pour s'assurer que votre installation de GSSAPI est suffisante avant de continuer.

`--with-ldap`

Compile avec le support de LDAP pour l'authentification et la recherche des paramètres de connexion (voir Section 34.18 et Section 21.10). Sur Unix, cela requiert l'installation du paquet

OpenLDAP. Sur Windows, la bibliothèque WinLDAP est utilisée par défaut. `configure` vérifiera l'existence des fichiers d'en-tête et des bibliothèques nécessaires pour s'assurer que votre installation d'OpenLDAP est suffisante avant de continuer.

`--with-pam`

Compile avec le support de PAM (*Pluggable Authentication Modules*).

`--with-bsd-auth`

Compile avec le support de l'authentification BSD. (Le framework BSD Authentication n'est actuellement disponible que sur OpenBSD.)

`--with-systemd`

Compile avec le support du système de notifications systemd. Ceci améliore l'intégration si le serveur est démarré par systemd, mais n'a pas d'impact sinon. Voir Section 19.3 pour plus de détails. libsystemd et les fichiers d'en-tête associés doivent être installés pour utiliser cette option.

`--with-bonjour`

Compile avec le support du service de découverte automatique Bonjour. Cela nécessite le support de Bonjour dans votre système d'exploitation. Recommandé sur macOS.

`--with-uuid=`*`LIBRARY`*

Compile le module uuid-ossp (qui fournit des fonctions pour générer des UUID), en utilisant la bibliothèque UUID spécifiée. *`LIBRARY`* doit correspondre à une de ces valeurs :

- `bsd` pour utiliser les fonctions UUID trouvées dans FreeBSD et d'autres systèmes dérivés de BSD
- `e2fs` pour utiliser la bibliothèque UUID créée par le projet `e2fsprogs` ; cette bibliothèque est présente sur la plupart des systèmes Linux et sur macOS, et peut être obtenue sur d'autres plateformes également
- `ossp` pour utiliser la bibliothèque OSSP UUID[2]

`--with-ossp-uuid`

Équivalent obsolète de `--with-uuid=ossp`.

`--with-libxml`

Compile avec libxml2, activant ainsi le support de SQL/XML. Une version 2.6.23 ou ultérieure de libxml2 est requise pour cette fonctionnalité.

Pour détecter les options requises pour le compilateur et l'éditeur de liens, PostgreSQL va demander à `pkg-config`, s'il est installé et s'il connaît libxml2. Sinon, le programme `xml2-config`, qui est installé par libxml2, sera utilisé s'il est trouvé. L'utilisation de `pkg-config` est préférée, parce qu'elle gère mieux les installations multiarchitectures.

Pour utiliser une installation libxml2 située dans un emplacement inhabituel, vous pouvez configurer les variables d'environnement relatives à `pkg-config` (voir sa documentation), ou configurer la variable d'environnement `XML2_CONFIG` pour qu'elle pointe sur le programme `xml2-config` appartenant à l'installation libxml2, ou configurer les variables `XML2_CFLAGS` et `XML2_LIBS`. (Si `pkg-config` est installé, alors, pour surcharger son idée de l'emplacement de libxml2, vous devez renseigner soit `XML2_CONFIG`, soit `XML2_CFLAGS` et `XML2_LIBS`, avec des chaînes non vides.)

[2] http://www.ossp.org/pkg/lib/uuid/

`--with-libxslt`

Compile avec libxslt, activant le module xml2 pour opérer des transformations XSL sur du XML. `--with-libxml` doit être spécifié aussi.

17.4.1.3. Anti-Fonctionnalités

Les options décrites dans cette section permettent de désactiver certaines fonctionnalités de PostgreSQL compilées par défaut, mais que vous pouvez désactiver si ne sont pas disponibles un logiciel ou des fonctionnalités système nécessaires. L'utilisation de ces options n'est pas recommandée si ce n'est pas vraiment nécessaire.

`--without-readline`

Empêche l'utilisation de la bibliothèque Readline (et libedit par la même occasion). Cette option désactive l'édition de la ligne de commande et l'historique dans psql.

`--with-libedit-preferred`

Favorise l'utilisation de la bibliothèque libedit (licence BSD). Cette option n'est importante que si vous avez les deux librairies installées ; le défaut dans ce cas est d'utiliser Readline.

`--without-zlib`

Empêche l'utilisation de la bibliothèque Zlib. Cela désactive le support des archives compressées dans pg_dump et pg_restore.

`--disable-spinlocks`

Autorise le succès de la compilation même si PostgreSQL n'a pas le support des spinlocks pour le CPU de la plateforme. Ce manque de support entraînera des performances très faibles ; du coup, cette option ne devra être utilisée que si la compilation échoue et vous informe du manque de support des spinlocks sur votre plateforme. Si cette option est nécessaire pour y compiler PostgreSQL, merci de rapporter le problème à ses développeurs.

`--disable-atomics`

Désactive l'utilisation des opérations atomiques sur le CPU. Cette option ne fait rien sur les plateformes qui n'ont pas ces opérations. Sur celles qui l'ont, cela entraînera de mauvaises performances. Cette option n'est utile que pour le débogage ou pour des comparatifs de performance.

`--disable-thread-safety`

Désactive la sécurité des threads pour les bibliothèques clients. Ceci interdit aux threads concurrents dans les programmes libpq et ECPG de contrôler en toute sécurité leurs pointeurs de connexion privés. N'utiliser que sur les plateformes avec un support des threads défaillant.

17.4.1.4. Détails du processus de compilation

`--with-includes=`*`RÉPERTOIRES`*

`RÉPERTOIRES` est une liste de répertoires, séparés par le caractère deux points (:), qui seront ajoutés à la liste de ceux où le compilateur recherche des fichiers d'en-tête. Si vous avez des paquets optionnels (comme GNU Readline) installés dans un emplacement non conventionnel, vous devez utiliser cette option, et probablement aussi l'option correspondante `--with-libraries`.

Exemple : `--with-includes=/opt/gnu/include:/usr/sup/include`.

`--with-libraries=`*`RÉPERTOIRES`*

`RÉPERTOIRES` est une liste de répertoires, séparés par le caractère deux points (:), où chercher des bibliothèques de fonctions. Si vous avez des paquets installés dans des emplacements non conventionnels, vous devez utiliser cette option (et probablement aussi l'option correspondante `--with-includess`).

Exemple : `--with-libraries=/opt/gnu/lib:/usr/sup/lib`.

`--with-system-tzdata=`*`RÉPERTOIRE`*

PostgreSQL inclut sa propre base de données des fuseaux horaires, nécessaire pour les opérations sur les dates et les heures. Cette base de données est en fait compatible avec la base de fuseaux horaires IANA fournie par de nombreux systèmes d'exploitation comme FreeBSD, Linux et Solaris, donc il semble redondant de l'installer une nouvelle fois. Quand cette option est utilisée, la base des fuseaux horaires fournie par le système, dans *`RÉPERTOIRE`*, est utilisée à la place de celle incluse dans la distribution des sources de PostgreSQL. *`RÉPERTOIRE`* doit être indiqué avec un chemin absolu. `/usr/share/zoneinfo` est un répertoire courant sur certains systèmes d'exploitation. Notez que la routine d'installation ne détectera pas les données de fuseau horaire différentes ou erronées. Si vous utilisez cette option, il est conseillé de lancer les tests de régression pour vérifier que les données de fuseau horaire que vous pointez fonctionnent correctement avec PostgreSQL.

Cette option est surtout destinée aux distributeurs de paquets binaires, qui connaissent bien leur système d'exploitation. Le principal avantage de cette option est que le paquet de PostgreSQL n'aura pas besoin de mise à jour à chaque changement des règles des fuseaux horaires. Un autre avantage est que PostgreSQL peut être cross-compilé plus simplement si les fichiers des fuseaux horaires n'ont pas besoin d'être construits lors de l'installation.

`--with-extra-version=`*`CHAÎNE`*

Ajoute *`CHAÎNE`* au numéro de version de PostgreSQL. Par exemple, vous pouvez utiliser cela pour marquer des binaires compilés depuis des snapshots Git, ou contenant des patchs, avec une chaîne supplémentaire, comme un identifiant `git describe` ou un numéro de version de distribution du paquet.

`--disable-rpath`

N'indique pas aux exécutables de PostgreSQL qu'ils doivent chercher les bibliothèques partagées dans le répertoire des bibliothèques de l'installation (voir `--libdir`). Sur la plupart des plateformes, cette indication utilise un chemin absolu vers le répertoire des bibliothèques, et sera inutile si vous déplacez l'installation plus tard. Cependant, vous devrez alors fournir aux exécutables un autre moyen pour trouver les bibliothèques partagées. Typiquement, cela implique de configurer l'éditeur de liens du système d'exploitation pour les rechercher ; voir Section 17.5.1 pour plus de détails.

17.4.1.5. Divers

Il est assez courant, particulièrement pour les compilations de test, de modifier le numéro de port avec l'option `--with-pgport`. Les autres options de cette section ne sont recommandées que pour les utilisateurs avancés.

`--with-pgport=`*`PORT`*

Positionne *`PORT`* comme numéro de port pour les serveurs et les clients. Le défaut est 5432. Le port peut toujours être changé plus tard ; mais, si vous le spécifiez ici, serveur et clients auront le même défaut dès la compilation, ce qui peut être très pratique. D'habitude, la seule bonne raison de sélectionner une autre valeur que le défaut est si vous avez l'intention de faire tourner plusieurs serveurs PostgreSQL sur la même machine.

`--with-krb-srvnam=`*`NOM`*

Le nom par défaut du service principal Kerberos utilisé par GSSAPI. `postgres` est le défaut. D'habitude, il n'y a aucune raison de changer cela, à moins que vous ne compiliez pour un environnement Windows auquel cas ce doit être, en majuscules, `POSTGRES`.

`--with-segsize=`*`SEGSIZE`*

Définit la taille d'un segment (*segment size*), en gigaoctets. Au niveau du système d'exploitation, les grandes tables sont divisées en plusieurs fichiers, chacun d'une taille égale à la taille d'un segment. Cela évite des problèmes avec les limites de taille de fichiers qui existent sur beaucoup de plateformes. La taille par défaut, 1 gigaoctet, est une valeur sûre pour toutes les plateformes supportées. Si votre système d'exploitation supporte les fichiers de grande taille (« largefile »), et la plupart le font, de nos jours, vous pouvez utiliser une plus grande taille de segment. Ce peut être utile pour réduire le nombre de descripteurs de fichiers consommés en travaillant avec de très grandes tables. Mais faites attention à ne pas choisir une valeur plus large que ce qui est supporté par votre plateforme et les systèmes de fichiers que vous voulez utiliser. D'autres outils que vous pourriez vouloir utiliser, comme tar, peuvent aussi poser des limites sur la taille de fichier utilisable. Il est recommandé que cette valeur soit une puissance de 2, même si ce n'est pas absolument nécessaire. Notez que changer cette valeur casse la compatibilité entre bases au niveau fichier, ce qui veut dire que vous ne pouvez pas utiliser `pg_upgrade` pour mettre à jour vers une version compilée avec une taille de segment différente.

`--with-blocksize=`*`TAILLEBLOC`*

Définit la taille de bloc (*block size*), en kilooctets. C'est l'unité de stockage et d'entrée-sortie dans les tables. Le défaut, 8 kilooctets, est adéquat pour la plupart des situations ; mais d'autres valeurs peuvent être utiles dans certains cas. La valeur peut être une puissance de 2 entre 1 et 32 (kilooctets). Notez que changer cette valeur casse la compatibilité entre bases au niveau fichier, ce qui veut dire que vous ne pouvez pas utiliser `pg_upgrade` pour mettre à jour vers une version compilée avec une taille de bloc différente.

`--with-wal-blocksize=`*`TAILLEBLOC`*

Définit la taille de bloc dans les journaux de transaction (*WAL block size*), en kilooctets. C'est l'unité de stockage et d'entrée-sortie en leur sein. Le défaut, 8 kilooctets, convient pour la plupart des situations ; mais d'autres valeurs peuvent être utiles dans certains cas. La valeur doit être une puissance de 2 entre 1 et 64 (kilooctets). Notez que changer cette valeur casse la compatibilité entre bases au niveau fichier, ce qui veut dire que vous ne pouvez pas utiliser `pg_upgrade` pour mettre à jour vers une version compilée avec une taille de bloc de WAL différente.

17.4.1.6. Options pour les développeurs

La plupart des options de cette section n'ont d'intérêt que pour développer ou déboguer PostgreSQL. Elles ne sont pas recommandées pour la production, sauf `--enable-debug`, qui peut être utile pour obtenir des rapports de bugs détaillés, dans l'éventualité malheureuse où vous rencontriez un bug. Sur les plateformes supportant DTrace, `--enable-dtrace` peut raisonnablement être utilisé en production.

Pour compiler une installation destinée à développer du code au sein du serveur, il est recommandé d'utiliser au moins les options `--enable-debug` et `--enable-cassert`.

`--enable-debug`

Compile tous les programmes et bibliothèques avec les symboles de débogage. Cela signifie que vous pouvez exécuter les programmes au sein d'un débogueur pour analyser les problèmes. Cela augmente considérablement la taille des exécutables et, avec des compilateurs autres que GCC, désactive habituellement les optimisations du compilateur, provoquant des ralentissements. Cependant, avoir les symboles en place est extrêmement utile pour traiter d'éventuels problèmes. Actuellement, cette option n'est recommandée pour les installations en production que si vous

utilisez GCC. Néanmoins, vous devriez toujours l'utiliser si vous développez, ou si vous utilisez une version bêta.

`--enable-cassert`

Active les vérifications des assertions (*assertion*) dans le serveur, qui testent de nombreuses conditions qui « ne peuvent pas arriver ». C'est inestimable pour le développement du code, mais les tests peuvent ralentir le serveur considérablement. Activer ces tests ne va pas améliorer la stabilité de votre serveur ! Les tests des assertions ne sont pas triés par sévérité, et un petit bug relativement inoffensif, s'il déclenche un échec d'assertion, peut mener à des redémarrages du serveur ! Cette option n'est pas recommandée en production, mais vous devriez l'avoir en développement, ou en utilisant une version bêta.

`--enable-tap-tests`

Active les tests avec les outils TAP de Perl. Cela nécessite une installation de Perl et de son module `IPC::Run`. Voir Section 33.4 pour plus d'information.

`--enable-depend`

Active le suivi automatique des dépendances. Avec cette option, les makefiles sont conçus pour que tous les fichiers objets soient recompilés si n'importe quel fichier d'en-tête change. C'est utile si vous faites du développement, mais n'est que gaspillage si vous ne devez compiler qu'une fois pour installer. Pour le moment, cette option ne fonctionne qu'avec GCC.

`--enable-coverage`

Si vous utilisez GCC, tous les programmes et bibliothèques sont compilés avec de l'instrumentation de test de couverture de code. Quand ils sont exécutés, ils génèrent des fichiers dans le répertoire de compilation avec des métriques de couverture de code. Voir Section 33.5 pour davantage d'informations. Cette option ne doit être utilisée qu'avec GCC et en développement.

`--enable-profiling`

En cas d'utilisation de GCC, tous les programmes et bibliothèques sont compilés pour pouvoir être profilés. À la sortie du processus serveur, un sous-répertoire sera créé pour contenir le fichier `gmon.out` contenant les données de profilage. Cette option n'est à utiliser qu'avec GCC et en développement.

`--enable-dtrace`

Compile PostgreSQL avec le support de l'outil de trace dynamique, DTrace. Voir Section 28.5 pour plus d'informations.

Pour pointer vers le programme `dtrace`, la variable d'environnement `DTRACE` peut être configurée. Ce sera souvent nécessaire, car `dtrace` est typiquement installé sous `/usr/sbin`, qui peut ne pas être dans votre `PATH`.

Des options supplémentaires en ligne de commande pour `dtrace` peuvent être indiquées dans la variable d'environnement `DTRACEFLAGS` pour le programme `dtrace`. Sur Solaris, pour inclure le support de DTrace dans un exécutable 64 bits, ajoutez l'option `DTRACEFLAGS="-64"`. Par exemple, en utilisant le compilateur GCC :

```
./configure CC='gcc -m64' --enable-dtrace DTRACEFLAGS='-64' ...
```

En utilisant le compilateur de Sun :

```
./configure CC='/opt/SUNWspro/bin/cc -xtarget=native64' --
enable-dtrace DTRACEFLAGS='-64' ...
```

17.4.2. Variables d'environnement de `configure`

En plus des options de ligne de commande ordinaires décrites plus haut, `configure` répond à nombre de variables d'environnement. Vous pouvez les spécifier sur la ligne de commande de `configure`, par exemple ainsi :

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

Dans ce cas, une variable d'environnement est peu différente d'une option de ligne de commande. Vous pouvez aussi les placer au préalable :

```
export CC=/opt/bin/gcc
export CFLAGS='-O2 -pipe'
./configure
```

Cette utilisation peut être pratique parce que les scripts de configuration de beaucoup de programmes répondent à ces variables de manière similaire.

Les plus utilisées de ces variables d'environnement sont `CC` et `CFLAGS`. Si vous préférez utiliser un compilateur C différent de celui choisi par `configure`, positionnez la variable `CC` vers le compilateur de votre choix. Par défaut, `configure` choisira `gcc` s'il est disponible, et sinon celui par défaut sur la plateforme (habituellement `cc`). De façon similaire, vous pouvez repositionner les options par défaut du compilateur à l'aide de la variable `CFLAGS`.

Voici une liste des variables importantes qui sont configurables de cette façon :

`BISON`

programme Bison

`CC`

compilateur C

`CFLAGS`

options à passer au compilateur C

`CLANG`

chemin vers le programme `clang` utilisé pour optimiser le code source lors de la compilation avec `--with-llvm`

`CPP`

préprocesseur C

`CPPFLAGS`

options à passer au préprocesseur C

`CXX`

compilateur C++

`CXXFLAGS`

options à passer au compilateur C++

`DTRACE`

emplacement du programme `dtrace`

`DTRACEFLAGS`

options à passer au programme `dtrace`

`FLEX`

programme Flex

`LDFLAGS`

options à utiliser lors de l'édition des liens des exécutables et des bibliothèques partagées

`LDFLAGS_EX`

options supplémentaires valables uniquement lors de l'édition des liens des exécutables

`LDFLAGS_SL`

options supplémentaires valables uniquement lors de l'édition des liens des bibliothèques partagées

`LLVM_CONFIG`

programme `llvm-config` à utiliser pour localiser l'installation LLVM.

`MSGFMT`

programme `msgfmt` pour le support des langues

`PERL`

programme interpréteur Perl. Il sera utilisé pour déterminer les dépendances pour la compilation de PL/Perl. La valeur par défaut est `perl`.

`PYTHON`

chemin complet vers l'interpréteur Python. Il sera utilisé pour déterminer les dépendances pour la compilation de PL/Python. S'il n'est pas configuré, les chemins suivants sont testés dans cet ordre : `python3 python`.

Si vous préférez utiliser un compilateur C différent de ceux listés par `configure`, positionnez la variable d'environnement `CC` pour qu'elle pointe sur le compilateur de votre choix. Par défaut, `configure` pointe sur `gcc` s'il est disponible, sinon il utilise celui par défaut de la plateforme (habituellement `cc`). De façon similaire, vous pouvez repositionner les options par défaut du compilateur à l'aide de la variable `CFLAGS`.

Les variables d'environnement peuvent être indiquées sur la ligne de commande `configure`, par exemple :

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

Voici une liste des variables importantes qui sont configurables de cette façon :

`BISON`

programme Bison

`CC`

compilateur C

`CFLAGS`

options à passer au compilateur C

`CPP`

préprocesseur C

`CPPFLAGS`

options à passer au préprocesseur C

`DTRACE`

emplacement du programme `dtrace`

`DTRACEFLAGS`

options à passer au programme `dtrace`

`FLEX`

programme Flex

`LDFLAGS`

options à utiliser lors de l'édition des liens des exécutables et des bibliothèques partagées

`LDFLAGS_EX`

options supplémentaires valables uniquement lors de l'édition des liens des exécutables

`LDFLAGS_SL`

options supplémentaires valables uniquement lors de l'édition des liens des bibliothèques partagées

`MSGFMT`

programme `msgfmt` pour le support des langues

`PERL`

chemin complet vers l'interpréteur Perl. Il sera utilisé pour déterminer les dépendances pour la compilation de PL/Perl.

`PYTHON`

programme interpréteur Python. Il sera utilisé pour déterminer les dépendances pour la compilation de PL/Python. De plus, si Python 2 ou 3 est spécifié ici (ou implicitement choisi), il détermine la variante de PL/Python qui devient disponible. Voir Section 46.10 pour plus d'informations. Si ce paramètre n'est pas en place, seront testés dans cet ordre : `python python3 python2`.

`TCLSH`

programme interpréteur Tcl. Il sera utilisé pour déterminer les dépendances pour la compilation de PL/Tcl. Si ce paramètre n'est pas en place, seront testés dans cet ordre : `tclsh tcl tclsh8.6 tclsh86 tclsh8.5 tclsh85 tclsh8.4 tclsh84`.

`XML2_CONFIG`

programme `xml2-config` utilisé pour trouver l'emplacement de l'installation de libxml2.

Parfois, ajouter des options de compilation après coup à celles choisies par `configure` peut se révéler utile. Un exemple parlant concerne l'option `-Werror` de gcc, qui ne peut pas être incluse dans la variable `CFLAGS` passée à `configure`, car elle casserait un grand nombre de tests internes de `configure`. Pour ajouter de telles options, incluez-les dans la variable d'environnement `COPT` lors de l'exécution de `make`. Le contenu de `COPT` est ajouté aux variables `CFLAGS` et `LDFLAGS` configurées par `configure`. Par exemple, vous pouvez faire :

```
make COPT='-Werror'
```

ou

```
export COPT='-Werror'
make
```

> **Note**
>
> Si vous utilisez GCC, il est préférable de compiler avec un niveau d'optimisation d'au moins `-O1`, parce que l'absence d'optimisation (`-O0`) désactive aussi certains messages importants du compilateur (comme l'utilisation de variables non initialisées). Néanmoins, les niveaux d'optimisations peuvent compliquer le débogage : un pas-à-pas sur le code compilé ne correspondra pas forcément directement aux lignes de code. Si vous avez du mal à déboguer du code optimisé, recompilez les fichiers qui vous intéressent avec `-O0`. Une façon simple de le faire est de passer une option à make: `make PROFILE=-O0 file.o`.
>
> En fait, les variables d'environnement `COPT` et `PROFILE` sont gérées de façon identique par les makefiles de PostgreSQL. Laquelle utiliser est une affaire de préférence, mais l'usage parmi les développeurs est d'utiliser `PROFILE` pour les ajustements ponctuels, alors que `COPT` peut être conservée en permanence.

17.5. Initialisation post-installation

17.5.1. Bibliothèques partagées

Sur certains systèmes gérant des bibliothèques partagées, il faut spécifier comment trouver les nouvelles bibliothèques partagées. Les systèmes sur lesquels ce n'est *pas* nécessaire comprennent FreeBSD, HP-UX, Linux, NetBSD, OpenBSD et Solaris.

La méthode pour le faire varie selon la plateforme, mais la plus répandue consiste à positionner la variable d'environnement `LD_LIBRARY_PATH` ainsi : avec les shells Bourne (`sh`, `ksh`, `bash`, `zsh`) :

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

ou en `csh` ou `tcsh` :

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Remplacez `/usr/local/pgsql/lib` par la valeur donnée à `--libdir` dans l'Étape 1. Vous pouvez mettre ces commandes dans un script de démarrage tel que `/etc/profile` ou `~/.bash_profile`. De bons conseils sur les mises en garde associées à cette méthode peuvent être trouvés sur http://xahlee.info/UnixResource_dir/_/ldpath.html.

Sur certains systèmes, il peut être préférable de renseigner la variable d'environnement `LD_RUN_PATH` *avant* la compilation.

Avec Cygwin, placez le répertoire des bibliothèques dans la variable `PATH`, ou déplacez les fichiers `.dll` dans le répertoire `bin`.

En cas de doute, référez-vous aux pages de man de votre système (peut-être `ld.so` ou `rld`). Si vous avez ultérieurement un message tel que

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or
 directory
```

alors cette étape est vraiment nécessaire. Occupez-vous en alors.

Si votre système d'exploitation est Linux et que vous avez les accès de superutilisateur, vous pouvez exécuter :

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(ou le répertoire équivalent) après l'installation pour permettre à l'éditeur de liens de trouver les bibliothèques partagées plus rapidement. Référez-vous aux pages man portant sur `ldconfig` pour plus d'informations. Pour les systèmes d'exploitation FreeBSD, NetBSD et OpenBSD, la commande est :

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

Les autres systèmes d'exploitation ne sont pas connus pour avoir de commande équivalente.

17.5.2. Variables d'environnement

Si l'installation a été réalisée dans `/usr/local/pgsql` ou à un autre chemin hors des répertoires où par défaut sont recherchés les exécutables, vous devez ajouter `/usr/local/pgsql/bin` (ou le répertoire fourni à `--bindir` au moment de l'Étape 1) dans votre `PATH`. À strictement parler, ce n'est pas une obligation, mais cela rendra l'utilisation de PostgreSQL plus confortable.

Pour ce faire, ajoutez ce qui suit dans le fichier d'initialisation de votre shell, comme `~/.bash_profile` (ou `/etc/profile`, si vous voulez que tous les utilisateurs l'aient) :

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

Si vous utilisez le `csh` ou le `tcsh`, alors utilisez la commande :

```
set path = ( /usr/local/pgsql/bin $path )
```

Pour que votre système trouve la documentation man, il vous faut ajouter des lignes telles que celles qui suivent à votre fichier d'initialisation du shell, à moins que vous installiez ces pages dans un répertoire où elles sont recherchées normalement :

```
MANPATH=/usr/local/pgsql/share/man:$MANPATH
export MANPATH
```

Les variables d'environnement `PGHOST` et `PGPORT` indiquent aux applications clientes l'hôte et le port du serveur de base. Elles surchargent les valeurs utilisées lors de la compilation. Si vous exécutez des applications clientes à distance, alors il est plus pratique que tous les utilisateurs prévoyant d'utiliser la base de données paramètrent `PGHOST`. Ce n'est pas une obligation, cependant, la configuration peut être communiquée via les options de lignes de commande à la plupart des programmes clients.

17.6. Plateformes supportées

Une plateforme (c'est-à-dire une combinaison d'un processeur et d'un système d'exploitation) est considérée comme supportée par la communauté des développeurs de PostgreSQL si le code permet le fonctionnement sur cette plateforme, et que la compilation et les tests de régression ont été récemment validés sur cette plateforme. Actuellement, la plupart des tests de compatibilité de plateforme se font automatiquement par des machines de tests dans la ferme de compilation de PostgreSQL[3]. Si vous êtes intéressé par l'utilisation de PostgreSQL sur une plateforme qui n'est pas représentée dans la ferme de compilation, mais pour laquelle le code fonctionne ou peut fonctionner, nous vous suggérons fortement de monter une machine qui sera membre de la ferme pour que la compatibilité puisse être assurée dans la durée.

En général, PostgreSQL doit fonctionner sur les architectures processeur suivantes : x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, ARM, MIPS, MIPSEL et PA-RISC. Un support du code existe pour M68K, M32R et VAX, mais ces architectures n'ont pas été testées récemment à notre connaissance. Il est souvent possible de compiler PostgreSQL sur un type de processeur non supporté en précisant `--disable-spinlocks` ; mais les performances seront mauvaises.

De manière générale, PostgreSQL doit fonctionner sur les systèmes d'exploitation suivants : Linux (toutes les distributions récentes), Windows (XP et ultérieurs), FreeBSD, OpenBSD, NetBSD, macOS, AIX, HP/UX et Solaris. D'autres systèmes de type Unix peuvent aussi fonctionner, mais ne sont pas testés pour le moment. Dans la plupart des cas, toutes les architectures processeurs supportées par un système d'exploitation donné fonctionneront. Cherchez dans le répertoire Section 17.7 ci-dessous pour voir s'il y a des informations spécifiques à votre système d'exploitation, tout particulièrement dans le cas d'un ancien système.

Si vous avez des problèmes d'installation sur une plateforme connue comme supportée d'après des résultats récents de la ferme de compilation, merci de rapporter cette information à `<pgsql-bugs@lists.postgresql.org>`. Si vous êtes intéressé pour porter PostgreSQL sur une nouvelle plateforme, `<pgsql-hackers@lists.postgresql.org>` est l'endroit approprié pour en discuter.

17.7. Notes spécifiques à des plateformes

Cette section documente des problèmes spécifiques additionnels liés à des plateformes, en ce qui concerne l'installation et le paramétrage de PostgreSQL. Assurez-vous de lire aussi les instructions d'installation, et en particulier Section 17.2. Par ailleurs, consultez Chapitre 33 à propos de l'interprétation des tests de régression.

Les plateformes qui ne sont pas traitées ici n'ont pas de problèmes d'installation spécifiques connus.

17.7.1. AIX

Pour construire PostgreSQL à partir des sources sur macOS, vous aurez besoin d'installer les outils développeur en ligne de commande d'Apple, ce qui se fait en exécutant

```
xcode-select --install
```

[3] https://buildfarm.postgresql.org/

(notez que cela affichera une fenêtre graphique pour confirmation). Vous pouvez aussi installer Xcode.

Vous pouvez utiliser GCC ou le compilateur natif d'IBM `xlc` pour construire PostgreSQL sur AIX.

Les versions d'AIX avant la 7.1 ne sont plus testés et supportées par la communauté PostgreSQL.

17.7.1.1. Gestion de la mémoire

AIX est particulier dans la façon dont il gère la mémoire. Vous pouvez avoir un serveur avec des gigaoctets de mémoire libre, et malgré tout avoir des erreurs de mémoire insuffisante ou des erreurs d'espace d'adressage quand vous lancez des applications. Un exemple est le chargement d'extensions qui échoue avec des erreurs inhabituelles. Par exemple, en exécutant en tant que propriétaire de l'installation PostgreSQL :

```
=# CREATE EXTENSION plperl;
ERROR:  could not load library "/opt/dbs/pgsql/lib/plperl.so": A
 memory address is not in the address space for the process.
```

En l'exécutant en tant que non-propriétaire, mais dans le groupe propriétaire de l'installation PostgreSQL :

```
=# CREATE EXTENSION plperl;
ERROR:  could not load library "/opt/dbs/pgsql/lib/plperl.so": Bad
 address
```

On a un autre exemple avec les erreurs *out of memory* dans les traces du serveur PostgreSQL, avec toutes les allocations de mémoire vers 256 Mo ou plus qui échouent.

La cause générale de ces problèmes est le nombre de bits et le modèle mémoire utilisés par le processus serveur. Par défaut, tous les binaires compilés sur AIX sont 32 bits. Cela ne dépend pas du matériel ou du noyau en cours d'utilisation. Ces processus 32 bits sont limités à 4 Go de mémoire, présentée en segments de 256 Mo utilisant un modèle parmi quelques-uns. Le modèle par défaut permet moins de 256 Mo dans le tas, comme il partage un seul segment avec la pile.

Dans le cas de l'exemple `plperl` ci-dessus, vérifiez votre umask et les droits des binaires de l'installation PostgreSQL. Les binaires de l'exemple étaient en 32 bits et installés en mode 750 au lieu de 755. En raison de ces droits, seul le propriétaire ou un membre du groupe propriétaire peut charger la bibliothèque. Puisqu'il n'est pas lisible par tout le monde, le chargeur place l'objet dans le tas du processus au lieu d'un segment de mémoire de bibliothèque où il aurait été sinon placé.

La solution « idéale » est d'utiliser une version 64-bits de PostgreSQL, mais ce n'est pas toujours pratique, parce que les systèmes équipés de processeurs 32 bits peuvent compiler, mais pas exécuter, des binaires 64 bits.

Si un binaire 32 bits est souhaité, positionnez `LDR_CNTRL` à `MAXDATA=0x`*n*`0000000`, où 1<=n <= 8 avant de démarrer le serveur PostgreSQL, et essayez différentes valeurs et paramètres de `postgresql.conf` pour trouver une configuration qui fonctionne de façon satisfaisante. Cette utilisation de `LDR_CNTRL` notifie à AIX que vous voulez que le serveur réserve `MAXDATA` octets pour le tas, alloués en segments de 256 Mo. Quand vous avez trouvé une configuration utilisable, `ldedit` peut être utilisé pour modifier les binaires pour qu'ils utilisent par défaut la taille de tas désirée. PostgreSQL peut aussi être recompilé, en passant à `configure LDFLAGS="-Wl,-bmaxdata:0x`*n*`0000000"` pour obtenir le même résultat.

Pour une compilation en 64 bits, positionnez `OBJECT_MODE` à 64 et passez `CC="gcc -maix64"` et `LDFLAGS="-Wl,-bbigtoc"` à `configure`. (Les options pour `xlc` pourraient différer.) Si vous omettez les exports de `OBJECT_MODE`, votre compilation échouera avec des erreurs de l'éditeur

de liens. Lorsque `OBJECT_MODE` est positionné, il indique aux outils de compilation d'AIX comme `ar`, `as` et `ld` quel type de fichiers à manipuler par défaut.

Par défaut, de la surallocation d'espace de pagination peut se produire. Bien que nous ne l'ayons jamais constaté, AIX tuera des processus quand il se trouvera à court de mémoire et que la zone surallouée sera accédée. Le comportement le plus proche de ceci que nous ayons constaté est l'échec d'un *fork*, parce que le système avait décidé qu'il n'y avait plus de suffisamment de mémoire disponible pour un nouveau processus. Comme beaucoup d'autres parties d'AIX, la méthode d'allocation de l'espace de pagination et l'arrêt suite à un « out-of-memory » sont configurables, soit pour le système, soit pour un processus, si cela devient un problème.

17.7.2. Cygwin

PostgreSQL peut être compilé avec Cygwin, un environnement similaire à Linux pour Windows, mais cette méthode est inférieure à la version native Windows (voir Chapitre 18) et faire tourner un serveur sur Cygwin n'est plus recommandé.

Quand vous compilez à partir des sources, suivant la procédure d'installation de style Unix (c'est-à-dire `./configure; make;`, etc.), notez les différences suivantes spécifiques à Cygwin :

- Positionnez le PATH pour utiliser le répertoire binaire Cygwin avant celui des utilitaires Windows. Cela évitera des problèmes à la compilation.
- La commande `adduser` n'est pas supportée ; utilisez les outils appropriés de gestion d'utilisateurs sous Windows NT, 2000 ou XP. Sinon, sautez cette étape.
- La commande `su` n'est pas supportée ; utilisez ssh pour simuler la commande `su` sous Windows NT, 2000 ou XP. Sinon, sautez cette étape.
- OpenSSL n'est pas supporté.
- Démarrez `cygserver` pour le support de la mémoire partagée. Pour cela, entrez la commande `/usr/sbin/cygserver &`. Ce programme doit fonctionner à chaque fois que vous démarrez le serveur PostgreSQL ou que vous initialisez un cluster de bases de données (`initdb`). La configuration par défaut de `cygserver` pourrait nécessiter des changements (par exemple, augmenter `SEMMNS`) pour éviter à PostgreSQL d'échouer en raison d'un manque de ressources système.
- Il se peut que la compilation échoue sur certains systèmes quand une locale autre que C est utilisée. Pour résoudre ce problème, positionnez la locale à C avec la commande `export LANG=C.utf8` avant de lancer la compilation, puis, une fois PostgreSQL installé, repositionnez-là à son ancienne valeur.
- Les tests de régression en parallèle (`make check`) peuvent générer des échecs de tests de régression aléatoires en raison d'un dépassement de capacité de la file d'attente de `listen()` qui cause des erreurs de connexion refusée ou des blocages. Vous pouvez limiter le nombre de connexions en utilisant la variable de make `MAX_CONNECTIONS` comme ceci :

  ```
  make MAX_CONNECTIONS=5 check
  ```

  (Sur certains systèmes, vous pouvez avoir jusqu'à 10 connexions simultanées).

Il est possible d'installer `cygserver` et le serveur PostgreSQL en tant que services Windows NT. Pour plus d'informations sur comment le faire, veuillez vous référer au document `README` inclus avec le package binaire PostgreSQL sur Cygwin. Il est installé dans le répertoire `/usr/share/doc/Cygwin`.

17.7.3. macOS

Sur les versions récentes de macOS, il est nécessaire d'embarquer le chemin « sysroot » dans les options d'inclusion utilisées pour trouver les fichiers d'en-tête système. Ceci a pour résultat la génération d'un script configure variant suivant la version du SDK utilisée durant configure. Ceci ne devrait pas poser de problèmes dans les scénarios simples, mais si vous essayez de faire quelque chose comme compiler une extension sur une machine différente de celle sur laquelle le code serveur a été compilé, vous pouvez avoir besoin de forcer l'utilisation d'un chemin sysroot différent. Pour cela, configurez `PG_SYSROOT` ainsi

```
make PG_SYSROOT=/desired/path all
```

Pour trouver le chemin approprié sur votre machine, lancez

```
xcrun --show-sdk-path
```

Notez que compiler une extension en utilisant une version sysroot différente de celle utilisée pour compiler le serveur n'est pas vraiment recommandée ; dans le pire des cas, cela peut entraîner des incohérences d'ABI difficiles à débugger.

Vous pouvez aussi sélectionner un chemin sysroot différent de celui par défaut lors du configure en indiquant `PG_SYSROOT` à configure :

```
./configure ... PG_SYSROOT=/desired/path
```

Ceci sera principalement utile pour faire de la cross-compilation pour d'autres versions de macOS. Il n'y a pas de garantie que les exécutables qui vont en résulter fonctionneront sur l'hôte actuel.

Pour supprimer les options `-isysroot`, utilisez

```
./configure ... PG_SYSROOT=none
```

(tout nom de chemin non existant fonctionnera). Ceci pourrait être utile si vous souhaitez compiler avec un compilateur autre que celui d'Apple, mais attention au fait que ce cas n'est ni testé ni supporté par les développeurs PostgreSQL.

La fonctionnalité « System Integrity Protection » (SIP) de macOS casse `make check`, car elle empêche de transmettre la configuration nécessaire de `DYLD_LIBRARY_PATH` vers les exécutables en cours de tests. Vous pouvez contourner cela en exécutant `make install` avant `make check`. Ceci étant dit, la plupart des développeurs Postgres désactivent simplement SIP.

17.7.4. MinGW/Windows Natif

PostgreSQL pour Windows peut être compilé en utilisant MinGW, un environnement de compilation similaire à celui disponible sous Unix pour les systèmes d'exploitation Microsoft, ou en utilisant la suite de compilation Visual C++ de Microsoft. La variante de compilation MinGW utilise le système de compilation normal décrit dans ce chapitre ; la compilation via Visual C++ fonctionne de façon totalement différente, et est décrite dans Chapitre 18.

Le port natif Windows requiert une version 32 ou 64 bits de Windows 2000 ou ultérieurs. Les systèmes d'exploitation antérieurs n'ont pas l'infrastructure nécessaire (mais Cygwin peut être utilisé pour ceux-ci). MinGW, le système de compilation similaire à Unix, et MSYS, une suite d'outils Unix nécessaires pour exécuter des scripts shell tels que `configure`, peuvent être téléchargés à partir de http://www.mingw.org/. Aucun de ces outils n'est nécessaire pour exécuter les binaires générés ; ils ne sont nécessaires que pour créer les binaires.

Pour compiler les binaires 64 bits avec MinGW, installez l'ensemble d'outils 64 bits à partir de https://mingw-w64.org/, ajoutez le répertoire des binaires de MinGW dans la variable d'environnement `PATH`, et lancez la commande `configure` avec l'option `--host=x86_64-w64-mingw32`.

Après que vous avez tout installé, il vous est conseillé de lancer psql dans `CMD.EXE`, car la console MSYS a des problèmes de tampons.

17.7.4.1. Récupérer des dumps suite aux plantages sous Windows

Si PostgreSQL sous Windows plante, il peut générer des minidumps qui peuvent être utilisés pour dépister la cause du plantage ; ils sont semblables aux *core dumps* d'Unix. Vous pouvez lire ces dumps avec Windows Debugger Tools ou avec Visual Studio. Pour permettre la génération des dumps sous Windows, créez un sous-répertoire nommé `crashdumps` dans le répertoire des données du cluster. Ainsi les dumps seront écrits dans ce répertoire avec un nom unique généré à partir de l'identifiant du processus planté et du moment du plantage.

17.7.5. Solaris

PostgreSQL est bien supporté sous Solaris. Plus le système d'exploitation est à jour, moins vous aurez de problèmes.

17.7.5.1. Outils requis

Vous pouvez compiler soit avec GCC, soit avec le compilateur de Sun. Pour une meilleure optimisation du code, le compilateur de Sun est fortement recommandé sur l'architecture SPARC. Si vous utilisez le compilateur de Sun, attention à ne pas sélectionner `/usr/ucb/cc` ; utilisez `/opt/SUNWspro/bin/cc`.

Vous pouvez télécharger Sun Studio sur https://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/. De nombreux outils GNU sont intégrés dans Solaris 10, ou sont présents sur le Solaris companion CD. Si vous avez besoin des packages pour des versions plus anciennes de Solaris, vous pouvez trouver ces outils sur http://www.sunfreeware.com. Si vous préférez les sources, allez sur https://www.gnu.org/order/ftp.html.

17.7.5.2. configure se plaint d'un programme de test en échec

Si `configure` se plaint d'un programme de test en échec, il s'agit probablement de l'éditeur de lien à l'exécution qui ne trouve pas une bibliothèque, probablement libz, libreadline ou une autre bibliothèque non standard telle que libssl. Pour lui indiquer le bon endroit, positionnez la variable d'environnement `LDFLAGS` sur la ligne de commande de `configure`, par exemple,

```
configure ... LDFLAGS="-R /usr/sfw/lib:/opt/sfw/lib:/usr/local/lib"
```

Voir la man page de ld pour plus d'informations.

17.7.5.3. Compiler pour des performances optimales

Sur l'architecture SPARC, Sun Studio est fortement recommandé pour la compilation. Essayez d'utiliser l'option d'optimisation `-xO5` pour générer des binaires sensiblement plus rapides. N'utilisez pas d'options modifiant le comportement des opérations à virgule flottante ni le traitement de `errno` (par exemple, `-fast`).

Si vous n'avez pas de raison d'utiliser des binaires 64 bits sur SPARC, préférez la version 32 bits. Les opérations et les binaires 64 bits sont plus lents que les variantes 32 bits. D'un autre côté, le code 32

bits sur un processeur de la famille AMD64 n'est pas natif, donc le code 32 bits est significativement plus lent sur cette famille de processeurs.

17.7.5.4. Utiliser DTrace pour tracer PostgreSQL

Oui, l'utilisation de DTrace est possible. Voir Section 28.5 pour davantage d'informations.

Si vous voyez l'édition de liens de l'exécutable `postgres` échouer avec un message d'erreur similaire à :

```
Undefined                       first referenced
 symbol                             in file
AbortTransaction                    utils/probes.o
CommitTransaction                   utils/probes.o
ld: fatal: Symbol referencing errors. No output written to postgres
collect2: ld returned 1 exit status
make: *** [postgres] Error 1
```

l'installation DTrace est trop ancienne pour gérer les sondes dans les fonctions statiques. Solaris 10u4 ou plus récent est nécessaire pour utiliser DTrace.

Chapitre 18. Installation à partir du code source sur Windows

Il est recommandé que la plupart des utilisateurs téléchargent la distribution binaire pour Windows, disponible sous la forme d'un package d'installation graphique à partir du site web du projet PostgreSQL sur https://www.postgresql.org/download/. La compilation à partir des sources n'est destinée qu'aux personnes qui développent PostgreSQL ou des extensions.

Il existe différentes façons de construire PostgreSQL sur Windows. La façon la plus simple de le faire avec les outils Microsoft est d'installer Visual Studio 2026, et d'utiliser son compilateur. Il est aussi possible de construire PostgreSQL avec Microsoft Visual C++ 2013 à 2026. Dans certains cas, il faut installer le Windows SDK en plus du compilateur.

Il est aussi possible de construire PostgreSQL avec les outils de compilation GNU fournis par MinGW, ou avec Cygwin pour les anciennes versions de Windows.

MinGW ou Cygwin utilisent le système habituel de compilation (voir Chapitre 17, et les notes spécifiques dans Section 17.7.4 et Section 17.7.2). Pour produire des binaires natifs 64 bits dans ces environnements, utilisez les outils de MinGW-w64. Ils peuvent également être utilisés pour cross-compiler vers des systèmes Windows 32 et 64 bits sur d'autres machines, comme Linux et macOS. Cygwin n'est pas recommandé pour un serveur de production, et ne devrait être utilisé qu'à destination d'anciennes versions de Windows, où la construction native ne fonctionne pas. Les exécutables officiels sont construits avec Visual Studio.

Les exécutables natifs de psql ne supportent pas l'édition de la ligne de commande. La version de psql construite avec Cygwin la supporte, donc elle devrait être utilisée là où on a besoin d'un psql interactif sous Windows.

18.1. Construire avec Visual C++ ou le Microsoft Windows SDK

PostgreSQL peut être compilés avec la suite de compilation Visual C++ de Microsoft. Les compilateurs peuvent être soit Visual Studio, soit Visual Studio Express soit certaines versions du Microsoft Windows SDK. Si vous n'avez pas déjà un environnement Visual Studio en place, le plus simple est d'utiliser les compilateurs de Visual Studio 2026 ou ceux du Windows SDK 10, tous les deux disponibles en téléchargement libre sur le site de Microsoft.

Compiler en 32 bits et 64 bits est possible avec la suite Microsoft Compiler. Compiler en 32 bits est possible à l'aide de Visual Studio 2013 jusqu'à Visual Studio 2026, ainsi qu'avec les versions autonomes Windows SDK, de la version 8.1a à la version 10. Les compilations 64 bits sont supportées avec Microsoft Windows SDK, versions 8.1a à version 10, ou Visual Studio 2013 et ses versions ultérieures. La compilation est supportée à partir de Windows 7 et Windows Server 2008 R2 SP1, en compilant avec Visual Studio 2013 jusqu'à Visual Studio 2026.

Les outils pour compiler avec Visual C++ ou Platform SDK sont dans le répertoire `src\tools\msvc`. Lors de la compilation, assurez-vous qu'il n'y a pas d'outils de MinGW ou Cygwin présents dans votre `PATH` système. Assurez-vous aussi que vous avez tous les outils Visual C++ requis disponibles dans le `PATH`. Dans Visual Studio, lancez le Visual Studio Command Prompt. Pour compiler une version 64 bits, vous devez utiliser la version 64 bits, et inversement. À partir de Visual Studio 2017, ceci peut se faire à partir de la ligne de commande en utilisant `VsDevCmd.bat` (voir `-help` pour les options disponibles et leur valeurs par défaut). `vsvars32.bat` est disponible dans Visual Studio 2015 et les versions antérieures dans le même but. À partir du Visual Studio Command Prompt, vous pouvez modifier l'architecture CPU ciblée, le type de compilation, et le système d'exploitation cible grâce à la commande `vcvarsall.bat`. Par exemple, `vcvarsall.bat`

`x64 10.0.10240.0` ciblera Windows 10 en 64 bits. Voir `-help` pour les autres options de `vcvarsall.bat`. Toutes les commandes doivent être exécutées à partir du répertoire `src\tools\msvc`.

Avant de compiler, vous pouvez créer le fichier `config.pl` pour y modifier toutes les options de configuration nécessaires, ainsi que les chemins utilisés par les bibliothèques tierces. La configuration complète est déterminée tout d'abord en lisant et en analysant le fichier `config_default.pl`, puis en appliquant les modifications provenant du fichier `config.pl`. Par exemple, pour indiquer l'emplacement de votre installation de Python, placez la ligne suivante dans `config.pl` :

```
$config->{python} = 'c:\python310';
```

Vous n'avez besoin de spécifier que les paramètres différents de ceux définis dans `config_default.pl`.

Si vous avez besoin de configurer d'autres variables d'environnement, créez un fichier `buildenv.pl`, et placez-y les commandes souhaitées. Par exemple, pour ajouter le chemin vers bison s'il ne se trouve pas dans le `PATH`, créez un fichier contenant :

```
$ENV{PATH}=$ENV{PATH} . ';c:\chemin\vers\bison\bin';
```

Pour passer d'autres arguments en ligne de commande à la commande de compilation de Visual Studio (msbuild or vcbuild) :

```
$ENV{MSBFLAGS}="/m";
```

18.1.1. Prérequis

Les outils suivants sont requis pour compiler PostgreSQL. Utilisez le fichier `config.pl` pour indiquer les chemins des bibliothèques.

Microsoft Windows SDK

> Si votre environnement de compilation ne contient pas une version supportée du Microsoft Windows SDK, il est recommandé de mettre à jour vers la dernière version supportée (actuellement la 10), téléchargeable sur le site de Microsoft[1].
>
> Vous devez toujours inclure la partie Windows Headers and Libraries du SDK. Si vous installez un Windows SDK incluant les Visual C++ Compilers, vous n'avez pas besoin de Visual Studio. Notez que la version 8.0a du Windows SDK ne contient plus d'environnement complet de compilation en ligne de commande.

Strawberry Perl

> Strawberry Perl est requis pour exécuter les scripts de compilation. La commande Perl de MinGW et de Cygwin ne fonctionnera pas. Strawberry Perl doit aussi être présent dans le `PATH`. Ses binaires sont téléchargeables à partir du site officiel[2].

Les produits suivants ne sont pas nécessaires pour démarrer, mais sont nécessaires pour construire la distribution complète. Utilisez le fichier `config.pl` pour indiquer les chemins des bibliothèques.

[1] https://www.microsoft.com/download/
[2] https://strawberryperl.com

Magicsplat TCL

Nécessaire pour construire PL/Tcl. Les binaires peuvent être téléchargés à partir du site officiel[3].

Bison et Flex

Bison et Flex sont nécessaires pour compiler à partir de Git, mais pas pour compiler à partir d'une version distribuée. Seul Bison 1.875, ou les versions 2.2 et ultérieures fonctionneront. Flex doit être en version 2.5.31 ou supérieure.

Bison et Flex sont inclus dans la suite d'outils msys, disponible à partir de son site officiel[4], comme partie de la suite de compilation MinGW.

Le répertoire contenant `flex.exe` et `bison.exe` devra être ajouté à la variable d'environnement `PATH` dans `buildenv.pl` s'il n'y est pas déjà. Dans le cas de MinGW, le répertoire est le sous-répertoire `\msys\1.0\bin` de votre répertoire d'installation de MinGW.

> **Note**
>
> La distribution Bison de GnuWin32 a apparemment un bug menant à un dysfonctionnement de Bison s'il est installé dans un répertoire dont le nom contient des espaces, tel que l'emplacement par défaut dans les installations en anglais : `C:\Program Files\GnuWin32`. Installez-le donc plutôt dans `C:\GnuWin32`, ou utilisez le chemin court NTFS de GnuWin32 dans votre variable d'environnement PATH (par exemple `C:\PROGRA~1\GnuWin32`).

Diff

Diff est nécessaire pour exécuter les tests de régression, et peut être téléchargé à partir de la page du projet gnuwin32 sur Sourceforge[5].

Gettext

Gettext est requis pour construire le support des langues (NLS), et peut être téléchargé à partir de la page du projet gnuwin32 sur Sourceforge[6]. Notez que les binaires, dépendances et fichiers d'en-tête sont tous nécessaires.

MIT Kerberos

Requis pour le support de l'authentification GSSAPI. MIT Kerberos est téléchargeable sur le site du MIT[7].

libxml2 et libxslt

Nécessaire pour le support du XML. Les binaires sont disponibles sur zlatkovic.com[8] et les sources sur le xmlsoft.org[9]. Notez que libxml2 nécessite iconv, disponible sur le même site.

LZ4

Nécessaire pour la méthode de compression LZ4. Binaires et sources peuvent être téléchargés depuis https://github.com/lz4/lz4/releases.

[3] https://www.magicsplat.com/tcl-installer/index.html
[4] http://www.mingw.org/wiki/MSYS
[5] http://gnuwin32.sourceforge.net
[6] http://gnuwin32.sourceforge.net
[7] https://web.mit.edu/Kerberos/dist/index.html
[8] https://zlatkovic.com/pub/libxml
[9] http://xmlsoft.org

Zstandard

Requis pour la méthode de compression Zstandard. Binaires et sources peuvent être téléchargés depuis https://github.com/facebook/zstd/releases.

OpenSSL

Nécessaire pour le support de SSL. Les binaires peuvent être téléchargés à partir de slproweb.com[10], ou les sources à partir de openssl.org[11].

ossp-uuid

Nécessaire pour le support d'UUID-OSSP (seulement en contrib). Les sources peuvent être récupérées sur ossp.org[12].

Python

Nécessaire pour la compilation de PL/Python. Les binaires sont téléchargeables sur python.org[13].

zlib

Nécessaire pour le support de la compression dans pg_dump et pg_restore. Les binaires sont disponibles sur le site officiel[14].

18.1.2. Considérations spéciales pour Windows 64 bits

PostgreSQL ne peut être compilé pour l'architecture x64 que sur un Windows 64 bits. Il n'y a pas de support pour les processeurs Itanium.

Le mélange de versions 32 bits et de versions 64 bits dans le même répertoire de compilation n'est pas supporté. Le système de compilation détecte automatiquement si l'environnement est 32 bits ou 64 bits, et compile PostgreSQL en accord. Pour cette raison, il est important de démarrer la bonne invite de commande avant de lancer la compilation.

Pour utiliser une bibliothèque tierce côté serveur, comme Python ou OpenSSL, cette bibliothèque *doit* aussi être en 64 bits. Il n'y a pas de support pour le chargement d'une bibliothèque 32 bits sur un serveur 64 bits. Plusieurs bibliothèques de tierce partie que PostgreSQL supporte ne sont disponibles qu'en version 32 bits, auquel cas elles ne peuvent pas être utilisées avec un serveur PostgreSQL 64 bits.

18.1.3. Construction

Pour compiler tout PostgreSQL dans la configuration publiée (par défaut), exécutez la commande :

```
build
```

Pour construire tout PostgreSQL dans la configuration de débogage, exécutez :

```
build DEBUG
```

Pour construire un seul projet, par exemple psql, exécutez l'une des deux commandes ci-dessous, suivant que vous souhaitez la configuration par défaut ou celle de débogage :

```
build psql
build DEBUG psql
```

[10] https://slproweb.com/products/Win32OpenSSL.html
[11] https://www.openssl.org
[12] http://www.ossp.org/pkg/lib/uuid/
[13] https://www.python.org
[14] https://www.zlib.net

Pour modifier la configuration de compilation par défaut, placez ce qui suit dans le fichier `buildenv.pl` :

```
$ENV{CONFIG}="Debug";
```

Il est aussi possible de compiler à partir de l'interface graphique de Visual Studio. Dans ce cas, vous devez exécuter :

```
perl mkvcbuild.pl
```

à partir de l'invite, puis ouvrir le fichier `pgsql.sln` généré (dans le répertoire racine des sources) dans Visual Studio.

18.1.4. Nettoyage et installation

La plupart du temps, la récupération automatique des dépendances dans Visual Studio prendra en charge les fichiers modifiés. Mais, s'il y a eu trop de modifications, il peut y avoir besoin de nettoyer l'installation. Pour cela, exécutez simplement la commande `clean.bat`, qui nettoiera automatiquement tous les fichiers générés. Vous pouvez aussi l'exécuter avec le paramètre *`dist`*, auquel cas elle se comporte comme **`make distclean`**, et supprime aussi les fichiers flex/bison.

Par défaut, tous les fichiers sont écrits dans un sous-répertoire de `debug` ou `release`. Pour installer ces fichiers en utilisant les emplacements standards, et pour générer aussi les fichiers nécessaire à l'installation et l'utilisation de la base de données, exécutez la commande :

```
install c:\destination\directory
```

Si vous voulez seulement installer les applications clientes et les bibliothèques, vous pouvez utiliser ces commandes :

```
install c:\destination\directory client
```

18.1.5. Exécuter les tests de régression

Pour exécuter les tests de régression, assurez-vous que vous avez terminé la construction de toutes les parties requises. Ensuite, assurez-vous que les DLL nécessaires au chargement de toutes les parties du système (comme les DLL Perl et Python pour les langages de procédure) sont présentes dans le chemin système. Dans le cas contraire, configurez-les dans le fichier `buildenv.pl`. Pour lancer les tests, exécutez une des commandes suivantes à partir du répertoire `src\tools\msvc` :

```
vcregress check
vcregress installcheck
vcregress plcheck
vcregress contribcheck
vcregress modulescheck
vcregress ecpgcheck
vcregress isolationcheck
vcregress bincheck
vcregress recoverycheck
vcregress taptest
```

Pour modifier la planification utilisée (en parallèle par défaut), ajoutez-la à la ligne de commande, comme :

```
vcregress check serial
```

`vcregress taptest` peut être utilisé pour exécuter les tests TAP d'un répertoire cible comme :

```
vcregress taptest src\bin\initdb\
```

Pour plus d'informations sur les tests de régression, voir Chapitre 33.

Les tests de régression des programmes clients, avec `vcregress bincheck`, sur les tests de restauration, avec `vcregress recoverycheck` ou sur les tests TAP avec `vcregress taptest`, nécessitent l'installation d'un module Perl supplémentaire :

IPC::Run

> Au moment où ces lignes sont écrites, `IPC::Run` n'est pas inclus dans l'installation de ActiveState Perl, ni dans la bibliothèque ActiveState Perl Package Manager (PPM). Pour l'installer, téléchargez l'archive source `IPC-Run-<version>.tar.gz` à partir du CPAN[15], et décompressez-la. Modifiez le fichier `buildenv.pl` en ajoutant une variable PERL5LIB pointant vers le sous-répertoire `lib` des fichiers extraits de l'archive. Par exemple :
>
> ```
> $ENV{PERL5LIB}=$ENV{PERL5LIB} . ';c:\IPC-Run-0.94\lib';
> ```

Les tests TAP lancés par `vcregress`) supportent les variables d'environnement `PROVE_TESTS`, étendues automatiquement selon le modèle donné, et `PROVE_FLAGS`. Elles peuvent être mises en place dans un terminal Windows, avant de lancer `vcregress` :

```
set PROVE_FLAGS=--timer --jobs 2
set PROVE_TESTS=t/020*.pl t/010*.pl
```

Il est possible de placer ces paramètres dans `buildenv.pl` :

```
$ENV{PROVE_FLAGS}='--timer --jobs 2'
$ENV{PROVE_TESTS}='t/020*.pl t/010*.pl'
```

Certains des tests TAP dépendent d'un ensemble de commandes externes qui pourraient, en option, déclencher des tests de trigger en relation. Chacune de ces variables peuvent être initialisées ou désinitialisées dans le fichier `buildenv.pl` :

`GZIP_PROGRAM`

> Chemin vers une commande gzip. La valeur par défaut est `gzip`, qui cherchera une commande de ce nom dans le `PATH` configuré.

`LZ4`

> Chemin vers une commande lz4. La valeur par défaut est `lz4`, qui cherchera une commande de ce nom dans le `PATH` configuré.

`TAR`

> Chemin vers une commande tar. La valeur par défaut est `tar`, qui cherchera une commande de ce nom dans le `PATH` configuré.

`ZSTD`

> Chemin vers une commande zstd. La valeur par défaut est `zstd`, qui cherchera une commande de ce nom dans `PATH` configuré.

[15] https://metacpan.org/release/IPC-Run

Chapitre 19. Configuration du serveur et mise en place

Ce chapitre discute de la configuration, du lancement du serveur de bases de données et de ses interactions avec le système d'exploitation.

Les directions dans ce chapitre supposent que vous travaillez avec un serveur PostgreSQL standard, sans infrastructure supplémentaire, par exemple une copie que vous aurez construit à partir des sources, d'après la méthode indiquée dans les chapitres précédents. Si vous travaillez avec une version de PostgreSQL pré-packagée ou fournie par un vendeur, il est probable que le développeur du package a fait quelques modifications pour l'installation et le démarrage du serveur suivant les conventions de votre système. Consultez la documentation du package pour les détails.

19.1. Compte utilisateur PostgreSQL

Comme avec tout démon serveur accessible au monde externe, il est conseillé de lancer PostgreSQL sous un compte utilisateur séparé. Ce compte devrait seulement être le propriétaire des données gérées par le serveur et ne devrait pas être partagé avec d'autres démons (par exemple, utiliser l'utilisateur `nobody` est une mauvaise idée). En particulier, il est conseillé que ce compte utilisateur ne soit pas le propriétaire des exécutables du serveur PostgreSQL pour s'assurer qu'un processus serveur compromis ne pourra pas modifier ces exécutables.

Les versions pré-packagées de PostgreSQL créera habituellement un compte utilisateur convenable, automatiquement, lors de l'installation du paquet.

Pour ajouter un compte utilisateur Unix, jetez un œil à la commande `useradd` ou `adduser` de votre système. Le nom de l'utilisateur postgres est souvent utilisé et l'est sur tout le livre, mais vous pouvez utiliser un autre nom si vous le souhaitez.

19.2. Créer un groupe de base de données

Avant de faire quoi que ce soit, vous devez initialiser un emplacement de stockage pour la base de données. Nous appelons ceci un *groupe de bases de données* (le standard SQL utilise le terme de groupe de catalogues). Un groupe de bases de données est une collection de bases données et est géré par une seule instance d'un serveur de bases de données en cours d'exécution. Après initialisation, un groupe de bases de données contiendra une base de données nommée `postgres`, qui a pour but d'être la base de données par défaut utilisée par les outils, les utilisateurs et les applications tiers. Le serveur de la base de données lui-même ne requiert pas la présence de la base de données `postgres` mais beaucoup d'outils supposent son existence. Il existe deux bases de données supplémentaires créées dans chaque instance lors de l'initialisation. Elles ont pour nom `template1` et `template0`. Comme leur nom le suggère, elles seront utilisées comme modèle pour les bases de données créées après ; elles ne devraient pas être utilisées pour un vrai travail (voir le Chapitre 23 pour des informations sur la création de nouvelles bases de données dans l'instance).

En terme de système de fichiers, un groupe de bases de données est un simple répertoire sous lequel les données seront stockées. Nous l'appelons le *répertoire de données* ou l'*emplacement des données*. Le choix de cet emplacement vous appartient complètement. Il n'existe pas de valeur par défaut bien que les emplacements tels que `/usr/local/pgsql/data` ou `/var/lib/pgsql/data` sont populaires. Le répertoire de données doit être initialisé avant d'être utilisé, en utilisant le programme initdb qui est installé avec PostgreSQL.

Si vous utilisez une version pré-packagée de PostgreSQL, il pourrait bien avoir une convention spécifique pour l'emplacement du répertoire de données, et il pourrait aussi fournir un script pour créer le répertoire de données. Dans ce cas, il est conseillé d'utiliser ce script plutôt que d'exécuter directement `initdb`. Consultez la documentation du paquet pour les détails.

Pour initialiser un groupe de bases de données, exécutez la commande `initdb` et indiquez l'emplacement désiré sur le système de fichiers de l'instance avec l'option `-D`, par exemple

```
$ initdb -D /usr/local/pgsql/data
```

Notez que vous devez exécuter cette commande en étant connecté sous le compte de l'utilisateur PostgreSQL décrit dans la section précédente.

> **Astuce**
>
> Comme alternative à l'option `-d`, vous pouvez initialiser la variable d'environnement `pgdata`.

Autrement, vous pouvez exécuter `initdb` via le programme pg_ctl ainsi :

```
$ pg_ctl -D /usr/local/pgsql/data initdb
```

C'est peut-être plus intuitif si vous utilisez déjà `pg_ctl` pour démarrer et arrêter le serveur (voir Section 19.3 pour les détails). Un gros intérêt est de ne connaître que cette seule commande pour gérer l'instance du serveur de bases de données.

`initdb` tentera de créer le répertoire que vous avez spécifié si celui-ci n'existe pas déjà. Bien sûr, cela peut échouer si `initdb` n'a pas les droits pour écrire dans le répertoire parent. Il est généralement recommandé que l'utilisateur PostgreSQL soit propriétaire du répertoire des données, mais aussi du répertoire parent pour que ce problème ne se présente pas. Si le répertoire parent souhaité n'existe pas non plus, vous aurez besoin de le créer, en utilisant les droits de l'utilisateur root si nécessaire. Le processus pourrait ressembler à ceci :

```
root# mkdir /usr/local/pgsql
root# chown postgres /usr/local/pgsql
```

`initdb` refusera de s'exécuter si le répertoire des données existe et contient déjà des fichiers. Cela permet de prévenir tout écrasement accidentel d'une installation existante.

Comme le répertoire des données contient toutes les données stockées dans la base, il est essentiel qu'il soit protégé contre les accès non autorisés. En conséquence, `initdb` supprime les droits d'accès à tout le monde sauf à l'utilisateur PostgreSQL, et optionnellement au groupe. L'accès au groupe, s'il est autorisé, est en lecture seule. Cela permet à un utilisateur non privilégié, du même groupe que le propriétaire de l'instance, de faire une sauvegarde des fichiers ou d'effectuer des opérations qui ne requièrent qu'un accès en lecture.

Notez qu'activer ou désactiver l'accès au groupe sur une instance préexistante exige qu'elle soit arrêtée et que les droits soient mis en place sur tous les répertoires et fichiers avant de redémarrer PostgreSQL. Sinon, un mélange des droits pourrait exister dans le répertoire de données. Pour les instances qui ne donnent accès qu'au propriétaire, les droits appropriés sont `0700` sur les répertoires et `0600` sur les fichiers. Pour les instances qui permettent aussi la lecture par le groupe, les droits appropriés sont `0750` sur les répertoires et `0640` sur les fichiers.

Néanmoins, bien que le contenu du répertoire soit sécurisé, la configuration d'authentification du client par défaut permet à tout utilisateur local de se connecter à la base de données et même à devenir le superutilisateur de la base de données. Si vous ne faites pas confiance aux utilisateurs locaux, nous vous recommandons d'utiliser une des options `-w` ou `--pwprompt` de la commande `initdb` pour affecter un mot de passe au superutilisateur de la base de données . De plus, spécifiez `-A scram-sha-256` de façon à ce que la méthode d'authentification `trust` par défaut ne soit pas utilisée ; ou modifiez le fichier `pg_hba.conf` généré après l'exécution d'`initdb` (d'autres approches raisonnables incluent l'utilisation de l'authentification `peer` ou les droits du système de fichiers pour restreindre les connexions. Voir le Chapitre 21 pour plus d'informations).

`initdb` initialise aussi la locale par défaut du groupe de bases de données. Normalement, elle prends seulement le paramétrage local dans l'environnement et l'applique à la base de données initialisée. Il est possible de spécifier une locale différente pour la base de données ; la Section 24.1 propose plus d'informations là-dessus. L'ordre de tri utilisé par défaut pour ce cluster de bases de données est initialisé par `initdb` et, bien que vous pouvez créer de nouvelles bases de données en utilisant des ordres de tris différents, l'ordre utilisé dans les bases de données modèle que initdb a créé ne peut pas être modifié sans les supprimer et les re-créer. Cela a aussi un impact sur les performances pour l'utilisation de locales autres que `C` ou `posix`. Du coup, il est important de faire ce choix correctement la première fois.

`initdb` configure aussi le codage par défaut de l'ensemble de caractères pour le groupe de bases de données. Normalement, cela doit être choisi pour correspondre au paramétrage de la locale. Pour les détails, voir la Section 24.3.

Les locales différentes de `C` et `POSIX` se basent sur la bibliothèque de collationnement du système pour le tri dépendant du jeu de caractères. Cela contrôle l'ordre des clés stockées dans les index. Pour cette raison, une instance ne peut pas basculer vers une version incompatible de la bibliothèque de collationnement, que ce soit pour une restauration d'une sauvegarde PITR mais aussi pour de la réplication binaire en flux ou pour un système d'exploitation différent, ou une mise à jour du système d'exploitation.

19.2.1. Utilisation de systèmes de fichiers secondaires

Beaucoup d'installations créent leur instance dans des systèmes de fichiers (volumes) autres que le volume racine de la machine. Si c'est votre choix, il est déconseillé d'utiliser le répertoire principal de ce volume secondaire (son point de montage) comme répertoire de données. Une meilleure pratique est de créer un répertoire au sein du point de montage, répertoire possédé par l'utilisateur PostgreSQL, puis de créer le répertoire de données à l'intérieur. Ceci évite des problèmes de droits, tout particulièrement lors d'opérations comme pg_upgrade, et garantit aussi des échecs propres si le volume secondaire n'est pas disponible.

19.2.2. Systèmes de fichiers

De manière générale, tout système de fichiers avec une sémantique POSIX est utilisable avec PostgreSQL. Les utilisateurs peuvent en choisir de différents pour diverses raisons, comme le support du fournisseur, la performance ou la familiarité. L'expérience montre que, toutes choses égales par ailleurs, on ne doit pas espérer de grosses différences de performances ou de comportement en changeant juste de système de fichiers ni en faisant de petites modifications de sa configuration.

19.2.2.1. NFS

Il est possible d'utiliser un système de fichier NFS pour le répertoire des données de PostgreSQL. PostgreSQL ne fait rien de spécial sur un un système de fichier NFS, c'est-à-dire qu'il suppose un comportement identique à celui de disques connectés locaux. PostgreSQL n'utilise aucune fonctionnalité connue pour un comportement non standard sur NFS, comme le verrouillage de fichier.

Le seul pré-requis ferme pour l'utilisation de NFS avec PostgreSQL est que celui-ci soit monté avec l'option `hard`. Avec cette option, des processus peuvent rester « pendants » indéfiniment s'il y a des problèmes réseau ;cette configuration nécessite donc une mise en place soigneuse. L'option `soft` interrompra les appels système en cas de problème réseau, mais PostgreSQL ne répétera pas ces appels interrompus ; une telle interruption mènera donc à la levée d'une erreur d'entrée-sortie.

Il n'est pas nécessaire d'utiliser l'option de montage `sync`. Le comportement de `async` est suffisant car PostgreSQL génère des appels `fsync` quand c'est approprié pour purger les caches en écriture. (C'est analogue au fonctionnement sur un système de fichiers local). Cependant, il est fortement conseillé d'utiliser l'option `sync` pour l'export depuis le *serveur* NFS sur les systèmes où elle existe (principalement Linux). Sinon, il n'y a pas de garantie qu'un `fsync`, ou son équivalent, sur le

client NFS, atteigne le stockage permanent du serveur, ce qui causerait une corruption, comme si on tournait avec fsync à off. Les valeurs par défaut des options de montage et d'export diffèrent selon les fournisseurs et les versions ; il est donc recommandé de les vérifier, voire de les préciser explicitement pour lever toute ambiguïté.

Dans certains cas, un stockage externe peut être utilisé soit par NFS, soit par un protocole de plus bas niveau comme iSCSI. Dans ce dernier cas, le stockage apparaît comme un périphérique par bloc et n'importe quel système de fichiers peut y être créé. Cette approche peut éviter au DBA de gérer les particularités du NFS, mais la complexité de la gestion du stockage distant est bien sûr repoussée à un autre niveau.

19.3. Lancer le serveur de bases de données

Avant qu'une personne ait accès à la base de données, vous devez démarrer le serveur de bases de données. Le programme serveur est appelé `postgres`.

Si vous utilisez une version pré-packagé de PostgreSQL, il inclut pratiquement à coup sûr de quoi exécuter le serveur en tâche de fond suivant les conventions de votre système d'exploitation. Utiliser l'infrastructure du paquet pour démarrer le serveur demande beaucoup moins d'efforts qu'essayer de le faire soi-même. Consultez la documentation du paquet pour les détails.

La façon brute pour démarrer le serveur manuellement est d'invoquer `postgres` directement, en précisant l'emplacement du répertoire des données avec l'option `-D`, par exemple :

```
$ postgres -D /usr/local/pgsql/data
```

qui laissera le serveur s'exécuter en avant plan. Pour cela, vous devez être connecté en utilisant le compte de l'utilisateur PostgreSQL. Sans l'option `-d`, le serveur essaiera d'utiliser le répertoire de données nommé par la variable d'environnement `pgdata`. Si cette variable ne le fournit pas non plus, le lancement échouera.

Habituellement, il est préférable de lancer `postgres` en tâche de fond. Pour cela, utilisez la syntaxe shell Unix habituelle :

```
$ postgres -D /usr/local/pgsql/data >journaux_trace 2>&1 &
```

Il est important de sauvegarder les sorties stdout et stderr du serveur quelque part, comme montré ci-dessus. Cela vous aidera dans des buts d'audits ou pour diagnostiquer des problèmes (voir la Section 25.3 pour une discussion plus détaillée de la gestion de journaux de trace).

Le programme `postgres` prend aussi un certain nombre d'autres options en ligne de commande. Pour plus d'informations, voir la page de référence postmaster ainsi que le Chapitre 20 ci-dessous.

Cette syntaxe shell peut rapidement devenir ennuyante. Donc, le programme d'emballage pg_ctl est fourni pour simplifier certaines tâches. Par exemple :

```
pg_ctl start -l journaux_trace
```

lancera le serveur en tâche de fond et placera les sorties dans le journal de trace indiqué. L'option `-d` a la même signification ici que pour `postgres`. `pg_ctl` est aussi capable d'arrêter le serveur.

Normalement, vous lancerez le serveur de bases de données lors du démarrage de l'ordinateur . Les scripts de lancement automatique sont spécifiques au système d'exploitation. Quelques scripts d'exemple sont distribués avec PostgreSQL dans le répertoire `contrib/start-scripts`. En installer un nécessitera les droits de root.

Différents systèmes ont différentes conventions pour lancer les démons au démarrage. La plupart des systèmes ont un fichier `/etc/rc.local` ou `/etc/rc.d/rc.local`. D'autres utilisent les répertoires `init.d` ou `rc.d`. Quoi que vous fassiez, le serveur doit être exécuté par le compte

utilisateur PostgreSQL *et non pas par root* ou tout autre utilisateur. Donc, vous devriez probablement former vos commandes en utilisant `su postgres -c '...'` . Par exemple :

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

Voici quelques suggestions supplémentaires par système d'exploitation (dans chaque cas, assurez-vous d'utiliser le bon répertoire d'installation et le bon nom de l'utilisateur où nous montrons des valeurs génériques).

- Pour freebsd, regardez le fichier `contrib/start-scripts/freebsd` du répertoire des sources de PostgreSQL.

- Sur openbsd, ajoutez les lignes suivantes à votre fichier `/etc/rc.local` :

  ```
  if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/
  postgres ]; then
      su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /
  var/postgresql/log -D /usr/local/pgsql/data'
      echo -n ' PostgreSQL'
  fi
  ```

- Sur les systèmes linux, soit vous ajoutez

  ```
  /usr/local/pgsql/bin/pg_ctl start -l journaux_trace -D /usr/
  local/pgsql/data
  ```

  à `/etc/rc.d/rc.local` ou `/etc/rc.local` soit vous jetez un œil à `contrib/start-scripts/linux` dans le répertoire des sources de PostgreSQL.

  Si vous utilisez systemd, vous pouvez utiliser le fichier de service (par exemple dans `/etc/systemd/system/postgresql.service`) :

  ```
  [Unit]
  Description=PostgreSQL database server
  Documentation=man:postgres(1)
  After=network-online.target
  Wants=network-online.target

  [Service]
  Type=notify
  User=postgres
  ExecStart=/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
  ExecReload=/bin/kill -HUP $MAINPID
  KillMode=mixed
  KillSignal=SIGINT
  TimeoutSec=infinity

  [Install]
  WantedBy=multi-user.target
  ```

  Utiliser `Type=notify` nécessite que le binaire du serveur soit construit avec `configure --with-systemd`.

  Faites bien attention au paramètre de délai. systemd a un délai par défaut de 90 secondes (au moment de l'écriture de cette documentation) et tuera un processus qui n'indique pas sa disponibilité après ce délai. Cependant, un serveur PostgreSQL qui aurait à réaliser une restauration suite à un crash pourrait prendre beaucoup plus de temps à démarrer. La valeur suggérée, `infinity`, désactive ce comportement.

- Sur netbsd, vous pouvez utiliser les scripts de lancement de freebsd ou de linux suivant vos préférences.

- Sur solaris, créez un fichier appelé `/etc/init.d/PostgreSQL` et contenant la ligne suivante :

  ```
  su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l
   journaux_trace -D /usr/local/pgsql/data"
  ```

  Puis, créez un lien symbolique vers lui dans `/etc/rc3.d` de nom `s99PostgreSQL`.

Tant que le serveur est lancé, son pid est stocké dans le fichier `postmaster.pid` du répertoire de données. C'est utilisé pour empêcher plusieurs instances du serveur d'être exécutées dans le même répertoire de données et peut aussi être utilisé pour arrêter le processus le serveur.

19.3.1. Échecs de lancement

Il existe de nombreuses raisons habituelles pour lesquelles le serveur échouerait au lancement. Vérifiez le journal des traces du serveur ou lancez-le manuellement (sans redirection des sorties standard et d'erreur) et regardez les messages d'erreurs qui apparaissent. Nous en expliquons certains ci-dessous parmi les messages d'erreurs les plus communs.

```
LOG:  could not bind IPv4 address "127.0.0.1": Address already in
 use
HINT:  Is another postmaster already running on port 5432? If not,
 wait a few seconds and retry.
FATAL:  could not create any TCP/IP sockets
```

Ceci signifie seulement ce que cela suggère : vous avez essayé de lancer un autre serveur sur le même port où un autre est en cours d'exécution. Néanmoins, si le message d'erreur du noyau n'est pas `address already in use` ou une quelconque variante, il pourrait y avoir un autre problème. Par exemple, essayer de lancer un serveur sur un numéro de port réservé pourrait avoir ce résultat :

```
$ postgres -p 666
LOG:  could not bind IPv4 address "127.0.0.1": Permission denied
HINT:  Is another postmaster already running on port 666? If not,
 wait a few seconds and retry.
FATAL:  could not create any TCP/IP sockets
```

Un message du type

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001,
 size=4011376640, 03600).
```

signifie probablement que les limites de votre noyau sur la taille de la mémoire partagée est plus petite que l'aire de fonctionnement que PostgreSQL essaie de créer (4011376640 octets dans cet exemple). Ceci n'est susceptible de se produire uniquement si `shared_memory_type` a été affecté à `sysv`. Dans ce cas, vous pouvez essayer de lancer le serveur avec un nombre de tampons plus petit que la normale (shared_buffers) ou vous pouvez reconfigurer votre noyau pour accroître la taille de mémoire partagée autorisée. Vous pourriez voir aussi ce message en essayant d'exécuter plusieurs serveurs sur la même machine si le total de l'espace qu'ils requièrent dépasse la limite du noyau.

Une erreur du type

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

ne signifie *pas* qu'il vous manque de l'espace disque. Elle signifie que la limite de votre noyau sur le nombre de sémaphores system v est inférieure au nombre que PostgreSQL veut créer. Comme ci-

dessus, vous pouvez contourner le problème en lançant le serveur avec un nombre réduit de connexions autorisées (max_connections) mais vous voudrez éventuellement augmenter la limite du noyau.

Des détails sur la configuration des capacités IPC System V sont donnés dans la Section 19.4.1.

19.3.2. Problèmes de connexion du client

Bien que les conditions d'erreurs possibles du côté client sont assez variées et dépendantes de l'application, certaines pourraient être en relation directe avec la façon dont le serveur a été lancé. Les conditions autres que celles montrées ici devraient être documentées avec l'application client respective.

```
psql: error: connection to server at
 "server.joe.com" (123.123.123.123), port 5432 failed: Connection
 refused
        Is the server running on that host and accepting TCP/IP
 connections?
```

Ceci est l'échec générique « je n'ai pas trouvé de serveur à qui parler ». Cela ressemble au message ci-dessus lorsqu'une connexion TCP/IP est tentée. Une erreur commune est d'oublier de configurer le serveur pour qu'il autorise les connexions TCP/IP.

Autrement, vous obtiendrez ceci en essayant une communication de type socket de domaine Unix vers un serveur local :

```
psql: error: connection to server on socket "/tmp/.s.PGSQL.5432"
 failed: No such file or directory
        Is the server running locally and accepting connections on
 that socket?
```

Si le serveur est vraiment en cours d'exécution, vérifiez que le chemin du socket pour le client (ici `/tmp`) est en accord avec le paramètre unix_socket_directories du serveur.

Un message d'échec de connexion affiche toujours l'adresse du serveur ou le chemin du socket, ce qui est utile pour vérifier que le client est en train de se connecter au bon endroit. Si aucun serveur n'est en écoute ici, le message d'erreur du noyau sera typiquement soit `connection refused` soit `no such file or directory`, comme ce qui est illustré (il est important de réaliser que `connection refused`, dans ce contexte, ne signifie *pas* que le serveur a obtenu une demande de connexion et l'a refusé. Ce cas produira un message différent comme indiqué dans la Section 21.15). D'autres messages d'erreurs tels que `connection timed out` pourraient indiquer des problèmes plus fondamentaux comme un manque de connexion réseau, ou un pare-feu bloquant la connexion.

19.4. Gérer les ressources du noyau

PostgreSQL peut quelque fois dépasser les limites des ressources du système d'exploitation, tout spécialement quand plusieurs copies du serveur s'exécutent sur le même système, ou sur des très grosses installations. Cette section explique les ressources du noyau utilisées par PostgreSQL et les étapes à suivre pour résoudre les problèmes liés à la consommation des ressources du noyau.

19.4.1. Mémoire partagée et sémaphore

PostgreSQL a besoin que le système d'exploitation fournisse des fonctionnalités de communication inter-processus (IPC), en particulier de la mémoire partagée et des sémaphores. Les systèmes dérivés d'Unix fournissent « System V » IPC, « POSIX » IPC ou les deux. Windows qui fournit sa propre implémentation de ces fonctionnalités ne sera pas approfondi ici.

Par défaut, PostgreSQL alloue une très petite quantité de mémoire partagée System V, ainsi qu'une quantité bien plus importante de mémoire partagée anonyme `mmap`. Autre possibilité : utiliser un seul

gros bloc de mémoire partagée System V (voir shared_memory_type). De plus, un nombre important de sémaphores de style System V ou POSIX sont créés au démarrage du serveur. Actuellement, les sémaphores POSIX sont utilisés sur les systèmes Linux et FreeBSD, alors que les autres plateformes utilisent les sémaphores System V.

Les fonctionnalités System V IPC sont habituellement restreintes par les limites d'allocation au niveau système. Quand PostgreSQL dépasse une des nombreuses limites IPC, le serveur refusera de s'exécuter et lèvera un message d'erreur instructif décrivant le problème rencontré et que faire avec (voir aussi la Section 19.3.1). Les paramètres adéquats du noyau sont nommés de façon cohérente parmi les différents systèmes ; le Tableau 19.1 donne un aperçu. Néanmoins, les méthodes pour les obtenir varient. Les suggestions pour quelques plateformes sont données ci-dessous.

Tableau 19.1. Paramètres System V IPC

Nom	Description	Valeurs nécessaires pour faire fonctionner une instance PostgreSQL
SHMMAX	Taille maximum du segment de mémoire partagée (en octets)	au moins 1 ko, mais la valeur par défaut est normalement bien plus grande
SHMMIN	Taille minimum du segment de mémoire partagée (en octets)	1
SHMALL	Total de la mémoire partagée disponible (en octets ou pages)	si octets, identique à SHMMAX ; si pages, ceil(SHMMAX/PAGE_SIZE), plus de la marge pour les autres applications
SHMSEG	Nombre maximum de segments de mémoire partagée par processus	seul un segment est nécessaire, mais la valeur par défaut est bien plus importante
SHMMNI	Nombre maximum de segments de mémoire partagée pour tout le système	comme SHMSEG plus de la place pour les autres applications
SEMMNI	Nombre maximum d'identifiants de sémaphores (c'est-à-dire d'ensembles)	au moins ceil((max_connections + autovacuum_max_workers + max_wal_senders + max_worker_processes + 6) / 16) plus de la marge pour les autres applications
SEMMNS	nombre maximum de sémaphores répartis dans le système	ceil((max_connections + autovacuum_max_workers + max_wal_senders + max_worker_processes + 6) / 16) * 17 plus la place pour les autres applications
SEMMSL	Nombre maximum de sémaphores par ensemble	au moins 17
SEMMAP	Nombre d'entrées dans la carte des sémaphores	voir le texte
SEMVMX	Valeur maximum d'un sémaphore	au moins 1000 (vaut souvent par défaut 32767 ; ne pas changer sauf si vous êtes forcé.)

PostgreSQL requiert quelques octets de mémoire partagée System V (typiquement 48 octets sur des plateformes 64 bits) pour chaque copie du serveur. Sur la plupart des systèmes d'exploitation modernes, cette quantité est facilement allouable. Néanmoins, si vous exécutez plusieurs copies du serveur ou si vous le configurez explicitement pour utiliser de grandes quantités de mémoire partagée System V (voir shared_memory_type et dynamic_shared_memory_type), il pourrait être nécessaire d'augmenter SHMALL, correspondant à la quantité totale de mémoire partagée System V pour tout le système. Notez que SHMALL est en nombre de blocs disques, et non pas en nombre d'octets sur de nombreux systèmes.

La taille minimum des segments de mémoire partagée (SHMMIN) est moins sensible aux problèmes. Elle devrait être au plus à environ 32 octets pour PostgreSQL (il est habituellement à 1). Le nombre maximum de segments au travers du système (SHMMNI) ou par processus (SHMSEG) a peu de chances de causer un problème sauf s'ils sont configurés à zéro sur votre système.

Lors de l'utilisation de sémaphores System V, PostgreSQL utilise un sémaphore par connexion autorisée (max_connections), par processus autovacuum worker autorisé (autovacuum_max_workers), par processus WAL sender autorisé (max_wal_senders) et par processus en tâche de fond autorisé réclamant un accès à la mémoire partagée, le tout par ensemble de 16. Chacun de ces ensembles contiendra aussi un 17è sémaphore qui contient un « nombre magique » pour détecter la collision avec des ensembles de sémaphore utilisés par les autres applications. Le nombre maximum de sémaphores dans le système est initialisé par SEMMNS, qui en conséquence doit être au moins aussi haut que `max_connections` plus `autovacuum_max_workers` plus `max_wal_senders` plus `max_worker_processes`, plus 1 encore pour chacune des 16 connexions autorisées, plus les workers (voir la formule dans le Tableau 19.1). Le paramètre SEMMNI détermine la limite du nombre d'ensembles de sémaphores qui peuvent exister sur le système à un instant précis. Ce paramètre doit donc être au moins égal à `ceil((max_connections + autovacuum_max_workers + max_wal_senders + max_worker_processes + 6) / 16)`. Baisser le nombre de connexions autorisées est un contournement temporaire pour les échecs qui sont habituellement indiqués par le message « no space left on device », à partir de la fonction `semget`.

Dans certains cas, il pourrait être nécessaire d'augmenter SEMMAP pour être au moins dans l'ordre de SEMMNS. Si le système dispose de ce paramètre (ce n'est pas le cas pour beaucoup d'entre eux), il définit la taille de la carte de ressources de sémaphores, dans laquelle chaque bloc contigu de sémaphores disponibles a besoin d'une entrée. Lorsqu'un ensemble de sémaphores est libéré, soit il est ajouté à une entrée existante, soit il est enregistré sous une nouvelle entrée dans la carte. Si la carte est pleine, les sémaphores libérés sont perdus (jusqu'au redémarrage). La fragmentation de l'espace des sémaphores peut mener au fil du temps à rendre moins de sémaphores disponibles qu'il ne devrait y avoir.

D'autres paramètres en relation avec l'« annulation de sémaphores », tels que SEMMNU et SEMUME, n'affectent pas PostgreSQL.

Lors de l'utilisation de sémaphores POSIX, le nombre de sémaphores nécessaires est le même que pour System V, c'est-à-dire un sémaphore par connexion autorisée (max_connections), par processus autovacuum worker autorisé (autovacuum_max_workers), par processus WAL sender autorisé (max_wal_senders) et par processus en tâche de fond (max_worker_processes). Sur les plateformes où cette option est préférée, le noyau ne spécifie pas de limite au nombre de sémaphores POSIX.

AIX

> Il ne doit plus être nécessaire de faire une configuration spéciale pour les paramètres tels que SHMMAX, car c'est configuré de façon à ce que toute la mémoire puisse être utilisée en tant que mémoire partagée. C'est le type de configuration habituellement utilisée pour d'autres bases de données comme DB/2.
>
> Néanmoins, il pourrait être nécessaire de modifier l'information globale `ulimit` dans `/etc/security/limits` car les limites en dur par défaut pour les tailles de fichiers (`fsize`) et les nombres de fichiers (`nofiles`) pourraient être trop bas.

FreeBSD

> Les paramètres par défaut de mémoire partagée sont en général suffisant, sauf si vous affectez `shared_memory_type` à `sysv`. Les sémaphores System V ne sont pas utilisés sur cette plateforme.
>
> Les paramètres IPC par défaut peuvent être modifiés en utilisant la commande `sysctl` ou `loader`. Les paramètres suivants peuvent être configurés en utilisant `sysctl` :

```
# sysctl kern.ipc.shmall=32768
# sysctl kern.ipc.shmmax=134217728
```

Pour que ces paramètres persistent après les redémarrages, modifiez `/etc/sysctl.conf`.

Si vous avez affecté `shared_memory_type` à `sysv`, vous pourriez aussi vouloir configurer votre noyau pour verrouiller la mémoire partagée System V en RAM et l'empêcher d'être envoyé dans le swap. Ceci peut être accompli en utilisant le paramètre `kern.ipc.shm_use_phys` de `sysctl`.

Dans une prison FreeBSD, il vous est recommandé de configurer le paramètre `sysvshm` à `new`, ainsi il a son propre espace de nommage de mémoire partagée System V. (Avant FreeBSD 11.0, il était nécessaire d'activer l'accès partagé à l'espace de nommage IPC de l'hôte depuis les prisons, et prendre les mesures pour éviter les collisions.)

NetBSD

La configuration par défaut de la mémoire partagée est en général suffisante, sauf si vous avez affecté `shared_memory_type` à `sysv`. Dans ce cas, vous voudrez habituellement augmenter `kern.ipc.semmni` et `kern.ipc.semmns` car les valeurs par défaut de ces paramètres sur NetBSD sont bien trop bas.

Les paramètres IPC peuvent être ajustés en utilisant `sysctl`. Par exemple :

```
# sysctl -w kern.ipc.semmni=100
```

Pour que ce paramètrage persiste après un redémarrage, modifiez le fichier `/etc/sysctl.conf`.

Si vous avez configuré `shared_memory_type` à `sysv`, vous pourriez aussi vouloir configurer votre noyau pour verrouiller la mémoire partagée System V en RAM et prévenir la mémoire d'être envoyée sur swap. Ceci peut être accompli en utilisant le paramètre `kern.ipc.shm_use_phys`de la commande `sysctl`.

OpenBSD

La configuration par défaut de la mémoire partagée est en général suffisante, sauf si vous avez configuré `shared_memory_type` à `sysv`. Vous voudrez habituellement augmenter `kern.seminfo.semmni` et `kern.seminfo.semmns` car les valeurs par défaut de ces paramètres sur OpenBSD sont bien trop bas.

Les paramètres IPC peuvent être ajustés en utilisant `sysctl`, par exemple :

```
# sysctl kern.seminfo.semmni=100
```

Pour rendre ce paramétrage persistant entre les redémarrages, modifiez `/etc/sysctl.conf`.

HP-UX

Les paramètres par défaut tendent à suffire pour des installations normales.

Les paramètres IPC peuvent être initialisés dans System Administration Manager (SAM) sous Kernel Configuration → Configurable Parameters. Choisissez Create A New Kernel une fois terminé.

Linux

La configuration par défaut de la mémoire partagée est en général suffisante, sauf si vous avez configuré `shared_memory_type` à `sysv`, et uniquement sur les anciennes versions de noyau fournies avec des valeurs par défaut trop basses. Les sémaphores de System V ne sont pas utilisés sur cette plateforme.

La configuration de la taille de mémoire partagée peut être modifiée avec l'interface proposée par la commande `sysctl`. Par exemple, pour permettre l'utilisation de 16 Go :

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

Pour rendre ce paramètrage persistant entre les redémarrages, voir `/etc/sysctl.conf`.

macOS

La configuration par défaut de la mémoire partagée et des sémaphores est en général suffisante, sauf si vous avez configuré `shared_memory_type` à `sysv`.

La méthode recommandée pour configurer la mémoire partagée sous macOS est de créer un fichier nommé `/etc/sysctl.conf` contenant des affectations de variables comme :

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

Notez que, dans certaines versions de macOS, *les cinq* paramètres de mémoire partagée doivent être configurés dans `/etc/sysctl.conf`, sinon les valeurs seront ignorées.

`SHMMAX` peut seulement être configuré avec un multiple exact de 4096.

`SHMALL` est mesuré en page de 4 ko sur cette plateforme.

Il est possible de tous les modifier en ligne sauf `SHMMNI`, grâce à sysctl. Mais il est toujours préférable de configurer vos valeurs préférées dans `/etc/sysctl.conf`, pour que les nouvelles valeurs soient conservées après un redémarrage.

Solaris
illumos

La configuration de la mémoire partagée et des sémaphores par défaut sont habituellement suffisamment bonnes pour la majorité des configurations de PostgreSQL. La valeur par défaut de Solaris pour `SHMMAX` correspond maintenant à un quart de la mémoire disponible sur le système. Pour configurer plus finement ce paramètre, vous devez utiliser une configuration de projet associé à l'utilisateur `postgres`. Par exemple, exécutez ce qui suit en tant qu'utilisateur `root` :

```
projadd -c "PostgreSQL DB User" -K "project.max-shm-
memory=(privileged,8GB,deny)" -U postgres -G postgres
 user.postgres
```

Cette commande ajoute le projet `user.postgres` et configure le maximum de mémoire partagée pour l'utilisateur `postgres` à 8 Go. Cela prend effet à chaque fois que l'utilisateur se connecte et quand vous redémarrez PostgreSQL. La ligne ci-dessus suppose que PostgreSQL est

exécuté par l'utilisateur `postgres` dans le groupe `postgres`. Aucun redémarrage du serveur n'est requis.

Sur un serveur de bases de données ayant beaucoup de connexions, les autres modifications recommandées pour le noyau sont :

```
project.max-shm-ids=(priv,32768,deny)
project.max-sem-ids=(priv,4096,deny)
project.max-msg-ids=(priv,4096,deny)
```

De plus, si vous exécutez PostgreSQL dans une zone, vous pourriez avoir besoin d'augmenter les limites d'utilisation des ressources pour la zone. Voir *Chapter2: Projects and Tasks* dans *System Administrator's Guide* pour plus d'informations sur les `projets` et `prctl`.

19.4.2. systemd RemoveIPC

Si systemd est utilisé, certaines précautions sont de mise pour que les ressources IPC (incluant la mémoire partagée) ne soient pas supprimées par le système d'exploitation. Cela est particulièrement important lors de l'installation de PostgreSQL via les sources. Les utilisateurs de versions packagées par la distribution ont moins de chance d'être affectés, l'utilisateur `postgres` étant habituellement créé en tant qu'utilisateur système.

Le paramètre `RemoveIPC` dans `logind.conf` contrôle si les objets IPC sont supprimés lors de déconnexion complète d'un utilisateur. Les utilisateurs système sont exclus. Ce paramètre est par défaut actif sur la version originale de systemd, mais certaines distributions positionnent ce paramètre à off.

Un effet typique observé quand ce paramètre est activé est que les objets en mémoire partagée utilisés pour l'exécution des requêtes en parallèle sont supprimés à des moments en apparence aléatoires, amenant ainsi des erreurs et messages d'avertissement lors des tentatives de les ouvrir ou supprimer, comme :

```
WARNING:  could not remove shared memory segment "/
PostgreSQL.1450751626": No such file or directory
```

Différents types d'objets IPC (mémoire partagée et sémaphores, System V et POSIX) sont traités de manière légèrement différente par systemd, et l'on peut observer que certaines ressources IPC ne sont pas supprimées de la même manière que les autres. Il n'est toutefois pas conseillé de compter sur ces subtiles différences.

Une « déconnexion utilisateur » peut survenir lors d'une opération de maintenance ou manuellement lorsqu'un administrateur se connecte avec le compte `postgres` ou un compte similaire, ce qui est difficile à éviter en général.

Ce qu'est un « utilisateur système » est déterminé à la compilation de systemd par le paramètre `SYS_UID_MAX` dans `/etc/login.defs`.

Les scripts de packaging et déploiement devront faire attention à créer l'utilisateur `postgres` en tant qu'utilisateur système avec `useradd -r`, `adduser --system`, ou une commande équivalente.

Sinon, si le compte utilisateur a été créé de manière incorrecte ou ne peut être modifié, il est recommandé de configurer

```
RemoveIPC=no
```

dans `/etc/systemd/logind.conf` ou un autre fichier de configuration approprié.

Attention

Au moins une de ces deux choses doit être garantie, sinon le serveur PostgreSQL ne pourra être considéré comme fiable.

19.4.3. Limites de ressources

Les systèmes d'exploitation style Unix renforcent différents types de limites de ressources qui pourraient interférer avec les opérations de votre serveur PostgreSQL. Les limites sur le nombre de processus par utilisateur, le nombre de fichiers ouverts par un processus et la taille mémoire disponible pour chaque processus sont d'une grande importance. Chacun d'entre elles ont une limite « dure » et une limite « souple ». La limite souple est réellement ce qui compte mais cela pourrait être changé par l'utilisateur jusqu'à la limite dure. La limite dure pourrait seulement être modifiée par l'utilisateur root. L'appel système `setrlimit` est responsable de l'initialisation de ces paramètres. La commande interne du shell `ulimit` (shells Bourne) ou `limit` (csh) est utilisé pour contrôler les limites de ressource à partir de la ligne de commande. Sur les systèmes dérivés BSD, le fichier `/etc/login.conf` contrôle les différentes limites de ressource initialisées à la connexion. Voir la documentation du système d'exploitation pour les détails. Les paramètres en question sont `maxproc`, `openfiles` et `datasize`. par exemple :

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

(`-cur` est la limite douce. Ajoutez `-max` pour configurer la limite dure.)

Les noyaux peuvent aussi avoir des limites sur le système complet pour certaines ressources.

- Sur linux, le paramètre noyau `fs.file-max` détermine le nombre maximum de fichiers ouverts que le noyau supportera. Ce nombre est modifiable avec `sysctl -w fs.file-max=`*`N`*. Pour rendre la configuration persistante après un redémarrage, ajouter cette affectation dans le fichier `/etc/sysctl.conf`. La limite des fichiers par processus est fixée lors de la compilation du noyau ; voir `/usr/src/linux/documentation/proc.txt` pour plus d'informations.

Le serveur PostgreSQL utilise un processus par connexion de façon à ce que vous puissiez fournir au moins autant de processus que de connexions autorisées, en plus de ce dont vous avez besoin pour le reste de votre système. Ceci n'est habituellement pas un problème mais si vous exécutez plusieurs serveurs sur une seule machine, cela pourrait devenir étroit.

La limite par défaut des fichiers ouverts est souvent initialisée pour être « amicalement sociale », pour permettre à de nombreux utilisateurs de coexister sur une machine sans utiliser une fraction inappropriée des ressources du système. Si vous lancez un grand nombre de serveurs sur une machine, cela pourrait être quelque chose que vous souhaitez mais sur les serveurs dédiés, vous pourriez vouloir augmenter cette limite.

D'un autre côté, certains systèmes autorisent l'ouverture d'un grand nombre de fichiers à des processus individuels ; si un plus grand nombre le font, alors les limites du système peuvent facilement être dépassées. Si vous rencontrez ce cas et que vous ne voulez pas modifier la limite du système, vous pouvez initialiser le paramètre de configuration max_files_per_process de PostgreSQL pour limiter la consommation de fichiers ouverts.

Une autre limite du noyau qui pourrait avoir un impact pour accepter un très grand nombre de connexions est la longueur maximale de la queue de connexion du socket. Si un plus grand nombre de

demandes de connexion arrive dans une petite période de temps, certaines pourraient être rejetées avant que le processus postmaster ne puisse traiter les demandes, avec des clients recevant des messages d'erreurs de connexion qui aident peu à résoudre le problème (par exemple « Resource temporarily unavailable » ou « Connection refused », en français respectivement « Ressource temporairement indisponible » ou « Connexion refusée ». La longueur limite de la queue par défaut est de 128 sur un grand nombre de plateformes. Pour l'augmenter, ajustez le paramètre noyau approprié avec sysctl, puis redémarrez postmaster. Le paramètre a différents noms : `net.core.somaxconn` sous Linux, `kern.ipc.soacceptqueue` sous les FreeBSD récents, et `kern.ipc.somaxconn` sur macOS et d'autres variantes BSD.

19.4.4. Linux memory overcommit

Le comportement par défaut de la mémoire virtuelle sur Linux n'est pas optimal pour PostgreSQL. Du fait de l'implémentation du « memory overcommit » par le noyau, celui-ci peut arrêter le serveur PostgreSQL (le processus serveur principal, « postmaster ») si les demandes de mémoire de PostgreSQL ou d'un autre processus provoque un manque de mémoire virtuelle au niveau du système.

Si cela se produit, un message du noyau qui ressemble à ceci (consulter la documentation et la configuration du système pour savoir où chercher un tel message) :

```
Out of Memory: Killed process 12345 (postgres)
```

peut survenir. Ceci indique que le processus `postgres` a été terminé à cause d'un problème de mémoire. Bien que les connexions en cours continuent de fonctionner normalement, aucune nouvelle connexion n'est acceptée. Pour revenir à un état normal, PostgreSQL doit être relancé.

Une façon d'éviter ce problème revient à lancer PostgreSQL sur une machine où vous pouvez vous assurer que les autres processus ne mettront pas la machine en manque de mémoire. S'il y a peu de mémoire, augmenter la swap peut aider à éviter le problème car un système peut tuer des processus lorsque la mémoire physique et la mémoire swap sont utilisées entièrement.

Si PostgreSQL lui-même est la cause d'un manque de mémoire du système, vous pouvez éviter le problème en modifiant votre configuration. Dans certains cas, baisser les paramètres de configuration de la mémoire peut aider, tout particulièrement `shared_buffers`, `work_mem` et `hash_mem_multiplier`. Dans d'autres cas, le problème peut être causé par l'autorisation d'un trop grand nombre de connexions au serveur de bases de données. Dans beaucoup de cas, il est préférable de réduire `max_connections` et d'utiliser à la place un logiciel de multiplexage de connexions (*connection pooling*).

Il est possible de modifier le comportement du noyau avec le « overcommit memory ». Bien que ce paramétrage n'empêchera pas ce comportement[1], il réduira sa fréquence de façon significative et contribuera du coup à un système plus robuste. Ceci se fait en sélectionnant le mode strict de l'overcommit via `sysctl` :

```
sysctl -w vm.overcommit_memory=2
```

ou en plaçant une entrée équivalente dans `/etc/sysctl.conf`. Vous pourriez souhaiter modifier le paramétrage relatif `vm.overcommit_ratio`. Pour les détails, voir la documentation du noyau (https://www.kernel.org/doc/Documentation/vm/overcommit-accounting).

Une autre approche, qui peut aussi utiliser la modification de `vm.overcommit_memory`, est de configurer la valeur de la variable d'*ajustement du score OOM*, valeur par processus, pour le processus postmaster à `-1000`, garantissant ainsi qu'il ne sera pas la cible de OOM. La façon la plus simple de le faire est d'exécuter

```
echo -1000 > /proc/self/oom_score_adj
```

[1] https://lwn.net/Articles/104179/

dans le script de démarrage de postmaster juste avant d'appeler postmaster. Notez que cette action doit être faite en tant qu'utilisateur root. Dans le cas contraire, elle n'aura aucun effet. Du coup, un script de démarrage, exécuté par root, est le meilleur endroit où placer ce code. Si vous le faites, vous devriez aussi configurer ces variables d'environnement dans le script de démarrage avant d'invoquer le processus postmaster :

```
export PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
export PG_OOM_ADJUST_VALUE=0
```

Ces paramètres vont faire en sorte que les processus fils du postmaster s'exécuteront avec l'ajustement de score OOm normal (0), pour que l'OOM puisse encore les cibler si cela s'avère nécessaire. Vous pouvez utiliser d'autres valeurs pour `PG_OOM_ADJUST_VALUE` si vous voulez que les processus fils s'exécutent avec un autre ajustement de score. (`PG_OOM_ADJUST_VALUE` peut aussi être omis, auquel cas sa valeur par défaut est zéro.) Si vous ne voulez pas configurer `PG_OOM_ADJUST_FILE`, les processus fils s'exécuteront avec le même ajustement de score OOM que le processus père postmaster, ce qui n'est pas conseillé car le but est de s'assurer que le processus postmaster soit protégé par la configuration.

19.4.5. Pages mémoire de grande taille (*huge pages*) sous Linux

L'utilisation des *huge pages* réduit la surcharge lors de l'utilisation de gros morceaux contigus de mémoire, ce que fait PostgreSQL, tout particulièrement lors de l'utilisation de grosses valeurs pour shared_buffers. Pour activer cette fonctionnalité avec PostgreSQL, vous avez besoin d'un noyau compilé avec `CONFIG_HUGETLBFS=y` et `CONFIG_HUGETLB_PAGE=y`. Vous aurez aussi besoin de configurer le système d'exploitation pour qu'il fournisse suffisament de Huge Pages de la taille désirée. Le paramètre shared_memory_size_in_huge_pages, calculé à l'exécution renvoie le nombre de Huge Pages requis. Ce paramètre peut être visualisé avant de démarrer le serveur avec une commande `postgres` comme :

```
$ postgres -D $PGDATA -C shared_memory_size_in_huge_pages
$ grep ^Hugepagesize /proc/meminfo
Hugepagesize:       2048 kB
$ ls /sys/kernel/mm/hugepages
hugepages-1048576kB  hugepages-2048kB
```

Dans cet exemple, la valeur par défaut est de 2 Mo, vous pouvez aussi demander explicitement soit 2 Mo soit 1 Go avec huge_page_size. Pour s'adapter au nombre de blocs calculés par `shared_memory_size_in_huge_pages`. Alors que nous avons besoin d'au moins `3170` Huge Pages dans cet exemple, une configuration plus importante serait appropriée si les autres programmes sur la machine ont besoin de Huge Pages eux-aussi. Nous pouvons les configurer avec :

```
# sysctl -w vm.nr_hugepages=3170
```

N'oubliez pas d'ajouter cette configuration à `/etc/sysctl.conf` pour qu'elle soit appliquée à chaque redémarrage. Pour les tailles personnalisées de Huge Page, nous pouvons utiliser à la place :

```
# echo 3170 > /sys/kernel/mm/hugepages/hugepages-2048kB/
nr_hugepages
```

Il est aussi possible de fournir ces paramètres au démarrage en utilisant les paramètres noyau tels que `hugepagesz=2M hugepages=3170`.

Parfois, le noyau n'est pas capable d'allouer immédiatement le nombre souhaité de *huge pages* à cause de la fragmentation, donc il peut être nécessaire de répéter cette commande ou de redémarrer. (Tout

de suite après un redémarrage, la plupart de la mémoire de la machine doit être disponible à une conversion en *huge pages*.) Pour vérifier la situation au niveau de l'allocation des *huge pages* pour une taille donnée, utilisez :

```
$ cat /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

Il pourrait être nécessaire de donner le droit à l'utilisateur du système d'exploitation du serveur de bases de données en configurant `vm.hugetlb_shm_group` via sysctl, et/ou en donnant le droit de verrouiller la mémoire avec `ulimit -l`.

Il est aussi nécessaire de donner le droit d'utiliser les *huge pages* à l'utilisateur système qui exécute PostgreSQL. Cela se fait en configurant `vm.hugetlb_shm_group` via sysctl, et le droit de verrouiller la mémoire avec `ulimit -l`.

Le comportement par défaut pour les *huge pages* dans PostgreSQL est de les utiliser quand cela est possible, avec la taille par défaut des Huge Pages du système, et de revenir aux pages normales dans le cas contraire. Pour forcer l'utilisation des *huge pages*, vous pouvez configurer huge_pages à `on` dans le fichier `postgresql.conf`. Notez que, avec ce paramètre configuré ainsi, PostgreSQL refusera de démarrer s'il ne peut pas récupérer suffisamment de *huge pages*.

Pour une description détaillée des *huge pages* sous Linux, lisez https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

19.5. Arrêter le serveur

Il existe plusieurs façons d'arrêter le serveur de bases de données. Au final, elles se basent toutes sur l'envoi d'un signal au processus serveur principal.

Si vous utilisez une version pré-packagée de PostgreSQL et que vous avez utilisé les outils du paquet pour démarrer le serveur, alors vous devriez aussi utiliser les outils du paquet pour arrêter le serveur. Consultez la documentation du paquet pour les détails.

Lors d'une gestion directe du serveur, vous pouvez contrôler le type d'arrêt en envoyant différents signaux au processus `postgres` :

SIGTERM

C'est le mode d'*arrêt intelligent*. Après réception de SIGTERM, le serveur désactive les nouvelles connexions mais permet aux sessions en cours de terminer leur travail normalement. Il s'arrête seulement après que toutes les sessions se sont terminées normalement. C'est l'arrêt intelligent (*smart shutdown*). Si le serveur est en restauration quand une demande d'arrêt intelligent est envoyée, la restauration et la réplication en flux seront stoppées seulement une fois que toutes les autres sessions ont terminé.

sigint

C'est le mode d'*arrêt rapide*. Le serveur désactive les nouvelles connexions et envoie à tous les processus serveur le signal sigterm, qui les fera annuler leurs transactions courantes pour quitter rapidement. Il attend ensuite la fin de tous les processus serveur et s'arrête finalement.

sigquit

C'est le mode d'*arrêt immédiat*. Le serveur enverra SIGQUIT à tous les processus fils et attendra qu'ils se terminent. Ceux qui ne se terminent pas au bout de cinq secondes se verront envoyés un signal SIGKILL par le processus père `postgres`, qui les arrêtera sans attendre plus. Ceci peut amener à un redémarrage en mode restauration (de ce fait, ceci n'est recommandé que dans les cas d'urgence).

Le programme pg_ctl fournit une interface agréable pour envoyer ces signaux dans le but d'arrêter le serveur. Autrement, vous pouvez envoyer le signal directement en utilisant `kill` sur les systèmes autres que Windows. Le PID du processus `postgres` peut être trouvé en utilisant le programme `ps` ou à partir du fichier `postmaster.pid` dans le répertoire des données. Par exemple, pour exécuter un arrêt rapide :

```
$ kill -int `head -1 /usr/local/pgsql/data/postmaster.pid`
```

> **Important**
>
> Il vaut mieux de ne pas utiliser SIGKILL pour arrêter le serveur. Le faire empêchera le serveur de libérer la mémoire partagée et les sémaphores. De plus, SIGKILL tue le processus `postgres` sans lui laisser le temps de relayer ce signal à ses sous-processus, donc il peut aussi être nécessaire de tuer chaque sous-processus à la main.

Pour terminer une session donnée tout en permettant aux autres de continuer, utilisez `pg_terminate_backend()` (voir Tableau 9.88) ou envoyez un signal SIGTERM au processus fils associé à cette session.

19.6. Mise à jour d'une instance PostgreSQL

Cette section concerne la mise à jour des données de votre serveur d'une version de PostgreSQL vers une version ultérieure.

Les numéros de versions actuelles de PostgreSQL se composent d'un numéro de version majeure et mineure. Par exemple, pour le numéro de version 10.1, 10 est le numéro de la version majeure et 1 est le numéro de la version mineure, ce qui signifie que c'est la première mise à jour mineure de la version majeure 10. Pour les versions précédant PostgreSQL 10.0, la numérotation des versions est composée de 3 numéros, par exemple 9.5.3. Dans ces cas, la version majeure est composée des deux premiers groupes de chiffres du numéro de version, par exemple 9.5, et la version mineure est le troisième chiffre, par exemple 3, signifiant que c'est la troisième version mineure de la version majeure 9.5.

Les versions mineures ne changent jamais le format de stockage interne et sont toujours compatibles avec les versions mineures précédentes et suivantes de la même version majeure. Par exemple, la version 10.1 est compatible avec la version 10.0 et la version 10.6. De même, par exemple, la version 9.5.3 est compatible avec 9.5.0, 9.5.1 et 9.5.6. Pour mettre à jour entre versions compatibles, il suffit de remplacer les exécutables lorsque le serveur est arrêté et de redémarrer le serveur. Le répertoire de données reste inchangé : les mises à jour mineures sont aussi simples que cela.

Pour les versions *majeures* de PostgreSQL, le format de stockage interne des données est sujet à modification, ce qui complique les mises à jour. La méthode traditionnelle de migration des données vers une nouvelle version majeure est de sauvegarder puis recharger la base de données, même si cela peut être lent. pg_upgrade est une méthode plus rapide. Des méthodes de réplication sont aussi disponibles, comme discuté ci-dessus. (Si vous utilisez une version pré-packagée de PostgreSQL, ce dernier pourrait fournir des scripts pour vous aider dans les mises à jour de versions majeures. Consultez la documentation du paquet pour les détails.)

De plus, les nouvelles versions majeures introduisent généralement des incompatibilités qui impactent les utilisateurs. Du coup, des modifications peuvent être nécessaires sur les applications clientes. Tous les changements visibles par les utilisateurs sont listés dans les notes de version (Annexe E). Soyez particulièrement attentif à la section Migration. Bien que vous puissiez mettre à jour d'une version majeure vers une autre sans passer par les versions intermédiaires, vous devez lire les notes de version de toutes les versions majeures intermédiaires.

Les utilisateurs précautionneux testeront leurs applications clientes sur la nouvelle version avant de basculer complètement. Du coup, il est souvent intéressant de mettre en place des installations

parallèles des anciennes et nouvelles versions. Lors d'un test d'une mise à jour majeure de PostgreSQL, pensez aux différentes catégories suivantes :

Administration

Les fonctionnalités disponibles pour les administrateurs pour surveiller et contrôler le serveur s'améliorent fréquemment à chaque nouvelle version.

SQL

Cela inclut généralement les nouvelles commandes ou clauses SQL, et non pas des changements de comportement sauf si c'est spécifiquement précisé dans les notes de version.

API

Les bibliothèques comme libpq se voient seulement ajouter de nouvelles fonctionnalités, sauf encore une fois si le contraire est mentionné dans les notes de version.

Catalogues systèmes

Les modifications dans les catalogues systèmes affectent seulement les outils de gestion des bases de données.

API serveur pour le langage C

Ceci implique des modifications dans l'API des fonctions du moteur qui est écrit en C. De telles modifications affectent le code qui fait référence à des fonctions du moteur.

19.6.1. Mettre à jour les données via pg_dumpall

Une méthode de mise à jour revient à sauvegarder les données d'une version majeure de PostgreSQL et de la recharger dans une autre -- pour cela, vous devez utiliser un outil de sauvegarde *logique* comme pg_dumpall ; une sauvegarde au niveau système de fichiers ne fonctionnera pas. Des vérifications sont faites pour vous empêcher d'utiliser un répertoire de données avec une version incompatible de PostgreSQL, donc aucun mal ne sera fait si vous essayez de lancer un serveur d'une version majeure sur un répertoire de données créé par une autre version majeure.)

Il est recommandé d'utiliser les programmes pg_dump et pg_dumpall provenant de la *nouvelle* version de PostgreSQL, pour bénéficier des améliorations apportées à ces programmes. Les versions actuelles de ces programmes peuvent lire des données provenant de tout serveur dont la version est supérieure ou égale à la 9.2.

Ces instructions supposent que votre installation existante se trouve dans le répertoire `/usr/local/pgsql` et que le répertoire des données est `/usr/local/pgsql/data`. Remplacez ces chemins pour correspondre à votre installation.

1. Si vous faites une sauvegarde, assurez-vous que votre base de données n'est pas en cours de modification. Cela n'affectera pas l'intégrité de la sauvegarde mais les données modifiées ne seront évidemment pas incluses. Si nécessaire, modifiez les droits dans le fichier `/usr/local/pgsql/data/pg_hba.conf` (ou équivalent) pour interdire l'accès à tout le monde sauf vous. Voir Chapitre 21 pour plus d'informations sur le contrôle des accès.

   Pour sauvegarder votre installation, exécutez la commande suivante :

   ```
   pg_dumpall > fichier_en_sortie
   ```

   Pour faire la sauvegarde, vous pouvez utiliser la commande pg_dumpall de la version en cours d'exécution ; voir Section 26.1.2 pour plus de détails. Néanmoins, pour de meilleurs résultats, essayez d'utiliser la commande pg_dumpall provenant de la version 15.19 de PostgreSQL, car

cette version contient des corrections de bugs et des améliorations par rapport aux anciennes versions. Bien que ce conseil puisse sembler étonnant, étant donné que vous n'avez pas encore été la nouvelle version, il est conseillé de le suivre si vous souhaitez installer la nouvelle version en parallèle de l'ancienne. Dans ce cas, vous pouvez terminer l'installation normalement et transférer les données plus tard. Cela diminuera aussi le temps d'immobilisation.

2. Arrêtez l'ancien serveur :

```
pg_ctl stop
```

Sur les systèmes qui lancent PostgreSQL au démarrage, il existe probablement un script de démarrage qui fera la même chose. Par exemple, sur un système Red Hat Linux, cette commande pourrait fonctionner :

```
/etc/rc.d/init.d/postgresql stop
```

Voir Chapitre 19 pour des détails sur le lancement et l'arrêt d'un serveur.

3. Lors de la restauration de la sauvegarde, renommez ou supprimez l'ancien répertoire d'installation si ce n'est pas spécifique à la version. Il est préférable de le renommer car, en cas de problème, vous pourrez le récupérer. Garder en tête que le répertoire peut prendre beaucoup d'espace disque. Pour renommer le répertoire, utilisez une commande comme celle-ci :

```
mv /usr/local/pgsql /usr/local/pgsql.old
```

(Assurez-vous de déplacer le répertoire en un seul coup, pour que les chemins relatifs restent inchangés.)

4. Installez la nouvelle version de PostgreSQL comme indiqué dans la section suivante Section 17.4.

5. Créez une nouvelle instance de bases de données si nécessaire. Rappelez-vous que vous devez exécuter ces commandes une fois connecté en tant que l'utilisateur de bases de données (que vous devez déjà avoir si vous faites une mise à jour).

```
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
```

6. Restaurez vos modifications dans les fichiers `pg_hba.conf` et `postgresql.conf`.

7. Démarrez le serveur de bases de données, en utilisant encore une fois l'utilisateur de bases de données :

```
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data
```

8. Enfin, restaurez vos données à partir de votre sauvegarde :

```
/usr/local/pgsql/bin/psql -d postgres -f outputfile
```

en utilisant le *nouveau* psql.

Il est possible de parvenir à une immobilisation moins longue en installant le nouveau serveur dans un autre répertoire et en exécutant l'ancien et le nouveau serveur, en parallèle, sur des ports différents. Vous pouvez ensuite utiliser quelque chose comme :

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

pour transférer vos données.

19.6.2. Mettre à jour les données via pg_upgrade

Le module pg_upgrade permet la mise à jour en ligne d'une installation d'une version majeure de PostgreSQL vers une autre. Les mises à jour se font en quelques minutes, notamment avec le mode `--link`. Il requiert les mêmes étapes que pour pg_dumpall ci-dessus, autrement dit lancer/arrêter le serveur, lancer initdb. La documentation de pg_upgrade surligne les étapes nécessaires.

19.6.3. Mettre à jour les données via la réplication

Il est aussi possible d'utiliser des méthodes de réplication logique pour créer un serveur secondaire avec une version plus récente de PostgreSQL. C'est possible car la réplication logique permet une réplication entre des versions majeures différentes de PostgreSQL. Le secondaire peut se trouver sur le même serveur ou sur un autre. Une fois qu'il est synchronisé avec le serveur primaire (qui utilise toujours l'ancienne version de PostgreSQL), vous pouvez basculer le serveur primaire sur le nouveau serveur et arrêter l'ancien primaire. Ce type de bascule fait que l'arrêt requis pour la mise à jour se mesure seulement en secondes.

Cette méthode de mise à jour peut être mise en œuvre avec la réplication logique intégrée comme avec des outils de réplication logique tels que pglogical, Slony, Londiste, et Bucardo.

19.7. Empêcher l'usurpation de serveur (spoofing)

Quand le serveur est en cours d'exécution, un utilisateur mal intentionné ne peut pas interférer dans les communications client/serveur. Néanmoins, quand le serveur est arrêté, un utilisateur local peut usurper le rôle du serveur normal en lançant son propre serveur. Le serveur usurpateur pourrait lire les mots de passe et les requêtes envoyés par les clients, mais ne pourrait pas renvoyer de données car le répertoire PGDATA est toujours sécurisé grâce aux droits d'accès du répertoire. L'usurpation est possible parce que tout utilisateur peut lancer un serveur de bases de données ; un client ne peut pas identifier un serveur invalide sauf s'il est configuré spécialement.

Un moyen d'empêcher les serveurs d'usurper des connexions `local`es est d'utiliser un répertoire de socket de domaine Unix (unix_socket_directories) qui n'a un droit en écriture que pour un utilisateur local de confiance. Ceci empêche un utilisateur mal intentionné de créer son propre fichier socket dans ce répertoire. Si vous craignez que certaines applications puissent encore référencer `/tmp` pour le fichier socket et, du coup, être vulnérable au « spoofing », créez un lien symbolique `/tmp/.s.PGSQL.5432` pointant vers le fichier socket déplacé. Vous pourriez alors avoir besoin de modifier votre script de nettoyage de `/tmp` pour empêcher la suppression du lien symbolique.

Une autre option pour les connexions de type `local` est que les clients utilisent `requirepeer` pour indiquer un propriétaire précis du processus serveur connecté au socket.

Pour éviter l'usurpation sur les connexions TCP, utilisez des certificats SSL, et assurez-vous que les clients vérifient le certificat du serveur, ou bien utilisez le chiffrage GSSAPI (ou les deux, s'il s'agit de connexions séparées).

Pour éviter l'usurpation avec SSL, le serveur doit être configuré pour accepter les connexions `hostssl` (Section 21.1) et avoir des fichiers SSL clé et certificat (Section 19.9). Le client TCP doit se connecter en utilisant `sslmode='verify-ca'` ou `verify-full` et le fichier certificat root approprié doit y être installé (Section 34.19.1).

Pour éviter l'usurpation avec GSSAPI, le serveur doit être configuré pour n'accepter que les connexions `hostgssenc` (Section 21.1) et l'authentification `gss` avec elles. Le client TCP doit se connecter en utilisant `gssencmode=require`.

19.8. Options de chiffrement

PostgreSQL offre du chiffrement à différents niveaux et offre une certaine flexibilité pour éviter que les données soient révélées à cause d'un vol du serveur de la base de données, d'administrateurs peu scrupuleux et de réseaux non sécurisés. Le chiffrement pourrait aussi être requis pour sécuriser des données sensibles, par exemple des informations médicales ou des transactions financières.

Chiffrement du mot de passe

Les mots de passe des utilisateurs de la base de données sont stockées suivant des hachages (déterminés par la configuration du paramètre password_encryption), donc l'administrateur ne peut pas déterminer le mot de passe actuellement affecté à l'utilisateur. Si le chiffrage MD5 ou SCRAM est utilisé pour l'authentification du client, le mot de passe non chiffré n'est jamais, y compris temporairement, présent sur le serveur parce que le client le chiffre avant de l'envoyer sur le réseau. SCRAM est à préférer parce qu'il s'agit d'un standard Internet et qu'il est bien plus sécurisé que le protocole d'authentification MD5 spécifique à PostgreSQL.

chiffrement de colonnes spécifiques

Le module pgcrypto autorise le stockage chiffré de certains champs. Ceci est utile si seulement certaines données sont sensibles. Le client fournit la clé, la donnée est déchiffrée sur le serveur puis envoyée au client.

La donnée déchiffrée et la clé de déchiffrement sont présentes sur le serveur pendant un bref moment où la donnée est déchiffrée, puis envoyée entre le client et le serveur. Il y a donc un bref moment où la donnée et les clés peuvent être interceptées par quelqu'un ayant un accès complet au serveur de bases de données, tel que l'administrateur du système.

chiffrement de la partition de données

Le chiffrement du stockage peut se réaliser au niveau du système de fichiers ou au niveau du bloc. Les options de chiffrement des systèmes de fichiers sous Linux incluent eCryptfs et EncFS, alors que FreeBSD utilise PEFS. Les options de chiffrement au niveau bloc ou au niveau disque incluent dm-crypt + LUKS sur Linux et les modules GEOM geli et gbde sur FreeBSD. Beaucoup d'autres systèmes d'exploitation supportent cette fonctionnalité, y compris Windows.

Ce mécanisme empêche les données non chiffrées d'être lues à partir des lecteurs s'ils sont volés. Ceci ne protège pas contre les attaques quand le système de fichiers est monté parce que, une fois monté, le système d'exploitation fournit une vue non chiffrée des données. Néanmoins, pour monter le système de fichiers, vous avez besoin d'un moyen pour fournir la clé de chiffrement au système d'exploitation et, quelquefois, la clé est stocké quelque part près de l'hôte qui monte le disque.

chiffrement des données sur le réseau

Les connexions SSL chiffrent toutes les données envoyées sur le réseau : le mot de passe, les requêtes et les données renvoyées. Le fichier `pg_hba.conf` permet aux administrateurs de spécifier quels hôtes peuvent utiliser des connexions non chiffrées (`host`) et lesquels requièrent des connexions SSL (`hostssl`). De plus, les clients peuvent spécifier qu'ils se connectent aux serveurs seulement via SSL.

Les connexions chiffrées par GSSAPI chiffrent toutes les données envoyées sur le réseau, dont les requêtes et les données renvoyées. (Aucun mot de passe n'est envoyé sur le réseau). Le fichier `pg_hba.conf` permet aux administrateurs d'indiquer quels clients peuvent utiliser des connexions non chiffrées (`host`) et lesquels nécessitent des connexions chiffrées par GSSAPI (`hostgssenc`). Les clients peuvent aussi préciser qu'ils ne se connecteront qu'avec des connexions chiffrées par GSSAPI (`gssencmode=require`).

Stunnel ou SSH peuvent aussi être utilisés pour chiffrer les transmissions.

Authentification de l'hôte SSL

Il est possible que le client et le serveur fournissent des certificats SSL à l'autre. Cela demande une configuration supplémentaire de chaque côté mais cela fournit une vérification plus forte de l'identité que la simple utilisation de mots de passe. Cela empêche un ordinateur de se faire passer pour le serveur assez longtemps pour lire le mot de passe envoyé par le client. Cela empêche aussi les attaques du type « man in the middle » où un ordinateur, entre le client et le serveur, prétend être le serveur, lit et envoie les données entre le client et le serveur.

Chiffrement côté client

Si vous n'avez pas confiance en l'administrateur système du serveur, il est nécessaire que le client crypte les données ; de cette façon, les données non cryptées n'apparaissent jamais sur le serveur de la base de données. Les données sont cryptées sur le client avant d'être envoyé au serveur, et les résultats de la base de données doivent être décryptés sur le client avant d'être utilisés.

19.9. Connexions TCP/IP sécurisées avec SSL

PostgreSQL dispose d'un support natif pour l'utilisation de connexions SSL et le chiffrage des communications client/serveur pour améliorer la sécurité améliorée. Ceci requiert l'installation d'OpenSSL à la fois sur le système client et sur le système serveur et que ce support soit activé au moment de la construction de PostgreSQL (voir le Chapitre 17).

Les termes SSL et TLS sont souvent utilisés de façon interchangeable pour signifier une connexion chiffrée sécurisée en utilisant un protocole TLS. Les protocoles SSL sont les précurseurs des protocoles TLS, et le terme SSL est encore utilisée pour des connexions chiffrées même si les protocoles SSL ne sont plus acceptées. SSL est utilisé de façon interchangeable avec TLS dans PostgreSQL.

19.9.1. Configuration basique

Avec SSL intégré à la compilation, le serveur PostgreSQL peut être démarré avec le support des connexions chiffrées utilisant les protocoles TLS activé en positionnant le paramètre ssl à `on` dans `postgresql.conf`. Le serveur écoutera les deux types de connexion, normal et SSL, sur le même port TCP, et négociera l'utilisation de SSL avec chaque client. Par défaut, c'est au choix du client ; voir Section 21.1 pour la configuration du serveur pour exiger SSL pour tout ou partie des connexions.

Pour démarrer dans le mode SSL, les fichiers contenant le certificat du serveur et la clé privée doivent exister. Par défaut, ces fichiers sont nommés respectivement `server.crt` et `server.key`, et sont placés dans le répertoire des données du serveur. D'autres noms et emplacements peuvent être spécifiés en utilisant les paramètres ssl_cert_file et ssl_key_file.

Sur les systèmes Unix, les droits de `server.key` doivent interdire l'accès au groupe et au reste du monde ; cela se fait avec la commande `chmod 0600 server.key`. Il est aussi possible de faire en sorte que le fichier ait root comme propriétaire et des droits de lecture pour le groupe (autrement dit, des droits `0640`). Cette configuration cible les installations où les fichiers certificat et clé sont gérés par le système d'exploitation. L'utilisateur qui exécute le serveur PostgreSQL doit être un membre du groupe qui a accès aux fichiers certificat et clé.

Si le répertoire des données permet l'accès en lecture au groupe, alors les fichiers de certificat doivent être placés hors de ce répertoire pour se conformer aux exigences de sécurité décrites ci-dessus. Généralement, l'accès au groupe est autorisé pour permettre à un utilisateur non privilégié de sauvegarder la base, et dans ce cas le logiciel de sauvegarde sera incapable de lire les certificats et retournera probablement une erreur.

Si la clé privée est protégée par une phrase de passe, le serveur la demandera et ne se lancera pas tant qu'elle n'aura pas été saisie. Utiliser une phrase de passe par défaut empêche également la possibilité de modifier la configuration SSL du serveur sans redémarrage. Voir

ssl_passphrase_command_supports_reload. De plus, les clés privées protégées par phrases de passe ne peuvent être utilisées sur Windows.

Le premier certificat dans `server.crt` doit être le certificat du serveur car il doit correspondre à la clé privée du serveur. Les certificats des autorités « intermédiaires » d'autorité peuvent aussi être ajoutés au fichier. Le faire permet d'éviter la nécessité d'enregistrer les certificats intermédiaires des clients, en supposant que le certificat racine et les certificats intermédiaires ont été créés avec les extensions `v3_ca`. (Ceci configure la contrainte basique du certificat de `CA` à `true`.) Ceci permet une expiration plus simple des certificats intermédiaires.

Il n'est pas nécessaire d'ajouter le certificat racine dans le fichier `server.crt`. À la place, les clients doivent avoir le certificat racine de la chaîne de certificats du serveur.

19.9.2. Configuration d'OpenSSL

PostgreSQL lit le fichier de configuration OpenSSL du système. Par défaut, ce fichier est nommé `openssl.cnf` et est situé dans le répertoire désigné par `openssl version -d`. Ce défaut peut être surchargé en remplissant la variable d'environnement `OPENSSL_CONF` avec le nom du fichier de configuration désiré.

OpenSSL supporte une large gamme d'algorithmes de chiffrement et d'authentification de forces variables. Bien qu'une liste des techniques de chiffrement soit spécifiée dans le fichier de configuration d'OpenSSL, vous pouvez préciser les chiffrements à utiliser par le serveur en modifiant ssl_ciphers dans `postgresql.conf`.

> **Note**
>
> Il est possible d'avoir une authentification sans le coût du chiffrement en utilisant les chiffrements `NULL-SHA` ou `NULL-MD5` ciphers. Cependant, un « homme du milieu » (*man-in-the-middle*) pourrait lire et transmettre des communications entre client et serveur. Pour ces raisons, les chiffrements NULL ne sont pas recommandés.

19.9.3. Utiliser des certificats clients

Pour réclamer au client de fournir un certificat de confiance, placez les certificats des autorités certificats racines (CA) dont vous avez confiance dans un fichier du répertoire des données, configurez le paramètre ssl_ca_file dans `postgresql.conf` au nouveau nom du fichier, et ajoutez l'option d'authentification `clientcert=verify-ca` ou `clientcert=verify-full` sur la ligne `hostssl` approprié dans `pg_hba.conf`. Un certificat sera alors réclamé du client lors du démarrage de la connexion SSL. (Voir Section 34.19 pour une description sur la configuration des certificats sur le client.)

Pour une entrée `hostssl` avec `clientcert=verify-ca` , le serveur vérifiera que le certificat du client est signé par l'une des autorités de certification de confiance. Si `clientcert=verify-full` est spécifié, le serveur ne vérifiera pas seulement la chaîne de certificats, mais il vérifiera également si l'utilisateur ou sa correspondance concorde au `cn` (Common Name) du certificat fourni. Notez que la validation de la chaîne de certificats est toujours assurée lorsque la méthode d'authentification `cert` est utilisée (voir Section 21.12).

Les certificats intermédiaires chaînés jusqu'aux certificats racines existants peuvent aussi apparaître dans le fichier `root.crt` si vous souhaitez éviter d'avoir à les stocker sur les clients (en supposant que les certificats racine et intermédiaires ont été créés avec les extensions `v3_ca`). Les entrées dans la liste de révocation de certificats (CRL) sont aussi vérifiées si le paramètre ssl_crl_file ou ssl_crl_dir est configuré.

L'option d'authentification `clientcert` est disponible pour toutes les méthodes d'authentification, mais seulement pour les lignes du fichier `pg_hba.conf` indiquées avec `hostssl`. Quand

`clientcert` n'est pas configuré, le serveur vérifie toujours avec son fichier CA le certificat client présenté, s'il est configuré.

Il y a deux approches pour forcer les utilisateurs à fournir un certificat à la connexion.

La première approche fait usage de la méthode d'authentification `cert` pour les entrées `hostssl` de `pg_hba.conf`, afin que le certificat lui-même soit utilisé pour l'authentification tout en fournissant la sécurité de la connexion SSL. Voir les détails dans Section 21.12. (Il n'est pas nécessaire de spécifier explicitement d'options `clientcert` avec la méthode d'authentification `cert`.) Dans ce cas, le `cn` (*Common Name*) dans le certificat est comparé au nom d'utilisateur ou au mappage applicable.

La seconde approche combine n'importe quelle méthode d'authentification pour les entrées `hostssl` avec la vérification des certificats des clients, en positionnant l'option d'authentification `clientcert` à `verify-ca` ou `verify-full`. La première option ne garantit que la validité du certificat, alors que la seconde vérifie aussi que le `cn` (*Common Name*) correspond au nom d'utilisateur ou au mappage applicable.

19.9.4. Utilisation des fichiers serveur SSL

Tableau 19.2 résume les fichiers qui ont un lien avec la configuration de SSL sur le serveur. (Les noms de fichiers indiqués sont les noms par défaut. Les noms configurés localement peuvent être différents.)

Tableau 19.2. Utilisation des fichiers serveur SSL

Fichier	Contenu	Effet
ssl_cert_file (`$PGDATA/server.crt`)	certificat du serveur	envoyé au client pour indiquer l'identité du serveur
ssl_key_file (`$PGDATA/server.key`)	clé privée du serveur	prouve que le certificat serveur est envoyé par son propriétaire n'indique pas que le propriétaire du certificat est de confiance
ssl_ca_file (`$PGDATA/root.crt`)	autorités de confiance pour les certificats	vérifie le certificat du client ; vérifie que le certificat du client est signé par une autorité de confiance
ssl_crl_file (`$PGDATA/root.crl`)	certificats révoqués par les autorités de confiance	le certificat du client ne doit pas être sur cette liste

Le serveur lit ces fichiers lors de son démarrage et à chaque rechargement de la configuration serveur. Sur les systèmes Windows, ils sont également relus chaque fois qu'un nouveau processus est démarré pour une nouvelle connexion client.

Si une erreur est détectée dans ces fichiers lors du démarrage du serveur, celui-ci refusera de démarrer. Par contre, si une erreur est détectée lors d'un rechargement de configuration, ces fichiers seront ignorés et l'ancienne configuration SSL continuera d'être utilisée. Sur les systèmes Windows, si une erreur est détectée dans ces fichiers au démarrage d'un processus backend, celui-ci ne pourra établir une connexion SSL. Dans tous les cas, l'erreur sera rapportée dans les journaux du serveur.

19.9.5. Créer des certificats

Pour créer un certificat simple auto-signé pour le serveur, valide pour 365 jours, utilisez la commande OpenSSL suivante, en remplaçant *`dbhost.yourdomain.com`* avec le nom d'hôte du serveur

```
openssl req -new -x509 -days 365 -nodes -text -out server.crt \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

Puis, exécutez :

```
chmod og-rwx server.key
```

car le serveur rejetera le fichier si ses droits sont plus importants. Pour plus de détails sur la façon de créer la clé privée et le certificat de votre serveur, référez-vous à la documentation d'OpenSSL.

Bien qu'un certificat auto-signé puisse être utilisé pour des tests, un certificat signé par une autorité de certificats (CA) (habituellement un CA racine entreprise) devrait être utilisé en production.

Pour créer un certificat serveur dont l'identité peut être validé par des clients, créez tout d'abord une demande de signature de certificat (CSR) et un fichier clés public/privé :

```
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
```

Puis, signez la demande avec la clé pour créer une autorité de certificat racine (en utilisant l'emplacement du fichier de configuration OpenSSL par défaut sur Linux) :

```
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt
```

Enfin, créez un certificat serveur signé par la nouvelle autorité de certificat racine :

```
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key

openssl x509 -req -in server.csr -text -days 365 \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out server.crt
```

`server.crt` et `server.key` doivent être stockés sur le serveur, et `root.crt` doit être stocké sur le client pour que le client puisse vérifier que le certificat feuille du serveur a été signé par son propre certificat racine de confiance. `root.key` doit être enregistré hors ligne pour l'utiliser pour créer les prochains certificats.

Il est aussi possible de créer une chaîne de confiance qui inclut les certificats intermédiaires :

```
# root
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt

# intermediate
openssl req -new -nodes -text -out intermediate.csr \
  -keyout intermediate.key -subj "/CN=intermediate.yourdomain.com"
chmod og-rwx intermediate.key
openssl x509 -req -in intermediate.csr -text -days 1825 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
```

```
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out intermediate.crt

# leaf
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
openssl x509 -req -in server.csr -text -days 365 \
  -CA intermediate.crt -CAkey intermediate.key -CAcreateserial \
  -out server.crt
```

`server.crt` et `intermediate.crt` doivent être concaténés dans un fichier certificat et stocké sur le serveur. `server.key` doit aussi être stocké sur le serveur. `root.crt` doit être stocké sur le client pour que le client puisse vérifier que le certificat feuille du serveur a été signé par une chaîne de certificats liés au certificat racine de confiance. `root.key` et `intermediate.key` doivent être stockées hors ligne pour être utilisé dans la création des certificats futurs.

19.10. Connexions TCP/IP sécurisées avec le chiffrage GSSAPI

PostgreSQL a aussi un support natif de GSSAPI pour améliorer la sécurité en chiffrant les communications client/serveur. Cela requiert que soit installée une implémentation de GSSAPI (comme MIT Kerberos) à la fois sur le client et le serveur, et que le support dans PostgreSQL soit activé à la compilation (voir Chapitre 17).

19.10.1. Mise en place de base

Le serveur PostgreSQL écoutera sur le même port TCP à la fois les connexions normales et celles chiffrées par GSSAPI, et négociera avec chaque client désirant se connecter s'il faut utiliser GSSAPI pour le chiffrage (et l'authentification). Par défaut, cette décision est celle du client (ce qui signifie qu'un attaquant peut en dégrader le niveau) ; voir Section 21.1 sur comment configurer le serveur pour exiger l'utilisation de GSSAPI pour tout ou partie des connexions.

Lors de l'utilisation de GSSAPI pour le chiffrement, il est commun d'utiliser GSSAPI aussi pour l'authentification car les mécanismes sous-jacents détermineront les identités du client et du serveur (d'après l'implémentation GSSAPI) dans tous les cas. Ceci n'est pas requis. Une autre méthode d'authentification PostgreSQL peut être utilisée pour réaliser les vérifications supplémentaires.

À part la configuration de cette négociation, GSSAPI ne requiert aucune configuration au-delà de ce qui est nécessaire pour l'authentification GSSAPI. (Pour plus d'informations sur sa mise en place, voir Section 21.6.)

19.11. Connexions TCP/IP sécurisées avec des tunnels ssh tunnels

Il est possible d'utiliser ssh pour chiffrer la connexion réseau entre les clients et un serveur PostgreSQL. Réalisé correctement, ceci fournit une connexion réseau sécurisée, y compris pour les clients non SSL.

Tout d'abord, assurez-vous qu'un serveur ssh est en cours d'exécution sur la même machine que le serveur PostgreSQL et que vous pouvez vous connecter via `ssh` en tant qu'un utilisateur quelconque. Ensuite, vous pouvez établir un tunnel sécurisé vers le serveur distant. Un tunnel sécurisé écoute sur un port local et envoie tout le trafic vers un port de la machine distante. Le trafic envoyé vers le port distant peut arriver sur son adresse `localhost` ou vers une autre adresse liée si désirée , il n'apparaît pas comme venant de votre machine locale. Cette commande crée un tunnel sécurisé de la machine cliente vers la machine distante `foo.com` :

```
ssh -L 63333:localhost:5432 joe@foo.com
```

Le premier numéro de l'argument `-l`, 63333, est le numéro de port local du tunnel ; cela peut être tout port inutilisé. (IANA réserve les ports 49152 à 65535 pour une utilisation privée.) Le nom ou l'adresse IP après ça est l'adresse distante liée à laquelle vous vous connectez, par exemple `localhost`, ce qui est la valeur par défaut. Le deuxième nombre, 5432, est la fin distante du tunnel, autrement dit le numéro de port du serveur de bases de données. Pour vous connecter au serveur en utilisant ce tunnel, vous vous connectez au port 63333 de la machine locale :

```
psql -h localhost -p 63333 postgres
```

Sur le serveur de bases de données, il semblera que vous êtes l'utilisateur `joe` sur l'hôte `foo.com` en vous connectant à l'adresse liée `localhost` dans ce contexte, et il utilisera la procédure d'authentification configurée pour les connexions de cet utilisateur et de cet hôte. Notez que le serveur ne pensera pas que la connexion est chiffrée avec SSL car, en effet, elle n'est pas chiffrée entre le serveur SSH et le serveur PostgreSQL. Cela ne devrait pas poser un risque de sécurité supplémentaire parce que les deux serveurs sont sur la même machine.

Pour réussir la configuration du tunnel, vous devez être autorisé pour vous connecter via `ssh` sur `joe@foo.com`, comme si vous aviez tenté d'utiliser `ssh` pour créer une session de terminal.

Vous pouvez aussi configurer la translation de port de cette façon :

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

mais alors le serveur de la base de données verra la connexion venir de son adresse liée `foo.com` qui n'est pas ouverte par son paramétrage par défaut `listen_addresses = 'localhost'`. Ceci n'est pas habituellement ce que vous êtes.

Si vous devez vous connecter au serveur de bases de données via un hôte de connexion, une configuration possible serait :

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

Notez que de cette façon la connexion de `shell.foo.com` à `db.foo.com` ne sera pas chiffrée par le tunnel SSH. SSH offre un certain nombre de possibilités de configuration quand le réseau est restreint. Merci de vous référer à la documentation de SSH pour les détails.

> **Astuce**
>
> Plusieurs autres applications existantes peuvent fournir des tunnels sécurisés en utilisant une procédure similaire dans le concept à celle que nous venons de décrire.

19.12. Enregistrer le journal des événements sous Windows

Pour enregistrer une bibliothèque pour le journal des événements de Windows, lancez la commande :

```
regsvr32 répertoire_bibliothèques_postgres/pgevent.dll
```

Ceci crée les clés de registre utilisé par le visualisateur des événements, sous la source d'événement par défaut, nommée `PostgreSQL`.

Pour indiquer un nom de source différent (voir event_source), utilisez les options `/n` et `/i` :

```
regsvr32 /n /
i:nom_source_evenement répertoire_bibliothèques_postgres/
pgevent.dll
```

Pour désenregistrer la bibliothèque du journal des événements de Windows, lancez la commande :

```
regsvr32 /u [/
i:nom_source_evenement] répertoire_bibliothèques_postgres/
pgevent.dll
```

> **Note**
>
> Pour activer la journalisation des événements dans le serveur de base de données, modifiez log_destination pour include `eventlog` dans `postgresql.conf`.

Chapitre 20. Configuration du serveur

Un grand nombre de paramètres de configuration permettent de modifier le comportement du système de bases de données. Dans la première section de ce chapitre, les méthodes de configuration de ces paramètres sont décrites ; les sections suivantes discutent de chaque paramètre en détail.

20.1. Paramètres de configuration

20.1.1. Noms et valeurs des paramètres

Tous les noms de paramètres sont insensibles à la casse. Chaque paramètre prend une valeur d'un de ces cinq types : booléen, chaîne de caractères, entier, nombre à virgule flottante ou énumération(enum). Le type détermine la syntaxe pour configurer le paramètre :

- *Booléen* : les valeurs peuvent être écrites sous les formes `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0` (toutes insensibles à la casse) ou tout préfixe non ambigu basé sur un d'entre eux.

- *Chaîne de caractères* : En général, entoure la valeur de guillemets simples, doublant tout guillemet simple compris dans la valeur. Les guillemets peuvent habituellement être omis si la valeur est un nombre ou un identifiant simple. (Les valeurs qui correspondent à un mot clé SQL nécessitent d'être mises (entre guillemets dans certains contextes.)

- *Numérique (entier ou nombre à virgule flottante)* : les paramètres numériques peuvent être indiqués au format entier ou virgule flottante ; dans ce dernier cas, la valeur est arrondie à l'entier le plus proche si le paramètre est de type entier. Les paramètres de type entier acceptent en plus une valeur hexadécimale (commençant par `0x`) et une valeur octale (commençant par `0`), mais ces formats n'ont pas de virgule flottante. Ne pas utiliser de séparateurs pour les milliers. Les guillemets ne sont pas requis en dehors du format hexadécimal.

- *Numérique avec unité* : Quelques paramètres numériques ont une unité implicite car elles décrivent des quantités de mémoire ou de temps. L'unité pourra être des octets, des kilo-octets, des blocs (généralement 8 ko), des millisecondes, des secondes ou des minutes. Une valeur numérique sans unité pour un de ces paramètres utilisera l'unité par défaut du paramètre, qui est disponible dans le champ `pg_settings.unit`. Pour plus de facilité, les valeurs de certains paramètres peuvent se voir ajouter une unité explicitement, par exemple `'120 ms'` pour une valeur d'intervalle, et elles seront automatiquement converties suivant l'unité par défaut du paramètre. Notez que la valeur doit être écrite comme une chaîne de caractères (avec des guillemets) pour utiliser cette fonctionnalité. Le nom de l'unité est sensible à la casse, et il peut y avoir des espaces blancs entre la valeur numérique et l'unité.

  - Les unités valides de mémoire sont `B` (octets), `kB` (kilo-octets), `MB` (méga-octets), `GB` (giga-octets) et `TB` (téra-octets). Le multiplieur pour les unités de mémoire est 1024, et non pas 1000.

  - Les unités valides d'intervalle sont `us` (microsecondes), `ms` (millisecondes), `s` (secondes), `min` (minutes), `h` (heures) et `d` (jours).

  Si une valeur à virgule flottante est indiquée avec une unité, elle sera arrondie au multiple de la prochaine unité la plus petite. Par exemple, `30.1 GB` sera convertie en `30822 MB`, et non pas `32319628902 B`. Si le paramètre est de type entier, un arrondissement final en entier survient après toute conversion d'unité.

- *Énuméré* : Les valeurs des paramètres de type énuméré sont écrits de la même façon que les valeurs des paramètres de type chaînes de caractères mais sont restreintes à un ensemble limité de valeurs. Les valeurs autorisées d'un paramètre spécifique sont disponibles dans le champ `pg_settings.enumvals`. Les valeurs des paramètres de type énuméré ne sont pas sensibles à la casse.

20.1.2. Interaction avec les paramètres via le fichier de configuration

La façon fondamentale de configurer les paramètres est d'éditer le fichier `postgresql.conf`, qui est normalement conservé dans le répertoire des données. Une copie par défaut est installé dans le répertoire de l'instance lors de l'initialisation. Un exemple de contenu peut être :

```
# Ceci est un commentaire
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

Un paramètre est indiqué par ligne. Le signe égal entre le nom et la valeur est optionnel. Les espaces n'ont pas de signification (sauf à l'intérieur d'une valeur entre guillemets) et les lignes vides sont ignorées. Les symboles dièse (#) désignent le reste de la ligne comme un commentaire. Les valeurs des paramètres qui ne sont pas des identificateurs simples ou des nombres doivent être placées entre guillemets simples. Pour intégrer un guillemet simple dans la valeur d'un paramètre, on écrit soit deux guillemets (c'est la méthode préférée) soit un antislash suivi du guillemet. Si le fichier contient plusieurs entrées pour le même paramètre, elles sont toutes ignorées sauf la dernière.

Les paramètres configurés de cette façon fournissent des valeurs par défaut pour l'instance. Le paramétrage considéré par les sessions actives sera ces valeurs sauf si elles sont surchargées. Les sections suivantes décrivent les différentes façons dont bénéficient l'administrateur et l'utilisateur pour surcharger les valeurs par défaut.

Il existe aussi une directive `include_if_exists`, qui agit de la même façon que la directive `include`, sauf si le fichier n'existe pas ou ne peut pas être lu. La directive `include` traitera cela comme une erreur, mais la directive `include_if_exists` tracera cet événement et continuera le traitement du fichier de configuration.

Le fichier de configuration est relu à chaque fois que le processus principal du serveur reçoit un signal SIGHUP ; ce signal est facilement envoyé en exécutant `pg_ctl reload` sur la ligne de commande shell ou en appelant la fonction SQL `pg_reload_conf()`. Le processus principal propage aussi ce signal aux processus serveurs en cours d'exécution, pour que les sessions existantes récupèrent aussi les nouvelles valeurs (ceci survient après qu'elles aient terminées d'exécuter la commande en cours d'exécution pour le client). Il est aussi possible d'envoyer le signal à un processus serveur directement. Certains paramètres ne sont pris en compte qu'au démarrage du serveur ; tout changement de ces paramètres dans le fichier de configuration sera ignoré jusqu'au redémarrage du serveur. Les configurations invalides de paramètres sont aussi ignorées (mais tracées) lors du traitement du signal SIGHUP.

En plus du fichier `postgresql.conf`, un répertoire des données d'un serveur PostgreSQL contient un fichier `postgresql.auto.conf` , qui a le même format que le fichier `postgresql.conf`. Cependant, il a pour but d'être édité automatiquement et non pas manuellement. Ce fichier contient les configurations réalisées avec la commande `ALTER SYSTEM`. Ce fichier est lu quand le fichier `postgresql.conf` et lu, et son contenu prend effet de la même façon. Les paramètres configurés dans `postgresql.auto.conf` surchargent ceux configurés dans `postgresql.conf`.

Des outils externes peuvent aussi modifier `postgresql.auto.conf`. Il n'est pas recommandé de le faire alors que le serveur est en cours d'exécution car une commande `ALTER SYSTEM` exécutée en même temps pourrait écraser d'autres modifications. De tels outils devraient simplement ajouter les nouveaux paramètres à la fin ou ils pourraient choisir de supprimer les paramètres dupliqués et/ou supprimer les commentaires (comme le fait `ALTER SYSTEM`).

La vue système `pg_file_settings` peut être utile pour tester par avance des modifications dans les fichiers de configuration, ou pour diagnostiquer des problèmes si un signal SIGHUP n'a pas eu les effets désirés.

20.1.3. Interaction avec les paramètres via SQL

PostgreSQL fournit trois commandes SQL pour établir les valeurs par défaut de la configuration. La première, déjà mentionnée, est la commande `ALTER SYSTEM`. Elle fournit un moyen accessible via le SQL pour modifier les valeurs globales par défaut ; c'est l'équivalent fonctionnel de l'édition manuelle du fichier `postgresql.conf`. Il existe aussi deux commandes qui permettent la configuration des valeurs par défaut par base de données et par rôle :

- La commande `ALTER DATABASE` permet de surcharger le paramétrage global suivant la base de connexion.
- La commande `ALTER ROLE` permet de surcharger le paramétrage global suivant la base et l'utilisateur de connexion.

Les paramètres configurés avec `ALTER DATABASE` et `ALTER ROLE` sont appliqués seulement lors du démarrage d'une nouvelle session. Ils surchargent les valeurs obtenues dans les fichiers de configuration ou sur la ligne de commande du lancement du serveur. Ils constituent les valeurs par défaut pour le reste de la session. Notez que certains paramétrages ne peuvent pas être modifiés après le démarrage du serveur, et ne peuvent donc pas être configurés avec ces commandes (ou celles citées ci-dessous).

Une fois qu'un client est connecté à la base de données, PostgreSQL fournit deux commandes SQL supplémentaires (et fonctions équivalentes) pour interagir avec les paramètres de configuration de la session :

- La commande `SHOW` autorise l'inspection de la valeur actuelle de tous les paramètres. La fonction SQL correspondante est `current_setting(setting_name text)` (voir Section 9.27.1).
- La commande `SET` permet la modification de la valeur actuelle de certains paramètres qui peuvent être configurés localement pour une session. Elle n'a pas d'effet sur les autres sessions. Beaucoup de paramètres peuvent être configurés de cette façon par un utilisateur, mais certains peuvent seulement être configurés par des superutilisateurs et des utilisateurs qui ont reçu les droits `SET` sur ces paramètres. La fonction SQL correspondante est `set_config(setting_name, new_value, is_local)` (voir Section 9.27.1).

De plus, la vue système `pg_settings` peut être utilisée pour visualiser et modifier les valeurs locales à la session :

- Exécuter une requête sur cette vue est similaire à l'utilisation de la commande `SHOW ALL`. Cependant, elle fournit plus de détails et est beaucoup plus flexible, vu qu'il est possible d'ajouter des conditions de filtre et des jointures vers d'autres relations.
- Utiliser `UPDATE` sur cette vue, pour mettre à jour la colonne `setting`, est équivalent à exécuter la commande `SET`. Par exemple, l'équivalent de

```
SET paramètre_configuration TO DEFAULT;
```

est :

```
UPDATE pg_settings SET setting = reset_val WHERE name =
'paramètre_configuration';
```

20.1.4. Interaction avec les paramètre via le shell

En plus de pouvoir configurer les valeurs globales des paramètres et d'attacher une configuration spécifique aux bases et aux rôles, vous pouvez fournir un paramétrage à PostgreSQL via des options du shell. Le serveur et la bibliothèque client libpq acceptent des valeurs de paramètres via le shell.

- Lors du démarrage du serveur, des configurations de paramètres peuvent être passées à la commande `postgres` via le paramètre en ligne de commande `-c`. Par exemple,

  ```
  postgres -c log_connections=yes -c log_destination='syslog'
  ```

  Les paramétrages réalisés de cette façon surchargent ceux fournis dans le fichier `postgresql.conf` ou via la commande `ALTER SYSTEM`, donc ils ne peuvent pas être changés globalement sans redémarrer le serveur.

- Lors du démarrage d'une session client via libpq, un paramétrage peut être spécifié en utilisant la variable d'environnement `PGOPTIONS`. Le paramétrage établi ainsi constitue des valeurs par défaut pour la durée de la session, mais n'affecte pas les autres sessions. Pour des raisons historiques, le format de `PGOPTIONS` est similaire à celui utilisé lors du lancement de la commande `postgres`. Spécifiquement, l'option `-c` doit être indiquée. Par exemple :

  ```
  env PGOPTIONS="-c geqo=off -c statement_timeout=5min" psql
  ```

  Les autres clients et autres bibliothèques peuvent fournir leur propres mécanismes via la shell ou autrement, pour permettre à l'utilisateur de modifier le paramétrage de la session sans avoir à utiliser des commandes SQL.

20.1.5. Gestion du contenu des fichiers de configuration

PostgreSQL fournit plusieurs fonctionnalités pour diviser le fichier de configuration `postgresql.conf` en plusieurs sous-fichiers. Ces fonctionnalités sont tout particulièrement utiles quand plusieurs serveurs sont à gérer alors qu'ils partagent une partie de la configuration.

En plus des paramètres, le fichier `postgresql.conf` peut contenir des *directives d'inclusion*, qui précisent les autres fichiers à lire et à traiter comme s'ils étaient insérés dans le fichier de configuration à cet emplacement. Cette fonctionnalité permet de diviser un fichier de configuration en plusieurs parties séparées. Les directives d'inclusion ressemblent à :

```
include 'nom_fichier'
```

Si le nom du fichier n'est pas un chemin absolu, il est considéré comme relatif au répertoire contenant le fichier de configuration de référence. Les inclusions peuvent être imbriquées.

Il existe aussi une directive `include_if_exists` qui agit de la même façon que la directive `include` sauf si le fichier référencé n'existe pas ou ne peut pas être lu. La directive `include` considère ces états comme une condition d'erreur mais `include_if_exists` ne fait que tracer un message et continue le traitement du fichier de configuration de référence.

Le fichier `postgresql.conf` peut aussi contenir `include_dir directives`, qui précise un répertoire entier de fichiers de configuration à inclure. Il s'utilise de la même façon :

```
include_dir 'répertoire'
```

Les noms de répertoire relatifs sont pris comme ayant comme base le répertoire du fichier de configuration. Dans ce répertoire spécifique, seuls les fichiers dont le nom finit avec le suffixe `.conf`

seront inclus. Les noms de fichiers qui commencent avec le caractère `.` sont aussi ignorés, pour éviter des erreurs vu que ces fichiers sont cachés sur certaines plateformes. Plusieurs fichiers dans un répertoire d'inclusion sont traités dans l'ordre des noms de fichiers (d'après les règles de la locale C, autrement dit les numéros avant les lettres, et les majuscules avant les minuscules).

Les fichiers et répertoires inclus peuvent être utilisés pour séparer logiquement les portions de la configuration de la base de données, plutôt que d'avoir un gigantesque fichier `postgresql.conf`. Songez à une société qui a deux serveurs de bases de données, chacun avec une quantité de mémoire différente. Il existe vraisemblablement des éléments de la configuration qui vont être partagés entre les deux serveurs, comme par exemple la configuration des traces. Mais les paramètres relatifs à la mémoire sur le serveur varieront entre les deux. Et il pourrait aussi y avoir une personnalisation des serveurs. Une façon de gérer cette situation est de casser les changements de configuration en trois fichiers. Vous pouvre ajouter cela à la fin de votre fichier `postgresql.conf` pour les inclure :

```
include 'commun.conf'
include 'memoire.conf'
include 'serveur.conf'
```

Tous les systèmes auraient le même `commun.conf`. Chaque serveur avec une quantité particulière de mémoire pourrait partager le même `memory.conf`. Vous pourriez en avoir un pour tous les serveurs disposant de 8 Go de RAM, un autre pour ceux ayant 16 Go. Enfin, `serveur.conf` pourrait avoir les configurations réellement spécifiques à un serveur.

Une autre possibilité revient à créer un répertoire de fichiers de configuration et de placer les fichiers dans ce répertoire. Par exemple, un répertoire `conf.d` pourrait être référencé à la fin du `postgresql.conf` :

```
include_dir 'conf.d'
```

Ensuite, vous pourriez renommer les fichiers dans le répertoire `conf.d` de cette façon :

```
00commun.conf
01memoire.conf
02serveur.conf
```

Cette convention de nommage établit un ordre clair dans lequel ces fichiers sont chargés. C'est important parce que seul le dernier paramétrage d'un paramètre particulier sera utilisé lors de la lecture de la configuration par le serveur. Dans cet exemple, un paramètre configuré dans `conf.d/02server.conf` surchargera la configuration du même paramètre dans `conf.d/01memory.conf`.

Vous pouvez utiliser à la place cette approche pour nommer les fichiers de façon claire :

```
00commun.conf
01memoire-8Go.conf
02serveur-truc.conf
```

Ce type d'arrangement donne un nom unique pour chaque variation du fichier de configuration. Ceci peut aider à éliminer l'ambiguïté quand plusieurs serveurs ont leur configuration stockée au même endroit, par exemple dans un dépôt de contrôle de version. (Stocker les fichiers de configuration de la base avec un outil de contrôle de version est une autre bonne pratique à considérer.)

20.2. Emplacement des fichiers

En plus du fichier `postgresql.conf` déjà mentionné, PostgreSQL utilise deux autres fichiers de configuration éditables manuellement. Ces fichiers contrôlent l'authentification du client (leur utilisation est discutée dans le Chapitre 21). Par défaut, les trois fichiers de configuration sont stockés dans le répertoire data du cluster de bases de données. Les paramètres décrits dans cette section permettent de déplacer les fichiers de configuration. Ce qui peut en faciliter l'administration. Il est, en particulier, souvent plus facile de s'assurer que les fichiers de configuration sont correctement sauvegardés quand ils sont conservés à part.

`data_directory` (`string`)

Indique le répertoire à utiliser pour le stockage des données. Ce paramètre ne peut être initialisé qu'au lancement du serveur.

`config_file` (`string`)

Indique le fichier de configuration principal du serveur (appelé `postgresql.conf`). Ce paramètre ne peut être initialisé que sur la ligne de commande de `postgres`.

`hba_file` (`string`)

Indique le fichier de configuration de l'authentification fondée sur l'hôte (appelé `pg_hba.conf`). Ce paramètre ne peut être initialisé qu'au lancement du serveur.

`ident_file` (`string`)

Indique le fichier de configuration pour la correspondance des noms d'utilisateurs, fichier appelé `pg_ident.conf`). Voir Section 21.2 pour plus de détails. Ce paramètre ne peut être initialisé qu'au lancement du serveur.

`external_pid_file` (`string`)

Indique le nom d'un fichier supplémentaire d'identifiant de processus (PID) créé par le serveur à l'intention des programmes d'administration du serveur. Ce paramètre ne peut être initialisé qu'au lancement du serveur.

Dans une installation par défaut, aucun des paramètres ci-dessus n'est configuré explicitement. À la place, le répertoire des données est indiqué par l'option `-D` en ligne de commande ou par la variable d'environnement `PGDATA`. Les fichiers de configuration sont alors tous disponibles dans le répertoire des données.

Pour conserver les fichiers de configuration dans un répertoire différent de data, l'option `-D` de la ligne de commande `postgres` ou la variable d'environnement `PGDATA` doit pointer sur le répertoire contenant les fichiers de configuration. Le paramètre `data_directory` doit alors être configuré dans le fichier `postgresql.conf` (ou sur la ligne de commande) pour préciser où est réellement situé le répertoire des données. `data_directory` surcharge `-D` et `PGDATA` pour l'emplacement du répertoire des données, mais pas pour l'emplacement des fichiers de configuration.

les noms des fichiers de configuration et leur emplacement peuvent être indiqués individuellement en utilisant les paramètres `config_file`, `hba_file` et/ou `ident_file`. `config_file` ne peut être indiqué que sur la ligne de commande de `postgres` mais les autres peuvent être placés dans le fichier de configuration principal. Si les trois paramètres et `data_directory` sont configurés explicitement, alors il n'est pas nécessaire d'indiquer `-D` ou `PGDATA`.

Lors de la configuration de ces paramètres, un chemin relatif est interprété d'après le répertoire d'où est lancé `postgres`.

20.3. Connexions et authentification

20.3.1. Paramètres de connexion

`listen_addresses` (`string`)

Indique les adresses TCP/IP sur lesquelles le serveur écoute les connexions en provenance d'applications clientes. La valeur prend la forme d'une liste de noms d'hôte ou d'adresses IP numériques séparés par des virgules. L'entrée spéciale `*` correspond à toutes les interfaces IP disponibles. L'enregistrement `0.0.0.0` permet l'écoute sur toutes les adresses IPv4 et `::` permet l'écoute sur toutes les adresses IPv6. Si la liste est vide, le serveur n'écoute aucune interface IP, auquel cas seuls les sockets de domaine Unix peuvent être utilisées pour s'y connecter. Si la liste n'est pas vide, le serveur démarrera s'il peut écouter sur au moins une adresse IP. Un message d'avertissement sera émis pour les adresses TCP/IP qui ne peuvent pas être ouvertes. La valeur par défaut est localhost, ce qui n'autorise que les connexions TCP/IP locales de type « loopback ».

Bien que l'authentification client (Chapitre 21) permet un contrôle très fin sur les accès au serveur, `listen_addresses` contrôle les interfaces pouvant accepter des tentatives de connexion, ce qui permet d'empêcher des demandes répétées de connexion malveillantes sur des interfaces réseau non sécurisées. Ce paramètre ne peut être configuré qu'au lancement du serveur.

`port` (`integer`)

Le port TCP sur lequel le serveur écoute ; 5432 par défaut. Le même numéro de port est utilisé pour toutes les adresses IP que le serveur écoute. Ce paramètre ne peut être configuré qu'au lancement du serveur.

`max_connections` (`integer`)

Indique le nombre maximum de connexions concurrentes au serveur de base de données. La valeur par défaut typique est de 100 connexions, mais elle peut être moindre si les paramètres du noyau ne le supportent pas (ce qui est déterminé lors de l'initdb). Ce paramètre ne peut être configuré qu'au lancement du serveur.

Lors de l'exécution d'un serveur en attente, vous devez configurer ce paramètre à la même valeur ou à une valeur plus importante que sur le serveur primaire. Sinon, des requêtes pourraient ne pas être autorisées sur le serveur en attente.

Lorsque vous modifiez cette valeur, pensez également à ajuster max_parallel_workers, max_parallel_maintenance_workers et max_parallel_workers_per_gather.

`superuser_reserved_connections` (`integer`)

Indique le nombre de connecteurs (« slots ») réservés aux connexions des superutilisateurs PostgreSQL. Au plus max_connections connexions peuvent être actives simultanément. Dès que le nombre de connexions simultanément actives atteint `max_connections` moins `superuser_reserved_connections`, les nouvelles connexions ne sont plus acceptées que pour les superutilisateurs, et aucune nouvelle connexion de réplication ne sera acceptée.

La valeur par défaut est de trois connexions. La valeur doit être plus petite que la valeur de `max_connections`. Ce paramètre ne peut être configuré qu'au lancement du serveur.

`unix_socket_directories` (`string`)

Indique le répertoire pour le(s) socket(s) de domaine Unix sur lequel le serveur va écouter les connexions des applications clientes. Plusieurs sockets peuvent être créés en listant plusieurs répertoires et en les séparant par des virgules. Les espaces blancs entre les entrées sont ignorés. Entourer un nom de répertoire avec des guillemets doubles si vous avez besoin d'inclure un espace

blanc ou une virgule dans son nom. Une valeur vide désactive l'utilisation des sockets de domaine Unix, auquel cas seules les sockets TCP/IP pourront être utilisées pour se connecter au serveur.

Une valeur commençant par `@` indique qu'un socket abstrait doit être créé (actuellement uniquement supporté sous Linux). Dans ce cas, la valeur ne doit pas spécifier un « répertoire » mais un préfixe à partir duquel le nom du socket est construit de la même manière que pour un socket de système de fichiers. Alors que le préfixe d'un socket réseau est totalement libre, puisqu'il ne s'agit pas d'un emplacement sur disque, il est conseillé par convention d'utiliser des valeurs qui ressemblent à celles des sockets de systèmes de fichiers comme `@/tmp`.

La valeur par défaut est habituellement `/tmp`, mais cela peut se changer au moment de la contruction. Sur Windows, la valeur par défaut est vide, ce qui signifie qu'aucun socket de domaine Unix n'est créé par défaut. Ce paramètre ne peut être configuré qu'au lancement du serveur.

En plus du fichier socket, qui est nommé `.s.PGSQL.`*`nnnn`* où *`nnnn`* est le numéro de port du serveur, un fichier ordinaire nommé `.s.PGSQL.`*`nnnn`*`.lock` sera créé dans chaque répertoire de `unix_socket_directories`. Les deux fichiers ne doivent pas être supprimés manuellement. Pour les sockets réseau, aucun fichier de verrou n'est créé.

Ce paramètre n'a pas de sens sur les systèmes qui ignorent complètement les droits sur les sockets, comme Solaris à partir de la version 10. Un effet similaire peut être atteint en pointant `unix_socket_directories` vers un répertoire ayant un droit de recherche limité à l'audience acceptée.

`unix_socket_group` (`string`)

Configure le groupe propriétaire des sockets de domaine Unix (l'utilisateur propriétaire des sockets est toujours l'utilisateur qui lance le serveur). En combinaison avec le paramètre `unix_socket_permissions`, ceci peut être utilisé comme un mécanisme de contrôle d'accès supplémentaire pour les connexions de domaine Unix. Par défaut, il s'agit d'une chaîne vide, ce qui sélectionne le groupe par défaut de l'utilisateur courant. Ce paramètre ne peut être configuré qu'au lancement du serveur.

Ce paramètre n'est pas supporté sous Windows. Sa configuration sera ignorée. De plus, les sockets réseaux n'ayant pas de propriétaire, ce paramètre est ignoré dans ce cas.

`unix_socket_permissions` (`integer`)

Configure les droits d'accès aux sockets de domaine Unix. Ce socket utilise l'ensemble habituel des droits du système de fichiers Unix. Ce paramètre doit être indiqué sous une forme numérique telle qu'acceptée par les appels système `chmod` et `umask` (pour utiliser le format octal, ce nombre doit commencer avec un `0` (zéro)).

Les droits par défaut sont `0777`, signifiant que tout le monde peut se connecter. Les alternatives raisonnables sont `0770` (utilisateur et groupe uniquement, voir aussi `unix_socket_group`) et `0700` (utilisateur uniquement) (pour un socket de domaine Unix, seul le droit d'accès en écriture importe ; il n'est donc pas nécessaire de donner ou de révoquer les droits de lecture ou d'exécution).

Ce mécanisme de contrôle d'accès est indépendant de celui décrit dans le Chapitre 21.

Ce paramètre ne peut être configuré qu'au lancement du serveur.

Ce paramètre est hors sujet sur certains systèmes comme Solaris, à partir de sa version 10, qui ignorent complètement les droits des sockets. Là, vous pouvez obtenir le même effet en pointant `unix_socket_directories` vers un répertoire ayant les droits de recherche limités à l'audience souhaité.

Les sockets réseaux n'ayant pas de droits de fichier, ce paramètre est également ignoré dans ce cas.

`bonjour` (`boolean`)

Active la promotion de l'existence du serveur via le protocole Bonjour. Désactivé par défaut, ce paramètre ne peut être configuré qu'au lancement du serveur.

`bonjour_name` (`string`)

Indique le nom du service Bonjour. Le nom de l'ordinateur est utilisé si ce paramètre est configuré avec une chaîne vide (ce qui est la valeur par défaut). Ce paramètre est ignoré si le serveur n'est pas compilé avec le support Bonjour. Ce paramètre ne peut être configuré qu'au lancement du serveur.

`tcp_keepalives_idle` (`integer`)

Indique la durée sans activité réseau après laquelle le système d'exploitation devra envoyer un message TCP au client. Si la valeur de ce paramètre est indiquée sans unité, la valeur est supposée être en secondes. Une valeur de 0 (valeur par défaut) utilise la valeur par défaut du système d'exploitation. Ce paramètre est seulement supporté par les systèmes qui supportent les symboles `TCP_KEEPIDLE` ou une option socket équivalente et sur Windows ; sur les autres systèmes, ce paramètre doit valoir zéro. Pour les sessions connectées via une socket de domaine Unix, ce paramètre est ignoré et vaut toujours zéro.

> **Note**
>
> Sur Windows, une valeur de 0 configurera ce paramètre à deux heures car Windows ne fournit pas un moyen de lire la valeur par défaut du système.

`tcp_keepalives_interval` (`integer`)

Indique la durée après laquelle un message TCP keepalive qui n'a pas été répondu par le client devra être transmis de nouveau. Si la valeur de ce paramètre est indiquée sans unité, la valeur est supposée être en secondes. Une valeur de 0 (valeur par défaut) utilise la valeur par défaut du système d'exploitation. Ce paramètre est seulement supporté par les systèmes qui supportent le symbole `TCP_KEEPINTVL` ou une option socket équivalente et sur Windows ; sur les autres systèmes, ce paramètre doit valoir zéro. Pour les sessions connectées via une socket de domaine Unix, ce paramètre est ignoré et vaut toujours zéro.

> **Note**
>
> Sur Windows, une valeur de 0 configurera ce paramètre à une seconde car Windows ne fournit pas un moyen de lire la valeur par défaut du système.

`tcp_keepalives_count` (`integer`)

Indique le nombre de messages TCP keepalive pouvant être perdus avant que la connexion au serveur soit considérée comme morte. Une valeur de 0 (valeur par défaut) revient à utiliser la valeur système par défaut. Ce paramètre est seulement supporté par les systèmes qui supportent le symbole `TCP_KEEPCNT` ou une option socket équivalente, et sur Windows ; sur les autres systèmes, ce paramètre doit valoir zéro. Pour les sessions connectées via une socket de domaine Unix, ce paramètre est ignoré et vaut toujours zéro.

> **Note**
>
> Ce paramètre n'est pas supporté sur Windows et doit donc valoir zéro.

`tcp_user_timeout` (`integer`)

Indique la durée pendant laquelle les données transmises peuvent rester sans réponse avec que la connexion TCP ne soit fermée.Si la valeur de ce paramètre est indiquée sans unité, la valeur est supposée être en millisecondes. Une valeur de 0 (valeur par défaut) utilise la valeur par défaut du système d'exploitation. Ce paramètre est uniquement accepté sur les paramètres qui acceptent `TCP_USER_TIMEOUT` ; sur les autres systèmes, il doit valoir zéro. Pour les sessions connectées via un socket de domaine Unix, ce paramètre est ignoré et vaut toujours zéro.

> **Note**
>
> Ce paramètre n'est pas supporté sur Windows et doit valoir zéro.

`client_connection_check_interval` (`integer`)

Permet de passer un intervalle de temps pendant lequel le server vérifie que le client est toujours connecté lorsqu'une requête s'exécute. La vérification est faite en interrogeant le socket et permet aux requêtes longues d'être annulées plus tôt si le noyau indique que la connexion est fermée.

Cette option se base sur les événements noyau exposés par Linux, macOS, illumos et la famille BSD de systèmes d'exploitation, et n'est actuellement pas disponibles sur les autres systèmes.

Si la valeur est passée sans unité, elle est comprise en millisecondes. La valeur par défaut est `0`, ce qui désactive la fonctionnalité. Sans cette fonctionnalité de vérification des connexions, le serveur ne détectera une perte de connexion qu'à la prochaine interaction avec le socket, lorsqu'il attend ou reçoit ou envoie des données.

Pour que le noyau puisse détecter de lui-même une perte de connexion TCP dans un certain délai quel que soit le scénario, y compris des pannes de réseau, il peut être nécessaire d'ajuster le paramètre de keepalive TCP au niveau du système d'exploitation ou les paramètres tcp_keepalives_idle, tcp_keepalives_interval et tcp_keepalives_count de PostgreSQL.

20.3.2. Authentification

`authentication_timeout` (`integer`)

Temps maximum pour terminer l'authentification du client, en secondes. Si un client n'a pas terminé le protocole d'authentification dans ce délai, le serveur ferme la connexion. Cela protège le serveur des clients bloqués occupant une connexion indéfiniment. Si la valeur de ce paramètre est donné sans unité, l'unité sera la seconde. La valeur par défaut est d'une minute. Ce paramètre peut être configuré au lancement du serveur et dans le fichier `postgresql.conf`.

`password_encryption` (`enum`)

Détermine l'algorithme utilisé pour chiffrer un mot de passe spécifié dans CREATE ROLE ou ALTER ROLE. Les valeurs possibles sont `scram-sha-256`, qui permettra un chiffrement du mot de passe avec SCRAM-SHA-256 et `md5`, qui stocke le mot de passe haché en MD5. La valeur par défaut est `scram-sha-256`.

Notez que des clients plus anciens pourraient ne pas disposer du support pour l'authentification SCRAM, et ne fonctionneraient pas avec des mots de passe chiffrés avec SCRAM-SHA-256. Voir Section 21.5 pour les détails.

`krb_server_keyfile` (`string`)

Indique l'emplacement du fichier de clés Kerberos du serveur. La valeur par défaut est `FILE:/usr/local/pgsql/etc/krb5.keytab` (où le chemin correspond à ce qui a été indiqué comme `sysconfdir` lors de la compilation ; utilisez `pg_config --sysconfdir` pour

le déterminer). Si ce paramètre est configuré à une chaîne vide, il est ignoré et une valeur par défaut dépendante du système est utilisée. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur. Voir Section 21.6 pour plus d'informations.

`krb_caseins_users` (`boolean`)

Indique si les noms d'utilisateur GSSAPI doivent être traités en respectant la casse. Le défaut est `off` (sensible à la casse). Ce paramètre ne peut être positionné que dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`db_user_namespace` (`boolean`)

Ce paramètre autorise des noms d'utilisateur par base de données. Il est à `off` par défaut. Ce paramètre ne peut être positionné que dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

S'il est activé, vous devez créer des utilisateurs en tant que *`username@dbname`*. Quand *`username`* est transmis par une connexion cliente, `@` et le nom de la base sont ajoutés au nom d'utilisateur, et ce nom spécifique à la base sera recherché par le serveur. Notez que si vous créez des utilisateurs dont le nom comprend un `@`, vous devrez ajouter des guillemets autour de ce nom.

Même avec ce paramètre activé, vous pouvez toujours créer les utilisateurs globaux ordinaires. Ajoutez simplement `@` en donnant le nom dans le client, comme par exemple `pierre@`. Le `@` sera supprimé avant la recherche de l'utilisateur par le serveur.

`db_user_namespace` provoque un écart entre les représentations des noms du client et du serveur. Les tests d'authentification sont toujours fait avec le nom d'utilisateur du serveur, donc les méthodes d'authentification doivent être configurées avec le nom connu du serveur, pas celui du client. Comme `md5` utilise le nom d'utilisateur comme sel sur le client comme sur le serveur, `md5` ne peut être utilisé avec `db_user_namespace`.

> **Note**
>
> Cette option est considérée comme une mesure provisoire jusqu'à ce qu'une solution complète soit trouvée. À ce moment, cette option sera supprimée.

20.3.3. SSL

Voir Section 19.9 pour plus d'informations sur la mise en œuvre de SSL. Les paramètres de configuration pour le contrôle du chiffrement de transfert utilisant les protocoles TLS sont nommés `ssl` pour des raisons historiques, même si le support du protocole SSL est considéré comme obsolète. Dans ce contexte, SSL est utilisé de façon interchangeable avec TLS.

`ssl` (`boolean`)

Active les connexions SSL. Ce paramètre peut uniquement être modifié dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. La valeur par défaut est `off`.

`ssl_ciphers` (`string`)

Donne une liste d'algorithmes SSL autorisées à être utilisés sur des connexions SSL. Voir la page de manuel de ciphers dans le paquet OpenSSL pour la syntaxe de ce paramètre et une liste des valeurs supportées. Seules les connexions utilisant TLS version 1.2 et antérieures sont impactées. Il n'existe actuellement pas de paramètre contrôlant le choix des algorithmes utilisés par les connexions avec TLS version 1.3. La valeur par défaut est `HIGH:MEDIUM:+3DES:!aNULL`. Cette valeur est généralement raisonnable, sauf si vous avez des besoins spécifiques en terme de sécurité.

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

Voici une explication de la valeur par défaut ;:

`HIGH`

Algorithmes du groupe `HIGH` (par exemple AES, Camellia, 3DES)

`MEDIUM`

Algorithmes du groupe `MEDIUM` (par exemple RC4, SEED)

`+3DES`

L'ordre par défaut d'OpenSSL pour `HIGH` est problématique car il positionne 3DES avant AES128. Ceci est mauvais parce que 3DES offre moins de sécurité que AES128, et il est aussi bien moins rapide. `+3DES` le réordonne après les algorithmes des groupes `HIGH` et `MEDIUM`.

`!aNULL`

Désactive les algorithmes anonymes qui ne font pas d'authentification. Ces algorithmes sont vulnérables à des attaques de type *man-in-the-middle* (MITM) et ne doivent donc pas être utilisés.

Les détails sur les algorithmes varient suivant les versions d'OpenSSL. Utiliser la commande `openssl ciphers -v 'HIGH:MEDIUM:+3DES:!aNULL'` pour voir les détails réels de la version OpenSSL actuellement installée. Notez que cette liste est filtrée à l'exécution suivant le type de clé du serveur.

`ssl_prefer_server_ciphers` (`boolean`)

Précise s'il faut utiliser les préférences du serveur en terme d'algorithmes, ou celles du client. Ce paramètre peut uniquement être modifié dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. La valeur par défaut est `on`.

Les versions de PostgreSQL antérieures à la 9.4 n'ont pas ce paramètre et utilisent toujours les préférences du client. Ce paramètre a principalement pour but de maintenir une compatibilité ascendante avec ces versions. Utiliser les préférences du serveur est généralement conseillé car il est plus probable que le serveur soit correctement configuré.

`ssl_ecdh_curve` (`string`)

Indique le nom de la courve à utiliser dans l'échange de clés ECDH. Elle doit être acceptée par tous les clients qui se connectent. Il n'est pas nécessaire que la même courbe soit utilisée par la clé Elliptic Curve. Ce paramètre peut uniquement être modifié dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. La valeur par défaut est `prime256v1`.

Noms OpenSSL pour les courbes les plus courantes : `prime256v1` (NIST P-256), `secp384r1` (NIST P-384), `secp521r1` (NIST P-521). La liste complète des courbes disponibles peut être récupérée avec la commande `openssl ecparam -list_curves`. Toutes ne sont pas utilisables dans TLS.

`ssl_ca_file` (`string`)

Indique le nom du fichier contenant l'autorité du certificat serveur SSL (CA). Les chemins relatifs sont relatifs par rapport au répertoire de données. Ce paramètre peut uniquement être modifié dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Par défaut le paramètre est vide, ce qui veut dire qu'il n'y a pas de fichier d'autorité du certificat chargé, et donc que la vérification du certificat client n'est pas effectuée.

`ssl_cert_file` (`string`)

Indique le nom du fichier contenant le certificat SSL du serveur. Les chemins relatifs sont relatifs par rapport au répertoire de données. Ce paramètre peut uniquement être modifié dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. La valeur par défaut est `server.crt`.

`ssl_crl_file` (`string`)

Indique le nom du fichier contenant la liste de révocation de certificat SSL client (CRL). Les chemins relatifs sont relatifs par rapport au répertoire de données. Ce paramètre peut uniquement être modifié dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Par défaut, ce paramètre est vide, ce qui veut dire qu'aucune liste de révocation de certificat n'est chargée. (unless ssl_crl_dir is set).

`ssl_crl_dir` (`string`)

Indique le nom du répertoire qui contient la liste de révocation de certificats du serveur (CRL). Les chemins relatifs sont pris à partir du répertoire de données. Ce paramètre peut seulement être indiqué dans le fichier `postgresql.conf` ou sur la ligne de commande. La valeur par défaut est vide, signifiant qu'aucun CRL n'est utilisé, à moins que le paramètre ssl_crl_file ne soit renseigné.

Le répertoire doit être préparé avec la commande OpenSSL `openssl rehash` ou `c_rehash`. Voir la documentation de OpenSSL pour plus d'informations.

Quand ce paramètre est utilisé, les CRL du répertoire spécifié sont chargés à la demande au moment de la connexion. Les nouvelles CRL qui sont ajoutées dans le répertoire pourront être utilisées immédiatement, contrairement au paramètre ssl_crl_file, avec lequel les CRL sont chargées au démarrage du serveur ou au rechargement de la configuration. Les deux paramètres peuvent être utilisés simultanément.

`ssl_key_file` (`string`)

Indique le nom du fichier contenant la clé privée SSL du serveur. Les chemins relatifs sont relatifs par rapport au répertoire de données. Ce paramètre peut uniquement être modifié dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. La valeur par défaut est `server.key`.

`ssl_min_protocol_version` (`enum`)

Configure la version minimale du protocole SSL/TLS à utiliser. Les valeurs valides sont actuellement : `TLSv1`, `TLSv1.1`, `TLSv1.2`, `TLSv1.3`. Les versions plus anciennes de la bibliothèque OpenSSL n'acceptent pas toutes les valeurs ; une erreur peut survenir si une configuration non supportée est choisie. Les versions du protocole avant TLS 1.0, donc SSL version 2 et 3, sont toujours désactivées.

La valeur par défaut est `TLSv1.2`, ce qui satisfait les bonnes pratiques au moment où ceci est écrit.

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`ssl_max_protocol_version` (`enum`)

Configure la version maximale du protocole SSL/TLS à utiliser. Les valeurs valides sont les mêmes que pour ssl_min_protocol_version, avec la possibilité d'accepter une chaîne vide, qui permet toute version du protocole. Le comportement par défaut est d'autoriser toute version. Configurer la version maximale du protocole est principalement utile pour des tests ou si certains composants ont des problèmes pour travailler avec un protocole plus récent.

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`ssl_dh_params_file` (`string`)

Indique le nom du fichier contenant les paramètres Diffie-Hellman utilisés pour la famille DH éphémère des algorithmes SSL. La valeur par défaut est une chaîne vide, auquel cas les paramètres DH par défaut sont utilisés. Utiliser des paramètres DH personnalisés réduit l'exposition si un attaquant réussit à craquer les paramètres DH internes bien connus. Vous pouvez créer votre propre fichier de paramètre DH avec la commande `openssl dhparam -out dhparams.pem 2048`.

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`ssl_passphrase_command` (`string`)

Indique une commande externe à appeler quand il faut obtenir une phrase de passe pour déchiffrer un fichier SSL tel que la clé privée. Par défaut, ce paramètre est vide, ce qui implique que le mécanisme interne de demande de cette phrase de passe sera utilisé.

La commande doit envoyer la phrase de passe dans sa sortie standard et quitter avec le code 0. Dans la valeur du paramètre, `%p` est remplacée par la chaîne d'interrogation. (Écrivez `%%` pour afficher un %.) Notez que la chaîne d'interrogation contiendra probablement des espaces, donc mettez les guillemets adéquats. S'il y a une nouvelle ligne unique à la fin de la sortie, elle sera supprimée.

En fait, la commande n'a pas besoin de demander une phrase de passe à l'utilisateur. Elle peut la lire depuis un fichier, l'obtenir d'un trousseau ou de quelque chose de ce genre. C'est à l'utilisateur de s'assurer que ce mécanisme est sûr.

Ce paramètre ne peut être renseigné que dans le fichier `postgresql.conf` ou sur la ligne de commande.

`ssl_passphrase_command_supports_reload` (`boolean`)

Ce paramètre indique si la commande pour la phrase de passe désignée dans `ssl_passphrase_command` doit aussi être appelée lors d'un rechargement de configuration, si le fichier clé veut une phrase de passe. Si ce paramètre est `off` (c'est le défaut), alors `ssl_passphrase_command` sera ignoré lors d'un rechargement et la configuration SSL ne sera pas rechargée si une phrase de passe était nécessaire. C'est la configuration correcte si la commande a besoin d'un TTY pour demander la phrase, qui pourrait ne pas être disponible quand le serveur fonctionne. Passer ce paramètre à `on` peut être approprié si la phrase de passe est obtenue, par exemple, depuis un fichier.

Ce paramètre doit être configuré à `on` lors d'une exécution sous Windows car toutes les connexions réaliseront un rechargement de la configuration à cause du modèle différent de processus de cette plateforme.

Ce paramètre ne peut être renseigné que dans le fichier `postgresql.conf` ou sur la ligne de commande.

20.4. Consommation des ressources

20.4.1. Mémoire

`shared_buffers` (`integer`)

Initialise la quantité de mémoire que le serveur de bases de données utilise comme mémoire partagée. La valeur par défaut, en général 128 Mo, peut être automatiquement abaissée si la configuration du noyau ne la supporte pas (déterminé lors de l'exécution de l'initdb). Ce paramètre

doit être au minimum de 128 ko + 16 ko par max_connections. Des valeurs significativement plus importantes que ce minimum sont généralement nécessaires pour de bonnes performances. Si cette valeur est spécifiée sans unité, elle sera interprétée comme un nombre de blocs, autrement dit BLCKSZ octets, typiquement 8 Ko. (Une valeur personnalisée de BLCKSZ changera la valeur minimale.) Ce paramètre ne peut être configuré qu'au lancement du serveur.

Si vous disposez d'un serveur dédié à la base de données, avec 1 Go de mémoire ou plus, une valeur de départ raisonnable pour ce paramètre est de 25% la mémoire de votre système. Certains cas peuvent nécessiter une valeur encore plus importante pour le shared_buffers mais comme PostgreSQL profite aussi du cache du système d'exploitation, il est peu probable qu'une allocation de plus de 40% de la mémoire fonctionnera mieux qu'une valeur plus restreinte. Des valeurs importantes pour le paramètre shared_buffers requièrent généralement une augmentation proportionnelle du max_wal_size, pour étendre dans le temps les écritures de grandes quantités de données, nouvelles ou modifiées.

Sur des systèmes comprenant moins d'1 Go de mémoire, un pourcentage plus restreint est approprié pour laisser une place suffisante au système d'exploitation.

huge_pages (enum)

Contrôle si les *huge pages* sont obligatoires pour la principale zone de mémoire partagée. Les valeurs valides sont try (le défaut), on et off. Ce paramètre est seulement configurable au démarrage du serveur. Avec huge_pages à try, le serveur tentera de demander des *huge pages* mais se rabattra sur le défaut en cas d'échec. Avec on, cet échec empêchera le serveur de démarrer. Avec off, il n'y aura pas de demande de *huge pages*.

Pour le moment, ce paramètre n'est supporté que sur Linux et Windows. Il est ignoré sur les autres systèmes quand il est à try. Sur Linux, il est seulement supporté quand shared_memory_type est configuré à mmap (la valeur par défaut).

L'utilisation des *huge pages* réduit la taille des tables de pages et la consommation CPU pour gérer la mémoire, améliorant ainsi les performances. Pour plus de détails sur la gestion des *huge pages* sur Linux, voir Section 19.4.5.

Sous Windows, les *huge pages* sont connues sous le nom de *large pages*. Pour les utiliser, vous devez assigner le droit « Verrouiller les pages en mémoire » (*Lock Pages in Memory*) à l'utilisateur Windows qui fait tourner PostgreSQL. Vous pouvez utiliser l'Éditeur de stratégie de groupe locale (gpedit.msc) pour assigner ce droit à l'utilisateur. Pour démarrer le serveur en ligne de commande en tant que processus autonome, et pas en tant que service Windows, l'invite de commande doit tourner en tant qu'administrateur, ou bien le contrôle de compte d'utilisateur (UAC, *User Access Control*) doit être désactivé. Quand l'UAC est activé, l'invite de commande normale révoque le droit « Verrouiller les pages en mémoire » de l'utilisateur au démarrage.

Notez que ce paramètre n'affecte que la partie principale de la mémoire partagée. Des systèmes d'exploitation comme Linux, FreeBSD et Illumos peuvent aussi utiliser les *huge pages* automatiquement pour les allocations mémoire normales sans demande explicite de PostgreSQL. Sur Linux, cela s'appelle « transparent huge pages » (THP). Elles sont connues pour causer une dégradation des performances avec PostgreSQL pour certains utilisateurs sur certaines versions de Linux ; leur usage est donc actuellement déconseillé (au contraire de l'utilisation explicite de huge_pages).

huge_page_size (integer)

Contrôle la taille des *huge pages*, quand elles sont activées avec huge_pages. La valeur par défaut est zéro (0). Si initialisé à 0, la valeur par défaut du système pour la taille des *huge pages* est utilisée.

Certaines valeurs communes de la taille des pages sur les serveurs à architecture 64 bits sont : 2MB et 1GB (Intel et AMD), 16MB et 16GB (IBM POWER), et 64kB, 2MB, 32MB et 1GB (ARM). Pour plus d'informations, voir Section 19.4.5.

L'utilisation de valeurs différentes des valeurs par défaut n'est actuellement supportée que sous Linux.

`temp_buffers` (`integer`)

Configure la quantité maximale de mémoire utilisée pour le cache des objets temporaires à l'intérieur de chaque session à la base. Ce sont des caches locaux à la session utilisés uniquement pour l'accès aux tables temporaires. Si cette valeur est spécifiée sans unité, elle est interprétée comme un nombre de blocs, de `BLCKSZ` octets, typiquement 8 Ko. La valeur par défaut est de 8 Mo (`8MB`). (Si `BLCKSZ` ne vaut pas 8 Ko, la valeur changera de façon proportionnée.) Ce paramètre peut être modifié à l'intérieur de sessions individuelles mais seulement jusqu'à la première utilisation des tables temporaires dans une session ; les tentatives suivantes de changement de cette valeur n'ont aucun effet sur cette session.

Une session alloue des tampons temporaires en fonction des besoins jusqu'à atteindre la limite donnée par `temp_buffers`. Positionner une valeur importante pour les sessions qui ne le nécessitent pas ne coûte qu'un descripteur de tampon, soit environ 64 octets, par incrément de `temp_buffers`. Néanmoins, si un tampon est réellement utilisé, 8192 autres octets sont consommés pour celui-ci (ou, plus généralement, `BLCKSZ` octets).

`max_prepared_transactions` (`integer`)

Configure le nombre maximum de transactions simultanément dans l'état « préparées » (voir PREPARE TRANSACTION). Zéro, la configuration par défaut, désactive la fonctionnalité des transactions préparées Ce paramètre ne peut être configuré qu'au lancement du serveur.

Si vous ne prévoyez pas d'utiliser les transactions préparées, ce paramètre devrait être positionné à zéro pour éviter toute création accidentelle de transactions préparées. Au contraire, si vous les utilisez, il peut être intéressant de positionner `max_prepared_transactions` au minimum à au moins max_connections pour que chaque session puisse avoir sa transaction préparée.

Lors de l'exécution d'un serveur en attente, vous devez configurer ce paramètre à la même valeur ou à une valeur plus importante que sur le serveur primaire. Sinon, des requêtes pourraient ne pas être autorisées sur le serveur en attente.

`work_mem` (`integer`)

Indique la quantité de mémoire maximale de base à utiliser pour l'exécution d'une requête (tel qu'un tri ou une table de hachage) avant d'écrire dans des fichiers temporaires sur disque. Si cette valeur est indiquée sans unité, elle est considérée être en Ko. La valeur par défaut est de 4 Mo (`4MB`). Une requête complexe peut réaliser plusieurs opérations de tri ou de hachage exécutées en même temps, chaque opération étant autorisée à utiliser autant de mémoire que cette valeur indique avant de commencer à écrire les données dans des fichiers temporaires. De plus, de nombreuses sessions peuvent exécuter de telles opérations simultanément. La mémoire totale utilisée peut, de ce fait, atteindre plusieurs fois la valeur de `work_mem` ; il est nécessaire de garder cela à l'esprit lors du choix de cette valeur. Les opérations de tri sont utilisées pour `ORDER BY`, `DISTINCT` et les jointures de fusion. Les tables de hachage sont utilisées dans les jointures de hachage, les agrégations, les nœuds Memoize et le traitement des sous-requêtes `IN` fondés sur le hachage.

Les opérations basées sur le hachage sont généralement plus sensibles à la disponibilité de la mémoire que leur équivalent basé sur le tri. La mémoire disponible pour les tables de hachages est calculée en multipliant `work_mem` par `hash_mem_multiplier`. Cela rend possible pour les opérations de hachage d'utiliser une quantité de mémoire qui dépasse la quantité de base proposée par `work_mem`.

`hash_mem_multiplier` (`floating point`)

Utilisé pour calculer la quantité maximale de mémoire que les opérations basées sur le hachage peuvent utiliser. La limite finale est déterminée en multipliant `work_mem` par

hash_mem_multiplier. La valeur par défaut est 2.0, ce qui rend les opérations de hachage sujettes au double de la valeur de work_mem pour les opérations basées sur le tri.

Pensez à augmenter hash_mem_multiplier dans les environnements où l'utilisation de fichiers temporaires survient fréquemment, tout spécialement quand augmenter uniquement work_mem a pour résultat une pression mémoire trop importante (la pression mémoire prend typiquement la forme d'erreurs pour manque de mémoire). La configuration par défaut de 2.0 est souvent efficace avec des charges de travail variées. Des configurations plus hautes (entre 2.0 et 8.0, voire encore plus) pourraient être plus efficaces dans les environnements où work_mem a déjà été augmenté à 40 Moi, voire plus.

maintenance_work_mem (integer)

Indique la quantité maximale de mémoire que peuvent utiliser les opérations de maintenance telles que VACUUM, CREATE INDEX et ALTER TABLE ADD FOREIGN KEY. Si cette valeur est indiquée sans unité, elle est considérée être en Ko. La valeur par défaut est de 64 Mo. Puisque seule une de ces opérations peut être exécutée à la fois dans une session et que, dans le cadre d'un fonctionnement normal, peu d'opérations de ce genre sont exécutées concurrentiellement sur une même installation, il est possible d'initialiser cette variable à une valeur bien plus importante que work_mem. Une grande valeur peut améliorer les performances des opérations VACUUM et de la restauration des sauvegardes.

Quand autovacuum fonctionne, un maximum de autovacuum_max_workers fois cette quantité de mémoire peut être utilisée. Il convient donc de s'assurer de ne pas configurer la valeur par défaut de façon trop importante. Il pourrait être utile de contrôler ceci en configurant autovacuum_work_mem séparément.

Notez que pour la récupération des identifiants de lignes mortes, VACUUM est capable d'utiliser uniquement 1 Go de mémoire.

autovacuum_work_mem (integer)

Indique la quantité maximale de mémoire à utiliser pour chaque processus autovacuum worker. Si cette valeur est indiquée sans unité, elle est considérée être en Ko. Ce paramètre vaut -1 par défaut, indiquant que la valeur de maintenance_work_mem doit être utilisée à la place. Ce paramétrage n'a pas d'effet sur le comportement de VACUUM lorsqu'il est exécuté dans d'autres contextes. Ce paramètre peut seulement être configuré dans le fichier postgresql.conf ou sur la ligne de commande du serveur.

Pour la récupération des identifiants de lignes mortes, autovacuum est capable d'utiliser au maximum 1 Go de mémoire, donc configurer autovacuum_work_mem à une valeur supérieure n'a aucun effet sur le nombre de lignes mortes que l'autovacuum peut récupérer lors du parcours d'une table.

logical_decoding_work_mem (integer)

Indique la quantité de mémoire à utiliser pour le décodage logique, avant que certaines données décodées ne soient écrites sur les disques locaux. Ceci limite la quantité de mémoire utilisée par la réplication logique en flux. Elle vaut par défaut 64 Mo (64MB). Comme chaque connexion de réplication utilise un seul tampon de cette taille et qu'une installation ne peut avoir normalement beaucoup de connexions simultanées de ce type (étant donné qu'elles sont limitées par max_wal_senders), il est plus sûr de configurer cette valeur bien plus haute que work_mem, réduisant la quantité de données décodées écrites sur disque.

max_stack_depth (integer)

Indique la profondeur maximale de la pile d'exécution du serveur. La configuration idéale pour ce paramètre est la limite réelle de la pile assurée par le noyau (configurée par ulimit -s ou équivalent local) à laquelle est soustraite une marge de sécurité d'un Mo environ. La marge de sécurité est nécessaire parce que la profondeur de la pile n'est pas vérifiée dans chaque routine

du serveur mais uniquement dans les routines clés potentiellement récursives. Si cette valeur est indiquée sans unité, elle est considérée être en Ko. Le paramétrage par défaut est de 2 Mo, valeur faible qui implique peu de risques. Néanmoins, elle peut s'avérer trop petite pour autoriser l'exécution de fonctions complexes. Seuls les superutilisateurs et les utilisateurs disposant des droits SET appropriés peuvent modifier ce paramètre.

Configurer ce paramètre à une valeur plus importante que la limite réelle du noyau signifie qu'une fonction récursive peut occasionner un arrêt brutal d'un processus serveur particulier. Sur les plateformes où PostgreSQL peut déterminer la limite du noyau, il interdit de positionner cette variable à une valeur inadéquate. Néanmoins, toutes les plateformes ne fournissent pas cette information, et une grande attention doit être portée au choix de cette valeur.

`shared_memory_type` (`enum`)

Indique l'implémentation de mémoire partagée que le serveur doit utiliser pour la principale région de mémoire partagée contenant le cache disque de PostgreSQL et d'autres données partagées. Les valeurs possibles sont `mmap` (pour la mémoire partagée anonyme allouée en utilisant `mmap`), `sysv` (pour la mémoire partagée System V allouée via `shmget`) et `windows` (pour la mémoire partagée Windows). Toutes les valeurs ne sont pas forcément supportées sur toutes les plateformes ; la première option supportée est la valeur par défaut pour cette plateforme. L'utilisation de l'option `sysv`, qui n'est pas la valeur par défaut quelque soit la plateforme, est généralement non conseillée car elle nécessite habituellement une configuration du noyau différente de la configuration par défaut pour permettre de grosses allocations (voir Section 19.4.1). Ce paramètre est seulement configurable au démarrage du serveur.

`dynamic_shared_memory_type` (`enum`)

Indique l'implémentation de mémoire partagée dynamique que le serveur doit utiliser. Les valeurs possibles sont `posix` (pour la mémoire partagée POSIX allouée en utilisant `shm_open`), `sysv` (pour la mémoire partagée System V allouée en* utilisant `shmget`), `windows` (pour la mémoire partagée Windows), `mmap` (pour simuler la mémoire partagée en utilisant les fichiers de mémoire enregistrés dans le répertoire des données). Toutes les valeurs ne sont pas forcément supportées sur toutes les plateformes ; la première option supportée est la valeur par défaut pour cette plateforme. L'utilisation de l'option `mmap`, qui n'est la valeur par défaut d'aucune plateforme, est généralement déconseillée car le système d'exploitation pourrait écrire des pages modifiées sur disque de manière répétée, augmentant la charge disque du système. Néanmoins, cela peut se révéler utile pour débugger, quand le répertoire `pg_dynshmem` est stocké dans un disque RAM ou quand les autres options de mémoire partagée ne sont pas disponibles. Ce paramètre est seulement configurable au démarrage du serveur.

`min_dynamic_shared_memory` (`integer`)

Indique la quantité de mémoire qui doit être allouée au démarrage du serveur pour l'execution en parallèle de requêtes. Quand cette partie de la mémoire est insuffisante pour les requêtes concurrentes, de nouvelles requêtes parallèles essayent d'allouer temporairement plus de mémoire partagée en utilisant la méthode configurée avec `dynamic_shared_memory_type`, ce qui peut être plus lent à cause de la consommation de ressources de la gestion de la mémoire. La mémoire allouée au démarrage avec `min_dynamic_shared_memory` est affectée par le paramètre `huge_pages` au niveau du système d'exploitation lorsque c'est supporté et peut bénéficier de plus grandes pages sur les systèmes d'exploitation où c'est géré automatiquement. La valeur par défaut est `0` (aucune).

20.4.2. Disque

`temp_file_limit` (`integer`)

Spécifie la quantité maximale d'espace disque qu'un processus peut utiliser pour les fichiers temporaires, comme par exemple ceux utilisés pour les tris et hachages, ou le fichier de stockage pour un curseur détenu. Une transaction tentant de dépasser cette limite sera annulée. La valeur

a pour unité le ko. La valeur spéciale `-1` (valeur par défaut) signifie sans limite. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` appropriés peuvent modifier cette configuration.

Ce paramètre contraint l'espace total utilisé à tout instant par tous les fichiers temporaires utilisés pour un processus PostgreSQL donnée. Il doit être noté que l'espace disque utilisé pour les tables temporaires explicites, à l'opposé des fichiers temporaires utilisés implicitement pour l'exécution des requêtes, n'est *pas* pris en compte pour cette limite.

20.4.3. Usage des ressources du noyau

`max_files_per_process` (`integer`)

Positionne le nombre maximum de fichiers simultanément ouverts par sous-processus serveur. La valeur par défaut est de 1000 fichiers. Si le noyau assure une limite par processus, il n'est pas nécessaire de s'intéresser à ce paramètre. Toutefois, sur certaines plateformes (notamment les systèmes BSD) le noyau autorise les processus individuels à ouvrir plus de fichiers que le système ne peut effectivement en supporter lorsqu'un grand nombre de processus essayent tous d'ouvrir ce nombre de fichiers. Si le message « Too many open files » (« Trop de fichiers ouverts ») apparaît, il faut essayer de réduire ce paramètre. Ce paramètre ne peut être configuré qu'au lancement du serveur.

20.4.4. Report du VACUUM en fonction de son coût

Lors de l'exécution des commandes VACUUM et ANALYZE, le système maintient un compteur interne qui conserve la trace du coût estimé des différentes opérations d'entrée/sortie réalisées. Quand le coût accumulé atteint une limite (indiquée par `vacuum_cost_limit`), le processus traitant l'opération s'arrête un court moment (précisé par `vacuum_cost_delay`). Puis, il réinitialise le compteur et continue l'exécution.

Le but de cette fonctionnalité est d'autoriser les administrateurs à réduire l'impact des entrées/sorties de ces commandes en fonction de l'activité des bases de données. Nombreuses sont les situations pour lesquelles il n'est pas très important que les commandes de maintenance telles que `VACUUM` et `ANALYZE` se finissent rapidement, mais il est généralement très important que ces commandes n'interfèrent pas de façon significative avec la capacité du système à réaliser d'autres opérations sur les bases de données. Le report du VACUUM en fonction de son coût fournit aux administrateurs un moyen d'y parvenir.

Cette fonctionnalité est désactivée par défaut pour les commandes `VACUUM` lancées manuellement. Pour l'activer, la variable `vacuum_cost_delay` doit être initialisée à une valeur différente de zéro.

`vacuum_cost_delay` (`floating point`)

Indique le temps, en millisecondes, de repos du processus quand la limite de coût a été atteinte. Si cette valeur est indiquée sans unité, elle est comprise comme des millisecondes. La valeur par défaut est zéro, ce qui désactive la fonctionnalité de report du VACUUM en fonction de son coût. Une valeur positive active cette fonctionnalité.

Lors d'utilisation de vacuum basée sur le coût, les valeurs appropriées pour `vacuum_cost_delay` sont habituellement assez petites, inférieures à 1 milliseconde. Bien que `vacuum_cost_delay` puisse être configuré à des valeurs à virgule flottante en millisecondes, ces délais pourraient ne pas pouvoir être mesurés de façon précise sur les anciennes plateformes. Sur ce type de plateforme, augmenter la consommation de ressource du `VACUUM` au-dessus de ce que vous obtiendrez avec 1ms nécessitera de modifier les autres paramètres de coût. De plus, vous devrez garder `vacuum_cost_delay` aussi petit que ce que votre plateforme peut mesurer correctement. De gros délais n'aident pas.

`vacuum_cost_page_hit` (`integer`)

Indique Le coût estimé du nettoyage par VACUUM d'un tampon trouvé dans le cache des tampons partagés. Cela représente le coût de verrouillage de la réserve de tampons, la recherche au sein de la table de hachage partagée et le parcours du contenu de la page. La valeur par défaut est 1.

`vacuum_cost_page_miss` (`integer`)

Indique le coût estimé du nettoyage par VACUUM d'un tampon qui doit être lu sur le disque. Cela représente l'effort à fournir pour verrouiller la réserve de tampons, rechercher dans la table de hachage partagée, lire le bloc désiré sur le disque et parcourir son contenu. La valeur par défaut est 2.

`vacuum_cost_page_dirty` (`integer`)

Indique le coût estimé de modification par VACUUM d'un bloc précédemment vide (*clean block*). Cela représente les entrées/sorties supplémentaires nécessaires pour vider à nouveau le bloc modifié (*dirty block*) sur le disque. La valeur par défaut est 20.

`vacuum_cost_limit` (`integer`)

Indique Le coût cumulé qui provoque l'endormissement du processus de VACUUM. La valeur par défaut est 200.

> **Note**
>
> Certaines opérations détiennent des verrous critiques et doivent donc se terminer le plus vite possible. Les reports de VACUUM en fonction du coût ne surviennent pas pendant ces opérations. De ce fait, il est possible que le coût cumulé soit bien plus important que la limite indiquée. Pour éviter des délais inutilement longs dans de tels cas, le délai réel est calculé de la façon suivante : `vacuum_cost_delay` * `accumulated_balance` / `vacuum_cost_limit` avec un maximum de `vacuum_cost_delay` * 4.

20.4.5. Processus d'écriture en arrière-plan

Il existe un processus serveur séparé appelé *background writer* dont le but est d'écrire les tampons « sales » (parce que nouveaux ou modifiés). Quand le nombre de tampons partagés propres semble insuffisant, le *background writer* écrit quelques tampons sales au système de fichiers et les marque comme étant propres. Ceci réduit la probabilité que les processus serveur gérant les requêtes des utilisateurs ne soient dans l'incapacité de trouver des tampons propres et doivent écrire eux-mêmes des tampons sales. Néanmoins, ce processus d'écriture en tâche de fond implique une augmentation globale de la charge des entrées/sorties disque car, quand une page fréquemment modifiée pourrait n'être écrite qu'une seule fois par CHECKPOINT, le processus d'écriture en tâche de fond pourrait l'avoir écrit plusieurs fois si cette page a été modifiée plusieurs fois dans le même intervalle. Les paramètres discutés dans cette sous-section peuvent être utilisés pour configurer finement son comportement pour les besoins locaux.

`bgwriter_delay` (`integer`)

Indique le délai entre les tours d'activité du processus d'écriture en arrière-plan. À chaque tour, le processus écrit un certain nombre de tampons modifiés (contrôlable par les paramètres qui suivent). Puis, il s'endort pour `bgwriter_delay` millisecondes et recommence. Quand il n'y a pas de tampons modifiés dans le cache, il s'endort plus profondément sans considération du `bgwriter_delay`. Si cette valeur est indiquée sans unité, elle est prise comme des millisecondes. La valeur par défaut est de 200 millisecondes. Sur de nombreux systèmes, la résolution réelle du `sleep` est de 10 millisecondes ; positionner `bgwriter_delay` à une valeur qui n'est pas un multiple de 10 peut avoir le même résultat que de le positionner au multiple de 10 supérieur. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`bgwriter_lru_maxpages` (`integer`)

Nombre maximum de tampons qui peuvent être écrits à chaque tour par le processus d'écriture en tâche de fond. Le configurer à zéro désactive l'écriture en tâche de fond. (Notez que les checkpoints ne sont pas affectés. Ils sont gérés par un autre processus, dédié à cette tâche.) La valeur par défaut est de 100 tampons. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`bgwriter_lru_multiplier` (`floating point`)

Le nombre de tampons sales écrits à chaque tour est basé sur le nombre de nouveaux tampons qui ont été requis par les processus serveur lors des derniers tours. Le besoin récent moyen est multiplié par `bgwriter_lru_multiplier` pour arriver à une estimation du nombre de tampons nécessaire au prochain tour. Les tampons sales sont écrits pour qu'il y ait ce nombre de tampons propres, réutilisables. (Néanmoins, au maximum `bgwriter_lru_maxpages` tampons sont écrits par tour.) De ce fait, une configuration de 1.0 représente une politique d'écriture « juste à temps » d'exactement le nombre de tampons prédits. Des valeurs plus importantes fournissent une protection contre les pics de demande, alors qu'une valeur plus petite laisse intentionnellement des écritures aux processus serveur. La valeur par défaut est de 2. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`bgwriter_flush_after` (`integer`)

Quand plus de ce nombre d'octets ont été écrit par le processus d'écriture en tâche de fond (bgwriter), tente de forcer le système d'exploitation à écrire les données sur disque. Faire cela limite la quantité de données modifiées dans le cache disque du noyau, réduisant le risque de petites pauses dues à l'exécution d'un `fsync` à la fin d'un checkpoint ou à l'écriture massive en tâche de fond des données modifiées. Souvent, cela réduira fortement la latence des transactions mais il existe aussi quelques cas de dégradation des performances, tout spécialement avec les charges de travail plus importantes que shared_buffers, mais plus petites que le cache disque du système d'exploitation. Ce paramètre pourrait ne pas avoir d'effet sur certaines plateformes. L'intervalle valide se situe entre `0`, qui désactive le « writeback » forcé, et `2MB`. Si cette valeur est indiquée sans unité, elle est compris comme des blocs, autrement dit `BLCKSZ` octets, typiquement 8 Ko. La valeur par défaut est `512KB` sur Linux, `0` ailleurs. (Si `BLCKSZ` ne vaut pas 8 ko, les valeurs par défaut et maximales évoluent de façon proportionnelles à cette constante.) Ce paramètre est seulement configurable dans le fichier `postgresql.conf` et à la ligne de commande.

Des valeurs plus faibles de `bgwriter_lru_maxpages` et `bgwriter_lru_multiplier` réduisent la charge supplémentaire des entrées/sorties induite par le processus d'écriture en arrière-plan. En contrepartie, la probabilité que les processus serveurs effectuent plus d'écritures par eux-mêmes augmente, ce qui retarde les requêtes interactives.

20.4.6. Comportement asynchrone

`backend_flush_after` (`integer`)

Lorsqu'une quantité de données supérieure à cette valeur doit être écrite par un seul processus, une tentative de forcer le système d'exploitation à écrire sur le stockage sous-jacent est faite. En agissant ainsi, on limite la quantité de données modifiée dans le cache du noyau, réduisant ainsi la probabilité d'une perte de temps quand la fonction `fsync` est appelée à la fin du checkpoint ou lorsque le système d'exploitation écrit beaucoup de données en arrière-plan. Souvent, cela permet de réduire grandement la latence des transactions mais il est possible dans certains cas, en particulier pour les volumes de donnés plus importants que shared_buffers mais plus petits qu'un bloc de cache du système d'exploitation, que les performances en soient dégradées. Ce paramètre peut aussi n'avoir aucun effet sur certaines plateformes. Si la valeur est indiquée sans unité, elle est comprise en blocs, c'est-à-dire en `BLCKSZ` octets, généralement 8 ko. Les valeurs doivent être comprises entre `0`, ce qui désactive la fonctionnalité d'écritures forcées et `2Mo`. La

valeur par défaut est `0`. (Si `BLCKSZ` n'est pas configuré à 8 ko, la valeur maximale est ajustée proportionnellement.)

`effective_io_concurrency` (`integer`)

Positionne le nombre d'opérations d'entrées/sorties disque concurrentes que PostgreSQL pense pouvoir exécuter simultanément. Augmenter cette valeur va augmenter le nombre d'opérations d'entrée/sortie que chaque session PostgreSQL individuelle essayera d'exécuter en parallèle. Les valeurs autorisées vont de 1 à 1000, ou zéro pour désactiver l'exécution de requêtes d'entrée/sortie asynchrones. Actuellement, ce paramètre ne concerne que les parcours de type *bitmap heap*.

Pour les disques magnétiques, un bon point départ pour ce paramètre est le nombre de disques que comprend un agrégat par bande RAID 0 ou miroir RAID 1 utilisé pour la base de données. (Pour du RAID 5, le disque de parité ne devrait pas être pris en compte.) Toutefois, si la base est souvent occupée par de nombreuses requêtes exécutées dans des sessions concurrentes, des valeurs plus basses peuvent être suffisantes pour maintenir le groupe de disques occupé. Une valeur plus élevée que nécessaire pour maintenir les disques occupés n'aura comme seul résultat que de surcharger le processeur. Les SSD et autres méthodes de stockage basées sur de la mémoire peuvent souvent traiter un grand nombre de demandes concurrentes, donc la meilleure valeur pourrait être dans les centaines.

Les entrées/sorties asynchrones dépendent de la présence d'une fonction `posix_fadvise` efficace, ce que n'ont pas certains systèmes d'exploitation. Si la fonction n'est pas présente, alors positionner ce paramètre à une valeur autre que zéro entraînera une erreur. Sur certains systèmes (par exemple Solaris), cette fonction est présente mais n'a pas d'effet.

La valeur par défaut est 1 sur les systèmes supportés, et 0 pour les autres. Cette valeur peut être surchargée pour les tables d'un tablespace particulier en configuration le paramètre tablespace du même nom (voir ALTER TABLESPACE).

`maintenance_io_concurrency` (`integer`)

Similaire à `effective_io_concurrency`, mais utilisé pour le travail de maintenance réaliser pour les nombreuses sessions clientes.

La valeur par défaut est 10 sur les systèmes supportés, 0 sinon. Cette valeur peut être surchargée pour les tables d'un tablespace particulier en configurant le paramètre de même nom pour ce tablespace (voir ALTER TABLESPACE).

`max_worker_processes` (`integer`)

Configure le nombre maximum de background workers acceptés par le système. Ce paramètre n'est configurable qu'au démarrage du serveur. La valeur par défaut est 8.

S'il s'agit de la configuration d'un serveur secondaire, vous devez configurer ce paramètre à une valeur supérieure ou égale à celui du serveur primaire. Dans le cas contraire, il ne sera pas possible d'exécuter des requêtes sur le serveur secondaire.

`max_parallel_workers_per_gather` (`integer`)

Configure le nombre maximum de processus parallèles pouvant être lancé par un seul nœud `Gather` ou `Gather Merge`. Les processus parallèles sont pris dans l'ensemble de processus établi par max_worker_processes, limité par max_parallel_workers. Notez que le nombre demandé de processus parallèles pourrait ne pas être disponible à l'exécution. Si cela survient, le plan s'exécutera avec moins de processus qu'attendu, ce qui pourrait être inefficace. La valeur par défaut est 2. Positionner cette valeur à 0 désactive l'exécution parallélisée de requête.

Notez que les requêtes parallélisées peuvent consommer considérablement plus de ressources que des requêtes non parallélisées parce que chaque processus parallèle est un processus totalement séparé qui a en gros le même impact sur le système qu'une session utilisateur supplémentaire. Ceci doit être pris en considération lors du choix d'une valeur pour ce paramètre, ainsi que lors de la

configuration d'autres paramètres qui contrôlent l'utilisation des ressources, comme par exemple work_mem. Les limites de ressources comme `work_mem` sont appliquées individuellement pour chaque processus, ce qui signifie que l'utilisation totale pourrait être bien plus importante que pour un seul processus. Par exemple, une requête parallélisée utilisant quatre processus pourrait utiliser jusqu'à cinq fois plus de CPU, de mémoire, de bande passante disque, et ainsi de suite qu'une requête non parallélisée.

Pour plus d'informations sur les requêtes parallélisées, voir Chapitre 15.

`max_parallel_maintenance_workers` (`integer`)

Indique le nombre maximum de workers parallèles qu'une commande utilitaire peut démarrer. Actuellement, les commandes utilitaires qui supportent les workers parallèles est `CREATE INDEX`, et seulement à la création d'un index B-tree, et `VACUUM` sans l'option `FULL`. Les workers parallèles sont déduits du pool de processus défini par max_worker_processes, dans la limite de max_parallel_workers. Notez que le nombre de workers demandé peut ne pas être disponible lors de l'exécution. Si cela arrive, l'opération utilitaire fonctionnera avec moins de workers qu'attendu. Le défaut est de 2. Passer cette valeur à 0 désactive l'utilisation des workers parallèles par les commandes utilitaires.

Notez que les commandes utilitaires parallélisées ne devraient pas consommer beaucoup plus de mémoire que leur équivalent non parallélisé. Cette stratégie diffère de celle adoptée pour les requêtes parallélisées, où les limites de ressources s'appliquent généralement par processus (worker). Les commandes utilitaires parallélisées traitent la limite de ressource `maintenance_work_mem` comme une limite à appliquer à la commande entière, sans considération du nombre de workers parallèles. Cependant, les commandes utilitaires parallélisées peuvent consommer nettement plus de CPU et de bande passante.

`max_parallel_workers` (`integer`)

Positionne le nombre maximum de workers que le système peut supporter pour le besoin des requêtes parallèles. La valeur par défaut est 8. Lorsque cette valeur est augmentée ou diminuée, pensez également à modifier max_parallel_maintenance_workers et max_parallel_workers_per_gather. De plus, veuillez noter que positionner cette valeur plus haut que max_worker_processes n'aura pas d'effet puisque les workers parallèles sont pris de la réserve de processus établie par ce paramètre.

`parallel_leader_participation` (`boolean`)

Permet au processus leader d'exécuter le plan de la requête sur les nœuds `Gather` et `Gather Merge` au lieu d'attendre le processus en cours. Positionner ce paramètre à `off` réduit la probabilité qu'un processus soit bloqué parce que le processus leader ne lit pas les données assez vite mais impose que le processus leader attende que les autres processus aient démarré avant que les premières lignes ne soient produites. Le degré à partir duquel le processus leader peut améliorer ou détériorer les performances dépend du type de plan d'exécution, du nombre de processus parallèles et de la durée d'exécution de la requête.

`old_snapshot_threshold` (`integer`)

Configure la durée minimale d'utilisation d'une image sans risque d'erreur `snapshot too old` survenant lors de l'utilisation de l'image. Les données mortes depuis plus longtemps que cette limite peuvent être nettoyées. Ceci peut aider à empêcher la fragmentation dans le cas de snapshots qui restent utiliser sur une longue période. Pour empêcher des résultats incorrects suite au nettoyage des données qui auraient été visibles par l'image, une erreur est générée quand l'image est plus ancienne que cette limite et que l'image est utilisée pour lire un bloc qui a été modifié depuis la construction du snapshot.

Si cette valeur est spécifiée sans unité, elle est considérée comme un nombre de minutes. Une valeur de `-1` (valeur par défaut) désactive cette fonctionnalité, en configurant la limite d'âge à l'infini. Ce paramètre peut seulement être configuré au démarrage du serveur.

Les valeurs utiles en production vont probablement d'un petit nombre d'heures à quelques jours. De petites valeurs (tels que `0` ou `1min`) sont seulement autorisés parce qu'ils pourraient être utiles pour des tests. Bien qu'une configuration aussi haute que `60d` est autorisée, notez que dans de nombreux cas, une fragmentation extrême ou une réutilisation des identifiants de transaction pourrait survenir très rapidement.

Quand cette fonctionnalité est activée, l'espace libérée à la fin de la relation ne peut pas être rendu au système d'exploitation car cela supprimerait les informations nécessaires pour détecter la condition « snapshot too old ». Tout l'espace alloué pour une relation reste associé avec cette relation pour une réutilisation par cette relation sauf si elle est explicitement libérée (par exemple, avec `VACUUM FULL`).

Ce paramètre ne tente pas de garantir qu'une erreur sera générée sous quelques circonstances. En fait, si les résultats corrects peuvent être générés à partir (par exemple) d'un curseur qui a matérialisé un ensemble de résultat, aucune erreur ne sera renvoyée même si les lignes impactées dans la table de référence ont été nettoyées. Certaines tables ne peuvent pas être nettoyées tôt proprement, et donc ne seront pas affectées par ce paramètre, comme les catalogues systèmes. Pour ces tables, ce paramètre ne réduira pas la fragmentation et ne pourra être la raison d'une erreur « snapshot too old » lors de son parcours.

20.5. Write Ahead Log

Voir aussi la Section 30.5 pour plus d'informations sur la configuration de ces paramètres.

20.5.1. Paramètres

`wal_level` (`enum`)

`wal_level` détermine la quantité d'informations écrite dans les journaux de transactions. `wal_level` détermine la quantité d'informations qui sera écrite dans les WAL. La valeur par défaut est `replica`, qui écrit suffisamment de données pour pouvoir utiliser l'archivage des WAL ainsi que la réplication, y compris exécuter des requêtes en lecture seule sur un serveur secondaire. `minimal` supprime toute la journalisation à l'exception des informations nécessaire pour pouvoir effectuer une récupération suite à un arrêt brutal ou un arrêt immédiat. Enfin, `logical` ajoute les informations nécessaires au support du décodage logique. Chaque niveau inclut les informations tracées dans les niveaux inférieurs. Ce paramètre peut seulement être configuré au lancement du serveur.

Le niveau `minimal` génère le moins de volume WAL. Il ne trace aucune ligne pour les relations permanentes dans des transactions qui les ont créées ou réécrites. Ceci peut rendre certaines opérations bien plus rapides (voir Section 14.4.7). Les opérations qui initient cette optimisation incluent :

`CREATE TABLE AS`
`CREATE INDEX`
`ALTER ... SET TABLESPACE`
`CLUSTER`
`COPY` dans des tables qui ont été créées ou tronquées dans la même transaction
`CREATE TABLE`
`REFRESH MATERIALIZED VIEW` (without `CONCURRENTLY`)
`REINDEX`
`TRUNCATE`

Mais, du coup, les journaux au niveau minimal ne contiennent pas suffisamment d'informations pour reconstruire les données à partir d'une sauvegarde de base et des journaux de transactions. Donc, les niveaux `replica` ou supérieurs doivent être utilisés pour activer l'archivage continue des journaux de transactions (archive_mode) et la réplication binaire en flux. En fait, le serveur ne démarrera pas dans ce mode si `max_wal_senders` est différent de zéro. Notez que changer

`wal_level` à la valeur `minimal` rend les sauvegardes de base précédentes inutilisables pour une restauration PITR et pour des serveurs secondaires.

Dans le niveau `logical`, les mêmes informations sont enregistrées que pour le mode `replica`. Des informations supplémentaires sont ajoutées pour extraire les modifications logiques depuis les journaux de transactions. En utilisant le niveau `logical`, le volume des journaux de transactions va augmenter, tout particulièrement si plusieurs tables sont configurées pour `REPLICA IDENTITY FULL` et que de nombreux `UPDATE` et `DELETE` sont exécutés.

Dans les versions antérieures à la 9.6, ce paramètre autorise aussi les valeurs `archive` et `hot_standby`. Elles sont toujours acceptées mais sont converties silencieusement en `replica`.

`fsync` (`boolean`)

Si ce paramètre est activé, le serveur PostgreSQL tente de s'assurer que les mises à jour sont écrites physiquement sur le disque à l'aide d'appels système `fsync()` ou de méthodes équivalentes (voir wal_sync_method). Cela permet de s'assurer que le cluster de bases de données peut revenir à un état cohérent après une panne matérielle ou du système d'exploitation.

Bien que désactiver `fsync` améliore fréquemment les performances, cela peut avoir pour conséquence une corruption des données non récupérables dans le cas d'une perte de courant ou d'un crash du système. Donc, il est seulement conseillé de désactiver `fsync` si vous pouvez facilement recréer la base de données complète à partir de données externes.

Quelques exemples de circonstances permettant de désactiver `fsync` : le chargement initial d'une nouvelle instance à partir d'une sauvegarde, l'utilisation de l'instance pour traiter un flot de données après quoi la base sera supprimée puis recréée, la création d'un clone d'une base en lecture seule, clone qui serait recréé fréquemment et n'est pas utilisé pour du failover. La haute qualité du matériel n'est pas une justification suffisante pour désactiver `fsync`.

Pour une restauration fiable lors de la modification de `fsync` de off à on, il est nécessaire de forcer tous les tampons modifiés disponibles dans le cache du noyau à être écrits sur un stockage durable. Ceci peut se faire alors que l'instance est arrêtée ou lorsque `fsync` est activé en exécutant `initdb --sync-only`, en exécutant `sync`, en démontant le système de fichiers ou en redémarrant le serveur.

Dans de nombreuses situations, désactiver synchronous_commit pour les transactions non critiques peut fournir une grande partie des performances de la désactivation de `fsync`, sans les risques associés de corruption de données.

`fsync` ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Si ce paramètre est désactivé (`off`), il est intéressant de désactiver aussi full_page_writes.

`synchronous_commit` (`enum`)

Indique quel traitement des WAL doit se faire avant que le serveur de bases de données ne renvoie une indication de « succès » au client. Les valeurs valides sont `remote_apply`, `on` (par défaut), `remote_write`, `local` et `off`.

Si `synchronous_standby_names` est vide, les seules valeurs sensées sont `on` et `off` ; `remote_apply`, `remote_write` et `local` fournissent toutes le même niveau de synchronisation locale que `on`. Le comportement local de tous les modes différents de `off` est d'attendre le vidage local sur disque des WAL. Dans le mode `off`, il n'y a pas d'attente, donc il peut y avoir un délai entre le retour du succès au client et le fait que la transaction est garantie d'être sécurisée contre un crash du serveur. (Le délai maximum est de trois fois wal_writer_delay.) Contrairement à fsync, la configuration de ce paramètre à `off` n'implique aucun risque d'incohérence dans la base de données : un arrêt brutal du système d'exploitation ou d'une base de données peut résulter en quelques transactions récentes prétendument validées perdues malgré

tout. Cependant, l'état de la base de données est identique à celui obtenu si les transactions avaient été correctement annulées. C'est pourquoi la désactivation de `synchronous_commit` est une alternative utile quand la performance est plus importante que la sûreté de la transaction. Pour plus de discussion, voir Section 30.4.

Si synchronous_standby_names n'est pas vide, `synchronous_commit` contrôle aussi si les validations de transactions attendront que leurs enregistrements WAL soient traités sur le serveur secondaire.

Quand il est configuré à `remote_apply`, les validations attendront la réponse des serveurs secondaires synchrones indiquant qu'ils ont bien reçu l'enregistrement de validation de la transaction et qu'ils l'ont bien appliqués, pour qu'elle devienne visible aux requêtes sur les serveurs secondaires, et aussi écrites sur un stockage durable. Ceci causera les plus gros délais de validation par rapport aux configurations précédentes car il faut attendre le rejeu des WAL. Quand il est configuré à `on`, les validations attendent que les réponses des serveurs secondaires synchrones indiquent qu'ils ont reçu l'enregistrement de validation de la transaction et qu'ils l'ont écrit sur un stockage durable. Ceci assure que la transaction ne sera pas perdu sauf si le primaire et les secondaires synchrones souffrent de corruption au niveau disque. Quand il est configuré à `remote_write`, les validations attendront que les réponses des serveurs secondaires synchrones indiquent avoir reçu l'enregistrement de validation de la transaction et l'avoir écrit sur disque. Ce paramétrage assure de la préservation des données si une instance secondaire de PostgreSQL s'arrête brutalement, mais pas si le serveur secondaire souffre d'un crash au niveau du système d'exploitation parce que les données n'ont pas nécessairement atteint un stockage durable sur le secondaire. Le paramétrage `local` fait que les validations attendent uniquement le vidage local sur disque, mais n'attendent pas le retour des serveurs secondaires synchrones. Ceci n'est généralement pas souhaité quand la réplication synchrone est utilisée mais est fourni pour être complet.

Ce paramètre peut être changé à tout moment ; le comportement pour toute transaction est déterminé par la configuration en cours lors de la validation. Il est donc possible et utile d'avoir certaines validations validées en synchrone et d'autres en asynchrone. Par exemple, pour réaliser une validation asynchrone de transaction à plusieurs instructions avec une valeur par défaut inverse, on exécute l'instruction `SET LOCAL synchronous_commit TO OFF` dans la transaction.

Tableau 20.1 résume les possibilités de configuration de `synchronous_commit`.

Tableau 20.1. Modes pour synchronous_commit

synchronous_commit	validation locale durable	valide durable du standby après un crash de PG	valide durable du standby après un crash de l'OS	cohérence des requêtes sur le standby
remote_apply	•	•	•	•
on	•	•	•	
remote_write	•	•		
local	•			
off				

`wal_sync_method` (`enum`)

Méthode utilisée pour forcer les mises à jour des WAL sur le disque. Si `fsync` est désactivé, alors ce paramètre est inapplicable, car les mises à jour des journaux de transactions ne sont pas du tout forcées. Les valeurs possibles sont :

- `open_datasync` (écrit les fichiers WAL avec l'option `O_DSYNC` de `open()`)
- `fdatasync` (appelle `fdatasync()` à chaque validation)

- `fsync_writethrough` (appelle `fsync()` à chaque validation, forçant le mode *write-through* de tous les caches disque en écriture)
- `fsync` (appelle `fsync()` à chaque validation)
- `open_sync` (écrit les fichiers WAL avec l'option `O_SYNC` de `open()`)

Ces options ne sont pas toutes disponibles sur toutes les plateformes. La valeur par défaut est la première méthode de la liste ci-dessus supportée par la plateforme, sauf que `fdatasync` est la valeur par défaut sur Linux et FreeBSD. Les options `open_*` utilisent aussi `O_DIRECT` s'il est disponible. L'outil `src/tools/fsync` disponible dans le code source de PostgreSQL permet de tester les performances des différentes méthodes de synchronisation. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`full_page_writes` (`boolean`)

Quand ce paramètre est activé, le serveur écrit l'intégralité du contenu de chaque page disque dans les WAL lors de la première modification de cette page qui intervient après un point de vérification. C'est nécessaire car l'écriture d'une page lors d'un plantage du système d'exploitation peut n'être que partielle, ce qui conduit à une page sur disque qui contient un mélange d'anciennes et de nouvelles données. Les données de modification de niveau ligne stockées habituellement dans les WAL ne sont pas suffisantes pour restaurer complètement une telle page lors de la récupération qui suit la panne. Le stockage de l'image de la page complète garantit une restauration correcte de la page, mais au prix d'un accroissement de la quantité de données à écrire dans les WAL. (Parce que la relecture des WAL démarre toujours à un point de vérification, il suffit de faire cela lors de la première modification de chaque page survenant après un point de vérification. De ce fait, une façon de réduire le coût d'écriture de pages complètes consiste à augmenter le paramètre réglant les intervalles entre points de vérification.)

La désactivation de ce paramètre accélère les opérations normales, mais peut aboutir soit à une corruption impossible à corriger soit à une corruption silencieuse, après un échec système. Les risques sont similaires à la désactivation de `fsync`, bien que moindres. Sa désactivation devrait se faire en se basant sur les mêmes recommandations que cet autre paramètre.

La désactivation de ce paramètre n'affecte pas l'utilisation de l'archivage des WAL pour la récupération d'un instantané, aussi appelé PITR (voir Section 26.3).

Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. Activé par défaut (`on`).

`wal_log_hints` (`boolean`)

Quand ce paramètre a la valeur `on`, le serveur PostgreSQL écrit le contenu entier de chaque page disque dans les journaux de transactions lors de la première modification de cette page après un checkpoint, même pour des modifications non critiques comme les hint bits.

Si les sommes de contrôle sont activées, la mise à jour des hint bits est toujours enregistrée dans les journaux et ce paramètre est ignoré. Vous pouvez utiliser ce paramètre pour tester le volume supplémentaire de journaux induit par l'activiation des sommes de contrôle sur les fichiers de données.

Ce paramètre n'est configurable qu'au démarrage du serveur. La valeur par défaut vaut `off`.

`wal_compression` (`enum`)

Ce paramètre active la compression des journaux de transactions en utilisant la méthode de compression indiquée. Lorsque la compression est activée, le serveur PostgreSQL compresse une image d'une page complète écrite dans les WAL (par exemple lorsque full_page_writes est à on, durant une sauvegarde de base, etc.). Une image compressée d'une page sera décompressée durant le rejeu des WAL. Les méthodes acceptées sont `pglz`, `lz4` (si PostgreSQL a été compilé avec `--with-lz4`) et `zstd` (si PostgreSQL a été compilé avec `--with-zstd`). La valeur `on` est

une valeur historique de `pglz`. La valeur par défaut est à `off`. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` appropriés peuvent modifier ce paramètre.

Activer la compression peut réduire le volume des WAL sans augmenter le risque de données corrompues irrécupérables, mais avec l'effet d'avoir un coût supplémentaire en terme de puissance CPU sur la compression durant l'écriture des WAL et sur la décompression lors du rejeu des WAL.

`wal_init_zero` (`boolean`)

Si configuré à `on` (la valeur par défaut), cette option fait que les nouveaux journaux de transactions sont remplis de zéro. Sur certains systèmes de fichiers, ceci assure que la place est allouée avant qu'il ne soit nécessaire d'écrire les enregistrements WAL. Néanmoins, les systèmes de fichiers *Copy-On-Write* (COW) pourraient ne pas bénéficier de cette technique, donc l'option est donnée pour éviter ce travail inutile. Si configuré à `off`, seul l'octet final est écrit quand le fichier est créé pour qu'il ait la taille attendue.

`wal_recycle` (`boolean`)

Si configuré à `on` (la valeur par défaut), cette option fait que les fichiers WAL sont recyclés en les renommant, pour éviter d'avoir à créer de nouveaux fichiers. Sur les systèmes de fichiers COW, il pourrait être plus rapide d'en créer de nouveaux, donc l'option est donnée pour désactiver ce comportement.

`wal_buffers` (`integer`)

La quantité de mémoire partagée utilisée pour les données des journaux de transactions qui n'ont pas encore été écrites sur disque. La configuration par défaut de -1 sélectionne une taille égale à 1/32 (environ 3%) de shared_buffers, mais pas moins de `64kB`, et pas plus que la taille d'un journal de transactions, soit généralement `16MB`. Cette valeur peut être configurée manuellement si le choix automatique est trop élevé ou trop faible, mais tout valeur positive inférieure à `32kB` sera traitée comme étant exactement `32kB`. Si cette valeur est spécifiée sans unité, elle est prise en tant que nombre de blocs de journaux de transactions, autrement dit `XLOG_BLCKSZ` octets, typiquement 8 Ko. Ce paramètre ne peut être configuré qu'au démarrage du serveur.

Le contenu du cache des journaux de transactions est écrit sur le disque à chaque validation d'une transaction, donc des valeurs très importantes ont peu de chance d'apporter un gain significatif. Néanmoins, configurer cette valeur à au moins quelques mégaoctets peut améliorer les performances en écriture sur un serveur chargé quand plusieurs clients valident en même temps. La configuration automatique sélectionnée par défaut avec la valeur -1 devrait être convenable.

`wal_writer_delay` (`integer`)

Indique à quelle fréquence (une durée) le walwriter vide les journaux sur disque. Après avoir vidé les journaux sur disque, ce processus s'endort pour la durée indiquée par le paramètre `wal_writer_delay` sauf s'il est réveillé par une transaction validée en asynchrone. Dans le cas où le dernier vidage est survenu il y a moins de `wal_writer_delay` millisecondes et que moins de `wal_writer_flush_after` octets ont été produits dans les WAL depuis, le WAL est seulement écrit via le système d'exploitation mais pas forcément écrit sur disque. Si cette valeur est spécifiée sans unité, elle est considérée être en millisecondes. La valeur par défaut est 200 millisecondes (`200ms`). Notez que sur de nombreux systèmes, la résolution réelle du délai d'endormissement est de 10 millisecondes ; configurer `wal_writer_delay` à une valeur qui n'est pas un multiple de 10 pourrait avoir le même résultat que de le configurer au prochain multiple de 10. Ce paramètre est seulement configurable dans le fichier `postgresql.conf` ainsi que sur la ligne de commande du serveur.

`wal_writer_flush_after` (`integer`)

Indique à quelle fréquence (une quantité) le walwriter vide les journaux sur disque. Dans le cas où le dernier vidage est arrivé il y a moins de `wal_writer_delay` millisecondes et que moins de `wal_writer_flush_after` octets de WAL ont été produits depuis, les WAL sont seulement

écrit via le système d'exploitation, et pas forcé sur disque. Si `wal_writer_flush_after` est configuré à `0`, le WAL est écrit et vidé à chaque fois que le walwriter doit écrire dans un WAL. Si cette valeur est indiquée sans unité, est est considérée comme un nombre de blocs dans les journaux de transactions, autrement dit `XLOG_BLCKSZ` octets, typiquement 8 Ko. La valeur par défaut est `1MB`. Ce paramètre est seulement configurable dans le fichier `postgresql.conf` ainsi que sur la ligne de commande du serveur.

`wal_skip_threshold` (`integer`)

Quand `wal_level` vaut `minimal` et une transaction valide après création et réécriture une table permanente, ce paramètre détermine comment les nouvelles données persistent. Si la donnée est plus petit que ce paramétrage, l'écrire dans les journaux de transactions ; sinon, utiliser une demande de synchronisation (fsync) des fichiers affectés. Suivant les propriétés de votre stockage, augmenter ou abaisser cette valeur pourrait aider si de telles validations de données ralentissent les transactions en cours. Si cette valeur est indiquée sans unité, elle est considérée être un nombre de Ko. La valeur par défaut est de deux mégaoctets (`2MB`).

`commit_delay` (`integer`)

Configurer `commit_delay` ajoute un délai avant qu'un vidage du journal de transactions ne soit effectué. Ceci peut améliorer les performances de la validation en groupe en permettant la validation d'un grand nombre transactions en un seul vidage des journaux, si la charge système est suffisamment importante pour que des transactions supplémentaires soient prêt ç être valider dans le même intervalle. Néanmoins, cela augmente aussi la latence jusqu'à la valeur de `commit_delay` pour chaque vidage de journaux. Comme le délai est perdu si aucune autre transaction n'est prête à être validée, un délai n'est respecté que si au moins `commit_siblings` autres transactions sont actives quand un vidage doit être initié. De plus, aucun délai ne sera pris en compte si `fsync` est désactivé. Si cette valeur est indiquée sans unité, elle est considérée comme un nombre de microsecondes. La valeur par défaut de `commit_delay` est zéro (aucun délai). Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` appropriés peuvent modifier cette configuration.

Dans les versions de PostgreSQL antérieures à la 9.3, `commit_delay` se comportait différemment et était bien moins efficace : il n'affectait que les validations plutôt que les vidages de journaux et attendait que le délai complet soit passé même si le vidage du journal était terminé avant. À partir de PostgreSQL 9.3, le premier processus prêt à vider le journal attend pendant l'intervalle configuré alors que les autres processus attendent que le premier termine l'opération de vidage.

`commit_siblings` (`integer`)

Nombre minimum de transactions concurrentes ouvertes en même temps nécessaires avant d'attendre le délai `commit_delay`. Une valeur plus importante rend plus probable le fait qu'au moins une autre transaction soit prête à valider pendant le délai. La valeur par défaut est de cinq transactions.

20.5.2. Points de vérification

`checkpoint_timeout` (`integer`)

Temps maximum entre deux points de vérification automatique des WAL. Si cette valeur est indiquée sans unité, elle est considérée comme un nombre de secondes. L'intervalle valide se situe entre 30 secondes et un jour. La valeur par défaut est de cinq minutes. Augmenter ce paramètre peut accroitre le temps nécessaire à une récupération après un arrêt brutal. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`checkpoint_completion_target` (`floating point`)

Précise la cible pour la fin du CHECKPOINT, sous le format d'une fraction de temps entre deux CHECKPOINT. La valeur par défaut est 0,9, ce qui répartit le checkpoint sur presque tout

l'interval de temps entre deux checkpoints, permettant ainsi d'absorber la charge d'écriture de manière régulière tout en laissant un peu de temps pour absorber la charge supplémentaire induite par le checkpoint. Réduire la valeur de ce paramètre n'est pas recommendé car cela engendrera des checkpoints se finissant plus tôt, ce qui entraînera un pic dans les I/O alors que la fin du checkpoint entraînera un creux dans les I/O entre la fin du checkpoint courant et le début du prochain checkpoint. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`checkpoint_flush_after` (`integer`)

Quand plus que cette quantité de données ont été écrites par le processus d'écriture en tâche de fond (bgwriter), tente de forcer le système d'exploitation à écrire les données sur disque. Faire cela limite la quantité de données modifiées dans le cache disque du noyau, réduisant le risque de petites pauses dues à l'exécution d'un fsync à la fin d'un checkpoint ou à l'écriture massive en tâche de fond des données modifiées. Souvent, cela réduira fortement la latence des transactions mais il existe aussi quelques cas de dégradation des performances, tout spécialement avec les charges de travail plus importantes que shared_buffers, mais plus petites que le cache disque du système d'exploitation. Ce paramètre pourrait ne pas avoir d'effet sur certaines plateformes. Si cette valeur est indiquée sans unité, elle est considérée comme un nombre de blocs, autrement dit `BLCKSZ` octets, typiquement 8 Ko. L'intervalle valide se situe entre `0`, qui désactive le « writeback » forcé, et `2MB`. La valeur par défaut est `256KB` sur Linux, `0` ailleurs. (Si `BLCKSZ` ne vaut pas 8 ko, les valeurs par défaut et maximale n'évoluent pas de façon proportionnelle à cette constante.) Ce paramètre est seulement configurable dans le fichier `postgresql.conf` et à la ligne de commande.

`checkpoint_warning` (`integer`)

Si deux points de vérification imposés par le remplissage des fichiers segment interviennent dans un délai plus court que celui indiqué par cette durée (ce qui laisse supposer qu'il faut augmenter la valeur du paramètre `max_wal_size`), un message est écrit dans le fichier de traces du serveur. Si cette valeur est indiquée sans unité, elle est considérée être un nombre de secondes. Par défaut, 30 secondes. Une valeur nulle (0) désactive cet avertissement. Aucun avertissement ne sera fait si `checkpoint_timeout` est inférieur à `checkpoint_warning`. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`max_wal_size` (`integer`)

Taille maximale de l'augmentation des WAL entre deux points de vérification automatique des WAL. C'est une limite souple ; la taille des WAL peut excéder `max_wal_size` sous certaines circonstances, comme une surcharge du serveur, une commande `archive_command` ou `archive_library` qui échoue, ou une configuration haute pour `wal_keep_size`. Si cette valeur est indiquée sans unité, elle est considérée être un nombre de Mo. La valeur par défaut est 1 Go. Augmenter ce paramètre peut augmenter le temps nécessaire pour le rejeu suite à un crash. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`min_wal_size` (`integer`)

Tant que l'occupation disque reste sous la valeur de ce paramètre, les anciens fichiers WAL sont toujours recyclés pour une utilisation future lors des points de vérification, plutôt que supprimés. Ceci peut être utilisé pour s'assurer qu'un espace suffisant est réservé pour faire face à des pics dans l'usage des WAL, par exemple lorsque d'importants travaux en lots sont lancés. Si cette valeur est indiquée sans unité, elle est considérée être un nombre de Mo. La valeur par défaut est 80 Mo. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

20.5.3. Archivage

`archive_mode` (`enum`)

Quand archive_mode est activé, les segments WAL remplis peuvent être archivés en configurant archive_command ou archive_library. En plus de off, pour désactiver, il y a deux autres modes : on, et always. Lors du fonctionnement normal du serveur, il n'y a pas de différences entre les deux modes, mais lorsqu'il est positionné sur always, l'archiveur des WAL est aussi activé lors d'un rejeu des archives et en mode standby. Dans le mode always, tous les fichiers restaurés à partir de l'archive ou envoyés lors de la réplication en continue seront archivés (à nouveau). Voir Section 27.2.9 pour des détails.

archive_mode est un paramétrage séparé de archive_command et archive_library pour que archive_command et archive_library puissent être modifiés sans abandonner le mode d'archivage. Ce paramètre ne peut être configuré qu'au lancement du serveur. archive_mode ne peut pas être activé quand wal_level est configuré à minimal.

archive_command (string)

Commande shell à exécuter pour archiver un segment terminé de la série des fichiers WAL. Tout %p dans la chaîne est remplacé par le chemin du fichier à archiver et tout %f par le seul nom du fichier. (Le chemin est relatif au répertoire de travail du serveur, c'est-à-dire le répertoire de données du cluster.) %% est utilisé pour intégrer un caractère % dans la commande. Il est important que la commande renvoie un code zéro seulement si elle a réussit l'archivage. Le processus d'archivage des journaux de transactions est redémarré par le postmaster quand ce paramètre est modifié. Pour plus d'informations, voir Section 26.3.1.

Ce paramètre ne peut être configuré que dans le fichier postgresql.conf ou indiqué sur la ligne de commande. Il est ignoré sauf si archive_mode a été activé au lancement du serveur et archive_library est configuré à une chaîne vide. Si archive_command est une chaîne vide (la valeur par défaut) alors que archive_mode est activé (et qu'archive_library est configuré à une chaîne vide), alors l'archivage des journaux de transactions est désactivé temporairement mais le serveur continue d'accumuler les fichiers des journaux de transactions dans l'espoir qu'une commande lui soit rapidement proposée. Configurer archive_command à une commande qui ne fait rien tout en renvoyant true, par exemple /bin/true (REM sur Windows), désactive l'archivage mais casse aussi la chaîne des fichiers des journaux de transactions nécessaires pour la restauration d'une archive. Cela ne doit donc être utilisé quand lors de circonstances inhabituelles.

archive_library (string)

Bibliothèque à utiliser pour l'archivage des journaux de transactions terminés. Si configuré à une chaîne vide (valeur par défaut), l'archivage via le shell est activé et archive_command est utilisé. Sinon, la bibliothèque partagée est utilisée pour l'archivage. Pour plus d'informations, voir Section 26.3.1 et Chapitre 51.

Ce paramètre peut seulement être configuré dans le fichier postgresql.conf et sur la ligne de commande du serveur.

archive_timeout (integer)

Le archive_command ou le archive_library n'est appelé que pour les segments WAL remplis. De ce fait, si le serveur n'engendre que peu de trafic WAL (ou qu'il y a des périodes de plus faible activité), il se peut qu'un long moment s'écoule entre la fin d'une transaction et son archivage certain. Pour limiter l'âge des données non encore archivées, archive_timeout peut être configuré pour forcer le serveur à basculer périodiquement sur un nouveau segment WAL. Lorsque ce paramètre est positif, le serveur bascule sur un nouveau segment à chaque fois que cette durée s'est écoulée depuis le dernier changement de segment et qu'il n'y a pas eu d'activité de la base de données, y compris un seul CHECKPOINT. (les points de reprise sont ne sont pas effectués s'il n'y a pas d'activité sur les bases.) Les fichiers archivés clos par anticipation suite à une bascule imposée sont toujours de la même taille que les fichiers complets. Il est donc déconseillé de configurer un temps très court pour archive_timeout -- cela va faire exploser la taille du stockage des archives. Un paramétrage d'archive_timeout de l'ordre de la minute est habituellement raisonnable. Cependant, vous devriez considérer l'utilisation de

la réplication en flux à la place de l'archivage si vous voulez que les données soient envoyées du serveur primaire plus rapidement que cela. Si cette valeur est indiquée sans unité, elle est considérée comme un nombre de secondes. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

20.5.4. Restauration

Cette section décrit les paramètres s'appliquant à la restauration en général, affectant donc la restauration après un crash, la réplication en flux et la réplication à partir des archives.

`recovery_prefetch` (`enum`)

Indique s'il faut récupérer en avance (*prefetch*) les blocs référencés dans le journal de transaction qui ne sont pas encore dans le cache lors de la restauration. Les valeurs valides sont `off`, `on` et `try` (la valeur par défaut). La configuration `try` active la récupération en avance uniquement si le système d'exploitation fournit la fonction `posix_fadvise`, qui est actuellement utilisé pour implémenter la récupération en avance. Notez que certains systèmes d'exploitation fournissent la fonction mais elle ne fait rien.

Récupérer en avance les blocs qui seront bientôt nécessaires peut réduire les durées d'attente pour les opérations d'entrée/sortie disques lors de la restauration avec certains types de charge. Voir aussi les paramètres wal_decode_buffer_size et maintenance_io_concurrency, qui limitent l'activité de récupération en avance.

`wal_decode_buffer_size` (`integer`)

Une limite sur l'avancée que peut avoir le serveur dans les WAL pour trouver les blocs à récupérer en avance. Si cette valeur n'a pas d'unité, elle sera considérée comme un nombre d'octets. La valeur par défaut est de 512 Ko.

20.5.5. Restauration des archives

Cette section décrit la configuration s'appliquant uniquement pendant la durée d'une restauration. Les paramètres doivent être reconfigurés pour toute restauration que vous souhaitez réaliser.

La « restauration » couvre l'utilisation d'un serveur en tant que standby ainsi que l'exécution d'une restauration ciblée. Typiquement, le mode standby sera utilisé pour fournir de la haute disponibilité et/ou de la répartition de charge en lecture, alors qu'une restauration ciblée sera utilisée dans le cas d'une perte de données.

Pour démarrer le serveur en mode standby, créez le fichier `standby.signal` dans le répertoire principal des données. Le serveur entrera en mode restauration et n'arrêtera la restauration que quand la fin d'un WAL archivé est rencontré, mais il essaiera de continuer la restauration en se connectant au serveur d'envoi spécifié par le paramètre `primary_conninfo` et/ou en récupérant les segments WAL avec la `restore_command`. Pour ce mode, les paramètres de cette section et de Section 20.6.3 sont intéressants. Les paramètres de Section 20.5.6 seront aussi appliqués mais ne sont généralement pas utiles dans ce mode.

Pour démarrer le serveur en mode restauration ciblée, créez le fichier `recovery.signal` dans le répertoire des données. Si les fichiers `standby.signal` et `recovery.signal` sont créés, le mode standby est prioritaire. Le mode de restauration ciblée s'arrêtera quand le WAL archivé est complètement rejoué ou quand `recovery_target` est atteint. Dans ce mode, les paramètres de cette section et de Section 20.5.6 seront utilisés.

`restore_command` (`string`)

La commande shell locale à exécuter pour récupérer un segment WAL archivé. Ce paramètre est requi pour une restauration d'archive, et optionnel pour une réplication en streaming. Tout `%f` dans la chaîne est remplacé par le nom du fichier à récupérer dans le répertoire d'archivage, et

tout %p est remplacé par le nom du chemin destination de la copie sur le serveur. (Le chemin est relatif au répertoire actuel, donc le répertoire de données principal de l'instance.) Tout %r est remplacé par le nom du fichier contenant le dernier point de redémarrage valide. C'est le fichier le plus récent à conserver pour permettre le lancement d'une restauration, pour que cette information puisse être utilisée pour tronquer l'archive au minimum requis pour permettre le redémarrage de la restauration en cours. %r est généralement utilisé seulement pour les configurations warm-standby (voir Section 27.2). Écrire %% pour ajouter un caractère %.

Il est important que la commande renvoie un code de sortie zéro uniquement en cas de succès. La commande *doit* gérer le fait que des fichiers ne soient pas présents dans les archives ; dans ce cas, elle doit renvoyer un code de sortie différent de zéro. Par exemple :

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'  #
 Windows
```

Il existe une exception quand la commande est arrêtée avec un signal (autre que SIGTERM, qui est utilisé pour l'arrêt du serveur de bases de données) ou une erreur provenant du shell (par exemple « command not found »), alors la restauration s'arrêtera et le serveur ne démarrera pas.

Cette paramètre peut seulement être configuré au démarrage du serveur.

`archive_cleanup_command` (`string`)

Ce paramètre optionnel indique une commande shell exécutée à chaque restartpoint. Le but de `archive_cleanup_command` est de fournir un mécanisme pour nettoyer les anciens fichiers WAL archivés qui ne sont plus nécessaire sur le serveur standby. Tout %r est remplacé par le nom du fichier contenant le dernier point de redémarrage valide. C'est le fichier le plus ancien à *conserver* pour permettre une restauration, et donc tous les fichiers plus ancien que %r peuvent être supprimés en toute sécurité. Cette information peut être utilisée pour tronquer les archives au minimum requis pour supporter le redémarrage à partir de la restauration en cours. Le module pg_archivecleanup est souvent utilisé dans `archive_cleanup_command` pour les configurations avec un seul standby, par exemple ::

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/
archivedir %r'
```

Notez néanmoins que si plusieurs serveurs standby exécutent leur restauration à partir du même répertoire d'archivage, vous aurez besoin de vous assurer que vous ne supprimez que des fichiers WAL dont aucun serveur n'a besoin. `archive_cleanup_command` serait typiquement utilisé dans une configuration warm-standby (voir Section 27.2). Écrire %% pour intégrer un vrai caractère % dans la commande.

Si la commande renvoie un code de sortie différent de zéro, alors un message d'avertissement sera écrit dans les traces. Une exception survient quand la commande est terminée par un signal ou une erreur du shell (tel que « command not found »), une erreur fatale sera renvoyée.

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`recovery_end_command` (`string`)

Ce paramètre indique une commande shell à exécuter une fois arrivé à la fin de la restauration. Ce paramètre est optionnel. Le but de `recovery_end_command` est de fournir un mécanisme pour nettoyer après une réplication ou une restauration. Tout %r est remplacé par le nom du fichier contenant le dernier point de redémarrage valide, comme dans archive_cleanup_command.

Si la commande renvoie un code de sortie différent de zéro, alors un message d'avertissement sera écrit dans les traces et la base de données continuera à démarrer. Une exception survient si la

commande a été terminée par un signal ou une erreur du shell (tel que « command not found »), la base de données ne continuera pas avec le démarrage.

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

20.5.6. Cible de restauration

Par défaut, la restauration continuera jusqu'à la fin des WAL. Les paramètres suivants peuvent être utilisés pour indiquer un point d'arrêt précédent. Au moins un des paramètres parmi `recovery_target`, `recovery_target_lsn`, `recovery_target_name`, `recovery_target_time` et `recovery_target_xid` peut être utilisé. Si plus d'un paramètre est indiqué dans le fichier de configuration, une erreur sera renvoyée. Ces paramètres ne peuvent être configurés qu'au démarrage du serveur.

`recovery_target = 'immediate'`

Ce paramètre indique que la restauration doit s'arrêter dès qu'un point de cohérence est atteint, autrement dit le plus tôt possible. Lors de la restauration d'une sauvegarde, cela signifie le moment où la sauvegarde s'est terminée.

Techniquement, la valeur est une chaîne de caractère, mais `'immediate'` est actuellement la seule valeur autorisée.

`recovery_target_name` (`string`)

Ce paramètre indique le point de restauration nommé (créé précédemment avec `pg_create_restore_point()`) où la restauration doit s'arrêter.

`recovery_target_time` (`timestamp`)

Ce paramètre indique jusqu'à quel date et heure la restauration doit s'arrêter. Le point d'arrêt précis est aussi influencé par recovery_target_inclusive.

`recovery_target_xid` (`string`)

Ce paramètre indique l'identifiant de transaction final où la restauration s'arrêtera. Gardez en tête que, bien que les identifiants soient affectés séquentiellement au début de la transaction, la transaction peut se terminer dans un ordre numérique différent. Les transactions qui seront restaurées sont celles validées avant (et en option en incluant) celle indiquée. Le point d'arrêt précis est aussi influencé par recovery_target_inclusive.

La valeur de ce paramètre est un horodatage dans le même format que celui accepté par le type de données `timestamp with time zone`, à l'exception que vous ne pouvez pas utiliser une abréviation de fuseau horaire (sauf si le paramètre timezone_abbreviations a été configuré précédemment dans le fichier de configuration). Le style préféré est d'utiliser un décalage numérique à partir d'UTC. Vous pouvez aussi écrire un nom complet de fuseau horaire, par exemple `Europe/Helsinki`, et non pas `EEST`.

`recovery_target_lsn` (`pg_lsn`)

Ce paramètre indique le LSN où la restauration s'arrêtera. Le point d'arrêt précis est aussi influencé par recovery_target_inclusive. Ce paramètre est analysé en utilisant le type de données système `pg_lsn`.

Les options suivants indiquent plus en détails la cible de restauration, et affectent ce qui survient quand la cible est atteinte :

`recovery_target_inclusive` (`boolean`)

Indique s'il faut arrêter juste après la cible de restauration indiquée (`on`) ou juste avant (`off`). S'applique quand recovery_target_lsn, recovery_target_time ou recovery_target_xid est

spécifié. Ce paramètre contrôle si les transactions ayant exactement, respectivement, le même emplacement WAL (LSN), heure de validation, ou identifiant de transaction seront incluses dans la restauration. La valeur par défaut est `on`.

`recovery_target_timeline` (`string`)

Indique la restauration jusqu'à une certaine timeline. La valeur peut être un identifiant numérique de timeline. La valeur `current` restaure uniquement sur la même timeline que celle de la sauvegarde de base. La valeur `latest` restaure jusqu'à la dernière timeline trouvée dans les archives, ce qui est utile pour un serveur standby. `latest` est la valeur par défaut.

Vous avez seulement besoin de configurer ce paramètre dans les situations complexes de re-restaurations, où vous avez besoin de retourner à un état qui a été lui-même atteint après une restauration à un point dans le temps. Voir Section 26.3.5 pour plus de détails.

`recovery_target_action` (`enum`)

Indique l'action que le serveur devra prendre une fois la cible de restauration atteinte. La valeur par défaut est `pause`, ce qui signifie que la restauration sera mise en pause. `promote` signifie que le processus de restauration finira et que le serveur démarrera pour accepter toute connexion. Enfin, `shutdown` arrêtera le serveur après avoir atteint la cible de restauration.

Le but de la configuration `pause` est de permettre d'exécuter des requêtes sur la base pour vérifier si la cible de restauration est le point réellement souhaité pour la fin de la restauration. La mise en pause peut être annulée en utilisant `pg_wal_replay_resume()` (voir Tableau 9.91), qui cause ainsi la fin de la restauration. Si la cible de restauration n'est pas le point d'arrêt souhaité, alors arrêtez le serveur, modifiez la configuration de la cible de restauration à un point ultérieur et redémarrer pour continuer la restauration.

La configuration `shutdown` est utile pour avoir l'instance prête au point de rejeu exact désiré. L'instance sera toujours capable de rejouer plus d'enregistrements WAL (et en fait, continuera à rejouer des enregistrements WAL depuis le dernier checkpoint à son redémarrage).

Notez que comme `recovery.signal` ne sera pas supprimé quand `recovery_target_action` est configuré à `shutdown`, tout redémarrage finira avec un arrêt immédiat à moins que la configuration ait changé ou que le fichier `recovery.signal` ait été supprimé manuellement.

Cette configuration n'a pas d'effet si aucune cible de restauration n'a été configurée. Si hot_standby n'est pas activé, une configuration à `pause` agira de la même façon qu'une configuration à `shutdown`. Si la cible de restauration est atteinte alors qu'une promotion est en cours, une configuration à `pause` agira de la même façon qu'une configuration à `promote`.

Dans tous les cas, si une cible de restauration est configurée mais que la restauration d'archive se termine avant d'avoir atteint la cible, le serveur s'arrêtera avec une erreur fatale.

20.6. Réplication

Ces paramètres contrôlent le comportement de la fonctionnalité interne de *réplication en flux* (voir Section 27.2.5). Les serveurs seront soit primaire soit secondaire. Les primaires peuvent envoyer des données alors que les secondaires sont toujours des récepteurs des données de réplication. Quand la réplication en cascade est utilisée (voir Section 27.2.7), les secondaires peuvent aussi envoyer des données en plus de les réceptionner. Les paramètres sont principalement pour les serveurs d'envoi et en standby, bien que certains n'ont un intérêt que pour le serveur primaire. Les paramètres peuvent varier dans l'instance sans problèmes si cela est requis.

20.6.1. Serveurs d'envoi

Ces paramètres peuvent être configurés sur les serveur qui va envoyer les données de réplication à un ou plusieurs serveurs. Le primaire est toujours un serveur en envoi. Donc ces paramètres doivent être

configurés sur le primaire. Le rôle et la signification de ces paramètres ne changent pas après qu'un serveur standby soit devenu le serveur primaire.

`max_wal_senders` (`integer`)

Indique le nombre maximum de serveurs standby (autrement dit, le nombre maximum de processus walsender en cours d'exécution). La valeur par défaut est `10`. La valeur `0` signifie que la réplication est désactivée. Une déconnexion abrute d'un client de réplication pourrait avoir pour effet un slot de connexion orpheline jusqu'au dépassement d'un délai, donc ce paramètre peut être configuré un peu au-dessus du nombre maximum de clients attendus pour que les clients déconnectés puissent immédiatement se reconnecter. Ce paramètre n'est configurable qu'au démarrage du serveur. `wal_level` doit être configuré au minimum à `replica` pour permettre des connexions des serveurs secondaires.

Lors de l'exécution d'un serveur standby, vous devez configurer ce paramètre à la même valeur ou à une valeur supérieure à celle se trouvant sur le serveur d'envoi. Dans le cas contraire, les requêtes ne seront pas autorisées sur le serveur standby.

`max_replication_slots` (`integer`)

Indique le nombre maximum de slots de réplication (voir Section 27.2.6) que le serveur peut accepter. La valeur par défaut est 10. Ce paramètre est seulement configurable au lancement du serveur. Descendre ce paramètre à une valeur inférieure au nombre de slots de réplication existants empêchera le serveur de démarrer. `wal_level` doit aussi être positionné à `replica` ou au-delà pour permettre l'utilisation des slots de réplication.

Du côté du souscripteur, indique le nombre d'origines de réplication (voir Chapitre 50) pouvant être tracées simultanément, limitant directement le nombre de souscriptions de réplication logique pouvant être créées sur le serveur. Le configurer à une valeur plus basse que le nombre actuel d'origines de réplication tracées (telle qu'indiquée dans pg_replication_origin_status, et non pas pg_replication_origin) empêchera le démarrage du serveur.

`output_plugin_libraries` (`string`)

Liste les bibliothèques partagées installées dans dynamic_library_path qui sont acceptées pour une utilisation comme plugins de sortie de réplication logique par des clients de réplication. Toute demande de décodage logique ou de replication pour d'autres bibliothèques sera refusée. Tous les utilisateurs sont sujets à cette restriction. La valeur par défaut est `'pgoutput, test_decoding'`, qui sont les deux plugins de sortie logique inclus dans la distribution standard de PostgreSQL.

Le format est une liste de noms de bibliothèques, séparés par des virgules, où chaque nom est interprété comme par la commande `LOAD` (mais les clients de décodage logique doivent indiquer un nom de plugin qui correspond *exactement* à une entrée dans la liste, sans variations dans la casse ou la structure du chemin). Les espaces blancs entre les entrées sont ignorés ; entourer un nom de bibliothèque avec des guillemets doubles si vous avez besoin d'inclure des espaces blancs ou des virugules dans le nom.

Il est de la responsabilité de l'administrateur du serveur de s'assurer que les bibliothèques ajoutées à cette liste ne donnent pas de droits supplémentaires à des utilisateurs standards quand elles sont chargées dans le serveur.

> **Note**
>
> Lors de la mise à jour du serveur à partir d'une version qui n'a pas le paramètre `output_plugin_libraries`, la requête suivante peut aider à construire la liste des plugins requis par tous les slots de réplication logiques persistents :

```
SE DISTINCT plugin FROM pg_replication_slots WHERE plugin
 IS NOT NULL;
```

Vérifier la liste précautionneusement avant d'ajuster `output_plugin_libraries`.

La requête ci-dessus affiche seulement les plugins qui ont été ajoutés avec succès aux slots de réplication dans le passé. Les requêtes nouvellement refusées apparaitront dans les traces avec un message similaire à :

```
ER  library "..." may not be used as an output plugin
DE:  The configuration parameter
 "output_plugin_libraries" (currently 'pgoutput,
 test_decoding') does not name this library as a trusted
 output plugin.
HI If it is safe for all REPLICATION users to
 use this library as an output plugin, add it to
 "output_plugin_libraries" and reload the server
 configuration.
```

`wal_keep_size` (`integer`)

Indique la taille minimale de journaux de transactions passés à conserver dans le répertoire `pg_wal`, au cas où un serveur en attente a besoin de les récupérer pour la réplication en flux. Si un serveur en attente connecté au primaire se laisse distancer par le serveur en envoi pour plus de `wal_keep_size` méga-octets, le serveur en envoi pourrait supprimer un journal de transactions toujours utile au serveur en attente, auquel cas la connexion de réplication serait fermée. Les connexions en aval seront également vouées à l'échec. (Néanmoins, le serveur en attente peut continuer la restauration en récupérant le segment des archives si l'archivage des journaux de transactions est utilisé.)

Cette option ne configure que la volumétrie minimale de fichiers à conserver dans `pg_wal` ; le système pourrait avoir besoin de conserver plus de fichiers pour l'archivage ou pour restaurer à partir d'un CHECKPOINT. Si `wal_keep_size` vaut zéro (ce qui est la valeur par défaut), le système ne conserve aucun fichier supplémentaire pour les serveurs en attente et le nombre des anciens journaux disponibles pour les serveurs en attente est seulement basé sur l'emplacement du dernier CHECKPOINT ainsi que sur l'état de l'archivage des journaux de transactions. Si cette valeur est configurée sans unité, elle est prise comme des méga-octets. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`max_slot_wal_keep_size` (`integer`)

Indique la taille maximale des journaux de transaction que les slots de réplication sont autorisés à conserver dans le répertoire `pg_wal` lors d'un checkpoint. Si `max_slot_wal_keep_size` vaut -1 (valeur par défaut), les slots de réplication contiennent une quantité illimitée de fichiers de journaux de transactions. Si la valeur du restart_lsn d'un slot de réplication est en retard de plus de ce nombre de mégaoctets depuis le LSN actuel, le standby utilisant le slot pourrait ne plus pouvoir continuer la réplication du fait de la suppression des fichiers WAL requis. Vous pouvez avoir la disponibilité des journaux de transactions pour les slots de réplication dans la vue pg_replication_slots. Si cette valeur est indiquée sans unité, elle est prise pour des mégaoctets. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`wal_sender_timeout` (`integer`)

Termine les connexions de réplication qui sont inactives pour plus longtemps que cette durée. Ceci est utile pour que le serveur d'envoi détecte le crash d'un standby ou une perte réseau. Si

cette valeur est spécifiée sans unité, elle est comprise comme un nombre de millisecondes. La valeur par défaut est de 60 secondes. À zéro, le mécanisme est désactivé.

Avec un cluster distribué sur plusieurs lieux géographiques, utiliser plusieurs valeurs par lieu fournit plus de flexibilité dans la gestion du cluster. Une valeur plus petite est utile pour une détection plus rapide des problèmes avec un standby ayant un réseau à basse latence. Une valeur plus haute aide à mieux juger la santé d'un standby si ce dernier est situé sur un lieu distant, avec une connexion réseau à haute latence.

`track_commit_timestamp` (`boolean`)

Enregistre la date et l'heure des transactions validées. Ce paramètre peut seulement être configuré au démarrage du serveur. La valeur par défaut est `off`.

20.6.2. Serveur primaire

Ces paramètres peuvent être configurés sur le serveur primaire pour envoyer des données de réplication à un ou plusieurs serveurs en standby. Notez qu'en plus de ces paramètres, wal_level doit être configuré correctement sur le serveur primaire et que l'archivage des journaux de transactions peut aussi être activé (voir Section 20.5.3). Les valeurs de ces paramètres ne sont pas pris en compte sur les serveurs en standby. Il peut être intéressant de les mettre en place malgré tout en préparation de la possibilité qu'un secondaire devienne le primaire.

`synchronous_standby_names` (`string`)

Précise une liste de noms de serveurs en standby acceptant une *réplication synchrone*, comme décrite dans Section 27.2.8. À tout moment, il y aura au moins un serveur standby synchrone actif ; les transactions en attente de validation seront autorisées à continuer après que les serveurs standbys synchrones auront confirmé la réception des données. Les standbys synchrones sont les serveurs standbys nommés dans cette liste, qui sont à la fois connectés et qui récupèrent les données en temps réel (comme indiqué par l'état `streaming` dans la vue `pg_stat_replication`). Indiquer plus d'un serveur standby synchrone permet une meilleure haute- disponibilité et une meilleure protection contre les pertes de données.

Le nom d'un serveur standby est indiqué dans ce cas au niveau du paramètre `application_name` du standby, tel qu'il est configuré dans l'information de connexion du standby. Dans le cas d'un standby en réplication physique, ceci doit être configuré dans le paramètre `primary_conninfo`. La valeur par défaut est la configuration de cluster_name si configuré, et sinon `walreceiver`. Pour la réplication logique, cela peut se configurer dans l'information de connexion de la souscription, et vaut par défaut le nom de la souscription. Pour les autres consommateurs de flux de réplication, veuillez consulter leur documentation.

Ce paramètre indique une liste de serveurs standbys en utilisant une des deux syntaxes suivantes :

```
[FIRST] nb_sync ( nom_standby [, ...] )
ANY nb_sync ( nom_standby [, ...] )
nom_standby [, ...]
```

où *`num_sync`* est le nombre de standbys synchrones dont les transactions doivent attendre des réponses, et *`nom_standby`* est le nom d'un serveur secondaire (standby). `FIRST` et `ANY` spécifie la méthode pour choisir les serveurs secondaires synchrones dans la liste des serveurs.

Le mot-clé `FIRST`, utilisé avec *`num_sync`*, spécifie une réplication synchrone basée sur la priorité, si bien que chaque validation de transaction attendra jusqu'à ce que les enregistrements des WAL soient répliqués de manière synchrone sur *`num_sync`* serveurs secondaires, choisis en fonction de leur priorités. Par exemple, utiliser la valeur `FIRST 3 (s1, s2, s3, s4)` forcera chaque commit à attendre la réponse de trois serveurs secondaires de plus haute priorité

choisis parmis les serveurs secondaires `s1`, `s2`, `s3` et `s4`. Les noms de serveurs secondaires qui apparaissent avant dans la liste reçoivent des priorités plus importantes et seront pris en considération pour être synchrones. Les autres serveurs secondaires apparaissant plus loin dans cette liste représentent les serveurs secondaire potentiellement synchrones. Si l'un des serveurs secondaires actuellement synchrones se déconnecte pour quelque raison que ce soit, il sera remplacé par le serveur secondaire de priorité la plus proche. Le mot clé `FIRST` est facultatif.

Le mot-clé `ANY`, utilisé avec *`num_sync`*, spécifie une réplication synchrone basée sur un quorum, si bien que chaque validation de transaction attendra jusqu'à ce que les enregistrements des WAL soient répliqués de manière synchrone sur *au moins* *`num_sync`* des serveurs secondaires listés. Par exemple, utiliser la valeur `ANY 3 (s1, s2, s3, s4)` ne bloquera chaque commit que le temps qu'au moins trois des serveurs de la liste `s1`, `s2`, `s3` and `s4` aient répondu, quels qu'ils soient.

`FIRST` et `ANY` sont insensibles à la casse. Si ces mots-clés sont utilisés comme nom d'un serveur secondaire, le paramètre *`standby_name`* doit être entouré de guillemets doubles.

La troisième syntaxe était utilisée avant PostgreSQL version 9.6 est toujours supportée. Cela revient à la nouvelle syntaxe avec `FIRST` et *`num_sync`* égal à 1. Par exemple, `FIRST 1 (s1, s2)` et `s1, s2` ont la même signification : soit `s1` soit `s2` est choisit comme serveur secondaire synchrone.

L'entrée spéciale `*` correspond à tout nom de standby.

Il n'existe pas de mécanisme pour forcer l'unicité des noms de standby. Dans le cas de noms en double, un des standbys concernés sera considéré d'une priorité plus haute mais il n'est pas possible de prévoir lequel.

> **Note**
>
> Chaque *`nom_standby`* doit avoir la forme d'un identifiant SQL valide, sauf si `*` est utilisé. Vous pouvez utiliser des guillemets doubles si nécessaire mais notez que les *`nom_standby`* sont comparés au nom d'application des standbys sans faire attention à la casse, qu'ils aient des guillemets doubles ou non.

Si aucun nom de serveur en standby synchrone n'est indiqué ici, alors la réplication synchrone n'est pas activée et la validation des transactions n'attendra jamais la réplication. Ceci est la configuration par défaut. Même si la réplication synchrone est activée, les transactions individuelles peuvent être configurées pour ne pas avoir à attendre la réplication en configurant le paramètre synchronous_commit à `local` ou `off`.

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`vacuum_defer_cleanup_age` (`integer`)

Indique le nombre de transactions pour lesquelles `VACUUM` et les mises à jour HOT vont différer le nettoyage des versions de lignes mortes. La valeur par défaut est de 0 transactions. Cela signifie que les versions de lignes mortes peuvent être supprimées dès que possible, autrement dit dès qu'elles ne sont plus visibles par les transactions ouvertes. Vous pouvez configurer ce paramètre à une valeur supérieure à 0 sur un serveur primaire qui dispose de serveurs en Hot Standby comme décrit dans Section 27.4. Ceci donne plus de temps aux requêtes des serveur en standby pour qu'elles se terminent sans engendrer de conflits dû à un nettoyage rapide des lignes. Néanmoins, comme la valeur correspond à un nombre de transactions en écriture survenant sur le serveur primaire, il est difficile de prédire le temps additionnel que cela donne aux requêtes exécutées sur le serveur en standby. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

Vous pouvez aussi configurer `hot_standby_feedback` sur les serveurs standby à la place de ce paramètre.

Ceci n'empêche pas le nettoyage des lignes mortes qui ont atteint l'âge spécifié par `old_snapshot_threshold`.

20.6.3. Serveurs secondaire/standby

Ces paramètres contrôlent le comportement d'un serveur en attente pour qu'il puisse recevoir les données de réplication. Leur configuration sur le serveur primaire n'a aucune importance.

`primary_conninfo` (`string`)

Indique une chaîne de connexion à utiliser pour la connexion du serveur standby vers le serveur primaire. Cette chaîne doit être dans le format décrit dans Section 34.1.1. Si une option n'est pas spécifiée dans cette chaîne, alors la variable d'environnement correspondante (voir Section 34.15) est vérifiée. Si la variable d'environnement n'est pas configurée, la valeur par défaut est utilisée.

La chaîne de connexion doit indiquer le nom d'hôte (ou l'adresse) du serveur d'envoi, ainsi que le numéro de port s'il ne s'agit pas du même numéro de port que la valeur par défaut du serveur standby. De plus, indiquez un nom d'utilisateur correspondant à un rôle suffisamment privilégié du serveur d'envoi (voir Section 27.2.5.1). Un mot de passe doit aussi être fourni si le serveur d'envoi réclame une authentification par mot de passe. Il peut être indiqué dans la chaîne `primary_conninfo` ou dans un fichier `~/.pgpass` sur le serveur standby (utiliser `replication` comme nom de base). N'indiquez pas de nom de base dans la chaîne `primary_conninfo`.

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur. Si ce paramètre est modifié alors que le processus walreceiver est en cours d'exécution, le processus reçoit une demandé d'arrêt et sera redémarré avec la nouvelle configuration (sauf si `primary_conninfo` est une chaîne vide). Il n'a pas d'effet si le serveur n'est pas en mode standby.

`primary_slot_name` (`string`)

Indique en option un slot de réplication existant à utiliser lors de la connexion au serveur d'envoi via la réplication en flux pour contrôler la suppression des journaux du serveur d'envoi (voir Section 27.2.6). Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur. Si ce paramètre est modifié alors que le processus walreceiver est en cours d'exécution, le processus reçoit une demandé d'arrêt et sera redémarré avec la nouvelle configuration (sauf si `primary_conninfo` est une chaîne vide). Il n'a pas d'effet si `primary_conninfo` n'est pas configuré ou si le serveur n'est pas en mode standby.

`promote_trigger_file` (`string`)

Indique un fichier trigger dont la présence termine la restauration sur le serveur secondaire. Même si cette valeur n'est pas configurée, vous pouvez toujours promouvoir le standby en utilisant `pg_ctl promote` ou en appelant `pg_promote()`. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`hot_standby` (`boolean`)

Indique si vous pouvez vous connecter et exécuter des requêtes lors de la restauration, comme indiqué dans Section 27.4. Activé par défaut. Ce paramètre peut seulement être configuré au lancement du serveur. Il a un effet seulement lors de la restauration des archives ou en mode serveur en attente.

`max_standby_archive_delay` (`integer`)

Quand le Hot Standby est activé, ce paramètre détermine le temps maximum d'attente que le serveur secondaire doit observer avant d'annuler les requêtes en lecture qui entreraient en conflit

avec des enregistrements des journaux de transactions à appliquer, comme c'est décrit dans Section 27.4.2. `max_standby_archive_delay` est utilisé quand les données de journaux de transactions sont lues à partir des archives de journaux de transactions (et du coup accuse un certain retard par rapport au serveur primaire). Si cette valeur est indiquée sans unité, elle est comprise comme un nombre de millisecondes. La valeur par défaut est de 30 secondes. L'unité est la milliseconde si cette dernière n'est pas spécifiée. Une valeur de -1 autorise le serveur en attente à attendre indéfiniment la fin d'exécution des requêtes en conflit. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

Notez que `max_standby_archive_delay` ne correspond pas au temps d'exécution maximum d'une requête avant son annulation ; il s'agit plutôt du temps maximum autorisé pour enregistrer les données d'un journal de transactions. Donc, si une requête a occasionné un délai significatif au début du traitement d'un journal de transactions, les requêtes suivantes auront un délai beaucoup moins important.

`max_standby_streaming_delay` (`integer`)

Quand Hot Standby est activé, ce paramètre détermine le délai maximum d'attente que le serveur secondaire doit observer avant d'annuler les requêtes en lecture qui entreraient en conflit avec les enregistrements de transactions à appliquer, comme c'est décrit dans Section 27.4.2. `max_standby_streaming_delay` est utilisé quand les données des journaux de données sont reçues via la connexion de la réplication en flux. Si cette valeur est indiquée sans unité, elle est comprise comme un nombre de millisecondes. La valeur par défaut est de 30 secondes. L'unité est la milliseconde si cette dernière n'est pas spécifiée. Une valeur de -1 autorise le serveur en attente à attendre indéfiniment la fin d'exécution des requêtes en conflit. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

Notez que `max_standby_streaming_delay` ne correspond pas au temps d'exécution maximum d'une requête avant son annulation ; il s'agit plutôt du temps maximum autorisé pour enregistrer les données d'un journal de transactions une fois qu'elles ont été récupérées du serveur primaire. Donc, si une requête a occasionné un délai significatif au début du traitement d'un journal de transactions, les requêtes suivantes auront un délai beaucoup moins important.

`wal_receiver_create_temp_slot` (`boolean`)

Indique si le processus walreceiver doit créer un slot de réplication temporaire sur l'instance distante lorsqu'aucun slot de réplication permanent n'a été configuré (en utilisant primary_slot_name). La valeur par défaut est off. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur. Si ce paramètre est modifié alors que le processus walreceiver est en cours d'exécution, alors ce processus reçoit un signal pour s'arrêter et utilisera le nouveau paramètre à son redémarrage.

`wal_receiver_status_interval` (`integer`)

Indique la fréquence minimale pour que le processus de réception (walreceiver) sur le serveur de standby envoie des informations sur la progression de la réplication au serveur en envoi, où elles sont disponibles en utilisant la vue `pg_stat_replication`. Le serveur en standby renvoie la dernière position écrite dans le journal de transactions, la dernière position vidée sur disque du journal de transactions, et la dernière position rejouée. La valeur de ce paramètre est la durée maximale entre les rapports. Les mises à jour sont envoyées à chaque fois que les positions d'écriture ou de vidage ont changées et aussi fréquemment que l'indique ce paramètre s'il est configuré à une valeur autre que zéro. Il existe aussi des cas ou les mises à jour sont envoyées en ignorant ce paramètre. Par exemple, lorsque le traitement du WAL courant se termine ou lorsque `synchronous_commit` est configuré à `remote_apply`. Du coup, la position de rejeu pourrait avoir un certain retard par rapport à la vraie position. Si cette valeur est indiquée sans unité, elle est considérée en secondes. La valeur par défaut est de 10 secondes. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`hot_standby_feedback` (`boolean`)

Spécifie si un serveur en Hot Standby enverra des informations au serveur en envoi sur les requêtes en cours d'exécution sur le serveur en standby. Ce paramètre peut être utilisé pour éliminer les annulations de requêtes nécessaires au nettoyage des enregistrements. Par contre, il peut causer une fragmentation plus importante sur le serveur principal pour certaines charges. Les messages d'informations ne seront pas envoyés plus fréquemment qu'une fois par `wal_receiver_status_interval`. La valeur par défaut est `off`. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

Si la réplication en cascade est utilisée, les informations sont passées à l'émetteur jusqu'à arriver au serveur primaire. Les serveurs en standby ne font aucun usage des informations qu'ils reçoivent, en dehors de les envoyer à leur émetteur des données de réplication.

Ce paramètre ne surcharge pas le comportement de `old_snapshot_threshold` sur le primaire ; une image de la base sur le standby qui dépasse la limite d'âge du primaire peut devenir invalide, résultant en une annulation des transactions sur le standby. Ceci a pour explication que `old_snapshot_threshold` a pour but de fournir une limite absolue sur la durée où des lignes mortes peuvent contribuer à la fragmentation, qui, dans le cas contraire, pourrait être transgressé à cause de la configuration du standby.

`wal_receiver_timeout` (`integer`)

Termine les connexions de réplication qui sont inactives pour plus longtemps que cette durée. Ceci est utile pour que le serveur de réception détecte le crash d'un serveur primaire ou une perte réseau. Si cette valeur est spécifiée sans unité, elle est comprise comme un nombre de millisecondes. La valeur par défaut est de 60 secondes. À zéro, le mécanisme est désactivé. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou en ligne de commande.

`wal_retrieve_retry_interval` (`integer`)

Indique combien de temps le serveur standby doit attendre lorsque les données des WAL ne sont pas disponibles auprès des sources habituelles (réplication en continu, localement à partir de `pg_wal` ou de l'archivage des WAL) avant d'essayer à nouveau de récupérer les WAL. Si cette valeur est spécifiée sans unité, elle est comprise comme un nombre de millisecondes. La valeur par défaut est de 5 secondes. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

Ce paramètre est utile dans les configurations où un nœud en cours de restauration a besoin de contrôler le temps à attendre pour la disponibilité de nouveaux WAL. Par exemple, en mode restauration à partir des archives, il est possible d'avoir une restauration plus réactive dans la détection d'un nouveau fichier WAL en réduisant la valeur de ce paramètre. Sur un système avec une génération faible de WAL, l'augmenter réduit le nombre de requêtes nécessaires pour accèder aux WAL archivés, quelque chose utile par exemple dans les environnements cloud où le nombre de fois où l'infrastructure est accédée est pris en compte.

`recovery_min_apply_delay` (`integer`)

Par défaut, un serveur standby restaure les enregistrements des WAL du serveur d'envoi dès que possible. Il peut être utile d'avoir une copie qui appliquer les données de réplication avec un délai spécifié pour prévenir en cas de perte de données. Ce paramètre vous permet de repousser l'application de la restauration sur une période de temps fixée. Par exemple, si vous configurez ce paramètre à `5min`, le standby ne rejouera chaque validation de transaction seulement quand l'heure système sur le standby est au moins cinq minutes après l'heure de validation rapportée par le primaire. Si cette valeur est spécifiée sans unité, elle est comprise comme un nombre de millisecondes. La valeur par défaut est zéro, pour ne pas ajouter de délai.

Il est possible que le délai de réplication entre les serveurs dépasse la valeur de ce paramètre, auquel cas aucun délai n'est ajouté. Notez que le délai est calculé entre l'horodatage du WAL écrit sur le primaire et l'heure système actuel sur le secondaire. Les délais dans le transfert, à cause d'un retard réseau ou de configuration de réplication en cascade, pourraient réduire le temps d'attente de

façon importante. Si les horloges systèmes du primaire et du secondaire ne sont pas synchronisées, ceci pourrait amener la restauration à appliquer des enregistrements plus rapidement que souhaité. Ceci n'est pas un problème important parce que la configuration intéressante de ce paramètre est bien plus importante que les déviations habituelles d'horloge entre serveurs.

Le délai n'est appliqué que sur les enregistrements WAL des validations de transaction. Les autres enregistrements sont rejoués aussi rapidement que possible, ce qui n'est pas un problème vu que les règles de visibilité avec MVCC nous assurent que les effets ne sont pas visibles tant que l'enregistrement de validation correspondant n'est pas appliqué.

Le délai survient une fois que la base de données en restauration a atteint un point de cohérence, et jusqu'à ce que le standby soit promu. Après cela, le standby arrêtera toute restauration et sans délai.

Les enregistrements des journaux de transactions sont conservés sur le serveur secondaire jusqu'à ce qu'ils soient prêts à être appliqués. De ce fait, des délais plus importants peuvent résulter en une accumulation plus importante de journaux de transactions, augmentant ainsi les besoins en espace disque du répertoire `pg_wal` sur le secondaire.

Ce paramètre vise les déploiements de réplication par flux. Néanmoins, la configuration de ce paramètre sera honorée dans tous les cas, sauf dans le cas de la restauration après un crash. `hot_standby_feedback` se verra imposé le délai, ce qui peut amener de la fragmentation sur le serveur primaire. N'activez les deux qu'avec précaution.

> **Avertissement**
>
> La réplication synchrone est affectée par ce paramétrage quand `synchronous_commit` est configuré à `remote_apply` ; chaque `COMMIT` aura besoin d'attendre son application.

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

20.6.4. Souscripteurs

Ces réglages contrôlent le comportement d'un souscripteur de réplication logique. Leurs valeurs sur le serveur publiant les données est sans importance.

Veuillez noter que les paramètres de configuration `wal_receiver_timeout`, `wal_receiver_status_interval` et `wal_retrieve_retry_interval` affectent également les workers de réplication logique.

`max_logical_replication_workers` (`integer`)

Spécifie le nombre maximal de workers de réplication logique. Cela inclue à la fois les workers ainsi que les workers de synchronisation de table.

Les workers de réplication logique sont pris de la réserve définie par `max_worker_processes`.

La valeur par défaut est 4. Ce paramètre peut seulement être configuré au démarrage du serveur.

`max_sync_workers_per_subscription` (`integer`)

Le nombre maximal de workers de synchronisation par souscription. Ce paramètre contrôle la quantité de parallélisme pour la copie initiale de données durant l'initialisation de la souscription ou quand de nouvelles tables sont ajoutées.

Pour le moment, il ne peut y avoir qu'un seul worker de synchronisation par table.

Les workers de synchronisation sont pris de la réserve définie par `max_logical_replication_workers`.

La valeur par défaut est 2. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

20.7. Planification des requêtes

20.7.1. Configuration de la méthode du planificateur

Ces paramètres de configuration fournissent une méthode brutale pour influencer les plans de requête choisis par l'optimiseur de requêtes. Si le plan choisi par défaut par l'optimiseur pour une requête particulière n'est pas optimal, une solution *temporaire* peut provenir de l'utilisation de l'un de ces paramètres de configuration pour forcer l'optimiseur à choisir un plan différent. De meilleures façons d'améliorer la qualité des plans choisis par l'optimiseur passent par l'ajustement des constantes de coût du planificateur (voir Section 20.7.2), le lancement manuel plus fréquent de ANALYZE, l'augmentation de la valeur du paramètre de configuration default_statistics_target et l'augmentation du nombre de statistiques récupérées pour des colonnes spécifiques en utilisant ALTER TABLE SET STATISTICS.

`enable_async_append` (`boolean`)

Active ou désactive la possibilité pour le planificateur d'utiliser des types de plans d'exécution tenant compte du caractère asynchrone des ajouts. La valeur par défaut est on.

`enable_bitmapscan` (`boolean`)

Active ou désactive l'utilisation des plans de parcours de bitmap (*bitmap-scan*) par le planificateur de requêtes. Activé par défaut (on).

`enable_gathermerge` (`boolean`)

Active ou désactive l'utilisation des plans de type gather merge. La valeur par défaut est on.

`enable_hashagg` (`boolean`)

Active ou désactive l'utilisation des plans d'agrégation hachée (*hashed aggregation*) par le planificateur. Activé par défaut (on).

`enable_hashjoin` (`boolean`)

Active ou désactive l'utilisation des jointures de hachage (*hash-join*) par le planificateur. Activé par défaut (on).

`enable_incremental_sort` (`boolean`)

Active ou désactive l'utilisation des tris incrémentaux par le planificateur. La valeur par défaut est on.

`enable_indexscan` (`boolean`)

Active ou désactive l'utilisation des parcours d'index (*index-scan* et *index-only-scan*) par le planificateur. Activé par défaut (on). Voir aussi enable_indexonlyscan.

`enable_indexonlyscan` (`boolean`)

Active ou désactive l'utilisation des parcours d'index seuls (*index-only-scan*) par le planificateur (voir Section 11.9). Activé par défaut (on).Le paramètre enable_indexscan doit aussi être activé pour que l'optimiseur considère l'utilisation de parcours d'index seuls.

`enable_material` (`boolean`)

Active ou désactive l'utilisation de la matérialisation par le planificateur. Il est impossible de supprimer complètement son utilisation mais la désactivation de cette variable permet d'empêcher le planificateur d'insérer des nœuds de matérialisation sauf dans le cas où son utilisation est obligatoire pour des raisons de justesse de résultat. Activé par défaut (`on`).

`enable_memoize` (`boolean`)

Active ou désactive l'utilisation par le planificateur des résultats des plans d'éxécution en mémoire pour récupérer les résultats des parcours paramétrés dans les jointures de type boucle impriquée (*nested loop join*). Ce type de plans permet aux parcours du plan sous-jacent d'être ignorés quand le résultat des paramètres actuels sont déjà en mémoire. Les résultats les moins fréquents peuvent être sortis de la mémoire lorsqu'il est nécessaire de faire de la place pour de nouvelles données. La valeur par défaut est `on`.

`enable_mergejoin` (`boolean`)

Active ou désactive l'utilisation des jointures de fusion (*merge-join*)par le planificateur. Activé par défaut (`on`).

`enable_nestloop` (`boolean`)

Active ou désactive l'utilisation des jointures de boucles imbriquées (*nested-loop*) par le planificateur. Il n'est pas possible de supprimer complètement les jointures de boucles imbriquées mais la désactivation de cette variable décourage le planificateur d'en utiliser une si d'autres méthodes sont disponibles. Activé par défaut (`on`).

`enable_parallel_append` (`boolean`)

Active ou désactive l'utilisation de plans Append parallélisés. La valeur par défaut est `on`.

`enable_parallel_hash` (`boolean`)

Active ou désactive l'utilisation des plans parallélisés de jointure par hachage. N'a pas d'effet si les plans de jointure par hachage ne sont pas activés. La valeur par défaut est `on`.

`enable_partition_pruning` (`boolean`)

Active ou désactive la capacité du planificateur à éliminer les partitions d'une table partitionnée dans les plans d'exécution. Cela contrôle aussi la capacité du planificateur à générer des plans de requête autorisant l'exécuteur à supprimer (ignorer) les partitions durant l'exécution. Le défaut est `on`. Voir Section 5.11.4 pour les détails.

`enable_partitionwise_join` (`boolean`)

Active ou désactive l'utilisation par le planificateur des jointures entre partitions, qui permettent aux jointures entre tables partitionnées d'être effectuées en joignant les partitions correspondantes. Pour le moment, une jointure entre partitions ne s'applique que si la condition de jointure inclut toutes les clés de partition, qui doivent être du même type et avoir exactement les mêmes ensembles de partitions filles. Avec ce paramètre activé, le nombre de nœuds dont l'utilisation mémoire est restreinte par `work_mem` apparaissant dans le plan final peut augmenter de façon linéaire suivant le nombre de partitions parcourues. Ceci peut engendrer une forte augmentation de la consommation de la mémoire globale lors de l'exécution de la requête. La planification de requête devient aussi significativement plus coûteuse en terme de mémoire et de CPU. La valeur par défaut est `off`.

`enable_partitionwise_aggregate` (`boolean`)

Active ou désactive l'utilisation par le planificateur des regroupements ou agrégations par partition, qui permettent, dans les tables partitionnées, d'exécuter regroupement ou agrégation

séparément pour chaque partition. Si la clause `GROUP BY` n'inclut pas les clés de partition, seule une agrégation partielle peut être effectuée par partition, et la finalisation interviendra plus tard. Avec ce paramètre activé, le nombre de nœuds dont l'utilisation mémoire est restreinte par `work_mem` apparaissant dans le plan final peut augmenter de façon linéaire suivant le nombre de partitions parcourues. Ceci peut engendrer une forte augmentation de la consommation de la mémoire globale lors de l'exécution de la requête. La planification de requête devient aussi significativement plus coûteuse en terme de mémoire et de CPU. La valeur par défaut est `off`.

`enable_seqscan` (`boolean`)

Active ou désactive l'utilisation des parcours séquentiels (*sequential scan*) par le planificateur. Il n'est pas possible de supprimer complètement les parcours séquentiels mais la désactivation de cette variable décourage le planificateur d'n utiliser un si d'autres méthodes sont disponibles. Activé par défaut (`on`).

`enable_sort` (`boolean`)

Active ou désactive l'utilisation des étapes de tri explicite par le planificateur. Il n'est pas possible de supprimer complètement ces tris mais la désactivation de cette variable décourage le planificateur d'en utiliser un si d'autres méthodes sont disponibles. Activé par défaut (`on`).

`enable_tidscan` (`boolean`)

Active ou désactive l'utilisation des parcours de TID par le planificateur. Activé par défaut (`on`).

20.7.2. Constantes de coût du planificateur

Les variables de *coût* décrites dans cette section sont mesurées sur une échelle arbitraire. Seules leurs valeurs relatives ont un intérêt. De ce fait, augmenter ou diminuer leurs valeurs d'un même facteur n'occasione aucun changement dans les choix du planificateur. Par défaut, ces variables de coût sont basées sur le coût de récupération séquentielle d'une page ; c'est-à-dire que `seq_page_cost` est, par convention, positionné à `1.0` et les autres variables de coût sont configurées relativement à cette référence. Il est toutefois possible d'utiliser une autre échelle, comme les temps d'exécution réels en millisecondes sur une machine particulière.

> **Note**
>
> Il n'existe malheureusement pas de méthode bien définie pour déterminer les valeurs idéales des variables de coût. Il est préférable de les considérer comme moyennes sur un jeu complet de requêtes d'une installation particulière. Cela signifie que modifier ces paramètres sur la seule base de quelques expériences est très risqué.

`seq_page_cost` (`floating point`)

Initialise l'estimation faite par le planificateur du coût de récupération d'une page disque incluse dans une série de récupérations séquentielles. La valeur par défaut est 1.0. Cette valeur peut être surchargée pour les tables et index d'un tablespace spécifique en configurant le paramètre du même nom pour un tablespace (voir ALTER TABLESPACE).

`random_page_cost` (`floating point`)

Initialise l'estimation faite par le planificateur du coût de récupération non-séquentielle d'une page disque. Mesurée comme un multiple du coût de récupération d'une page séquentielle, sa valeur par défaut est 4.0. Cette valeur peut être surchargée pour les tables et index d'un tablespace spécifique en configurant le paramètre du même nom pour un tablespace (voir ALTER TABLESPACE).

Réduire cette valeur par rapport à `seq_page_cost` incite le système à privilégier les parcours d'index ; l'augmenter donne l'impression de parcours d'index plus coûteux. Les deux valeurs

peuvent être augmentées ou diminuées concomitament pour modifier l'importance des coûts d'entrées/sorties disque par rapport aux coûts CPU, décrits par les paramètres qui suivent.

Les accès aléatoires sur du stockage permanent sont généralement bien plus coûteux que quatre fois un accès séquentiel. Néanmoins, une valeur plus basse est utilisée (4,0) car la majorité des accès aléatoires au stockage, comme les lectures d'index, est supposée survenir en cache. De plus, la latence des stockages réseaux tend à réduire la surcharge relative des accès aléatoires.

Si vous pensez que le cache est moins fréquent que ce que reflète la valeur par défaut et que la latence réseau est minimale, vous pouvez augmenter la valeur du paramètre random_page_cost pour que cela corresponde mieux au coût réel d'un accès aléatoire. Le stockage qui a un coût d'accès aléatoire plus haut que le séquentiel, typiquement les disques magnétiques, pourrait profiter d'une valeur plus haute de random_page_cost. De même, si vos données tiennent entièrement en cache, par exemple quand la volumétrie de la base est inférieure à la quantité totale de mémoire du serveur ou que la latence réseau est haute, diminuer random_page_cost pourrait être approprié.

> **Astuce**
>
> Bien que le système permette de configurer `random_page_cost` à une valeur inférieure à celle de `seq_page_cost`, cela n'a aucun intérêt. En revanche, les configurer à des valeurs identiques prend tout son sens si la base tient entièrement dans le cache en RAM. En effet, dans ce cas, il n'est pas pénalisant d'atteindre des pages qui ne se suivent pas. De plus, dans une base presque entièrement en cache, ces valeurs peuvent être abaissées relativement aux paramètres CPU car le coût de récupération d'une page déjà en RAM est bien moindre à celui de sa récupération sur disque.

`cpu_tuple_cost` (`floating point`)

Initialise l'estimation faite par le planificateur du coût de traitement de chaque ligne lors d'une requête. La valeur par défaut est 0.01.

`cpu_index_tuple_cost` (`floating point`)

Initialise l'estimation faite par le planificateur du coût de traitement de chaque entrée de l'index lors d'un parcours d'index. La valeur par défaut est 0.005.

`cpu_operator_cost` (`floating point`)

Initialise l'estimation faite par le planificateur du coût de traitement de chaque opérateur ou fonction exécutée dans une requête. La valeur par défaut est 0.0025.

`parallel_setup_cost` (`floating point`)

Configure le coût estimé par l'optimiseur pour le lancement de processus de travail parallèle. La valeur par défaut est 1000.

`parallel_tuple_cost` (`floating point`)

Configure le coût estimé par l'optimiseur pour le transfert d'une ligne d'un processus de travail parallèle à un autre. La valeur par défaut est 0,1.

`min_parallel_table_scan_size` (`integer`)

Spécifie la quantité minimale de donnée de la table qui doit être parcourue pour qu'un parcours parallèle soit envisagé. Pour un parcours séquentiel parallèle, la quantité de données de la table parcourue est toujours égale à la taille de la table, mais quand des index sont utilisés la quantité de données de la table parcourue sera normalement moindre. Si cette valeur est spécifiée sans unité,

elle est comprise comme un nombre de blocs, autrement dit BLCKSZ octets, typiquement 8 Ko. La valeur par défaut est 8 Mo (8MB).

`min_parallel_index_scan_size` (`integer`)

Spécifie la quantité minimale de donnée d'index qui doit être parcourue pour qu'un parcours parallèle soit envisagé. Veuillez noter qu'un parcours d'index parallèle ne touchera en général pas la totalité de l'index ; il s'agit du nombre de page que l'optimisateur pensera réellement toucher durant le parcours qui est important. Ce paramètre est aussi utilisé pour décider si un index particulier peut participer à un vacuum parallélisé. Voir VACUUM. Si cette valeur est spécifiée sans unité, elle est comprise comme un nombre de blocs, autrement dit BLCKSZ octets, typiquement 8 Ko. La valeur par défaut est 512 kilooctets (512kB).

`effective_cache_size` (`integer`)

Initialise l'estimation faite par le planificateur de la taille réelle du cache disque disponible pour une requête. Ce paramètre est lié à l'estimation du coût d'utilisation d'un index ; une valeur importante favorise les parcours d'index, une valeur faible les parcours séquentiels. Pour configurer ce paramètre, il est important de considérer à la fois les tampons partagés de PostgreSQL et la portion de cache disque du noyau utilisée pour les fichiers de données de PostgreSQL, bien que certaines données pourraient être présentes aux deux endroits. Il faut également tenir compte du nombre attendu de requêtes concurrentes sur des tables différentes car elles partagent l'espace disponible. Ce paramètre n'a pas d'influence sur la taille de la mémoire partagée allouée par PostgreSQL, et ne réserve pas non plus le cache disque du noyau ; il n'a qu'un rôle estimatif. Le système ne suppose pas non plus que les données reste dans le cache du disque entre des requêtes. Si cette valeur est indiquée sans unité, elle est pris comme un nombre de blocs, autrement dit BLCKSZ octets, typiquement 8 Ko. La valeur par défaut est de 4 Go (4GB). (Si BLCKSZ n'est pas 8 Ko, les valeurs par défaut changent de façon proportionnée.)

`jit_above_cost` (`floating point`)

Configure le coût de la requête au-dessus duquel la compilation JIT est activée (voir Chapitre 32). Exécuter JIT coûte en temps de planification mais peut accélérer l'exécution de la requête. Configurer ce paramètre à `-1` désactive la compilation JIT. Le défaut est `100000`.

`jit_inline_above_cost` (`floating point`)

Configure le coût de requête au-dessus duquel la compilation JIT tente de mettre à plat fonctions et opérateurs. Ceci ajoute au temps de planification mais peut améliorer la durée d'exécution. Il n'y a pas de sens à le configurer à une valeur inférieure à celle de `jit_above_cost`. Le configurer à `-1` désactive cette mise à plat. Le défaut est `500000`.

`jit_optimize_above_cost` (`floating point`)

Configure le coût de requête au-dessus duquel la compilation JIT utilise aussi les optimisations coûteuses. De telles optimisations ajoutent au temps de planification mais peuvent améliorer la durée d'exécution. Il n'y a pas de sens à le configurer à une valeur inférieure à celle de `jit_above_cost`, et il y a peu d'intérêt à le configurer à une valeur supérieure à `jit_inline_above_cost`. Le configurer à `-1` désactive ces optimisations. Le défaut est `500000`.

20.7.3. Optimiseur génétique de requêtes

L'optimiseur génétique de requête (GEQO) est un algorithme qui fait la planification d'une requête en utilisant une recherche heuristique. Cela réduit le temps de planification pour les requêtes complexes (celles qui joignent de nombreuses relations), au prix de plans qui sont quelques fois inférieurs à ceux trouvés par un algorithme exhaustif. Pour plus d'informations, voir Chapitre 62.

`geqo` (`boolean`)

Active ou désactive l'optimisation génétique des requêtes. Activé par défaut. Il est généralement préférable de ne pas le désactiver sur un serveur en production. La variable `geqo_threshold` fournit un moyen plus granulaire de désactiver le GEQO.

`geqo_threshold` (`integer`)

L'optimisation génétique des requêtes est utilisée pour planifier les requêtes si, au minimum, ce nombre d'éléments est impliqué dans la clause `FROM` (une construction `FULL OUTER JOIN` ne compte que pour un élément du `FROM`). La valeur par défaut est 12. Pour des requêtes plus simples, il est préférable d'utiliser le planificateur standard, à recherche exhaustive. Par contre, pour les requêtes avec un grand nombre de tables, la recherche exhaustive prend trop de temps, souvent plus de temps que la pénalité à l'utilisation d'un plan non optimal. Du coup, une limite sur la taille de la requête est un moyen simple de gérer l'utilisation de GEQO.

`geqo_effort` (`integer`)

Contrôle le compromis entre le temps de planification et l'efficacité du plan de requête dans GEQO. Cette variable est un entier entre 1 et 10. La valeur par défaut est de cinq. Des valeurs plus importantes augmentent le temps passé à la planification de la requête mais aussi la probabilité qu'un plan de requête efficace soit choisi.

`geqo_effort` n'a pas d'action directe ; il est simplement utilisé pour calculer les valeurs par défaut des autres variables influençant le comportement de GEQO (décrites ci-dessous). Il est également possible de les configurer manuellement.

`geqo_pool_size` (`integer`)

Contrôle la taille de l'ensemble utilisé par GEQO. C'est-à-dire le nombre d'individus au sein d'une population génétique. Elle doit être au minimum égale à deux, les valeurs utiles étant généralement comprises entre 100 et 1000. Si elle est configurée à zéro (valeur par défaut), alors une valeur convenable est choisie en fonction de `geqo_effort` et du nombre de tables dans la requête.

`geqo_generations` (`integer`)

Contrôle le nombre de générations utilisées par GEQO. C'est-à-dire le nombre d'itérations de l'algorithme. Il doit être au minimum de un, les valeurs utiles se situent dans la même plage que la taille de l'ensemble. S'il est configuré à zéro (valeur par défaut), alors une valeur convenable est choisie en fonction de `geqo_pool_size`.

`geqo_selection_bias` (`floating point`)

Contrôle le biais de sélection utilisé par GEQO. C'est-à-dire la pression de sélectivité au sein de la population. Les valeurs s'étendent de 1.50 à 2.00 (valeur par défaut).

`geqo_seed` (`floating point`)

Contrôle la valeur initiale du générateur de nombres aléatoires utilisé par GEQO pour sélectionner des chemins au hasard dans l'espace de recherche des ordres de jointures. La valeur peut aller de zéro (valeur par défaut) à un. Varier la valeur modifie l'ensemble des chemins de jointure explorés et peut résulter en des chemins meilleurs ou pires.

20.7.4. Autres options du planificateur

`default_statistics_target` (`integer`)

Initialise la cible de statistiques par défaut pour les colonnes de table pour lesquelles aucune cible de colonne spécifique n'a été configurée via `ALTER TABLE SET STATISTICS`. Des valeurs élevées accroissent le temps nécessaire à l'exécution d'`ANALYZE` mais peuvent permettre

d'améliorer la qualité des estimations du planificateur. La valeur par défaut est 100. Pour plus d'informations sur l'utilisation des statistiques par le planificateur de requêtes, se référer à la Section 14.2.

`constraint_exclusion` (`enum`)

Contrôle l'utilisation par le planificateur de requête des contraintes pour optimiser les requêtes. Les valeurs autorisées de `constraint_exclusion` sont `on` (examiner les contraintes pour toutes les tables), `off` (ne jamais examiner les contraintes) et `partition` (n'examiner les contraintes que pour les tables enfants d'un héritage et pour les sous-requêtes `UNION ALL`). `partition` est la valeur par défaut. C'est souvent utilisé avec les tables héritées pour améliorer les performances.

Quand ce paramètre l'autorise pour une table particulière, le planificateur compare les conditions de la requête avec les contraintes `CHECK` sur la table, et omet le parcourt des tables pour lesquelles les conditions contredisent les contraintes. Par exemple :

```
CREATE TABLE parent(clef integer, ...);
CREATE TABLE fils1000(check (clef between 1000 and 1999))
 INHERITS(parent);
CREATE TABLE fils2000(check (clef between 2000 and 2999))
 INHERITS(parent);
...
SELECT * FROM parent WHERE clef = 2400;
```

Avec l'activation de l'exclusion par contraintes, ce `SELECT` ne parcourt pas `fils1000`, ce qui améliore les performances.

À l'heure actuelle, l'exclusion de contraintes est activée par défaut seulement pour les cas souvent utilisés pour implémenter le partitionnement de tables via les arbres d'héritage. L'activer pour toutes les tables impose une surcharge de planification qui est visible pour de simples requêtes, sans apporter de bénéfices pour ces requêtes. Si vous n'avez pas de tables partitionnées utilisant l'héritage traditionnel, vous pourriez vouloir le désactiver. (Notez que la fonctionnalité équivalente pour les tables partitionnées est contrôlée par un paramètre séparé, enable_partition_pruning.)

Reportez vous à Section 5.11.5 pour plus d'informations sur l'utilisation d'exclusion de contraintes pour implémenter le partitionnement.

`cursor_tuple_fraction` (`floating point`)

Positionne la fraction, estimée par le planificateur, d'enregistrements d'un curseur qui sera récupérée. La valeur par défaut est 0.1. Des valeurs plus petites de ce paramètre rendent le planificateur plus enclin à choisir des plans à démarrage rapide (« fast start »), qui récupèreront les premiers enregistrements rapidement, tout en mettant peut être un temps plus long à récupérer tous les enregistrements. Des valeurs plus grandes mettent l'accent sur le temps total estimé. À la valeur maximum 1.0 du paramètre, les curseurs sont planifiés exactement comme des requêtes classiques, en ne prenant en compte que le temps total estimé et non la vitesse à laquelle les premiers enregistrements seront fournis.

`from_collapse_limit` (`integer`)

Le planificateur assemble les sous-requêtes dans des requêtes supérieures si la liste `FROM` résultante contient au plus ce nombre d'éléments. Des valeurs faibles réduisent le temps de planification mais conduisent à des plans de requêtes inférieurs. La valeur par défaut est de 8. Pour plus d'informations, voir Section 14.3.

Configurer cette valeur à geqo_threshold ou plus pourrait déclencher l'utilisation du planificateur GEQO, ce qui pourrait aboutir à la génération de plans non optimaux. Voir Section 20.7.3.

`jit` (`boolean`)

Détermine si la compilation JIT peut être utilisée par PostgreSQL, quand elle est disponible (voir Chapitre 32). La valeur par défaut est `on`.

`join_collapse_limit` (`integer`)

Le planificateur réécrit les constructions `JOIN` explicites (à l'exception de `FULL JOIN`) en une liste d'éléments `FROM` à chaque fois qu'il n'en résulte qu'une liste ne contenant pas plus de ce nombre d'éléments. Des valeurs faibles réduisent le temps de planification mais conduisent à des plans de requêtes inférieurs.

Par défaut, cette variable a la même valeur que `from_collapse_limit`, valeur adaptée à la plupart des utilisations. Configurer cette variable à 1 empêche le réordonnancement des `JOIN`tures explicites. De ce fait, l'ordre des jointures explicites indiqué dans la requête est l'ordre réel dans lequel les relations sont jointes. Le planificateur de la requête ne choisit pas toujours l'ordre de jointure optimal ; les utilisateurs aguerris peuvent choisir d'initialiser temporairement cette variable à 1 et d'indiquer explicitement l'ordre de jointure souhaité. Pour plus d'informations, voir Section 14.3.

Configurer cette valeur à geqo_threshold ou plus pourrait déclencher l'utilisation du planificateur GEQO, ce qui pourrait aboutir à la génération de plans non optimaux. Voir Section 20.7.3.

`recursive_worktable_factor` (`floating point`)

Configure l'estimation du planificateur sur la taille moyenne de la table de travail d'une requête récursive, sous la forme d'un multiple de la taille estimée du terme initial non récursif de la requête. Ceci aide le planificateur à choisir la méthode la plus appropriée pour joindre la table de travail avec les autres tables de la requête. La valeur par défaut est `10.0`. Une valeur plus petite, telle que `1.0`, peut être utile quand la récursion a un petit « fan-out » d'une étape à une autre, comme par exemple dans les requêtes au chemin court. Les requêtes d'analyse de graphes pourraient bénéficier de valeurs plus importantes que celle par défaut.

`plan_cache_mode` (`enum`)

Les requêtes préparées (soit explicitement préparées soit implicitement générées, par exemple dans PL/pgSQL) peuvent être exécutées en utilisant des plans personnalisés ou génériques. Les plans personnalisés sont de nouveau créés à chaque exécution en utilisant son ensemble spécifique de valeurs de paramètres, alors que les plans génériques ne se basent pas sur les valeurs des paramètres et peuvent être ré-utilisés au fil des exécutions. De ce fait, l'utilisation d'un plan générique permet d'éviter de gâcher du temps de planification mais si le plan idéal dépend fortement des valeurs de paramètres, alors un plan générique pourrait être inefficace. Le choix entre ces options est généralement fait automatiquement mais il peut être forcé avec le paramètre `plan_cache_mode`. Les valeurs autorisées sont `auto` (valeur par défaut), `force_custom_plan` et `force_generic_plan`. Ce paramètre est considéré quand un plan en cache va être exécuté, et non pas quand il va être préparé. Pour plus d'informations, voir PREPARE.

20.8. Remonter et tracer les erreurs

20.8.1. Où tracer

`log_destination` (`string`)

PostgreSQL supporte plusieurs méthodes pour la journalisation des messages du serveur, dont stderr, csvlog, jsonlog et syslog. Sur Windows, eventlog est aussi supporté. Ce paramètre se configure avec la liste des destinations souhaitées séparées par des virgules. Par défaut, les traces ne sont dirigées que vers stderr. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

Si csvlog est la valeur de `log_destination`, les entrées du journal applicatif sont enregistrées dans le format CSV (« comma separated value »), ce qui est bien pratique pour les charger dans des programmes. Voir Section 20.8.4 pour les détails. logging_collector doit être activé pour produire des journaux applicatifs au format CSV.

Si jsonlog est inclus dans `log_destination`, les enregistrements des traces sont faits dans le format JSON, ce qui peut être adéquat pour le chargement des fichiers de traces dans des programmes. Voir Section 20.8.5 pour les détails. logging_collector doit être activé pour générer des traces au format JSON.

Quand soit stderr, soit csvlog, soit jsonlog sont inclus, le fichier `current_logfiles` est créé pour enregistrer l'emplacement des fichiers de trace actuellement utilisés par le récupérateur des traces, ainsi que la destination associée des traces. Ceci fournit un moyen agréable pour trouver les fichiers en cours d'utilisation par l'instance. Voici un exemple du contenu de ce fichier :

```
stderr log/postgresql.log
 csvlog log/postgresql.csv
+jsonlog log/postgresql.json
```

`current_logfiles` est recréé quand un nouveau fichier de trace est crée du à une rotation, et quand `log_destination` est rechargé. Il est supprimé quand ni stderr ni csvlog ni jsonlog ne sont inclus dans `log_destination`, et quand le collecteur de traces es désactivé.

> **Note**
>
> Sur la plupart des systèmes Unix, il est nécessaire de modifier la configuration du démon syslog pour utiliser l'option syslog de `log_destination`. PostgreSQL peut tracer dans les niveaux syslog `LOCAL0` à `LOCAL7` (voir syslog_facility) mais la configuration par défaut de syslog sur la plupart des plateformes ignore de tels messages. Il faut ajouter une ligne similaire à :
>
> ```
> local0.*    /var/log/postgresql
> ```
>
> dans le fichier de configuration de syslog pour obtenir ce type de journalisation.
>
> Sur Windows, quand vous utilisez l'option `eventlog` pour `log_destination`, vous devez enregistrer une source d'événement et sa bibliothèque avec le système d'exploitation, pour que le visualisateur des événements Windows puisse affiche correctement les traces. Voir Section 19.12 pour les détails.

`logging_collector` (`boolean`)

Ce paramètre active le collecteur de traces (*logging collector*), qui est un processus en tâche de fond capturant les traces envoyées sur stderr et les enregistrant dans des fichiers. Cette approche est souvent plus utile que la journalisation avec syslog, car certains messages peuvent ne pas apparaître dans syslog. (Un exemple standard concerne les messages d'échec de l'édition dynamique ; un autre concerne les messages d'erreurs produits par les scripts comme `archive_command`.). Ce paramètre ne peut être configuré qu'au lancement du serveur.

> **Note**
>
> Il est possible de tracer sur stderr sans utiliser le collecteur de traces. Les messages iront à l'endroit où est redirigé la sortie des erreurs (stderr) du système. Néanmoins, cette méthode est seulement acceptable pour les petits volumes de traces car il ne fournit pas de moyens

corrects pour gérer la rotation des fichiers de traces. Ainsi, sur certaines plateformes n'utilisant pas le collecteur des traces, cela peut avoir pour résultat la perte ou la corruption des traces, notamment si plusieurs processus écrivent en même temps dans le même fichier de traces, écrasant ainsi les traces des autres processus.

Note

Le collecteur des traces est conçu pour ne jamais perdre de messages. Cela signifie que, dans le cas d'une charge extrêmement forte, les processus serveur pourraient se trouver bloqués lors de l'envoi de messages de trace supplémentaires. Le collecteur pourrait accumuler dans ce cas du retard. syslog préfère supprimer des messages s'il ne peut pas les écrire. Il pourrait donc ne pas récupérer certains messages dans ces cas mais il ne bloquera pas le reste du système.

`log_directory` (`string`)

Lorsque `logging_collector` est activé, ce paramètre détermine le répertoire dans lequel les fichiers de trace sont créés. Il peut s'agir d'un chemin absolu ou d'un chemin relatif au répertoire des données du cluster. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. La valeur par défaut est `log`.

`log_filename` (`string`)

Lorsque `logging_collector` est activé, ce paramètre indique les noms des journaux applicatifs créés. La valeur est traitée comme un motif `strftime`. Ainsi les échappements % peuvent être utilisés pour indiquer des noms de fichiers horodatés. (S'il y a des échappements % dépendant des fuseaux horaires, le calcul se fait dans le fuseau précisé par log_timezone.) Les échappements % supportés sont similaires à ceux listés dans la spécification de strftime[1] par l'Open Group. Notez que la fonction `strftime` du système n'est pas utilisée directement, ce qui entraîne que les extensions spécifiques à la plateforme (non-standard) ne fonctionneront pas.

Si vous spécifiez un nom de fichier sans échappements, vous devriez prévoir d'utiliser un utilitaire de rotation des journaux pour éviter le risque de remplir le disque entier. Dans les versions précédentes à 8.4, si aucun échappement % n'était présent, PostgreSQL aurait ajouté l'epoch de la date de création du nouveau journal applicatif mais ce n'est plus le cas.

Si la sortie au format CSV est activée dans `log_destination`, `.csv` est automatiquement ajouté au nom du journal horodaté. (Si `log_filename` se termine en `.log`, le suffixe est simplement remplacé.)

Si la sortie en format JSON est activée dans `log_destination`, `.json` sera ajouté au nom du fichier de trace pour créer le nom du fichier au format JSON. (Si `log_filename` se termine en `.log`, le suffise est remplacé.)

Ce paramètre ne peut être positionné que dans le fichier `postgresql.conf` ou en ligne de commande. La valeur par défaut est `postgresql-%Y-%m-%d_%H%M%S.log`.

`log_file_mode` (`integer`)

Sur les systèmes Unix, ce paramètre configure les droits pour les journaux applicatifs quand `logging_collector` est activé. (Sur Microsoft Windows, ce paramètre est ignoré.) La valeur de ce paramètre doit être un mode numérique spécifié dans le format accepté par les appels systèmes `chmod` et `umask`. (Pour utiliser le format octal, ce nombre doit être précédé d'un zéro, `0`.)

[1] https://pubs.opengroup.org/onlinepubs/009695399/functions/strftime.html

Les droits par défaut sont `0600`, signifiant que seul l'utilisateur qui a lancé le serveur peut lire ou écrire les journaux applicatifs. Un autre paramétrage habituel est `0640`, permettant aux membres du groupe propriétaire de lire les fichiers. Notez néanmoins que pour utiliser ce paramètre, vous devez modifier log_directory pour enregistrer les fichiers en dehors du répertoire des données de l'instance. Dans ce cas, il est déconseillé de rendre les journaux applicatifs lisibles par tout le monde car ils pourraient contenir des données sensibles.

Ce paramètre ne peut être positionné que dans le fichier `postgresql.conf` ou en ligne de commande.

`log_rotation_age` (`integer`)

Lorsque `logging_collector` est activé, ce paramètre détermine la durée maximale pour utiliser un fichier de trace individuel, après lequel un nouveau fichier de trace sera créé. Si cette valeur est indiquée sans unité, elle sera compris comme un nombre de minutes. La valeur par défaut est de 24 heures. Configurer à zéro désactive la création de nouveaux fichiers de trace en se basant sur la date et heure. Ce paramètre ne peut qu'être configuré dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`log_rotation_size` (`integer`)

Lorsque `logging_collector` est activé, ce paramètre détermine la taille maximale (en kilooctets) d'un journal individuel. Après cette quantité de données a été émise dans un fichier de trace, un nouveau fichier de trace sera créé. Si cette valeur est spécifiée sans unité, elle est comprise comme un nombre de Ko. La valeur par défaut est de 10 Mo. Configurer à zéro désactive la création de nouveaux fichiers de trace en se basant sur la taille. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`log_truncate_on_rotation` (`boolean`)

Lorsque `logging_collector` est activé, ce paramètre impose à PostgreSQL de vider (écraser), plutôt qu'ajouter à, tout fichier journal dont le nom existe déjà. Toutefois, cet écrasement ne survient qu'à partir du moment où un nouveau fichier doit être ouvert du fait d'une rotation par temps compté, et non pas à la suite du démarrage du serveur ou d'une rotation par taille comptée. Si ce paramètre est désactivé (off), les traces sont, dans tous les cas, ajoutées aux fichiers qui existent déjà.

Par exemple, si ce paramètres est utilisé en combinaison avec un `log_filename` tel que `postgresql-%H.log`, il en résulte la génération de 24 journaux (un par heure) écrasés de façon cyclique.

Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

Exemple : pour conserver sept jours de traces, un fichier par jour nommé `server_log.Mon`, `server_log.Tue`, etc. et écraser automatiquement les traces de la semaine précédente avec celles de la semaine courante, on positionne `log_filename` à `server_log.%a`, `log_truncate_on_rotation` à `on` et `log_rotation_age` à `1440`.

Exemple : pour conserver 24 heures de traces, un journal par heure, toute en effectuant la rotation plus tôt si le journal dépasse 1 Go, on positionne `log_filename` à `server_log.%H%M`, `log_truncate_on_rotation` à `on`, `log_rotation_age` à `60` et `log_rotation_size` à `1000000`. Inclure `%M` dans `log_filename` permet à toute rotation par taille comptée qui survient d'utiliser un nom de fichier distinct du nom initial horodaté.

`syslog_facility` (`enum`)

Lorsque les traces syslog sont activées, ce paramètre fixe le niveau (« facility ») utilisé par syslog. Les différentes possibilités sont `LOCAL0`, `LOCAL1`, `LOCAL2`, `LOCAL3`, `LOCAL4`, `LOCAL5`, `LOCAL6`, `LOCAL7` ; `LOCAL0` étant la valeur par défaut. Voir aussi la documentation du démon

syslog du serveur. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`syslog_ident` (`string`)

Si syslog est activé, ce paramètre fixe le nom du programme utilisé pour identifier les messages PostgreSQL dans les traces de syslog. La valeur par défaut est `postgres`. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`syslog_sequence_numbers` (`boolean`)

Lorsque les traces ont pour destination syslog et que ce paramètre vaut on (c'est la valeur par défaut), alors chaque message est préfixé par un numéro de séquence en constante augmentation (par exemple `[2]`). Ceci permet d'éviter la suppression du type « --- last message repeated N times --- » qu'un grand nombre d'implémentations de syslog réalisent par défaut. Dans les implémentations plus modernes de syslog, la suppression des messages répétés peut être configurée (par exemple, `$RepeatedMsgReduction` dans rsyslog), ce paramètre pourrait ne plus être nécessaire. De plus, vous pouvez désactiver cette fonction si vous voulez vraiment supprimer des messages répétés

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`syslog_split_messages` (`boolean`)

Lorsque les traces ont pour destination syslog, ce paramètre détermine comment les messages sont délivrées à syslog. Si ce paramètre vaut on (ce qui correspond à la valeur par défaut), les messages sont divisés en ligne, et les longues lignes sont divisées pour qu'elles tiennent sur 1024 octets, qui est la limite typique en taille pour les implémentations syslog traditionnelles. Si ce paramètre est à off, les messages du serveur PostgreSQL sont délivrés au service syslog tel quel, et c'est au service syslog de se débrouiller avec les messages potentiellement gros.

Si syslog enregistre au final les messages dans un fichier texte, alors l'effet sera le même de toute façon et il est préférable de laisser ce paramètre à la valeur on car la plupart des implémentations syslog ne peuvent pas gérer de grands messages ou auraient besoin d'être configurés spécialement pour les gérer. Si syslog écrit au final dans un autre média, il pourrait être nécessaire ou utile de conserver les messages dans un ensemble logique.

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`event_source` (`string`)

Si la journalisation applicative se fait au travers du journal des événements (event log), ce paramètre détermine le nom du programme utilisé pour identifier les messages de PostgreSQL dans la trace. La valeur par défaut est `PostgreSQL`. Ce paramètre peut seulement être configuré au démarrage du serveur.

20.8.2. Quand tracer

`log_min_messages` (`enum`)

Contrôle les niveaux de message écrits dans les traces du serveur. Les valeurs valides sont `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` et `PANIC`. Chaque niveau inclut tous les niveaux qui le suivent. Plus on progresse dans la liste, plus le nombre de messages envoyés est faible. `WARNING` est la valeur par défaut. `LOG` a ici une portée différente de celle de client_min_messages. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` appropriés peuvent modifier la valeur de ce paramètre.

`log_min_error_statement` (`enum`)

Contrôle si l'instruction SQL à l'origine d'une erreur doit être enregistrée dans les traces du serveur. L'instruction SQL en cours est incluse dans les traces pour tout message de sévérité indiquée ou supérieure. Les valeurs valides sont `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL` et `PANIC`. `ERROR` est la valeur par défaut, ce qui signifie que les instructions à l'origine d'erreurs, de messages applicatifs, d'erreurs fatales ou de paniques sont tracées. Pour réellement désactiver le traçage des instructions échouées, ce paramètre doit être positionné à `PANIC`. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` appropriés peuvent modifier la valeur de ce paramètre.

`log_min_duration_statement` (`integer`)

Trace la durée de toute instruction terminée dont le temps d'exécution égale ou dépasse cette durée. Par exemple, si le paramètre est positionné à `250ms`, alors toutes les instructions SQL dont la durée est supérieure ou égale à 250 ms sont tracées. Il est utile d'activer ce paramètre pour tracer les requêtes non optimisées des applications. Si cette valeur est indiquée sans unité, elle est considérée comme un nombre de millisecondes. La configurer à zéro affiche la durée de toutes les requêtes. `-1` (valeur par défaut) désactive la trace de la durée des requêtes. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` appropriés peuvent modifier cette configuration.

Ceci surcharge log_min_duration_sample, ceci signifiant que les requêtes dont la durée dépasse cette configuration ne sont pas sujet à l'échantillonnage et sont toujours tracées.

Pour les clients utilisant le protocole de requêtage étendu, les durées des étapes Parse (analyse), Bind (lien) et Execute (exécution) sont tracées indépendamment.

> **Note**
>
> Lorsque cette option est utilisée avec log_statement, le texte des instructions tracées du fait de `log_statement` n'est pas répété dans le message de trace de la durée. Si syslog n'est pas utilisé, il est recommandé de tracer le PID ou l'ID de session à l'aide de log_line_prefix de façon à pouvoir lier le message de l'instruction au message de durée par cet identifiant.

`log_min_duration_sample` (`integer`)

Permet l'échantillonnage de la durée des requêtes traitées pour celles qui durent au moins la durée indiquée. Ceci produit le même type de traces que log_min_duration_statement, mais seulement pour un sous-ensemble des requêtes exécutées, avec un taux d'échantillonnage contrôlé avec le paramètre log_statement_sample_rate. Par exemple, si vous voulez le configurer à `100ms`, alors toutes les requêtes SQL qui durent 100 millisecondes ou plus seront considérées pour l'échantillonnage. Activer ce paramètre peut être utile quand le trafic est trop important pour tracer toutes les requêtes. Si cette valeur est indiquée sans unité, elle est considérée comme un nombre de milltisecondes. Le configurer à zéro échantillonne toutes les durées de requêtes. La valeur `-1` (valeur par défaut) désactive l'échantillonnage des durées des requêtes. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` appropriés peuvent modifier ce paramètre.

Ce paramètre a une priorité plus basse que `log_min_duration_statement`, signifiant que les requêtes dont la durée dépasse `log_min_duration_statement` ne sont pas sujets à l'échantillonnage et seront toujours tracées.

D'autres notes pour `log_min_duration_statement` s'appliquent aussi à ce paramètre.

`log_statement_sample_rate` (`floating point`)

Détermine la fraction de requêtes dont la durée dépasse log_min_duration_sample et qui seront tracées. L'échantillonnage est stochastique, par exemple `0.5` signifie qu'il y a une chance sur qu'une requête soit tracée. La valeur par défaut est `1.0`, signifiant toutes les requêtes. Le configurer à zéro désactive la trace des durées des requêtes, tout comme configurer le paramètre

`log_min_duration_sample` à `-1`. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` appropriés peuvent modifier ce paramètre.

`log_transaction_sample_rate` (`floating point`)

Initialise la fraction de transactions dont les requêtes seront toutes tracées en plus des requêtes tracées pour d'autres raisons. Cela s'applique à toute nouvelle transaction, quelque soit les durées des requêtes. L'échantillonnage est stochastique, par exemple `0.1` signifie qu'il y a statistiquement une chance sur dix qu'une transaction donnée soit tracée. `log_transaction_sample_rate` peut être utilisé pour construire un échantillon de transactions. La valeur par défaut (`0`) signifie qu'aucune requête d'une transaction supplémentaire ne sera tracée. Positionner ce paramètre à `1` permet de tracer toutes les requêtes de toutes les transactions. Seuls les superutilisateurs peuvent changer ce paramètre.

> **Note**
>
> Comme toutes les options sur les traces des requêtes, cette option peut ajouter une surcharge importante.

`log_startup_progress_interval` (`integer`)

Configure la durée après laquelle le processus de démarrage tracera un message sur l'opération longue toujours en cours, ainsi que l'intervalle entre les messages de progression*. La valeur par défaut est de 10 secondes. Une configuration à `0` désactive cette fonctionnalité. Si cette valeur est indiquée sans unité, elle est considérée comme étant un nombre de millisecondes. Cette configuration est appliquée séparément pour chaque opération. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` et sur la ligne de commande du serveur.

Par exemple, si la synchronisation du répertoire de données prends 25 secondes et qu'après cela, la réinitialisation des relations non journalisées prend 8 secondes et que ce paramètre a sa valeur par défaut de 10 secondes, alors un message sera tracé pour la synchronisation du répertoire de données une fois que cette étape est en cours depuis plus de 10 secondes, puis de nouveau au bout de 20 secondes, mais rien ne sera tracé pour les opérations de réinitialisation des relations non journalisées.

Tableau 20.2 explique les niveaux de sévérité des messages utilisés par PostgreSQL. Si la journalisation est envoyée àsyslog ou à l'eventlog de Windows, les niveaux de sévérité sont traduits comme indiqué ci-dessous.

Tableau 20.2. Niveaux de sévérité des messages

Sévérité	Usage	syslog	eventlog
`DEBUG1 .. DEBUG5`	Fournit des informations successivement plus détaillées à destination des développeurs.	`DEBUG`	`INFORMATION`
`INFO`	Fournit des informations implicitement demandées par l'utilisateur, par exemple la sortie de `VACUUM VERBOSE`.	`INFO`	`INFORMATION`
`NOTICE`	Fournit des informations éventuellement utiles aux utilisateurs, par exemple la troncature des identifiants longs.	`NOTICE`	`INFORMATION`
`WARNING`	Fournit des messages d'avertissement sur d'éventuels problèmes. Par	`NOTICE`	`WARNING`

Sévérité	Usage	syslog	eventlog
	exemple, un COMMIT en dehors d'un bloc de transaction.		
ERROR	Rapporte l'erreur qui a causé l'annulation de la commande en cours.	WARNING	ERROR
LOG	Rapporte des informations à destination des administrateurs. Par exemple, l'activité des points de vérification.	INFO	INFORMATION
FATAL	Rapporte l'erreur qui a causé la fin de la session en cours.	ERR	ERROR
PANIC	Rapporte l'erreur qui a causé la fin de toutes les sessions.	CRIT	ERROR

20.8.3. Que tracer

> **Note**
>
> Ce que vous choisissez de tracer peut avoir des implications sur la sécurité ; voir Section 25.3.

`application_name` (`string`)

Le paramètre `application_name` peut être n'importe quelle chaîne de caractères inférieure à `NAMEDATALEN` caractères (64 caractères après une compilation standard). Il est typiquement configuré lors de la connexion d'une application au serveur. Le nom sera affiché dans la vue `pg_stat_activity` et inclus dans les traces du journal au format CSV. Il peut aussi être inclus dans les autres formats de traces en configurant le paramètre log_line_prefix. Tout caractère ASCII affichable peut être utilisé. Les autres caractères seront remplacés par des points d'interrogation (?).

`debug_print_parse` (`boolean`)
`debug_print_rewritten` (`boolean`)
`debug_print_plan` (`boolean`)

Ces paramètres activent plusieurs sorties de débogage. Quand positionnés, il affichent l'arbre d'interprétation résultant, la sortie de la réécriture de requête, ou le plan d'exécution pour chaque requête exécutée. Ces messages sont émis au niveau de trace `LOG` , par conséquent ils apparaîtront dans le journal applicatif du serveur, mais ne seront pas envoyés au client. Vous pouvez changer cela en ajustant client_min_messages et/ou log_min_messages. Ces paramètres sont désactivés par défaut.

`debug_pretty_print` (`boolean`)

Quand positionné, `debug_pretty_print` indente les messages produits par `debug_print_parse`, `debug_print_rewritten`, ou `debug_print_plan`. Le résultat est une sortie plus lisible mais plus verbeuse que le format « compact » utilisé quand ce paramètre est à off. La valeur par défaut est 'on'.

`log_autovacuum_min_duration` (`integer`)

Permet de tracer chaque action de l'autovacuum dans les journaux de PostgreSQL si celui-ci tourne pour une durée supérieure à la valeur spécifiée. Configurer ce paramètre à zéro permet de tracer toutes les actions de l'autovacuum. La valeur `-1` désactive cette fonctionnalité. Si la valeur ne comporte pas d'unité, elle est comprise en millisecondes. Par exemple, si la valeur `250ms` est

fournie, toutes les actions des opérations automatiques de vacuum et d'analyze durant plus de 250ms seront tracées dans les journaux de PostgreSQL. De plus, lorsque ce paramètre prend une valeur différente de `-1`, un message sera inscrit dans les journaux de PostgreSQL si une action d'autovacuum doit être omise à cause d'un conflit de verrou ou parce qu'une relation a été supprimée entretemps. La valeur par défaut est `10min`. Activer ce paramètre peut aider à mieux comprendre les activités de l'autovacuum. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou sur la ligne de commande, mais le paramétrage peut être surchargé pour chaque table au niveau des paramètres de stockage de chaque table.

`log_checkpoints` (`boolean`)

Trace les points de vérification and restartpoints dans les journaux applicatifs. Diverses statistiques sont incluses dans les journaux applicatifs, dont le nombre de tampons écrits et le temps passé à les écrire. Activé par défaut, ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`log_connections` (`boolean`)

Trace chaque tentative de connexion sur le serveur, ainsi que la réussite de l'authentification du client (si nécessaire) et son autorisation. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier ce paramètre au démarrage d'une session, et il ne peut pas être changé du tout à l'intérieur d'une session. La valeur par défaut est `off`.

> **Note**
>
> Quelques programmes clients, comme psql, tentent de se connecter deux fois pour déterminer si un mot de passe est nécessaire, des messages « connection received » dupliqués n'indiquent donc pas forcément un problème.

`log_disconnections` (`boolean`)

Entraîne l'enregistrement dans les traces du serveur de la fin des sessions. Les sorties des traces fournissent une information similaire à `log_connections`, plus la durée de la session. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier ce paramètre au démarrage d'une session, et il ne peut pas être changé du tout à l'intérieur d'une session. La valeur par défaut est `off`.

`log_duration` (`boolean`)

Trace la durée de toute instruction exécutée. Désactivé par défaut (`off`), seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier ce paramètre.

Pour les clients utilisant le protocole de requêtage étendu, les durées des étapes Parse (analyse), Bind (lien) et Execute (exécution) sont tracées indépendamment.

> **Note**
>
> À la différence de log_min_duration_statement, `log_duration` ne force pas le traçage du texte des requêtes. De ce fait, si `log_duration` est activé (`on`) et que `log_min_duration_statement` a une valeur positive, toutes les durées sont tracées mais le texte de la requête n'est inclus que pour les instructions qui dépassent la limite. Ce comportement peut être utile pour récupérer des statistiques sur les installations à forte charge.

`log_error_verbosity` (`enum`)

Contrôle la quantité de détails écrit dans les traces pour chaque message tracé. Les valeurs valides sont `TERSE`, `DEFAULT` et `VERBOSE`, chacun ajoutant plus de champs aux messages affichés. `TERSE` exclut des traces les informations de niveau `DETAIL`, `HINT`, `QUERY` et `CONTEXT`. La sortie `VERBOSE` inclut le code d'erreur `SQLSTATE` (voir aussi Annexe A), le nom du code source, le nom de la fonction et le numéro de la ligne qui a généré l'erreur. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier ce paramètre.

`log_hostname` (`boolean`)

Par défaut, les traces de connexion n'affichent que l'adresse IP de l'hôte se connectant. Activer ce paramètre permet de tracer aussi le nom de l'hôte. En fonction de la configuration de la résolution de nom d'hôte, les performances peuvent être pénalisées. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`log_line_prefix` (`string`)

Il s'agit d'une chaîne de style `printf` affichée au début de chaque ligne de trace. Les caractères `%` débutent des « séquences d'échappement » qui sont remplacées avec l'information de statut décrite ci-dessous. Les échappement non reconnus sont ignorés. Les autres caractères sont copiés directement dans la trace. Certains échappements ne sont reconnus que par les processus de session et seront traités comme vide par les processus en tâche de fond tels que le processus principal du serveur. L'information de statut pourrait être alignée soit à gauche soit à droite en indiquant un nombre après le signe pourcent et avant l'option. Une valeur négative implique un alignement à droite par ajout d'espaces alors qu'une valeur positive est pour un alignement à gauche. L'alignement peut être utile pour aider à la lecture des fichiers de trace.

Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande. La valeur par défaut est `'%m [%p] '` ce qui affiche dans les trace l'heure courante ainsi que l'identifiant de processus.

Échappement	Produit	Session seule
`%a`	Nom de l'application	yes
`%u`	Nom de l'utilisateur	oui
`%d`	Nom de la base de données	oui
`%r`	Nom ou adresse IP de l'hôte distant et port distant	oui
`%h`	Nom d'hôte distant ou adresse IP	oui
`%b`	Type de processus serveur	no
`%p`	ID du processus	non
`%P`	L'ID du processus leader du groupe de parallélisation, si ce processus est parallélisé	no
`%t`	Estampille temporelle sans millisecondes	non
`%m`	Estampille temporelle avec millisecondes	non
`%n`	Estampille temporelle avec millisecondes (sous la forme d'un epoch Unix)	non
`%i`	Balise de commande : type de commande	oui
`%e`	code d'erreur correspondant à l'état SQL	no
`%c`	ID de session : voir ci-dessous	non
`%l`	Numéro de la ligne de trace de chaque session ou processus, commençant à 1	non
`%s`	Estampille temporelle du lancement du processus	oui
`%v`	Identifiant virtuel de transaction (backendID/localXID)	no
`%x`	ID de la transaction (0 si aucune affectée)	non

Échappement	Produit	Session seule
%q	Ne produit aucune sortie, mais indique aux autres processus de stopper à cet endroit de la chaîne. Ignoré par les processus de session.	non
%Q	Identifiant de la requête actuelle. Les identifiants de requête ne sont pas calculés par défaut, ce champ vaudra donc zéro à moins que le paramètre compute_query_id ne soit activé ou qu'un module externe qui calcule ce champ ne soit utilisé.	oui
%%	%	non

Le type de processus serveur correspond à la colonne `backend_type` dans la vue `pg_stat_activity`, mais des types supplémentaires, non visibles dans cette vue, peuvent apparaître dans les traces.

L'échappement `%c` affiche un identifiant de session quasi-unique constitué de deux nombres hexadécimaux sur quatre octets (sans les zéros initiaux) et séparés par un point. Les nombres représentent l'heure de lancement du processus et l'identifiant du processus, `%c` peut donc aussi être utilisé comme une manière de raccourcir l'affichage de ces éléments. Par exemple, pour générer l'identifiant de session à partir de `pg_stat_activity`, utilisez cette requête :

```
SELECT to_hex(trunc(EXTRACT(EPOCH FROM backend_start))::integer)
 || '.' ||
       to_hex(pid)
FROM pg_stat_activity;
```

Astuce

Si `log_line_prefix` est différent d'une chaîne vide, il est intéressant d'ajouter une espace en fin de chaîne pour créer une séparation visuelle avec le reste de la ligne. Un caractère de ponctuation peut aussi être utilisé.

Astuce

syslog produit ses propres informations d'horodatage et d'identifiant du processus. Ces échappements n'ont donc que peu d'intérêt avec syslog.

Astuce

L'échappement `%q` est utile quand des informations qui ne sont disponibles que dans le contexte d'une session (processus client) est utilisé, comme le nom de l'utilisateur ou de la base. Par exemple :

```
log_line_prefix = '%m [%p] %q%u@%d/%a '
```

Note

L'échappement `%Q` retourne toujours un identifiant à zéro pour les sorties de lignes ordonnées par log_statement car `log_statement` génère la sortie avant que les

identifiants ne soient calculés, y compris pour les requêtes invalides pour lesquelles un identifiant ne peut pas être calculé.

`log_lock_waits` (`boolean`)

Contrôle si une trace applicative est écrite quand une session attend plus longtemps que deadlock_timeout pour acquérir un verrou. Ceci est utile pour déterminer si les attentes de verrous sont la cause des pertes de performance. Désactivé (`off`) par défaut. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier ce paramétrage.

`log_recovery_conflict_waits` (`boolean`)

Contrôle si un message doit être tracé dans les journaux de PostgreSQL au cas où le processus de démarrage doit attendre plus longtemps que `deadlock_timeout` en cas de conflit de récupération. C'est utile pour savoir si un conflit de restauration empêche l'application des journaux de transactions.

La valeur par défaut est `off`. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou sur la ligne de commande.

`log_parameter_max_length` (`integer`)

Si supérieur à zéro, chaque valeur de paramètre lié tracée avec un message qui n'est pas dû à une erreur est raccourcie à ce nombre d'octets. Zéro désactive la trace des paramètres liés pour les traces de requêtes sans erreur. `-1` (valeur par défaut) autorise la trace complète des paramètres liés. Si cette valeur est indiquée sans unité, elle est prise pour un nombre d'octets. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` appropriés peuvent modifier ce paramètre.

Ce paramètre affecte seulement les messages affichés en résultat de la configuration de log_statement, log_min_duration_statement, et d'autres paramètres liés. Une valeur différente de zéro pour ce paramètre ajoute une surcharge, particulièrement si les paramètres sont envoyés en forme binaire, parce qu'une conversion en texte est requise.

`log_parameter_max_length_on_error` (`integer`)

Si supérieur à zéro, chaque valeur de paramètre lié dans une trace d'erreur est raccourcie à ce nombre d'octets. Zéro (valeur par défaut) désactive la trace des paramètres liés pour les traces de requêtes en erreur. `-1` autorise la trace complète des paramètres liés. Si cette valeur est indiquée sans unité, elle est prise pour un nombre d'octets.

Une configuration de ce paramètre à une valeur différente de zéro ajoute une surcharge car PostgreSQL aura besoin de stocker les représentations textuelles des valeurs de paramètres en mémoire au début de chaque requête, qu'une erreur survienne ou pas. La surcharge est supérieure quand les paramètres liés sont envoyés au format binaire que quand ils sont envoyés au format car le premier cas nécessite la conversion des données alors que le dernier cas nécessite seulement la copie de la chaîne.

`log_statement` (`enum`)

Contrôle les instructions SQL à tracer. Les valeurs valides sont `none` (off), `ddl`, `mod` et `all` (toutes les instructions). `ddl` trace toutes les commandes de définition comme `CREATE`, `ALTER` et `DROP`. `mod` trace toutes les instructions `ddl` ainsi que les instructions de modification de données `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` et `COPY FROM`. Les instructions `PREPARE`, `EXECUTE` et `EXPLAIN ANALYZE` sont aussi tracées si la commande qui les contient est d'un type approprié. Pour les clients utilisant le protocole de requêtage étendu, la trace survient quand un message Execute est reçu et les valeurs des paramètres de Bind sont incluses (avec doublement de tout guillemet simple embarqué).

La valeur par défaut est `none`. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` appropriés peuvent changer ce paramétrage.

> **Note**
>
> Les instructions qui contiennent de simples erreurs de syntaxe ne sont pas tracées même si `log_statement` est positionné à `all` car la trace n'est émise qu'après qu'une analyse basique soit réalisée pour déterminer le type d'instruction. Dans le cas du protocole de requêtage étendu, ce paramètre ne trace pas les instructions qui échouent avant la phase Execute (c'est-à-dire pendant l'analyse et la planification). `log_min_error_statement` doit être positionné à `ERROR` pour tracer ce type d'instructions.
>
> Les requêtes tracées pourraient révéler des données sensibles et même contenir des mots de passe en clair.

`log_replication_commands` (`boolean`)

A pour effet d'enregistrer dans le fichier des traces du serveur chaque commande de réplication. Voir Section 55.4 pour plus d'informations à propos des commandes de réplication. La valeur par défaut est `off`. Seuls les superutilisateurs et les utilisateurs avec les droits `SET` adéquats peuvent modifier ce paramètre.

`log_temp_files` (`integer`)

Contrôle l'écriture de traces sur l'utilisation des fichiers temporaires (noms et tailles). Les fichiers temporaires peuvent être créés pour des tris, des hachages et des résultats temporaires de requête. Si activé, une entrée de journal est générée pour chaque fichier temporaire au moment où il est effacé. Zéro implique une trace des informations sur tous les fichiers temporaires alors qu'une valeur positive ne trace que les fichiers dont la taille est supérieure ou égale au nombre indiqué (en kilo-octets). Si cette valeur est indiquée sans unité, elle est comprise comme un nombre de Ko. La valeur par défaut est -1, ce qui a pour effet de désactiver les traces. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` peuvent modifier ce paramètre.

`log_timezone` (`string`)

Configure le fuseau horaire utilisé par l'horodatage des traces. Contrairement à TimeZone, cette valeur est valable pour le cluster complet, de façon à ce que toutes les sessions utilisent le même. La valeur par défaut est `GMT`, mais elle est généralement surchargée dans le fichier `postgresql.conf` ; initdb installera une configuration correspondant à l'environnement système. Voir Section 8.5.3 pour plus d'informations.

20.8.4. Utiliser les journaux au format CSV

L'ajout de `csvlog` dans la liste `log_destination` est une manière simple d'importer des journaux dans une table de base de données. Cette option permet de créer des journaux au format CSV avec les colonnes : l'horodatage en millisecondes, le nom de l'utilisateur, le nom de la base de données, le PID du processus serveur, l'hôte et le numéro de port du client, l'identifiant de la session, le numéro de ligne dans la session, le tag de la commande, l'horodatage de début de la session, l'identifiant de transaction virtuelle, l'identifiant de transaction standard, la sévérité de l'erreur, le code SQLSTATE, le message d'erreur, les détails du message d'erreur, une astuce, la requête interne qui a amené l'erreur (si elle existe), le nombre de caractères pour arriver à la position de l'erreur, le contexte de l'erreur, la requête utilisateur qui a amené l'erreur (si elle existe et si `log_min_error_statement` est activé), le nombre de caractères pour arriver à la position de l'erreur, l'emplacement de l'erreur dans le code source de PostgreSQL (si `log_error_verbosity` est configuré à `verbose`), le nom de l'application, le type du processus serveur, l'identifiant du processus du leader et l'identifiant de la requête.

Exemple de définition d'une table de stockage de journaux au format CSV :

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  backend_type text,
  leader_pid integer,
  query_id bigint,
  PRIMARY KEY (session_id, session_line_num)
);
```

Pour importer un journal dans cette table, on utilise la commande `COPY FROM` :

```
COPY postgres_log FROM '/chemin/complet/vers/le/logfile.csv' WITH
 csv;
```

Il est aussi possible d'accéder au fichier via une table externe en utilisant le module file_fdw.

Quelques conseils pour simplifier et automatiser l'import des journaux CVS :

1. configurer `log_filename` et `log_rotation_age` pour fournir un schéma de nommage cohérent et prévisible des journaux. Cela permet de prédire le nom du fichier et le moment où il sera complet (et donc prêt à être importé) ;

2. initialiser `log_rotation_size` à 0 pour désactiver la rotation par taille comptée, car elle rend plus difficile la prévision du nom du journal ;

3. positionner `log_truncate_on_rotation` à `on` pour que les données anciennes ne soient pas mélangées aux nouvelles dans le même fichier ;

4. la définition de la table ci-dessus inclut une clé primaire. C'est utile pour se protéger de l'import accidentel de la même information à plusieurs reprises. La commande `COPY` valide toutes les données qu'elle importe en une fois. Toute erreur annule donc l'import complet. Si un journal

incomplet est importé et qu'il est de nouveau importé lorsque le fichier est complet, la violation de la clé primaire cause un échec de l'import. Il faut attendre que le journal soit complet et fermé avant de l'importer. Cette procédure protége aussi de l'import accidentel d'une ligne partiellement écrite, qui causerait aussi un échec de `COPY`.

20.8.5. Utiliser le format de trace JSON

Ajouter `jsonlog` dans la liste `log_destination` permet d'importer facilement les fichiers de trace dans différents programmes. Cette option émet des lignes de trace au format JSON.

Les champs texte avec des valeurs NULL sont exclus de la sortie. Des champs supplémentaires pourraient être ajoutés dans le futur. Les applications utilisateurs qui traitent le format `jsonlog` doivent ignorer les champs inconnus.

Chaque ligne de trace est sérialisée sous la forme d'un objet JSON avec un ensemble de clés et de valeurs associées, clés indiquées dans Tableau 20.3.

Tableau 20.3. Clés et valeurs des enregistrements JSON

Clé	Type	Description
`timestamp`	string	Horodatage en millisecondes
`user`	string	Nom d'utilisateur
`dbname`	string	Nom de base de données
`pid`	number	Identifiant de processus
`remote_host`	string	Hôte client
`remote_port`	number	Port client
`session_id`	string	Identifiant de session
`line_num`	number	Numéro de ligne par session
`ps`	string	Affichage actuel de ps
`session_start`	string	Heure de début de session
`vxid`	string	Identifiant de transaction virtuelle
`txid`	string	Identifiant de transaction standard
`error_severity`	string	Niveau de l'erreur
`state_code`	string	Code SQLSTATE
`message`	string	Message d'erreur
`detail`	string	Détails du message d'erreur
`hint`	string	Astuce du message d'erreur
`internal_query`	string	Requête interne qui amène à l'erreur
`internal_position`	number	Curseur dans la requête interne
`context`	string	Contexte d'erreur
`statement`	string	Chaîne de requête fournie par le client
`cursor_position`	number	Curseur dans la chaîne de requête
`func_name`	string	Nom de la fonction ayant déclenché l'erreur

Clé	Type	Description
file_name	string	Nom du fichier contenant la fonction ayant déclenché l'erreur
file_line_num	number	Numéro de ligne du fichier contenant la fonction ayant déclenché l'erreur
application_name	string	Nom d'application du client
backend_type	string	Type de backend
leader_pid	number	Identifiant du processus leader pour les workers parallélisés
query_id	number	Identifiant de requête

20.8.6. Titre des processus

Ces paramètres contrôlent comment les titres de processus des processus serveurs sont modifiés. Les titres de processus sont affichées typiquement en utilisant des programmes comme ps ou, sur Windows, Process Explorer. Voir Section 28.1 pour plus de détails.

cluster_name (string)

Configure un nom qui identifie cette instance pour différentes raisons. Le nom de l'instance apparaît dans le titre de tous les processus de l'instance. De plus, c'est le nom d'application par défaut pour une connexion standby (voir synchronous_standby_names.)

Le nom peut être n'importe quelle chaîne de caractères de longueur inférieure à NAMEDATALEN (64 caractères dans une compilation standard du serveur). Seuls les caractères ASCII imprimables peuvent être utilisés dans cluster_name. Les autres caractères seront remplacés par des points d'interrogation (?). Aucun nom n'est affiché si ce paramètre est positionné sur la chaîne vide '' (ce qui est la valeur par défaut). Ce paramètre ne peut être positionné qu'au démarrage du serveur.

update_process_title (boolean)

Active la mise à jour du titre du processus chaque fois qu'une nouvelle commande SQL est reçue par le serveur. Ce paramètre est à on par défaut sur la plupart des plateformes mais il est à off sur Windows car cette plateforme souffre de lenteurs plus importantes pour la mise à jour du titre du processus. Seuls les superutilisateurs et les utilisateurs disposant des droits SET adéquats peuvent modifier ce paramètre.

20.9. Statistiques d'exécution

20.9.1. Statistiques cumulatives sur les requêtes et index

Ces paramètres contrôlent le système de statistiques cumulatives. Lorsqu'il est activé, les données récupérées peuvent être visualisées à travers la famille de vues systèmes pg_stat et pg_statio. Consultez Chapitre 28 pour plus d'informations.

track_activities (boolean)

Active la collecte d'informations sur la commande en cours d'exécution dans chaque session, avec son identifiant et l'heure de démarrage de la commande. Ce paramètre est activé par défaut. Même si le paramètre est activé, cette information est seulement visible aux superutilisateurs, aux rôles ayant les droits du rôle pg_read_all_stats et au rôle propriétaire des sessions tracées (y compris celles appartenant à un rôle dont ils sont membres) ; de ce fait, cela ne représente pas une

faille de sécurité. Seuls les superutilisateurs et les utilisateurs disposant des droits SET adéquats peuvent modifier ce paramètre.

`track_activity_query_size` (`integer`)

Indique la quantité de mémoire réservée pour enregistrer le texte de la commande en cours d'exécution pour chaque session active, pour le champ `pg_stat_activity.query`. Si cette valeur est spécifiée sans unité, elle est comprise comme un nombre d'octets. La valeur par défaut est 1024 octets. Ce paramètre peut seulement être configuré au démarrage du serveur.

`track_counts` (`boolean`)

Active la récupération de statistiques sur l'activité de la base de données. Ce paramètre est activé par défaut car le processus autovacuum utilise les informations ainsi récupérées. Seuls les superutilisateurs et les utilisateurs disposant des droits SET adéquats peuvent modifier ce paramètre.

`track_io_timing` (`boolean`)

Active le chronométrage des appels d'entrées/sorties de la base de données. Ce paramètre est désactivé par défaut car il demandera sans cesse l'heure courante au système d'exploitation, ce qui peut causer une surcharge significative sur certaines plateformes. Vous pouvez utiliser l'outil pg_test_timing pour mesurer la surcharge causée par le chronométrage sur votre système. Les informations de chronométrage des entrées/sorties sont affichées dans `pg_stat_database`, dans la sortie de EXPLAIN quand l'option BUFFERS est utilisée, dans la sortie de VACUUM quand l'option VERBOSE est utilisée, par autovacuum pour les auto-vacuums et auto-analyzes quand log_autovacuum_min_duration est configurée, et par pg_stat_statements. Seuls les superutilisateurs et les utilisateurs disposant des droits SET adéquats peuvent modifier cette configuration.

`track_wal_io_timing` (`boolean`)

Active le chronométrage des I/O pour les journaux de transactions. Ce paramètre est désactivé par défaut car il implique de nombreux appels supplémentaires au système d'exploitation, ce qui peut être consommateur de ressources pour la plateforme. Vous pouvez utiliser l'outil pg_test_timing pour mesurer la surcharge du chronométrage sur votre système. Les informations de chronométrage des I/O sont disponibles dans la vue `pg_stat_wal`. Seuls les superutilisateurs et les utilisateurs disposant des droits SET adéquats peuvent modifier ce paramètre.

`track_functions` (`enum`)

Active le suivi du nombre et de la durée des appels aux fonctions. Précisez `pl` pour ne tracer que les fonctions de langages procéduraux, ou `all` pour suivre aussi les fonctions SQL et C. La valeur par défaut est `none`, qui désactive le suivi des statistiques de fonctions. Seuls les superutilisateurs et les utilisateurs disposant des droits SET adéquats peuvent modifier ce paramètre.

> **Note**
>
> Les fonctions en langage SQL qui sont assez simples pour être « inlined », c'est à dire substituées dans le code de la requête appelante, ne seront pas suivies, quelle que soit la valeur de ce paramètre.

`stats_fetch_consistency` (`enum`)

Détermine le comportement quand des statistiques cumulatives sont accédées plusieurs fois dans une transaction. Si configuré à `none`, chaque accès récupère de nouveau les compteurs à partir de la mémoire partagée. Si configuré à `cache`, le premier accès aux statistiques pour un objet met en cache ces statistiques jusqu'à la fin de la transaction sauf si `pg_stat_clear_snapshot()` est appelée. Si configuré à `snapshot`, le premier accès aux statistiques met en cache toutes les

statistiques accessibles pour la base de données de connexion, jusqu'à la fin de la transaction, sauf si `pg_stat_clear_snapshot()` est appelée. Modifier ce paramètre dans une transaction annule le snapshot de statistiques. La valeur par défaut est `cache`.

> **Note**
>
> `none` est conseillé pour les systèmes de supervision. Si les valeurs ne sont accédées qu'une seule fois, cette configuration est plus efficace. `cache` assure que des accès répétées rapportent les mêmes valeurs, ce qui est important pour des requêtes impliquant, par exemple;, des jointures sur soi-même. `snapshot` peut être utile lors d'une inspection itérative des statistiques mais a une surcharge importante, en particulier si beaucoup d'objets existent.

20.9.2. Surveillance et statistiques

`compute_query_id` (`enum`)

Active le calcul des identifiants de requête. Les identifiants de requête sont disponibles dans la vue `pg_stat_activity`, en utilisant `EXPLAIN`, ou dans les traces de PostgreSQL si cela a été configuré avec le paramètre log_line_prefix. L'extension pg_stat_statements nécessite aussi un identifiant de requête. Notez bien qu'un module externe peut être une alternative acceptable pour calculer un identifiant de requête. Dans ce cas, pensez à désactiver le calcul automatique des identifiants de requête par PostgreSQL. Les valeurs acceptées pour ce paramètre sont `off` (toujours désactivé), `on` (toujours activé), `auto` qui permet à des modules externes comme pg_stat_statements de l'activer automatiquement, et `regress` qui a le même effet que `auto`, sauf que l'identifiant de requête n'est pas affiché dans la sortie `EXPLAIN` pour faciliter les tests de régression automatisés. La valeur par défaut est `auto`.

> **Note**
>
> Pour garantir qu'un seul identifiant de requête soit généré et affiché, les extensions qui génèrent des identifiants de requête doivent lever une erreur si un identifiant de requête a déjà été généré pour une requête.

`log_statement_stats` (`boolean`)
`log_parser_stats` (`boolean`)
`log_planner_stats` (`boolean`)
`log_executor_stats` (`boolean`)

Écrivent, pour chaque requête, les statistiques de performance du module respectif dans les traces du serveur. C'est un outil de profilage très simpliste, similaire aux possibilités de l'appel `getrusage()` du système d'exploitation Unix. `log_statement_stats` rapporte les statistiques d'instructions globales, tandis que les autres fournissent un rapport par module. `log_statement_stats` ne peut pas être activé conjointement à une option de module. Par défaut, toutes ces options sont désactivées. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier ces paramètres.

20.10. Nettoyage (`vacuum`) automatique

Ces paramètres contrôlent le comportement de la fonctionnalité appelée *autovacuum*. Se référer à la Section 25.1.6 pour plus de détails. Notez que beaucoup de ces paramètres peuvent être surchargés au niveau de chaque table ; voir Paramètres de stockage.

`autovacuum` (`boolean`)

Contrôle si le serveur doit démarrer le démon d'autovacuum. Celui-ci est activé par défaut. track_counts doit aussi être activé pour que ce démon soit démarré. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande; cependant, le processus d'autovacuum peut être désactivé au niveau de chaque table en modifiant les paramètres de stockage de la table.

Même si ce paramètre est désactivé, le système lance les processus autovacuum nécessaires pour empêcher le bouclage des identifiants de transaction. Voir Section 25.1.5 pour plus d'informations.

`autovacuum_max_workers` (`integer`)

Indique le nombre maximum de processus autovacuum (autre que le lanceur d'autovacuum) qui peuvent être exécutés simultanément. La valeur par défaut est 3. Ce paramètre ne peut être configuré qu'au lancement du serveur.

`autovacuum_naptime` (`integer`)

Indique le délai minimum entre les tours d'activité du démon autovacuum sur une base. À chaque tour, le démon examine une base de données et lance les commandes `VACUUM` et `ANALYZE` nécessaires aux tables de cette base. Si cette valeur est indiquée sans unité, elle est comprise comme un nombre de secondes. Il vaut, par défaut, une minute (`1min`). Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`autovacuum_vacuum_threshold` (`integer`)

Indique le nombre minimum de lignes mises à jour ou supprimées nécessaire pour déclencher un `VACUUM` sur une table. La valeur par défaut est de 50 lignes. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande mais il est possible de surcharger ce paramètre pour toute table en modifiant les paramètres de stockage de la table.

`autovacuum_vacuum_insert_threshold` (`integer`)

Indique le nombre de lignes enregistrées avant de déclencher un `VACUUM` sur une table. La valeur par défaut est 1000 lignes. Si -1 est indiqué, autovacuum ne déclenchera pas d'opération `VACUUM` sur une table en se basant sur le nombre d'insertions. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` et sur la ligne de commande du serveur ; mais la configuration peut être surchargée par des tables individuelles en modifiant leur paramètres de stockage.

`autovacuum_analyze_threshold` (`integer`)

Indique le nombre minimum de lignes insérées, mises à jour ou supprimées nécessaire pour déclencher un `ANALYZE` sur une table. La valeur par défaut est de 50 lignes. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande mais il est possible de surcharger ce paramètre pour toute table en modifiant les paramètres de stockage de la table.

`autovacuum_vacuum_insert_scale_factor` (`floating point`)

Indique une fraction de la taille de la table à ajouter à `autovacuum_vacuum_insert_threshold` avant de décider de déclencher un `VACUUM` sur une table. La valeur par défaut est 0,2 (soit 20% de la table). Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` et sur la ligne de commande du serveur ; mais la configuration peut être surchargée par des tables individuelles en modifiant leur paramètres de stockage.

`autovacuum_vacuum_scale_factor` (`floating point`)

Indique la fraction de taille de la table à ajouter à `autovacuum_vacuum_threshold` pour décider du moment auquel déclencher un `VACUUM`. La valeur par défaut est 0.2 (20 % de la taille de la table). Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande mais il est possible de surcharger ce paramètre pour toute table en modifiant les paramètres de stockage de la table.

`autovacuum_analyze_scale_factor` (`floating point`)

Indique la fraction de taille de la table à ajouter à `autovacuum_analyze_threshold` pour décider du moment auquel déclencher une commande `ANALYZE`. La valeur par défaut est 0.1 (10 % de la taille de la table). Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande mais il est possible de surcharger ce paramètre pour toute table en modifiant les paramètres de stockage de la table.

`autovacuum_freeze_max_age` (`integer`)

Indique l'âge maximum (en transactions) que le champ `pg_class.relfrozenxid` d'une table peut atteindre avant qu'une opération `VACUUM` ne soit forcée pour empêcher la réinitialisation de l'ID de transaction sur cette table. Le système lance les processus autovacuum pour éviter ce bouclage même si l'autovacuum est désactivé.

L'opération VACUUM supprime aussi les anciens fichiers du sous-répertoire `pg_xact`, ce qui explique pourquoi la valeur par défaut est relativement basse (200 millions de transactions). Ce paramètre n'est lu qu'au démarrage du serveur, mais il peut être diminué pour toute table en modifiant les paramètres de stockage de la table. Pour plus d'informations, voir Section 25.1.5.

`autovacuum_multixact_freeze_max_age` (`integer`)

Indique l'âge maximum (en multixacts) que le champ `pg_class.relminmxid` d'une table peut atteindre avant qu'une opération `VACUUM` ne soit forcé pour empêcher une réutilisation des identifiants multixact dans la table. Notez que le système lancera les processus autovacuum pour empêcher la réutilisation même si l'autovacuum est normalement désactivé.

Un VACUUM des multixacts s'occupe aussi de la suppression des anciens fichiers à partir des sous-répertoires `pg_multixact/members` et `pg_multixact/offsets`, ce qui explique pourquoi la valeur par défaut est relativement basse (400 million de multixacts). Ce paramètre est seulement configurable au démarrage du serveur mais sa valeur peut être réduite pour des tables individuelles en modifiant les paramètres de stockage de la table. Pour plus d'informations, voir Section 25.1.5.1.

`autovacuum_vacuum_cost_delay` (`integer`)

Indique la valeur du coût de délai utilisée dans les opérations de `VACUUM`. Si -1 est indiqué, la valeur habituelle de vacuum_cost_delay est utilisée. Si cette valeur est spécifiée sans unité, elle est comprise comme un nombre de millisecondes. La valeur par défaut est 2 millisecondes. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande mais il est possible de le surcharger pour toute table en modifiant les paramètres de stockage de la table.

`autovacuum_vacuum_cost_limit` (`integer`)

Indique la valeur de coût limite utilisée dans les opérations de `VACUUM` automatiques. Si -1 est indiqué (valeur par défaut), la valeur courante de vacuum_cost_limit est utilisée. La valeur est distribuée proportionnellement entre les processus autovacuum en cours d'exécution, s'il y en a plus d'un, de sorte que la somme des limites de chaque processus ne dépasse jamais la limite de cette variable. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande mais il est possible de le surcharger pour toute table en modifiant les paramètres de stockage.

20.11. Valeurs par défaut des connexions client

20.11.1. Comportement des instructions

`client_min_messages` (`enum`)

Contrôle les niveaux de message envoyés au client. Les valeurs valides sont DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, LOG, NOTICE, WARNING et ERROR. Chaque niveau inclut tous les niveaux qui le suivent. Plus on progresse dans la liste, plus le nombre de messages envoyés est faible. NOTICE est la valeur par défaut. LOG a ici une portée différente de celle de log_min_messages.

Les messages de niveau INFO sont toujours envoyés au client.

search_path (string)

Cette variable précise l'ordre dans lequel les schémas sont parcourus lorsqu'un objet (table, type de données, fonction, etc.) est référencé par un simple nom sans précision du schéma. Lorsque des objets de noms identiques existent dans plusieurs schémas, c'est le premier trouvé dans le chemin de recherche qui est utilisé. Il ne peut être fait référence à un objet qui ne fait partie d'aucun des schémas indiqués dans le chemin de recherche qu'en précisant son schéma conteneur avec un nom qualifié (avec un point).

search_path doit contenir une liste de noms de schémas séparés par des virgules. Tout nom qui ne correspond pas à un schéma existant ou qui correspond à un schéma pour lequel l'utilisateur n'a pas le droit USAGE, est ignoré silencieusement.

Si un des éléments de la liste est le nom spécial $user, alors le schéma dont le nom correspond à la valeur retournée par CURRENT_USER est substitué, s'il existe et que l'utilisateur ait le droit USAGE sur ce schéma (sinon $user est ignoré).

Le schéma du catalogue système, pg_catalog, est toujours parcouru, qu'il soit ou non mentionné dans le chemin. Mentionné, il est alors parcouru dans l'ordre indiqué. Dans le cas contraire, il est parcouru *avant* tout autre élément du chemin.

De même, le schéma des tables temporaires, pg_temp_*nnn*, s'il existe, est toujours parcouru. Il peut être explicitement ajouté au chemin à l'aide de l'alias pg_temp. S'il n'en fait pas partie, la recherche commence par lui (avant même pg_catalog). Néanmoins, seuls les noms de relation (table, vue, séquence, etc.) et de type de données sont recherchés dans le schéma temporaire. Aucune fonction et aucun opérateur n'y est jamais recherché.

Lorsque des objets sont créés sans précision de schéma cible particulier, ils sont placés dans le premier schéma valide listé dans le chemin de recherche. Une erreur est rapportée si le chemin de recherche est vide.

La valeur par défaut de ce paramètre est "$user", public. Elle permet l'utilisation partagée d'une base de données (dans laquelle aucun utilisateur n'a de schéma privé et tous partagent l'utilisation de public), les schémas privés d'utilisateur ainsi qu'une combinaison de ces deux modes. D'autres effets peuvent être obtenus en modifiant le chemin de recherche par défaut, globalement ou par utilisateur.

La valeur courante réelle du chemin de recherche peut être examinée via la fonction SQL current_schemas() (voir Section 9.26). Elle n'est pas identique à la valeur de search_path car current_schemas affiche la façon dont les requêtes apparaissant dans search_path sont résolues.

row_security (boolean)

Cette variable indique s'il convient de lever une erreur au lieu d'appliquer la politique de sécurité au niveau ligne. Lorsque positionnée à on, les politiques s'appliquent normalement. Lorsque positionnée à off, les requêtes qui remplissent les conditions d'au moins une politique de sécurité échouent. La valeur par défaut est on. Positionnez la valeur sur off dans le cas où une visibilité limitée des lignes pourrait causer des résultats incorrects ; par exemple, pg_dump effectue ce changement par défaut. Cette variable n'a aucun effet sur les rôles qui outrepassent toutes les politiques de sécurité niveau ligne, à savoir, les superutilisateurs et les rôles qui possèdent l'attribut BYPASSRLS.

Pour plus d'informations sur les politiques de sécurité niveau ligne, voir CREATE POLICY.

`default_table_access_method` (`string`)

Ce paramètre spécifie la méthode d'accès par défaut aux tables. Ce paramètre est utilisé lors de la création des tables ou des vues matérialisées si la commande `CREATE` n'indique pas spécifiquement de méthode d'accès ou quand `SELECT ... INTO` est utilisé, ce qui ne permet pas de spécifier une méthode d'accès à la table. La valeur par défaut est `heap`.

`default_tablespace` (`string`)

Cette variable indique le *tablespace* par défaut dans lequel sont créés les objets (tables et index) quand une commande `CREATE` ne l'explicite pas.

La valeur est soit le nom d'un *tablespace* soit une chaîne vide pour indiquer l'utilisation du *tablespace* par défaut de la base de données courante. Si la valeur ne correspond pas au nom d'un tablespace existant, PostgreSQL utilise automatiquement le tablespace par défaut de la base de données courante. Si un *tablespace* différent de celui par défaut est indiqué, l'utilisateur doit avoir le droit `CREATE`. Dans le cas contraire, la tentative de création échouera.

Cette variable n'est pas utilisée pour les tables temporaires ; pour elles, temp_tablespaces est consulté à la place.

Cette variable n'est pas utilisée non plus lors de la création de bases de données. Par défaut, une nouvelle base de données hérite sa configuration de tablespace de la base de données modèle qui sert de copie.

Si ce paramètre est configuré à une valeur autre que la chaîne vide quand une table partitionnée est créée, le tablespace de la table partitionnée sera configuré à cette valeur, qui sera utilisé comme tablespace par défaut pour les partitions créées dans le futur, même si `default_tablespace` a été modifié depuis.

Pour plus d'informations sur les tablespaces, voir Section 23.6.

`default_toast_compression` (`enum`)

Ce paramètre configure la valeur par défaut pour la méthode de compression des colonnes TOAST des nouvelles tables. La commande `CREATE TABLE` peut surcharger cette valeur par défaut si la clause `COMPRESSION` est utilisée au niveau des colonnes. Les méthodes de compression supportées sont `pglz` et, si PostgreSQL a été compilé avec l'option `--with-lz4`, `lz4`. La valeur par défaut est `pglz`.

`temp_tablespaces` (`string`)

Cette variable indique le (ou les) *tablespace*(s) dans le(s)quel(s) créer les objets temporaires (tables temporaires et index sur des tables temporaires) quand une commande `CREATE` n'en explicite pas. Les fichiers temporaires créés par les tris de gros ensembles de données sont aussi créés dans ce *tablespace*.

Cette valeur est une liste de noms de *tablespace*s. Quand cette liste contient plus d'un nom, PostgreSQL choisit un membre de la liste au hasard à chaque fois qu'un objet temporaire doit être créé. En revanche, dans une transaction, les objets temporaires créés successivement sont placés dans les *tablespace*s successifs de la liste. Si l'élément sélectionné de la liste est une chaîne vide, PostgreSQL utilise automatiquement le *tablespace* par défaut de la base en cours.

Si `temp_tablespaces` est configuré interactivement, l'indication d'un *tablespace* inexistant est une erreur. Il en est de même si l'utilisateur n'a pas le droit `CREATE` sur le *tablespace* indiqué. Néanmoins, lors de l'utilisation d'une valeur précédemment configurée, les *tablespace*s qui n'existent pas sont ignorés comme le sont les *tablespace*s pour lesquels l'utilisateur n'a pas le droit `CREATE`. Cette règle s'applique, en particulier, lors de l'utilisation d'une valeur configurée dans le fichier `postgresql.conf`.

La valeur par défaut est une chaîne vide. De ce fait, tous les objets temporaires sont créés dans le *tablespace* par défaut de la base de données courante.

Voir aussi default_tablespace.

`check_function_bodies` (`boolean`)

Ce paramètre est habituellement positionné à `on`. Positionné à `off`, il désactive la validation du corps de la routine lors de CREATE FUNCTION et CREATE PROCEDURE. Désactiver la validation évite les effets de bord du processus de validation, en particulier pour éviter les faux positifs dûs à des problèmes tels que les références. Configurer ce paramètre à `off` avant de charger les fonctions à la place des autres utilisateurs ; pg_dump le fait automatiquement.

`default_transaction_isolation` (`enum`)

Chaque transaction SQL a un niveau d'isolation. Celui-ci peut être « read uncommitted », « read committed », « repeatable read » ou « serializable ». Ce paramètre contrôle le niveau d'isolation par défaut de chaque nouvelle transaction. La valeur par défaut est « read committed ».

Consulter le Chapitre 13 et SET TRANSACTION pour plus d'informations.

`default_transaction_read_only` (`boolean`)

Une transaction SQL en lecture seule ne peut pas modifier les tables permanentes. Ce paramètre contrôle le statut de lecture seule par défaut de chaque nouvelle transaction. La valeur par défaut est `off` (lecture/écriture).

Consulter SET TRANSACTION pour plus d'informations.

`default_transaction_deferrable` (`boolean`)

Lors du fonctionnement avec le niveau d'isolation `serializable`, une transaction SQL en lecture seule et différable peut subir un certain délai avant d'être autorisée à continuer. Néanmoins, une fois qu'elle a commencé son exécution, elle n'encourt aucun des frais habituels nécessaires pour assurer sa sériabilité. Donc le code de sérialisation n'a aucune raison de forcer son annulation à cause de mises à jour concurrentes, ce qui rend cette option très intéressante pour les longues transactions en lecture seule.

Ce paramètre contrôle le statut différable par défaut de chaque nouvelle transaction. Il n'a actuellement aucun effet sur les transactions en lecture/écriture ou celles opérant à des niveaux d'isolation inférieurs à `serializable`. La valeur par défaut est `off`.

Consultez SET TRANSACTION pour plus d'informations.

`transaction_isolation` (`enum`)

Ce paramètre reflète le niveau d'isolation de la transaction. Au début de chaque transaction, il est configuré à la valeur courante du paramètre default_transaction_isolation. Toute tentative ultérieure de modification est équivalente à une commande SET TRANSACTION.

`transaction_read_only` (`boolean`)

Ce paramètre reflète le statut lecture-seule de la transaction courante. Au début de chaque transaction, il est configuré à la valeur courante du paramètre default_transaction_read_only. Toute tentative de modification ultérieure est équivalente à une commande SET TRANSACTION.

`transaction_deferrable` (`boolean`)

Ce paramètre reflète le statut de reportabilité de la transaction courante. Au début de chaque transaction, il est configuré à la valeur courante du paramètre default_transaction_deferrable. Toute tentative de modification ultérieure est équivalente à une commande SET TRANSACTION.

`session_replication_role` (`enum`)

Contrôle l'exécution des triggers et règles relatifs à la réplication pour la session en cours. Les valeurs possibles sont `origin` (la valeur par défaut), `replica` et `local`. Configurer ce paramètre résulte en l'annulation de tout plan de requête précédemment mis en cache. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier sa configuration.

L'utilisation prévue de ce paramètre est que les systèmes de réplication logique le passent à `replica` quand ils répliquent des changements. L'effet sera que les triggers et les règles (quand on n'a pas modifié la configuration par défaut) ne se déclencheront pas sur la réplique. Voir les clauses `ALTER TABLE ENABLE TRIGGER` et `ENABLE RULE` pour plus d'informations.

En interne, PostgreSQL traite de la même manière les paramètres `origin` et `local`. Les systèmes de réplication tiers peuvent utiliser ces deux valeurs pour leurs besoins internes, par exemple en utilisant `local` pour désigner la session dont les changements ne seront pas répliqués.

Puisque les clés étrangères sont implémentées comme des triggers, passer ce paramètre à `replica` désactive aussi toutes les vérifications de clés étrangères, ce qui peut laisser les données dans un état incohérent en cas d'utilisation inappropriée.

`statement_timeout` (`integer`)

Interrompt toute instruction qui dure plus longtemps que cette durée. Si `log_min_error_statement` est configuré à `ERROR`, ou plus bas, l'instruction en cause est tracée. Si cette valeur est indiquée sans unité, elle est comprise comme un nombre de millisecondes. La valeur zéro (par défaut) désactive le décompte.

Le délai est mesuré à partir du moment où une commande arrive au serveur jusqu'à sa fin par le serveur. Si plusieurs requêtes SQL apparaissent dans un seul message simple-Query, le délai est appliqué à chaque requête séparément. (Les versions de PostgreSQL antérieures à la 13 traitaient le délai pour la chaîne complète de requêtes.) Dans le protocole de requête étendue, le délai commence lors du début de l'exécution de tout message relatif à la requête (Parse, Bind, Execute, Describe) et elle est annulée à la fin du message Execute ou Sync.

Il n'est pas recommandé de configurer `statement_timeout` dans `postgresql.conf` car cela affecte toutes les sessions.

`lock_timeout` (`integer`)

Annule toute requête qui attend plus longtemps que la durée indiquée par ce paramètre lors de la tentative d'acquisition d'un verrou sur une table, un index, une ligne ou tout autre objet d'une base de données. La limite de temps s'applique séparément pour chaque tentative d'acquisition d'un verrou. La limite s'applique pour les demandes de verrous explicites (comme `LOCK TABLE`, ou `SELECT FOR UPDATE` sans `NOWAIT`) et pour ceux acquis implicitement. Si cette valeur est indiquée sans unité, elle est comprise comme un nombre de millisecondes. Une valeur de zéro (valeur par défaut) désactive ce comportement.

Contrairement à `statement_timeout`, ce délai peut seulement intervenir lors de l'attente de verrous. Notez que si `statement_timeout` est différent de zéro, il est plutôt inutile de configurer `lock_timeout` à la même valeur ou à une valeur plus importante puisque le délai sur la requête se déclenchera toujours avant. Si `log_min_error_statement` est configuré à `ERROR` ou plus bas, l'instruction qui dépasse ce délai sera tracé.

Configurer `lock_timeout` dans `postgresql.conf` n'est pas recommandé car cela affecterait toutes les sessions.

`idle_in_transaction_session_timeout` (`integer`)

Termine toute session inactive (c'est-à-dire qui attend que le client envoie une requête) alors qu'une transaction est en cours depuis plus longtemps que le temps indiqué. Si cette valeur est

fournie sans unité, elle est comprise en millisecondes. Une valeur à zéro (qui est la valeur par défaut) désactive ce délai.

Cette option peut être utilisée pour s'assurer que des sesions inactives ne conservent pas des verrous sur une durée déraisonnable. Même lorsqu'aucun verrou non important est réservé, une transaction ouverte empêche de nettoyer correctement les lignes mortes les plus récentes qui peuvent n'être plus visibles que par cette transaction. Conserver des transactions inactives alors qu'une transaction est ouverte peut donc ainsi augmenter la fragmentation des données. Voir Section 25.1 pour plus d'information.

`idle_session_timeout` (`integer`)

Termine toute transaction inactive depuis plus longtemps que le temps indiqué(c'est-à-dire qui attend que le client envoie une requête), mais pour laquelle aucune transaction n'est en cours. Si cette valeur est fournie sans unité, elle est comprise en millisecondes. Une valeur à zéro (qui est la valeur par défaut) désactive ce délai.

Contrairement au cas des transactions en cours, une session inactive sans transaction en cours ne prend pas beaucoup de ressources sur le serveur. Il est donc moins nécessaire d'activer ce paramètre que `idle_in_transaction_session_timeout`.

Attention à l'utilisation de ce paramètre avec certains logiciels comme un pooler de connexions ou un autre logiciel tiers car ce type de logiciels pourrait mal gérer la fermeture inopinée des connexions. Il peut être intéressant de n'activer ce délai que pour les sessions interactives, peut-être en appliquant ce paramètre qu'à certains utilisateurs.

`vacuum_freeze_table_age` (`integer`)

`VACUUM` effectuera un parcours agressif de la table si le champ `pg_class.relfrozenxid` de la table a atteint l'âge spécifié par ce paramètre. Un parcours agressif diffère d'un `VACUUM` standard dans le sens où il visite chaque bloc qui pourrait contenir des XID ou MXID non gelés, pas seulement ceux qui pourraient contenir des lignes mortes. La valeur par défaut est 150 millions de transactions. Même si les utilisateurs peuvent positionner cette valeur à n'importe quelle valeur comprise entre zéro et 2 milliards, `VACUUM` limitera silencieusement la valeur effective à 95% de autovacuum_freeze_max_age, afin qu'un vacuum périodique manuel ait une chance de s'exécuter avant un autovacuum anti-bouclage ne soit lancé pour la table. Pour plus d'informations voir Section 25.1.5.

`vacuum_freeze_min_age` (`integer`)

Indique l'âge limite (en transactions) que `VACUUM` doit utiliser pour décider de geler les versions de ligne lors du parcours d'une table. La valeur par défaut est 50 millions. Bien que les utilisateurs puissent configurer une valeur quelconque comprise entre zéro et 1 milliard, `VACUUM` limite silencieusement la valeur réelle à la moitié de la valeur de autovacuum_freeze_max_age afin que la valeur entre deux autovacuums forcés ne soit pas déraisonnablement courte. Pour plus d'informations, voir Section 25.1.5.

`vacuum_failsafe_age` (`integer`)

Spécifie l'âge maximum (en transactions) que le champ `pg_class.relfrozenxid` d'une table peut atteindre avant qu'un `VACUUM` ne prenne des mesures exceptionnelles pour empêcher un arrêt du système lié à un risque de bouclage des identifiants de transaction (*ID wraparound*). Il s'agit d'une stratégie que le `VACUUM` n'applique qu'en dernier recours. Cette sécurité est déclenchée généralement quand un autovacuum pour empêcher le bouclage des identifiants de transaction tourne depuis quelque temps, même si rien n'empêche ce fonctionnement exceptionnel de se déclencher pendant un `VACUUM`.

Lorsque cette sécurité est déclenchée, tout délai basé sur les coûts est ignoré et d'autres tâches de maintenance non essentielles (comme le vacuum des index) sont reportées.

La valeur par défaut est 1,6 milliards de transactions. Même s'il est possible de fournir une valeur entre 0 et 2,1 milliards, VACUUM ajustera cette valeur pour qu'elle ne dépasse pas 105% de autovacuum_freeze_max_age.

`vacuum_multixact_freeze_table_age` (`integer`)

VACUUM réalise un parcours agressif de la table si le champ `pg_class.relminmxid` de la table a atteint l'âge indiqué par ce paramètre. Un parcours agressif diffère d'un VACUUM standard dans le sens où il visite chaque bloc qui pourrait contenir des XID ou MXID non gelés, pas seulement ceux qui pourraient contenir des lignes mortes. La valeur par défaut est de 150 millions de multixacts. Bien que les utilisateurs peuvent configurer cette valeur entre zéro et deux milliards, VACUUM limitera silencieusement la valeur réelle à 95% de autovacuum_multixact_freeze_max_age, pour qu'un VACUUM manuel périodique ait une chance d'être exécuté avant qu'une opération anti-réutilisation d'identifiants ne soit exécutée sur la table. Pour plus d'informations, voir Section 25.1.5.1.

`vacuum_multixact_freeze_min_age` (`integer`)

Précise l'âge limite (en multixacts) que VACUUM doit utiliser pour décider s'il doit remplacer les identifiants multixact avec un nouvel identifiant de transaction ou de multixact lors de son parcours de la table. La valeur par défaut est de 5 millions de multixacts. Bien que les utilisateurs peuvent configurer cette valeur entre zéro et un milliard, VACUUM limitera silencieusement la valeur réelle à la moitié de la valeur de autovacuum_multixact_freeze_max_age, pour qu'il y ait un délai raisonnable entre deux autovacuums forcés. Pour plus d'informations, voir Section 25.1.5.1.

`vacuum_multixact_failsafe_age` (`integer`)

Spécifie l'âge maximum (en transactions) que le champ `pg_class.relminxid` d'une table peut atteindre avant qu'un VACUUM ne prenne des mesures exceptionnelles pour empêcher un arrêt du système lié à un risque de bouclage des identifiants de transaction (*ID wraparound*). Il s'agit d'une stratégie que le VACUUM n'applique qu'en dernier recours. Cette sécurité est déclenchée généralement quand un autovacuum pour empêcher le bouclage des identifiants de transaction tourne depuis quelque temps, même si rien n'empêche ce fonctionnement exceptionnel de se déclencher pendant un VACUUM.

Lorsque cette sécurité est déclenchée, tout délai basé sur les coûts est ignoré et d'autres tâches de maintenance non essentielles (comme le vacuum des index) sont reportées.

La valeur par défaut est 1,6 milliards de transactions. Même s'il est possible de fournir une valeur entre 0 et 2,1 milliards, VACUUM ajustera cette valeur pour qu'elle ne dépasse pas 105% de autovacuum_multixact_freeze_max_age.

`bytea_output` (`enum`)

Configure le format de sortie pour les valeurs de type `bytea`. Les valeurs valides sont `hex` (la valeur par défaut) et `escape` (le format traditionnel de PostgreSQL). Voir Section 8.4 pour plus d'informations. Le type `bytea` accepte toujours les deux formats en entrée, quelque soit la valeur de cette configuration.

`xmlbinary` (`enum`)

Définit la manière de coder les valeurs binaires en XML. Ceci s'applique, par exemple, quand les valeurs `bytea` sont converties en XML par les fonctions `xmlelement` et `xmlforest`. Les valeurs possibles sont `base64` et `hex`, qui sont toutes les deux définies dans le standard XML Schema. La valeur par défaut est `base64`. Pour plus d'informations sur les fonctions relatives à XML, voir Section 9.15.

Le choix effectif de cette valeur est une affaire de sensibilité, la seule restriction provenant des applications clientes. Les deux méthodes supportent toutes les valeurs possibles, et ce bien que le codage hexadécimal soit un peu plus grand que le codage en base64.

`xmloption` (`enum`)

Définit si `DOCUMENT` ou `CONTENT` est implicite lors de la conversion entre XML et valeurs chaînes de caractères. Voir Section 8.13 pour la description. Les valeurs valides sont `DOCUMENT` et `CONTENT`. La valeur par défaut est `CONTENT`.

D'après le standard SQL, la commande pour configurer cette option est :

```
SET XML OPTION { DOCUMENT | CONTENT };
```

Cette syntaxe est aussi disponible dans PostgreSQL.

20.11.2. Préchargement de bibliothèques partagées

Plusieurs paramètres sont disponibles pour le préchargement de bibliothèques partagées sur le serveur. Ces bibliothèques peuvent servir à ajouter des fonctionnalités supplémentaires ou à améliorer les performances. Par exemple, une configuration à `'$libdir/mabibliotheque'` force le chargement de la bibliothèque `mabibliotheque.so` (ou sur certaines plateformes de `mabibliotheque.sl`) à partir du répertoire standard d'installation. Les différences entre les paramètres concernent la prise d'effet et les droits requis pour les modifier.

Les bibliothèques de procédures stockées pour PostgreSQL peuvent être préchargées de cette façon, habituellement en utilisant la syntaxe `'$libdir/plXXX'` où `XXX` est `pgsql`, `perl`, `tcl` ou `python`.

Seules les bibliothèques partagées spécifiquement codées pour PostgreSQL peuvent être chargées de cette façon. Chaque bibliothèque supportée par PostgreSQL a un « bloc magique » qui est vérifié pour garantir sa comptabilité. De ce fait, les bibliothèques non compatibles avec PostgreSQL ne peuvent pas être gérées ainsi. Vous devriez pouvoir utiliser les capacités du système pour cela, tel que la variable d'environnement `LD_PRELOAD`.

En général, il est préférable de se référer à la documentation d'un module spécifique pour trouver le bon moyen permettant de charger le module.

`local_preload_libraries` (`string`)

Cette variable indique une ou plusieurs bibliothèques partagées chargées au début de la connexion. Elle contient une liste de noms de bibliothèques, séparés par des virgules, où chaque nom est interprété comme par la commande `LOAD`. Les espaces blancs entre les entrées sont ignorés. Placer le nom d'une bibliothèque entre guillemets doubles si vous avez besoin d'inclure des espaces blancs ou des virgules dans le nom. La valeur de ce paramètre n'est pris en compte qu'au début de la connexion. Les modifications ultérieures n'ont pas d'effet sur les connexions déjà établies. Si une bibliothèque indiquée est introuvable, la tentative de connexion échouera. Seuls les superutilisateurs peuvent modifier cette configuration.

Cette option est configurable par tout utilisateur. De ce fait, les bibliothèques pouvant être chargées sont restreintes à celles disponibles dans le sous-répertoire `plugins` du répertoire des bibliothèques de l'installation. C'est de la responsabilité de l'administrateur de s'assurer que seules des bibliothèques « sûres » y soient installées.) Les éléments de `local_preload_libraries` peuvent indiquer ce répertoire explicitement, par exemple `$libdir/plugins/mabibliotheque`, ou indiquer seulement le nom de la bibliothèque -- `mabibliotheque`, ce qui aurait le même effet que `$libdir/plugins/mabibliotheque`.

Le but de cette fonctionnalité est de permettre aux utilisateurs non privilégiés de charger des bibliothèques de débuggage ou de mesures de performances dans des sessions explicites sans avoir à exécuter manuellement une commande `LOAD`. À cette fin, une configuration classique de

ce paramètre serait d'utiliser la variable d'environnement PGOPTIONS sur le client ou d'utiliser la commande ALTER ROLE SET.

Néanmoins, sauf si un module est conçu spécifiquement pour être utilisé de cette façon par des utilisateurs non administrateurs, ceci n'est pas le bon paramétrage pour vous. Regardez plutôt session_preload_libraries.

`session_preload_libraries` (`string`)

Cette variable indique une ou plusieurs bibliothèques partagées chargées au début de la connexion. Elle contient une liste de noms de bibliothèques, séparés par des virgules, où chaque nom est interprété comme par la commande LOAD. Les espaces blancs entre les entrées sont ignorés. Placer le nom d'une bibliothèque entre guillemets doubles si vous avez besoin d'inclure des espaces blancs ou des virgules dans le nom. La valeur de ce paramètre n'est pris en compte qu'au début de la connexion. Les modifications ultérieures n'ont pas d'effet sur les connexions déjà établies. Si une bibliothèque indiquée est introuvable, la tentative de connexion échouera. Seuls les superutilisateurs et les utilisateurs disposant des droits SET adéquats peuvent modifier cette configuration.

Le but de cette fonctionnalité est de permettre le chargement de bibliothèques de débuggage ou de mesure de performances dans des sessions explicites sans avoir à exécuter manuellement une commande LOAD. Par exemple, auto_explain pourrait être activé pour toutes les sessions si un certain utilisateur se connecte, en configurant son compte avec la commande ALTER ROLE SET. De plus, ce paramètre peut être modifié sans avoir à redémarrer le serveur (les changements ne prennent effet que pour les connexions suivantes), donc il est plus facile d'ajouter de nouveaux modules de cette façon, même s'ils s'appliquent à toutes les sessions.

Contrairement à shared_preload_libraries, il n'y a pas vraiment un gros avantage en terme de performances à charger une bibliothèque en début de session plutôt qu'à sa première utilisation. Néanmoins, ceci n'est plus vrai si un système de pooling de connexions est mis en place.

`shared_preload_libraries` (`string`)

Cette variable indique une ou plusieurs bibliothèques partagées chargées au démarrage du serveur. Elle contient une liste de noms de bibliothèques, séparés par des virgules, où chaque nom est interprété comme par la commande LOAD. Les espaces blancs entre les entrées sont ignorés. Placer le nom d'une bibliothèque entre guillemets doubles si vous avez besoin d'inclure des espaces blancs ou des virgules dans le nom. La valeur de ce paramètre n'est pris en compte qu'au démarrage du serveur. Si une bibliothèque indiquée est introuvable, la tentative de démarrage échouera. Seuls les superutilisateurs peuvent modifier cette configuration.

Certaines bibliothèques ont besoin de réaliser certaines opérations qui ne peuvent se faire qu'au démarrage du processus postmaster, comme allouer de la mémoire partagée, réserver des verrous à faible poids, ou démarrer des background workers. Ces bibliothèques doivent être chargées au démarrage du serveur via ce paramètre. Voir la documentation de chaque bibliothèque pour les détails.

Les autres bibliothèques peuvent aussi être préchargées. En préchargeant une bibliothèque partagée, le temps de démarrage de la bibliothèque est évité lorsque la bibliothèque est utilisée pour la première fois. Néanmoins, le temps de démarrer chaque nouveau processus serveur pourrait augmenter légèrement, même si le processus n'utilise jamais cette bibliothèque. Donc ce paramètre est seulement recommandé pour les bibliothèques qui seront utilisées par la majorité des sessions. De plus, changer ce paramètre requiert un redémarrage du serveur, donc ce n'est pas le bon paramètre pour les tâches de débuggage par exemple. Utilisez session_preload_libraries pour cela.

> **Note**
>
> Sur les hôtes Windows, précharger une bibliothèque au démarrage du serveur ne réduira pas le temps nécessaire pour démarrer un nouveau processus serveur.

> Chaque processus serveur rechargera toutes les bibliothèques préchargées. Néanmoins, `shared_preload_libraries` est toujous utile sur les hôtes Windows pour les bibliothèques qui ont besoin de réaliser des opérations au démarrage du postmaster.

`jit_provider` (`string`)

Cette variable contient le nom de la bibliothèque du fournisseur JIT à utiliser (voir Section 32.4.2). La valeur par défaut est `llvmjit`. Ce paramètre n'est configurable qu'au démarrage du serveur.

Si ce paramètre pointe vers une bibliothèque inexistante, JIT ne sera pas disponible, mais aucune erreur ne sera levée. Cela permet à l'infrastructure de JIT d'être installée séparément de l'installation PostgreSQL principale.

`gin_pending_list_limit` (`integer`)

Positionne la taille maximale de la liste d'attente GIN qui est utilisée lorsque `fastupdate` est activé. Si la liste dépasse cette taille maximale, elle est allégée en déplaçant des entrées en masse vers la structure de données principale GIN. Si cette valeur est indiquée sans unité, elle est comprise comme un nombre de Ko. La valeur par défaut est quatre mégaoctets (`4MB`). Ce paramètre peut être surchargé pour chaque index GIN en modifiant les paramètres de stockage de l'index. Voir Section 70.4.1 et Section 70.5 pour plus d'informations.

`restrict_nonsystem_relation_kind` (`string`)

Configure le type de relation dont l'accès est restreint pour les relations utilisateurs. La valeur configurée doit être une liste séparée par des virgules de type de relation. Actuellement, les types de relation acceptés sont `view` et `foreign-table`.

20.11.3. Locale et formatage

`datestyle` (`string`)

Configure le format d'affichage des valeurs de type date et heure, ainsi que les règles d'interprétation des valeurs ambiguës de dates saisies. Pour des raisons historiques, cette variable contient deux composantes indépendantes : la spécification du format en sortie (`ISO`, `Postgres`, `SQL` ou `German`) et la spécification en entrée/sortie de l'ordre année/mois/jour (`DMY`, `MDY` ou `YMD`). Elles peuvent être configurées séparément ou ensemble. Les mots clés `Euro` et `European` sont des synonymes de `DMY` ; les mots clés `US`, `NonEuro` et `NonEuropean` sont des synonymes de `MDY`. Voir la Section 8.5 pour plus d'informations. La valeur par défaut est `ISO, MDY`, mais initdb initialise le fichier de configuration avec une valeur qui correspond au comportement de la locale `lc_time` choisie.

`IntervalStyle` (`enum`)

Positionne le format d'affichage pour les valeurs de type intervalle. La valeur `sql_standard` produira une sortie correspondant aux litéraux d'intervalles du standard SQL. La valeur `postgres` (qui est la valeur par défaut) produira une sortie correspondant à celle des versions de PostgreSQL antérieures à la 8.4 quand le paramètre datestyle était positionné à `ISO`. La valeur `postgres_verbose` produira une sortie correspondant à celle des versions de PostgreSQL antérieures à la 8.4 quand le paramètre `DateStyle` était positionné à une valeur autre que `ISO` La valeur `iso_8601` produira une sortie correspondant au « format avec designateurs » d'intervalle de temps défini dans le paragraphe 4.4.3.2 de l'ISO 8601.

Le paramètre `IntervalStyle` affecte aussi l'interprétation des entrées ambigües d'intervalles. Voir Section 8.5.4 pour plus d'informations.

`TimeZone` (`string`)

Configure le fuseau horaire pour l'affichage et l'interprétation de la date et de l'heure. La valeur par défaut est `GMT`, mais elle est généralement surchargée dans le fichier `postgresql.conf` ;

initdb installera une configuration correspondant à l'environnement système. Voir Section 8.5.3 pour plus d'informations.

`timezone_abbreviations` (`string`)

Configure la liste des abréviations de fuseaux horaires acceptés par le serveur pour la saisie de données de type datetime. La valeur par défaut est `'Default'`, qui est une liste qui fonctionne presque dans le monde entier ; il y a aussi `'Australia'` et `'India'`. D'autres listes peuvent être définies pour une installation particulière. Voir Section B.4 pour plus d'informations.

`extra_float_digits` (`integer`)

Ce paramètre ajuste le nombre de chiffres utilisés par l'affichage textuel par les valeurs à virgule flottante, ce qui inclut `float4`, `float8` et les types de données géométriques.

Si la valeur est 1 (valeur par défaut) ou au-dessus, les valeurs à virgule flottante sont renvoyées dans le format le plus court et le plus précis ; voir Section 8.1.3. Le nombre réel de chiffres générés dépends seulement de la valeur en sortie, et non pas de la valeur de ce paramètre. Au plus 17 chiffres sont requis pour les valeurs `float8` et 9 pour les valeurs `float4`. Ce format est à la fois rapide et précis, préservant exactement la valeur flottante binaire originale lorsqu'elle est correctement lue. Pour la compabilité historique, des valeurs jusqu'à 3 sont autorisées.

Si la valeur est zéro ou négative, alors la sortie est arrondie avec une précision décimale donnée. La précision utilisée est le nombre standard de chiffres pour le type (`FLT_DIG` ou `DBL_DIG` comme approprié) réduit suivant la valeur de ce paramètre. (Par exemple, indiquer -1 fera que les valeurs `float4` soient arrondies en sortie à 5 chiffres significatifs et les valeurs `float8` à 14 chiffres.) Ce format est plus lent et ne préserve pas tous les buts de la valeur flottante binaire, mais pourrait être plus facile à lire.

> **Note**
>
> La signification de ce paramètre, et de sa valeur par défaut, a changé avec PostgreSQL 12 ; voir Section 8.1.3 pour plus de discussions.

`client_encoding` (`string`)

Initialise l'encodage client (jeu de caractères). Par défaut, il s'agit de celui de la base de données. Les ensembles de caractères supportés par PostgreSQL sont décrits dans Section 24.3.1.

`lc_messages` (`string`)

Initialise la langue d'affichage des messages. Les valeurs acceptables dépendent du système ; voir Section 24.1 pour plus d'informations. Si cette variable est initialisée à une chaîne vide (valeur par défaut), alors la valeur est héritée de l'environnement d'exécution du serveur.

Avec certains systèmes, cette catégorie de locale n'existe pas. Initialiser cette variable fonctionne toujours mais n'a aucun effet. De même, il est possible qu'il n'existe pas de traduction des messages dans la langue sélectionnée. Dans ce cas, les messages sont affichés en anglais.

Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier ce paramètre.

`lc_monetary` (`string`)

Initialise la locale à utiliser pour le formatage des montants monétaires (pour la famille de fonctions `to_char`, par exemple). Les valeurs acceptables dépendent du système ; voir la Section 24.1 pour plus d'informations. Si cette variable est initialisée à une chaîne vide (valeur par défaut), alors la valeur est héritée de l'environnement d'exécution du serveur, et une valeur incorrecte pourrait dégrader la lisibilité des traces du serveur.

`lc_numeric` (`string`)

Initialise la locale à utiliser pour le formatage des nombres (pour la famille de fonctions `to_char`, par exemple). Les valeurs acceptables dépendent du système ; voir la Section 24.1 pour plus d'informations. Si cette variable est initialisée à une chaîne vide (valeur par défaut), alors la valeur est héritée de l'environnement d'exécution du serveur.

`lc_time` (`string`)

Initialise la locale à utiliser pour le formatage des valeurs de date et d'heure, par exemple avec la famille de fonctions `to_char`. Les valeurs acceptables dépendent du système ; voir la Section 24.1 pour plus d'informations. Si cette variable est initialisée à une chaîne vide (valeur par défaut), alors la valeur est héritée de l'environnement d'exécution du serveur.

`default_text_search_config` (`string`)

Sélectionne la configuration de recherche plein texte utilisée par les variantes des fonctions de recherche plein texte qui n'ont pas d'argument explicite pour préciser la configuration. Voir Chapitre 12 pour plus d'informations. La valeur par défaut est `pg_catalog.simple` mais initdb initialise le fichier de configuration avec une valeur qui correspond à la locale choisie pour `lc_ctype` s'il est possible d'identifier une configuration correspondant à la locale.

20.11.4. Autres valeurs par défaut

`dynamic_library_path` (`string`)

Chemin de recherche utilisé lorsqu'un module chargeable dynamiquement doit être ouvert et que le nom de fichier indiqué dans la commande `CREATE FUNCTION` ou `LOAD` ne contient pas d'indication de répertoire (c'est-à-dire que le nom ne contient pas de slash).

La valeur de `dynamic_library_path` doit être une liste de chemins absolus séparés par des virgules (ou des points virgules sous Windows). Si un élément de la liste débute par la chaîne spéciale `$libdir`, le répertoire des bibliothèques internes du paquetage PostgreSQL est substitué à `$libdir`. C'est l'emplacement où sont installés les modules fournis par la distribution PostgreSQL standard. (La commande `pg_config --pkglibdir` permet de connaître le nom de ce répertoire.) Par exemple :

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/
my_project/lib:$libdir'
```

ou dans un environnement Windows :

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;
$libdir'
```

Pour plus d'informations sur la gestion des schémas, voir Section 5.9. En particulier, la configuration par défaut est seulement convenable quand la base de données a un seul utilisateur ou quelques utilisateurs qui se font confiance mutuellement.

La valeur par défaut de ce paramètre est `'$libdir'`. Si la valeur est une chaîne vide, la recherche automatique du chemin est désactivée.

Ce paramètre peut être modifié à l'exécution par les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats, mais un tel paramétrage ne persiste que pour la durée de la connexion du client. Il est donc préférable de ne réserver cette méthode qu'à des fins de développement. Il est recommandé d'initialiser ce paramètre dans le fichier de configuration `postgresql.conf`.

`gin_fuzzy_search_limit` (`integer`)

Limite souple haute de la taille de l'ensemble renvoyé par un index GIN. Pour plus d'informations, voir Section 70.5.

20.12. Gestion des verrous

`deadlock_timeout` (`integer`)

Durée totale d'attente d'un verrou avant de tester une condition de verrou mort (*deadlock*). Le test de verrou mort est très coûteux, le serveur ne l'effectue donc pas à chaque fois qu'il attend un verrou. Les développeurs supposent (de façon optimiste ?) que les verrous morts sont rares dans les applications en production et attendent simplement un verrou pendant un certain temps avant de lancer une recherche de blocage. Augmenter cette valeur réduit le temps perdu en recherches inutiles de verrous morts mais retarde la détection de vraies erreurs de verrous morts. Si cette valeur est indiquée sans unité, elle est comprise comme un nombre de millisecondes. La valeur par défaut est une seconde (`1s`), ce qui est probablement la plus petite valeur pratique. Sur un serveur en pleine charge, elle peut être augmentée. Idéalement, ce paramétrage doit dépasser le temps typique d'une transaction de façon à augmenter la probabilité qu'un verrou soit relâché avant que le processus en attente ne décide de lancer une recherche de verrous morts. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier cette configuration.

Quand log_lock_waits est configuré, ce paramètre détermine aussi la durée d'attente avant qu'un message ne soit enregistré dans les journaux concernant cette attente. Pour comprendre ces délais de verrouillage, il peut être utile de configurer `deadlock_timeout` à une valeur extraordinairement basse.

`max_locks_per_transaction` (`integer`)

La table des verrous partagés trace les verrous sur `max_locks_per_transaction` * (max_connections + max_prepared_transactions) objets (c'est-à-dire des tables) ; de ce fait, au maximum ce nombre d'objets distincts peuvent être verrouillés simultanément. Ce paramètre contrôle le nombre moyen de verrous d'objets alloués pour chaque transaction ; des transactions individuelles peuvent verrouiller plus d'objets tant que l'ensemble des verrous de toutes les transactions tient dans la table des verrous. Il *ne* s'agit *pas* du nombre de lignes qui peuvent être verrouillées ; cette valeur n'a pas de limite. La valeur par défaut, 64, s'est toujours avérée suffisante par le passé, mais il est possible de l'augmenter si des clients accèdent à de nombreuses tables différentes au sein d'une unique transaction, par exemple une requête sur une table parent ayant de nombreux enfants. Ce paramètre ne peut être initialisé qu'au lancement du serveur.

Lors de l'exécution d'un serveur en attente, vous devez configurer ce paramètre à la même valeur ou à une valeur plus importante que sur le serveur primaire. Sinon, des requêtes pourraient ne pas être autorisées sur le serveur en attente.

`max_pred_locks_per_transaction` (`integer`)

La table de verrous de prédicat partagée garde une trace des verrous sur `max_pred_locks_per_transaction` * (max_connections + max_prepared_transactions) objets (autrement dit tables). Du coup, pas plus que ce nombre d'objets distincts peut être verrouillé à un instant. Ce paramètre contrôle le nombre moyen de verrous d'objet alloués pour chaque transaction ; les transactions individuelles peuvent verrouillées plus d'objets à condition que les verrous de toutes les transactions tiennent dans la table des verrous. Ce n'est *pas* le nombre de lignes qui peuvent être verrouillées, cette valeur étant illimitée. La valeur par défaut, 64, a été généralement suffisante dans les tests mais vous pouvez avoir besoin d'augmenter cette valeur si vous avez des clients qui touchent beaucoup de tables différentes dans une seule transaction sérialisable. Ce paramètre n'est configurable qu'au lancement du serveur.

`max_pred_locks_per_relation` (`integer`)

Cela contrôle le nombre de pages ou de lignes d'une unique relation peut verrouiller au niveau du prédicat avant que le verrou soit promis pour couvrir l'intégralité de la relation. Des valeurs supérieures ou égales à zéro signifient une limite absolue, alors que des valeur négatives signifient max_pred_locks_per_transaction par la valeur absolue de ce paramètre. La valeur par défaut est -2, ce qui permet de conserver le comportement des anciennes version de PostgreSQL. Ce

paramètre peut uniquement être modifié dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`max_pred_locks_per_page` (`integer`)

Ceci contrôle combien de lignes sur une seule page peuvent être verrouillées avec prédicat avant que le verrou ne soit promu pour couvrir la page complète. La valeur par défaut est 2. Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

20.13. Compatibilité de version et de plateforme

20.13.1. Versions précédentes de PostgreSQL

`array_nulls` (`boolean`)

Contrôle si l'analyseur de saisie de tableau reconnaît `NULL` non-encadré par des guillemets comme élément de tableaux NULL. Activé par défaut (`on`), il autorise la saisie de valeurs NULL dans un tableau. Néanmoins, les versions de PostgreSQL antérieures à la 8.2 ne supportent pas les valeurs NULL dans les tableaux. De ce fait, ces versions traitent `NULL` comme une chaîne dont le contenu est « NULL ». Pour une compatibilité ascendante avec les applications nécessitant l'ancien comportement, ce paramètre peut être désactivé (`off`).

Il est possible de créer des valeurs de tableau contenant des valeurs NULL même quand cette variable est à `off`.

`backslash_quote` (`enum`)

Contrôle si un guillemet simple peut être représenté par un `\'` dans une chaîne. Il est préférable, et conforme au standard SQL, de représenter un guillemet simple en le doublant (`''`) mais, historiquement, PostgreSQL a aussi accepté `\'`. Néanmoins, l'utilisation de `\'` présente des problèmes de sécurité car certains encodages client contiennent des caractères multi-octets dans lesquels le dernier octet est l'équivalent ASCII numérique d'un \. Si le code côté client ne fait pas un échappement correct, alors une attaque par injection SQL est possible. Ce risque peut être évité en s'assurant que le serveur rejette les requêtes dans lesquelles apparaît un guillemet échappé avec un antislash. Les valeurs autorisées de `backslash_quote` sont `on` (autorise `\'` en permanence), `off` (le rejette en permanence) et `safe_encoding` (ne l'autorise que si l'encodage client n'autorise pas l'ASCII \ dans un caractère multioctet). `safe_encoding` est le paramétrage par défaut.

Dans une chaîne littérale conforme au standard, \ ne signifie que \. Ce paramètre affecte seulement la gestion des chaînes non conformes, incluant la syntaxe de chaînes d'échappement (`E'...'`).

`escape_string_warning` (`boolean`)

S'il est activé (`on`), un message d'avertissement est affiché lorsqu'un antislash (\) apparaît dans une chaîne littérale ordinaire (syntaxe `'...'`) et que `standard_conforming_strings` est désactivé. Il est activé par défaut (`on`).

Les applications qui souhaitent utiliser l'antislash comme échappement doivent être modifiées pour utiliser la syntaxe de chaîne d'échappement (`E'...'`) car le comportement par défaut des chaînes ordinaires est maintenant de traiter les antislashs comme un caractère ordinaire, d'après le standard SQL. Cette variable peut être activée pour aider à localiser le code qui doit être changé

`regex_flavor` (`enum`)

La « flaveur » des expressions rationnelles peut être configurée à `advanced` (avancée), `extended` (étendue) ou `basic` (basique). La valeur par défaut est `advanced`. La configuration `extended` peut être utile pour une compatibilité ascendante avec les versions antérieures à PostgreSQL 7.4. Voir Section 9.7.3.1 pour plus de détails.

`lo_compat_privileges` (`boolean`)

Dans les versions antérieures à la 9.0, les « Large Objects » n'avaient pas de droits d'accès et étaient, en réalité, toujours lisibles et modifiables par tous les utilisateurs. L'activation de cette variable désactive les nouvelles vérifications sur les droits, pour améliorer la compatibilité avec les versions précédentes. Désactivé par défaut, seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier ce paramètre.

Configurer cette variable ne désactive pas toutes les vérifications de sécurité pour les « Large Objects » -- seulement ceux dont le comportement par défaut a changé avec PostgreSQL 9.0.

`quote_all_identifiers` (`boolean`)

Quand la base de données génère du SQL, ce paramètre force tous les identifiants à être entre guillemets, même s'ils ne sont pas (actuellement) des mots-clés. Ceci affectera la sortie de la commande `EXPLAIN` ainsi que le résultat des fonctions comme `pg_get_viewdef`. Voir aussi l'option `--quote-all-identifiers` de pg_dump et pg_dumpall.

`standard_conforming_strings` (`boolean`)

Contrôle si les chaînes ordinaires (`'...'`) traitent les antislashs littéralement, comme cela est indiqué dans le standard SQL. À partir de PostgreSQL 9.1, ce paramètre est activé par défaut, donc à `on` (les versions précédentes avaient `off` par défaut). Les applications peuvent vérifier ce paramètre pour déterminer la façon dont elles doivent traiter les chaînes littérales. La présence de ce paramètre indique aussi que la syntaxe de chaîne d'échappement (`E'...'`) est supportée. La syntaxe de chaîne d'échappement (Section 4.1.2.2) doit être utilisée pour les applications traitant les antislashs comme des caractères d'échappement.

`synchronize_seqscans` (`boolean`)

Cette variable permet la synchronisation des parcours séquentiels de grosses tables pour que les parcours concurrents lisent le même bloc à peu près au même moment, et donc partagent la charge d'entrées/sorties. Quand ce paramètre est activé, un parcours peut commencer au milieu de la table, aller jusqu'à la fin, puis « revenir au début » pour récupérer toutes les lignes, ce qui permet de le synchroniser avec l'activité de parcours déjà entamés. Il peut en résulter des modifications non prévisibles dans l'ordre des lignes renvoyées par les requêtes qui n'ont pas de clause `ORDER BY`. Désactiver ce paramètre assure un comportement identique aux versions précédant la 8.3 pour lesquelles un parcours séquentiel commence toujours au début de la table. Activé par défaut (`on`).

20.13.2. Compatibilité entre la plateforme et le client

`transform_null_equals` (`boolean`)

Lorsque ce paramètre est activé (`on`), les expressions de la forme *`expr`* `= NULL` (ou `NULL =` *`expr`*) sont traitées comme *`expr`* `IS NULL`, c'est-à-dire qu'elles renvoient vrai si *`expr`* s'évalue à la valeur NULL, et faux sinon. Le bon comportement, compatible avec le standard SQL, de *`expr`* `= NULL` est de toujours renvoyer NULL (inconnu). De ce fait, ce paramètre est désactivé par défaut.

Toutefois, les formulaires filtrés dans Microsoft Access engendrent des requêtes qui utilisent *`expr`* `= NULL` pour tester les valeurs NULL. Il peut donc être souhaitable, lorsque cette intarface est utilisée pour accéder à une base de données, d'activer ce paramètre. Comme les expressions de la forme *`expr`* `= NULL` renvoient toujours la valeur NULL (en utilisant l'interprétation du standard SQL), elles ne sont pas très utiles et n'apparaissent pas souvent dans les applications normales. De ce fait, ce paramètre a peu d'utilité en pratique. Mais la sémantique des expressions

impliquant des valeurs NULL est souvent source de confusion pour les nouveaux utilisateurs. C'est pourquoi ce paramètre n'est pas activé par défaut.

Ce paramètre n'affecte que la forme exacte `= NULL`, pas les autres opérateurs de comparaison ou expressions équivalentes en terme de calcul à des expressions qui impliquent l'opérateur égal (tels que `IN`). De ce fait, ce paramètre ne doit pas être considéré comme un correctif général à une mauvaise programmation.

De plus amples informations sont disponibles dans la Section 9.2.

20.14. Gestion des erreurs

`exit_on_error` (`boolean`)

Si positionné à on, toute erreur terminera la session courante. Par défaut, ce paramètre est à off, pour que seules des erreurs de niveau FATAL puissent terminer la session.

Ce paramètre peut seulement être configuré dans le fichier `postgresql.conf` ou sur la ligne de commande du serveur.

`restart_after_crash` (`boolean`)

Quand ce paramètre est configuré à on, ce qui est sa valeur par défaut, PostgreSQL redémarrera automatiquement après un arrêt brutal d'un processus serveur. Il est généralement préférable de laisser cette valeur à on car cela maximise la disponibilité de la base de données. Néanmoins, dans certaines circonstances, comme le fait que PostgreSQL soit lancé par un outil de clustering, il pourrait être utile de désactiver le redémarrage pour que l'outil puisse avoir le contrôle et prendre toute action qui lui semble approprié.

`data_sync_retry` (`boolean`)

Lorsqu'il est configuré à off, ce qui est la valeur par défaut, PostgreSQL lèvera une erreur de niveau PANIC en cas d'échec de synchronisation des fichiers de données modifiés sur le système de fichiers. Ceci causera le crash du serveur de bases de données. Ce paramètre peut seulement être configuré au lancement du serveur.

Sur certains systèmes d'exploitation, le statut des données dans le cache disque du noyau n'est pas connu après un échec de synrhconisation. Dans certains cas, ce statut peut être entièrement oublié, rendant risquée toute nouvelle tentative. La deuxième tentative pourrait être rapportée comme réussi alors qu'en fait la donnée a été perdue. Dans ces circonstances, la seule façon d'éviter une perte de données est de rejouer les journaux de transactions après chaque statut d'échec, de préférence après une investigation sur la cause originale de l'échec et de remplacer tout matériel défectueux.

S'il est configuré à on, PostgreSQL renverra une erreur mais continuera à s'exécuter pour que l'opération de synchronisation sur disque soit tentée de nouveau au prochain checkpoint. Il faut le configurer à on après avoir investigué sur le traitement par le système d'exploitation des données en cache dans le cas d'un échec de synchronisation.

`recovery_init_sync_method` (`enum`)

Quand configuré à `fsync`, qui est la valeur par défaut, PostgreSQL va ouvrir récursivement et synchroniser tous les fichiers du répertoire data avant de commencer la récupération après un crash. Les fichiers accessibles par des liens symboliques dans le répertoire des journaux de transactions et dans les tablespaces sont également concernés, mais pas les autres liens symboliques. Cette opération permet de s'assurer que tous les journaux de transactions et fichiers de données sont stockés de manière durable sur disque avant de rejouer les changements. Ce comportement s'applique dès que le serveur n'a pas été arrêté proprement, ce qui inclue les copies faites avec pg_basebackup.

Sous Linux, `syncfs` peut être utilisé en lieu et place de `fsync` pour demander au système d'exploitation de synchroniser la totalité du système de fichiers qui contient le répertoire des données, des journaux de transactions et de chaque tablespace (mais pas d'autres systèmes de fichiers accessibles par des liens symboliques). Il est possible que cela soit beaucoup plus rapide que l'utilisation du paramètre `fsync`, car il n'est pas nécessaire d'ouvrir chaque fichier un par un. D'un autre côté, ce paramétrage pourra aussi être plus lent si le système de fichiers est partagé avec d'autres applications qui modifient beaucoup de fichiers, étant donné que ces fichiers seront aussi écrits sur le disque. De plus, sur les versions de Linux inférieure à la 5.8, les erreurs d'entrée/sortie rencontrées lors de l'écriture des données sur le disque peuvent ne pas être remontées à PostgreSQL et les messages d'erreurs correspondants pourraient n'apparaître que dans les fichiers de trace du noyau.

20.15. Options préconfigurées

Les « paramètres » suivants sont en lecture seule. De ce fait, ils sont exclus du fichier `postgresql.conf` d'exemple. Ces paramètres décrivent différents aspects du comportement de PostgreSQL qui peuvent s'avérer intéressants pour certaines applications, en particulier pour les interfaces d'administration. La plupart d'entre eux sont déterminés à la compilation ou à l'installation de PostgreSQL.

`block_size` (`integer`)

Informe sur la taille d'un bloc disque. Celle-ci est déterminée par la valeur de `BLCKSZ` à la construction du serveur. La valeur par défaut est de 8192 octets. La signification de diverses variables de configuration (shared_buffers, par exemple) est influencée par `block_size`. Voir la Section 20.4 pour plus d'informations.

`data_checksums` (`boolean`)

Informe sur l'activation des sommes de contrôle sur cette instance. Voir data checksums pour plus d'informations.

`data_directory_mode` (`integer`)

Sur les systèmes Unix, ce paramètre rapporte les droits sur le répertoire des données défini par data_directory au démarrage. (Sur Microsoft Windows, ce paramètre sera toujours `0700`). Voir group access pour plus d'information.

`debug_assertions` (`boolean`)

Indique si PostgreSQL a été compilé avec les assertions activées. C'est le cas si la macro `USER_ASSERT_CHECKING` est définie lorsque PostgreSQL est compilé (réalisé par exemple par l'option `--enable-cassert` de `configure`). Par défaut, PostgreSQL est compilé sans les assertions.

`integer_datetimes` (`boolean`)

Affiche si PostgreSQL a été compilé avec le support des date et heures en tant qu'entiers sur 64 bits. Depuis PostgreSQL 10, la valeur est toujours à `on`.

`in_hot_standby` (`boolean`)

Indique si le serveur est actuellement en mode hot standby. Quand ce paramètre est positionné sur `on`, toutes les transactions sont forcées en mode lecture seule. Au sein d'une session, il est possible de repasser en lecture-écriture uniquement si le serveur est promu primaire. Voir Section 27.4 pour plus d'informations.

`lc_collate` (`string`)

Affiche la locale utilisée pour le tri des données de type texte. Voir la Section 24.1 pour plus d'informations. La valeur est déterminée lors de la création d'une base de données.

`lc_ctype` (`string`)

Affiche la locale qui détermine les classifications de caractères. Voir la Section 24.1 pour plus d'informations. La valeur est déterminée lors de la création d'une base de données. Elle est habituellement identique à `lc_collate`. Elle peut, toutefois, pour des applications particulières, être configurée différemment.

`max_function_args` (`integer`)

Affiche le nombre maximum d'arguments des fonctions. Ce nombre est déterminé par la valeur de `FUNC_MAX_ARGS` lors de la construction du serveur. La valeur par défaut est de 100 arguments.

`max_identifier_length` (`integer`)

Affiche la longueur maximale d'un identifiant. Elle est déterminée à `NAMEDATALEN` - 1 lors de la construction du serveur. La valeur par défaut de `NAMEDATALEN` est 64 ; la valeur par défaut de `max_identifier_length` est, de ce fait, de 63 octets mais peut être moins de 63 caractères lorsque des encodages multi-octets sont utilisés.

`max_index_keys` (`integer`)

Affiche le nombre maximum de clés d'index. Ce nombre est déterminé par la valeur de `INDEX_MAX_KEYS` lors de la construction du serveur. La valeur par défaut est de 32 clés.

`segment_size` (`integer`)

Indique la taille des segments de journaux de transactions. La valeur par défaut est 16 Mo. Voir Section 30.5 pour plus d'informations.

`server_encoding` (`string`)

Affiche l'encodage de la base de données (jeu de caractères). Celui-ci est déterminé lors de la création de la base de données. Les clients ne sont généralement concernés que par la valeur de client_encoding.

`server_version` (`string`)

Affiche le numéro de version du serveur. Celui-ci est déterminé par la valeur de `PG_VERSION` lors de la construction du serveur.

`server_version_num` (`integer`)

Affiche le numéro de version du serveur sous la forme d'un entier. Celui-ci est déterminé par la valeur de `PG_VERSION_NUM` lors de la construction du serveur.

`shared_memory_size` (`integer`)

Indique la taille de la mémoire partagée principale, arrondie au Mo le plus proche.

`shared_memory_size_in_huge_pages` (`integer`)

Indique le nombre de huge pages nécessaires pour la mémoire partagée principale, en se basant sur le paramètre huge_page_size. Si les huge pages ne sont pas acceptés, cela vaudra `-1`.

Ce paramètre est uniquement accepté sur Linux. Il est toujours configuré à `-1` sur les autres plateformes. Pour plus de détails sur l'utilisation des huge pages on Linux, voir Section 19.4.5.

`ssl_library` (`string`)

Rapporte le nom de la bibliothèque SSL qui a été utilisée pour construire ce serveur PostgreSQL (même si SSL n'est pas actuellement configuré ou utilisé sur cette instance), par exemple `OpenSSL` ou une chaîne vide si aucune.

`wal_block_size` (`integer`)

Retourne la taille d'un bloc disque de WAL. C'est déterminé par la valeur `XLOG_BLCKSZ` à la compilation du serveur. La valeur par défaut est 8192 octets.

`wal_segment_size` (`integer`)

Retourne le nombre de blocs (pages) dans un fichier de segment WAL. La taille totale d'un fichier de segment WAL en octets est égale à `wal_segment_size` multiplié par `wal_block_size` ; Par défaut, c'est 16 Mo. Voir Section 30.5 pour plus d'informations.

20.16. Options personnalisées

Cette fonctionnalité a été conçue pour permettre l'ajout de paramètres habituellement inconnus de PostgreSQL par des modules complémentaires (comme les langages procéduraux). Cela permet de configurer ces extensions de façon standard.

Les options personnalisées ont des noms en deux parties : un nom d'extension, suivi d'un point, suivi du nom du paramètre, tout comme les noms qualifiés en SQL. Voici un exemple : `plpgsql.variable_conflict`.

Comme les options personnalisées peuvent avoir besoin d'être configurées par des processus qui n'ont pas chargé le module d'extension associé, PostgreSQL acceptera une configuration pour tout paramètre ayant un nom en deux parties. Ces variables sont traitées comme des espaces réservés et n'ont pas de fonction tant que le module qui les définit n'est pas chargé. Quand un module d'extension est chargé, il ajoute ses définitions de variables, et convertit les valeurs déjà initialisées suivant leur définition. S'il existe des variables non reconnues dont le nom commence par son nom d'extension, des avertissements sont envoyés et ces variables sont supprimées.

20.17. Options pour les développeurs

Les paramètres qui suivent ont été développés à des fins de test et ne doivent pas être utilisés sur une base de production. Cependant, certains d'entre eux peuvent aider à restaurer des bases de données corrompues. En tant que tel, ils sont exclus du fichier d'exemple de `postgresql.conf`. Un certain nombre d'entre eux requièrent des options de compilation spéciales pour fonctionner.

`allow_in_place_tablespaces` (`boolean`)

Autorise la création de tablespaces dans des répertoires à l'intérieur de `pg_tblspc`, quand une chaîne vide d'emplacement est fournie à la commande. Ceci a pour but de permettre de tester des scénarios de réplication où les serveurs primaire et secondaire sont exécutées sur la même machine. De tels répertoires pourraient gêner les outils de sauvegarde qui s'attendent à ne trouver que des lien symboliques dans ce répertoire. Seuls les superutilisateurs et les utilisateurs avec le droit `SET` approprié peuvent modifier cette configuration.

`allow_system_table_mods` (`boolean`)

Permet les modifications de la structure de tables systèmes ainsi que certaines autres actions risquées sur les tables systèmes. Ceci n'est autrement pas autorisé, y compris pour les superutilisateurs. Une mauvaise utilisation de ce paramètre peut causer des pertes de données irrécupérables ou une corruption sérieuse du système de bases de données. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier ce paramètre.

`backtrace_functions` (`string`)

Ce paramètre contient une liste de noms de fonctions C séparés par des virgules. Si une erreur est levée et que le nom d'une fonction C où une erreur survient correspond à une valeur de la liste,

alors la pile d'appel est écrite dans les traces du serveur avec le message d'erreur. Ceci peut être utilisé pour déboguer des aires spécifiques du code source.

Cette fonctionnalité n'est pas disponibles sur toutes les plateformes et la qualité des traces dépend des options de compilation.

Ce paramètre peut seulement être configuré par les superutilisateurs et les utilisateurs disposant des droits SET adéquats.

`debug_discard_caches` (`integer`)

Quand il est configuré à `1`, chaque entrée cache du catalogue système est invalidé dès la première opportunité, que ces entrées soient invalides ou pas. Le maintien en cache du catalogue système est de fait désactivé, le serveur sera donc plus lent. Des valeurs plus grandes permettent de faire tourner l'invalidation du cache de manière récursive, ce qui rendra le serveur encore plus lent. Ce paramètre n'est utile que pour tester la logique du cache elle-même. La valeur par défaut est `0` et permet d'avoir un comportement normal du cache du catalogue système.

Ce paramètre peut être très utile lorsqu'on essaye de reproduire un bug lié à des changements concurrents du catalogue système. Dans tous les autres cas, il est rarement utile. Lisez le code source de `inval.c` et `pg_config_manual.h` pour plus d'informations.

Ce paramètre est accepté quand `CLOBBER_CACHE_ENABLED` a été défini lors de la compilation (ce qui se fait automatiquement lorsqu'on utilise l'outil configure avec l'option `--enable-cassert`). Sur un système de production, la valeur de ce paramètre sera toujours `0` et tenter de le changer ramènera une erreur.

`force_parallel_mode` (`enum`)

Permet la parallélisation des requêtes à des fins de test, même dans les cas où aucune amélioration de performance n'est attendue. Les valeurs autorisées de `force_parallel_mode` sont `off` (la parallélisation n'est utilisée que si elle doit être bénéfique), `on`(la parallélisation est forcée pour toutes les requêtes pour lesquelles elle est considérée comme sans risque) et `regress` (comme `on`, mais avec quelques différences indiquées ci-dessous).

Positionner ce paramètre à `on` va ajouter un nœud de type `Gather` au début de chaque plan d'exécution lorsque cela n'est pas risqué, de manière à ce que la requête tourne au sein d'un processus parallèle. Ainsi, même lorsqu'un processus parallèle n'est pas disponible ou lorsqu'il ne peut pas être utilisé, pour la création de sous-transactions par exemple, un processus parallèle est créé à moins que le planificateur ne repère une opération ne pouvant pas être parallélisée. Si des erreurs ou des comportements inattendus sont constatés alors que ce paramètre est configuré à `on`, certaines fonctions utilisées dans la requête devront être marquées comme `PARALLEL UNSAFE` (ou sinon `PARALLEL RESTRICTED`).

Positionner ce paramètre à `regress` permet de bénéficier de tous les effets de la valeur `on` tout en ajoutant des fonctionnalités ayant pour but d'améliorer les tests de non régression automatisés. Normalement, les messages d'un processus de parallélisation incluent une ligne de contexte indiquant qu'il s'agit d'un processus de parallélisation, mais le positionnement du paramètre à `regress` supprime cette ligne de telle manière que la sortie est la même que pour une exécution non parallélisée. De plus, les nœuds de type `Gather` qui sont ajoutés au plan d'exécution sont cachées dans l'affichage de `EXPLAIN` de manière à ce que l'affichage soit le même que si le paramètre était positionné à `off`.

`ignore_system_indexes` (`boolean`)

Ignore les index système lors de la lecture des tables système (mais continue de les mettre à jour lors de modifications des tables). Cela s'avère utile lors de la récupération d'index système endommagés. Ce paramètre ne peut pas être modifié après le démarrage de la session.

`post_auth_delay` (`integer`)

Durée d'attente, après l'étape d'authentification, lorsqu'un nouveau processus serveur est lancé. Ceci a pour but de donner l'opportunité aux développeurs d'attacher un débogueur au processus serveur. Si cette valeur est indiquée sans unité, elle est comprise pour un nombre de secondes. La valeur par défaut, 0, désactive ce délai. Ce paramètre ne peut pas être modifié après le démarrage de la session.

`pre_auth_delay` (`integer`)

Durée d'attente, juste après la création d'un nouveau processus, avant le processus d'authentification. Ceci a pour but de donner une opportunité aux développeurs d'attacher un débogueur au processus serveur pour tracer les mauvais comportements pendant l'authentification. Si cette valeur est indiquée sans unité, elle est comprise pour un nombre de secondes. La valeur par défaut, 0, désactive ce délai. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`trace_notify` (`boolean`)

Produit un grand nombre de sorties de débogage pour les commandes `LISTEN` et `NOTIFY`. client_min_messages ou log_min_messages doivent être positionnées à `DEBUG1` ou plus bas pour envoyer cette sortie sur les traces client ou serveur, respectivement.

`trace_recovery_messages` (`enum`)

Contrôle les niveaux des traces écrites dans le journal applicatif pour les modules nécessaires lors du traitement de la restauration. Cela permet à l'utilisateur de surcharger la configuration normale de log_min_messages, mais seulement pour des messages spécifiques. Ça a été ajouté principalement pour débugger Hot Standby. Les valeurs valides sont `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1` et `LOG`. La valeur par défaut, `LOG`, n'affecte pas les décisions de trace. Les autres valeurs causent l'apparition de messages de débogage relatifs à la restauration pour tous les messages de ce niveau et des niveaux supérieurs. Elles utilisent malgré tout le niveau `LOG` ; pour les configurations habituelles de `log_min_messages`, cela résulte en un envoi sans condition dans les traces du serveur. Ce paramètre ne peut être configuré que dans le fichier `postgresql.conf` ou indiqué sur la ligne de commande.

`trace_sort` (`boolean`)

Si ce paramètre est actif, des informations concernant l'utilisation des ressources lors des opérations de tri sont émises. Ce paramètre n'est disponible que si la macro `TRACE_SORT` a été définie lors de la compilation de PostgreSQL (néanmoins, `TRACE_SORT` est actuellement définie par défaut).

`trace_locks` (`boolean`)

Si activé, émet des informations à propos de l'utilisation des verrous. L'information fournie inclut le type d'opération de verrouillage, le type de verrou et l'identifiant unique de l'objet verrouillé ou déverrouillé. Sont aussi inclus les masques de bits pour les types de verrous déjà accordés pour cet objet, ainsi que pour les types de verrous attendus sur cet objet. Pour chaque type de verrou un décompte du nombre de verrous accordés et en attente est aussi retourné, ainsi que les totaux. Un exemple de sortie dans le journal applicatif est montré ici :

```
LOG:  LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(2) req(1,0,0,0,0,0,0)=1 grant(1,0,0,0,0,0,0)=1
      wait(0) type(AccessShareLock)
LOG:  UnGrantLock: updated: lock(0xb7acd844)
 id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
```

```
        wait(0) type(AccessShareLock)
    LOG:  CleanUpLock: deleting: lock(0xb7acd844)
     id(24688,24696,0,0,0,1)
        grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
        wait(0) type(INVALID)
```

Les détails de la structure retournée peuvent être trouvés dans `src/include/storage/lock.h`.

Ce paramètre n'est disponible que si la macro `LOCK_DEBUG` a été définie quand PostgreSQL a été compilé.

`trace_lwlocks` (`boolean`)

Si à on, génère des informations à propos de l'utilisation de verrous légers (lightweight lock). Les verrous légers servent principalement à fournir une exclusion mutuelle d'accès aux structures de données en mémoire partagée.

Ce paramètre n'est disponible que si la macro `LOCK_DEBUG` a été définie quand PostgreSQL a été compilé.

`trace_userlocks` (`boolean`)

Si activé, génère des informations à propos de l'utilisation de verrous utilisateurs. La sortie est la même que pour `trace_locks`, mais restreinte aux verrous informatifs.

`trace_lock_oidmin` (`integer`)

Si positionné, ne trace pas les verrouillages pour des tables en dessous de cet OID. (à utiliser pour ne pas avoir de sortie pour les tables systèmes)

Ce paramètre n'est disponible que si la macro `LOCK_DEBUG` a été définie quand PostgreSQL a été compilé.

`trace_lock_table` (`integer`)

Tracer les verrouillages sur cette table de façon inconditionnelle.

Ce paramètre n'est disponible que si la macro `LOCK_DEBUG` a été définie quand PostgreSQL a été compilé.

`debug_deadlocks` (`boolean`)

Si positionné, génère des informations à propos de tous les verrous en cours quand l'expiration de temps d'attente d'un verrou mortel se produit.

Ce paramètre n'est disponible que si la macro `LOCK_DEBUG` a été définie quand PostgreSQL a été compilé.

`log_btree_build_stats` (`boolean`)

Si positionné, trace des statistiques d'utilisation de ressource système (mémoire et processeur) sur différentes opérations B-tree.

Ce paramètre n'est disponible que si la macro `BTREE_BUILD_STATS` a été définie quand PostgreSQL a été compilé.

`wal_consistency_checking` (`string`)

Ce paramètre est destiné à être utiliser pour vérifier la présence de bugs dans les routines d'application de WAL. Une fois activé, des images de l'intégralité des pages sont ajoutés aux

enregistrements. Si l'enregistrement est ensuite rejoué, le système appliquera d'abord chaque enregistrement et testera ensuite si les tampons modifiés par l'enregistrement correspond aux images stockées. Dans certains cas (comme les hint bits), des variations mineures sont acceptables, et seront ignorées. Toute différence inattendue provoquera une erreur fatale, ce qui arrêtera la restauration.

La valeur par défaut pour ce paramètre est une chaîne cide, ce qui désactive la fonctionnalité. Le paramètre peut être positionné à `all` pour vérifier tous les enregistrements, ou une liste séparée par des virgules de gestionnaires de sources afin de vérifier uniquement les enregistrements en fonction de ces gestionnaires de ressource. Actuellement, les gestionnaires de ressource supportés sont `heap`, `heap2`, `btree`, `hash`, `gin`, `gist`, `sequence`, `spgist`, `brin`, et `generic`. Les extensions peuvent définir des gestionnaires de ressources supplémentaires. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier ce paramètre.

`wal_debug` (`boolean`)

Si ce paramètre est positionné à `on`, une sortie de débogage relative aux WAL est émise. Ce paramètre n'est disponible que si la macro `WAL_DEBUG` a été définie au moment de la compilation de PostgreSQL.

`ignore_checksum_failure` (`boolean`)

Ne fonctionne que si data checksums est activé.

La détection d'un échec des sommes de vérification lors d'une lecture cause habituellement la levée d'une erreur par PostgreSQL, ce qui annule la transaction en cours. Activer `ignore_checksum_failure` fait que le système ignore l'échec (mais rapporte toujours un message d'avertissement) et continue le traitement. Ce comportement pourrait être la *cause d'arrêts brutaux, de propagation ou de dissimulation de corruption, ou d'autres problème sérieux*. Néanmoins, il peut vous permettre de dépasser l'erreur et de récupérer les lignes endommagées qui pourraient toujours être présentes dans la table si l'en-tête du bloc est sain. Si l'en-tête est corrompu, une erreur sera rapportée même si cette option est activée. La configuration par défaut est `off`, et elle ne peut être modifiée que par un superutilisateur et par les utilisateurs disposant des droits `SET` adéquats.

`zero_damaged_pages` (`boolean`)

La détection d'un en_tête de page endommagé cause normalement le renvoi d'une erreur par PostgreSQL, annulant du même coup la transaction en cours. Activer `zero_damaged_pages` fait que le système renvoie un message d'avertissement, efface la page endommagée en mémoire et continue son traitement. Ce comportement *détruit des données*, très exactement toutes les lignes comprises dans la page endommagée. Néanmoins, il vous permet de passer l'erreur et de récupérer les lignes des pages non endommagées qui pourraient être présentes dans la table. C'est intéressant pour récupérer des données si une corruption est survenue à cause d'une erreur logicielle ou matérielle. Vous ne devriez pas activer cette option sauf si vous avez perdu tout espoir de récupérer les données des pages endommagées d'une table. L'effacement des pages n'est pas vidée sur disque donc il est recommandé de recréer la table ou l'index avant de désactiver de nouveau ce paramètre. La configuration par défaut est `off`, et peut seulement être modifiée par un superutilisateur ou par un utilisateur disposant des droits `SET` adéquats.

`ignore_invalid_pages` (`boolean`)

Si configuré à `off` (la valeur par défaut), la détection d'enregistrements de journaux de transactions ayant des références à des pages invalides lors de la restauration fait que PostgreSQL lève une erreur de niveau PANIC, annulant la restauration. Configurer `ignore_invalid_pages` à `on` fait que le système ignore les références invalides de page dans les enregistrements des journaux de transactions (tout en renvoyant malgré tout un message d'avertissement) et continue la restauration. Ce comportement peut *causer des crashs, des pertes de données, propager ou cacher la corruption, ainsi que différents autres problèmes sérieux*.

Néanmoins, il peut vous permettre de passer l'erreur de niveau PANIC pour finir la restauration, et permettre ainsi au serveur de démarrer. Ce paramètre peut seulement être configuré au démarrage du serveur. Il a seulement un effet lors de la restauration et dans le mode standby.

`jit_debugging_support` (`boolean`)

Si LLVM en est capable, enregistre les fonctions générées auprès de GDB. Cela facilite le débogage. Le paramètrage par défaut est `off`. Seuls les superutilisateurs peuvent modifier ce paramètre au lancement d'une session, et il ne peut jamais être modifié à l'intérieur d'une session.

`jit_dump_bitcode` (`boolean`)

Écrit l'IR (*intermediate representation*) de LLVM dans le système de fichiers, dans data_directory. Ce n'est utile que pour travailler sur le fonctionnement interne de l'implémentation JIT. Le défaut est `off`. Seuls les superutilisateurs et les utilisateurs disposant des droits `SET` adéquats peuvent modifier ce paramètre.

`jit_expressions` (`boolean`)

Détermine si les expressions sont compilées par JIT quand la compilation JIT est activée (voir Section 32.2). La valeur par défaut est `on`.

`jit_profiling_support` (`boolean`)

Si LLVM le peut, pour que perf puisse profiler les fonctions générées par le JIT, écrit les données nécessaires dans des fichiers dans `~/.debug/jit/` ; l'utilisateur est responsable du nettoyage en temps voulu. Le paramétrage par défaut est `off`. Seuls les superutilisateurs peuvent modifier ce paramètre au lancement d'une session, et il ne peut jamais être modifié à l'intérieur d'une session.

`jit_tuple_deforming` (`boolean`)

Détermine si le décodage d'enregistrement est compilé par le JIT, quand la compilation JIT est activée (voir Section 32.2). Le défaut est `on`.

`remove_temp_files_after_crash` (`boolean`)

Quand initialisé à `on`, qui est la valeur par défaut, PostgreSQL va supprimer automatiquement les fichiers temporaires après un crash. Si désactivé, les fichiers vont rester et peuvent être utilisés pour le débogage, par exemple. De nombreux crashs peuvent mener cependant à l'accumulation de fichiers inutiles.

20.18. Options courtes

Pour des raisons pratiques, il existe également des commutateurs en ligne de commandes sur une seule lettre pour certains paramètres. Ceux-ci sont décrits dans le Tableau 20.4. Certaines des options existent pour des raisons historiques et leur présence en tant qu'option courte ne doit pas être vue comme une incitation à son utilisation massive.

Tableau 20.4. Clé d'option courte

Option courte	Équivalent
`-B` *x*	`shared_buffers = x`
`-d` *x*	`log_min_messages = DEBUG`*x*
`-e`	`datestyle = euro`
`-fb`, `-fh`, `-fi`, `-fm`, `-fn`, `-fo`, `-fs`, `-ft`	`enable_bitmapscan = off`, `enable_hashjoin = off`, `enable_indexscan = off`, `enable_mergejoin = off`, `enable_nestloop`

Option courte	Équivalent
	= off, enable_indexonlyscan = off, enable_seqscan = off, enable_tidscan = off
-F	fsync = off
-h *x*	listen_addresses = *x*
-i	listen_addresses = '*'
-k *x*	unix_socket_directories = *x*
-l	ssl = on
-N *x*	max_connections = *x*
-O	allow_system_table_mods = on
-p *x*	port = *x*
-P	ignore_system_indexes = on
-s	log_statement_stats = on
-S *x*	work_mem = *x*
-tpa, -tpl, -te	log_parser_stats = on, log_planner_stats = on, log_executor_stats = on
-W *x*	post_auth_delay = *x*

Chapitre 21. Authentification du client

Quand une application client se connecte au serveur de bases de données, elle indique le nom de l'utilisateur de base de données à utiliser pour la connexion, de la même façon qu'on se connecte à un ordinateur Unix sous un nom d'utilisateur particulier. Au sein de l'environnement SQL, le nom d'utilisateur de la base de données active détermine les droits régissant l'accès aux objets de la base de données -- voir le Chapitre 22 pour plus d'informations. Ainsi, il est essentiel de limiter le nombre de bases de données auxquelles les utilisateurs peuvent se connecter.

> **Note**
>
> Comme expliqué dans le Chapitre 22, PostgreSQL gère les droits par l'intermédiaire des « rôles ». Dans ce chapitre, le terme *utilisateur de bases de données* est utilisé pour signifier « rôle disposant du droit `LOGIN` ».

L'*authentification* est le processus par lequel le serveur de bases de données établit l'identité du client et, par extension, détermine si l'application client (ou l'utilisateur qui l'utilise) est autorisée à se connecter avec le nom d'utilisateur de bases de données indiqué.

PostgreSQL offre une quantité de méthodes d'authentification différentes. La méthode utilisée pour authentifier une connexion client particulière peut être sélectionnée d'après l'adresse (du client), la base de données et l'utilisateur.

Les noms d'utilisateur de bases de données sont séparés de façon logique des noms d'utilisateur du système d'exploitation sur lequel tourne le serveur. Si tous les utilisateurs d'un serveur donné ont aussi des comptes sur la machine serveur, il peut être pertinent d'attribuer aux utilisateurs de bases de données des noms qui correspondent à ceux des utilisateurs du système d'exploitation. Cependant, un serveur qui accepte des connexions distantes peut avoir des utilisateurs de bases de données dépourvus de compte correspondant sur le système d'exploitation. Dans ce cas, aucune correspondance entre les noms n'est nécessaire.

21.1. Le fichier `pg_hba.conf`

L'authentification du client est contrôlée par un fichier, traditionnellement nommé `pg_hba.conf` et situé dans le répertoire data du groupe de bases de données, par exemple `/usr/local/pgsql/data/pg_hba.conf` (HBA signifie « host-based authentication » : authentification fondée sur l'hôte.) Un fichier `pg_hba.conf` par défaut est installé lorsque le répertoire data est initialisé par initdb. Néanmoins, il est possible de placer le fichier de configuration de l'authentification ailleurs ; voir le paramètre de configuration hba_file.

Le format général du fichier `pg_hba.conf` est un ensemble d'enregistrements, un par ligne. Les lignes vides sont ignorées tout comme n'importe quel texte placé après le caractère de commentaire #. Un enregistrement peut être continué sur la ligne suivante en terminant la ligne avec un antislash. (Les antislashs ne sont pas spéciaux sauf à la fin d'une ligne.) Un enregistrement est constitué d'un certain nombre de champs séparés par des espace et/ou des tabulations. Les champs peuvent contenir des espaces si la valeur du champ est mise entre guillemets doubles. Mettre entre guillemets un des mots-clés dans un champ base de données, utilisateur ou adresse (par exemple, `all` ou `replication`) fait que le mot perd son interprétation spéciale, ou correspond à la base de données, à l'utilisateur ou à l'hôte ayant ce nom. La continuation d'une ligne avec un antislash s'applique même dans le texte entre guillemets et les commentaires.

Chaque enregistrement précise un type de connexion, une plage d'adresses IP (si approprié au type de connexion), un nom de base de données, un nom d'utilisateur et la méthode d'authentification à utiliser pour les connexions correspondant à ces paramètres. Le premier enregistrement qui correspond au type de connexion, à l'adresse client, à la base de données demandée et au nom d'utilisateur est utilisé

pour effectuer l'authentification. Il n'y a pas de suite après une erreur (« fall-through » ou « backup ») : si un enregistrement est choisi et que l'authentification échoue, les enregistrements suivants ne sont pas considérés. Si aucun enregistrement ne correspond, l'accès est refusé.

Un enregistrement peut avoir différents formats :

```
local         database  user  auth-method [auth-options]
host          database  user  address     auth-method  [auth-
options]
hostssl       database  user  address     auth-method  [auth-
options]
hostnossl     database  user  address     auth-method  [auth-
options]
hostgssenc    database  user  address     auth-method  [auth-
options]
hostnogssenc  database  user  address     auth-method  [auth-
options]
host          database  user  IP-address  IP-mask      auth-method
 [auth-options]
hostssl       database  user  IP-address  IP-mask      auth-method
 [auth-options]
hostnossl     database  user  IP-address  IP-mask      auth-method
 [auth-options]
hostgssenc    database  user  IP-address  IP-mask      auth-method
 [auth-options]
hostnogssenc  database  user  IP-address  IP-mask      auth-method
 [auth-options]
```

La signification des champs est la suivante :

`local`

Cet enregistrement intercepte les tentatives de connexion qui utilise les sockets du domaine Unix. Sans enregistrement de ce type, les connexions de sockets du domaine Unix ne sont pas autorisées.

`host`

Cet enregistrement intercepte les tentatives de connexion par TCP/IP. Les lignes `host` s'appliquent à toute tentative de connexion, SSL ou non, ainsi qu'à toute tentative de connexion GSSAPI chiffrée ou non.

> **Note**
>
> Les connexions TCP/IP ne sont pas autorisées si le serveur n'est pas démarré avec la valeur appropriée du paramètre de configuration listen_addresses. En effet, par défaut, le serveur n'écoute que les connexions TCP/IP en provenance de l'adresse `loopback` locale, `localhost`.

`hostssl`

Cet enregistrement intercepte les seules tentatives de connexions par TCP/IP qui utilisent le chiffrement SSL.

Pour utiliser cette fonction, le serveur doit être compilé avec le support de SSL. De plus, SSL doit être activé en positionnant le paramètre de configuration ssl (voir la Section 19.9 pour plus d'informations). Dans le cas contraire, l'enregistrement `hostssl` est ignoré à l'exception d'une alerte dans les traces indiquant qu'il n'y a aucune connexion correspondante.

`hostnossl`

Cet enregistrement a un comportement opposé à `hostssl` : il n'intercepte que les tentatives de connexion qui n'utilisent pas SSL.

`hostgssenc`

Cet enregistrement intercepte les tentatives de connexions par TCP/IP, mais seulement si la connexion est faite avec un chiffrement GSSAPI encryption.

Pour utiliser cette fonction, le serveur doit être compilé avec le support de GSSAPI activé. Dans le cas contraire, l'enregistrement `hostgssenc` est ignoré à l'exception d'une alerte dans les traces indiquant qu'il n'y a aucune connexion correspondante.

`hostnogssenc`

Cet enregistrement a un comportement opposé à `hostgssenc`; il n'intercepte que les tentatives de connexion qui n'utilisent pas le chiffrement GSSAPI.

`database`

Indique les noms des bases de données concernées par l'enregistrement. La valeur `all` indique qu'il concerne toutes les bases de données. Le terme `sameuser` indique que l'enregistrement coïncide si la base de données demandée a le même nom que l'utilisateur demandé. Le terme `samerole` indique que l'utilisateur demandé doit être membre du rôle portant le même nom que la base de données demandée (`samegroup` est obsolète bien qu'il soit toujours accepté comme écriture alternative de `samerole`.). Les superutilisateurs ne sont pas considérés comme membres d'un rôle dans le cadre de `samerole` à moins qu'ils ne soient explicitement membres du rôle, de manière directe ou indirecte, et non pas juste par ses droits de superutilisateur. La valeur `replication` indique que l'enregistrement établit une correspondance si une connexion de réplication physique est demandée. Néanmoins, il n'établit pas de correspondance pour les connexions de réplication logique. Notez que les connexions de réplication ne ciblent pas une base de données particulière alors que les connexions de réplication logique le font. Dans tous les autres cas, c'est le nom d'une base de données particulière. Plusieurs noms de base de données peuvent être fournis en les séparant par des virgules. Un fichier contenant des noms de base de données peut être indiqué en faisant précéder le nom du fichier de `@`.

`user`

Indique les utilisateurs de bases de données auxquels cet enregistrement correspond. La valeur `all` indique qu'il concerne tous les utilisateurs. Dans le cas contraire, il s'agit soit du nom d'un utilisateur spécifique de bases de données ou d'un nom de groupe précédé par un `+` (il n'existe pas de véritable distinction entre les utilisateurs et les groupes dans PostgreSQL ; un `+` signifie exactement « établit une correspondance pour tous les rôles faisant parti directement ou indirectement de ce rôle » alors qu'un nom sans `+` établit une correspondance avec ce rôle spécifique). Ainsi, un superutilisateur n'est considéré comme membre d'un rôle que s'il est explicitement membre du rôle, directement ou indirectement, et non pas juste par ses droits de superutilisateur. Plusieurs noms d'utilisateurs peuvent être fournis en les séparant par des virgules. Un fichier contenant des noms d'utilisateurs peut être indiqué en faisant précéder le nom du fichier de `@`.

`address`

Indique l'adresse IP ou la plage d'adresses IP à laquelle correspond cet enregistrement. Ce champ peut contenir soit un nom de machine (FQDN), soit le suffixe d'un domaine (sous la forme .exemple.com), soit une adresse ou une plage d'adresses IP, soit enfin l'un des mots-clés mentionnés ci-après.

Une plage d'adresses IP est spécifiée en utilisant la notation numérique standard (adresse de début de plage, suivi d'un slash (`/`) et suivi de la longueur du masque CIDR. La longueur du masque indique le nombre de bits forts pour lesquels une correspondance doit être trouvée avec l'adresse

IP du client. Les bits de droite doivent valoir zéro dans l'adresse IP indiquée. Il ne doit y avoir aucune espace entre l'adresse IP, le `/` et la longueur du masque CIDR.

À la place du *`CIDR-address`*, vous pouvez écrire `samehost` pour correspondre aux adresses IP du serveur ou `samenet` pour correspondre à toute adresse du sous-réseau auquel le serveur est directement connecté.

Une plage d'adresses IPv4 spécifiée au format CIDR est typiquement `172.20.143.89/32` pour un hôte seul, `172.20.143.0/24` pour un petit réseau ou `10.6.0.0/16` pour un réseau plus grand. Une plage d'adresses IPv6 spécifiée au format CIDR est par exemple `::1/128` pour un hôte seul (dans ce cas la boucle locale IPv6) ou `fe80::7a31:c1ff:0000:0000/96` pour un petit réseau. `0.0.0.0/0` représente toutes les adresses IPv4, et `::0/0` représente l'ensemble des adresses IPv6. Pour n'indiquer qu'un seul hôte, on utilise une longueur de masque de 32 pour IPv4 ou 128 pour IPv6. Dans une adresse réseau, ne pas oublier les zéros terminaux.

Une entrée donnée dans le format IPv4 correspondra seulement aux connexions IPv4, et une entrée donnée dans le format IPv6 correspondra seulement aux connexions IPv6, même si l'adresse représentée est dans la plage IPv4-in-IPv6. Notez que les entrées au format IPv6 seront rejetées si la bibliothèque C du système n'a pas de support des adresses IPv6.

La valeur `all` permet de cibler n'importe quelle adresse IP cliente, `samehost` n'importe quelle adresse IP du serveur ou `samenet` pour toute adresse IP faisant partie du même sous-réseau que le serveur.

Si un nom d'hôte est renseigné (dans les faits tout ce qui ne correspond pas à une plage d'adresse ou une plage d'adresses IP, ni à un mot clé, sera traité comme un nom d'hôte), ce nom est comparé au résultat d'une résolution de nom inverse de l'adresse IP du client (ou une recherche DNS inverse si un DNS est utilisé). Les comparaisons de noms d'hôtes ne sont pas sensibles à la casse. En cas de correspondance, une nouvelle recherche récursive de nom sera lancée afin de déterminer que le nom d'hôte concorde bel et bien avec l'adresse IP du client. L'enregistrement n'est validé qu'en cas de concordance entre la résolution inverse et la résolution récursive pour l'adresse IP cliente. (Le nom d'hôte fourni dans le fichier `pg_hba.conf` doit donc correspondre à au moins l'une des adresses IP fournies par le mécanisme de résolution de noms, sinon l'enregistrement ne sera pas pris en considération. Certains serveurs de noms réseau permettent d'associer une adresse IP à de multiples noms d'hôtes (alias DNS), mais bien souvent le système d'exploitation ne retourne qu'un seul nom d'hôte lors de la résolution d'une adresse IP.)

Un nom d'hôte débutant par un point (`.`) ciblera le suffixe du nom d'hôte du poste client. Du coup, indiquer `.exemple.com` correspondra à la machine `foo.exemple.com` (mais pas au client `exemple.com`).

Lorsque vous spécifiez des noms d'hôtes dans le fichier `pg_hba.conf`, vous devez vous assurer que la résolution de noms soit raisonnablement rapide. À défaut, il peut être avantageux de configurer un serveur-cache local pour effectuer la résolution de noms, tel que `nscd`. Vous pouvez également valider le paramètre de configuration `log_hostname` afin de retrouver dans les journaux le nom d'hôte du client au lieu de sa simple adresse IP.

Ces champs ne concernent pas les enregistrements `local`.

> **Note**
>
> Les utilisateurs se demandent parfois pourquoi les noms d'hôte sont gérés de cette manière apparemment si compliquée, avec deux résolutions de nom incluant une résolution inverse de l'adresse IP du client. Cela complique l'utilisation de cette fonctionnalité dans le cas où l'entrée de reverse-DNS n'est pas remplie ou retourne un nom d'hôte indésirable. Cela est fait essentiellement pour raison d'efficacité : de cette manière, une tentative de connexion nécessite au plus deux recherches de résolution, dont une inversée. S'il y a un problème de résolution avec une adresse, cela devient le problème du client. Une alternative d'implémentation hypothétique qui ne ferait pas de recherche inverse se

verrait obligée de résoudre chaque nom d'hôte mentionné dans `pg_hba.conf` à chaque tentative de connexion. Cela serait plutôt lent si de nombreux noms étaient listés. De plus, s'il y a un problème de résolution pour un seul des noms d'hôte, cela devient le problème de tout le monde.

De plus, une résolution inverse est nécessaire pour implémenter la fonctionnalité de correspondance par suffixe dans la mesure où le nom d'hôte du candidat à la connexion doit être connu afin de pouvoir effectuer cette comparaison.

Enfin, cette méthode est couramment adoptée par d'autres implémentations du contrôle d'accès basé sur les noms d'hôtes, tels que le serveur web Apache ou TCP-wrapper.

IP-address
IP-mask

Ces champs peuvent être utilisés comme alternative à la notation *adresse IP/longueur masque*. Au lieu de spécifier la longueur du masque, le masque réel est indiquée dans une colonne distincte. Par exemple, `255.0.0.0` représente une longueur de masque CIDR IPv4 de 8, et `255.255.255.255` représente une longueur de masque de 32.

Ces champs ne concernent pas les enregistrements `local`.

auth-method

Indique la méthode d'authentification à utiliser lors d'une connexion via cet enregistrement. Les choix possibles sont résumés ici ; les détails se trouvent dans la Section 21.3. Toutes les options sont en minuscules et traitées avec une sensibilité à la casse, donc même les acronymes comme `ldap` doivent être écrits en minuscule.

`trust`

Autorise la connexion sans condition. Cette méthode permet à quiconque peut se connecter au serveur de bases de données de s'enregistrer sous n'importe quel utilisateur PostgreSQL de son choix sans mot de passe ou autre authentification. Voir la Section 21.4 pour les détails.

`reject`

Rejette la connexion sans condition. Ce cas est utile pour « filtrer » certains hôtes d'un groupe, par exemple une ligne `reject` peut bloquer la connexion d'un hôte spécifique alors qu'une ligne plus bas permettra aux autres hôtes de se connecter à partir d'un réseau spécifique.

`scram-sha-256`

Réalise une authentification SCRAM-SHA-256 afin de vérifier le mot de passe utilisateur. Voir Section 21.5 pour les détails.

`md5`

Réalise une authentification SCRAM-SHA-256 ou MD5 afin de vérifier le mot de passe utilisateur. Voir Section 21.5 pour les détails.

`password`

Requiert que le client fournisse un mot de passe non chiffré pour l'authentification. Comme le mot de passe est envoyé en clair sur le réseau, ceci ne doit pas être utilisé sur des réseaux non dignes de confiance. Voir la Section 21.5 pour les détails.

`gss`

Utilise GSSAPI pour authentifier l'utilisateur. Disponible uniquement pour les connexions TCP/IP. Voir Section 21.6 pour les détails. Il peut être utilisé conjointement avec le chiffrement GSSAPI.

`sspi`

Utilise SSPI pour authentifier l'utilisateur. Disponible uniquement sur Windows. Voir Section 21.7 pour plus de détails.

`ident`

Récupère le nom de l'utilisateur en contactant le serveur d'identification sur le poste client, et vérifie que cela correspond au nom d'utilisateur de base de données demandé. L'authentification Ident ne peut être utilisée que pour les connexions TCP/IP. Pour les connexions locales, elle sera remplacée par l'authentification peer.

`peer`

Récupère le nom d'utilisateur identifié par le système d'exploitation du client et vérifie que cela correspond au nom d'utilisateur de base de données demandé. Peer ne peut être utilisée que pour les connexions locales. Voir la Section 21.9 ci-dessous pour les details.

`ldap`

Authentification par un serveur LDAP. Voir la Section 21.10 pour les détails.

`radius`

Authentification par un serveur RADIUS. Voir Section 21.11 pour les détails.

`cert`

Authentification par certificat client SSL. Voir Section 21.12 pour les détails.

`pam`

Authentification par les Pluggable Authentification Modules (PAM) fournis par le système d'exploitation. Voir la Section 21.13 pour les détails.

`bsd`

Authentification utilisant le service BSD Authentication fourni par le système d'exploitation. Voir Section 21.14 pour plus de détails.

`auth-options`

Après le champ *`auth-method`*, on peut trouver des champs de la forme *`nom=valeur`* qui spécifient des options pour la méthode d'authentification. Les détails sur les options disponibles apparaissent ci-dessous pour chaque méthode d'authentification.

En plus des options spécifiques à une méthode listées ci-dessous, il existe une option d'authentification indépendante de la méthode, appelée `clientcert`, qui peut être indiquée dans tout enregistrement `hostssl`. Cette option peut être positionnée à `verify-ca` ou `verify-full`. Ces deux options nécessitent que le client présente un certificat valide (de confiance), alors que `verify-full` valide également que le `cn` (Common Name) dans le certificat corresponde au nom d'utilisateur ou a une correspondance adéquate. Ce comportement est similaire à la méthode d'authentification `cert` (voir Section 21.12) mais autorise à appairer la vérification du certificat client avec toute méthode d'authentification qui supporte les entrées `hostssl`.

Pour tous les enregistrements utilisant une authentification par certificat client (autrement dit, ceux qui utilisent la méthode d'authentification `cert` ou ceux utilisant l'option `clientcert`), vous pouvez indiquer quel partie du certificat client doit faire correspondance en utilisant l'option `clientname`. Cette option doit avoir une valeur parmi deux. Si vous indiquez `clientname=CN`, qui est le comportement par défait, le nom de l'utilisateur doit établir

une correspondance avec le `Common Name (CN)` du certificat. Si vous indiquez à la place `clientname=DN`, le nom de l'utilisateur doit faire une correspondance avec le `Distinguished Name (DN)` entier du certificat. Cette option est probablement mieux utilisée en conjonction avec une carte de noms d'utilisateurs. La comparaison est faire avec le `DN` au format RFC 2253[1]. Pour voir le `DN` d'un certificat client dans ce format, faire

```
openssl x509 -in myclient.crt -noout -subject -nameopt RFC2253 |
 sed "s/^subject=//"
```

Il est important de faire attention lors de l'utilisation de cette option, tout spécialement lors de l'utilisation d'expression rationnelle avec le `DN`.

Les fichiers inclus par les constructions `@` sont lus comme des listes de noms, séparés soit par des espaces soit par des virgules. Les commentaires sont introduits par le caractère # comme dans `pg_hba.conf`, et les constructions `@` imbriquées sont autorisées. À moins que le nom du fichier qui suit `@` ne soit un chemin absolu, il est supposé relatif au répertoire contenant le fichier le référençant.

Les enregistrements du fichier `pg_hba.conf` sont examinés séquentiellement à chaque tentative de connexion, l'ordre des enregistrements est donc significatif. Généralement, les premiers enregistrements ont des paramètres d'interception de connexions stricts et des méthodes d'authentification peu restrictives tandis que les enregistrements suivants ont des paramètres plus larges et des méthodes d'authentification plus fortes. Par exemple, on peut souhaiter utiliser l'authentification `trust` pour les connexions TCP/IP locales mais demander un mot de passe pour les connexion TCP/IP distantes. Dans ce cas, l'enregistrement précisant une authentification `trust` pour les connexions issues de 127.0.0.1 apparaît avant un enregistrement indiquant une authentification par mot de passe pour une plage plus étendue d'adresses IP client autorisées.

Le fichier `pg_hba.conf` est lu au démarrage et lorsque le processus serveur principal reçoit un signal SIGHUP. Si le fichier est édité sur un système actif, on peut signaler au postmaster (en utilisant `pg_ctl reload`, en appelant la fonction SQL `pg_reload_conf()`, ou `kill -HUP`) de relire le fichier.

> **Note**
>
> L'information précédente n'est pas vraie sous Microsoft Windows : ici, tout changement dans le fichier `pg_hba.conf` est immédiatement appliqué à toute nouvelle connexion.

La vue système `pg_hba_file_rules` peut aider pour pré-tester les changements dans le fichier `pg_hba.conf`, ou pour diagnostiquer des problèmes si le rechargement du fichier n'a pas eu les effets escomptés. Les lignes dans la vue avec des champs `error` non vides indiquent des problèmes dans les lignes correspondantes du fichier.

> **Astuce**
>
> Pour se connecter à une base particulière, un utilisateur doit non seulement passer les vérifications de `pg_hba.conf` mais doit également avoir le droit `CONNECT` sur cette base. Pour contrôler qui peut se connecter à quelles bases, il est en général plus facile de le faire en donnant ou retirant le privilège `CONNECT` plutôt qu'en plaçant des règles dans le fichier `pg_hba.conf`.

Quelques exemples d'entrées de `pg_hba.conf` sont donnés ci-dessous dans l'Exemple 21.1. Voir la section suivante pour les détails des méthodes d'authentification.

[1] https://datatracker.ietf.org/doc/html/rfc2253

Exemple 21.1. Exemple d'entrées de `pg_hba.conf`

```
# Permettre à n'importe quel utilisateur du système local de se
 connecter
# à la base de données sous n'importe quel nom d'utilisateur au
 travers
# des sockets de domaine Unix (par défaut pour les connexions
 locales).
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
local   all             all
 trust

# La même chose en utilisant les connexions TCP/IP locales
 loopback.
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    all             all             127.0.0.1/32
 trust

# Pareil mais en utilisant une colonne netmask distincte.
#
# TYPE  DATABASE    USER        IP-ADDRESS          IP-mask
      METHOD
host    all         all         127.0.0.1           255.255.255.255
      trust

# Pareil mais en IPv6.
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    all             all             ::1/128
 trust

# À l'identique en utilisant le nom d'hôte (qui doit typiquement
 fonctionner en IPv4 et IPv6).
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    all             all             localhost
 trust

# Permettre à n'importe quel utilisateur de n'importe quel hôte
 d'adresse IP
# 192.168.93.x de se connecter à la base de données "postgres" sous
 le nom
# d'utilisateur qu'ident signale à la connexion (généralement le
# nom utilisateur du système d'exploitation).
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    postgres        all             192.168.93.0/24
 ident

# Permet à un utilisateur de l'hôte 192.168.12.10 de se connecter à
 la base de
```

```
# données "postgres" si le mot de passe de l'utilisateur est
 correctement fourni.
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    postgres        all             192.168.12.10/32
 scram-sha-256

# Permet la connexion à n'importe quel utilisateur depuis toutes
 les machines du
# domaine exemple.com à n'importe quelle base de données si le mot
 de passe
# correct est fourni.
#
# Exige l'authentification SCRAM pour la plupart des utilisateurs,
 exception
# faite pour l'utilisateur "mike", qui utilise un client plus
 ancien ne prenant
# pas en charge l'authentification SCRAM.
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    all             mike            .example.com            md5
host    all             all             .example.com
 scram-sha-256

# Si aucune ligne "host" ne précède, ces trois lignes rejettent
 toutes
# les connexions en provenance de 192.168.54.1 (puisque cette
 entrée déclenche
# en premier), mais autorisent les connexions chiffrées GSSAPI de
 n'importe où
# ailleurs sur l'Internet. Le masque zéro signifie qu'aucun bit de
 l'ip de
# l'hôte n'est considéré, de sorte à correspondre à tous les hôtes.
 Les
# connexions GSSAPI non chiffrée (qui "échouent" jusqu'à la
 troisième ligne
# puisque "hostgssenc" ne correspond qu'aux connections GSSAPI
 chiffrées) sont
# autorisées, mais seulement depuis 192.168.12.10.
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    all             all             192.168.54.1/32
 reject
hostgssenc all          all             0.0.0.0/0               gss
host    all             all             192.168.12.10/32        gss

# Permettre à tous les utilisateurs de se connecter depuis
 192.168.x.x à n'importe
# quelle base de données s'ils passent la vérification
 d'identification. Si,
# par exemple, ident indique que l'utilisateur est "bryanh" et
 qu'il
# demande à se connecter en tant qu'utilisateur PostgreSQL
 "guest1", la
```

```
# connexion n'est permise que s'il existe une entrée dans
 pg_ident.conf pour la
# correspondance "omicron" disant que "bryanh" est autorisé à se
 connecter en
# tant que "guest1".
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    all             all             192.168.0.0/16
 ident map=omicron

# Si ces trois lignes traitent seules les connexions locales, elles
# n'autorisent les utilisateurs locaux qu'à se connecter à leur
 propre
# base de données (base ayant le même nom que leur nom
# d'utilisateur) exception faite des administrateurs
# et des membres du rôle "support" qui peuvent se connecter à
 toutes les bases
# de données. Le fichier $PGDATA/admins contient une liste de noms
# d'administrateurs. Un mot de passe est requis dans tous les cas.
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
local   sameuser        all                                     md5
local   all             @admins                                 md5
local   all             +support                                md5

# Les deux dernières lignes ci-dessus peuvent être combinées en une
 seule ligne :
local   all         @admins,+support                md5

# La colonne database peut aussi utiliser des listes et des noms de
 fichiers :
local   db1,db2,@demodbs  all                       md5
```

21.2. Correspondances d'utilisateurs

Lorsqu'on utilise une authentification externe telle que Ident ou GSSAPI, le nom de l'utilisateur du système d'exploitation qui a initié la connexion peut ne pas être le même que celui de l'utilisateur de la base à laquelle il tente de se connecter. Dans ce cas, une table de correspondance d'identités peut être mise en place pour faire correspondre le nom d'utilisateur système au nom d'utilisateur base de donnée. Pour utiliser une table de correspondance d'identités, spécifiez `map=`*`nom-table`* dans le champ options de `pg_hba.conf`. Cette option est supportée pour toutes les méthodes d'authentification qui reçoivent des noms d'utilisateurs externes. Comme différentes correspondances peuvent être nécessaires pour différentes connexions, le nom de la table à utiliser doit être spécifié dans le paramètre *`nom-table`* de `pg_hba.conf` afin d'indiquer quelle table utiliser pour chaque connexion.

Les tables de correspondance de noms d'utilisateurs sont définies dans le fichier de correspondance, qui par défaut s'appelle `pg_ident.conf` et est stocké dans le répertoire de données du cluster. (Toutefois, il est possible de placer la table de correspondance ailleurs ; voir le paramètre de configuration ident_file.) Le fichier de table de correspondance contient des lignes de la forme suivante :

`nom-table nom-d-utilisateur-systeme nom-d-utilisateur-base`

Les commentaires, les blancs et les continuations de ligne sont traités de la même façon que dans `pg_hba.conf`. Le *`nom-table`* est un nom arbitraire qui sera utilisé pour faire référence à cette table de correspondance dans `pg_hba.conf`. Les deux autres champs spécifient un nom d'utilisateur du système d'exploitation et un nom d'utilisateur de la base de données correspondant. Le même *`nom-correspondance`* peut être utilisé de façon répétée pour indiquer plusieurs correspondances d'utilisateur dans la même carte.

Il n'y a aucune restriction sur le nombre d'utilisateurs de base de données auxquels un utilisateur du système d'exploitation peut correspondre et vice-versa. Du coup, les entrées dans une carte signifient que « cet utilisateur du système d'exploitation est autorisé à se connecter en tant que cet utilisateur de la base de données », plutôt que supposer qu'ils sont équivalents. La connexion sera autorisée s'il existe une entrée dans la carte qui correspond au nom d'utilisateur obtenu à partir du système d'authentification externe pour le nom de l'utilisateur de la base de données que l'utilisateur a indiqué.

Si le champ *`system-username`* commence avec un slash (`/`), le reste du champ est traité comme une expression rationnelle. (Voir Section 9.7.3.1 pour les détails de la syntaxe des expressions rationnelles avec PostgreSQL.). L'expression rationnelle peut inclure une copie simple ou une sous-expression entre parenthèses. La portion du nom d'utilisateur système qui correspond à la copie peut ensuite être référencée dans le champ *`database-username`* avec le joker `\1` (antislash-un). Ceci permet la correspondance de plusieurs noms d'utilisateurs sur une seule ligne, ce qui est particulièrement utile pour les substitutions simples. Par exemple, ces entrées

```
mymap   /^(.*)@mydomain\.com$       \1
mymap   /^(.*)@otherdomain\.com$    guest
```

supprimeront la partie domaine pour les utilisateurs de système d'exploitation dont le nom finissent avec `@mydomain.com`, et permettront aux utilisateurs dont le nom se termine avec `@otherdomain.com` de se connecter en tant que `guest`.

> **Astuce**
>
> Gardez en tête que, par défaut, une expression rationnelle peut correspondre à une petite partie d'une chaîne. Il est généralement conseillé d'utiliser les jokers `^` et `$`, comme indiqué dans l'exemple ci-dessus, pour forcer une correspondance sur le nom entier de l'utilisateur du système d'exploitation.

Le fichier `pg_ident.conf` est lu au démarrage et quand le processus principal du serveur reçoit un signal SIGHUP. Si vous éditez le fichier sur un système en cours d'utilisation, vous devez notifier le postmaster (en utilisant`pg_ctl reload`, en appelant la fonction SQL `pg_reload_conf()`, ou `kill -HUP`) pour lui faire relire le fichier.

La vue système `pg_ident_file_mappings` peut être utile pour tester des modifications du fichier `pg_ident.conf` et pour diagnostiquer des problèmes si le chargement du fichier n'a pas eu les effets désirés. Les lignes de la vue avec le champ `error` non NULL indiquent des problèmes dans les lignes correspondantes du fichier.

Un fichier `pg_ident.conf` qui pourrait être utilisé avec le fichier `pg_hba.conf` de Exemple 21.1 est montré en Exemple 21.2. Dans cet exemple, toute personne connectée sur une machine du réseau 192.168 qui n'a pas le nom d'utilisateur du système d'exploitation `bryanh`, `ann`, ou `robert` verrait son accès refusé. L'utilisateur Unix `robert` ne verrait son accès autorisé que lorsqu'il essaie de se connecter en tant qu'utilisateur PostgreSQL `bob`, pas en tant que `robert` ou qui que ce soit d'autre. `ann` ne serait autorisée à se connecter qu'en tant que `ann`. L'utilisateur `bryanh` aurait le droit de se connecter soit en tant que `bryanh`, soit en tant que `guest1`.

Exemple 21.2. Un exemple de fichier `pg_ident.conf`

```
# MAPNAME     SYSTEM-USERNAME   PG-USERNAME

omicron       bryanh            bryanh
omicron       ann               ann
# bob has user name robert on these machines
omicron       robert            bob
# bryanh can also connect as guest1
omicron       bryanh            guest1
```

21.3. Méthodes d'authentification

PostgreSQL fournit différentes méthodes pour l'authentification des utilisateurs :

- Authentification Trust, qui fait confiance aux utilisateurs à l'identification connue.
- Authentification Password, qui réclame un mot de passe aux utilisateurs.
- Authentification GSSAPI, qui se base sur une bibliothèque de sécurité compatible GSSAPI. C'est typiquement utilisé pour accéder à un serveur d'authentification tel que Kerberos ou Microsoft Active Directory server.
- Authentification SSPI, qui utilise un protocole spécifique Windows similaire à GSSAPI.
- Authentification Ident, qui se base sur le service « Identification Protocol » (RFC 1413[2]) de la machine cliente. (Pour des connexions locales par socket Unix, ceci est traité comme une authentification peer.)
- Authentification Peer, qui se base sur les capacités du système d'exploitation pour identifier le processus à l'autre bout d'une connexion locale. Ceci n'est pas supporté pour les connexions distantes.
- Authentification LDAP, qui se base sur un serveur d'authentification LDAP.
- Authentification RADIUS, un serveur d'authentification RADIUS.
- Authentification Certificate, qui nécessite une connexion SSL et authentifie les utilisateurs en vérifiant le certificat SSL qu'ils envoient.
- Authentification PAM, qui se base sur la bibliothèque PAM (Pluggable Authentication Modules).
- Authentification BSD, qui se base sur le système d'authentification BSD (actuellement seulement disponible sur OpenBSD).

L'authentification peer est recommandable généralement pour les connexions locales, même si l'authentification trust pourrait être suffisante dans certains contextes. L'authentification password est le choix le plus simple pour les connexions distantes. Toutes les autres options nécessitent une forme d'infrastructure de sécurité externe (habituellement un serveur d'authentification ou une autorité de certificat pour créer des certificats SSL) ou sont spécifiques à la plateforme.

Les sections suivantes décrivent chacune des méthodes d'authentification en détail.

21.4. Authentification trust

Quand l'authentification `trust` est utilisée, PostgreSQL considère que quiconque peut se connecter au serveur est autorisé à accéder à la base de données quel que soit le nom d'utilisateur de bases de données qu'il fournit (même les noms des superutilisateurs). Les restrictions apportées dans les colonnes `database` et `user` continuent évidemment de s'appliquer. Cette méthode ne doit être utilisée que si le système assure un contrôle adéquat des connexions au serveur.

[2] https://datatracker.ietf.org/doc/html/rfc1413

L'authentification `trust` est appropriée et très pratique pour les connexions locales sur une station de travail mono-utilisateur. Elle n'est généralement *pas* appropriée en soi sur une machine multi-utilisateur. Cependant, `trust` peut tout de même être utilisé sur une machine multi-utilisateur, si l'accès au fichier socket de domaine Unix est restreint par les permissions du système de fichiers. Pour ce faire, on peut positionner les paramètres de configuration `unix_socket_permissions` (et au besoin `unix_socket_group`) comme cela est décrit dans la Section 20.3. On peut également positionner le paramètre de configuration `unix_socket_directories` pour placer le fichier de socket dans un répertoire à l'accès convenablement restreint.

Le réglage des droits du système de fichiers n'a d'intérêt que le cas de connexions par les sockets Unix. Les droits du système de fichiers ne restreignent pas les connexions TCP/IP locales. Ainsi, pour utiliser les droits du système de fichiers pour assurer la sécurité locale, il faut supprimer la ligne `host ...127.0.0.1 ...` de `pg_hba.conf` ou la modifier pour utiliser une méthode d'authentification différente de `trust`.

L'authentification `trust` n'est envisageable, pour les connexions TCP/IP, que si chaque utilisateur de chaque machine autorisée à se connecter au serveur par les lignes `trust` du fichier `pg_hba.conf` est digne de confiance. Il est rarement raisonnable d'utiliser `trust` pour les connexions autres que celles issues de localhost (127.0.0.1).

21.5. Authentification par mot de passe

Il existe plusieurs méthodes d'authentification basées sur un mot de passe. Ces méthodes opèrent de façon similaire mais diffèrent sur la façon dont les mots de passe des utilisateurs sont enregistrés sur le serveur et comment le mot de passe fourni par un client est envoyé au serveur.

`scram-sha-256`

La méthode `scram-sha-256` réalise une authentification SCRAM-SHA-256, tel qu'elle est décrite dans la RFC 7677[3]. Il s'agit d'un système de question/réponse qui empêche la récupération du mot de passe sur des connexions non sécurisées et supporte l'enregistrement des mots de passe sur le serveur avec un hachage cryptographique normalement sécurisé.

C'est actuellement la méthode interne la plus sécurisée, mais elle n'est pas supportée par les anciennes bibliothèques clients.

`md5`

La méthode `md5` utilise un mécanisme de question/réponse moins sécurisé. Il empêche la récupération du mot de passe et évite d'enregistrer les mots de passe en clair sur le serveur, mais ne fournit aucune protection si l'attaquant réussit à voler le mot de passe haché du serveur. De plus, l'algorithme de hachage MD5 n'est plus considéré de nos jours comme suffisamment sécurisé avec des attaques déterminées.

La méthode `md5` ne peut pas être utilisé avec la fonctionnalité db_user_namespace.

Pour faciliter la transition de la méthode `md5` à la méthode SCRAM, si `md5` est indiqué comme méthode d'authentification dans `pg_hba.conf` mais que le mot de passe de l'utilisateur sur le serveur est chiffré avec SCRAM (voir ci-dessous), l'authentification SCRAM est automatiquement utilisée à la place.

`password`

La méthode `password` envoie le mot de passe en clair et est de ce fait vulnérable aux attaques de type « sniffing ». Elle doit être évitée chaque fois que possible. Si la connexion est protégée par le chiffrage SSL, alors `password` peut être utilisé de façon sécurisée. (Ceci étant dit, l'authentification par certificat SSL serait un meilleur choix en cas d'utilisation de SSL).

[3] https:///datatracker.ietf.org/doc/html/rfc7677

Les mots de passe PostgreSQL sont distincts des mots de passe du système d'exploitation. Le mot de passe de chaque utilisateur est enregistré dans le catalogue système `pg_authid`. Ils peuvent être gérés avec les commandes SQL CREATE ROLE et ALTER ROLE. Ainsi, par exemple, **`CREATE ROLE foo WITH LOGIN PASSWORD 'secret';`** ou la méta-commande `\password` de psql. Si aucun mot de passe n'est enregistré pour un utilisateur, le mot de passe enregistré est nul et l'authentification par mot de passe échoue systématiquement pour cet utilisateur.

La disponibilité des différentes méthodes d'authentification basées sur des mots de passe dépend de comment un mot de passe utilisateur est chiffré sur le serveur (ou haché pour être plus précis). Ceci est contrôlé par le paramètre de configuration password_encryption au moment où le mot de passe est configuré. Si un mot de passe est chiffré en utilisant le paramètre `scram- sha-256`, alors il peut être utilisé par les méthodes d'authentification `scram-sha-256` et `password` (mais la transmission du mot de passe sera en clair dans ce dernier cas). La méthode d'authentification `md5` sera automatiquement basculée vers la méthode `scram-sha-256` dans ce cas, comme expliqué ci-dessus, donc cela fonctionnera aussi. Si un mot de passe était chiffré en utilisant la configuration `md5`, alors il peut seulement être utilisé pour les méthodes d'authentification `md5` et `password` (de nouveau, avec le mot de passe transmis en clair dans ce dernier cas). (Les anciennes versions de PostgreSQL supportaient le stockage en clair du mot de passe sur le serveur. Ceci n'est plus possible.) Pour vérifier les hachages de mot de passe actuellement enregistrés, voir le catalogue système `pg_authid`.

Pour mettre à jour une installation existante de `md5` vers `scram-sha-256`, après s'être assuré que toutes les bibliothèques courantes sont suffisamment récentes pour supporter SCRAM, configurez `password_encryption = 'scram-sha-256'` dans `postgresql.conf`, demandez à chaque utilisateur de configurer un nouveau mot de passe, et modifiez la méthode d'authentification dans `pg_hba.conf` avec `scram-sha-256`.

21.6. Authentification GSSAPI

GSSAPI est un protocole du standard de l'industrie pour l'authentification sécurisée définie dans RFC 2743[4]. PostgreSQL supporte GSSAPI pour l'authentification, le chiffrement des communications, ou les deux. GSSAPI fournit une authentification automatique (single sign-on) pour les systèmes qui le supportent. L'authentification elle-même est sécurisée. Si le chiffrement GSSAPI ou le chiffrement SSL est utilisé, les données envoyées au travers de la connexion à la base de données seront chiffrées, sinon elles ne le seront pas.

Le support de GSSAPI doit être activé quand PostgreSQL est compilé ; voir Chapitre 17 pour plus d'informations.

Quand GSSAPI passe par Kerberos, il utilise un principal service standard au format *`nom_service/nom_hôte@domaine`*. Le nom utilisé pour le principal sur une installation particulière n'est pas encodé dans le serveur PostgreSQL ; il est en fait indiqué dans le fichier *keytab* que le serveur lit pour déterminer son identité. Si plusieurs principaux sont listés dans le fichier keytab, le serveur acceptera n'importe lequel parmi eux. Le nom de royaume du serveur est le royaume préféré indiqué dans le fichier de configuration Kerberos accessible au serveur.

Lors de la connexion, le client doit connaître le nom du principal du serveur où il souhaite se connecter. La partie *`servicename`* est habituellement `postgres`, mais une autre valeur peut être sélectionnée via le paramètre de connexion krbsrvname de libpq. La partie *`hostname`* est le nom d'hôte complètement qualifié auquel libpq se connecte. Le nom de royaume est le royaume préféré indiqué dans le fichier de configuration Kerberos accessible au client.

Le client aura aussi un nom de principal pour sa propre identité (et il doit avoir un ticket valide pour ce principal). Pour utiliser GSSAPI pour l'authentification, le principal du client doit être associé avec un nom d'utilisateur de base PostgreSQL. Le fichier de configuration `pg_ident.conf` peut être utilisé pour établir une correspondance entre les principaux et les noms d'utilisateurs. Par exemple, `pgusername@realm` pourrait correspondre à `pgusername`. De la même façon, vous pouvez

[4] https://datatracker.ietf.org/doc/html/rfc2743

utiliser le principal complet `username@realm` comme nom de rôle dans PostgreSQL sans aucune correspondance.

PostgreSQL supporte aussi la correspondance entre principaux de clients et noms d'utilisateur en supprimant le royaume du principal. Cette méthode est supportée pour des raisons de compatibilité ascendante et est fortement déconseillé car il est ensuite impossible de distinguer différents utilisateurs avec le même nom d'utilisateur mais un domaine différent. Pour l'activer, configurez `include_realm` à 0. Pour des installations simples à un seul royaume, le faire en combinant avec la configuration du paramètre `krb_realm` (qui vérifie que le royaume du principal correspond exactement à ce qui est dans le paramètre `krb_realm`) est toujours sécurisé mais cette approche offre moins de possibilités en comparaison à la spécification d'une correspondance explicite.

L'emplacement du fichier keytab du serveur est indiqué grâce au paramètre de configuration krb_server_keyfile. Pour des raisons de sécurité, il est recommandé d'utiliser un fichier de clés séparé pour PostgreSQL uniquement plutôt que d'autoriser le serveur à lire le fichier keytab du système. Assurez-vous que le fichier keytab du serveur est lisible (et de préférence seulement lisible, pas modifiable) par le compte serveur PostgreSQL (voir aussi Section 19.1).

Le fichier de clés est généré en utilisant le logiciel Kerberos ; voir la documentation de Kerberos pour plus de détails. L'exemple suivant montre comment le faire en utilisant l'outil kadmin des implémentations compatibles MIT de Kerberos 5 :

```
kadmin% addprinc -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

Les options d'authentification suivantes sont supportées pour la méthode d'authentification GSSAPI :

`include_realm`

Si configuré à 0, le nom du royaume provenant du principal de l'utilisateur authentifié est supprimé avant d'être passé à la correspondance du nom d'utilisateur (Section 21.2). Ceci n'est pas conseillé mais reste disponible principalement pour des raisons de compatibilité ascendante car ce n'est pas sûr dans des environnements avec plusieurs royaumes sauf si `krb_realm` est aussi utilisé. Il est recommandé de laisser `include_realm` configurer à sa valeur par défaut et de fournir une correspondance explicite dans `pg_ident.conf` pour convertir les noms de principaux en noms d'utilisateurs PostgreSQL.

`compat_realm`

Si configuré à 1, le nom, compatible SAM, du domaine (aussi connu en tant que nom NetBIOS) est utilisé pour l'option `include_realm`. C'est la valeur par défaut. Si configuré à 0, le vrai nom de royaume provenant de nom du principal Kerberos est utilisé.

Ne désactivez pas cette option sauf si votre serveur est exécuté sous un compte domaine (ceci inclut les comptes de service virtuels sur un système membre du domaine) et si tous les clients s'authentifiant via SSPI utilisent aussi des comptes domaines. Dans le cas contraire, l'authentification va échouer.

`upn_username`

Si cette option est activée avec `compat_realm`, le nom de l'utilisateur provenant du UPN Kerberos est utilisé pour l'authentification. Si elle est désactivée (par défaut), le nom d'utilisateur provenant de l'UPN Kerberos est utilisé pour l'authentification. Si elle est désactivée (par défaut), le nom d'utilisateur compatible SAM est utilisé. Par défaut, ces deux noms sont identiques pour les nouveaux comptes utilisateurs.

Notez que libpq utilise le nom compatible SAM si aucun nom d'utilisateur explicite n'est spécifié. Si vous utilisez la libpq ou un connecteur basé sur cette bibliothèque, vous devriez laisser cette option désactivée ou indiquer explicitement le nom d'utilisateur dans la chaîne de connexion.

`map`

Permet la mise en correspondance entre les principaux clients et les noms d'utilisateurs de base. Voir Section 21.2 pour plus de détails. Pour un principal GSSAPI/Kerberos, tel que `username@EXAMPLE.COM` (ou, moins communément, `username/hostbased@EXAMPLE.COM`), le nom d'utilisateur utilisé pour la correspondance est `username@EXAMPLE.COM` (ou `username/hostbased@EXAMPLE.COM`, respectivement), sauf si `include_realm` a été configuré à 0, auquel cas `username` (ou `username/hostbased`) est ce qui est vu comme le nom d'utilisateur du système lors de la recherche de correspondance.

`krb_realm`

Configure le domaine pour la correspondance du principal de l'utilisateur. Si ce paramètre est configuré, seuls les utilisateurs de ce domaine seront acceptés. S'il n'est pas configuré, les utilisateurs de tout domaine peuvent se connecter, à condition que la correspondance du nom de l'utilisateur est faite.

En plus de ces paramètres, qui peuvent être différents pour différentes entrées `pg_hba.conf`, il existe le paramètre serveur de configuration krb_caseins_users. S'il est configuré à true, les principaux clients sont comparés aux entrées de la carte utilisateurs, sans attention à la casse. Si `krb_realm` est configuré, il est aussi comparé sans attention à la casse.

21.7. Authentification SSPI

SSPI est une technologie Windows pour l'authentification sécurisée avec *single sign-on*. PostgreSQL utilise SSPI dans un mode de négociation (`negotiate`) qui utilise Kerberos si possible et NTLM sinon. L'authentification SSPI et GSSAPI interopèrent comme clients et serveurs, par exemple un client SSPI peut s'authentifier avec un serveur GSSAPI. Il est recommandé d'utiliser SSPI sur les clients et serveurs Windows et GSSAPI sur les autres plateformes.

Lorsque Kerberos est utilisé, SSPI fonctionne de la même façon que GSSAPI. Voir Section 21.6 pour les détails.

Les options de configuration suivantes sont supportées pour SSPI :

`include_realm`

Si configuré à 0, le nom du royaume provenant du principal de l'utilisateur authentifié est supprimé avant d'être passé à la correspondance du nom d'utilisateur (Section 21.2). Ceci n'est pas conseillé mais reste disponible principalement pour des raisons de compatibilité ascendante car ce n'est pas sûr dans des environnements avec plusieurs royaumes sauf si `krb_realm` est aussi utilisé. Il est recommandé aux utilisateurs de laisser include_realm configuré à sa valeur par défaut (1) et de fournir une correspondance explicite dans `pg_ident.conf`.

`map`

Permet la mise en correspondance entre les noms système et base de données. Voir Section 21.2 pour plus de détails. Pour un principal GSSAPI/Kerberos, tel que `username@EXAMPLE.COM` (ou, moins communément, `username/hostbased@EXAMPLE.COM`), le nom d'utilisateur utilisé pour la correspondance est `username@EXAMPLE.COM` (ou `username/hostbased@EXAMPLE.COM`, respectivement), sauf si `include_realm` a été configuré à 0, auquel cas `username` (ou `username/hostbased`) est ce qui est vu comme le nom d'utilisateur du système lors de la recherche de correspondance.

`krb_realm`

Configure le domaine pour la correspondance du principal de l'utilisateur. Si ce paramètre est configuré, seuls les utilisateurs de ce domaine seront acceptés. S'il n'est pas configuré, les

utilisateurs de tout domaine peuvent se connecter, à condition que la correspondance du nom de l'utilisateur est faite.

21.8. Authentification fondée sur ident

La méthode d'authentification ident fonctionne en obtenant le nom de l'opérateur du système depuis le serveur ident distant et en l'appliquant comme nom de l'utilisateur de la base de données (et après une éventuelle mise en correspondance). Cette méthode n'est supportée que pour les connexions TCP/IP.

> **Note**
>
> Lorsqu'ident est spécifié pour une connexion locale (c'est-à-dire non TCP/IP), l'authentification peer (voir Section 21.9) lui est automatiquement substituée.

Les options de configuration suivantes sont supportées pour `ident` :

`map`

Permet la mise en correspondance entre les noms système et base de données. Voir Section 21.2 pour plus de détails.

Le « protocole d'identification » est décrit dans la RFC 1413[5]. Théoriquement, chaque système d'exploitation de type Unix contient un serveur ident qui écoute par défaut sur le port TCP 113. La fonctionnalité basique d'un serveur ident est de répondre aux questions telles que : « Quel utilisateur a initié la connexion qui sort du port *X* et se connecte à mon port *Y*? ». Puisque PostgreSQL connaît *X* et *Y* dès lors qu'une connexion physique est établie, il peut interroger le serveur ident de l'hôte du client qui se connecte et peut ainsi théoriquement déterminer l'utilisateur du système d'exploitation pour n'importe quelle connexion.

Le revers de cette procédure est qu'elle dépend de l'intégrité du client : si la machine cliente est douteuse ou compromise, un attaquant peut lancer n'importe quel programme sur le port 113 et retourner un nom d'utilisateur de son choix. Cette méthode d'authentification n'est, par conséquent, appropriée que dans le cas de réseaux fermés dans lesquels chaque machine cliente est soumise à un contrôle strict et dans lesquels les administrateurs système et de bases de données opèrent en étroite collaboration. En d'autres mots, il faut pouvoir faire confiance à la machine hébergeant le serveur d'identification. Cet avertissement doit être gardé à l'esprit :

> Le protocole d'identification n'a pas vocation à être un protocole d'autorisation ou de contrôle d'accès.
>
> —RFC 1413

Certains serveurs ident ont une option non standard qui chiffre le nom de l'utilisateur retourné à l'aide d'une clé connue du seul administrateur de la machine dont émane la connexion. Cette option *ne doit pas* être employée lorsque le serveur ident est utilisé avec PostgreSQL car PostgreSQL n'a aucun moyen de déchiffrer la chaîne renvoyée pour déterminer le nom réel de l'utilisateur.

21.9. Authentification Peer

La méthode d'authentification peer utilise les services du système d'exploitation afin d'obtenir le nom de l'opérateur ayant lancé la commande client de connexion et l'utilise (après une éventuelle mise en correspondance) comme nom d'utilisateur de la base de données. Cette méthode n'est supportée que pour les connexions locales.

Les options de configuration suivantes sont supportées pour l'authentification `peer` :

[5] https://datatracker.ietf.org/doc/html/rfc1413

`map`

Autorise la mise en correspondance entre le nom d'utilisateur fourni par le système d'exploitation et le nom d'utilisateur pour la base de données. Voir Section 21.2 pour plus de détails.

L'authentification peer n'est disponible que sur les systèmes d'exploitation fournissant la fonction `getpeereid()`, le paramètre `SO_PEERCRED` pour les sockets ou un mécanisme similaire. Actuellement, cela inclut Linux, la plupart des variantes BSD (et donc macOS), ainsi que Solaris.

21.10. Authentification LDAP

Ce mécanisme d'authentification opère de façon similaire à `password` à ceci près qu'il utilise LDAP comme méthode de vérification des mots de passe. LDAP n'est utilisé que pour valider les paires nom d'utilisateur/mot de passe. De ce fait, pour pouvoir utiliser LDAP comme méthode d'authentification, l'utilisateur doit préalablement exister dans la base.

L'authentification LDAP peut opérer en deux modes. Dans le premier mode, que nous appelerons le mode « simple bind », le serveur fera un « bind » sur le nom distingué comme *préfixe* *nom_utilisateur* *suffixe*. Typiquement, le paramètre *prefix* est utilisé pour spécifier `cn=` ou *DOMAIN*\ dans un environnement Active Directory. *suffix* est utilisé pour spécifier le reste du DN dans un environnement autre qu'Active Directory.

Dans le second mode, que nous appelerons mode « search+bind », le serveur commence un « bind » sur le répertoire LDAP avec un nom d'utilisateur et un mot de passe fixés, qu'il indique à *ldapbinddn* et *ldapbindpasswd*. Il réalise une recherche de l'utilisateur en essayant de se connecter à la base de données. Si aucun utilisateur et aucun mot de passe n'est configuré, un « bind » anonyme sera tenté sur le répertoire. La recherche sera réalisée sur le sous-arbre sur *ldapbasedn*, et essaiera une correspondance exacte de l'attribut indiqué par *ldapsearchattribute*. Une fois que l'utilisateur a été trouvé lors de cette recherche, le serveur se déconnecte et effectue un nouveau « bind » au répertoire en tant que cet utilisateur, en utilisant le mot de passe indiqué par le client pour vérifier que la chaîne de connexion est correcte. Ce mode est identique à celui utilisé par les schémas d'authentification LDAP dans les autres logiciels, tels que les modules Apache `mod_authnz_ldap` et `pam_ldap`. Cette méthode permet une plus grande flexibilité sur l'emplacement des objets utilisateurs dans le répertoire mais demandera deux connexions au serveur LDAP.

Les options de configuration suivantes sont utilisées dans les deux modes :

`ldapserver`

Noms ou adresses IP des serveurs LDAP auxquels se connecter. Plusieurs serveurs peuvent être indiqués, en les séparant par des espaces.

`ldapport`

Numéro de port du serveur LDAP auquel se connecter. Si aucun port n'est spécifié, le port par défaut de la bibliothèque LDAP sera utilisé.

`ldapscheme`

Positionner à `ldaps` pour utiliser LDAPS. Il s'agit d'une utilisation non standard de LDAP sur SSL, supportée par certaines implémentations de serveurs LDAP. Voir aussi l'option `ldaptls` pour une méthode alternative.

`ldaptls`

Positionnez à 1 pour que la connexion entre PostgreSQL et le serveur LDAP utilise du chiffrage TLS. Ceci utilise l'opération `StartTLS` d'après la RFC 4513[6]. Voir aussi l'option `ldapscheme` pour une alternative.

[6] https://datatracker.ietf.org/doc/html/rfc4513

Veuillez noter que l'utilisation de `ldapscheme` ou de `ldaptls` ne chiffre que le trafic entre le serveur PostgreSQL et le serveur LDAP. La connexion entre le serveur PostgreSQL et le client PostgreSQL ne sera pas pour autant chiffée, à moins que SSL ne soit également utilisé pour la connexion.

Les options suivantes sont utilisées uniquement dans le mode « simple bind » :

`ldapprefix`

Chaîne à préfixer au nom de l'utilisateur pour former le DN utilisé comme lien lors d'une simple authentification *bind*.

`ldapsuffix`

Chaîne à suffixer au nom de l'utilisateur pour former le DN utilisé comme lien lors d'une simple authentification *bind*.

Les options suivantes sont utilisées uniquement dans le mode « search+bind » :

`ldapbasedn`

Racine DN pour commencer la recherche de l'utilisateur lors d'une authentification *search+bind*.

`ldapbinddn`

DN de l'utilisateur pour se lier au répertoire avec lequel effectuer la recherche lors d'une authentification *search+bind*.

`ldapbindpasswd`

Mot de passe de l'utilisateur pour se lier au répertoire avec lequel effectuer la recherche lors d'une authentification *search+bind*.

`ldapsearchattribute`

Attribut à faire correspondre au nom d'utilisateur dans la recherche lors d'une authentification *search+bind*. Si aucun attribut n'est indiqué, l'attribut `uid` sera utilisé.

`ldapsearchfilter`

Le filtre de recherche à utiliser lors d'une authentification search+bind. Toutes les occurrences de `$username` seront remplacées par le nom d'utilisateur. Cela permet des filtres de recherche plus flexibles que `ldapsearchattribute`.

`ldapurl`

Une URL LDAP dont le format est spécifié par la RFC 4516[7]. C'est une autre façon d'écrire certaines options LDAP d'une façon plus compacte et standard. Le format est :

`ldap[s]://`*`hote`*`[:`*`port`*`]/`*`basedn`*`[?[`*`attribut`*`][?[`*`scope`*`][?[`*`filtre`*`]]]]`

`scope` doit faire partie des possibilités suivantes : `base`, `one`, `sub`. Ce sera généralement la dernière possibilité. (La valeur par défaut est `base`, qui n'est généralement pas utile dans ce cadre là.) *`attribute`* peut désigner un unique attribut, auquel cas il sera utilisé comme valeur pour `ldapsearchattribute`. Si *`attribute`* est vide, alors *`filter`* peut être utilisé comme valeur pour `ldapsearchfilter`.

Le schéma d'URL `ldaps` choisit la méthode LDAPS pour établir une connexion LDAP sur SSL, qui est équivalent à l'utilisation de `ldapscheme=ldaps`. Pour utiliser une connexion LDAP

[7] https://datatracker.ietf.org/doc/html/rfc4516

chiffrée en utilisant l'opération `StartTLS`, utilisez le schéma d'URL normal `ldap` et spécifiez l'option `ldaptls` en plus de `ldapurl`.

Pour les « bind » non anonymes, `ldapbinddn` et `ldapbindpasswd` doivent être spécifiées comme des options séparées.

Les URL LDAP sont actuellement seulement supportées par OpenLDAP, et pas sous Windows.

Mixer les options de configurations du mode « simple bind » et du mode « search+bind » est une erreur.

Lorsque vous utilisez le mode search+bind mode, la recherche peut être effectuée en utilisant l'attribut unique spécifié avec `ldapsearchattribute`, ou en utilisant un filtre de recherche personnalisé avec `ldapsearchfilter`. Spécifier `ldapsearchattribute=foo` est équivalent à spécifier `ldapsearchfilter="(foo=$username)"`. Si aucune de ces options n'est spécifiée, le comportement par défaut est `ldapsearchattribute=uid`.

Si PostgreSQL a été compilé avec OpenLDAP comme bibliothèque LDAP cliente, l'option de configuration `ldapserver` peut être omise. Dans ce cas, une liste de noms d'hôtes et ports sera recherchée via les enregistrements RFC 2782[8] DNS SRV. Le nom `_ldap._tcp.DOMAIN` est recherché, où `DOMAIN` est extrait de `ldapbasedn`.

Voici un exemple de configuration LDAP pour le mode « simple bind » :

```
host ... ldap ldapserver=ldap.example.net ldapprefix="cn="
 ldapsuffix=", dc=example, dc=net"
```

Quand une connexion au serveur de base de données est demandée en tant que `un_utilisateur`, PostgreSQL tentera un « bind » vers le serveur LDAP en utilisant le DN `cn=un_utilisateur, dc=example, dc=net` et le mot de passe fourni par le client. Si cette connexion réussit, l'accès à la base de données est accepté.

Voici un exemple de configuration LDAP pour le mode « search+bind » :

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example,
 dc=net" ldapsearchattribute=uid
```

Quand une connexion au serveur de base de données est tentée en tant que `un_utilisateur`, PostgreSQL tentera un « bind » anonyme (car `ldapbinddn` n'a pas été précisé) au serveur LDAP, effectuera une recherche pour `(uid=un_utilisateur)` sous le DN de base spécifié. Si une entrée est trouvée, il tentera alors de faire un « bind » en utilisant l'information trouvée et le mot de passe fourni par le client. Si cette deuxième connexion réussit, l'accès à la base est autorisé.

Voici la même configuration « search+bind » écrite sous la forme d'une URL :

```
host ... ldap ldapurl="ldap://ldap.example.net/dc=example,dc=net?
uid?sub"
```

D'autres logiciels qui supportent l'authentification LDAP utilisent le même format d'URL donc cela facilitera le partage de configuration.

Voici un exemple de configuration search+bind configuration qui utilise `ldapsearchfilter` plutôt que `ldapsearchattribute` pour permettre l'authentification par nom d'utilisateur ou adresse mail :

[8] https://datatracker.ietf.org/doc/html/rfc2782

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example,
 dc=net" ldapsearchfilter="(|(uid=$username)(mail=$username))"
```

Voici un exemple de configuration search+bind qui utilise la découverte DNS SRV pour trouver le(s) nom(s) d'hôte et port(s) pour le service LDAP pour le nom de domaine `example.net` :

```
host ... ldap ldapbasedn="dc=example,dc=net"
```

> **Astuce**
>
> Comme LDAP utilise souvent des virgules et des espaces pour séparer les différentes parties d'un DN, il est souvent nécessaire d'utiliser des paramètres entourés de guillemets durant le paramétrage des options LDAP, comme montré dans les exemples.

21.11. Authentification RADIUS

Cette méthode d'authentification opère de façon similaire à `password` sauf qu'il existe la méthode RADIUS pour la vérification du mot de passe. RADIUS est seulement utilisé pour valider des pairs nom utilisateur / mot de passe. Du coup, l'utilisateur doit déjà exister dans la base de données avant que RADIUS puisse être utilisé pour l'authentification.

Lors de l'utilisation de l'authentification RADIUS, un message de demande d'accès (*Access Request*) sera envoyé au serveur RADIUS configuré. Cette demande sera du type « authentification seule » (`Authenticate Only`) et incluera les paramètres pour le nom de l'utilisateur, son mot de passe (chiffré) et un identifiant NAS (`NAS Identifier`). La demande sera chiffrée en utilisant un secret partagé avec le serveur. Le serveur RADIUS répond à cette demande avoir soit `Access Accept` soit `Access Reject`. Il n'y a pas de support des comptes RADIUS.

Plusieurs serveurs RADIUS peuvent être spécifiés, auquel cas ils seront testés de manière séquentielle. Si une réponse négative est reçue d'un serveur, l'authentification échouera. Si aucune réponse n'est reçue, le serveur qui suit dans la liste sera essayé. Pour spécifier plusieurs serveurs, séparez les noms de serveurs par des virgules et placez la liste entre guillemets double. Si plusieurs serveurs sont spécifiés, toutes les autres options RADIUS peuvent également être indiquées en tant que liste séparée par des virgules, pour s'appliquer individuellement à chaque serveur. Elles peuvent également être spécifiées indépendemment. Dans ce cas, elles seront appliquées à tous les serveurs.

Les options de configuration suivantes sont supportées par RADIUS :

`radiusservers`

Les noms DNS et/ou adresses IP des serveurs RADIUS pour l'authentification. Ce paramètre est requis.

`radiussecrets`

Les secrets partagés utilisés lors de discussions sécurisées avec les serveurs RADIUS. Il doit y avoir exactement la même valeur sur le serveur PostgreSQL et sur le serveur RADIUS. Il est recommandé d'utiliser une chaîne d'au moins 16 caractères. Ce paramètre est requis.

> **Note**
>
> Le vecteur de chiffrement utilisé sera un chiffrement fort seulement si PostgreSQL a été compilé avec le support d'OpenSSL. Dans les autres cas, la transmission au serveur

RADIUS peut seulement être considérée comme caché, et non pas sécurisé, et des mesures de sécurité externes doivent être appliquées si nécessaire.

`radiusports`

Les numéros de port sur les serveurs RADIUS pour la connexion. Si aucun port n'est indiqué, le port RADIUS par défaut (`1812`) sera utilisé.

`radiusidentifiers`

Les chaînes à utiliser comme identifiants NAS (`NAS Identifier`) dans les demandes RADIUS. Ce paramètre peut être utilisé, par exemple, pour identifier l'instance auquel l'utilisateur tente de se connecter. C'est utilisable pour des vérifications sur les serveurs RADIUS. Si aucune identifiant n'est spécifié, la valeur par défaut, `postgresql`, sera utilisée.

S'il est nécessaire d'avoir une virgule ou un espace blanc dans la valeur d'un paramètre RADIUS, ceci peut se faire en entourant la valeur de guillemets doubles, mais ceci peut devenir compliqué car deux niveaux de guillemets doubles sont maintenant nécessaires. Voici un exemple d'espace bland dans les chaînes de secret RADIUS :

```
host ... radius radiusservers="server1,server2"
 radiussecrets="""secret one"",""secret two"""
```

21.12. Authentification de certificat

Cette méthode d'authentification utilise des clients SSL pour procéder à l'authentification. Elle n'est par conséquent disponible que pour les connexions SSL ; voir Section 19.9.2 pour les instructions de configuration SSL. Quand cette méthode est utilisée, le serveur exigera que le client fournisse un certificat valide et de confiance. Aucune invite de saisie de mot de passe ne sera envoyée au client. L'attribut `cn` (Common Name) du certificat sera comparé au nom d'utilisateur de base de données demandé. S'ils correspondent, la connexion sera autorisée. La correspondance des noms d'utilisateurs peut être utilisé pour permettre au `cn` d'être différent du nom d'utilisateur de la base de données.

Les options de configuration suivantes sont supportées pour l'authentification par certificat SSL :

`map`

Permet la correspondance entre les noms d'utilisateur système et les noms d'utilisateurs de bases de données. Voir Section 21.2 pour les détails.

Il est redondant d'utiliser l'option `clientcert` avec l'authentification `cert` parce que l'authentification `cert` est réellement l'authentification `trust` avec `clientcert=verify-full`.

21.13. Authentification PAM

Ce mécanisme d'authentification fonctionne de façon similaire à `password` à ceci près qu'il utilise PAM (Pluggable Authentication Modules) comme méthode d'authentification. Le nom du service PAM par défaut est `postgresql`. PAM n'est utilisé que pour valider des paires nom utilisateur/mot de passe et en option le nom de l'hôte distant connecté ou de l'adresse IP. De ce fait, avant de pouvoir utiliser PAM pour l'authentification, l'utilisateur doit préalablement exister dans la base de données. Pour plus d'informations sur PAM, merci de lire la page Linux-PAM[9].

Les options suivantes sont supportées pour PAM :

[9] https://www.kernel.org/pub/linux/libs/pam/

`pamservice`

Nom de service PAM.

`pam_use_hostname`

Détermine si l'adresse IP ou le nom d'hôte distant est fourni aux modules PAM via l'élément `PAM_RHOST`. Par défaut, l'adresse IP est utilisé. Configurez cette option à 1 pour utiliser à la place le nom d'hôte résolu. La résolution de nom d'hôte peut amener des délais de connexion. (La plupart des configurations PAM n'utilise pas cette information, donc il est seulement nécessaire de considérer ce paramètre si une configuration PAM a été créée spécifiquement pour l'utiliser.)

> **Note**
>
> Si PAM est configuré pour lire `/etc/shadow`, l'authentification échoue car le serveur PostgreSQL est exécuté en tant qu'utilisateur standard. Ce n'est toutefois pas un problème quand PAM est configuré pour utiliser LDAP ou les autres méthodes d'authentification.

21.14. Authentification BSD

Cette méthode d'authentification opère de façon similaire à `password` sauf qu'elle utilise l'authentification BSD pour vérifier le mot de passe. L'authentification BSD est seulement utilisée pour valider la paire nom d'utilisateur/mot de passe. De ce fait, le rôle de l'utilisation doit déjà exister dans la base de données avant que l'authentification BSD puisse être utilisée pour l'authentification. Cette méthode est actuellement uniquement disponible sur OpenBSD.

L'authentification BSD dans PostgreSQL utilise le type de login `auth-postgresql` et s'authentifie avec la classe de login `postgresql` si c'est défini dans `login.conf`. Par défaut, cette classe de login n'existe pas, et PostgreSQL utilisera la classe de login par défaut.

> **Note**
>
> Pour utiliser l'authentification BSD, le compte utilisateur PostgreSQL (c'est-à-dire l'utilisateur système qui exécute le serveur) doit d'abord être ajouté dans le groupe `auth`. Le groupe `auth` existe par défaut sur les systèmes OpenBSD.

21.15. Problèmes d'authentification

Les erreurs et problèmes d'authentification se manifestent généralement par des messages d'erreurs tels que ceux qui suivent.

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user
 "andym", database "testdb"
```

ou, en français,

```
FATAL:  pas d'entrée pg_hba.conf pour l'hôte "123.123.123.123",
 utilisateur "andym", base "testdb"
```

C'est le message le plus probable lorsque le contact peut être établi avec le serveur mais qu'il refuse de communiquer. Comme le suggère le message, le serveur a refusé la demande de connexion parce qu'il n'a trouvé aucune entrée correspondante dans son fichier de configuration `pg_hba.conf`.

```
FATAL:  password authentication failed for user "andym"
```

ou, en français,

```
FATAL:  l'authentification par mot de passe a échoué pour
 l'utilisateur "andym"
```

Les messages de ce type indiquent que le serveur a été contacté et qu'il accepte la communication, mais pas avant que la méthode d'authentification indiquée dans le fichier `pg_hba.conf` n'ait été franchie avec succès. Le mot de passe fourni, le logiciel d'identification ou le logiciel Kerberos doivent être vérifiés en fonction du type d'authentification mentionné dans le message d'erreur.

```
FATAL:  user "andym" does not exist
```

ou, en français,

```
FATAL:  l'utilisateur "andym" n'existe pas
```

Le nom d'utilisateur indiqué n'a pas été trouvé.

```
FATAL:  database "testdb" does not exist
```

ou, en français,

```
FATAL:  la base "testdb" n'existe pas
```

La base de données utilisée pour la tentative de connexion n'existe pas. Si aucune base n'est précisée, le nom de la base par défaut est le nom de l'utilisateur, ce qui peut être approprié ou non.

> **Astuce**
>
> Les traces du serveur contiennent plus d'informations sur une erreur d'authentification que ce qui est rapporté au client. En cas de doute sur les raisons d'un échec, il peut s'avérer utile de les consulter.

Chapitre 22. Rôles de la base de données

PostgreSQL gère les droits d'accès aux bases de données en utilisant le concept de *rôles*. Un rôle peut être vu soit comme un utilisateur de la base de données, soit comme un groupe d'utilisateurs de la base de données, suivant la façon dont le rôle est configuré. Les rôles peuvent posséder des objets de la base de données (par exemple des tables et des fonctions) et peuvent affecter des droits sur ces objets à d'autres rôles pour contrôler qui a accès à ces objets. De plus, il est possible de donner l'*appartenance* d'un rôle à un autre rôle, l'autorisant du coup à utiliser les droits affectés à un autre rôle.

Le concept des rôles comprend les concepts des « utilisateurs » et des « groupes ». Dans les versions de PostgreSQL antérieures à la 8.1, les utilisateurs et les groupes étaient des types d'entité distincts mais, maintenant, ce ne sont que des rôles. Tout rôle peut agir comme un utilisateur, un groupe ou les deux.

Ce chapitre décrit comment créer et gérer des rôles. Section 5.7 donne plus d'informations sur les effets des droits des rôles pour les différents objets de la base de données.

22.1. Rôles de la base de données

Conceptuellement, les rôles de la base sont totalement séparés des utilisateurs du système d'exploitation. En pratique, il peut être commode de maintenir une correspondance mais cela n'est pas requis. Les rôles sont globaux à toute une installation de groupe de bases de données (et non individuelle pour chaque base). Pour créer un rôle, utilisez la commande SQL `CREATE ROLE` :

```
CREATE ROLE nom_utilisateur;
```

`nom_utilisateur` suit les règles des identifiants SQL : soit sans guillemets et sans caractères spéciaux, soit entre double guillemets (en pratique, vous voudrez surtout ajouter des options supplémentaires, comme `LOGIN`, à cette commande. Vous trouverez plus de détails ci-dessous). Pour supprimer un rôle existant, utilisez la commande analogue `DROP ROLE` :

```
DROP ROLE nom_utilisateur;
```

Pour une certaine facilité d'utilisation, les programmes createuser et dropuser sont fournis comme emballage de ces commandes SQL et peuvent être appelés depuis la ligne de commande du shell :

```
createuser nom_utilisateur
dropuser nom_utilisateur
```

Pour déterminer l'ensemble des rôles existants, examinez le catalogue système `pg_roles` existant, par exemple

```
SELECT rolname FROM pg_roles;
```

La méta-commande `\du` du programme psql est aussi utile pour lister les rôles existants.

Afin d'amorcer le système de base de données, un système récemment installé contient toujours un rôle prédéfini. Ce rôle est un superutilisateur et aura par défaut le même nom que l'utilisateur du système d'exploitation qui a initialisé le groupe de bases de données (à moins que cela ne soit modifié en lançant la commande `initdb`). Par habitude, ce rôle sera nommé `postgres`. Pour créer plus de rôles, vous devez d'abord vous connecter en tant que ce rôle initial.

Chaque connexion au serveur de la base de données est faite au nom d'un certain rôle et ce rôle détermine les droits d'accès initiaux pour les commandes lancées sur cette connexion. Le nom du rôle à employer pour une connexion à une base particulière est indiqué par le client initialisant la demande de connexion, et ce, de la manière qui lui est propre. Par exemple, le programme `psql` utilise l'option de

ligne de commandes `-U` pour préciser sous quel rôle il se connecte. Beaucoup d'applications (incluant `createuser` et `psql`) utilisent par défaut le nom courant de l'utilisateur du système d'exploitation. Par conséquence, il peut souvent être pratique de maintenir une correspondance de nommage entre les rôles et les utilisateurs du système d'exploitation.

La configuration de l'authentification du client détermine avec quel rôle de la base, la connexion cliente donnée se connectera, comme cela est expliqué dans le Chapitre 21 (donc, un client n'est pas obligé de se connecter avec le rôle du même nom que son nom d'utilisateur dans le système d'exploitation ; de la même façon que le nom de connexion d'un utilisateur peut ne pas correspondre à son vrai nom). Comme le rôle détermine l'ensemble des droits disponibles pour le client connecté, il est important de configurer soigneusement les droits quand un environnement multi-utilisateurs est mis en place.

22.2. Attributs des rôles

Un rôle de bases de données peut avoir un certain nombre d'attributs qui définissent ses droits et interagissent avec le système d'authentification du client.

droit de connexion

Seuls les rôles disposant de l'attribut `LOGIN` peuvent être utilisés comme nom de rôle initial pour une connexion à une base de données. Un rôle avec l'attribut `LOGIN` peut être considéré de la même façon qu'un « utilisateur de la base de données ». Pour créer un rôle disposant du droit de connexion, utilisez :

```
CREATE ROLE nom LOGIN;
CREATE USER nom;
```

(`CREATE USER` est équivalent à `CREATE ROLE` sauf que `CREATE USER` utilise `LOGIN` par défaut alors que `CREATE ROLE` ne le fait pas)

statut de superutilisateur

Les superutilisateurs ne sont pas pris en compte dans les vérifications des droits, sauf le droit de connexion ou d'initier la réplication. Ceci est un droit dangereux et ne devrait pas être utilisé sans faire particulièrement attention ; il est préférable de faire la grande majorité de votre travail avec un rôle qui n'est pas superutilisateur. Pour créer un nouveau superutilisateur, utilisez `CREATE ROLE nom SUPERUSER`. Vous devez le faire en tant que superutilisateur.

création de bases de données

Les droits de création de bases doivent être explicitement données à un rôle (à l'exception des superutilisateurs qui passent au travers de toute vérification de droits). Pour créer un tel rôle, utilisez `CREATE ROLE nom_utilisateur CREATEDB`.

création de rôle

Un rôle doit se voir explicitement donné le droit de créer plus de rôles (sauf pour les superutilisateurs vu qu'ils ne sont pas pris en compte lors des vérifications de droits). Pour créer un tel rôle, utilisez `CREATE ROLE nom CREATEROLE`. Un rôle disposant de l'attribut `CREATEROLE` peut aussi modifier et supprimer d'autres rôles, ainsi que donner ou supprimer l'appartenance à ces rôles. Modifier un rôle inclut beaucoup de changements possibles en utilisant la commande `ALTER ROLE`, ceci incluant, par exemple, le changement de mot de passe. Cela inclut aussi les modifications à un rôle qui peuvent se faire en utilisant les commandes `COMMENT` et `SECURITY LABEL`.

Néanmoins, `CREATEROLE` ne donne pas la possibilité de créer des rôles `SUPERUSER`, pas plus qu'il ne donne de pouvoirs sur les rôles existants disposant de l'attribut `SUPERUSER`. De plus, `CREATEROLE` n'offre pas la possibilité de créer des utilisateurs disposant de l'attribut `REPLICATION`, pas plus que la capacité de donner ou supprimer l'attribut `REPLICATION` ou que la possibilité de modifier les propriétés de ce genre d'utilisateurs. Néanmoins, il permet

l'utilisation des commandes `ALTER ROLE ... SET` et `ALTER ROLE ... RENAME` sur des rôles disposant de l'attribut `REPLICATION`, ainsi que l'utilisation des commandes `COMMENT ON ROLE`, `SECURITY LABEL ON ROLE` et `DROP ROLE`. Enfin, `CREATEROLE` n'offre pas la possibilité de donner ou supprimer l'attribut `BYPASSRLS`.

Comme l'attribut `CREATEROLE` autorise un utilisateur à donner ou supprimer des rôles même pour des rôles auxquels il n'a pas (encore) accès, un utilisateur disposant de `CREATEROLE` peut obtenir l'accès aux capacités de chaque rôle prédéfini dans le système, incluant les rôles hautement privilégiés tels que `pg_execute_server_program` et `pg_write_server_files`.

initier une réplication

Un rôle doit se voir explicitement donné le droit d'initier une réplication en flux (sauf pour les superutilisateurs, puisqu'ils ne sont pas soumis aux vérifications de permissions). Un rôle utilisé pour la réplication en flux doit avoir le droit `LOGIN`. Pour créer un tel rôle, utilisez `CREATE ROLE` *`nom`* `REPLICATION LOGIN`.

mot de passe

Un mot de passe est seulement significatif si la méthode d'authentification du client exige que le client fournisse un mot de passe quand il se connecte à la base. Les méthodes d'authentification par `mot de passe` et `md5` utilisent des mots de passe. Les mots de passe de la base de données ne sont pas les mêmes que ceux du système d'exploitation. Indiquez un mot de passe lors de la création d'un rôle avec `CREATE ROLE` *`nom_utilisateur`* `PASSWORD '`*`le_mot_de_passe`*`'`.

héritage des droits

Un rôle a la possibilité d'hériter des droits de rôles dont il est membre, par défaut. Néanmoins, pour créer un rôle sans cette possibilité, utilisez `CREATE ROLE` *`nom`* `NOINHERIT`.

contourner la sécurité niveau lignes

Un rôle doit se voir donner explicitement le droit de contourner chaque politique de sécurité niveau lignes (RLS) (sauf pour les superutilisateurs car ils contournent toutes les vérifications de droits). Pour créer un tel rôle, utilisez `CREATE ROLE` *`nom`* `BYPASSRLS` en tant que superutilisateur.

limite de connexions

La limite de connexions peut indiquer combien de connexions concurrentes un rôle peut faire. -1 (la valeur par défaut) signifie qu'il n'y a pas de limite. Précisez une limite de connexion lors de la création d'un rôle avec `CREATE ROLE` *`nom`* `CONNECTION LIMIT '`*`integer`*`'`.

héritage des droits

Un rôle se voit donner par défaut le droit d'hériter des droits des rôles dont il est membre. Néanmoins, pour créer un rôle sans le droit, utilisez `CREATE ROLE` *`nom`* `NOINHERIT`.

contourner la sécurité niveau ligne

Un rôle doit se voir donner explicitement le droit de contourner chaque politique de sécurité niveau ligne (RLS), sauf pour les super-utilisateurs, car ces derniers contournent toutes les vérifications de droits). Pour créer un tel rôle, utilisez `CREATE ROLE` *`nom`* `BYPASSRLS` comme un super-utilisateur.

limite de connexion

La limite de connexions peut indiquer combien de connexions concurrentes un même rôle peut réaliser. La valeur par défaut, -1, signifie qu'il n'y a aucune limite. Indiquez une limite de connexion lors de la création d'un rôle avec `CREATE ROLE` *`nom`* `CONNECTION LIMIT '`*`integer`*`'`.

Les attributs d'un rôle peuvent être modifiés après sa création avec `ALTER ROLE`. Regardez les pages de références de CREATE ROLE et de ALTER ROLE pour plus de détails.

Un rôle peut aussi configurer ses options par défaut pour de nombreux paramètres de configuration décris dans le Chapitre 20. Par exemple, si, pour une raison ou une autre, vous voulez désactiver les parcours d'index (conseil : ce n'est pas une bonne idée) à chaque fois que vous vous connectez, vous pouvez utiliser :

```
ALTER ROLE myname SET enable_indexscan TO off;
```

Cela sauve les paramètres (mais ne les applique pas immédiatement). Dans les connexions ultérieures de ce rôle, c'est comme si `SET enable_indexscan TO off` avait été appelé juste avant le démarrage de la session. Vous pouvez toujours modifier les paramètres durant la session. Pour supprimer une configuration par défaut spécifique à un rôle, utilisez `ALTER ROLE` *`nom_utilisateur`* `RESET` *`nom_variable`*. Notez que les valeurs par défaut spécifiques aux rôles sans droit de connexion (`LOGIN`) sont vraiment inutiles car ils ne seront jamais appelés.

22.3. Appartenance d'un rôle

Il est souvent intéressant de grouper les utilisateurs pour faciliter la gestion des droits : de cette façon, les droits peuvent être donnés ou supprimés pour tout un groupe. Dans PostgreSQL, ceci se fait en créant un rôle représentant le groupe, puis en ajoutant les rôles utilisateurs individuels *membres* de ce groupe.

Pour configurer un rôle en tant que groupe, créez tout d'abord le rôle :

```
CREATE ROLE nom;
```

Typiquement, un rôle utilisé en tant que groupe n'aura pas l'attribut `LOGIN` bien que vous puissiez le faire si vous le souhaitez.

Une fois que ce rôle existe, vous pouvez lui ajouter et lui supprimer des membres en utilisant les commandes `GRANT` et `REVOKE` :

```
GRANT role_groupe TO role1, ... ;
REVOKE role_groupe FROM role1, ... ;
```

Vous pouvez aussi faire en sorte que d'autres rôles groupes appartiennent à ce groupe (car il n'y a pas réellement de distinction entre les rôles groupe et les rôles non groupe). La base de données ne vous laissera pas configurer des boucles circulaires d'appartenance. De plus, il est interdit de faire en sorte qu'un membre appartienne à `PUBLIC`.

Les membres d'un rôle groupe peuvent utiliser les droits du rôle de deux façons. Tout d'abord, chaque membre d'un groupe peut exécuter explicitement `SET ROLE` pour « devenir » temporairement le rôle groupe. Dans cet état, la session de la base de données a accès aux droits du rôle groupe plutôt qu'à ceux du rôle de connexion original et tous les objets créés sont considérés comme appartenant au rôle groupe, et non pas au rôle utilisé lors de la connexion. Deuxièmement, les rôles membres qui ont l'attribut `INHERIT` peuvent utiliser automatiquement les droits des rôles dont ils sont membres, ceci incluant les droits hérités par ces rôles. Comme exemple, supposons que nous avons lancé les commandes suivantes :

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

Immédiatement après connexion en tant que `joe`, la session de la base de données peut utiliser les droits donnés directement à `joe` ainsi que ceux donnés à `admin` parce que `joe` « hérite » des droits de `admin`. Néanmoins, les droits donnés à `wheel` ne sont pas disponibles parce que, même si `joe` est un

membre indirect de `wheel`, l'appartenance se fait via `admin` qui dispose de l'attribut `NOINHERIT`. Après :

```
SET ROLE admin;
```

la session aura la possibilité d'utiliser les droits donnés à `admin` mais n'aura plus accès à ceux de `joe`. Après :

```
SET ROLE wheel;
```

la session pourra utiliser uniquement ceux de `wheel`, mais ni ceux de `joe` ni ceux de `admin`. L'état du droit initial peut être restauré avec une des instructions suivantes :

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

> **Note**
>
> La commande `SET ROLE` autorisera toujours la sélection de tout rôle dont le rôle de connexion est membre directement ou indirectement. Du coup, dans l'exemple précédent, il n'est pas nécessaire de devenir `admin` pour devenir `wheel`.

> **Note**
>
> Dans le standard SQL, il existe une distinction claire entre les utilisateurs et les rôles. Les utilisateurs ne peuvent pas hériter automatiquement alors que les rôles le peuvent. Ce comportement est obtenu dans PostgreSQL en donnant aux rôles utilisés comme des rôles SQL l'attribut `INHERIT`, mais en donnant aux rôles utilisés en tant qu'utilisateurs SQL l'attribut `NOINHERIT`. Néanmoins, par défaut, PostgreSQL donne à tous les rôles l'attribut `INHERIT` pour des raisons de compatibilité avec les versions précédant la 8.1 dans lesquelles les utilisateurs avaient toujours les droits des groupes dont ils étaient membres.

Les attributs `LOGIN`, `SUPERUSER`, `CREATEDB` et `CREATEROLE` peuvent être vus comme des droits spéciaux qui ne sont jamais hérités contrairement aux droits ordinaires sur les objets de la base. Vous devez réellement utiliser `SET ROLE` vers un rôle spécifique pour avoir un de ces attributs et l'utiliser. Pour continuer avec l'exemple précédent, nous pourrions très bien choisir de donner les droits `CREATEDB` et `CREATEROLE` au rôle `admin`. Puis, une session connectée en tant que le rôle `joe` n'aurait pas ces droits immédiatement, seulement après avoir exécuté `SET ROLE admin`.

Pour détruire un rôle groupe, utilisez `DROP ROLE` :

```
DROP ROLE nom;
```

Toute appartenance à ce rôle est automatiquement supprimée (mais les rôles membres ne sont pas autrement affectés).

22.4. Supprimer des rôles

Comme les rôles peuvent posséder des objets dans une base de données et peuvent détenir des droits pour accéder à d'autres objets, supprimer un rôle n'est généralement pas la seule exécution d'un `DROP ROLE`. Tout objet appartenant à un rôle doit d'abord être supprimé ou réaffecté à d'autres propriétaires ; et tout droit donné à un rôle doit être révoqué.

L'appartenance des objets doit être transférée, un à la fois, en utilisant des commandes `ALTER`, par exemple :

```
ALTER TABLE table_de_bob OWNER TO alice;
```

Il est aussi possible d'utiliser la commande `REASSIGN OWNED` pour réaffecter tous les objets du rôle à supprimer à un autre rôle. Comme `REASSIGN OWNED` ne peut pas accéder aux objets dans les autres bases, il est nécessaire de l'exécuter dans chaque base qui contient des objets possédés par le rôle. (Notez que la première exécution de `REASSIGN OWNED` changera le propriétaire de tous les objets partagés entre bases de données, donc les bases et les tablespaces, qui appartiennent au rôle à supprimer.)

Une fois que tous les objets importants ont été transférés aux nouveaux propriétaires, tout objet restant possédé par le rôle à supprimer peut être supprimé avec la commande `DROP OWNED`. Encore une fois, cette commande ne peut pas accéder aux objets des autres bases de données, donc il est nécessaire de l'exécuter sur chaque base qui contient des objets dont le propriétaire correspond au rôle à supprimer. De plus, `DROP OWNED` ne supprimera pas des bases ou tablespaces entiers, donc il est nécessaire de le faire manuellement si le rôle possède des bases et/ou des tablespaces qui n'auraient pas été transférés à d'autres rôles.

`DROP OWNED` fait aussi attention à supprimer tout droit donné au rôle cible pour les objets qui ne lui appartiennent pas. Comme `REASSIGN OWNED` ne touche pas à ces objets, il est souvent nécessaire d'exécuter à la fois `REASSIGN OWNED` et `DROP OWNED` (dans cet ordre !) pour supprimer complètement les dépendances d'un rôle à supprimer.

En bref, les actions de suppression d'un rôle propriétaire d'objets sont :

```
REASSIGN OWNED BY role_a_supprimer TO role_remplacant;
DROP OWNED BY role_a_supprimer;
-- répétez les commandes ci-dessus pour chaque base de données de
 l'instance
DROP ROLE role_a_supprimer;
```

Lorsque les objets ne sont pas tous transférés au même rôle, il est préférable de gérer les exceptions manuellement, puis de réaliser les étapes ci-dessus pour le reste.

Si `DROP ROLE` est tenté alors que des objets dépendants sont toujours présents, il enverra des messages identifiant les objets à réaffecter ou supprimer.

22.5. Rôles prédéfinis

PostgreSQL fournit un ensemble de rôles prédéfinis qui permettent d'accéder à certaines fonctionnalités et informations privilégiées, couramment nécessaires. Les administrateurs (y compris les rôles qui ont le privilège `CREATEROLE`) peuvent octroyer (`GRANT`) ces rôles aux utilisateurs et/ou à d'autres rôles dans leur environnement, fournissant à ces utilisateurs un accès aux fonctionnalités et informations spécifiées.

Les rôles prédéfinis sont décrits dans Tableau 22.1. Notez que les autorisations spécifiques pour chacun des rôles peuvent changer à l'avenir à mesure que des fonctionnalités supplémentaires sont ajoutées. Les administrateurs doivent surveiller les notes de version pour les changements.

Tableau 22.1. Rôles prédéfinis

Rôle	Accès autorisé
pg_read_all_data	Lit toutes les données (tables, vues, séquences), comme si le rôle avait le droit `SELECT` sur ces objets, et le droit `USAGE` sur tous les schémas, même sans les avoir explicitement. Ce rôle n'a

Rôle	Accès autorisé
	pas l'attribut BYPASSRLS de configuré. Si RLS est utilisé, un administrateur pourrait vouloir ajouter BYPASSRLS sur les rôles ayant cet accès.
pg_write_all_data	Écrit toutes les données (tables, vues, séquences), comme si le rôle avait les droits INSERT, UPDATE, and DELETE sur ces objets, et le droit USAGE sur tous les schémas, même sans les avoir explicitement. Ce rôle n'a pas l'attribut BYPASSRLS de configuré. Si RLS est utilisé, un administrateur pourrait vouloir ajouter BYPASSRLS sur les rôles ayant cet accès.
pg_read_all_settings	Lit toutes les variables de configuration, y compris celles normalement visibles des seuls superutilisateurs.
pg_read_all_stats	Lit toutes les vues pg_stat_* et utilise plusieurs extensions relatives aux statistiques, y compris celles normalement visibles des seuls superutilisateurs.
pg_database_owner	None. Membership consists, implicitly, of the current database owner.
pg_stat_scan_tables	Exécute des fonctions de monitoring pouvant prendre des verrous ACCESS SHARE sur les tables, potentiellement pour une longue durée.
pg_monitor	Lit et exécute plusieurs vues et fonctions de monitoring. Ce rôle est membre de pg_read_all_settings, pg_read_all_stats et pg_stat_scan_tables.
pg_signal_backend	Envoie un signal à un autre processus serveur (par exemple pour annuler une requête ou fermer une session).
pg_read_server_files	Autoriser la lecture de fichiers sur tous les emplacements accessibles par le serveur de bases de données avec COPY et les autres fonctions d'accès de fichiers.
pg_write_server_files	Autoriser l'écriture de fichiers sur tous les emplacements accessibles par le serveur de bases de données avec COPY et les autres fonctions d'accès de fichiers.
pg_execute_server_program	Autoriser l'exécution de programmes sur le serveur de bases de données en tant que l'utilisateur qui exécute le moteur de bases de données, avec COPY et les autres fonctions qui permettent l'exécution d'un programme sur le serveur.
pg_checkpoint	Autoriser l'exécution de la commande CHECKPOINT.

Les rôles pg_monitor, pg_read_all_settings, pg_read_all_stats et pg_stat_scan_tables ont pour but de permettre aux administrateurs de configurer aisément un rôle en vue de superviser le serveur de base de données. Ils accordent un ensemble de privilèges permettant au rôle de lire plusieurs paramètres de configuration, statistiques et information système normalement réservés aux superutilisateurs.

Le rôle pg_database_owner a un membre implicite, à savoir le propriétaire de la base de données en cours. Comme tout rôle, il peut être propriétaire des objets et recevoir des autorisations d'accès. En conséquence, une fois que pg_database_owner a des droits dans une base modèle, chaque propriétaire d'une base créée à partir de ce modèle peut exercer ces droits. pg_database_owner ne peut pas être un membre d'un autre rôle, et il ne peut pas avoir de membres non implicites. Au départ, ce rôle est le propriétaire du schéma public, pour que chaque propriétaire de base gouverne l'utilisation locale de ce schéma.

Le rôle pg_signal_backend a pour but de permettre aux administrateurs d'autoriser des rôles de confiance mais non superutilisateurs à envoyer des signaux à d'autres processus. Actuellement, ce rôle active l'envoi de signaux pour annuler une requête exécutée par un autre processus ou pour terminer

une session. Néanmoins, un utilisateur membre de ce rôle ne peut pas envoyer ce type de signaux à un processus appartenant à un superutilisateur. Voir Section 9.27.2.

Les rôles `pg_read_server_files`, `pg_write_server_files` et `pg_execute_server_program` ont pour but de permettre aux administrateurs de disposer de rôles de confiance, mais non superutilisateur, capables d'accéder à des fichiers et d'exécuter des programmes sur le serveur de bases de données en tant que l'utilisateur qui exécute le moteur de bases de données. Comme ces rôles sont capables d'accéder à tout fichier sur le système de fichiers du serveur, ils contournent toutes les vérifications d'accès au niveau de la base de données lors d'accès directs aux fichiers. Ils peuvent utiliser cela pour obtenir un accès de niveau superutilisateur. De ce fait, une attention toute particulière doit être prise lors de l'affectation de ces rôles aux utilisateurs.

On portera une attention particulière en accordant ces rôles afin de garantir qu'ils ne sont utilisés qu'en cas de nécessiter et en comprenant que ces rôles donnent accès à des informations importantes.

Les administrateurs peuvent autoriser l'accès à ces rôles aux utilisateurs en utilisant la commande `GRANT` :

```
GRANT pg_signal_backend TO admin_user;
```

22.6. Sécurité des fonctions

Les fonctions, les triggers et les politiques de sécurité au niveau ligne autorisent à l'intérieur du serveur les utilisateurs à insérer du code que d'autres utilisateurs peuvent exécuter sans en avoir l'intention. Par conséquent, ces mécanismes permettent aux utilisateurs d'utiliser un « cheval de Troie » contre d'autres utilisateurs avec une relative facilité. La protection la plus forte est un contrôle strict sur qui peut définir des objets. Quand cela n'est pas possible, écrivez les requêtes en se référant seulement aux objets dont les propriétaires sont dignes de confiance. Supprimez de `search_path` tout schéma permettant à des utilisateurs non dignes de confiance de créer des objets.

Les fonctions sont exécutées à l'intérieur du processus serveur avec les droits au niveau système d'exploitation du démon serveur de la base de données. Si le langage de programmation utilisé par la fonction autorise les accès mémoire non contrôlés, il est possible de modifier les structures de données internes du serveur. Du coup, parmi d'autres choses, de telles fonctions peuvent dépasser les contrôles d'accès au système. Les langages de fonctions qui permettent un tel accès sont considérées « sans confiance » et PostgreSQL autorise uniquement les superutilisateurs à écrire des fonctions dans ces langages.

Chapitre 23. Administration des bases de données

Chaque instance d'un serveur PostgreSQL gère une ou plusieurs bases de données. Les bases de données sont donc le niveau hiérarchique le plus élevé pour organiser des objets SQL (« objets de base de données »). Ce chapitre décrit les propriétés des bases de données et comment les créer, les administrer et les détruire.

23.1. Aperçu

Un petit nombre d'objets, comme les rôles, les bases de données et les tablespaces, sont définis au niveau de l'instance et stockés dans le tablespace `pg_global`. À l'intérieur de l'instance résident plusieurs bases de données, isolées les unes des autres mais pouvant accéder aux objets du niveau instance. Dans chaque base se trouvent plusieurs schémas contenant des objets comme les tables et les fonctions. La hiérarchie complète est donc : instance, base de données, schéma, table (et autre type d'objet comme une fonction).

Lors de la connexion à un serveur de bases de données, un client doit indiquer le nom de la base dans sa demande de connexion. Il n'est pas possible d'accéder à plus d'une base par connexion. Néanmoins, les clients peuvent ouvrir plusieurs connexions à la même base ou à des bases différentes. La sécurité au niveau de la base a deux composantes : le contrôle d'accès (voir Section 21.1), géré au niveau de la connexion, et le contrôle d'autorisation (voir Section 5.7), géré par le système de droits. Les *foreign data wrappers* (voir postgres_fdw) permettent à des objets d'une base d'agir comme des proxys pour les objets d'une autre base ou d'une autre instance. L'ancien module dblink (voir dblink) fournit la même fonctionnalité. Par défaut, tous les utilisateurs peuvent se connecter à toutes les bases en utilisant toutes les méthodes d'authentification.

Si une instance PostgreSQL doit contenir des projets différents, sans relation, ou des utilisateurs qui, pour la plupart ne devraient pas être au courant des autres, il est recommandé de les placer dans des bases séparées et d'ajuster les autorisations et les contrôles d'accès de façon approprié. Si les projets et les utilisateurs sont liés et, de ce fait, devraient être capables d'utiliser les ressources des autres, ils devraient être placés dans la même base mais probablement dans des schémas différents ; ceci fournit une structure modulaire avec une isolation par espace de nom et un contrôle d'autorisation. Section 5.9 fournit plus d'informations sur la gestion des schémas.

Quand plusieurs bases sont créées dans la même instance, il est conseillé de faire bien attention que les bénéfices dépassent les risques et limitations. En particulier, l'impact qu'avoir des journaux de transactions partagés (voir Chapitre 30) a sur les options de sauvegarde et de restauration. Alors que des bases individuelles dans l'instance sont isolées de la perspective de l'utilisateur, elles sont fortement liées du point de vue de l'administrateur de bases de données.

Les bases de données sont créées avec la commande `CREATE DATABASE` (voir la Section 23.2) et détruites avec la commande `DROP DATABASE` (voir la Section 23.5). Pour déterminer l'ensemble des bases de données existantes, examinez le catalogue système `pg_database`, par exemple

```
SELECT datname
   FROM pg_database;
```

La méta-commande `\l` du programme psql et l'option en ligne de commande `-l` sont aussi utiles pour afficher les bases de données existantes.

> **Note**
>
> Le standard SQL appelle les bases de données des « catalogues » mais il n'y a aucune différence en pratique.

23.2. Création d'une base de données

Pour pouvoir créer une base de données, il faut que le serveur PostgreSQL soit lancé (voir la Section 19.3).

Les bases de données sont créées à l'aide de la commande SQL CREATE DATABASE :

```
CREATE DATABASE
   nom;
```

ou *nom* suit les règles habituelles pour les identifiants SQL. Le rôle actuel devient automatiquement le propriétaire de la nouvelle base de données. C'est au propriétaire de la base de données qu'il revient de la supprimer par la suite (ce qui supprime aussi tous les objets qu'elle contient, même s'ils ont un propriétaire différent).

La création de bases de données est une opération protégée. Voir la Section 22.2 sur la manière d'attribuer des droits.

Comme vous devez être connecté au serveur de base de données pour exécuter la commande `CREATE DATABASE`, reste à savoir comment créer la première base de données d'un site. La première base de données est toujours créée par la commande `initdb` quand l'aire de stockage des données est initialisée (voir la Section 19.2). Cette base de données est appelée `postgres`. Donc, pour créer la première base de données « ordinaire », vous pouvez vous connecter à `postgres`.

Deux bases de données supplémentaires, `template1`, et `template1`, sont aussi créées durant l'initialisation de l'instance de bases de données. Quand une nouvelle base de données est créée à l'intérieur du groupe, `template1` est généralement cloné. Cela signifie que tous les changements effectués sur `template1` sont propagés à toutes les bases de données créées ultérieurement. À cause de cela, évitez de créer des objets dans `template1` sauf si vous voulez les propager à chaque nouvelle base de données créée. `template0` a pour but d'être une copie vierge du contenu original de `template1`. Elle peut être clonée à la place de `template1` quand il est important de créer une base sans ces ajouts locaux. Pour plus de détails, voir la Section 23.3.

Pour plus de confort, il existe aussi un programme que vous pouvez exécuter à partir du shell pour créer de nouvelles bases de données, `createdb`.

```
createdb nom_base
```

`createdb` ne fait rien de magique. Il se connecte à la base de données `postgres` et exécute la commande `CREATE DATABASE`, exactement comme ci-dessus. La page de référence sur createdb contient les détails de son invocation. Notez que `createdb` sans aucun argument crée une base de données portant le nom de l'utilisateur courant.

> **Note**
>
> Le Chapitre 21 contient des informations sur la manière de restreindre l'accès à une base de données.

Parfois, vous voulez créer une base de données pour quelqu'un d'autre. Ce rôle doit devenir le propriétaire de la nouvelle base de données afin de pouvoir la configurer et l'administrer lui-même. Pour faire ceci, utilisez l'une des commandes suivantes :

```
CREATE DATABASE nom_base OWNER nom_role;
```

dans l'environment SQL ou

```
createdb -O nom_role nom_base
```

dans le shell. Seul le superutilisateur est autorisé à créer une base de données pour quelqu'un d'autre c'est-à-dire pour un rôle dont vous n'êtes pas membre.

23.3. Bases de données modèles

En fait, CREATE DATABASE fonctionne en copiant une base de données préexistante. Par défaut, cette commande copie la base de données système standard `template1`. Ainsi, cette base de données est le « modèle » à partir duquel de nouvelles bases de données sont créées. Si vous ajoutez des objets à `template1`, ces objets seront copiés dans les bases de données utilisateur créées ultérieurement. Ce comportement permet d'apporter des modifications locales au jeu standard d'objets des bases de données. Par exemple, si vous installez le langage de procédures PL/Perl dans `template1`, celui-ci sera automatiquement disponible dans les bases de données utilisateur sans qu'il soit nécessaire de faire quelque chose de spécial au moment où ces bases de données sont créées.

Néanmoins, CREATE DATABASE ne copie pas les droits GRANT au niveau base de données, attachés à la base source. La nouvelle base de données a les droits par défaut au niveau base.

Il y a une seconde base de données système standard appelée `template0`. Cette base de données contient les mêmes données que le contenu initial de `template1`, c'est-à-dire seulement les objets standards prédéfinis dans votre version de PostgreSQL. `template0` ne devrait jamais être modifiée après que le cluster des bases de données ait été créé. En indiquant à CREATE DATABASE de copier `template0` au lieu de `template1`, vous pouvez créer une base de données utilisateur « vierge » (une base où aucun objet défini par un utilisateur n'existe et où les objets systèmes n'ont pas été modifiés) qui ne contient aucun des ajouts locaux à `template1`. Ceci est particulièrement pratique quand on restaure une sauvegarde réalisé avec `pg_dump` : le script de dump devrait être restauré dans une base de données vierge pour être sûr de recréer le contenu correct de la base de données sauvegardée, sans survenue de conflits avec des objets qui auraient été ajoutés à `template1`.

Une autre raison habituelle de copier `template0` au lieu de `template1` est que les nouvelles options d'encodage et de locale peuvent être indiquées lors de la copie de `template0`, alors qu'une copie de `template1` doit utiliser les mêmes options. Ceci est dû au fait que `template1` pourrait contenir des données spécifiques à l'encodage ou à la locale alors que `template0` n'est pas modifiable.

Pour créer une base de données à partir de `template0`, on écrit :

```
CREATE DATABASE nom_base TEMPLATE template0;
```

dans l'environnement SQL ou

```
createdb -T template0 nom_base
```

dans le shell.

Il est possible de créer des bases de données modèles supplémentaires et, à vrai dire, on peut copier n'importe quelle base de données d'un cluster en la désignant comme modèle pour la commande CREATE DATABASE. Cependant, il importe de comprendre, que ceci n'est pas (encore) à prendre comme une commande « COPY DATABASE » de portée générale. La principale limitation est qu'aucune autre session ne peut être connectée à la base source tant qu'elle est copiée. CREATE DATABASE échouera si une autre connexion existe à son lancement. Lors de l'opération de copie, les nouvelles connexions à la base source sont empêchées.

Deux drapeaux utiles existent dans `pg_database` pour chaque base de données : les colonnes `datistemplate` et `datallowconn`. `datistemplate` peut être positionné à vrai pour indiquer qu'une base de données a vocation à servir de modèle à CREATE DATABASE. Si ce drapeau est positionné à vrai, la base de données peut être clonée par tout utilisateur ayant le droit CREATEDB ; s'il est positionné à faux, seuls les superutilisateurs et le propriétaire de la base de données peuvent la cloner. Si `datallowconn` est positionné à faux, alors aucune nouvelle connexion à cette base de données n'est autorisée (mais les sessions existantes ne sont pas terminées simplement en positionnant

ce drapeau à faux). La base de données `template0` est normalement marquée `datallowconn = false` pour empêcher qu'elle ne soit modifiée. Aussi bien `template0` que `template1` devraient toujours être marquées `datistemplate = true`.

> **Note**
>
> `template1` et `template0` n'ont pas de statut particulier en dehors du fait que `template1` est la base de données source par défaut pour la commande `CREATE DATABASE`. Par exemple, on pourrait supprimer `template1` et la recréer à partir de `template0` sans effet secondaire gênant. Ce procédé peut être utile lorsqu'on a encombré `template1` d'objets inutiles. (Pour supprimer `template1`, cette dernière doit avoir le statut `pg_database.datistemplate` à `false`.
>
> La base de données `postgres` est aussi créé quand le groupe est initialisé. Cette base de données a pour but de devenir une base de données par défaut pour la connexion des utilisateurs et applications. C'est une simple copie de `template1` et peut être supprimée et re-créée si nécessaire.

23.4. Configuration d'une base de données

Comme il est dit dans le Chapitre 20, le serveur PostgreSQL offre un grand nombre de variables de configuration à chaud. Vous pouvez spécifier des valeurs par défaut, valables pour une base de données particulière, pour nombre de ces variables.

Par exemple, si pour une raison quelconque vous voulez désactiver l'optimiseur GEQO pour une base de données particulière, vous n'avez pas besoin de le désactiver pour toutes les bases de données ou de faire en sorte que tout client se connectant exécute la commande `SET geqo TO off;`. Pour appliquer ce réglage par défaut à la base de données en question, vous pouvez exécuter la commande :

```
ALTER DATABASE ma_base SET geqo TO off;
```

Cela sauvegarde le réglage (mais ne l'applique pas immédiatement). Lors des connexions ultérieures à cette base de données, tout se passe comme si la commande `SET geqo TO off` est exécutée juste avant de commencer la session. Notez que les utilisateurs peuvent cependant modifier ce réglage pendant la session ; il s'agit seulement d'un réglage par défaut. Pour annuler un tel réglage par défaut, utilisez `ALTER DATABASE` *`nom_base`* `RESET` *`nomvariable`*.

23.5. Détruire une base de données

Les bases de données sont détruites avec la commande DROP DATABASE :

```
DROP DATABASE nom;
```

Seul le propriétaire de la base de données ou un superutilisateur peut supprimer une base de données. Supprimer une base de données supprime tous les objets qui étaient contenus dans la base. La destruction d'une base de données ne peut pas être annulée.

Vous ne pouvez pas exécuter la commande `DROP DATABASE` en étant connecté à la base de données cible. Néanmoins, vous pouvez être connecté à une autre base de données, ceci incluant la base `template1`. `template1` pourrait être la seule option pour supprimer la dernière base utilisateur d'un groupe donné.

Pour une certaine facilité, il existe un script shell qui supprime les bases de données, dropdb :

```
dropdb nom_base
```

(Contrairement à `createdb`, l'action par défaut n'est pas de supprimer la base possédant le nom de l'utilisateur en cours.)

23.6. Tablespaces

Les tablespaces dans PostgreSQL permettent aux administrateurs de bases de données de définir l'emplacement dans le système de fichiers où seront stockés les fichiers représentant les objets de la base de données. Une fois créé, un tablespace peut être référencé par son nom lors de la création d'objets.

En utilisant les tablespaces, un administrateur peut contrôler les emplacements sur le disque d'une installation PostgreSQL. Ceci est utile dans au moins deux cas. Tout d'abord, si la partition ou le volume sur lequel le groupe a été initialisé arrive à court d'espace disque mais ne peut pas être étendu, un tablespace peut être créé sur une partition différente et utilisé jusqu'à ce que le système soit reconfiguré.

Deuxièmement, les tablespaces permettent à un administrateur d'utiliser sa connaissance des objets de la base pour optimiser les performances. Par exemple, un index qui est très utilisé peut être placé sur un disque très rapide et disponible, comme un périphérique mémoire. En même temps, une table stockant des données archivées et peu utilisée ou dont les performances ne portent pas à conséquence pourra être stockée sur un disque système plus lent, moins cher.

> **Avertissement**
>
> Même s'ils sont positionnés en dehors du répertoire de données principal de PostgreSQL, les tablespaces font partie intégrante de l'instance et *ne peuvent pas* être considérés comme des ensembles autonomes de fichiers de données et ne peuvent par conséquent pas être rattachés à une autre instance ou sauvegardés individuellement. De la même façon, si un tablespace est perdu (fichier supprimé, défaillance du disque dur, etc), l'instance pourrait devenir illisible ou même incapable de démarrer. Positionner un tablespace sur un système de fichiers temporaire comme un disque RAM met en péril la fiabilité de l'instance entière.

Pour définir un tablespace, utilisez la commande CREATE TABLESPACE, par exemple :

```
CREATE TABLESPACE espace_rapide LOCATION '/ssd1/postgresql/data';
```

L'emplacement doit être un répertoire existant, dont le propriétaire doit être l'utilisateur du système d'exploitation démarrant PostgreSQL. Tous les objets créés par la suite dans le tablespace seront stockés dans des fichiers contenus dans ce répertoire. Cet emplacement ne doit pas être amovible ou volatile, sinon l'instance pourrait cesser de fonctionner si le tablespace venait à manquer ou être perdu.

> **Note**
>
> Il n'y a généralement aucune raison de créer plus d'un tablespace sur un système de fichiers logique car vous ne pouvez pas contrôler l'emplacement des fichiers individuels à l'intérieur de ce système de fichiers logique. Néanmoins, PostgreSQL ne vous impose aucune limitation et, en fait, il n'est pas directement conscient des limites du système de fichiers sur votre système. Il stocke juste les fichiers dans les répertoires que vous lui indiquez.

La création d'un tablespace lui-même doit être fait en tant que superutilisateur de la base de données mais, après cela, vous pouvez autoriser des utilisateurs standards de la base de données à l'utiliser. Pour cela, donnez-leur le droit `CREATE` sur le tablespace.

Les tables, index et des bases de données entières peuvent être affectés à des tablespaces particuliers. Pour cela, un utilisateur disposant du droit `CREATE` sur un tablespace donné doit passer le nom

du tablespace comme paramètre de la commande. Par exemple, ce qui suit crée une table dans le tablespace `espace1` :

```
CREATE TABLE foo(i int) TABLESPACE espace1;
```

Autrement, utilisez le paramètre default_tablespace :

```
SET default_tablespace = espace1;
CREATE TABLE foo(i int);
```

Quand `default_tablespace` est configuré avec autre chose qu'une chaîne vide, il fournit une clause `TABLESPACE` implicite pour les commandes `CREATE TABLE` et `CREATE INDEX` qui n'en ont pas d'explicites.

Il existe aussi un paramètre temp_tablespaces, qui détermine l'emplacement des tables et index temporaires, ainsi les fichiers temporaires qui sont utilisés pour le tri de gros ensembles de données. Ce paramètre peut aussi contenir une liste de tablespaces, plutôt qu'une seule, pour que la charge associée aux objets temporaires soit répartie sur plusieurs tablespaces. Un membre de la liste est pris au hasard à chaque fois qu'un objet temporaire doit être créé.

Le tablespace associé avec une base de données est utilisé pour stocker les catalogues système de la base. De plus, il est l'espace par défaut pour les tables, index et fichiers temporaires créés à l'intérieur de cette base de données si aucune clause `TABLESPACE` n'est fournie et qu'aucune sélection n'est spécifiée par `default_tablespace` ou `temp_tablespaces` (comme approprié). Si une base de données est créée sans spécifier de tablespace pour elle, le serveur utilise le même tablespace que celui de la base modèle utilisée comme copie.

Deux tablespaces sont automatiquement créés lors de l'initialisation du cluster de bases de données. Le tablespace `pg_global` est utilisé pour les catalogues système partagés. Le tablespace `pg_default` est le tablespace par défaut des bases de données `template1` et `template0` (et, du coup, sera le tablespace par défaut pour les autres bases de données sauf en cas de surcharge par une clause `TABLESPACE` dans `CREATE DATABASE`).

Une fois créé, un tablespace peut être utilisé à partir de toute base de données si l'utilisateur le souhaitant dispose du droit nécessaire. Ceci signifie qu'un tablespace ne peut pas supprimé tant que tous les objets de toutes les bases de données utilisant le tablespace n'ont pas été supprimés.

Pour supprimer un tablespace vide, utilisez la commande DROP TABLESPACE.

Pour déterminer l'ensemble des tablespaces existants, examinez le catalogue système `pg_tablespace`, par exemple

```
SELECT spcname FROM pg_tablespace;
```

La métacommande `\db` du programme psql est aussi utile pour afficher les tablespaces existants.

PostgreSQL utilise des liens symboliques pour simplifier l'implémentation des tablespaces. Ceci signifie que les tablespaces peuvent être utilisés *seulement* sur les systèmes supportant les liens symboliques.

Le répertoire `$PGDATA/pg_tblspc` contient des liens symboliques qui pointent vers chacun des tablespaces utilisateur dans le groupe. Bien que non recommandé, il est possible d'ajuster la configuration des tablespaces à la main en redéfinissant ces liens. Cette opération ne doit jamais être réalisée alors que le serveur est en cours d'exécution. Notez qu'avec les versions 9.1 et antérieures de PostgreSQL 9.1, vous aurez aussi besoin de mettre à jour le catalogue `pg_tablespace` avec les nouveaux emplacements. (Si vous ne le faites pas, `pg_dump` continuera à afficher les anciens emplacements des tablespaces.)

Chapitre 24. Localisation

Ce chapitre décrit les fonctionnalités de régionalisation (ou localisation) disponibles, du point de vue de l'administrateur. PostgreSQL fournit deux approches différentes pour la gestion de la localisation :

- l'utilisation des fonctionnalités de locales du système d'exploitation pour adapter à la locale l'ordre des tris, le formatage des chiffres, la traduction des messages et d'autres aspects. Ceci est couvert dans Section 24.1 et Section 24.2 ;
- la fourniture d'un certain nombre d'encodages différents, afin de pouvoir stocker du texte dans toutes les langues, et de la capacité à traduire l'encodage entre client et serveur. Ceci est couvert dans Section 24.3.

24.1. Support des locales

Le support des *locales* fait référence à une application respectant les préférences culturelles portant sur les alphabets, le tri, le format des nombres, etc. PostgreSQL utilise les outils C et POSIX du standard ISO fournis par le système d'exploitation du serveur. Pour plus d'informations, consulter la documentation du système.

24.1.1. Aperçu

Le support des locales est configuré automatiquement lorsqu'une instance de base de données est créée avec `initdb`. `initdb` initialise l'instance avec la valeur de locale de son environnement d'exécution par défaut. Si le système est déjà paramétré pour utiliser la locale souhaitée pour le cluster, il n'y a donc rien d'autre à faire. Si l'on désire une locale différente, ou si celle du serveur n'est pas connue avec certitude, il est possible d'indiquer à `initdb` la locale à l'aide de l'option `--locale`. Par exemple :

```
initdb --locale=sv_SE
```

Cet exemple pour les systèmes Unix positionne la locale au suédois (`sv`), tel que parlé en Suède (`SE`). Parmi les autres possibilités, on peut inclure `en_US` (l'anglais américain) ou `fr_CA` (français canadien). Si plusieurs ensembles de caractères peuvent être utilisés pour une locale, alors les spécifications peuvent prendre la forme *`langage_territoire.codeset`*. Par exemple, `fr_BE.UTF-8` représente la langue française, telle qu'elle est parlée en Belgique (BE), avec un encodage UTF-8.

Les locales disponibles et leurs noms dépendent de l'éditeur du système d'exploitation et de ce qui est installé. Sur la plupart des systèmes Unix, la commande `locale -a` fournit la liste des locales disponibles. Windows utilise des noms de locale plus verbeux, comme `German_Germany` ou `Swedish_Sweden.1252`, mais le principe est le même.

Il est parfois utile de mélanger les règles de plusieurs locales, par exemple d'utiliser les règles de tri anglais avec des messages en espagnol. Pour cela, des sous-catégories de locales existent qui ne contrôlent que certains aspects des règles de localisation :

`LC_COLLATE`	Ordre de tri des chaînes de caractères
`LC_CTYPE`	Classification de caractères (Qu'est-ce qu'une lettre ? Et la majuscule équivalente ?)
`LC_MESSAGES`	Langue des messages
`LC_MONETARY`	Formatage des valeurs monétaires
`LC_NUMERIC`	Formatage des nombres
`LC_TIME`	Formatage des dates et heures

Les noms des catégories se traduisent en ceux des options d'`initdb` pour surcharger le choix de locale d'une catégorie donnée. Par exemple, pour utiliser la locale français canadien, mais avec

des règles américaines de formatage monétaire, utilisez `initdb --locale=fr_CA --lc-monetary=en_US`.

Si vous voulez un système qui se comporte comme s'il n'avait aucun support des locales, utilisez les locales spéciales `C`, ou l'équivalent `POSIX`.

Certaines catégories de locales doivent être fixées à la création de la base de données. Elles peuvent différer entre bases de données, mais, une fois la base créée, elles ne peuvent plus être modifiées. `LC_COLLATE` et `LC_CTYPE` sont ces catégories. Elles affectent l'ordre de tri des index, et doivent donc rester inchangées, les index sur les colonnes de texte risquant d'être corrompus dans le cas contraire. (Mais vous pouvez lever ces restrictions grâce aux collations, comme discuté dans Section 24.2.) Les valeurs par défaut de ces catégories sont déterminées à l'exécution d'`initdb`, et utilisées à la création de nouvelles bases de données, sauf indication contraire dans la commande `CREATE DATABASE`.

Les autres catégories de locale peuvent être modifiées à n'importe quel moment en configurant les variables d'environnement de même nom (voir Section 20.11.3 pour de plus amples détails). Les valeurs choisies par `initdb` sont en fait juste écrites dans le fichier de configuration `postgresql.conf` pour servir de valeurs par défaut au démarrage du serveur. Si elles sont supprimées du fichier `postgresql.conf`, le serveur hérite des paramètres de son environnement d'exécution.

Notez que les locales du serveur sont déterminées par les variables d'environnement vues par le serveur, pas par celles de l'environnement d'un quelconque client. Il est donc important de configurer les bons paramètres de locales avant le démarrage du serveur. En conséquence, si les locales du client et du serveur diffèrent, les messages peuvent apparaître dans des langues différentes en fonction de leur provenance.

> **Note**
>
> Hériter la locale de l'environnement d'exécution signifie, sur la plupart des systèmes d'exploitation, la chose suivante : pour une catégorie de locales donnée (disons la collation) les variables d'environnement suivantes sont consultées dans cet ordre jusqu'à en trouver une qui est définie : `LC_ALL`, `LC_COLLATE` (ou la variable correspondant à la catégorie) et `LANG`. Si aucune de ces variables n'est définie, la locale devient par défaut `C`.
>
> Certaines bibliothèques de localisation regardent aussi la variable d'environnement `LANGUAGE`, laquelle surcharge tout autre paramètre pour fixer la langue des messages. En cas de doute, lire la documentation du système d'exploitation, en particulier la partie concernant gettext.

Pour permettre la traduction des messages dans la langue préférée de l'utilisateur, NLS doit avoir été activé à la compilation (`configure --enable-nls`). Le reste de l'outillage des locales est automatiquement inclus.

24.1.2. Comportement

Le paramétrage de la locale influence les fonctionnalités SQL suivantes :

- l'ordre de tri dans les requêtes utilisant `ORDER BY` ou les opérateurs de comparaison standards sur des données de type texte ;
- les fonctions `upper`, `lower` et `initcap` ;
- les opérateurs de correspondance de motifs (`LIKE`, `SIMILAR TO` et les expressions rationnelles de type POSIX) les locales affectent les opérateurs insensibles à la classe, et le classement des caractères par les expressions rationnelles portant sur des caractères ;

- la famille des fonctions `to_char` ;
- l'utilisation d'index avec des clauses `LIKE`.

L'inconvénient du support dans PostgreSQL des locales (autres que `C` ou `POSIX`) est l'impact sur les performances. Il ralentit la gestion des caractères, et empêche l'utilisation des index ordinaires lors d'un `LIKE`. Pour cette raison, il est préférable de n'utiliser les locales qu'en cas de réel besoin.

Toutefois, il existe plusieurs classes d'opérateurs personnalisées pour permettre à PostgreSQL d'utiliser des index avec les clauses `LIKE` et une locale autre que `C`. Ces classes permettent la création d'un index réalisant une stricte comparaison caractère par caractère, en ignorant les règles de comparaison des locales ; se référer à la Section 11.10 pour plus d'informations. Une autre possibilité est de créer des index en utilisant la collation `C`, comme discuté dans Section 24.2.

24.1.3. Sélectionner les locales

Les locales peuvent être sélectionnées dans différentes configurations suivant les besoins. L'aperçu ci-dessus montrait comment les locales sont spécifiées en utilisant `initdb` pour configurer les valeurs par défaut pour l'instance complète. La liste suivante montre où les locales peuvent être sélectionnées. Chaque élément fournit les valeurs par défaut pour les éléments suivants, et chaque élément inférieur permet de surcharger les valeurs par défaut sur une granularité plus fine.

1. Comme expliqué ci-dessus, l'environnement du système d'exploitation fournit les valeurs par défaut pour les locales d'une instance tout juste initialisée. Dans de nombreux cas, c'est suffisant : si le système d'exploitation est configuré pour le langage/territoire désiré, alors PostgreSQL se comportera par défaut suivant cette locale.

2. Comme affiché ci-dessus, les options en ligne de commande d'`initdb` indiquent les paramétrages de la locale pour une instance tout juste initialisée. Utilisez-les si le système d'exploitation n'a pas la configuration de locale que vous souhaitez pour votre système de bases de données.

3. Une locale peut être sélectionnée séparément pour chaque base de données. La commande SQL `CREATE DATABASE` et son équivalent en ligne de commande, `createdb`, ont des options pour cela. Utilisez-la par exemple si une instance contient des bases avec plusieurs propriétaires ayant des besoins différents.

4. Les configurations de locale peuvent être faites pour les colonnes individuelles des tables. Il faut passer par un objet SQL appelé une *collation*, qui est expliqué dans Section 24.2. Utilisez-la par exemple pour trier des données dans différentes langues ou pour personnaliser l'ordre de tri d'une table particulière.

5. Enfin, les locales peuvent être sélectionnées pour une requête individuelle. De nouveau, cela utilise des objets SQL de collation. Cela permet de modifier l'ordre de tri en se basant sur un choix au moment de l'exécution ou pour une expérimentation.

24.1.4. Fournisseurs de locale

PostgreSQL accepte plusieurs *fournisseurs de locale*. Elle indique la bibliothèque qui fournit les données de locale. Un nom de fournisseur standard est `libc`, qui utilise les locales fournies par la bibliothèque C du système d'exploitation. Ce sont les locales utilisées par la plupart des outils fournis par le système d'exploitation. Un autre fournisseur est `icu`, qui utilise la bibliothèque externe ICU. Les locales ICU peuvent seulement être utilisées si le support d'ICU a été configuré lors de la construction de PostgreSQL.

Les commandes et outils qui sélectionnent les paramètres de la locale, comme décrit ci-dessus, ont tous une option pour sélectionner le fournisseur de la locale. Les exemples montrés ci-dessus utilisent le fournisseur `libc`, qui est le fournisseur par défaut. Voici un exemple pour initialiser une instance de bases en utilisant le fournisseur ICU :

```
initdb --locale-provider=icu --icu-locale=en
```

Voir la description des commandes et programmes respectifs pour les détails. Notez que vous pouvez mélanger les fournisseurs de locale sur différentes granularités, par exemple utiliser `libc` par défaut pour l'instance, mais avoir une base qui utilise le fournisseur `icu`, puis avoir les objets de collation utilisant un des fournisseurs dans ces bases.

Le fournisseur de locale à utiliser dépend des besoins individuels. Pour des utilisations basiques, chaque fournisseur donnera des résultats adéquats. Pour le fournisseur libc, il dépend de ce qu'offre le système d'exploitation ; certains systèmes d'exploitation sont meilleurs que d'autres. Pour des utilisations avancées, ICU offre plus de variants de locale et plus d'options de personnalisation.

24.1.5. Problèmes

Si le support des locales ne fonctionne pas au regard des explications ci-dessus, vérifiez que le support au niveau du système d'exploitation est correctement configuré. Pour vérifier les locales installées sur le système, on peut utiliser la commande `locale -a`, si elle est fournie avec le système d'exploitation.

Il faut vérifier que PostgreSQL utilise effectivement la locale que vous pensez. Les paramètres `LC_COLLATE` et `LC_CTYPE` sont déterminés lors de la création de la base de données, et ne peuvent pas être modifiés, sauf en créant une nouvelle base de données. D'autres paramètres de locale, y compris `LC_MESSAGES` et `LC_MONETARY`, sont déterminés initialement par l'environnement dans lequel le serveur est lancé, mais peuvent être modifiés pendant l'exécution. Pour vérifier la locale active, utilisez la commande `SHOW`.

Le répertoire `src/test/locale` de la distribution source contient une série de tests pour le support des locales dans PostgreSQL.

Certaines applications clientes, qui gèrent les erreurs en provenance du serveur en analysant le texte des messages associés, auront évidemment des problèmes lorsque ces messages du serveur seront dans une autre langue. Les auteurs de telles applications sont invités à utiliser plutôt le mécanisme de code d'erreur.

Le maintien de catalogues de traduction des messages nécessite les efforts permanents de beaucoup de volontaires souhaitant voir PostgreSQL parler correctement leur langue préférée. Si certains messages dans une langue ne sont pas disponibles, ou pas complètement traduits, toute aide est la bienvenue. Pour apporter votre aide à ce projet, consultez le Chapitre 57, ou écrivez à la liste de diffusion des développeurs.

24.2. Support des collations

Cette fonctionnalité permet de définir par colonne, ou pour chaque requête, la collation utilisée pour déterminer l'ordre de tri et le classement des caractères. Elle permet de lever la restriction sur les paramètres `LC_COLLATE` et `LC_CTYPE` d'une base de données, qui ne pouvaient pas être modifiés après sa création.

24.2.1. Concepts

Conceptuellement, toute expression d'un type de donnée supportant les collations a une collation. (Les types de données intégrés supportant les collations sont `text`, `varchar`, et `char`. Les types de données définis par l'utilisateur peuvent aussi être marqués comme supportant les collations. Bien entendu, un *domaine* défini sur un type de données supportant les collations les supporte aussi.) Si l'expression est une référence de colonne, la collation de l'expression est déterminée par la collation de la colonne. Si l'expression est une constante, la collation sera celle par défaut du type de la constante. La collation d'une expression plus complexe est déterminée à partir des différentes collations de ses entrées, comme cela est décrit ci-dessous.

Une expression peut prendre la collation par défaut, « default », c'est-à-dire la collation définie au niveau de la base de données. Il est possible que la collation d'une expression soit indéterminée. Dans un tel cas, les tris et les autres opérations ayant besoin de connaître la collation vont échouer.

Lorsque la base de données doit réaliser un tri ou classement de caractères, elle utilise la collation de l'expression entrée. Ce cas se présentera, par exemple, avec des clauses `ORDER BY`, ou des appels à des fonctions ou opérateurs tels que `<`. La collation à appliquer à une clause `ORDER BY` est simplement la collation de la clé de tri. La collation à appliquer pour l'appel à une fonction ou à un opérateur est dérivée des arguments, comme décrit plus bas. En plus de s'appliquer aux opérateurs de comparaison, les collations sont également prises en compte par les fonctions qui convertissent entre minuscules et majuscules, comme `lower`, `upper` et `initcap`, par les opérateurs de correspondance de motifs, et par `to_char` et les fonctions affiliées.

Pour un appel à une fonction ou un opérateur, la collation dérivée des collations des arguments est utilisée à l'exécution de l'opération. Si le type du résultat de la fonction ou de l'opérateur supporte les collations, alors cette collation est utilisée dès l'analyse en tant que la collation de la fonction ou de l'opérateur, au cas où une expression voisine nécessiterait de la connaître.

Le *calcul de la collation* d'une expression est réalisé implicitement ou explicitement. Cette distinction affecte la façon dont les collations sont combinées entre elles quand plusieurs collations différentes apparaissent dans une expression. Une dérivation explicite utilise la clause `COLLATE` ; dans tous les autres cas, la dérivation de collation est implicite. Les règles suivantes s'appliquent lorsque plusieurs collations doivent être utilisée en même temps, par exemple dans un appel à une fonction :

1. Si une expression en entrée porte une dérivation de collation explicite, alors toutes les collations explicitement dérivées des autres expressions en entrée doivent être identiques, sinon une erreur est levée. Si une collation explicitement dérivée est présente, elle est le résultat de la combinaison des collations.

2. Dans les autres cas, toutes les expressions en entrée doivent avoir la même collation, qu'elle soit implicitement dérivée, ou celle par défaut. Si est présente une collation autre que celle par défaut, alors elle est aussi le résultat de la combinaison des collations. Sinon, le résultat correspond à la collation par défaut.

3. S'il existe, parmi les expressions en entrée, des collations implicites, qui ne sont pas celles par défaut, et qui entrent en conflit, alors la combinaison ne peut aboutir qu'à une collation indéterminée. Ce n'est pas une erreur, sauf si la fonction appelée requiert une application de la collation. Dans ce cas, une erreur est renvoyée lors de l'exécution.

Par exemple, considérez la table définie de la façon suivante:

```
CREATE TABLE test1 (
    a text COLLATE "de_DE",
    b text COLLATE "es_ES",
    ...
);
```

Ensuite, dans la requête

```
SELECT a < 'foo' FROM test1;
```

la comparaison `<` est réalisée selon les règles de la locale `de_DE`, car l'expression combine une collation dérivée implicitement avec la collation par défaut. Mais, dans la requête

```
 SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

la comparaison est effectuée en utilisant les règles de la locale `fr_FR`, parce que l'utilisation explicite de cette locale prévaut sur la locale implicite. De plus, avec la requête

```
SELECT a < b FROM test1;
```

l'analyseur ne dispose pas des éléments pour déterminer quelle collation employer, car les collations des colonnes `a` et `b` sont différentes. Comme l'opérateur `<` a besoin de savoir la locale à utiliser, on obtiendra une erreur. Elle peut être résolue en attachant une déclaration de collation explicite à l'une ou l'autre des expressions d'entrées, soit :

```
SELECT a < b COLLATE "de_DE" FROM test1;
```

ou de manière équivalente

```
SELECT a COLLATE "de_DE" < b FROM test1;
```

Toutefois, dans ce cas suivant, structurellement similaire,

```
SELECT a || b FROM test1;
```

il n'y aura pas d'erreur, car l'opérateur `||` ne tient pas compte des collations : son résultat sera le même quelle qu'elle soit.

La collation assignée à une fonction ou à une combinaison d'un opérateur avec ses expressions d'entrées s'applique également au résultat de la fonction ou de l'opérateur, si le résultat est d'un type supportant les collations. Ainsi, dans la requête

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

le tri sera réalisé en fonction des règles de la locale `de_DE`. Mais cette requête :

```
SELECT * FROM test1 ORDER BY a || b;
```

retournera une erreur car, bien que l'opérateur `||` ne tienne pas compte des collations de ses expressions, la clause `ORDER BY`, elle, en tient compte. Comme précédemment, ce conflit peut être résolu par l'emploi d'une déclaration explicite de la collation :

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

24.2.2. Gestion des collations

Une collation est un objet du catalogue SQL qui associe un nom SQL à une locale fournie par les bibliothèques installées sur le système. Une définition de collation a un *fournisseur*, qui spécifie quelle bibliothèque fournit les données locales. L'un des fournisseurs standards est `libc`, qui utilise les locales fournies par la bibliothèque C du système. Ce sont les locales utilisées par la plupart des outils

du système. Un autre fournisseur est `icu`, qui utilise la bibliothèque externe ICU. Les locales ICU ne peuvent être utilisées que si le support d'ICU a été configuré lors de la compilation de PostgreSQL.

Un objet de type collation fourni par la `libc` pointe sur une combinaison de paramètres `LC_COLLATE` et `LC_CTYPE`, comme acceptés par l'appel système `setlocale()`. (Comme le nom le suggère, le principal objectif d'une collation est de positionner `LC_COLLATE`, qui contrôle l'ordre de tri. Dans la pratique, il est très rarement nécessaire de définir un paramètre `LC_CTYPE` différent de `LC_COLLATE`. Il est donc plus facile de regrouper ces deux paramètres dans un même concept, que de créer une infrastructure différente simplement pour pouvoir positionner `LC_CTYPE` pour chaque expression.) De la même façon, une collation `libc` est liée à un encodage de jeu de caractère (voir Section 24.3). Le même nom de collation peut exister pour différents encodages.

Un objet de type collation fourni par `icu` pointe sur un collateur nommé fourni par la bibliothèque ICU. ICU ne permet pas de séparer « collate » et « ctype », ils sont donc toujours les mêmes. De même, les collations ICU sont indépendantes de l'encodage ; il n'y a donc toujours qu'une seule collation ICU pour un nom donné dans une base de données.

24.2.2.1. Collations standard

Les collations nommées `default`, `C`, et `POSIX` sont disponibles sur toutes les plateformes. Des collations complémentaires peuvent être disponibles, ou non, en fonction du support au niveau du système d'exploitation. La collation `default` utilise les valeurs de `LC_COLLATE` et `LC_CTYPE` définies à la création de la base de données. Les collations `C` et `POSIX` spécifient toutes deux le comportement « C traditionnel », dans lequel seuls les caractères ASCII de « `A` » à « `Z` » sont considérés comme des lettres, et les tris sont ordonnés strictement par valeur de l'octet du code caractère.

En complément, la collation du standard SQL, nommée `ucs_basic`, est disponible avec l'encodage `UTF8`. Elle est équivalente à `C`, et trie les données par le point de code Unicode.

24.2.2.2. Collations prédéfinies

Si le système d'exploitation permet de supporter plusieurs locales dans un même programme (fonction `newlocale` et fonctions conjointes), ou si le support d'ICU est configuré, alors `initdb` peuplera le catalogue système `pg_collation` avec toutes les locales trouvées sur le système d'exploitation lors de l'initialisation de l'instance.

Pour inspecter les locales actuellement disponibles, utilisez la requête `SELECT * FROM pg_collation`, ou la commande `\dOS+` dans psql.

24.2.2.2.1. Collations libc

Par exemple, le système d'exploitation peut fournir une locale appelée `de_DE.utf8`. `initdb` créera alors une collation nommée `de_DE.utf8` pour le jeu de caractère `UTF8`, pour laquelle `LC_COLLATE` et `LC_CTYPE` seront positionnés à `de_DE.utf8`. Il créera aussi une collation avec un nom amputé de l'étiquette `.utf8`. Ainsi, vous pourrez utiliser cette collation sous le nom `de_DE`, plus facile à écrire, et moins dépendant du jeu de caractères. Néanmoins, notez que l'ensemble initial des noms de collation `initdb` est dépendant de la plateforme.

Le jeu de collation par défaut fourni par `libc` pointe directement vers les locales installées sur le système, qui peuvent être listées avec la commande `locale -a`. Dans le cas où l'on a besoin d'une collation `libc` avec des valeurs `LC_COLLATE` et `LC_CTYPE` différentes, ou si de nouvelles locales sont installées sur le système après l'initialisation de la base de données, une nouvelle collation peut être créée par la commande CREATE COLLATION. On peut aussi importer en masse de nouvelles locales du système d'exploitation avec la fonction `pg_import_system_collations()`.

Dans une même base de données, seules les collations qui utilisent le jeu de caractères de la base de données sont prises en compte. Les autres entrées de `pg_collation` sont ignorées. Ainsi, une collation au nom tronqué, comme `de_DE`, peut être considérée unique au sein d'une base, bien qu'elle ne le soit pas globalement. L'utilisation de collations au nom tronqué est d'ailleurs recommandée : ce sera une chose de moins à changer si vous avez besoin de changer le jeu de caractères de la base

de données. Notez toutefois que les collations `default`, `C`, et `POSIX` peuvent être utilisées sans se soucier de l'encodage de la base de données.

PostgreSQL considère les collations comme des objets distincts et incompatibles entre eux, même si elles possèdent des propriétés identiques. Ainsi, par exemple,

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

va afficher une erreur alors que les collations `C` et `POSIX` possèdent des propriétés strictement identiques. Il n'est donc pas recommandé de mélanger des collations dont le nom est complet avec des collations dont le nom l'est pas.

24.2.2.2.2. Collations ICU

Avec ICU, il n'est pas raisonnable d'énumérer tous les noms de locales possibles. ICU utilise un système de nommage particulier pour les locales, mais il y a plus de façons de nommer une locale qu'il n'y en a réellement de distinctes. `initdb` utilise les API d'ICU pour extraire un jeu de locales distinctes, afin de peupler le jeu initial de collations. Les collations fournies par ICU sont créées dans l'environnement SQL avec des noms suivant le format d'étiquettes de langues BCP 47, plus une extension d'« utilisation privée » `-x-icu`, ajoutée pour les distinguer des locales de libc.

Voici quelques exemples de collations pouvant être créées :

`de-x-icu`

Collation allemande, variante par défaut

`de-AT-x-icu`

Collation allemande pour l'Autriche, variante par défaut

(Il y a aussi, par exemple, `de-DE-x-icu` ou `de-CH-x-icu` mais, au moment de l'écriture de ces lignes, elles sont équivalentes à `de-x-icu`.)

`und-x-icu` (pour « undefined »)

Collation « racine » ICU. Utilisez ceci pour avoir un ordre de tri raisonnable et linguistiquement agnostique.

Certains encodages (les moins fréquemment utilisés) ne sont pas supportés par ICU. Si c'est le cas pour l'encodage de la base de données, les enregistrements de collations ICU dans `pg_collation` sont ignorés. Tenter d'en utiliser un renverra une erreur du type « collation "de-x-icu" for encoding "WIN874" does not exist ».

24.2.2.3. Créer de nouveaux objets de collation

Si les collations standards et prédéfinies ne sont pas suffisantes, les utilisateurs peuvent créer leur propres objets de collation en utilisant la commande SQL CREATE COLLATION.

Les collations standards et prédéfinies sont dans le schéma `pg_catalog`, comme tous les objets prédéfinis. Les collations définies par les utilisateurs doivent être créées dans des schémas utilisateurs. Ceci assure qu'elles seront sauvegardées par `pg_dump`.

24.2.2.3.1. Collations libc

Les nouvelles collations libc peuvent être créées ainsi :

```
CREATE COLLATION german (provider = libc, locale = 'de_DE');
```

Les valeurs exactes de la clause `locale` acceptées par cette commande dépendent du système d'exploitation. Sur les systèmes Unix, la commande `locale -a` affichera une liste.

Comme les collations libc prédéfinies incluent déjà toutes les collations définies dans le système d'exploitation au moment de l'initialisation de l'instance, il n'est pas souvent nécessaire d'en créer de nouvelles. Des raisons possibles sont l'utilisation d'un autre système de nommage (auquel cas, voir aussi Section 24.2.2.3.3), ou une mise à jour du système d'exploitation pour fournir les définitions des nouvelles locales (auquel cas, voir aussi `pg_import_system_collations()`).

24.2.2.3.2. Collations ICU

ICU permet la personnalisation des collations en dehors de l'ensemble pré-enregistré langue/pays, préchargé par `initdb`. Les utilisateurs sont encouragés à définir leur propres objets de collation pour adapter le comportement du tri à leurs besoins. Voir https://unicode-org.github.io/icu/userguide/locale/ et https://unicode-org.github.io/icu/userguide/collation/api.html pour plus d'informations sur le nommage des locales ICU. L'ensemble des noms et attributs acceptables dépend de la version ICU précise.

Voici quelques exemples :

```
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale =
'de-u-co-phonebk');
CREATE COLLATION "de-u-co-phonebk-x-icu" (provider = icu, locale =
'de@collation=phonebook');
```

Collationnement allemand avec le type de collationnement d'un carnet d'adresses

Le premier exemple sélectionne la locale ICU en utilisant une « étiquette de langue » d'après BCP 47. Le deuxième exemple utilise la syntaxe de locale traditionnelle spécifique à ICU. La préférence va au premier style, mais il n'est pas supporté par les anciennes versions d'ICU.

Notez que, dans l'environnement SQL, vous pouvez nommer les objets de collation comme vous le voulez. Dans cet exemple, nous suivons le style de nommage que les collations prédéfinies utilisent, qui suit lui-même BCP 47, mais ce n'est pas obligatoire pour les collations définies par l'utilisateur.

```
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale =
'und-u-co-emoji');
CREATE COLLATION "und-u-co-emoji-x-icu" (provider = icu, locale =
'@collation=emoji');
```

Collationnement racine avec un type de collationnement Emoji, d'après l'Unicode Technical Standard #51

Observez comment, dans le système de nommage traditionnel des locales ICU, la locale racine est sélectionnée par une chaîne vide.

```
CREATE COLLATION latinlast (provider = icu, locale = 'en-u-kr-grek-
latn');
CREATE COLLATION latinlast (provider = icu, locale =
'en@colReorder=grek-latn');
```

Trie les lettres grecques avant les lettres latines. (Par défaut, le latin est avant le grec.)

```
CREATE COLLATION upperfirst (provider = icu, locale = 'en-u-kf-
upper');
CREATE COLLATION upperfirst (provider = icu, locale =
'en@colCaseFirst=upper');
```

Trie les lettres majuscules avant les lettres minuscules. (La valeur par défaut est les minuscules avant).

```
CREATE COLLATION special (provider = icu, locale = 'en-u-kf-upper-
kr-grek-latn');
CREATE   COLLATION   special   (provider   =   icu,   locale   =
'en@colCaseFirst=upper;colReorder=grek-latn');
```

Combine ces deux options.

```
CREATE COLLATION numeric (provider = icu, locale = 'en-u-kn-true');
CREATE   COLLATION   numeric   (provider   =   icu,   locale   =
'en@colNumeric=yes');
```

Ordre numérique, trie les séquences de chiffres par leur valeur numérique. Par exemple : `A-21` < `A-123` (aussi connu sous le nom de tri naturel).

Voir Unicode Technical Standard #35[1] et BCP 47[2] pour les détails. La liste des types de collationnement possibles (sous-ensemble `co`) peut être trouvée dans le dépôt CLDR[3].

Notez qu'alors que, si ce système permet la création de collations qui « ignorent la casse » ou « ignorent les accents » ou quelque chose de similaire (en utilisant la clé `ks`), pour que ces collations se comportent d'une manière réellement insensible à la casse ou aux accents, elles doivent aussi être déclarées comme non *déterministes* dans `CREATE COLLATION`; voir Section 24.2.2.4. Sinon, les chaînes qui sont considérées comme égales d'après la collation, mais non équivalentes au niveau des octets seront triées en fonction des valeurs de leurs octets.

> **Note**
>
> De par sa conception, ICU acceptera pratiquement n'importe quelle chaîne comme nom de locale, et la fera correspondre à la locale la plus proche qu'il peut fournir, en utilisant la procédure de « fallback » décrite dans sa documentation. De ce fait, il n'y aura pas de retour direct si la spécification d'une collation est composée avec des fonctionnalités que l'installation ICU donnée ne supporte pas. Il est donc recommandé de créer des cas de tests au niveau applicatif pour vérifier que les définitions de collations satisfont les besoins.

24.2.2.3.3. Copier les collations

La commande CREATE COLLATION peut également être utilisée pour créer une nouvelle collation depuis une collation existante, ce qui peut être utile pour utiliser une collation indépendante du système dans les applications, de créer des noms compatibles, ou d'utiliser une collation fournie par ICU avec un nom plus lisible. Par exemple :

```
CREATE COLLATION german FROM "de_DE";
CREATE COLLATION french FROM "fr-x-icu";
```

24.2.2.4. Collations non déterministes

Une collation est soit *déterministe*, soit *non déterministe*. Une collation déterministe utilise des comparaisons déterministes, ce qui signifie qu'elle considère les chaînes de caractères comme égales seulement si elles sont constituées des mêmes séquences d'octets. Les comparaisons non déterministes peuvent considérer des chaînes comme égales même si elles sont constituées d'octets différents. Les cas d'usage typiques comprennent des comparaisons insensibles à la casse, ou insensibles aux accents, de même que la comparaison de chaînes dans différentes formes normales Unicode. C'est au fournisseur de collation d'implémenter de telles comparaisons insensibles ; le drapeau donnant

[1] https://www.unicode.org/reports/tr35/tr35-collation.html
[2] https://www.rfc-editor.org/info/bcp47
[3] https://github.com/unicode-org/cldr/blob/master/common/bcp47/collation.xml

la collation comme déterministe n'indique que si les chaînes égales doivent être départagées ou non par une comparaison au niveau de l'octet. Voir aussi Unicode Technical Standard 10[4] pour plus d'information sur la terminologie.

Pour créer une collation non déterministe, spécifiez la propriété `deterministic = false` à `CREATE COLLATION`, par exemple:

```
CREATE COLLATION ndcoll (provider = icu, locale = 'und',
 deterministic = false);
```

Cet exemple utiliserait la collation Unicode standard d'une manière non déterministe. En particulier, elle permettrait à des chaînes exprimées dans différentes formes normales d'être comparées correctement. D'autres exemples plus intéressants utilisent les facultés de personnalisation d'ICU expliquées ci-dessus. Par exemple :

```
CREATE COLLATION case_insensitive (provider = icu, locale = 'und-u-
ks-level2', deterministic = false);
CREATE COLLATION ignore_accents (provider = icu, locale = 'und-u-
ks-level1-kc-true', deterministic = false);
```

Toutes les collations standards et prédéfinies sont déterministes, et toutes les collations définies par les utilisateurs sont déterministes par défaut. Bien que les collations non déterministes aient un comportement plus « correct », particulièrement considérant la puissance d'Unicode et ses nombreux cas spécifiques, elles ont aussi quelques inconvénients. D'abord, leur utilisation génère une pénalité de performance. Il est à noter en particulier qu'un B-tree ne peut pas utiliser la déduplication dans les index utilisant une collation non déterministe. Par ailleurs, certaines opérations ne sont pas possibles avec des collations non déterministes, comme la recherche par motif. Par conséquent, elles ne devraient être utilisées que dans les cas où elles sont spécifiquement désirables.

> **Astuce**
>
> Pour gérer du texte sous différentes formes de normalisation Unicode, il est possible d'utiliser les fonctions ou expressions `normalize` et `is normalized` pour prétraiter ou contrôler les chaînes de caractères, au lieu d'utiliser des collations non déterministes. Il y a des avantages et inconvénients différents à chaque approche.

24.3. Support des jeux de caractères

PostgreSQL permet d'insérer du texte dans différents jeux de caractères (aussi appelés encodages), soit mono-octet comme série ISO 8859, soit multi-octets comme EUC (Extended Unix Code), UTF-8 ou le codage interne Mule. Tous les jeux de caractères supportés peuvent être utilisés de façon transparente par les clients, mais certains ne sont pas supportés par le serveur (c'est-à-dire comme encodage côté serveur). Le jeu de caractères par défaut est choisi pendant l'initialisation de l'instance avec `initdb`. Ce choix peut être surchargé à la création de la base. Il est donc possible de disposer de bases utilisant chacune un jeu de caractères différent.

Cependant, une importante restriction existe : le jeu de caractère de chaque base doit être compatible avec les variables `LC_CTYPE` (classification des caractères) et `LC_COLLATE` (ordre de tri des chaînes) de cette même base. Pour les locales `C` ou `POSIX`, tous les jeux de caractères sont autorisés, mais pour d'autres locales provenant de la libc, il n'y a qu'un seul jeu de caractères qui fonctionne correctement. (Néanmoins, sur Windows, l'encodage UTF-8 peut être utilisé avec toute locale.) Si le

[4] https://www.unicode.org/reports/tr10

support d'ICU est configuré, les locales fournies par ICU peuvent être utilisées avec la plupart des encodages côté serveur.

24.3.1. Jeux de caractères supportés

Le Tableau 24.1 présente les jeux de caractères utilisables avec PostgreSQL.

Tableau 24.1. Jeux de caractères de PostgreSQL

Nom	Description	Langue	Serveur ?	ICU ?	Octets/ Caractère	Alias
BIG5	Big Five	Chinois traditionnel	Non	Non	1–2	WIN950, Windows950
EUC_CN	Code-CN Unix étendu	Chinois simplifié	Oui	Oui	1–3	
EUC_JP	Code-JP Unix étendu	Japonais	Oui	Oui	1–3	
EUC_JIS_2004	Code-JP Unix étendu, JIS X 0213	Japonais	Oui	Non	1–3	
EUC_KR	Code-KR Unix étendu	Coréen	Oui	Oui	1–3	
EUC_TW	Code-TW Unix étendu	Chinois traditionnel, taïwanais	Oui	Oui	1–4	
GB18030	Standard national	Chinois	Non	Non	1–4	
GBK	Standard national étendu	Chinois simplifié	Non	Non	1–2	WIN936, Windows936
ISO_8859_5	ISO 8859-5, ECMA 113	Latin/ Cyrillique	Oui	Oui	1	
ISO_8859_6	ISO 8859-6, ECMA 114	Latin/Arabe	Oui	Oui	1	
ISO_8859_7	ISO 8859-7, ECMA 118	Latin/Grec	Oui	Oui	1	
ISO_8859_8	ISO 8859-8, ECMA 121	Latin/Hébreu	Oui	Oui	1	
JOHAB	JOHAB	Coréen (Hangul)	Non	Non	1-3	
KOI8R	KOI8-R	Cyrillique (Russie)	Oui	Oui	1	KOI8
KOI8U	KOI8-U	Cyrillique (Ukraine)	Oui	Oui	1	
LATIN1	ISO 8859-1, ECMA 94	Europe de l'ouest	Oui	Oui	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	Europe centrale	Oui	Oui	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	Europe du sud	Oui	Oui	1	ISO88593

Nom	Description	Langue	Serveur ?	ICU ?	Octets/Caractère	Alias
LATIN4	ISO 8859-4, ECMA 94	Europe du nord	Oui	Oui	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	Turque	Oui	Oui	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Nordique	Oui	Oui	1	ISO885910
LATIN7	ISO 8859-13	Baltique	Oui	Oui	1	ISO885913
LATIN8	ISO 8859-14	Celtique	Oui	Oui	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 avec l'Euro et les accents	Oui	Oui	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Roumain	Oui	Non	1	ISO885916
MULE_INTERNAL	Code interne Mule	Emacs multi-langues	Oui	Non	1–4	
SJIS	Shift JIS	Japonais	Non	Non	1–2	Mskanji, ShiftJIS, WIN932, Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	Japonais	Non	Non	1–2	
SQL_ASCII	non spécifié (voir le texte)	*toutes*	Oui	Non	1	
UHC	Code unifié Hangul	Coréen	Non	Non	1–2	WIN949, Windows949
UTF8	Unicode, 8-bit	*toutes*	Oui	Oui	1–4	Unicode
WIN866	Windows CP866	Cyrillique	Oui	Oui	1	ALT
WIN874	Windows CP874	Thaï	Oui	Non	1	
WIN1250	Windows CP1250	Europe centrale	Oui	Oui	1	
WIN1251	Windows CP1251	Cyrillique	Oui	Oui	1	WIN
WIN1252	Windows CP1252	Europe de l'ouest	Oui	Oui	1	
WIN1253	Windows CP1253	Grec	Oui	Oui	1	
WIN1254	Windows CP1254	Turque	Oui	Oui	1	
WIN1255	Windows CP1255	Hébreux	Oui	Oui	1	
WIN1256	Windows CP1256	Arabe	Oui	Oui	1	
WIN1257	Windows CP1257	Baltique	Oui	Oui	1	

Nom	Description	Langue	Serveur ?	ICU ?	Octets/Caractère	Alias
WIN1258	Windows CP1258	Vietnamien	Oui	Oui	1	ABC, TCVN, TCVN5712, VSCII

Toutes les API clients ne supportent pas tous les jeux de caractères de la liste. Le pilote JDBC de PostgreSQL, par exemple, ne supporte pas MULE_INTERNAL, LATIN6, LATIN8 et LATIN10.

SQL_ASCII se comporte de façon considérablement différente des autres valeurs. Quand le jeu de caractères du serveur est SQL_ASCII, le serveur interprète les valeurs des octets 0–127 suivant le standard ASCII alors que les valeurs d'octets 128–255 sont considérées comme des caractères non interprétés. Aucune conversion de codage n'est effectuée avec SQL_ASCII. De ce fait, cette valeur ne déclare pas tant un encodage spécifique que l'ignorance de l'encodage. Dans la plupart des cas, si des données non ASCII doivent être traitées, il est déconseillé d'utiliser la valeur SQL_ASCII car PostgreSQL est alors incapable de convertir ou de valider les caractères non ASCII.

24.3.2. Choisir le jeu de caractères

initdb définit le jeu de caractères par défaut (encodage) pour une instance. Par exemple,

```
initdb -E EUC_JP
```

paramètre le jeu de caractères à EUC_JP (Extended Unix Code for Japanese). Vous préférerez peut-être l'option longue --encoding à la place de -E. Si ni -E ni --encoding n'est donné, initdb tente de déterminer l'encodage approprié en se basant sur la locale indiquée ou celle par défaut.

Vous pouvez indiquer un encodage autre que celui par défaut lors de la création de la base de données, à condition que l'encodage soit compatible avec la locale sélectionnée :

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-
ctype=ko_KR.euckr korean
```

Cela crée une base de données appelée korean qui utilise le jeu de caractères EUC_KR, et la locale ko_KR. Un autre moyen de réaliser cela est d'utiliser la commande SQL suivante :

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR'
 LC_COLLATE='ko_KR.euckr' LC_CTYPE='ko_KR.euckr'
 TEMPLATE=template0;
```

Notez que les commandes ci-dessus précisent de copier la base de données template0. Lors de la copie d'une autre base, les paramètres d'encodage et de locale repris de la base de données source ne peuvent pas être modifiés, car cela pourrait corrompre les données. Pour plus d'informations, voir Section 23.3.

L'encodage de la base de données est conservé dans le catalogue système pg_database. Cela est visible à l'aide de l'option -l ou de la commande \l de psql.

```
$ psql -l
                                        List of databases
   Name    |  Owner   | Encoding  |  Collation  |    Ctype    |
      Access Privileges
-----------+----------+-----------+-------------+-------------
+-----------------------------------
 clocaledb | hlinnaka | SQL_ASCII | C           | C           |
 englishdb | hlinnaka | UTF8      | en_GB.UTF8  | en_GB.UTF8  |
 japanese  | hlinnaka | UTF8      | ja_JP.UTF8  | ja_JP.UTF8  |
 korean    | hlinnaka | EUC_KR    | ko_KR.euckr | ko_KR.euckr |
```

```
 postgres  | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 template0 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 {=c/hlinnaka,hlinnaka=CTc/hlinnaka}
 template1 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 {=c/hlinnaka,hlinnaka=CTc/hlinnaka}
(7 rows)
```

Important

Sur la plupart des systèmes d'exploitation modernes, PostgreSQL peut déterminer le jeu de caractères impliqué par la variable `LC_CTYPE`, et s'assurer que l'encodage correspondant de la base de données est utilisé. Sur les systèmes plus anciens, il est de la responsabilité de l'utilisateur de s'assurer que l'encodage attendu par la locale est bien utilisé. Une erreur à ce niveau risque fort de conduire à un comportement étrange des opérations liées à la locale, tel le tri.

PostgreSQL autorise les superutilisateurs à créer des bases de données avec le jeu de caractère `SQL_ASCII` même lorsque la variable `LC_CTYPE` n'est pas à `C` ou `POSIX`. Comme indiqué plus haut, `SQL_ASCII` n'impose aucun encodage particulier aux données stockées en base, ce qui rend ce paramétrage vulnérable aux comportements erratiques lors d'opérations liées à la locale. Cette combinaison de paramètres est dépréciée et pourrait un jour être interdite.

24.3.3. Conversion automatique d'encodage entre serveur et client

PostgreSQL automatise la conversion de jeux de caractères entre client et serveur pour un grand nombre de combinaisons de jeux de caractères (la Section 24.3.4 montre lesquels).

Pour activer la conversion automatique des jeux de caractères, il faut indiquer à PostgreSQL le jeu de caractères (encodage) souhaité côté client. Il y a plusieurs façons de le faire :

- en utilisant la commande `\encoding` dans psql. `\encoding` permet de changer l'encodage client à la volée. Par exemple, pour changer le codage en `SJIS`, taper :

  ```
  \encoding SJIS
  ```

- la libpq (Section 34.11) a des fonctions de contrôle de l'encodage client ;

- en utilisant `SET client_encoding TO`. L'encodage client peut être fixé avec la commande SQL suivante :

  ```
  SET CLIENT_ENCODING TO 'valeur';
  ```

  La syntaxe SQL plus standard `SET NAMES` peut également être utilisée pour cela :

  ```
  SET NAMES 'valeur';
  ```

  Pour connaître l'encodage client courant :

  ```
  SHOW client_encoding;
  ```

  Pour revenir à l'encodage par défaut :

  ```
  RESET client_encoding;
  ```

- en utilisant `PGCLIENTENCODING`. Si la variable d'environnement `PGCLIENTENCODING` est définie dans l'environnement client, cet encodage client est automatiquement sélectionné lors de

l'établissement d'une connexion au serveur (cette variable peut être surchargée à l'aide de toute autre méthode décrite ci-dessus) ;

- en utilisant la variable de configuration client_encoding. Si la variable `client_encoding` est définie, l'encodage client est automatiquement sélectionné lors de l'établissement d'une connexion au serveur (cette variable peut être surchargée à l'aide de toute autre méthode décrite ci-dessus).

Si la conversion d'un caractère particulier n'est pas possible -- par exemple dans le cas d'encodages `EUC_JP` pour le serveur et `LATIN1` pour le client, et que certains caractères japonais renvoyés n'ont pas de représentation en `LATIN1` -- une erreur est remontée.

Si l'encodage client est défini en tant que `SQL_ASCII`, la conversion de l'encodage est désactivée quelque soit celui du serveur. (Toutefois, si l'encodage serveur n'est pas `SQL_ASCII`, le serveur testera toujours que les données en entrée sont valides pour son encodage ; le résultat final est identique à avoir comme encodage client celui du serveur). Comme pour le serveur, `SQL_ASCII` est déconseillé, sauf à ne travailler qu'avec des données ASCII.

24.3.4. Conversions disponibles entre jeux de caractères

PostgreSQL permet les conversions entre deux jeux de caractères quand une fonction de conversion est listée dans le catalogue système `pg_conversion`. PostgreSQL est fourni avec des conversions prédéfinies, comme résumé dans Tableau 24.2 et détaillé dans Tableau 24.3. On peut créer une nouvelle conversion en utilisant la commande SQL CREATE CONVERSION. (Pour être utilisée pour les conversions client/serveur automatiques, une conversion doit être marquée en tant que « default » pour son couple de jeux de caractères).

Tableau 24.2. Conversions client/serveur fournies de base pour les jeux de caractères

Jeux de caractères sur le serveur	Jeux de caractères disponibles sur le client
`BIG5`	*non supporté comme encodage serveur*
`EUC_CN`	*EUC_CN*, `MULE_INTERNAL`, `UTF8`
`EUC_JP`	*EUC_JP*, `MULE_INTERNAL`, `SJIS`, `UTF8`
`EUC_JIS_2004`	*EUC_JIS_2004*, `SHIFT_JIS_2004`, `UTF8`
`EUC_KR`	*EUC_KR*, `MULE_INTERNAL`, `UTF8`
`EUC_TW`	*EUC_TW*, `BIG5`, `MULE_INTERNAL`, `UTF8`
`GB18030`	*non supporté comme encodage serveur*
`GBK`	*non supporté comme encodage serveur*
`ISO_8859_5`	*ISO_8859_5*, `KOI8R`, `MULE_INTERNAL`, `UTF8`, `WIN866`, `WIN1251`
`ISO_8859_6`	*ISO_8859_6*, `UTF8`
`ISO_8859_7`	*ISO_8859_7*, `UTF8`
`ISO_8859_8`	*ISO_8859_8*, `UTF8`
`JOHAB`	*non supporté comme encodage serveur*
`KOI8R`	*KOI8R*, `ISO_8859_5`, `MULE_INTERNAL`, `UTF8`, `WIN866`, `WIN1251`
`KOI8U`	*KOI8U*, `UTF8`
`LATIN1`	*LATIN1*, `MULE_INTERNAL`, `UTF8`
`LATIN2`	*LATIN2*, `MULE_INTERNAL`, `UTF8`, `WIN1250`
`LATIN3`	*LATIN3*, `MULE_INTERNAL`, `UTF8`

Jeux de caractères sur le serveur	Jeux de caractères disponibles sur le client
LATIN4	*LATIN4*, MULE_INTERNAL, UTF8
LATIN5	*LATIN5*, UTF8
LATIN6	*LATIN6*, UTF8
LATIN7	*LATIN7*, UTF8
LATIN8	*LATIN8*, UTF8
LATIN9	*LATIN9*, UTF8
LATIN10	*LATIN10*, UTF8
MULE_INTERNAL	*MULE_INTERNAL*, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8R, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	*non supporté comme encodage serveur*
SHIFT_JIS_2004	*non supporté comme encodage serveur*
SQL_ASCII	*tous (aucune conversion ne sera effectuée)*
UHC	*non supporté comme encodage serveur*
UTF8	*tous les encodages supportés*
WIN866	*WIN866*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN1251
WIN874	*WIN874*, UTF8
WIN1250	*WIN1250*, LATIN2, MULE_INTERNAL, UTF8
WIN1251	*WIN1251*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866
WIN1252	*WIN1252*, UTF8
WIN1253	*WIN1253*, UTF8
WIN1254	*WIN1254*, UTF8
WIN1255	*WIN1255*, UTF8
WIN1256	*WIN1256*, UTF8
WIN1257	*WIN1257*, UTF8
WIN1258	*WIN1258*, UTF8

Tableau 24.3. Toutes les conversions fournies de base entre jeux de caractères

Nom de la conversion [a]	Encodage source	Encodage de destination
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5

Nom de la conversion [a]	Encodage source	Encodage de destination
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1

Nom de la conversion [a]	Encodage source	Encodage de destination
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
windows_1258_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_windows_1258	UTF8	WIN1258

Nom de la conversion [a]	Encodage source	Encodage de destination
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
utf8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
utf8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis_2004	EUC_JIS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis_2004	SHIFT_JIS_2004	EUC_JIS_2004

[a] Les noms des conversions suivent un standard de nommage : le nom officiel de l'encodage source, avec tous les caractères non alphanumériques remplacés par un souligné, suivi de _to_, suivi du nom de l'encodage de destination transformé de la même manière. Par conséquent, ces noms diffèrent parfois des noms d'encodage montrés dans le Tableau 24.1.

24.3.5. Pour aller plus loin

Il existe quelques sources intéressantes pour commencer à découvrir les différents systèmes d'encodage.

CJKV Information Processing: Chinese, Japanese, Korean & Vietnamese Computing

Contient des explications détaillées sur `EUC_JP`, `EUC_CN`, `EUC_KR`, `EUC_TW`.

https://www.unicode.org/

Le site web du consortium Unicode.

RFC 3629[5]

UTF-8 (8-bit UCS/Unicode Transformation Format) est défini là.

[5] https://datatracker.ietf.org/doc/html/rfc3629

Chapitre 25. Planifier les tâches de maintenance

PostgreSQL, comme tout SGBD, requiert que certaines tâches soient réalisées de façon régulière pour atteindre des performances optimales. Ces tâches sont *requises*, mais elles sont répétitives par nature et peuvent être facilement automatisées en utilisant des outils standard comme les scripts cron ou le Task Scheduler de Windows. La responsabilité de la mise en place de ces scripts et du contrôle de leur bon fonctionnement relève de l'administrateur de la base.

Une opération de maintenance évidente est la sauvegarde régulière des données. Sans une sauvegarde récente, il est impossible de restaurer après un dommage grave (perte d'un disque, incendie, table supprimée par erreur, etc.). Les mécanismes de sauvegarde et restauration disponibles dans PostgreSQL sont détaillés dans le Chapitre 26.

L'autre tâche primordiale est la réalisation périodique d'un « vacuum », c'est-à-dire un « nettoyage » de la base de données. Cette opération est détaillée dans la Section 25.1. De la même façon, la mise à jour des statistiques utilisées par l'optimiseur de requêtes sera discutée dans Section 25.1.3.

La gestion des fichiers de traces mérite aussi une attention régulière. Cela est détaillé dans la Section 25.3.

check_postgres[1] est disponible pour surveiller la santé des bases de données et pour rapporter des conditions inhabituelles. check_postgres s'intègre bien avec Nagios et MRTG, mais il peut aussi fonctionner de façon autonome.

PostgreSQL demande peu de maintenance par rapport à d'autres SGBD. Néanmoins, un suivi vigilant de ces tâches participera beaucoup à rendre le système productif et agréable à utiliser.

25.1. Nettoyages réguliers

Les bases de données de PostgreSQL nécessitent des opérations de maintenance périodiques, connues sous le nom de *VACUUM*. Pour de nombreuses installations, il est suffisant de laisser travailler le *démon autovacuum*, qui est décrit dans Section 25.1.6. En fonction des cas, les paramètres de cet outil peuvent être optimisés pour obtenir de meilleurs résultats. Certains administrateurs de bases de données voudront suppléer ou remplacer les activités du démon avec une gestion manuelle des commandes `VACUUM`, qui seront typiquement exécutées suivant un planning par des scripts cron ou par le Task Scheduler. Pour configurer une gestion manuelle et correcte du VACUUM, il est essentiel de bien comprendre les quelques sous-sections suivantes. Les administrateurs qui se basent sur l'autovacuum peuvent toujours lire ces sections pour les aider à comprendre et à ajuster l'autovacuum.

25.1.1. Bases du VACUUM

La commande `VACUUM` de PostgreSQL doit traiter chaque table régulièrement pour plusieurs raisons :

1. pour récupérer ou ré-utiliser l'espace disque occupé par les lignes supprimées ou mises à jour ;
2. pour mettre à jour les statistiques utilisées par l'optimiseur de PostgreSQL ;
3. Pour mettre à jour la carte de visibilité qui accélère les parcours d'index seuls ;
4. pour prévenir la perte des données les plus anciennes à cause d'un *cycle de l'identifiant de transaction (XID)* ou d'un *cycle de l'identifiant multixact*.

Chacune de ces raisons impose de réaliser des opérations `VACUUM` de différentes fréquences et portées, comme expliqué dans les sous-sections suivantes.

[1] https://bucardo.org/check_postgres/

Il existe deux variantes de la commande VACUUM : le VACUUM standard et le VACUUM FULL. VACUUM FULL peut récupérer davantage d'espace disque mais s'exécute beaucoup plus lentement. Par ailleurs, la forme standard de VACUUM peut s'exécuter en parallèle avec les opérations de production des bases. Des commandes comme SELECT, INSERT, UPDATE et DELETE continuent de fonctionner de façon normale, mais la définition d'une table ne peut être modifiée avec des commandes telles que ALTER TABLE pendant le VACUUM. VACUUM FULL nécessite un verrou de type ACCESS EXCLUSIVE sur la table sur laquelle il travaille, et ne peut donc pas être exécuté en parallèle avec une autre activité sur la table. Par conséquent, en règle générale, les administrateurs doivent s'efforcer d'utiliser la commande standard VACUUM et éviter VACUUM FULL.

VACUUM génère un nombre important d'entrées/sorties, ce qui peut entraîner de mauvaises performances pour les autres sessions actives. Des paramètres de configuration peuvent être ajustés pour réduire l'impact d'un nettoyage en arrière plan sur les performances -- voir Section 20.4.4.

25.1.2. Récupérer l'espace disque

Avec PostgreSQL, les versions périmées des lignes ne sont pas immédiatement supprimées après une commande UPDATE ou DELETE. Cette approche est nécessaire pour la cohérence des accès concurrents (MVCC, voir le Chapitre 13) : la version de la ligne ne doit pas être supprimée tant qu'elle est susceptible d'être lue par une autre transaction. Mais en fin de compte, une ligne qui est plus vieille que toutes les transactions en cours n'est plus utile du tout. La place qu'elle utilise doit être rendue pour être réutilisée par d'autres lignes afin d'éviter un accroissement constant, sans limite, du volume occupé sur le disque. Cela se fait en exécutant un VACUUM.

La forme standard de VACUUM élimine les versions d'enregistrements morts dans les tables et les index, et marque l'espace comme réutilisable. Néanmoins, il ne rend pas cet espace au système d'exploitation, sauf dans le cas spécial où des pages à la fin d'une table deviennent totalement vides et qu'un verrou exclusif sur la table peut être obtenu aisément. Par opposition, VACUUM FULL compacte activement les tables en écrivant une nouvelle version complète du fichier de la table, sans espace vide. Ceci réduit la taille de la table mais peut prendre beaucoup de temps. Cela requiert aussi un espace disque supplémentaire pour la nouvelle copie de la table jusqu'à la fin de l'opération.

Le but habituel d'un nettoyage régulier est de lancer des VACUUM standard suffisamment souvent pour éviter d'avoir recours à un VACUUM FULL. Le démon autovacuum essaie de fonctionner de cette façon, et n'exécute jamais de VACUUM FULL. Avec cette approche, l'idée directrice n'est pas de maintenir les tables à leur taille minimale, mais de maintenir l'utilisation de l'espace disque à un niveau constant : chaque table occupe l'espace équivalent à sa taille minimale plus la quantité d'espace consommée entre deux nettoyages. Bien que VACUUM FULL puisse être utilisé pour qu'une table retrouve sa taille minimale et rendre l'espace disque au système d'exploitation, cela ne sert pas à grand chose, si cette table recommence à grossir rapidement après. Par conséquent, cette approche s'appuyant sur des commandes VACUUM exécutées à intervalles modérément rapprochés est une meilleure approche que d'exécuter des VACUUM FULL espacés pour des tables mises à jour de façon intensive.

Certains administrateurs préfèrent planifier le passage de VACUUM eux-mêmes, par exemple faire le travail de nuit, quand la charge est faible. La difficulté avec cette stratégie est que si une table a un pic d'activité de mise à jour inattendu, elle peut grossir au point qu'un VACUUM FULL soit vraiment nécessaire pour récupérer l'espace. L'utilisation du démon d'autovacuum minore ce problème, puisque le démon planifie les nettoyages de façon dynamique, en réponse à l'activité de mise à jour. Il est peu raisonnable de désactiver totalement le démon, sauf si l'activité de la base est extrêmement prévisible. Un compromis possible est de régler les paramètres du démon afin qu'il ne réagisse qu'à une activité exceptionnellement lourde de mise à jour, de sorte à seulement éviter de perdre totalement le contrôle de la volumétrie, tout en laissant les VACUUM planifiés faire le gros du travail quand la charge est normale.

Pour ceux qui n'utilisent pas autovacuum, une approche typique alternative est de planifier un VACUUM sur la base complète une fois par jour lorsque l'utilisation n'est pas forte, avec en plus des opérations de VACUUM plus fréquentes pour les tables très impactées par des mises à jour, de la façon adéquate. (Certaines installations avec énormément de mises à jour peuvent exécuter des nettoyages toutes les

quelques minutes.) Lorsqu'il y a plusieurs bases dans une instance, il faut penser à exécuter un VACUUM sur chacune d'elles ; le programme vacuumdb peut se révéler utile dans ce contexte.

> **Astuce**
>
> Le VACUUM standard peut ne pas suffire quand une table contient un grand nombre d'enregistrements morts comme conséquence d'une mise à jour ou d'une suppression massive. Dans ce cas, s'il est nécessaire de récupérer l'espace disque gaspillé, plusieurs commandes sont utilisables : VACUUM FULL, CLUSTER ou une des variantes de ALTER TABLE. Ces commandes écrivent une nouvelle copie de la table et reconstruisent ses index. Toutes ces options nécessitent un verrou de type ACCESS EXCLUSIVE. Elles utilisent aussi temporairement un espace disque supplémentaire, approximativement égal à la taille de la table, car les anciennes copies de la table et des index ne peuvent pas être supprimées avant la fin de l'opération.

> **Astuce**
>
> Si le contenu d'une table est supprimé périodiquement, il est préférable d'envisager l'utilisation de TRUNCATE, plutôt que DELETE suivi de VACUUM. TRUNCATE supprime le contenu entier de la table immédiatement sans nécessiter de VACUUM ou VACUUM FULL pour réclamer l'espace disque maintenant inutilisé. L'inconvénient est la violation des sémantiques MVCC strictes.

25.1.3. Maintenir les statistiques de l'optimiseur

L'optimiseur de requêtes de PostgreSQL s'appuie sur des informations statistiques produites à partir du contenu des tables dans l'optique de produire des plans d'exécutions efficaces pour les requêtes. Ces statistiques sont collectées par la commande ANALYZE , qui peut être invoquée seule ou comme option de VACUUM. Il est important d'avoir des statistiques relativement à jour, ce qui permet d'éviter les choix de mauvais plans d'exécution, pénalisant les performances de la base.

Si activé, le démon autovacuum va automatiquement exécuter des commandes ANALYZE à chaque fois que le contenu d'une table aura changé suffisamment. Toutefois, un administrateur peut préférer se fier à des opérations ANALYZE planifiées manuellement, en particulier s'il est connu que l'activité de mise à jour de la table n'a pas d'impact sur les statistiques des colonnes « intéressantes ». Le démon planifie des ANALYZE en fonction du nombre d'enregistrements insérés, mis à jour ou supprimés ; il n'a aucune connaissance du fait que cela générera ou pas des modifications intéressantes au niveau des statistiques.

Les lignes modifiées dans les partitions et les enfants, dans le cadre de l'héritage, ne déclenchent pas d'analyse sur la table parent. Si la table parent est vide ou rarement modifiée, elle pourrait ne jamais être traitée par l'autovacuum, et les statistiques pour l'arbre d'héritage en entier ne seront pas récupérées. Il est nécessaire d'exécuter ANALYZE manuellement sur la table parent pour conserver des statistiques à jour.

À l'instar du nettoyage pour récupérer l'espace, les statistiques doivent être plus souvent collectées pour les tables intensément modifiées que pour celles qui le sont moins. Mais même si la table est très modifiée, il se peut que ces collectes soient inutiles si la distribution probabiliste des données évolue peu. Une règle simple pour décider est de voir comment évoluent les valeurs minimale et maximale des données. Par exemple, une colonne de type `timestamp` qui contient la date de mise à jour de la ligne aura une valeur maximale en continuelle croissance au fur et à mesure des modifications ; une telle colonne nécessitera plus de collectes statistiques qu'une colonne qui contient par exemple les URL des pages accédées sur un site web. La colonne qui contient les URL peut très bien être aussi souvent modifiée mais la distribution probabiliste des données changera certainement moins rapidement.

Il est possible d'exécuter ANALYZE sur des tables spécifiques, voire des colonnes spécifiques ; il y a donc une grande flexibilité pour mettre à jour certaines statistiques plus souvent que les autres en fonction des besoins de l'application. Quoi qu'il en soit, dans la pratique, il est généralement mieux de simplement analyser la base entière car il s'agit d'une opération rapide. ANALYZE utilise un système d'échantillonnage des lignes d'une table, ce qui lui évite de lire chaque ligne.

> **Astuce**
>
> Même si il n'est pas très productif de règler précisément la fréquence de ANALYZE pour chaque colonne, il peut être intéressant d'ajuster le niveau de détail des statistiques collectées pour chaque colonne. Les colonnes très utilisées dans les clauses WHERE et dont la distribution n'est pas uniforme requièrent des histogrammes plus précis que les autres colonnes. Voir ALTER TABLE SET STATISTICS, ou modifier les paramètres par défaut de la base de données en utilisant le paramètre de configuration default_statistics_target.
>
> De plus, par défaut, il existe peu d'informations sur la sélectivité des fonctions. Néanmoins, si vous créez un objet statistique ou un index sur une fonction, des statistiques utiles seront récupérées sur la fonction, ce qui peut grandement améliorer les plans de requêtes qui utilisent l'index.

> **Astuce**
>
> Le démon autovacuum ne lance pas de commandes ANALYZE sur les tables distantes car il n'a aucun moyen de déterminer la fréquence à laquelle la mise à jour des statistiques serait utile. Si vos requêtes ont besoin des statistiques sur les tables distantes pour disposer d'un plan d'exécution correct, une bonne idée serait de lancer manuellement des commandes ANALYZE sur ces tables au moment adéquat.

> **Astuce**
>
> Le démon autovacuum n'exécute pas de commandes ANALYZE pour les tables partitionnées. Les parents seront seulement analysées si le parent lui-même est modifié. Les changements dans les tables enfants ne déclenchent pas d'analyse automatique sur la table parent. Si vos requêtes nécessitent des statistiques sur les tables parents pour être correctement planifiées, il sera nécessaire d'exécuter périodiquement un ANALYZE manuels sur ces tables pour garder des statistiques à jour.

25.1.4. Mettre à jour la carte de visibilité

La commande VACUUM maintient le contenu de la carte de visibilité de chaque table, pour conserver la trace de chaque bloc contenant seulement des lignes connues pour être visibles par toutes les transactions actives (ainsi que les futures transactions, jusqu'à la prochaine modification de la page). Cette carte a deux buts. Tout d'abord, le nettoyage peut ignorer ce type de blocs à la prochaine exécution du VACUUM comme il n'y a rien à nettoyer dans ces blocs.

Ensuite, il permet à PostgreSQL de répondre à certaines requêtes en utilisant seulement l'index, et donc sans faire référence à la table sous-jacente. Comme les index dans PostgreSQL ne contiennent pas d'informations sur la visibilité des lignes, un parcours d'index normal récupère la ligne de la table pour chaque entrée d'index correspondante, ce qui permet de vérifier si la ligne correspondante est bien visible par la transaction en cours. Un parcours d'index seuls vérifie en premier lieu la carte de visibilité. S'il est connu que toutes les lignes du bloc sont visibles, la lecture de la table peut être évitée. Ceci est très utile sur les gros ensembles de données où la carte de visibilité peut éviter des accès

disques. La carte de visibilité est bien plus petite que la table, donc elle peut facilement être mise en cache même quand la table est très grosse.

25.1.5. Éviter les cycles des identifiants de transactions

Le mécanisme de contrôle de concurrence multiversion (MVCC) de PostgreSQL s'appuie sur la possibilité de comparer des identifiants de transactions (XID) : la version d'une ligne dont le XID d'insertion est supérieur au XID de la transaction en cours est « dans le futur » et ne doit pas être visible de la transaction courante. Comme les identifiants ont une taille limitée (32 bits), une instance qui est en activité depuis longtemps (plus de 4 milliards de transactions) pourrait connaître un cycle complet des identifiants de transaction : le XID reviendra à 0 et soudainement les transactions du passé sembleront appartenir au futur - ce qui signifie qu'elles deviennent invisibles. En bref, perte de données totale. (En réalité, les données sont toujours là mais c'est un piètre réconfort puisqu'elles restent inaccessibles.) Pour éviter ceci, il est nécessaire d'exécuter un VACUUM sur chaque table de chaque base au moins une fois tous les deux milliards de transactions.

La raison pour laquelle un nettoyage régulier résout le problème est qu'un `VACUUM` marquera les lignes comme *gelées*, indiquant qu'elles ont été insérées par une transaction suffisamment ancienne pour que les effets de cette transaction soient visibles à coup sûr par toutes les transactions actuelles et futures. Les XID normaux sont comparés sur une base modulo-2^{32}. Cela signifie que pour chaque XID normal, il y en a deux milliards qui sont plus vieux et deux milliards qui sont plus récents. Une autre manière de le dire est que l'ensemble de définition des XID est circulaire et sans limite. De ce fait, pour une ligne créée avec un XID normal donné, cette version de la ligne apparaît comme appartenant au passé pour les deux milliards de transactions qui suivront quelque soit le XID de ces transactions. Si cette version de ligne existe toujours après deux milliards de transactions, elle apparaîtra soudainement comme appartenant au futur. Pour empêcher cela, PostgreSQL réserve un XID spécial, appelé `FrozenTransactionId`, qui ne suit pas les règles normales de comparaison de XID et qui est toujours considéré comme plus ancien que chaque XID normal. Les versions de lignes gelées sont traitées comme si la XID d'insertion était `FrozenTransactionId`, pour qu'elles apparaissent dans le passé pour les autres transactions normales, quelque soit les soucis de cycle d'identifiants de transactions, et donc ces versions de lignes seront valides jusqu'à leur suppression, quelque soit la durée que cela représente.

> **Note**
>
> Dans les versions de PostgreSQL antérieures à la 9.4, le gel était implémenté en remplaçant le XID d'insertion d'une ligne avec `FrozenTransactionId`, qui était visible dans la colonne système `xmin` de la ligne. Les nouvelles versions ajoutent un drapeau, préservant le `xmin` original de la ligne pour une utilisation ultérieure (notamment pour du débugage). Néanmoins, il est toujours possible d'avoir des lignes pour lesquelles `xmin` vaut `FrozenTransactionId` (2) dans les bases de données antérieures à la version 9.4 traitées par pg_upgrade.
>
> De plus, les catalogues systèmes pourraient contenir des lignes avec `xmin` égale à `BootstrapTransactionId` (1), indiquant qu'elles ont été insérées lors de la première phase d'initdb. Comme `FrozenTransactionId`, cet XID spécial est traité comme étant plus ancien que tout autre XID normal.

vacuum_freeze_min_age contrôle l'âge que doit avoir une valeur XID avant que des lignes comportant ce XID ne soient gelées. Augmenter la valeur de ce paramètre peut permettre d'éviter un travail inutile si les lignes à geler vont bientôt être modifiées. Diminuer ce paramètre augmente le nombre de transactions qui peuvent survenir avant un nouveau nettoyage de la table.

`VACUUM` utilise la carte de visibilité pour déterminer les blocs à parcourir sur une table. Habituellement, il ignore les blocs qui n'ont aucune ligne morte même si ces blocs pourraient toujours

avoir des versions de lignes avec des identifiants très anciens de transactions. De ce fait, les VACUUM normaux ne vont pas toujours geler chaque ancienne version de ligne dans la table. Quand cela arrive, VACUUM aura besoin de réaliser un *vacuum agressif*, qui gèlera toutes les valeurs XID et MXID non gelées et éligibles, ceci incluant celles de pages dont les enregistrements d'un bloc sont tous visibles mais pas gelés. En pratique, la plupart des tables requiert un nettoyage agressif périodique. vacuum_freeze_table_age contrôle quand VACUUM se comporte ainsi : les blocs ne contenant que des lignes vivantes mais non gelées sont parcourus si le nombre de transactions exécutées depuis le dernier parcours de ce type est plus grand que vacuum_freeze_table_age moins vacuum_freeze_min_age. Configurer vacuum_freeze_table_age à 0 force VACUUM à toujours utiliser cette stratégie plus agressive.

Le délai maximum où une table peut rester sans nettoyage est de deux millions de transactions moins vacuum_freeze_min_age lors du dernier VACUUM agressif. Si elle devait rester sans nettoyage après cela, des pertes de données pourraient survenir. Pour s'assurer que cela n'arrive pas, autovacuum est appelé sur chaque table qui pourrait contenir des lignes non gelées dont les XID ont un âge plus avancé que le paramètre de configuration autovacuum_freeze_max_age. (Ceci arrivera même si autovacuum est désactivé.)

Ceci implique que, si aucune opération de VACUUM n'est demandée sur une table, l'autovacuum sera automatiquement déclenché une fois toutes les autovacuum_freeze_max_age moins vacuum_freeze_min_age transactions. Pour les tables qui bénéficient régulièrement de l'opération de VACUUM pour réclamer l'espace perdu, ceci a peu d'importance. Néanmoins, pour les tables statiques (en incluant les tables qui ont des insertions mais pas de mises à jour et de suppressions), il n'est pas nécessaire d'exécuter un VACUUM pour récupérer de la place et donc il peut être utile d'essayer de maximiser le délai entre les autovacuums forcés sur de très grosses tables statiques. Évidemment, vous pouvez le faire soit en augmentant autovacuum_freeze_max_age soit en diminuant vacuum_freeze_min_age.

Le maximum efficace pour vacuum_freeze_table_age est 0.95 * autovacuum_freeze_max_age ; un paramétrage plus haut que ça sera limité à ce maximum. Une valeur plus importante que autovacuum_freeze_max_age n'aurait pas de sens car un autovacuum de préservation contre le cycle des identifiants de transactions serait déclenché, et le multiplicateur 0,95 laisse un peu de place pour exécuter un VACUUM manuel avant que cela ne survienne. Comme règle d'or, vacuum_freeze_table_age devrait être configuré à une valeur légèrement inférieure à autovacuum_freeze_max_age, laissant suffisamment d'espace pour qu'un VACUUM planifié régulièrement ou pour qu'un autovacuum déclenché par des activités normales de suppression et de mise à jour puissent être activés pendant ce laps de temps. Le configurer de façon trop proche pourrait déclencher des autovacuum de protection contre la ré-utilisation des identifiants de transactions, même si la table a été récemment l'objet d'un VACUUM pour récupérer l'espace, alors que des valeurs basses amènent à des VACUUM agressifs plus fréquents.

Le seul inconvénient à augmenter autovacuum_freeze_max_age (et vacuum_freeze_table_age avec elle) est que les sous- répertoires pg_xact et pg_commit_ts de l'instance prendront plus de place car ils doivent stocker le statut et l'horodatage (si track_commit_timestamp est activé) du COMMIT pour toutes les transactions depuis autovacuum_freeze_max_age. L'état de COMMIT utilise deux bits par transaction, donc si autovacuum_freeze_max_age et vacuum_freeze_table_age ont une valeur maximale permise de deux milliards, pg_xact peut grossir jusqu'à la moitié d'un Go et pg_commit_ts jusqu'à 20 Go. Si c'est rien comparé à votre taille de base totale, configurer autovacuum_freeze_max_age à son maximum permis est recommandé. Sinon, le configurer suivant ce que vous voulez comme stockage maximum dans pg_xact et dans pg_commit_ts. (La valeur par défaut, 200 millions de transactions, se traduit en à peu près 50 Mo de stockage dans pg_xact et à peu près 2 Go de stockage pour pg_commit_ts.)

Un inconvénient causé par la diminution de vacuum_freeze_min_age est que cela pourrait faire que VACUUM travaille sans raison : geler une version de ligne est une perte de temps si la ligne est modifiée rapidement après (ce qui fait qu'elle obtiendra un nouveau XID). Donc ce paramètre doit être suffisamment important pour que les lignes ne soient pas gelées jusqu'à ce qu'il soit pratiquement certain qu'elles ne seront plus modifiées.

Pour tracer l'âge des plus anciens XID non gelés de la base, VACUUM stocke les statistiques sur XID dans les tables systèmes pg_class et pg_database. En particulier, la colonne relfrozenxid de la ligne pg_class d'une table contient le XID final du gel qui a été utilisé par le dernier VACUUM agressif pour cette table. Il est garanti que tous les XID plus anciens que ce XID ont été remplacés par FrozenXID pour cette table. Toutes les lignes insérées par des transactions dont le XID est plus ancien que ce XID sont garanties d'avoir été gelées. En particulier, la colonne relfrozenxid de la ligne de pg_class pour une table contient les XID non gelés les plus anciens restants à la fin du plus récent VACUUM qui a avancé avec succès relfrozenxid (typiquement le VACUUM agressif le plus récent). De façon similaire, la colonne datfrozenxid de la ligne pg_database de la base est une limite inférieure des XID non gelés apparaissant dans cette base -- c'est tout simplement le minimum des valeurs relfrozenxid par table dans cette base. Pour examiner cette information, le plus simple est d'exécuter des requêtes comme :

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');

SELECT datname, age(datfrozenxid) FROM pg_database;
```

La colonne age mesure le nombre de transactions à partir du XID final vers le XID de transaction en cours.

> **Astuce**
>
> Quand la clause VERBOSE de la commande VACUUM est utilisée, VACUUM affiche différentes statistiques sur la table. Ceci inclut des informations sur l'avancement de relfrozenxid et de relminmxid. Les mêmes détails apparaissent dans le journal applicatif du serveur si les actions de l'log_autovacuum_min_duration) sont tracées sur une opération VACUUM exécutée par autovacuum.

Habituellement, VACUUM parcourt seulement les blocs qui ont été modifiés depuis le dernier nettoyage mais relfrozenxid peut seulement être avancé quand tous les blocs d'une table pouvant contenir des XID gelés sont parcourus. Ceci survient quand relfrozenxid a plus de vacuum_freeze_table_age transactions antérieures, quand l'option FREEZE de VACUUM est utilisée ou quand tous les blocs qui ne sont pas encore gelés nécessitent un nettoyage pour supprimer les versions de lignes mortes. Quand VACUUM parcourt chaque bloc d'une table qui n'est pas déjà entièrement gelé, il doit configurer age (relfrozenxid) à une valeur un peu au-dessus de la configuration utilisée pour vacuum_freeze_min_age (plus par le nombre de transactions démarrées depuis le lancement de VACUUM). VACUUM initialisera relfrozenxid au XID le plus ancien qui reste dans la table, donc il est possible que la valeur finale soit bien plus récente que strictement nécessaire. Si aucun VACUUM avec avancement de relfrozenxid n'est lancé sur la table, une fois arrivé à autovacuum_freeze_max_age, un autovacuum est forcé sur la table.

Si, pour une certaine raison, l'autovacuum échoue à effacer les anciens XID d'une table, le système commencera à émettre des messages d'avertissement comme ceci quand les plus anciens XID de la base atteignent les 40 millions de transactions à partir du point de réinitialisation :

```
WARNING:  database "mydb" must be vacuumed within 39985967
 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM
 in that database.
```

(Une commande VACUUM manuelle devrait résoudre le problème, comme suggéré par le message HINT ; mais notez que la commande VACUUM doit être exécutée par un super-utilisateur, sinon elle

échouera à mettre à jour les catalogues, ce qui l'empêchera d'avancer le champ `datfrozenxid` de la base.) Si ces avertissements sont ignorés, le système refusera d'assigner de nouveaux identifiants de transactions dès qu'il n'en restera que trois millions avant la réinitialisation :

```
ERROR:  database is not accepting commands to avoid wraparound data
 loss in database "mydb"
HINT:  Stop the postmaster and vacuum that database in single-user
 mode.
```

Dans ce cas, toute transaction en cours peut continuer, mais seules des transactions en lecture seule peuvent démarrées. Les opérations qui modifient les enregistrements de la base de données ou qui tronquent des relations échoueront. La commande `VACUUM` peut toujous être exécutée normalement. Contrairement ce qu'indique le message, il n'est ni nécessaire ni dsouhaité d'arrêter le processus postmaster ou d'entrer en mode utilisateur-seul pour restaurer le fonctionnement normal. À la place, suivez ces étapes :

1. Terminez les anciennes transactions préparées. Vous pouvez les trouver en récupérant les lignes de la vue `pg_prepared_xacts` dont `age(transactionid)` est grand. De telles transactions doivent être validées ou annulées.

2. Terminez les transactions ouvertes s'exécutant depuis longtemps. Vous pouvez les trouver en récupérant les lignes de la vue `pg_stat_activity` où `age(backend_xid)` ou `age(backend_xmin)` est grand. De telles transactions doivent être validées ou annulées. Si ce n'est pas possible, les sessions associées doivent être fermées en utilisant la fonction `pg_terminate_backend`.

3. Supprimez les anciens slots de réplication. Utilisez `pg_stat_replication` pour trouver les slots dont `age(xmin)` ou `age(catalog_xmin)` est grand. Dans de nombreux cas, ces slots ont été créées pour de la réplication vers un serveur qui n'existe plus ou qui est arrêté depuis longtemps. Si vous supprimez un slot pour un serveur qui existe toujours ou pourrait de nouveau se connecter à ce slot, cette réplication pourrait nécessiter d'être reconstruite.

4. Exécutez `VACUUM` dans la base cible. Un `VACUUM` sur la base complète est le plus simple ; pour réduire le temps nécessaire pour l'opération, il est aussi possible d'exécuter des commandes `VACUUM` manuelles sur les tables ayant le champ `relminxid` le plus ancien. N'utilisez pas `VACUUM FULL` dans ce cas car il nécessite un identifiant de transaction, ce qui fera échouer l'opération, sauf dans le mode super-utilisateur où il consommera un identifiant de transaction et augmentera de ce fait le risque d'une réutilisation des identifiants de transaction. N'utilisez pas non plus `VACUUM FREEZE` parce qu'il fera plus que la quantité minimal de travail requis pour restaurer le fonctionnement normal.

5. Une fois que le fonctionnement est de nouveau normal, assurez-vous que l'autovacuum est correctement configuré dans la base de données cible pour éviter des problèmes futures.

> **Note**
>
> Dans les version sprécédentes, il était parfois nécessaire d'arrêter le processus postlaster et de lancer la commande `VACUUM` en mode utilisateur-simple. Dans les scénarios habituels, ceci n'est plus nécessaire et doit être éviter autant que possible car cela implique d'arrêter le système. C'est aussi plus risqué car cela désactive les protections sur la réutilisation des identifiants de transaction, prévues pour éviter les pertes de données. La seule raison pour utiliser le mode utilisateur-simple est si vous souhaitez utiliser `TRUNCATE` ou `DROP` sur des tables inutiles pour éviter de lancer `VACUUM` sur celles-ci. La marge de sécurité de trois millions de transactions existe pour permettre à l'administrateur de le faire. Voir la page de référence postgres pour des détails sur l'utilisation du mode utilisateur-simple.

25.1.5.1. Multixacts et cycle

Les *identifiants multixact* sont utilisés pour supporter le verrouillage de lignes par des transactions multiples. Comme l'espace est limité dans un en-tête de ligne pour y stocker des informations, cette information est codée sous la forme d'un « identifiant de transaction multiple », ou ID multixact pour faire court, à chaque fois qu'il y a plus d'une transaction cherchant à verrouiller en parallèle une ligne. Les informations sur les identifiants de transactions inclus dans tout identifiant multixact sont enregistrées séparément dans le sous-répertoire `pg_multixact` et seul l'identifiant multixact apparaît dans le champ `xmax` de l'en-tête de ligne. Comme les identifiants de transactions, les identifiants multi-transactions sont implémentés avec un compteur 32 bits et le stockage correspondant, ce qui nécessite une gestion attentive, un nettoyage du stockage et la gestion du cycle (plus exactement de la ré-utilisation des identifiants). Il existe un espace de stockage séparé qui détient la liste des membres dans chaque multixact, qui utilise aussi un compteur sur 32 bits et qui doit aussi être géré. La fonction système `pg_get_multixact_members()` décrite dans Tableau 9.78 peut être utilisé pour examiner les identifiants de transaction associés avec un identifiant MultiXact.

Quand `VACUUM` parcourt une partie d'une table, il remplacera tout ID multixact qu'il rencontre, plus âgé que vacuum_multixact_freeze_min_age, par une valeur différente, qui peut être la valeur zéro, un identifiant de transaction ou un nouvel identifiant multixact. Pour chaque table, `pg_class.relminmxid` enregistre le plus ancien identifiant multixact possible apparaissant déjà dans un enregistrement de cette table. Si cette valeur est plus ancienne que vacuum_multixact_freeze_table_age, un vacuum agressif est forcé. Comme indiqué dans la section précédente, un vacuum agressif signifie que seuls les blocs connus pour être entièrement gelés sont ignorés. `mxid_age()` peut être utilisé sur `pg_class.relminmxid` pour trouver son âge.

Des `VACUUM` agressifs, quel qu'en soit la cause, sont *garantie* de pouvoir avancer le `relminmxid` de la table. Comme toutes les tables de toutes les bases sont parcourues et que leurs plus anciennes valeurs multixact sont avancées, le stockage sur disque pour les anciens multixacts peut être supprimé.

Comme moyen de sécurité supplémentaire, un VACUUM agressif surviendra pour toute table dont l'âge en identifiant multixact est supérieur à autovacuum_multixact_freeze_max_age. De plus, si le stockage occupé par les membres multixacts dépasse environ 10 Go, des parcours de vacuum agressif surviendront plus fréquemment pour toutes les tables, en commençant par celles qui ont le plus ancien multixact-age. Ces deux types de parcours agressifs de tables surviendront seulement si l'autovacuum est désactivé spécifiquement. L'aire de stockage des membres pourraient grossir jusqu'à environ 20 Go avant d'atteindre la limite de réutilisation des identifiants.

Similaire au cas de l'identifiant de transaction, si l'autovacuum échoue à effacer les anciens identifiants MultiXact d'une table, le système commencera à emetre des messages d'avertissement quand les plus anciens identifiants MultiXact de la base attendront les 40 millions de transactions du point de réutilisation. Et, tout comme dans le cas des identifiants de transactions, si ces messages sont ignorés, le système refusera de générer de nouveaux identifiants MultiXact une fois qu'il en restera moins de trois millions.

Quand les identifiants MultiXact sont épuisés, le retour à une situation normale s'obtient de la même façon que quand des identifiants de transactions sont épuisés. Suivez les mêmes étapes que dans la section précédente mais avec les différences suivantes :

1. Les transactions standards et préparées peuvent être ignorées s'il n'y a pas de risques qu'elles apparaissent dans une MultiXact.

2. Contrairement à la réutilisation des identifiants de transaction, les slots de réplication ne retiennent pas directement le nettoyage des MultiXact ID. Supprimer les slots de réplications inactifs n'est de ce fait pas habituellement pertinent pour résoudre les problèmes de réutilisations des MultiXact ID.

3. L'information MXID n'est pas directement visible dans les vues systèmes telles que `pg_stat_activity` ; néanmoins, rechercher les anciens identifiants de transactions est toujours une bonne façcon de déterminer les transactions pouvant poser des soucis de réutilisation des identifiants de MultiXact.

4. L'épuisement d'identifiants de transactions bloquera toutes les transactions en écriture, tout spécifiquement celles qui imliquent des verrous de ligne nécessitant un identifiant MultiXact.

25.1.6. Le démon auto-vacuum

PostgreSQL dispose d'une fonctionnalité optionnelle mais hautement recommandée appelée *autovacuum*, dont le but est d'automatiser l'exécution des commandes VACUUM et ANALYZE. Quand il est activé, autovacuum vérifie les tables ayant un grand nombre de lignes insérées, mises à jour ou supprimées. Ces vérifications utilisent la fonctionnalité de récupération de statistiques ; du coup, autovacuum ne peut être utilisé que si track_counts est configuré à `true`. Dans la configuration par défaut, l'autovacuum est activé et les paramètres liés sont correctement configurés.

Le « démon autovacuum » est constitué de plusieurs processus. Un processus démon permanent appelé *autovacuum launcher*(autrement dit le lanceur d'autovacuum) est en charge de lancer des processus de traitement (*autovacuum worker*) pour toutes les bases de données. Le lanceur distribue le travail dans le temps en lançant un nouveau processus de traitement sur chaque base de données chaque autovacuum_naptime secondes. (Du coup, si l'installation a *N* bases de données, un nouveau *autovacuum worker* sera lancé tous les `autovacuum_naptime`/*N* secondes.) Un maximum de autovacuum_max_workers processus *autovacuum worker* est autorisé à s'exécuter en même temps. S'il y a plus de `autovacuum_max_workers` bases à traiter, la prochaine base de données sera traitée dès qu'un autre processus de traitement a terminé. Chaque processus de traitement vérifie chaque table de la base de données de connexion et exécute un VACUUM et/ou un ANALYZE suivant les besoins. log_autovacuum_min_duration peut être utilisé pour superviser l'activité des processus autovacuum worker.

Si plusieurs grosses tables deviennent toutes éligibles à un VACUUM dans un court espace de temps, tous les processus de traitement pourraient avoir à exécuter des VACUUM sur ces tables pendant un long moment. Ceci pourraient avoir pour résultat que d'autres tables et d'autres bases de données ne puissent pas être traitées tant qu'un processus de traitement ne sera pas disponible. Il n'y a pas de limite sur le nombre de processus de traitement sur une seule base, mais ils essaient d'éviter de répéter le travail qui a déjà été fait par d'autres. Notez que le nombre de processus de traitement en cours d'exécution n'est pas décompté des limites max_connections et superuser_reserved_connections.

Les tables dont la valeur de `relfrozenxid` est supérieure à autovacuum_freeze_max_age font toujours l'objet d'un VACUUM (cela s'applique aussi aux tables dont le *freeze max age* a été modifié avec les paramètres de stockage ; voir plus bas). Sinon, si le nombre de lignes obsolètes depuis le dernier VACUUM dépasse une « limite de vacuum », la table bénéficie d'un VACUUM. La limite est définie ainsi :

```
limite du vacuum = limite de base du vacuum + facteur d'échelle du
 vacuum * nombre de lignes
```

où la limite de base du vacuum est autovacuum_vacuum_threshold, le facteur d'échelle du vacuum est autovacuum_vacuum_scale_factor et le nombre de lignes est `pg_class.reltuples`.

La table est aussi traitée si le nombre de lignes insérées depuis le dernier VACUUM a dépassé la limite d'insertion définie d'après cette formule :

```
limite insertion vacuum = limite insertion base vacuum + facteur
 échelle insertion vacuum * nombre de lignes
```

où la limite d'insertion de base du VACUUM correspond au paramètre autovacuum_vacuum_insert_threshold, et le facteur d'échelle d'insertion du VACUUM correspond au paramètre autovacuum_vacuum_insert_scale_factor. De tels VACUUM permettent de marquer des portions de la table comme *tout visible* et permettent aussi de geler les lignes, ce qui peut réduire le travail requis par les VACUUM suivant. Pour les tables recevant des opérations INSERT mais aucune ou très peu d'opérations UPDATE/DELETE, il peut être bénéfique de diminuer la valeur du paramètre

autovacuum_freeze_min_age pour la table car cela permet le gel des lignes par des VACUUM rapides. Le nombre de lignes obsolètes et le nombre de lignes insérées sont obtenus via le système de statistiques cumulatives ; ce décompte moyennement précis est mis à jour par chaque opération UPDATE, DELETE et INSERT. (C'est seulement moyennement précis car certaines informations pourraient être perdues en cas de fortes charges.) Si la valeur du champ relfrozenxid pour la table est supérieure à vacuum_freeze_table_age, un VACUUM agressif est réalisé pour geler les anciennes lignes et avancer la valeur de relfrozenxid, sinon seules les blocs qui ont été modifiés depuis le dernier VACUUM sont parcourues par l'opération de VACUUM.

Pour ANALYZE, une condition similaire est utilisée : la limite, définie comme

```
limite du analyze = limite de base du
    analyze + facteur d'échelle du analyze * nombre de
    lignes
```

est comparée au nombre de lignes insérées, mises à jour et supprimées depuis le dernier ANALYZE.

Les tables partitionnées ne contiennent pas de lignes et ne sont donc pas traitées par l'autovacuum. (L'autovacuum traite les partitions comme toute autre table.)) Malheureusement, ceci signifie que l'autovacuum n'exécute pas d'ANALYZE sur les tables partitionnées, et ceci peut occasionner des plans non optimaux pour les requêtes qui référencent les statistiques de la table partitionnée. Vous pouvez contourner ce problème en exécutant manuellement ANALYZE sur les tables partitionnées quand elles sont peuplées la première fois, et à chaque fois que la distribution des données changent dans les partitions.

Les tables temporaires ne peuvent pas être accédées par l'autovacuum. De ce fait, les opérations appropriées de VACUUM et d'ANALYZE devraient être traitées par des commandes SQL de session.

Les limites et facteurs d'échelle par défaut sont pris dans postgresql.conf, mais il est possible de les surcharger table par table (ainsi que beaucoup d'autres paramètres de contrôle de l'autovacuum) ; voir Paramètres de stockage pour plus d'informations. Si une configuration a été modifiée via les paramètres de stockage d'une table, cette valeur est utilisée lors du traitement de cette table. Dans le cas contraire, les paramètres globaux sont utilisés. Voir Section 20.10 pour plus de détails sur les paramètres globaux.

Quand plusieurs autovacuum workers travaillent, les paramètres de délai de coût de l'autovacuum (voir Section 20.4.4) sont « réparties » parmi tous les processus pour que l'impact total en entrée/sortie sur le système soit identique quelque soit le nombre de processus en cours d'exécution. Néanmoins, tout autovacuum worker traitant des tables et dont les paramètres de stockage autovacuum_vacuum_cost_delay ou autovacuum_vacuum_cost_limit ont été configurés spécifiquement ne sont pas considérés dans l'algorithme de répartition.

Les processus de traitement de l'autovacuum ne bloquent généralement pas les autres commandes. Si un processus tente d'acquérir un verrou qui entre en conflit avec le verrou SHARE UPDATE EXCLUSIVE détenu par autovacuum, la demande de verrou interrompra l'autovacuum. Pour les modes de verrou entrant en conflit, voir Tableau 13.2. Néanmoins, si l'autovacuum s'exécute pour empêcher une réutilisation des identifiants de transaction (c'est-à-dire si le texte de la requête de l'autovacuum dans la vue pg_stat_activity se termine avec le texte (to prevent wraparound)), l'autovacuum n'est pas automatiquement interrompu.

> **Avertissement**
>
> Exécuter régulièrement des commandes qui acquièrent des verrous en conflit avec un verrou SHARE UPDATE EXCLUSIVE (par exemple ANALYZE) peut gêner fortement le traitement réalisé par l'autovacuum.

25.2. Ré-indexation régulière

Dans certains cas, il est intéressant de reconstruire périodiquement les index par la commande REINDEX ou par une série d'étapes individuelles de reconstruction.

Les blocs devenus complètement vides d'un index B-tree sont réclamées pour leur ré-utilisation. Mais, il existe toujours une possibilité d'utilisation peu efficace de l'espace : si, sur un bloc, seulement quelques clés d'index ont été supprimés, le bloc reste alloué. En conséquence, si seulement quelques clés sont supprimées, vous devrez vous attendre à ce que l'espace disque soit très mal utilisé. Dans de tels cas, la réindexation périodique est recommandée.

Le potentiel d'inflation des index qui ne sont pas des index B-tree n'a pas été particulièrement analysé. Surveiller périodiquement la taille physique de ces index est une bonne idée.

De plus, pour les index B-tree, un index tout juste construit est légèrement plus rapide qu'un index qui a été mis à jour plusieurs fois parce que les blocs adjacents logiquement sont habituellement aussi physiquement adjacents dans un index nouvellement créé(cette considération ne s'applique pas aux index non B-tree). Il pourrait être intéressant de ré-indexer périodiquement simplement pour améliorer la vitesse d'accès.

REINDEX peut être utilisé en toute sécurité et très facilement dans tous les cas. Cette commande nécessite un verrou `ACCESS EXCLUSIVE` par défaut. De ce fait, il est souvent préférable de l'exécuter avec l'option `CONCURRENTLY` qui requiert seulement un verrou `SHARE UPDATE EXCLUSIVE`.

25.3. Maintenance du fichier de traces

Sauvegarder les journaux de trace du serveur de bases de données dans un fichier plutôt que dans `/dev/null` est une bonne idée. Les journaux sont d'une utilité incomparable lorsqu'arrive le moment où des problèmes surviennent.

> **Note**
>
> Les traces peuvent contenir des informations sensibles et doivent donc être protégées, peu importe où et comment ils sont enregistrées, ou la destination où ils sont envoyés. Par exemple, certaines requêtes DDL pourraient contenir des mots de passe en clair ou d'autres informations d'authentification. Les requêtes tracées au niveau `ERROR` pourraient afficher le source SQL des applications et pourraient aussi contenir certaines parties des lignes de données. Enregistrer les données, événements et les informations relatives est la fonction souhaitée de cette fonctionnalité, donc ce n'est ni une information affichée sans raison ni un bug. Merci de vous assurer que les traces du serveur sont visibles uniquement aux personnes dûment autorisées.

Néanmoins, les journaux ont tendance à être volumineux (tout spécialement à des niveaux de débogage importants) et vous ne voulez pas les sauvegarder indéfiniment. Vous avez besoin de faire une « rotation » des journaux pour que les nouveaux journaux soient commencés et que les anciens soient supprimés après une période de temps raisonnable.

Si vous redirigez simplement la sortie stderr du processus `postgres` dans un fichier, vous aurez un journal des traces mais la seule façon de le tronquer sera d'arrêter et de relancer le serveur. Ceci peut convenir si vous utilisez PostgreSQL dans un environnement de développement mais peu de serveurs de production trouveraient ce comportement acceptable.

Une meilleure approche est d'envoyer la sortie stderr du serveur dans un programme de rotation de journaux. Il existe un programme interne de rotation que vous pouvez utiliser en configurant le paramètre `logging_collector` à `true` dans `postgresql.conf`. Les paramètres de contrôle pour ce programme sont décrits dans Section 20.8.1. Vous pouvez aussi utiliser cette approche pour capturer les données des journaux applicatifs dans un format CSV (valeurs séparées par des virgules) lisible par une machine

Sinon, vous pourriez préférer utiliser un programme externe de rotation de journaux si vous en utilisez déjà un avec d'autres serveurs. Par exemple, l'outil rotatelogs inclus dans la distribution Apache peut être utilisé avec PostgreSQL. Pour cela, envoyez via un tube la sortie stderr du serveur dans le programme désiré. Si vous lancez le serveur avec `pg_ctl`, alors stderr est déjà directement renvoyé dans stdout, donc vous avez juste besoin d'ajouter la commande via un tube, par exemple :

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

Vous pouvez combiner ces approches en configurant logrotate pour qu'il récupère les fichiers de trace produits par le collecteur de trace de PostgreSQL. Dans ce cas, le collecteur définit les noms et emplacement des fichiers de trace, alors que logrotate les archive périodiquement. Lors d'une rotation de fichier de traces, logrotate doit s'assurer que l'application envoie les nouvelles traces dans le nouveau fichier. Ceci se fait habituellement avec un script `postrotate` qui envoie un signal `SIGHUP` à l'application, qui ouvre ensuite de nouveau le fichier de traces. Dans PostgreSQL, vous pouvez exécuter `pg_ctl` avec l'option `logrotate`. Quand le serveur reçoit cette commande, soit le serveur bascule vers un nouveau fichier de trace soit il ré-ouvre le fichier existant, suivant la configuration des traces (voir Section 20.8.1).

> **Note**
>
> Lors de l'utilisation de noms de fichier statique, le serveur pourrait échouer lors de la réouverture du fichier de trace si la limite du nombre maximum de fichiers ouvert est atteint ou qu'un dépassement de la table de fichiers survient. Dans ce cas, les traces sont envoyées à l'ancien fichier de traces jusqu'à la réussite de la rotation du fichier de trace. Si logrotate est configuré pour compresser le fichier de trace et le supprimer, le serveur pourrait perdre les messages tracées pendant cette fenêtre de temps. Pour éviter ce problème, vous pouvez configurer le collecteur de traces pour qu'il affecte dynamiquement les noms des fichiers de trace et utilise un script `prerotate` pour ignorer les fichiers de trace ouverts.

Une autre approche de production pour la gestion des journaux de trace est de les envoyer à syslog et de laisser syslog gérer la rotation des fichiers. Pour cela, initialisez le paramètre de configuration `log_destination` à `syslog`(pour tracer uniquement via syslog) dans `postgresql.conf`. Ensuite, vous pouvez envoyer un signal `SIGHUP` au démon syslog quand vous voulez le forcer à écrire dans un nouveau fichier. Si vous voulez automatiser la rotation des journaux, le programme logrotate peut être configuré pour fonctionner avec les journaux de traces provenant de syslog.

Néanmoins, sur beaucoup de systèmes, syslog n'est pas très fiable, particulièrement avec les messages très gros ; il pourrait tronquer ou supprimer des messages au moment où vous en aurez le plus besoin. De plus, sur Linux, syslog synchronisera tout message sur disque, amenant à des performances assez pauvres. (Vous pouvez utiliser un « `-` » au début du nom de fichier dans le fichier de configuration syslog pour désactiver la synchronisation.)

Notez que toutes les solutions décrites ci-dessus font attention à lancer de nouveaux journaux de traces à des intervalles configurables mais ils ne gèrent pas la suppression des vieux fichiers de traces, qui ne sont probablement plus très utiles. Vous voudrez probablement configurer un script pour supprimer périodiquement les anciens journaux. Une autre possibilité est de configurer le programme de rotation pour que les anciens journaux de traces soient écrasés de façon cyclique.

pgBadger[2] est un projet externe qui analyse les journaux applicatifs d'une façon très poussée. check_postgres[3] fournit des alertes Nagios quand des messages importants apparaissent dans les journaux applicatifs, mais détecte aussi de nombreux autres cas.

[2] https://pgbadger.darold.net/
[3] https://bucardo.org/check_postgres/

Chapitre 26. Sauvegardes et restaurations

Comme tout ce qui contient des données importantes, les bases de données PostgreSQL doivent être sauvegardées régulièrement. Bien que la procédure soit assez simple, il est important de comprendre les techniques et hypothèses sous-jacentes.

Il y a trois approches fondamentalement différentes pour sauvegarder les données de PostgreSQL :

- la sauvegarde SQL ;
- la sauvegarde au niveau du système de fichiers ;
- l'archivage continu.

Chacune a ses avantages et ses inconvénients. Elles sont toutes analysées, chacune leur tour, dans les sections suivantes.

26.1. Sauvegarde SQL

Le principe est de produire un fichier texte de commandes SQL (appelé « fichier dump »), qui, si on le renvoie au serveur, recrée une base de données identique à celle sauvegardée. PostgreSQL propose pour cela le programme utilitaire pg_dump. L'usage basique est :

```
pg_dump base_de_donnees > fichier_sauvegarde
```

pg_dump écrit son résultat sur la sortie standard. Son utilité est expliquée plus loin. Bien que la commande ci-dessus crée un fichier texte, pg_dump peut créer des fichiers dans d'autres formats qui permettent le parallélisme et un contrôle plus fin de la restauration des objets.

pg_dump est un programme client PostgreSQL classique (mais plutôt intelligent). Cela signifie que la sauvegarde peut être effectuée depuis n'importe quel ordinateur ayant accès à la base. Mais pg_dump n'a pas de droits spéciaux. En particulier, il doit avoir un accès en lecture à toutes les tables que vous voulez sauvegarder, donc pour sauvegarder une base complète, vous devez pratiquement toujours utiliser un superutilisateur. si vous n'avez pas les droits suffisants pour sauvegarder la base entière, vous pouvez toujours sauvegarder les parties pour lesquels vous avez le droit d'accès en utilisant des options telles que `-n` *`schéma`* et `-t` *`table`*.)

Pour préciser le serveur de bases de données que pg_dump doit contacter, on utilise les options de ligne de commande `-h` *`serveur`* et `-p` *`port`*. Le serveur par défaut est le serveur local ou celui indiqué par la variable d'environnement `PGHOST`. De la même façon, le port par défaut est indiqué par la variable d'environnement `PGPORT` ou, en son absence, par la valeur par défaut précisée à la compilation. Le serveur a normalement reçu les mêmes valeurs par défaut à la compilation.

Comme tout programme client PostgreSQL, pg_dump se connecte par défaut avec l'utilisateur de base de données de même nom que l'utilisateur système courant. L'utilisation de l'option `-U` ou de la variable d'environnement `PGUSER` permettent de modifier le comportement par défaut. Les connexions de pg_dump sont soumises aux mécanismes normaux d'authentification des programmes clients (décrits dans le Chapitre 21).

Un des gros avantages de pg_dump sur les autres méthodes de sauvegarde décrites après est que la sortie de pg_dump peut être généralement re-chargée dans des versions plus récentes de PostgreSQL, alors que les sauvegardes au niveau fichier et les sauvegardes continues sont toutes les deux très spécifiques à la version du serveur. pg_dump est aussi la seule méthode qui fonctionnera lors du transfert d'une base de données vers une machine d'une architecture différente (comme par exemple d'un serveur 32 bits à un serveur 64 bits).

Les sauvegardes créées par pg_dump sont cohérentes, ce qui signifie que la sauvegarde représente une image de la base de données au moment où commence l'exécution de pg_dump. pg_dump ne bloque pas les autres opérations sur la base lorsqu'il fonctionne (sauf celles qui nécessitent un verrou exclusif, comme la plupart des formes d'`ALTER TABLE`.)

26.1.1. Restaurer la sauvegarde

Les fichiers texte créés par pg_dump peuvent être lus par le programme psql en utilisant sa configuration par défaut. La syntaxe générale d'une commande de restauration est :

```
psql -X base_de_donnees < fichier_sauvegarde
```

où *`fichier_sauvegarde`* est le fichier en sortie de la commande pg_dump. La base de données *`base_de_donnees`* n'est pas créée par cette commande. Elle doit être créée à partir de `template0` avant d'exécuter psql (par exemple avec `createdb -T template0 base_de_donnees`). Pour s'assurer que psql s'exécute avec sa configuration par défaut, utilisez l'option `-X` (`--no-psqlrc`). psql propose des options similaires à celles de pg_dump pour indiquer le serveur de bases de données sur lequel se connecter et le nom d'utilisateur à utiliser. La page de référence de psql donne plus d'informations.

Les sauvegardes binaires doivent être restaurées en utilisant l'outil pg_restore.

Tous les utilisateurs possédant des objets ou ayant certains droits sur les objets de la base sauvegardée doivent exister préalablement à la restauration de la sauvegarde. S'ils n'existent pas, la restauration échoue pour la création des objets dont ils sont propriétaires ou sur lesquels ils ont des droits (quelque fois, cela est souhaitable mais ce n'est pas toujours le cas).

Par défaut, le script psql continue de s'exécuter après la détection d'une erreur SQL. Vous pouvez exécuter psql avec la variable `ON_ERROR_STOP` configurée pour modifier ce comportement. psql quitte alors avec un code d'erreur 3 si une erreur SQL survient :

```
psql -X --set ON_ERROR_STOP=on base_de_données < fichier_sauvegarde
```

Dans tous les cas, une sauvegarde partiellement restaurée est obtenue. Si cela n'est pas souhaitable, il est possible d'indiquer que la sauvegarde complète doit être restaurée au cours d'une transaction unique. De ce fait, soit la restauration est validée dans son ensemble, soit elle est entièrement annulée. Ce mode est choisi en passant l'option `-1` ou `--single-transaction` en ligne de commande à psql. Dans ce mode, la plus petite erreur peut annuler une restauration en cours depuis plusieurs heures. Néanmoins, c'est probablement préférable au nettoyage manuel d'une base rendu complexe par une sauvegarde partiellement restaurée.

La capacité de pg_dump et psql à écrire et à lire dans des tubes permet de sauvegarder une base de données directement d'un serveur sur un autre. Par exemple :

```
pg_dump -h serveur1 base_de_donnees | psql -X -
h serveur2 base_de_donnees
```

> **Important**
>
> Les fichiers de sauvegarde produits par pg_dump sont relatifs à `template0`. Cela signifie que chaque langage, procédure, etc. ajouté à `template1` est aussi sauvegardé par pg_dump. En conséquence, si une base `template1` modifiée est utilisée lors de la restauration, il faut créer la base vide à partir de `template0`, comme dans l'exemple ci-dessus.

Après la restauration d'une sauvegarde, il est conseillé d'exécuter `ANALYZE` sur chaque base de données pour que l'optimiseur de requêtes dispose de statistiques utiles ; voir Section 25.1.3 et

Section 25.1.6 pour plus d'informations. Pour plus de conseils sur le chargement efficace de grosses quantités de données dans PostgreSQL, on peut se référer à la Section 14.4.

26.1.2. Utilisation de pg_dumpall

pg_dump ne sauvegarde qu'une seule base à la fois, et ne sauvegarde pas les informations relatives aux rôles et *tablespaces* (parce que ceux-ci portent sur l'ensemble des bases du cluster, et non sur une base particulière). Pour permettre une sauvegarde aisée de tout le contenu d'une instance, le programme pg_dumpall est fourni. pg_dumpall sauvegarde toutes les bases de données d'une instance PostgreSQL et préserve les données communes à l'instance, telles que les rôles et tablespaces. L'utilisation basique de cette commande est :

```
pg_dumpall > fichier_sauvegarde
```

Le fichier de sauvegarde résultant peut être restauré avec psql :

```
psql -X -f fichier_sauvegarde postgres
```

(N'importe quelle base de données peut être utilisée pour la connexion mais si le rechargement est exécuté sur une instance vide, il est préférable d'utiliser `postgres`.) Il faut obligatoirement avoir le profil superutilisateur pour restaurer une sauvegarde faite avec pg_dumpall, afin de pouvoir restaurer les informations sur les rôles et les tablespaces. Si les *tablespaces* sont utilisés, il faut s'assurer que leurs chemins sauvegardés sont appropriés à la nouvelle installation.

pg_dumpall fonctionne en émettant des commandes pour recréer les rôles, les *tablespaces* et les bases vides, puis en invoquant pg_dump pour chaque base de données. Cela signifie que, bien que chaque base de données est cohérente en interne, les images des différentes bases de données ne sont pas synchronisées.

Les données globales à l'instance peuvent être sauvegardées seules en utilisant l'option `--globals-only` de pg_dumpall. Ceci est nécessaire pour sauvegarder entièrement l'instance si la commande pg_dump est utilisée pour sauvegarder les bases individuelles.

26.1.3. Gérer les grosses bases de données

Certains systèmes d'exploitation ont des limites sur la taille maximale des fichiers qui peuvent poser problème lors de la création de gros fichiers de sauvegarde avec pg_dump. Heureusement, pg_dump peut écrire sur la sortie standard, donc vous pouvez utiliser les outils Unix standards pour contourner ce problème potentiel. Il existe plusieurs autres méthodes :

Compresser le fichier de sauvegarde. Tout programme de compression habituel est utilisable. Par exemple gzip :

```
pg_dump base_de_donnees | gzip > nom_fichier.gz
```

Pour restaurer :

```
gunzip -c nom_fichier.gz | psql base_de_donnees
```

ou

```
cat nom_fichier.gz | gunzip | psql base_de_donnees
```

Couper le fichier avec `split`. La commande `split` permet de découper le fichier en fichiers plus petits, de taille acceptable par le système de fichiers sous-jacent. Par exemple, pour faire des morceaux de 2 Go :

```
pg_dump base_de_donnees | split -b 2G - nom_fichier
```

Pour restaurer :

```
cat nom_fichier* | psql base_de_donnees
```

Si vous utilisez GNU split, il est possible de l'utiliser avec gzip :

```
pg_dump dbname | split -b 2G --filter='gzip > $FILE.gz'
```

Le résultat peut être restauré en utilisant `zcat`.

Utilisation du format de sauvegarde personnalisé de pg_dump. Si PostgreSQL est installé sur un système où la bibliothèque de compression zlib est disponible, le format de sauvegarde personnalisé peut être utilisé pour compresser les données à la volée. Pour les bases de données volumineuses, cela produit un fichier de sauvegarde d'une taille comparable à celle du fichier produit par `gzip`, avec l'avantage supplémentaire de permettre de restaurer des tables sélectivement. La commande qui suit sauvegarde une base de données en utilisant ce format de sauvegarde :

```
pg_dump -Fc base_de_donnees > nom_fichier
```

Le format de sauvegarde personnalisé ne produit pas un script utilisable par psql. Ce script doit être restauré avec pg_restore, par exemple :

```
pg_restore -d nom_base nom_fichier
```

Voir les pages de référence de pg_dump et pg_restore pour plus de détails.

Pour les très grosses bases de données, il peut être nécessaire de combiner `split` avec une des deux autres approches.

Utiliser la fonctionnalité de sauvegarde en parallèle de pg_dump. Pour accélérer la sauvegarde d'une grosse base de données, vous pouvez utiliser le mode parallélisé de pg_dump. Cela sauvegardera plusieurs tables à la fois. Vous pouvez contrôler le degré de parallélisme avec le paramètre `-j`. Les sauvegardes en parallèle n'acceptent que le format répertoire.

```
pg_dump -j num -F d -f sortie.dir nom_base
```

Vous pouvez utiliser `pg_restore -j` pour restaurer une sauvegarde en parallèle. Ceci fonctionnera pour n'importe quelle archive, qu'elle soit dans le mode personnalisé ou répertoire. Elle n'a pas besoin d'avoir été créée avec le mode parallélisé de `pg_dump`.

26.2. Sauvegarde de niveau système de fichiers

Une autre stratégie de sauvegarde consiste à copier les fichiers utilisés par PostgreSQL pour le stockage des données. Dans la Section 19.2, l'emplacement de ces fichiers est précisé. N'importe quelle méthode de sauvegarde peut être utilisée, par exemple :

```
tar -cf sauvegarde.tar /usr/local/pgsql/data
```

Cependant, deux restrictions rendent cette méthode peu pratique ou en tout cas inférieure à la méthode pg_dump.

1. Le serveur de base de données *doit* être arrêté pour obtenir une sauvegarde utilisable. Toutes les demi-mesures, comme la suppression des connexions, ne fonctionnent *pas* (principalement parce que `tar` et les outils similaires ne font pas une image atomique de l'état du système de fichiers, mais aussi à cause du cache disque du serveur). Les informations concernant la façon d'arrêter le serveur PostgreSQL se trouvent dans la Section 19.5.

Le serveur doit également être arrêté avant de restaurer les données.

2. Quiconque s'est aventuré dans les détails de l'organisation de la base de données peut être tenté de ne sauvegarder et restaurer que certaines tables ou bases de données particulières. Ce n'est *pas* utilisable sans les fichiers de validation `pg_xact/*` qui contiennent l'état de la validation de chaque transaction. Un fichier de table n'est utilisable qu'avec cette information. Bien entendu, il est impossible de ne restaurer qu'une table et les données `pg_xact` associées car cela rendrait toutes les autres tables du serveur inutilisables. Les sauvegardes du système de fichiers fonctionnent, de ce fait, uniquement pour les sauvegardes et restaurations complètes d'une instance.

Une autre approche à la sauvegarde du système de fichiers consiste à réaliser une « image cohérente » (*consistent snapshot*) du répertoire des données. Il faut pour cela que le système de fichiers supporte cette fonctionnalité (et que l'administrateur fasse confiance à cette technologie). La procédure typique consiste à réaliser une « image gelée »(*frozen snapshot*) du volume contenant la base de données et ensuite de copier entièrement le répertoire de données(pas seulement quelques parties, voir plus haut) de l'image sur un périphérique de sauvegarde, puis de libérer l'image gelée. Ceci fonctionne même si le serveur de la base de données est en cours d'exécution. Néanmoins, une telle sauvegarde copie les fichiers de la base de données dans un état où le serveur n'est pas correctement arrêté ; du coup, au lancement du serveur à partir des données sauvegardées, PostgreSQL peut penser que le serveur s'est arrêté brutalement et rejouer les journaux de transactions. Ce n'est pas un problème, mais il faut en être conscient (et s'assurer d'inclure les journaux de transactions dans la sauvegarde). Vous pouvez réaliser un `CHECKPOINT` avant de prendre la sauvegarde pour réduire le temps de restauration.

Si la base de données est répartie sur plusieurs systèmes de fichiers, il n'est peut-être pas possible d'obtenir des images gelées exactement simultanées de tous les disques. Si les fichiers de données et les journaux de transactions se trouvent sur des disques différents, par exemple, ou si les tablespaces sont sur des systèmes de fichiers différents, une sauvegarde par image n'est probablement pas utilisable parce que ces dernières *doivent* être simultanées. La documentation du système de fichiers doit être étudiée avec attention avant de faire confiance à la technique d'images cohérentes dans de telles situations.

S'il n'est pas possible d'obtenir des images simultanées, il est toujours possible d'éteindre le serveur de bases de données suffisamment longtemps pour établir toutes les images gelées. Une autre possibilité est de faire une sauvegarde de la base en archivage continu (Section 26.3.2) parce que ces sauvegardes ne sont pas sensibles aux modifications des fichiers pendant la sauvegarde. Cela n'impose d'activer l'archivage en continu que pendant la période de sauvegarde ; la restauration est faite en utilisant la restauration d'archive en ligne(Section 26.3.4).

Une autre option consiste à utiliser rsync pour réaliser une sauvegarde du système de fichiers. Ceci se fait tout d'abord en lançant rsync alors que le serveur de bases de données est en cours d'exécution, puis en arrêtant le serveur juste assez longtemps pour lancer `rsync --checksum` une deuxième fois (`--checksum` est nécessaire car `rsync` n'a une granularité que d'une seconde quand il teste par horodatage de modification). Le deuxième rsync est beaucoup plus rapide que le premier car il a relativement peu de données à transférer et le résultat final est cohérent, le serveur étant arrêté. Cette méthode permet de réaliser une sauvegarde du système de fichiers avec un arrêt minimal.

Une sauvegarde des fichiers de données va être généralement plus volumineuse qu'une sauvegarde SQL. (pg_dump ne sauvegarde pas le contenu des index, mais la commande pour les recréer). Cependant, une sauvegarde des fichiers de données peut être plus rapide.

26.3. Archivage continu et récupération d'un instantané (PITR)

PostgreSQL maintient en permanence des journaux de transactions(*Write Ahead Log* ou WAL) dans le sous-répertoire `pg_wal/` du répertoire de données de l'instance. Ces journaux enregistrent chaque modification effectuée sur les fichiers de données des bases. Ils existent principalement pour se

prémunir des suites d'un arrêt brutal : si le système s'arrête brutalement, la base de données peut être restaurée dans un état cohérent en « rejouant » les entrées des journaux enregistrées depuis le dernier checkpoint. Néanmoins, l'existence de ces journaux rend possible l'utilisation d'une troisième stratégie pour la sauvegarde des bases de données : la combinaison d'une sauvegarde de niveau système de fichiers avec la sauvegarde des journaux de transactions. Si la restauration est nécessaire, la sauvegarde des fichiers est restaurée, puis les journaux de transactions sauvegardés sont rejoués pour amener la sauvegarde jusqu'à la date actuelle. Cette approche est plus complexe à administrer que toutes les autres approches mais elle apporte des bénéfices importants :

- Il n'est pas nécessaire de disposer d'une sauvegarde des fichiers parfaitement cohérente comme point de départ. Toute incohérence dans la sauvegarde est corrigée par la ré-exécution des journaux (ceci n'est pas significativement différent de ce qu'il se passe lors d'un rejeu après un arrêt brutal). La fonctionnalité d'image du système de fichiers n'est alors pas nécessaire, tar ou tout autre outil d'archivage est suffisant.

- Puisqu'une longue séquence de journaux de transactions peut être assemblée pour être rejoués, une sauvegarde continue est obtenue en continuant simplement à archiver les journaux de transactions. C'est particulièrement intéressant pour les grosses bases de données dont une sauvegarde complète fréquente est difficilement réalisable.

- Les entrées des journaux de transactions ne doivent pas obligatoirement être rejouées intégralement. La ré-exécution peut être stoppée en tout point, tout en garantissant une image cohérente de la base de données telle qu'elle était à ce moment-là. Ainsi, cette technique autorise la *restauration d'un instantané* (PITR) : il est possible de restaurer l'état de la base de données telle qu'elle était en tout point dans le temps depuis la dernière sauvegarde de base.

- Si la série de journaux de transactions est fournie en continu à une autre machine chargée avec le même fichier de sauvegarde de base, on obtient un système « de reprise intermédiaire » (*warm standby*) : à tout moment, la deuxième machine peut être montée et disposer d'une copie quasi-complète de la base de données.

> **Note**
>
> pg_dump et pg_dumpall ne font pas de sauvegardes au niveau système de fichiers. Ce type de sauvegarde est qualifié de *logique* et ne contient pas suffisamment d'informations pour permettre le rejeu des journaux de transactions.

Tout comme la technique de sauvegarde standard du système de fichiers, cette méthode ne supporte que la restauration d'une instance de bases de données complet, pas d'un sous-ensemble. De plus, un espace d'archivage important est requis : la sauvegarde de la base peut être volumineuse et un système très utilisé engendre un trafic de journaux de transactions à archiver de plusieurs Mo. Malgré tout, c'est la technique de sauvegarde préférée dans de nombreuses situations où une haute fiabilité est requise.

Une restauration fructueuse à partir de l'archivage continu (aussi appelé « sauvegarde à chaud » par certains vendeurs de SGBD) nécessite une séquence ininterrompue de journaux de transactions archivés qui s'étend au moins jusqu'au point de départ de la sauvegarde. Pour commencer, il faut configurer et tester la procédure d'archivage des journaux de transactions *avant* d'effectuer la première sauvegarde de base. C'est pourquoi la suite du document commence par présenter les mécanismes d'archivage des journaux de transactions.

26.3.1. Configurer l'archivage des journaux de transactions

Au sens abstrait, un système PostgreSQL fonctionnel produit une séquence infinie d'enregistrements de journaux de transactions. Le système divise physiquement cette séquence en *fichiers de segment* WAL de 16 Mo chacun (en général, mais cette taille peut être modifiée lors de l'exécution d'initdb). Les

fichiers segment reçoivent des noms numériques pour refléter leur position dans la séquence abstraite des journaux de transactions. Lorsque le système n'utilise pas l'archivage des journaux de transactions, il ne crée que quelques fichiers segments, qu'il « recycle » en renommant les fichiers devenus inutiles. Un fichier segment dont le contenu précède le dernier checkpoint est supposé inutile et peut être recyclé.

Lors de l'archivage des journaux de transactions, le contenu de chaque fichier segment doit être capturé dès qu'il est rempli pour sauvegarder les données ailleurs avant son recyclage. En fonction de l'application et du matériel disponible, « sauvegarder les données ailleurs » peut se faire de plusieurs façons : les fichiers segment peuvent être copiés dans un répertoire NFS monté sur une autre machine, être écrits sur une cartouche (après s'être assuré qu'il existe un moyen d'identifier le nom d'origine de chaque fichier) ou être groupés pour gravure sur un CD, ou toute autre chose. Pour fournir autant de flexibilité que possible à l'administrateur de la base de données, PostgreSQL essaie de ne faire aucune supposition sur la façon dont l'archivage est réalisé. À la place, PostgreSQL permet de préciser une commande shell ou une bibliothèque d'archivage à exécuter pour copier le fichier segment complet à l'endroit désiré. Cela peut être aussi simple qu'une commande shell qui utilise `cp` ou cela peut appeler une fonction C complexe -- c'est l'utilisateur qui décide.

Pour activer l'archivage des journaux de transaction, configurez le paramètre de configuration wal_level à `replica` ou supérieur, archive_mode à `on`, et précisez la commande shell à utiliser dans le paramètre de configuration archive_command ou précisez la bibliothèque à utiliser dans le paramètre archive_library de la configuration. En fait, ces paramètres seront toujours placés dans le fichier `postgresql.conf`.

Un moyen simple d'archiver est de configurer le paramètre `archive_library` avec une chaîne vide et de spécifier une commande shell au niveau du paramètre archive_command. Dans cette chaîne, un `%p` est remplacé par le chemin absolu de l'archive alors qu'un `%f` n'est remplacé que par le nom du fichier. (Le nom du chemin est relatif au répertoire de travail du serveur, c'est-à-dire le répertoire des données de l'instance.) `%%` est utilisé pour écrire le caractère `%` dans la commande. La commande la plus simple ressemble à :

```
archive_command = 'test ! -f /mnt/serveur/repertoire_archive/%f &&
 cp %p /mnt/serveur/repertoire_archive/%f'  # Unix
  archive_command = 'copy "%p" "C:\\serveur\\repertoire_archive\\
%f"'  # Windows
```

qui copie les segments WAL archivables dans le répertoire `/mnt/serveur/repertoire_archive`. (Ceci est un exemple, pas une recommandation, et peut ne pas fonctionner sur toutes les plateformes.) Après le remplacement des paramètres `%p` et `%f`, la commande réellement exécutée peut ressembler à :

```
test ! -f /mnt/serveur/repertoire_archive/000000010000000A900000065
 && cp pg_wal/000000010000000A900000065 /mnt/serveur/
repertoire_archive/000000010000000A900000065
```

Une commande similaire est produite pour chaque nouveau fichier à archiver.

La bibliothèque d'archivage est exécutée sous l'identité de l'utilisateur propriétaire du serveur PostgreSQL. La série de journaux de transactions en cours d'archivage contient absolument tout ce qui se trouve dans la base de données, il convient donc de s'assurer que les données archivées sont protégées des autres utilisateurs ; on peut, par exemple, archiver dans un répertoire sur lequel les droits de lecture ne sont positionnés ni pour le groupe ni pour le reste du monde.

Il est important que la commande d'archivage renvoie le statut de sortie zéro si et seulement si elle réussit. En cas de réception d'un code zéro, PostgreSQL supposera que le fichier a été archivé avec succès, et le supprimera ou le recyclera. Néanmoins, un code différent de zéro indique à PostgreSQL que le fichier n'a pas été archivé ; il essaiera de nouveau périodiquement jusqu'à sa réussite.

Une autre façon d'archiver est d'utiliser un module d'archivage personnalisé avec `archive_library`. Comme de tels modules sont écrits en C, créer son propre module nécessite

considérablement plus d'effort que d'écrire une commande shell. Néanmoins, les modules d'archivage peuvent être plus performants que l'archivage via le shell, et ils auront accès à un grand nombre de ressources serveurs utiles. Pour plus d'informations sur les modules d'archivage, voir Chapitre 51.

Quand la commande d'archivage est terminée par un signal (autre que le signal SIGTERM qui est utilisé comme partie d'un arrêt du serveur) ou une erreur par le shell avec un statut de sortie plus grand que 125 (tel que « command not found »), ou si la fonction d'archivage émet un message ERROR ou FATAL, le processus d'archivage annule ou est redémarré par le processus postmaster. Dans de tels cas, l'échec n'est pas reporté dans pg_stat_archiver.

En général, les commande et bibliothèques d'archivage doivent être conçue pour refuser d'écraser tout fichier archive déjà existant. C'est une fonctionnalité de sécurité importante pour préserver l'intégrité de l'archive dans le cas d'une erreur de l'administrateur (comme l'envoi de la sortie de deux serveurs différents dans le même répertoire d'archivage).

Il est conseillé de tester la commande ou la bibliothèque d'archivage proposée pour s'assurer, qu'en effet, elle n'écrase pas un fichier existant, *et qu'elle renvoie, respectivement, un statut différent de zéro ou `false` dans ce cas*. La commande pour Unix en exemple ci-dessus le garantit en incluant une étape `test` séparée. Sur certaines plateformes Unix, la commande `cp` dispose d'une option, comme `-i` pouvant être utilisé pour faire la même chose, mais en moins verbeux. Cependant, vous ne devriez pas vous baser là-dessus sans vous assurer que le code de sortie renvoyé est le bon. (En particulier, la commande `cp` de GNU renvoie un code zéro quand `-i` est utilisé et que le fichier cible existe déjà, ce qui n'est *pas* le comportement désiré.)

Lors de la conception de votre configuration d'archivage, il faut considérer ce qui peut se produire si la commande ou la bibliothèque d'archivage échoue de façon répétée, que ce soit parce qu'une intervention de l'opérateur s'avère nécessaire ou par manque d'espace dans le répertoire d'archivage. Ceci peut arriver, par exemple, lors de l'écriture sur une cartouche sans changeur automatique ; quand la cartouche est pleine, rien ne peut être archivé tant que la cassette n'est pas changée. Toute erreur ou requête à un opérateur humain doit être rapportée de façon appropriée pour que la situation puisse être résolue rapidement. Le répertoire `pg_wal/` continue à se remplir de fichiers de segment WAL jusqu'à la résolution de la situation.(Si le système de fichiers contenant `pg_wal/` se remplit, PostgreSQL s'arrête en mode PANIC. Aucune transaction validée n'est perdue mais la base de données est inaccessible tant que de l'espace n'a pas été libéré.)

La vitesse de la commande ou de la bibliothèque d'archivage n'est pas importante, tant qu'elle suit le rythme de génération des journaux de transactions du serveur. Les opérations normales continuent même si le processus d'archivage est un peu plus lent. Si l'archivage est significativement plus lent, alors la quantité de données qui peut être perdue croît. Cela signifie aussi que le répertoire `pg_wal/` contient un grand nombre de fichiers segment non archivés, qui peuvent finir par remplir l'espace disque disponible. Il est conseillé de surveiller le processus d'archivage pour s'assurer que tout fonctionne normalement.

Lors de l'écriture de la commande ou de la bibliothèque d'archivage, il faut garder à l'esprit que les noms de fichier à archiver peuvent contenir jusqu'à 64 caractères et être composés de toute combinaison de lettres ASCII, de chiffres et de points. Il n'est pas nécessaire de conserver le chemin relatif original (`%p`) mais il est nécessaire de se rappeler du nom du fichier (`%f`).

Bien que l'archivage des journaux de transactions autorise à restaurer toute modification réalisée sur les données de la base, il ne restaure pas les modifications effectuées sur les fichiers de configuration (c'est-à-dire `postgresql.conf`, `pg_hba.conf` et `pg_ident.conf`) car ceux-ci sont édités manuellement et non au travers d'opérations SQL. Il est souhaitable de conserver les fichiers de configuration à un endroit où ils sont sauvegardés par les procédures standard de sauvegarde du système de fichiers. Voir la Section 20.2 pour savoir comment modifier l'emplacement des fichiers de configuration.

La commande ou la bibliothèque d'archivage n'est appelée que sur les fichiers segments WAL complets. Du coup, si le serveur engendre peu de trafic dans les journaux de transactions (ou qu'il y a des périodes de calme où le trafic est léger), il peut y avoir un long délai entre la fin d'une transaction

et son enregistrement garanti dans le stockage d'archive. Pour placer une limite sur l'ancienneté des données archivées, il faut configurer archive_timeout, ce qui permet de forcer le serveur à changer de fichier segment WAL passé ce délai. Les fichiers archivés lors d'un tel forçage ont toujours la même taille que les fichiers complets. Il est donc déconseillé de configurer un délai `archive_timeout` trop court -- cela fait grossir anormalement le stockage. Une minute pour `archive_timeout` est généralement raisonnable.

De plus, le changement d'un segment peut être forcé manuellement avec `pg_switch_wal`. Cela permet de s'assurer qu'une transaction tout juste terminée est archivée aussi vite que possible. D'autres fonctions utilitaires relatives à la gestion des journaux de transactions sont disponibles dans Tableau 9.89.

Quand `wal_level` est configuré à la valeur `minimal`, certaines commandes SQL sont optimisées pour éviter la journalisation des transactions, de la façon décrite dans Section 14.4.7. Si l'archivage ou la réplication en flux sont activés lors de l'exécution d'une de ces instructions, les journaux de transaction ne contiennent pas suffisamment d'informations pour une restauration via les archives. (La restauration après un arrêt brutal n'est pas affectée.) Pour cette raison, `wal_level` ne peut être modifié qu'au lancement du serveur. Néanmoins, `archive_command` et `archive_library` peut être modifié par rechargement du fichier de configuration. Si vous archivez via le shell et souhaitez arrêter temporairement l'archivage, une façon de le faire est de peut placer une chaîne vide (`''`) dans `archive_command`. Les journaux de transactions sont alors accumulés dans `pg_wal/` jusqu'au rétablissement d'un paramètre `archive_command` fonctionnel.

26.3.2. Réaliser une sauvegarde de base

La manière la plus simple pour effectuer une sauvegarde revient à utiliser l'outil pg_basebackup. Il peut créer une sauvegarde de base sous la forme soit de fichiers standards, soit d'une archive tar. Pour les cas plus complexes, il est possible de réaliser une sauvegarde de base en utilisant l'API bas niveau (voir Section 26.3.3).

La durée d'une sauvegarde de base n'est pas toujours un critère déterminant. Toutefois, si vous exploitez votre serveur avec l'option `full_page_writes` désactivée, vous constaterez une baisse des performances lorsque la sauvegarde est effectuée car l'option `full_page_writes` est activée de force pendant les opérations de sauvegarde.

Pour utiliser une sauvegarde, vous devez conserver tous les fichiers segments WAL générés pendant et après la sauvegarde des fichiers. Pour vous aider dans cette tâche, le processus de sauvegarde crée un *fichier historique de sauvegarde* qui est immédiatement enregistré dans la zone d'archivage des journaux de transactions. Le nom de ce fichier reprend le nom du premier fichier segment WAL que vous devez conserver. Par exemple, si le premier fichier segment WAL à garder est `000000010000123400000055CD`, alors le fichier historique de sauvegarde sera nommé de la manière suivante `000000010000123400000055CD.007C9330.backup`. (La seconde partie du nom de fichier indique la position exacte à l'intérieur du fichier segment WAL. Cette information peut être ignorée.) Une fois que vous avez archivé avec précaution la sauvegarde de base et les fichiers segments WAL générés pendant la sauvegarde (tel qu'indiqué par le fichier historique de sauvegarde), tous les fichiers segments WAL antérieurs ne sont plus nécessaires pour restaurer votre sauvegarde de base. Ils peuvent être supprimés. Toutefois, il est conseillé de conserver plusieurs groupes de sauvegardes pour être absolument certain de récupérer vos données.

Le fichier historique de sauvegarde est un simple fichier texte. Il contient le label que vous avez attribué à l'opération pg_basebackup, ainsi que les dates de début, de fin et la liste des fichiers segments WAL de la sauvegarde. Si vous avez utilisé le label pour identifier le fichier de sauvegarde associé, alors le fichier historique vous permet de savoir quel fichier de sauvegarde vous devez utiliser pour la restauration.

Puisque vous devez archiver tous les fichiers segments WAL depuis votre dernière sauvegarde de base, l'intervalle entre deux sauvegardes de base doit être déterminé en fonction de l'espace de stockage que vous avez alloué pour l'archivage des fichiers segments WAL. Vous devez également prendre

en compte le temps de restauration (le système devra rejouer tous les fichiers segments WAL, cela prendra un certain temps si la sauvegarde de base est ancienne).

26.3.3. Effectuer une sauvegarde de base avec l'API bas niveau

La procédure pour créer une sauvegarde de base en utilisant l'API bas niveau contient plus d'étapes que la méthode pg_basebackup mais elle est relativement simple. Il est très important que ces étapes soient exécutées séquentiellement et de vérifier que chaque étape s'est déroulée correctement avant de passer à la suivante.

Plusieurs sauvegardes peuvent s'exécuter en même temps (à la fois celles démarrées en utilisant cette API, soit celles démarrées en utilisant pg_basebackup).

1. S'assurer que l'archivage des journaux de transactions est activé et fonctionnel.

2. Se connecter au serveur (peu importe la base) en tant qu'utilisateur ayant les droits d'exécuter la fonction `pg_backup_start` (donc soit un superutilisateur, soit un utilisateur ayant gagné le droit `EXECUTE` sur cette fonction) et lancer la commande :

   ```
   SELECT pg_backup_start(label => 'label', fast => false);
   ```

   où `label` est une chaîne utilisée pour identifier de façon unique l'opération de sauvegarde. La connexion appelant `pg_backup_start` doit être maintenue jusqu'à la fin de la sauvegarde. Dans le cas contraire, la sauvegarde sera automatiquement avortée.

   Les sauvegardes en ligne sont toujours démarrées au début d'un checkpoint. Par défaut, `pg_backup_start` attendra le prochain checkpoint planifié pour se terminer, ce qui peut prendre un long moment (voir les paramètres de configuration checkpoint_timeout et checkpoint_completion_target). C'est généralement préférable car cela minimise l'impact sur le système en cours d'exécution. Si vous voulez démarrer la sauvegarde aussi vite que possible, passez `true` sur le second paramètre de `pg_backup_start` et il demandera un checkpoint immédiat, qui finira aussi rapidement que possible en utilisant autant de bande passante disque que possible.

3. Effectuer la sauvegarde à l'aide de tout outil de sauvegarde du système de fichiers, tel tar ou cpio (mais ni pg_dump ni pg_dumpall). Il n'est ni nécessaire ni désirable de stopper les opérations normales de la base de données pour cela. Voir la section Section 26.3.3.1 pour les considérations à prendre en compte durant cette sauvegarde.

4. Dans la même connexion que précédemment, lancer la commande :

   ```
   SELECT * FROM pg_backup_stop(wait_for_archive => true);
   ```

   Cela met fin au processus de sauvegarde. Sur un serveur primaire, cela réalise aussi une bascule automatique au prochain segment de journal de transactions. Sur un serveur secondaire, il n'est pas possible de basculer automatiquement les segments des journaux de transactions, donc vous pourriez vouloir utiliser `pg_switch_wal` sur le primaire pour réaliser une bascule manuelle. Cette bascule est nécessaire pour permettre au dernier fichier de segment WAL écrit pendant la sauvegarde d'être immédiatement archivable.

   La fonction `pg_backup_stop` retournera une ligne avec trois valeurs. Le second de ces champs devra être écrit dans un fichier nommé `backup_label` dans le répertoire racine de la sauvegarde. Le troisième champ devra être écrit dans un fichier nommé `tablespace_map` sauf si le champ est vide. Ces fichiers sont vitaux pour le fonctionnement de la sauvegarde et doivent être écrits octet par octet sans modification, ce qui nécessite de les ouvrir en mode binaire.

5. Une fois que les fichiers segment WAL utilisés lors de la sauvegarde sont archivés, c'est terminé. Le fichier identifié par le résultat de `pg_backup_stop` est le dernier segment nécessaire pour

produire un jeu complet de fichiers de sauvegarde. Sur un serveur primaire, si `archive_mode` est activé et que le paramètre `wait_for_archive` est activé (valeur `true`), `pg_backup_stop` ne rend pas la main avant que le dernier segment n'ait été archivé. Sur un serveur secondaire, le paramètre `archive_mode` doit valoir `always` pour que la fonction `pg_backup_stop` soit en attente. L'archivage de ces fichiers est automatique puisque `archive_command` ou `archive_library` est déjà configuré. Dans la plupart des cas, c'est rapide, mais il est conseillé de surveiller le système d'archivage pour s'assurer qu'il n'y a pas de retard. Si le processus d'archivage a pris du retard en raison d'échecs de la commande ou de la bibliothèque d'archivage, il continuera d'essayer jusqu'à ce que l'archivage réussisse et que la sauvegarde soit complète. Pour positionner une limite au temps d'exécution de `pg_backup_stop`, il faut positionner `statement_timeout` à une valeur appropriée, mais il faut noter que si `pg_backup_stop` est interrompu à cause de cette configuration, la sauvegarde peut ne pas être correcte.

Si le processus de sauvegarde surveille et s'assure que tous les fichiers segments WAL nécessaires à la sauvegarde soient archivés avec succès, le paramètre `wait_for_archive` (positionné à true par défaut) peut être positionné à false pour que `pg_backup_stop` renvoie la main dès que l'enregistrement de fin de sauvegarde est écrit dans le journal de transaction en cours. Par défaut, `pg_backup_stop` attendra jusqu'à ce que tous les journaux de transactions aient été archivés, ce qui peut prendre un certain temps. Cette option doit être utilisée avec précaution : si l'archivage des journaux de transactions n'est pas supervisé correctement, alors la sauvegarde pourrait ne pas inclure tous les fichiers segments WAL, et serait donc incomplète et impossible à restaurer.

26.3.3.1. Sauvegarder le répertoire de données

Certains outils de sauvegarde de fichiers émettent des messages d'avertissement ou d'erreur si les fichiers qu'ils essaient de copier sont modifiés au cours de la copie. Cette situation, normale lors de la sauvegarde d'une base active, ne doit pas être considérée comme une erreur ; il suffit de s'assurer que ces messages puissent être distingués des autres messages. Certaines versions de rsync, par exemple, renvoient un code de sortie distinct en cas de « disparition de fichiers source ». Il est possible d'écrire un script qui considère ce code de sortie comme normal.

De plus, certaines versions de GNU tar retournent un code d'erreur qu'on peut confondre avec une erreur fatale si le fichier a été tronqué pendant sa copie par tar. Heureusement, les versions 1.16 et suivantes de GNU tar retournent 1 si le fichier a été modifié pendant la sauvegarde et 2 pour les autres erreurs. Avec GNU tar version 1.23 et les versions ultérieures, vous pouvez utiliser les options d'avertissement `--warning=no-file-changed --warning=no-file-removed` pour cacher les messages d'avertissement en relation.

La sauvegarde doit inclure tous les fichiers du répertoire dde l'instance de bases de données(`/usr/local/pgsql/data`, par exemple). Si des *tablespaces* qui ne se trouvent pas dans ce répertoire sont utilisés, il ne faut pas oublier de les inclure (et s'assurer également que la sauvegarde archive les liens symboliques comme des liens, sans quoi la restauration va corrompre les *tablespaces*).

Néanmoins, les fichiers du sous-répertoire `pg_wal/`, contenu dans le répertoire de l'instance, devraient être omis. Ce léger ajustement permet de réduire le risque d'erreurs lors de la restauration. C'est facile à réaliser si `pg_wal/` est un lien symbolique vers quelque endroit extérieur au répertoire de l'instance, ce qui est une configuration courante pour des raisons de performance. Il peut être aussi intéressant d'exclure `postmaster.pid` et `postmaster.opts`, qui enregistrent des informations sur le postmaster en cours d'exécution, mais pas sur le postmaster qui va utiliser cette sauvegarde. De plus, ces fichiers peuvent poser problème à pg_ctl.

C'est souvent une bonne idée d'omettre de la sauvegarde les fichiers provenant du répertoire `pg_replslot/` de l'instance, pour que les slots de réplication existant sur le primaire ne deviennent pas partie intégrante de la sauvegarde. Dans le cas contraire, l'utilisation de la sauvegarde pour créer un secondaire pourrait résulter en une rétention infinie des journaux de transactions sur le secondaire et aussi de la fragmentation sur le primaire si les messages de retour d'un secondaire en Hot Standby sont activés, parce que les clients qui utilisent ces slots de réplication se connecteront toujours et mettront à jour les slots sur le primaire et non pas sur le secondaire. Même si la sauvegarde a pour

but d'être utilisée pour la création d'un nouveau primaire, copier les slots de réplication n'est pas un comportement attendu car il n'a pas de raison d'être, le contenu de ces slots sera très probablement obsolète au moment où le nouveau primaire sera en ligne.

Le contenu des répertoires `pg_dynshmem/`, `pg_notify/`, `pg_serial/`, `pg_snapshots/`, `pg_stat_tmp/` et `pg_subtrans/` (mais pas les répertoires eux-même) peut être exclu de la sauvegarde puisqu'il sera réinitialisé au démarrage de postmaster.

N'importe quel fichier ou répertoire commençant par `pgsql_tmp` peut être exclu de la sauvegarde. Ces fichiers sont supprimés au démarrage de postmaster et les répertoires seront recréés si nécessaire.

Les fichiers `pg_internal.init` peuvent être omis de la sauvegarde quand un fichier de ce nom est trouvé. Ces fichiers contiennent les données du cache des relations qui est toujours reconstruit lors de la restauration.

Le fichier de label de la sauvegarde inclut la chaîne de label passée à `pg_backup_start`, l'heure à laquelle `pg_backup_start` a été exécutée et le nom du fichier segment WAL initial. En cas de confusion, il est ainsi possible de regarder dans ce fichier de label et de déterminer avec précision de quelle session de sauvegarde il provient. Le fichier des tablespaces inclut les noms des liens symboliques s'ils existent dans le répertoire `pg_tblspc/` et le chemin complet de chaque lien symbolique. Néanmoins, ces fichiers n'existent pas uniquement pour vous informer. Leurs présences et contenus sont critiques au bon déroulement du processus de restauration.

Il est aussi possible de faire une sauvegarde alors que le serveur est arrêté. Dans ce cas, les fonctions `pg_backup_start` et `pg_backup_stop` ne peuvent pas être utilisées. L'utilisateur doit alors se débrouiller pour identifier les fichiers de sauvegarde et déterminer jusqu'où remonter avec les fichiers segments WAL associés. Il est généralement préférable de suivre la procédure d'archivage continu décrite ci-dessus.

26.3.4. Restauration à partir d'un archivage continu

Le pire est arrivé et il faut maintenant repartir d'une sauvegarde. Voici la procédure :

1. Arrêter le serveur s'il est en cours d'exécution.

2. Si la place nécessaire est disponible, copier le répertoire complet de données de l'instance et tous les *tablespaces* dans un emplacement temporaire en prévision d'un éventuel besoin ultérieur. Cette précaution nécessite qu'un espace suffisant sur le système soit disponible pour contenir deux copies de la base de données existante. S'il n'y a pas assez de place disponible, il faut au minimum copier le contenu du sous-répertoire `pg_wal` du répertoire des données de l'instance car il peut contenir des journaux qui n'ont pas été archivés avant l'arrêt du serveur.

3. Effacer tous les fichiers et sous-répertoires existant sous le répertoire des données de l'instance et sous les répertoires racines des *tablespaces*.

4. Restaurer les fichiers de la base de données à partir de la sauvegarde des fichiers. Il faut veiller à ce qu'ils soient restaurés avec le bon propriétaire (l'utilisateur système de la base de données, et non pas `root` !) et avec les bons droits. Si des *tablespaces* sont utilisés, il faut s'assurer que les liens symboliques dans `pg_tblspc/` ont été correctement restaurés.

5. Supprimer tout fichier présent dans `pg_wal/` ; ils proviennent de la sauvegarde et sont du coup probablement obsolètes. Si `pg_wal/` n'a pas été archivé, il suffit de recréer ce répertoire en faisant attention à le créer en tant que lien symbolique, si c'était le cas auparavant.

6. Si des fichiers segments WAL non archivés ont été sauvegardés dans l'étape 2, les copier dans `pg_wal/`. Il est préférable de les copier plutôt que de les déplacer afin qu'une version non modifiée de ces fichiers soit toujours disponible si un problème survient et qu'il faille recommencer.

7. Configurer les commandes de restauration dans `postgresql.conf`(voir Section 20.5.5) et créer un fichier `recovery.signal` dans le répertoire des données de l'instance. De plus, il peut être judicieux de modifier temporairement le fichier `pg_hba.conf` pour empêcher les utilisateurs ordinaires de se connecter tant qu'il n'est pas certain que la restauration a réussi.

8. Démarrer le serveur. Le serveur se trouve alors en mode restauration et commence la lecture des fichiers segments WAL archivés dont il a besoin. Si la restauration se termine sur une erreur externe, le serveur peut tout simplement être relancé. Il continue alors la restauration. À la fin du processus de restauration, le serveur supprime le fichier `recovery.signal`(pour éviter de retourner accidentellement en mode de restauration), puis passe en mode de fonctionnement normal.

9. Inspecter le contenu de la base de données pour s'assurer que la restauration a bien fonctionné. Dans le cas contraire, retourner à l'étape 1. Si tout va bien, le fichier `pg_hba.conf` peut être restauré pour autoriser les utilisateurs à se reconnecter.

Le point clé de tout ceci est la configuration de la restauration qui décrit comment et jusqu'où restaurer. La seule chose qu'il faut absolument préciser dans `postgresql.conf`, c'est la valeur du paramètre `restore_command` qui indique à PostgreSQL comment récupérer les fichiers segments WAL archivés. À l'instar d'`archive_command`, c'est une chaîne de commande shell. Elle peut contenir `%f`, qui est remplacé par le nom du journal souhaité, et `%p`, qui est remplacé par le chemin du répertoire où copier le journal.(Le nom du chemin est relatif au répertoire de travail du serveur, c'est-à-dire le répertoire des données du cluster.) Pour écrire le caractère `%` dans la commande, on utilise `%%`. La commande la plus simple ressemble à :

```
restore_command = 'cp /mnt/serveur/répertoire_archive/%f %p'
```

qui copie les fichiers segments WAL précédemment archivés à partir du répertoire `/mnt/serveur/répertoire_archive`. Il est toujours possible d'utiliser une commande plus compliquée, voire même un script shell qui demande à l'utilisateur de monter la cassette appropriée.

Il est important que la commande renvoie un code de sortie différent de zéro en cas d'échec. Des fichiers absents de l'archive *seront* demandés à la commande ; elle doit renvoyer autre chose que zéro dans ce cas. Ce n'est pas une condition d'erreur. Une exception est possible si la commande a été terminée par un signal (autre que SIGTERM, qui est utilisé pour l'arrêt du serveur) ou par une erreur shell (comme une commande introuvable). Dans ces cas, la restauration va s'arrêter et le serveur ne démarrera plus.

Tous les fichiers demandés ne seront pas des fichiers segments WAL ; vous pouvez aussi vous attendre à des demandes de fichiers suffixés par `.history`. Il faut également garder à l'esprit que le nom de base du chemin `%p` diffère de `%f` ; ils ne sont pas interchangeables.

Les segments WAL qui ne se trouvent pas dans l'archive sont recherchés dans `pg_wal/` ; cela autorise l'utilisation de segments récents non archivés. Néanmoins, les segments disponibles dans l'archive sont utilisés de préférence aux fichiers contenus dans `pg_wal/`.

Normalement, la récupération traite tous les segments WAL disponibles, restaurant du coup la base de données à l'instant présent (ou aussi proche que possible, en fonction des fichiers segments WAL disponibles). Une récupération normale se finit avec un message « fichier non trouvé », le texte exact du message d'erreur dépendant du choix de `restore_command`. Un message d'erreur au début de la restauration peut également apparaître concernant un fichier nommé dont le nom ressemble à `00000001.history`. Ceci est aussi normal et n'indique pas un problème dans les situations de restauration standards. Voir Section 26.3.5 pour plus d'informations.

Pour restaurer jusqu'à un moment précis (avant que le DBA junior n'ait supprimé la table principale par exemple), il suffit d'indiquer le point d'arrêt requis. Le point d'arrêt, aussi nommé « cible de récupération » (« recovery target » dans la version originale) , peut être précisé par une combinaison de date et d'heure, par un point de restauration nommé ou par un identifiant de transaction. Actuellement, seules les options date/heure et point de restauration nommé sont vraiment utilisables car il n'existe pas d'outils permettant d'identifier avec précision l'identifiant de transaction à utiliser.

> **Note**
>
> Le point d'arrêt doit être postérieur à la fin de la sauvegarde de la base (le moment où `pg_backup_stop` se termine). Une sauvegarde ne peut pas être utilisée pour repartir d'un instant où elle était encore en cours (pour ce faire, il faut récupérer la sauvegarde précédente et rejouer à partir de là).

Si la restauration fait face à une corruption des données des journaux de transactions, le rejeu s'interrompt juste avant la corruption détectée et le serveur ne démarre pas. Dans un tel cas, le processus de restauration peut alors être ré-exécuté à partir du début en précisant une « cible de restauration » antérieure au point de corruption pour permettre à cette dernière de se terminer correctement. Si la restauration échoue pour une raison externe (arrêt brutal du système ou archive devenue inaccessible), la restauration peut être simplement relancée. Elle redémarre alors quasiment là où elle a échoué. Le redémarrage de la restauration fonctionne comme les checkpoints du déroulement normal : le serveur force une écriture régulière de son état sur les disques et actualise alors le fichier `pg_control` pour indiquer que les données déjà traitées des journaux de transactions n'ont plus à être parcourues.

26.3.5. Lignes temporelles (*Timelines*)

La possibilité de restaurer la base de données à partir d'un instantané crée une complexité digne des histoires de science-fiction traitant du voyage dans le temps et des univers parallèles.

Par exemple, dans l'historique original de la base de données, supposez qu'une table critique ait été supprimée à 17h15 mardi soir, mais que personne n'ait réalisé cette erreur avant mercredi midi. Sans stress, la sauvegarde est récupérée et restaurée dans l'état où elle se trouvait à 17h14 mardi soir. La base est fonctionnelle. Dans *cette* histoire de l'univers de la base de données, la table n'a jamais été supprimée. Or, l'utilisateur réalise peu après que ce n'était pas une si grande idée et veut revenir à un quelconque moment du mercredi matin. Cela n'est pas possible, si, alors que la base de données est de nouveau fonctionnelle, elle réutilise certaines séquences de fichiers segments WAL qui permettent de retourner à ce point. Il est donc nécessaire de pouvoir distinguer les séries d'enregistrements des journaux de transactions engendrées après la restauration de l'instantané de celles issues de l'historique originel de la base.

Pour gérer ces difficultés, PostgreSQL inclut la notion de *lignes temporelles* (ou *timelines*). Quand une récupération d'archive est terminée, une nouvelle ligne temporelle est créée pour identifier la série d'enregistrements de journaux de transactions produits après cette restauration. Le numéro d'identifiant de la timeline est inclus dans le nom des fichiers segments WAL. De ce fait, une nouvelle timeline ne réécrit pas sur les données engendrées par des timelines précédentes. En fait, il est possible d'archiver plusieurs timelines différentes. Bien que cela semble être une fonctionnalité inutile, cela peut parfois sauver des vies. Dans une situation où l'instantané à restaurer n'est pas connu avec certitude, il va falloir tester les restaurations de différents instantanés jusqu'à trouver le meilleur. Sans les timelines, ce processus engendre vite un bazar ingérable. Avec les timelines, il est possible de restaurer *n'importe quel* état précédent, même les états de branches temporelles abandonnées.

Chaque fois qu'une nouvelle timeline est créée, PostgreSQL crée un fichier d'« historique des timelines » qui indique à quelle timeline il est attaché, et depuis quand. Ces fichiers d'historique sont nécessaires pour permettre au système de choisir les bons fichiers segments WAL lors de la restauration à partir d'une archive qui contient plusieurs timelines. Ils sont donc archivés comme tout fichier segment WAL. Puisque ce sont de simples fichiers texte, il est peu coûteux et même judicieux de les conserver indéfiniment(contrairement aux fichiers segments, bien plus volumineux). Il est possible d'ajouter des commentaires au fichier d'historique expliquant comment et pourquoi cette timeline a été créée. De tels commentaires s'avèrent précieux lorsque l'expérimentation conduit à de nombreuses timelines.

Par défaut, la restauration s'effectue jusqu'à la dernière timeline trouvée dans l'archive. Si vous souhaitez effectuer la restauration uniquement pour la timeline de la sauvegarde ou jusqu'à une

timeline précise (c'est-à-dire retourner à un état enregistré après une tentative de restauration), il faut préciser `current` ou l'identifiant de la timeline cible dans recovery_target_timeline. Il n'est pas possible de restaurer dans des timelines antérieures à la sauvegarde.

26.3.6. Conseils et exemples

Quelques conseils de configuration de l'archivage continu sont donnés ici.

26.3.6.1. Configuration de la récupération

Il est possible d'utiliser les capacités de sauvegarde de PostgreSQL pour produire des sauvegardes autonomes à chaud. Ce sont des sauvegardes qui ne peuvent pas être utilisées pour la restauration à un instant donné, mais ce sont des sauvegardes qui sont typiquement plus rapides à obtenir et à restaurer que celles issues de pg_dump. (Elles sont aussi bien plus volumineuses qu'un export pg_dump, il se peut donc que l'avantage de rapidité soit négatif.)

Comme pour les sauvegardes de base, la manière la plus simple de créer une sauvegarde à chaud autonome est d'utiliser l'outil pg_basebackup. Si vous ajoutez le paramètre `-X` au lancement de la sauvegarde, tout l'historique des transactions nécessaire sera inclus automatiquement dans la sauvegarde et vous n'aurez pas d'action supplémentaire à effectuer pour restaurer votre sauvegarde.

26.3.6.2. Compression des fichiers archives

Si la taille du stockage des archives est un problème, vous pouvez utiliser gzip pour compresser les fichiers archives :

```
archive_command = 'gzip < %p > /mnt/server/archivedir/%f'
```

Vous aurez alors besoin d'utiliser gunzip pendant la restauration :

```
restore_command = 'gunzip < /mnt/server/archivedir/%f.gz > %p'
```

26.3.6.3. Scripts `archive_command`

Nombreux sont ceux qui choisissent d'utiliser des scripts pour définir leur `archive_command`, de sorte que leur `postgresql.conf` semble très simple :

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

Utiliser un script séparé est conseillé à chaque fois qu'il est envisagé d'utiliser plusieurs commandes pour le processus d'archivage. Ainsi, toute la complexité est gérée dans le script qui peut être écrit dans un langage de scripts populaires comme bash ou perl.

Quelques exemples de besoins résolus dans un script :

- copier des données vers un stockage distant ;
- copier les journaux de transactions en groupe pour qu'ils soient transférés toutes les trois heures plutôt qu'un à la fois ;
- s'interfacer avec d'autres outils de sauvegarde et de restauration ;
- s'interfacer avec un outil de surveillance pour y renvoyer les erreurs.

> **Astuce**
>
> Lors de l'utilisation d'un script pour le paramètre `archive_command`, il est préférable d'activer logging_collector. Tout message écrit sur stderr à partir du script apparaîtra ensuite

dans les traces du serveur, permettant un diagnostic facilité de configurations complexes en cas de problème.

26.3.7. Restrictions

Au moment où ces lignes sont écrites, plusieurs limitations de la technique d'archivage en continu sont connues. Elles seront probablement corrigées dans une prochaine version :

- Si une commande `CREATE DATABASE` est exécutée alors qu'une sauvegarde est en cours, et que la base de données modèle utilisée par l'instruction `CREATE DATABASE` est à son tour modifiée pendant la sauvegarde, il est possible que la récupération propage ces modifications dans la base de données créée. Pour éviter ce risque, il est préférable de ne pas modifier les bases de données modèle lors d'une sauvegarde de base.

- Les commandes `CREATE TABLESPACE` sont tracées dans les journaux de transactions avec le chemin absolu et sont donc rejouées en tant que créations de *tablespace* suivant le même chemin absolu. Cela n'est pas forcément souhaitable si le journal est rejoué sur une autre machine. De plus, cela peut s'avérer dangereux même lorsque le journal est rejoué sur la même machine, mais dans un répertoire différent : la ré-exécution surcharge toujours le contenu du *tablespace* original. Pour éviter de tels problèmes, la meilleure solution consiste à effectuer une nouvelle sauvegarde de la base après la création ou la suppression de *tablespace*.

De plus, il faut garder à l'esprit que le format actuel des journaux de transactions est extrêmement volumineux car il inclut de nombreuses images complètes des blocs disques. Ces images de bloc sont conçues pour supporter la restauration après un arrêt brutal, puisqu'il peut être nécessaire de corriger des blocs disque partiellement écrits. En fonction du matériel et des logiciels composant le système, le risque d'écriture partielle peut être suffisamment faible pour être ignoré, auquel cas le volume total des traces archivées peut être considérablement réduit par la désactivation des images de bloc à l'aide du paramètre full_page_writes(lire les notes et avertissements dans Chapitre 30 avant de le faire). Désactiver les images de bloc n'empêche pas l'utilisation des traces pour les opérations PITR. Un piste éventuelle de développement futur consiste à compresser les données des journaux de transactions archivés en supprimant les copies inutiles de blocs même si `full_page_writes` est actif. Entre temps, les administrateurs peuvent souhaiter réduire le nombre d'images de blocs inclus dans les journaux de transactions en augmentant autant que possible les paramètres d'intervalle entre les checkpoints.

Chapitre 27. Haute disponibilité, répartition de charge et réplication

Des serveurs de bases de données peuvent travailler ensemble pour permettre à un serveur secondaire (*standby* dans la version originale) de prendre rapidement la main si le serveur principal échoue (haute disponibilité, ou *high availability*), ou pour permettre à plusieurs serveurs de servir les mêmes données (répartition de charge, ou *load balancing*). Idéalement, les serveurs de bases de données peuvent travailler ensemble sans jointure.

Il est aisé de faire coopérer des serveurs web qui traitent des pages web statiques en répartissant la charge des requêtes web sur plusieurs machines. Dans les faits, les serveurs de bases de données en lecture seule peuvent également coopérer facilement. Malheureusement, la plupart des serveurs de bases de données traitent des requêtes de lecture/écriture et, de ce fait, collaborent plus difficilement. En effet, alors qu'il suffit de placer une seule fois les données en lecture seule sur chaque serveur, une écriture sur n'importe quel serveur doit, elle, être propagée à tous les serveurs afin que les lectures suivantes sur ces serveurs renvoient des résultats cohérents.

Ce problème de synchronisation représente la difficulté fondamentale à la collaboration entre serveurs. Comme la solution au problème de synchronisation n'est pas unique pour tous les cas pratiques, plusieurs solutions coexistent. Chacune répond de façon différente et minimise cet impact au regard d'une charge spécifique.

Certaines solutions gèrent la synchronisation en autorisant les modifications des données sur un seul serveur. Les serveurs qui peuvent modifier les données sont appelés serveur en lecture/écriture, *primaire* ou serveur *primaire*. Les serveurs qui suivent les modifications du primaire sont appelés des serveurs *standby* ou des serveurs *secondaires*. Un serveur secondaire auquel on ne peut pas se connecter tant qu'il n'a pas été promu en serveur primaire est appelé un *warm standby*, et un serveur qui peut accepter des connexions et répondre à des requêtes en lecture seule est appelé un *hot standby*.

Certaines solutions sont synchrones, ce qui signifie qu'une transaction de modification de données n'est pas considérée valide tant que tous les serveurs n'ont pas validé la transaction. Ceci garantit qu'un *failover* ne perd pas de données et que tous les serveurs en répartition de charge retournent des résultats cohérents, quel que soit le serveur interrogé. Au contraire, les solutions asynchrones autorisent un délai entre la validation et sa propagation aux autres serveurs. Cette solution implique une éventuelle perte de transactions lors de la bascule sur un serveur de sauvegarde, ou l'envoi de données obsolètes par les serveurs à charge répartie. La communication asynchrone est utilisée lorsque la version synchrone est trop lente.

Les solutions peuvent aussi être catégorisées par leur granularité. Certaines ne gèrent que la totalité d'un serveur de bases de données alors que d'autres autorisent un contrôle par table ou par base.

Il importe de considérer les performances dans tout choix. Il y a généralement un compromis à trouver entre les fonctionnalités et les performances. Par exemple, une solution complètement synchrone sur un réseau lent peut diviser les performances par plus de deux, alors qu'une solution asynchrone peut n'avoir qu'un impact minimal sur les performances.

Le reste de cette section souligne différentes solutions de *failover*, de réplication et de répartition de charge.

27.1. Comparaison de différentes solutions

Failover sur disque partagé

Le *failover* (ou bascule sur incident) sur disque partagé élimine la surcharge de synchronisation par l'existence d'une seule copie de la base de données. Il utilise un seul ensemble de disques

partagé par plusieurs serveurs. Si le serveur principal échoue, le serveur en attente est capable de monter et démarrer la base comme s'il récupérait d'un arrêt brutal. Cela permet un *failover* rapide sans perte de données.

La fonctionnalité de matériel partagé est commune aux périphériques de stockage en réseau. Il est également possible d'utiliser un système de fichiers réseau bien qu'il faille porter une grande attention au système de fichiers pour s'assurer qu'il a un comportement POSIX complet (voir Section 19.2.2.1). Cette méthode comporte une limitation significative : si les disques ont un problème ou sont corrompus, le serveur primaire et le serveur en attente sont tous les deux non fonctionnels. Un autre problème est que le serveur en attente ne devra jamais accéder au stockage partagé tant que le serveur principal est en cours d'exécution.

Réplication de système de fichiers (périphérique bloc)

Il est aussi possible d'utiliser cette fonctionnalité d'une autre façon avec une réplication du système de fichiers, où toutes les modifications d'un système de fichiers sont renvoyées sur un système de fichiers situé sur un autre ordinateur. La seule restriction est que ce miroir doit être construit de telle sorte que le serveur en attente dispose d'une version cohérente du système de fichiers -- spécifiquement, les écritures sur le serveur en attente doivent être réalisées dans le même ordre que celles sur le primaire. DRBD est une solution populaire de réplication de systèmes de fichiers pour Linux.

Envoi des journaux de transactions

Les serveurs secondaires, *warm et hot standby*, (voir Section 27.2) peuvent conserver leur cohérence en lisant un flux d'enregistrements de WAL. Si le serveur principal échoue, le serveur secondaire contient pratiquement toutes les données du serveur principal et peut rapidement devenir le nouveau serveur primaire. Ça peut être synchrone mais ça ne peut se faire que pour le serveur de bases complet.

Un serveur secondaire peut être implémenté en utilisant la recopie de journaux par fichier(Section 27.2) ou la *streaming replication* (réplication en continu ou en flux, voir Section 27.2.5), ou une combinaison des deux. Pour des informations sur le *hot standby*, voyez Section 27.4.

Réplication logique

La réplication logique autorise un serveur de bases de données à envoyer un flux de modifications de données à un autre serveur. La réplication logique de PostgreSQL construit un flux de modifications logiques de données à partir des journaux de transactions. La réplication logique permet la réplication des modifications de données de tables individuelles. La réplication logique ne requiert pas qu'un serveur particulier soit désigné comme serveur primaire ou secondaire, mais autorise le flux de données dans plusieurs directions. Pour plus d'informations sur la réplication logique, voir Chapitre 31. Au travers de l'interface de décodage logique (Chapitre 49), les extensions tierces peuvent aussi fournir des fonctionnalités similaires.

Réplication primaire/secondaire basé sur des triggers

Une architecture de réplication basée sur des triggers canalise habituellement les requêtes de modification de données vers un serveur primaire précis. Opérant table par table, le serveur primaire envoie alors les données modifiées (généralement) de façon asynchrone vers les serveurs secondaires. Les serveurs secondaires peuvent répondre aux requêtes qu'ils reçoivent alors que le serveur primaire est fonctionnel. Ils peuvent parfois permettre quelques modifications de données localement ou même une activité en écriture. Cette forme de réplication est souvent utilisée pour gérer de grosses requêtes analytiques ou de type *Data Warehouse*.

Slony-I est un exemple de ce type de réplication, avec une granularité par table et un support des secondaires multiples. Comme il met à jour le serveur secondaire de façon asynchrone (par lots), il existe une possibilité de perte de données pendant un *failover*.

Middleware de réplication basé sur le SQL

Avec les *middleware* de réplication basés sur le SQL, un programme intercepte chaque requête SQL et l'envoie à un ou tous les serveurs. Chaque serveur opère indépendamment. Les requêtes en lecture/écriture doivent être envoyées à tous les serveurs pour que chaque serveur reçoive les modifications. Les requêtes en lecture seule ne peuvent être envoyées qu'à un seul serveur, ce qui permet de distribuer la charge de lecture.

Si les requêtes sont envoyées sans modification, les fonctions comme `random()`, `CURRENT_TIMESTAMP` ainsi que les séquences ont des valeurs différentes sur les différents serveurs. Cela survient parce que chaque serveur opère indépendamment alors que les requêtes SQL sont diffusées (et non les données modifiées). Si cette solution est inacceptable, le *middleware* ou l'application doivent déterminer ces valeurs à partir d'une seule source, et les utiliser dans les requêtes d'écriture. Il est impératif que toute transaction soit validée ou annulée sur tous les serveurs, éventuellement par validation en deux phases (PREPARE TRANSACTION et COMMIT PREPARED). Pgpool-II et Continuent Tungsten sont des exemples de ce type de réplication.

Réplication asynchrone multi-primaires

Pour les serveurs qui ne sont pas connectés en permanence ou qui ont des liens de communication lents, comme les ordinateurs portables ou les serveurs distants, conserver la cohérence des données entre les serveurs est un challenge. L'utilisation de la réplication asynchrone multi-primaires permet à chaque serveur de fonctionner indépendamment. Il communique alors périodiquement avec les autres serveurs pour identifier les transactions conflictuelles. La gestion des conflits est alors confiée aux utilisateurs ou à un système de règles de résolution. Bucardo est un exemple de ce type de réplication.

Réplication synchrone multi-primaires

Dans les réplications synchrones multi-primaires, tous les serveurs acceptent les requêtes en écriture. Les données modifiées sont transmises du serveur d'origine à tous les autres serveurs avant toute validation de transaction.

Une activité importante en écriture peut être la cause d'un verrouillage excessif et de délai dans la validation des transactions, ce qui peut conduire à un effondrement des performances. Dans les faits, les performances en écriture sont souvent pis que celles d'un simple serveur.

Tous les serveurs acceptent les requêtes en lecture.

Certaines implantations utilisent les disques partagés pour réduire la surcharge de communication.

Les performances de la réplication synchrone multi-primaires sont meilleures lorsque les opérations de lecture représentent l'essentiel de la charge, alors que son gros avantage est l'acceptation des requêtes d'écriture par tous les serveurs -- il n'est pas nécessaire de répartir la charge entre les serveurs primaires et secondaires et, parce que les modifications de données sont envoyées d'un serveur à l'autre, les fonctions non déterministes, comme `random()`, ne posent aucun problème.

PostgreSQL n'offre pas ce type de réplication, mais la validation en deux phases de PostgreSQL (PREPARE TRANSACTION et COMMIT PREPARED) autorise son intégration dans une application ou un *middleware*.

La Tableau 27.1 résume les possibilités des différentes solutions listées plus-haut.

Tableau 27.1. Matrice de fonctionnalités : haute disponibilité, répartition de charge et réplication

Fonctionnalité	Disques partagés	Répl. par système de fichiers	Envoi des journaux de transactions	Répl. logique	Répl. par triggers	*Middleware* de Répl. SQL	Répl. asynch. MM	Répl. synch. MM
Exemple populaires	NAS	DRBD	répl. en flux interne	répl. logique interne, pglogical	Londiste, Slony	pgpool-II	Bucardo	
Méthode de comm.	Disque partagé	Blocs disque	WAL	décodage logique	Lignes de tables	SQL	Lignes de tables	Lignes de tables et verrous de ligne
Ne requiert aucun matériel spécial		•	•	•	•	•	•	•
Autorise plusieurs serveurs primaires				•		•	•	•
Pas de surcharge sur le serveur primaire	•		•	•		•		
Pas d'attente entre serveurs	•		with sync off	with sync off	•		•	
Pas de perte de données en cas de panne du primaire	•	•	with sync on	with sync on		•		•
Les secondaires acceptent les requêtes en lecture seule			avec un hot standby	•	•	•	•	•
Granularité de niveau table				•	•		•	•
Ne nécessite pas de	•	•	•		•	•		•

Fonctionnalité	Disques partagés	Répl. par système de fichiers	Envoi des journaux de transactions	Répl. logique	Répl. par triggers	*Middleware* de Répl. SQL	Répl. asynch. MM	Répl. synch. MM
résolution de conflit								

Certaines solutions n'entrent pas dans les catégories ci-dessus :

Partitionnement de données

Le partitionnement des données divise les tables en ensembles de données. Chaque ensemble ne peut être modifié que par un seul serveur. Les données peuvent ainsi être partitionnées par bureau, Londres et Paris, par exemple, avec un serveur dans chaque bureau. Si certaines requêtes doivent combiner des données de Londres et Paris, il est possible d'utiliser une application qui requête les deux serveurs ou d'implanter une réplication primaire/secondaire pour conserver sur chaque serveur une copie en lecture seule des données de l'autre bureau.

Exécution de requêtes en parallèle sur plusieurs serveurs

La plupart des solutions ci-dessus permettent à plusieurs serveurs de répondre à des requêtes multiples, mais aucune ne permet à une seule requête d'être exécutée sur plusieurs serveurs pour se terminer plus rapidement. Cette solution autorisent plusieurs serveurs à travailler ensemble sur une seule requête. Ceci s'accomplit habituellement en répartissant les données entre les serveurs, chaque serveur exécutant une partie de la requête pour renvoyer les résultats à un serveur central qui les combine et les renvoie à l'utilisateur. Cela peut également se faire en utilisant PL/Proxy.

Il faut aussi noter que puisque PostgreSQL est un outil libre et facilement extensible, un certain nombre de sociétés se sont basées sur PostgreSQL pour créer leurs solutions propriétaires avec des possibilités spécifiques de *failover*, réplication ou répartition de charge. Cela ne sera toutefois pas abordé ici.

27.2. Serveurs secondaires par transfert de journaux

L'archivage en continu peut être utilisé pour créer une configuration de cluster en *haute disponibilité* (HA) avec un ou plusieurs *serveurs secondaires* prêts à prendre la main sur les opérations si le serveur primaire fait défaut. Cette fonctionnalité est généralement appelée *warm standby* ou *log shipping*.

Les serveurs primaire et secondaire travaillent de concert pour fournir cette fonctionnalité, bien que les serveurs ne soient que faiblement couplés. Le serveur primaire opère en mode d'archivage en continu, tandis que le serveur secondaire opère en mode de récupération en continu, en lisant les fichiers WAL provenant du primaire. Aucune modification des tables de la base ne sont requises pour activer cette fonctionnalité, elle entraîne donc moins de travail d'administration par rapport à d'autres solutions de réplication. Cette configuration a aussi un impact relativement faible sur les performances du serveur primaire.

Déplacer directement des enregistrements de WAL d'un serveur de bases de données à un autre est habituellement appelé log shipping. PostgreSQL implémente le log shipping par fichier, ce qui signifie que les enregistrements de WAL sont transférés un fichier (segment de WAL) à la fois. Les fichiers de WAL (16Mo) peuvent être transférés facilement et de façon peu coûteuse sur n'importe quelle distance, que ce soit sur un système adjacent, un autre système sur le même site, ou un autre système à l'autre bout du globe. La bande passante requise pour cette technique varie en fonction du débit de transactions du serveur primaire. La technique de la réplication en flux permet d'optimiser cette bande passante en utilisant une granularité plus fine que le log shipping par fichier. Pour cela, les modifications apportées au journal de transactions sont traitées sous forme de flux au travers d'une connexion réseau (voir Section 27.2.5).

Il convient de noter que le log shipping est asynchrone, c'est à dire que les enregistrements de WAL sont transférés après que la transaction ait été validée. Par conséquent, il y a un laps de temps pendant lequel une perte de données pourrait se produire si le serveur primaire subissait un incident majeur; les transactions pas encore transférées seront perdues. La taille de la fenêtre de temps de perte de données peut être réduite par l'utilisation du paramètre `archive_timeout`, qui peut être abaissé à des valeurs de quelques secondes. Toutefois, un paramètre si bas augmentera de façon considérable la bande passante nécessaire pour le transfert de fichiers. L'utilisation de la technique de la réplication en flux (voir Section 27.2.5) permet de diminuer la taille de la fenêtre de temps de perte de données.

La performance de la récupération est suffisamment bonne pour que le serveur secondaire ne soit en général qu'à quelques instants de la pleine disponibilité à partir du moment où il aura été activé. C'est pour cette raison que cette configuration de haute disponibilité est appelée warm standby. Restaurer un serveur d'une base de sauvegarde archivée, puis appliquer tous les journaux prendra largement plus de temps, ce qui fait que cette technique est une solution de 'disaster recovery' (reprise après sinistre), pas de haute disponibilité. Un serveur secondaire peut aussi être utilisé pour des requêtes en lecture seule, dans quel cas il est appelé un serveur Hot Standby. Voir Section 27.4 pour plus d'information.

27.2.1. Préparatifs

Il est habituellement préférable de créer les serveurs primaire et secondaires de façon à ce qu'ils soient aussi similaires que possible, au moins du point de vue du serveur de bases de données. En particulier, les chemins associés avec les tablespaces seront passés d'un nœud à l'autre sans conversion, ce qui implique que les serveurs primaire et secondaires doivent avoir les mêmes chemins de montage pour les tablespaces si cette fonctionnalité est utilisée. Gardez en tête que si CREATE TABLESPACE est exécuté sur le primaire, tout nouveau point de montage nécessaire pour cela doit être créé sur le primaire et tous les secondaires avant que la commande ne soit exécutée. Le matériel n'a pas besoin d'être exactement le même, mais l'expérience monte que maintenir deux systèmes identiques est plus facile que maintenir deux différents sur la durée de l'application et du système. Quoi qu'il en soit, l'architecture hardware doit être la même -- répliquer par exemple d'un serveur 32 bits vers un 64 bits ne fonctionnera pas.

De manière générale, le log shipping entre serveurs exécutant des versions majeures différentes de PostgreSQL est impossible. La politique du PostgreSQL Global Development Group est de ne pas réaliser de changement sur les formats disques lors des mises à jour mineures, il est par conséquent probable que l'exécution de versions mineures différentes sur le primaire et le secondaire fonctionne correctement. Toutefois, il n'y a aucune garantie formelle de cela et il est fortement conseillé de garder le serveur primaire et celui de secondaire au même niveau de version autant que faire se peut. Lors d'une mise à jour vers une nouvelle version mineure, la politique la plus sûre est de mettre à jour les serveurs secondaires d'abord -- une nouvelle version mineure est davantage susceptible de lire les enregistrements WAL d'une ancienne version mineure que l'inverse.

27.2.2. Fonctionnement du serveur secondaire

Un serveur entre en mode standby si un fichier `standby.signal` existe dans le répertoire de données du serveur.

En mode standby, le serveur applique continuellement les WAL reçus du serveur primaire. Le serveur secondaire peut lire les WAL d'une archive WAL (voir restore_command) ou directement du primaire via une connexion TCP (réplication en flux). Le serveur secondaire essaiera aussi de restaurer tout WAL trouvé dans le répertoire `pg_wal` de l'instance du secondaire. Cela se produit habituellement après un redémarrage de serveur, quand le secondaire rejoue à nouveau les WAL qui ont été reçu du primaire avant le redémarrage, mais vous pouvez aussi copier manuellement des fichiers dans `pg_wal` à tout moment pour qu'ils soient rejoués.

Au démarrage, le serveur secondaire commence par restaurer tous les WAL disponibles à l'endroit où se trouvent les archives, en appelant la `restore_command`. Une fois qu'il a épuisé tous les WAL disponibles à cet endroit et que `restore_command` échoue, il essaie de restaurer tous les WAL disponibles dans le répertoire `pg_wal`. Si cela échoue, et que la réplication en flux a été activée, le

serveur secondaire essaie de se connecter au serveur primaire et de démarrer la réception des WAL depuis le dernier enregistrement valide trouvé dans les archives ou `pg_wal`. Si cela échoue ou que la réplication en flux n'est pas configurée, ou que la connexion est plus tard déconnectée, le secondaire retourne à l'étape 1 et essaie de restaurer le fichier à partir de l'archive à nouveau. Cette boucle de retentatives de l'archive, `pg_wal` et par la réplication en flux continue jusqu'à ce que le serveur soit arrêté ou promu. déclenché.

Le mode standby est quitté et le serveur bascule en mode de fonctionnement normal quand `pg_ctl promote` est exécuté, que `pg_promote()` est appelé ou qu'un fichier de trigger est trouvé (`promote_trigger_file`). Avant de basculer, tout WAL immédiatement disponible dans l'archive ou le `pg_wal` sera restauré, mais aucune tentative ne sera faite pour se connecter au primaire.

27.2.3. Préparer le primaire pour les serveurs secondaires

Mettez en place un archivage en continu sur le primaire vers un répertoire d'archivage accessible depuis le secondaire, comme décrit dans Section 26.3. La destination d'archivage devrait être accessible du secondaire même quand le primaire est inaccessible, c'est à dire qu'il devrait se trouver sur le serveur secondaire lui-même ou un autre serveur de confiance, pas sur le serveur primaire.

Si vous voulez utiliser la réplication en flux, mettez en place l'authentification sur le serveur primaire pour autoriser les connexions de réplication à partir du (ou des) serveur(s) secondaire(s) ; c'est-à-dire, créez un rôle et mettez en place une ou des entrées appropriées dans `pg_hba.conf` avec le champ database positionné à `replication`. Vérifiez aussi que `max_wal_senders` est positionné à une valeur suffisamment grande dans le fichier de configuration du serveur primaire. Si des slots de réplication seront utilisés, il faut s'assurer que `max_replication_slots` est également positionné à une valeur suffisamment grande.

Effectuez une sauvegarde de base comme décrit dans Section 26.3.2 pour initialiser le serveur secondaire.

27.2.4. Paramétrer un serveur secondaire

Pour paramétrer le serveur secondaire, restaurez la sauvegarde de base effectué sur le serveur primaire (voir Section 26.3.4). Créez un fichier `standby.signal` dans le répertoire de données de l'instance du secondaire. Positionnez restore_command à une simple commande qui recopie les fichiers de l'archive de WAL. Si vous comptez disposer de plusieurs serveurs secondaires pour mettre en œuvre de la haute disponibilité, assurez-vous que `recovery_target_timeline` est configuré à `latest` (la valeur par défaut), pour indiquer que le serveur secondaire devra prendre en compte la ligne temporelle définie lors de la bascule à un autre serveur secondaire.

> **Note**
>
> restore_command devrait retourner immédiatement si le fichier n'existe pas ; le serveur essaiera la commande à nouveau si nécessaire.

Si vous souhaitez utiliser la réplication en flux, renseignez primary_conninfo avec une chaîne de connexion libpq, contenant le nom d'hôte (ou l'adresse IP) et tout détail supplémentaire nécessaire pour se connecter au serveur primaire. Si le primaire a besoin d'un mot de passe pour l'authentification, le mot de passe doit aussi être spécifié dans primary_conninfo.

Si vous mettez en place le serveur secondaire pour des besoins de haute disponibilité, mettez en place l'archivage de WAL, les connexions et l'authentification à l'identique du serveur primaire, parce que le serveur secondaire fonctionnera comme un serveur primaire après la bascule.

Si vous utilisez une archive WAL, sa taille peut être réduite en utilisant l'option archive_cleanup_command pour supprimer les fichiers qui ne sont plus nécessaires au

serveur secondaire. L'outil pg_archivecleanup est conçu spécifiquement pour être utilisé avec `archive_cleanup_command` dans des configurations typiques de secondaire, voir pg_archivecleanup. Notez toutefois que si vous utilisez l'archive à des fins de sauvegarde, vous avez besoin de garder les fichiers nécessaires pour restaurer à partir de votre dernière sauvegarde de base, même si ces fichiers ne sont plus nécessaires pour le serveur secondaire.

Voici un simple exemple :

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo
 password=foopass options=''-c wal_sender_timeout=5000'''
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

Vous pouvez avoir n'importe quel nombre de serveurs secondaires, mais si vous utilisez la réplication en flux, assurez vous d'avoir positionné `max_wal_senders` suffisamment haut sur le primaire pour leur permettre de se connecter simultanément.

27.2.5. Réplication en flux

La réplication en flux permet à un serveur secondaire de rester plus à jour qu'il n'est possible avec l'envoi de journaux par fichiers. Le standby se connecte au primaire, qui envoie au standby les enregistrements de WAL dès qu'ils sont générés, sans attendre qu'un fichier de WAL soit rempli.

La réplication en flux est asynchrone par défaut (voir Section 27.2.8), auquel cas il y a un petit délai entre la validation d'une transaction sur le primaire et le moment où les changements sont visibles sur le standby. Le délai est toutefois beaucoup plus petit qu'avec l'envoi de fichiers, habituellement en dessous d'une seconde en partant de l'hypothèse que le standby est suffisamment puissant pour supporter la charge. Avec la réplication en flux, `archive_timeout` n'est pas nécessaire pour réduire la fenêtre de perte de données.

Si vous utilisez la réplication en flux sans archivage en continu des fichiers, le serveur primaire pourrait recycler de vieux journaux de transactions avant que le serveur en standby ne les ait reçus. Si cela arrive, le serveur en standby devra être recréé à partir d'une nouvelle sauvegarde de l'instance. Vous pouvez éviter cela en positionnant `wal_keep_size` à une valeur suffisamment grande pour s'assurer que les journaux de transactions ne sont pas recyclés trop tôt, ou en configurant un slot de réplication pour le serveur en standby. Si un archivage des journaux de transactions est en place, et que les fichiers archivés sont disponibles depuis le serveur en standby, cette solution n'est pas nécessaire, puisque le serveur en standby peut toujours utiliser les fichiers archivés pour rattraper son retard, sous réserve que suffisamment de fichiers soient conservés.

Pour utiliser la réplication en flux, mettez en place un serveur secondaire en mode fichier comme décrit dans Section 27.2. L'étape qui transforme un secondaire en mode fichier en secondaire dans une réplication en flux est de faire pointer `primary_conninfo` vers le serveur primaire. Positionnez listen_addresses et les options d'authentification (voir `pg_hba.conf`) sur le primaire pour que le serveur secondaire puisse se connecter à la pseudo-base `replication` sur le serveur primaire (voir Section 27.2.5.1).

Sur les systèmes qui supportent l'option de keepalive sur les sockets, positionner tcp_keepalives_idle, tcp_keepalives_interval et tcp_keepalives_count aide le primaire à reconnaître rapidement une connexion interrompue.

Positionnez le nombre maximum de connexions concurrentes à partir des serveurs secondaires (voir max_wal_senders pour les détails).

Quand le secondaire est démarré et que `primary_conninfo` est positionné correctement, le secondaire se connectera au primaire après avoir rejoué tous les fichiers WAL disponibles dans l'archive. Si la connexion est établie avec succès, vous verrez un processus `walreceiver` sur le secondaire, et un processus `walsender` correspondant sur le primaire.

27.2.5.1. Authentification

Il est très important que les droits d'accès pour la réplication soient paramétrés pour que seuls les utilisateurs de confiance puissent lire le flux WAL, parce qu'il est facile d'en extraire des informations privilégiées. Les serveurs secondaires doivent s'authentifier au serveur primaire en tant qu'un compte disposant de l'attribut `REPLICATION` ou `SUPERUSER`. Il est recommandé de créer un compte utilisateur dédié pour la réplication. Il doit disposer des attributs `REPLICATION` et `LOGIN`. Alors que l'attribut `REPLICATION` donne beaucoup de droits, il ne permet pas à l'utilisateur de modifier de données sur le serveur primaire, contrairement à l'attribut `SUPERUSER`.

L'authentification cliente pour la réplication est contrôlée par un enregistrement de `pg_hba.conf` spécifiant `replication` dans le champ *`database`*. Par exemple, si le secondaire s'exécute sur un hôte d'IP `192.168.1.100` et que le nom de l'utilisateur pour la réplication est `foo`, l'administrateur peut ajouter la ligne suivante au fichier `pg_hba.conf` sur le primaire:

```
# Autoriser l'utilisateur "foo" de l'hôte 192.168.1.100 à se
 connecter au primaire
# en tant que secondaire si le mot de passe de l'utilisateur est
 correctement fourni
#
# TYPE  DATABASE        USER            ADDRESS
 METHOD
host    replication     foo             192.168.1.100/32        md5
```

Le nom d'hôte et le numéro de port du primaire, le nom d'utilisateur de la connexion, et le mot de passe sont spécifiés dans le paramètre primary_conninfo. Le mot de passe peut aussi être enregistré dans le fichier `~/.pgpass` sur le serveur en attente (en précisant `replication` dans le champ *`database`*). Par exemple, si le primaire s'exécute sur l'hôte d'IP `192.168.1.50`, port `5432`, que le nom de l'utilisateur pour la réplication est `foo`, et que le mot de passe est `foopass`, l'administrateur peut ajouter la ligne suivante au fichier `postgresql.conf` sur le secondaire :

```
# Le secondaire se connecte au primaire qui s'exécute sur l'hôte
 192.168.1.50
# et port 5432 en tant qu'utilisateur "foo" dont le mot de passe
 est "foopass"
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo
 password=foopass'
```

27.2.5.2. Supervision

Un important indicateur de santé de la réplication en flux est le nombre d'enregistrements générés sur le primaire, mais pas encore appliqués sur le secondaire. Vous pouvez calculer ce retard en comparant le point d'avancement des écritures du WAL sur le primaire avec le dernier point d'avancement reçu par le secondaire. Ils peuvent être récupérés en utilisant `pg_current_wal_lsn` sur le primaire et `pg_last_wal_receive_lsn` sur le secondaire (voir Tableau 9.89 et Tableau 9.90 pour plus de détails). Le point d'avancement de la réception dans le secondaire est aussi affiché dans le statut du processus de réception des WAL (wal receiver), affiché par la commande `ps` (voir Section 28.1 pour plus de détails).

Vous pouvez obtenir la liste des processus émetteurs de WAL au moyen de la vue `pg_stat_replication` D'importantes différences entre les champs `pg_current_wal_lsn` et `sent_lsn` peuvent indiquer que le serveur primaire est en surcharge, tandis que des différences entre `sent_lsn` et `pg_last_wal_receive_lsn` sur le secondaire peuvent soit indiquer une latence réseau importante, soit que le secondaire est surchargé.

Sur un secondaire en hot standby, le statut du processus wal receiver est récupérable avec la vue `pg_stat_wal_receiver`. Une différence importante entre `pg_last_wal_replay_lsn` et la colonne `flushed_lsn` de la vue indique que les WAL sont reçus plus rapidement qu'ils ne sont rejoués.

27.2.6. Slots de réplication

Les slots de réplication fournissent une manière automatisée de s'assurer que le primaire ne supprime pas les journaux de transactions avant qu'ils n'aient été reçus par tous les serveurs secondaires, et que le serveur primaire ne supprime pas des lignes qui pourraient causer un conflit de restauration même si le serveur secondaire est déconnecté.

Au lieu d'utiliser des slots de réplication, il est possible d'empêcher la suppression des anciens journaux de transactions en utilisant wal_keep_size, ou en les stockant dans un répertoire d'archive en utilisant archive_command ou archive_library. Cependant, ces méthodes ont souvent pour résultat le stockage de plus de journaux de transactions que nécessaire, alors que les slots de réplication ne conservent que le nombre nécessaire de journaux de transactions. D'un autre côté, les slots de réplication peuvent retenir tellement de journaux de transactions qu'ils rempliraient l'espace disque alloué à `pg_wal` ; max_slot_wal_keep_size limite la taille des journaux de transactions à retenir par les slots de réplication.

De la même manière, hot_standby et vacuum_defer_cleanup_age fournissent des protections contre la suppression de lignes utiles par vacuum, mais le premier paramètre n'offre aucune protection durant la période pendant laquelle le serveur secondaire n'est pas connecté, et le second nécessite souvent d'être positionné à une grande valeur pour fournir une protection adéquate. Les slots de réplication surmontent ces désavantages.

27.2.6.1. Requêter et manipuler des slots de réplication

Chaque slot de réplication à un nom, qui peut contenir des lettres en minuscule, des nombres ou un tiret bas.

Les slots de réplication existants et leur états peuvent être vus dans la vue `pg_replication_slots`.

Les slots de réplication peuvent être créés et supprimés soit via le protocole de réplication en flux (voir Section 55.4) soit via des fonctions SQL (voir Section 9.27.6).

27.2.6.2. Exemple de configuration

Il est possible de créer un slot de réplication ainsi :

```
postgres=# SELECT * FROM
 pg_create_physical_replication_slot('node_a_slot');
  slot_name  | lsn
-------------+-----
 node_a_slot |

postgres=# SELECT slot_name, slot_type, active FROM
 pg_replication_slots;
  slot_name  | slot_type | active
-------------+-----------+--------
 node_a_slot | physical  | f
```

Pour configurer le serveur secondaire pour utiliser ce slot, `primary_slot_name` devrait être configuré sur le secondaire. Voici un exemple simple :

```
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo
 password=foopass'
primary_slot_name = 'node_a_slot'
```

27.2.7. Réplication en cascade

La fonctionnalité de réplication en cascade permet à un serveur secondaire d'accepter les connexions de réplication et d'envoyer un flux d'enregistrements de journaux de transactions à d'autres secondaires, agissant ainsi comme un relais. C'est généralement utilisé pour réduire le nombre de connexions directes au primaire et minimiser ainsi l'utilisation de bande passante entre sites distants.

Un serveur secondaire agissant à la fois comme un receveur et comme un émetteur est connu sous le nom de secondaire en cascade (*cascading standby*). Les secondaires qui sont plus proches du serveur primaire sont connus sous le nom de serveurs *upstream* alors que les serveurs secondaires en bout de chaîne sont des serveurs *downstream*. La réplication en cascade ne pose pas de limites sur le nombre ou l'arrangement des serveurs *downstream*. Chaque secondaire se connecte à un seul serveur *upstream*, qui finit par arriver à un seul serveur primaire.

Un secondaire en cascade envoie non seulement les enregistrements reçus de journaux de transactions mais aussi ceux restaurés des archives. Donc, même si la connexion de réplication d'une connexion upstream est rompue, la réplication en flux continue vers le serveur downstream tant que de nouveaux enregistrements de journaux de transactions sont disponibles.

La réplication en cascade est actuellement asynchrone. La réplication synchrone (voir Section 27.2.8) n'a aucun effet sur la réplication en cascade.

Les messages en retour des serveurs Hot Standby se propagent vers les serveurs upstream, quelle que soit la configuration de la réplication en cascade.

Si un serveur secondaire upstream est promu pour devenir le nouveau serveur primaire, les serveurs downstream continueront à recevoir le flux de réplication du nouveau primaire si le paramètre `recovery_target_timeline` est configuré à `'latest'` (valeur par défaut).

Pour utiliser la réplication en cascade, configurez le secondaire en cascade de façon à ce qu'il accepte les connexions de réplication (configurez max_wal_senders et hot_standby, ainsi que l'authentification). Vous aurez aussi besoin de configurer la variable `primary_conninfo` dans le secondaire downstream pour qu'elle pointe vers le secondaire en cascade.

27.2.8. Réplication synchrone

La réplication en flux mise en œuvre par PostgreSQL est asynchrone par défaut. Si le serveur primaire est hors-service, les transactions produites alors peuvent ne pas avoir été répliquées sur le serveur secondaire, impliquant une perte de données. La quantité de données perdues est proportionnelle au délai de réplication au moment de la bascule.

La réplication synchrone permet de confirmer que tous les changements effectués par une transaction ont bien été transférées vers un ou plusieurs serveurs secondaires synchrones. Cette propriété étend le niveau de robustesse standard offert par un commit. En science informatique, ce niveau de protection est appelé réplication à deux états (*2-safe replication*) et *group-1-safe* (*group-safe* et *1-safe*) quand `synchronous_commit` est configuré à la valeur `remote_write`.

Lorsque la réplication synchrone est utilisée, chaque validation portant sur une écriture va nécessiter d'attendre la confirmation de l'écriture de cette validation dans les journaux de transactions sur les disques du serveur primaire et des serveurs secondaires. Le seul moyen possible pour que des données soient perdues est que les serveur primaire et secondaire soient hors service au même moment. Ce mécanisme permet d'assurer un niveau plus élevé de robustesse, en admettant que l'administrateur système ait pris garde à l'emplacement et à la gestion de ces deux serveurs. Attendre après la

confirmation de l'écriture augmente la confiance que l'utilisateur pourra avoir sur la conservation des modifications dans le cas où un serveur serait hors service mais il augmente aussi en conséquence le temps de réponse de chaque requête. Le temps minimum d'attente est celui de l'aller-retour entre les serveurs primaire et secondaire.

Les transactions où seule une lecture est effectuée ou qui consistent à annuler une transaction ne nécessitent pas d'attendre les serveurs secondaires. Les validations concernant les transactions imbriquées ne nécessitent pas non plus d'attendre la réponse des serveurs secondaires, cela n'affecte en fait que les validations principales. De longues opérations comme le chargement de données ou la création d'index n'attendent pas le commit final pour synchroniser les données. Toutes les actions de validation en deux étapes nécessitent d'attendre la validation du secondaire, incluant autant l'opération de préparation que l'opération de validation.

Un secondaire synchrone peut être un secondaire de réplication physique ou un abonné d'une réplication logique. Il peut aussi être tout autre consommateur de flux de réplication physique ou logique qui sait comment gérer les messages de retour appropriés. En plus des systèmes intégrés de réplication physique et logique, cela inclut le support des outils tels que `pg_receivewal` et `pg_recvlogical` ainsi que les systèmes de réplication et outils personnalisés tiers.

27.2.8.1. Configuration de base

Une fois la réplication en flux configurée, la configuration de la réplication synchrone ne demande qu'une unique étape de configuration supplémentaire : la variable synchronous_standby_names doit être définie à une valeur non vide. La variable `synchronous_commit` doit aussi être définie à `on`, mais comme il s'agit d'une valeur par défaut, il n'est pas nécessaire de la modifier. (Voir Section 20.5.1 et Section 20.6.2.) Cette configuration va entraîner l'attente de la confirmation de l'écriture permanente de chaque validation sur le serveur secondaire. La variable `synchronous_commit` peut être définie soit par des utilisateurs, soit par le fichier de configuration pour des utilisateurs ou des bases de données fixées, soit dynamiquement par des applications, pour contrôler la robustesse des échanges transactionnels.

Suite à l'enregistrement sur disque d'une validation sur le serveur primaire, l'enregistrement WAL est envoyé au serveur secondaire. Le serveur secondaire retourne une réponse à chaque fois qu'un nouveau lot de données WAL est écrit sur disque, à moins que le paramètre `wal_receiver_status_interval` soit défini à zéro sur le serveur secondaire. Dans le cas où le paramètre `synchronous_commit` est configuré à la valeur `remote_apply`, le serveur secondaire envoie des messages de réponse quand l'enregistrement de validation (commit) est rejoué, rendant la transaction visible. Si le serveur secondaire est configuré en serveur synchrone d'après la configuration du paramètre `synchronous_standby_names` sur le primaire, le message de réponse provenant du secondaire sera considéré parmi ceux des autres serveurs secondaires pour décider du moment de libération des transactions attendant la confirmation de la bonne réception de l'enregistrement de commit. Ces paramètres permettent à l'administrateur de spécifier quels serveurs secondaires suivront un comportement synchrone. Remarquez ici que la configuration de la réplication synchrone se situe sur le serveur primaire. Les serveurs secondaires nommés doivent être directement connectés au primaire ; le primaire ne connaît rien des serveurs secondaires utilisant la réplication en cascade.

Configurer `synchronous_commit` à `remote_write` fera que chaque COMMIT attendra la confirmation de la réception en mémoire de l'enregistrement du COMMIT par le secondaire et son écriture via la système d'exploitation, sans que les données du cache du système ne soient vidées sur disque au niveau du serveur secondaire. Cette configuration fournit une garantie moindre de durabilité que la configuration `on` : le secondaire peut perdre les données dans le cas d'un crash du système d'exploitation, mais pas dans le cas du crash de PostgreSQL. Cependant, il s'agit d'une configuration utile en pratique car il diminue le temps de réponse pour la transaction. Des pertes de données ne peuvent survenir que si le serveur primaire et le secondaire tombent en même temps et que la base de données du primaire est corrompue.

Configurer `synchronous_commit` à `remote_apply` fera en sorte que chaque commit devra attendre le retour des secondaires synchrones actuels indiquant qu'ils ont bien rejoué la transaction,

la rendant visible aux requêtes des utilisateurs. Dans des cas simples, ceci permet une répartition de charge sans incohérence.

Habituellement, un signal d'arrêt rapide (*fast shutdown*) annule les transactions en cours sur tous les processus serveur. Cependant, dans le cas de la réplication asynchrone, le serveur n'effectuera pas un arrêt complet avant que chaque enregistrement WAL ne soit transféré aux serveurs secondaires connectés.

27.2.8.2. Multiple secondaires synchrones

La réplication synchrone supporte un ou plusieurs serveurs secondaires synchrones. Les transactions attendront que tous les serveurs secondaires considérés synchrones confirment la réception de leurs données. Le nombre de secondaires dont les transactions doivent attendre la réponse est indiqué dans le paramètre `synchronous_standby_names`. Ce paramètre indique aussi une liste des noms des serveurs secondaires ou l'emploi de la méthode (`FIRST` ou `ANY`) pour choisir sur quel serveur synchrone basculer parmi l'ensemble des serveurs listés.

La méthode `FIRST` définit une réplication synchrone priorisée : elle temporise la validation de la transaction jusqu'à ce que les enregistrements WAL soient répliqués en fonction de la priorité définie des serveurs secondaires dans une liste ordonnée. Le serveur secondaire dont le nom apparaît en premier sur la liste est prioritaire et est celui qui est considéré comme synchrone. Les serveurs secondaires suivants sont considérés comme un/des serveurs secondaires synchrones potentiels. Si le premier serveur synchrone venait à tomber, il serait immédiatement remplacé par le serveur secondaire prioritaire suivant.

Voici un exemple de configuration de `synchronous_standby_names` pour la réplication synchrone priorisée :

```
synchronous_standby_names = 'FIRST 2 (s1, s2, s3)'
```

Dans cet exemple, si les quatre serveurs secondaires `s1`, `s2`, `s3` et `s4` sont fonctionnels et en cours d'exécution, les deux serveurs `s1` et `s2` seront choisis comme secondaires synchrones car leurs noms apparaissent en premier dans la liste des serveurs secondaires. `s3` est un serveur secondaire synchrone potentiel et prendra le rôle d'un secondaire synchrone si `s1` ou `s2` tombe. `s4` est un secondaire asynchrone et son nom n'est pas dans la liste.

La méthode `ANY` définit une réplication synchrone basée sur un quorum : elle temporise la validation de la transaction jusqu'à ce que les enregistrements WAL soient répliqués *au moins* sur le nombre de serveurs définis dans la liste.

Voici un exemple de configuration du paramètre `synchronous_standby_names` pour la réplication synchrone avec un quorum :

```
synchronous_standby_names = 'ANY 2 (s1, s2, s3)'
```

Dans cet exemple, sur quatre serveurs secondaires démarrés `s1`, `s2`,`s3` et `s4`, pour obtenir la validation d'une transaction, le serveur primaire attendra la réponse d'au minimum deux secondaires parmi `s1`, `s2` et `s3`. `s4` est un secondaire asynchrone et son nom n'est pas dans la liste.

L'état de synchronicité des serveurs secondaires peut être consulté avec la vue `pg_stat_replication`.

27.2.8.3. S'organiser pour obtenir de bonnes performances

La réplication synchrone nécessite souvent d'organiser avec une grande attention les serveurs secondaires pour apporter un bon niveau de performances aux applications. Les phases d'attente

d'écriture n'utilisent pas les ressources systèmes, mais les verrous transactionnels restent positionnés jusqu'à ce que le transfert vers les serveurs secondaires soit confirmé. En conséquence, une utilisation non avertie de la réplication synchrone aura pour impact une baisse des performances de la base de données d'une application due à l'augmentation des temps de réponses et à un moins bon support de la charge.

PostgreSQL permet aux développeurs d'application de spécifier le niveau de robustesse à employer pour la réplication. Cela peut être spécifié pour le système entier, mais aussi pour des utilisateurs ou des connexions spécifiques, ou encore pour des transactions individuelles.

Par exemple, une répartition de charge pour une application pourrait être constituée de : 10 % de modifications concernant des articles de clients importants, et 90 % de modifications de moindre importance et qui ne devraient pas avoir d'impact sur le métier si elles venaient à être perdues, comme des dialogues de messagerie entre utilisateurs.

Les options de réplication synchrone spécifiées par une application (sur le serveur primaire) permettent de n'utiliser la réplication synchrone que pour les modifications les plus importantes, sans affecter les performances sur la plus grosse partie des traitements. Les options modifiables par les applications sont un outil important permettant d'apporter les bénéfices de la réplication synchrone aux applications nécessitant de la haute performance.

Il est conseillé de disposer d'une bande passante réseau supérieure à la quantité de données WAL générées.

27.2.8.4. S'organiser pour la haute disponibilité

`synchronous_standby_names` indique le nombre et les noms des serveurs secondaires synchrones pour lesquels les validations de transactions effectuées lorsque `synchronous_commit` est configurée à `on`, `remote_apply` ou `remote_write`, attendront leur réponse. Ces validations de transactions pourraient ne jamais se terminer si un des secondaires synchrones s'arrêtait brutalement.

La meilleure solution pour la haute disponibilité est de s'assurer que vous conservez autant de serveurs secondaires synchrones que demandés. Ceci se fait en nommant plusieurs secondaires synchrones potentiels avec `synchronous_standby_names`.

Dans la réplication synchrone dite priorisée, les serveurs secondaires dont les noms apparaissent en premier seront utilisé comme secondaires synchrones. Les secondaires définis ensuite prendront la place de serveur synchrone si l'un des serveurs venait à tomber.

Dans la réplication dite de quorum, tous les secondaires spécifiés dans la liste seront utilisés comme des secondaires synchrones potentiels. Même si l'un d'entre eux tombe, les autres secondaires continueront de prétendre au rôle de secondaire synchrone.

Au moment où le premier serveur secondaire s'attache au serveur primaire, il est possible qu'il ne soit pas exactement synchronisé. Cet état est appelé le mode `catchup`. Une fois la différence entre le serveur secondaire et le serveur primaire ramenée à zéro, le mode `streaming` est atteint. La durée du mode catchup peut être longue surtout juste après la création du serveur secondaire. Si le serveur secondaire est arrêté sur cette période, alors la durée du mode catchup sera d'autant plus longue. Le serveur secondaire ne peut devenir un serveur secondaire synchrone que lorsque le mode `streaming` est atteint. L'état de synchronicité des serveurs secondaires peut être consulté avec la vue `pg_stat_replication`.

Si le serveur primaire redémarre alors que des opérations de commit étaient en attente de confirmation, les transactions en attente ne seront réellement enregistrées qu'au moment où la base de données du serveur primaire sera redémarrée. Il n'y a aucun moyen de savoir si tous les serveurs secondaires ont reçu toutes les données WAL nécessaires au moment où le serveur primaire est déclaré hors-service. Des transactions pourraient ne pas être considérées comme sauvegardées sur le serveur secondaire, même si elles l'étaient sur le serveur primaire. La seule garantie offerte dans ce cadre est

que l'application ne recevra pas de confirmation explicite de la réussite d'une opération de validation avant qu'il ne soit sûr que les données WAL sont reçues proprement par tous les serveurs secondaires synchrones.

Si vous ne pouvez vraiment pas conserver autant de serveurs secondaires synchrones que demandés, alors vous devriez diminuer le nombre de secondaires synchrones dont le système doit attendre les réponses aux validations de transactions, en modifiant `synchronous_standby_names` (ou en le désactivant) et en rechargeant le fichier de configuration du serveur primaire.

Si le serveur primaire n'est pas accessible par les serveurs secondaires restants, il est conseillé de basculer vers le meilleur candidat possible parmi ces serveurs secondaires.

S'il est nécessaire de recréer un serveur secondaire alors que des transactions sont en attente de confirmation, prenez garde à ce que les commandes pg_backup_start() et pg_backup_stop() soient exécutées dans un contexte où la variable `synchronous_commit` vaut `off` car, dans le cas contraire, ces requêtes attendront indéfiniment l'apparition de ce serveur secondaire.

27.2.9. Archivage continu côté secondaire

Lorsque l'archivage continu est utilisé sur un secondaire, il existe deux scénarios possibles : soit les archives sont partagées entre le serveur primaire et le serveur secondaire, soit le secondaire peut avoir ses propres archives. Si le serveur possède ses propres archives, en définissant le paramètre `archive_mode` à `always`, le secondaire exécutera la commande d'archivage pour chaque segment de WAL qu'il aura reçu, peu importe qu'il utilise la réplication par les archives ou la réplication en flux. La gestion par archivage partagé peut être faite de la même manière, mais `archive_command` ou `archive_library` doit d'abord tester si le segment de WAL existe, et si le fichier existant contient les mêmes informations. Cela demande plus de précaution lors de la définition de la commande, car elle doit vérifier qu'elle n'écrase pas un fichier existant avec un contenu différent, et doit renvoyer un succès si le même fichier est archivé deux fois. Tout ceci devant être en plus effectué sans concurrence si deux serveurs essaient d'archiver le même fichier au même moment.

Si `archive_mode` est défini à `on`, l'archivage n'est pas actif pendant les modes recovery et standby. Si le serveur secondaire est promu, il commencera à réaliser l'archivage après sa promotion, et il archivera uniquement les fichiers (WAL et historique) qu'il a lui même produit. Pour être sûr d'obtenir un jeu complet d'archives, vous devez vous assurer que tous les fichiers WAL ont été archivés avant qu'ils atteignent le secondaire. C'est implicitement toujours le cas avec un log shipping s'appuyant sur les archives, car le secondaire ne récupère que des informations provenant de ces mêmes fichiers archivés. Ce n'est pas le cas dans le cadre de la réplication en flux. Lorsqu'un serveur n'est pas en mode recovery, il n'y a aucune différence entre les modes `on` et `always`.

27.3. Bascule (*Failover*)

Si le serveur primaire plante alors le serveur secondaire devrait commencer les procédures de failover.

Si le serveur secondaire plante alors il n'est pas nécessaire d'effectuer un failover. Si le serveur secondaire peut être redémarré, même plus tard, alors le processus de récupération peut aussi être redémarré au même moment, en bénéficiant du fait que la récupération sait reprendre où elle en était. Si le serveur secondaire ne peut pas être redémarré, alors une nouvelle instance secondaire complète devrait être créée.

Si le serveur primaire plante, que le serveur secondaire devient le nouveau primaire, et que l'ancien primaire redémarre, vous devez avoir un mécanisme pour informer l'ancien primaire qu'il n'est plus primaire. C'est aussi quelquefois appelé STONITH (Shoot The Other Node In The Head, ou Tire Dans La Tête De L'Autre Nœud), qui est nécessaire pour éviter les situations où les deux systèmes pensent qu'ils sont le primaire, ce qui amènerait de la confusion, et finalement de la perte de données.

Beaucoup de systèmes de failover n'utilisent que deux systèmes, le primaire et le secondaire, connectés par un mécanisme de type ligne de vie (heartbeat) pour vérifier continuellement la connexion entre les

deux et la viabilité du primaire. Il est aussi possible d'utiliser un troisième système (appelé un serveur témoin) pour éviter certains cas de bascule inappropriés, mais la complexité supplémentaire peut ne pas être justifiée à moins d'être mis en place avec suffisamment de précautions et des tests rigoureux.

PostgreSQL ne fournit pas le logiciel système nécessaire pour identifier un incident sur le primaire et notifier le serveur de bases de données secondaire. De nombreux outils de ce genre existent et sont bien intégrés avec les fonctionnalités du système d'exploitation nécessaires à la bascule, telles que la migration d'adresse IP.

Une fois que la bascule vers le secondaire se produit, il n'y a plus qu'un seul serveur en fonctionnement. C'est ce qu'on appelle un état dégradé. L'ancien secondaire est maintenant le primaire, mais l'ancien primaire est arrêté et pourrait rester arrêté. Pour revenir à un fonctionnement normal, un serveur secondaire doit être recréé, soit sur l'ancien système primaire quand il redevient disponible, ou sur un troisième, peut-être nouveau, système. L'utilitaire pg_rewind peut être utilisé pour accélérer ce processus sur de gros clusters. Une fois que ceci est effectué, le primaire et le secondaire peuvent être considérés comme ayant changé de rôle. Certaines personnes choisissent d'utiliser un troisième serveur pour fournir une sauvegarde du nouveau primaire jusqu'à ce que le nouveau serveur secondaire soit recréé, bien que ceci complique visiblement la configuration du système et les procédures d'exploitation.

Par conséquent, basculer du primaire vers le serveur secondaire peut être rapide mais requiert du temps pour re-préparer le cluster de failover. Une bascule régulière du primaire vers le secondaire est utile, car cela permet une période d'interruption de production sur chaque système pour maintenance. Cela vous permet aussi pour vous assurer que votre mécanisme de bascule fonctionnera réellement quand vous en aurez besoin. Il est conseillé que les procédures d'administration soient écrites.

Pour déclencher le failover d'un serveur secondaire en log shipping, exécutez la commande `pg_ctl promote`, lancez la fonction `pg_promote()` ou créez un fichier trigger avec le nom de fichier et le chemin spécifiés par le paramètre `promote_trigger_file`. Si vous comptez utiliser la commande `pg_ctl promote` ou la fonction `pg_promote()` pour effectuer la bascule, la variable `promote_trigger_file` n'est pas nécessaire. S'il s'agit d'ajouter des serveurs qui ne seront utilisés que pour alléger le serveur primaire des requêtes en lecture seule, et non pas pour des considérations de haute disponibilité, il n'est pas nécessaire de les réveiller (*promote*).

27.4. Hot Standby

Hot Standby est le terme utilisé pour décrire la possibilité de se connecter et d'exécuter des requêtes en lecture seule alors que le serveur est en récupération d'archive ou mode standby. C'est utile à la fois pour la réplication et pour restaurer une sauvegarde à un état désiré avec une grande précision. Le terme Hot Standby fait aussi référence à la capacité du serveur à passer de la récupération au fonctionnement normal tandis-que les utilisateurs continuent à exécuter des requêtes et/ou gardent leurs connexions ouvertes.

Exécuter des requêtes en mode hot standby est similaire au fonctionnement normal des requêtes, bien qu'il y ait quelques différences d'utilisation et d'administration notées ci-dessous.

27.4.1. Aperçu pour l'utilisateur

Quand le paramètre hot_standby est configuré à true sur un serveur en attente, le serveur commencera à accepter les connexions une fois que la restauration est parvenue à un état cohérent. Toutes les connexions qui suivront seront des connexions en lecture seule ; même les tables temporaires ne pourront pas être utilisées.

Les données sur le secondaire mettent un certain temps pour arriver du serveur primaire, il y aura donc un délai mesurable entre primaire et secondaire. La même requête exécutée presque simultanément sur le primaire et le secondaire pourrait par conséquent retourner des résultats différents. On dit que la donnée est *cohérente à terme* avec le primaire. Une fois que l'enregistrement de validation (COMMIT) d'une transaction est rejoué sur le serveur en attente, les modifications réalisées par cette transaction

seront visibles par toutes les images de bases obtenues par les transactions en cours sur le serveur en attente. Ces images peuvent être prises au début de chaque requête ou de chaque transaction, suivant le niveau d'isolation des transactions utilisé à ce moment. Pour plus de détails, voir Section 13.2.

Les transactions exécutées pendant la période de restauration sur un serveur en mode hot standby peuvent inclure les commandes suivantes :

- Accès par requête : SELECT, COPY TO
- Commandes de curseur : DECLARE, FETCH, CLOSE
- Paramètres : SHOW, SET, RESET
- Commandes de gestion de transaction :
  - BEGIN, END, ABORT, START TRANSACTION
  - SAVEPOINT, RELEASE, ROLLBACK TO SAVEPOINT
  - Blocs d'EXCEPTION et autres sous-transactions internes
- LOCK TABLE, mais seulement quand explicitement dans un de ces modes: ACCESS SHARE, ROW SHARE ou ROW EXCLUSIVE.
- Plans et ressources : PREPARE, EXECUTE, DEALLOCATE, DISCARD
- Plugins et extensions : LOAD
- UNLISTEN

Les transactions lancées pendant la restauration d'un serveur en hot standby ne se verront jamais affectées un identifiant de transactions et ne peuvent pas être écrites dans les journaux de transactions. Du coup, les actions suivantes produiront des messages d'erreur :

- Langage de Manipulation de Données (LMD ou DML) : INSERT, UPDATE, DELETE, MERGE, COPY FROM, TRUNCATE. Notez qu'il n'y a pas d'action autorisée qui entraînerait l'exécution d'un trigger pendant la récupération. Cette restriction s'applique même pour les tables temporaires car les lignes de ces tables ne peuvent être lues et écrites s'il n'est pas possible d'affecter un identifiant de transactions, ce qui n'est actuellement pas possible dans un environnement Hot Standby.
- Langage de Définition de Données (LDD ou DDL) : CREATE, DROP, ALTER, COMMENT. Cette restriction s'applique aussi aux tables temporaires car, pour mener à bien ces opérations, cela nécessiterait de mettre à jour les catalogues systèmes.
- SELECT ... FOR SHARE | UPDATE, car les verrous de lignes ne peuvent pas être pris sans mettre à jour les fichiers de données.
- Règles sur des ordres SELECT qui génèrent des commandes LMD.
- LOCK qui demandent explicitement un mode supérieur à ROW EXCLUSIVE MODE.
- LOCK dans sa forme courte par défaut, puisqu'il demande ACCESS EXCLUSIVE MODE.
- Commandes de gestion de transaction qui positionnent explicitement un état n'étant pas en lecture-seule :
  - BEGIN READ WRITE, START TRANSACTION READ WRITE
  - SET TRANSACTION READ WRITE, SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE
  - SET transaction_read_only = off

- Commandes de two-phase commit : `PREPARE TRANSACTION`, `COMMIT PREPARED`, `ROLLBACK PREPARED` parce que même les transactions en lecture seule ont besoin d'écrire dans le WAL durant la phase de préparation (la première des deux phases du two-phase commit).

- Mise à jour de séquence : `nextval()`, `setval()`

- `LISTEN`, `NOTIFY`

Dans le cadre normal, les transactions « en lecture seule » permettent l'utilisation des instructions `LISTEN` et `NOTIFY`, donc les sessions Hot Standby ont des restrictions légèrement inférieures à celles de sessions en lecture seule ordinaires. Il est possible que certaines des restrictions soient encore moins importantes dans une prochaine version.

Lors du fonctionnement en serveur hot standby, le paramètre `transaction_read_only` est toujours à true et ne peut pas être modifié. Tant qu'il n'y a pas de tentative de modification sur la base de données, les connexions sur un serveur en hot standby se comportent de façon pratiquement identiques à celles sur un serveur normal. Quand une bascule (*failover* ou *switchover*) survient, la base de données bascule dans le mode de traitement normal. Les sessions resteront connectées pendant le changement de mode. Quand le mode hot standby est terminé, il sera possible de lancer des transactions en lecture/écriture, y compris pour les sessions connectées avant la bascule.

Les utilisateurs peuvent déterminer si le Hot Standby est actuellement actif pour leur session en exécutant `SHOW in_hot_standby`. (Le paramètre `in_hot_standby` n'existant pas avant la version 14, `SHOW transaction_read_only` est donc à utiliser sur les serveurs en version plus ancienne.) De plus, un jeu de fonctions (Tableau 9.90) permettent aux utilisateurs d'accéder à des informations à propos du serveur secondaire. Ceci vous permet d'écrire des programmes qui sont conscients de l'état actuel de la base. Vous pouvez vous en servir pour superviser l'avancement de la récupération, ou pour écrire des programmes complexes qui restaurent la base dans des états particuliers.

27.4.2. Gestion des conflits avec les requêtes

Les nœuds primaire et secondaire sont faiblement couplés à bien des égards. Des actions sur le primaire auront un effet sur le secondaire. Par conséquent, il y a un risque d'interactions négatives ou de conflits entre eux. Le conflit le plus simple à comprendre est la performance : si un gros chargement de données a lieu sur le primaire, il générera un flux similaire d'enregistrements WAL sur le secondaire, et les requêtes du secondaire pourrait entrer en compétition pour les ressources systèmes, comme les entrées-sorties.

Il y a aussi d'autres types de conflits qui peuvent se produire avec le Hot Standby. Ces conflits sont des *conflits durs* dans le sens où des requêtes pourraient devoir être annulées et, dans certains cas, des sessions déconnectées, pour les résoudre. L'utilisateur dispose de plusieurs moyens pour gérer ces conflits. Voici les différents cas de conflits possibles :

- Des verrous en accès exclusif pris sur le serveur primaire, incluant à la fois les commandes `LOCK` exclusives et quelques actions de type DDL, entrent en conflit avec les accès de table des requêtes en lecture seule.

- La suppression d'un tablespace sur le serveur primaire entre en conflit avec les requêtes sur le serveur secondaire qui utilisent ce tablespace pour les fichiers temporaires.

- La suppression d'une base de données sur le serveur primaire entre en conflit avec les sessions connectées sur cette base de données sur le serveur en attente.

- La copie d'un enregistrement nettoyé par un VACUUM entre en conflit avec les transactions sur le serveur en attente qui peuvent toujours « voir » au moins une des lignes à supprimer.

- La copie d'un enregistrement nettoyé par un VACUUM entre en conflit avec les requêtes accédant à la page cible sur le serveur en attente, qu'elles voient ou non les données à supprimer.

Sur le serveur primaire, ces cas résultent en une attente supplémentaire ; l'utilisateur peut choisir d'annuler une des actions en conflit. Néanmoins, sur le serveur en attente, il n'y a pas de choix possibles : l'action enregistrée dans les journaux de transactions est déjà survenue sur le serveur primaire et le serveur secondaire doit absolument réussir à l'appliquer. De plus, permettre que l'enregistrement de l'action attende indéfiniment pourrait avoir des effets fortement non désirables car le serveur en attente sera de plus en plus en retard par rapport au primaire. Du coup, un mécanisme est fourni pour forcer l'annulation des requêtes sur le serveur en attente qui entreraient en conflit avec des enregistrements des journaux de transactions en attente.

Voici un exemple de problème type : un administrateur exécute un `DROP TABLE` sur une table du serveur primaire qui est actuellement utilisé dans des requêtes du serveur en attente. Il est clair que la requête ne peut pas continuer à s'exécuter si l'enregistrement dans les journaux de transactions, correspondant au `DROP TABLE` est appliqué sur le serveur en attente. Si cette situation survient sur le serveur primaire, l'instruction `DROP TABLE` attendra jusqu'à ce que l'autre requête se termine. Par contre, quand le `DROP TABLE` est exécuté sur le serveur primaire, ce dernier ne sait pas les requêtes en cours d'exécution sur le serveur en attente, donc il n'attendra pas la fin de l'exécution des requêtes sur le serveur en attente. L'enregistrement de cette modification dans les journaux de transactions arrivera au serveur en attente alors que la requête sur le serveur en attente est toujours en cours d'exécution, causant un conflit. Le serveur en attente doit soit retarder l'application des enregistrements des journaux de transactions (et tous ceux qui sont après aussi) soit annuler la requête en conflit, pour appliquer l'instruction `DROP TABLE`.

Quand une requête en conflit est courte, il est généralement préférable d'attendre un peu pour l'application du journal de transactions. Mais un délai plus long n'est généralement pas souhaitable. Donc, le mécanisme d'annulation dans l'application des enregistrements de journaux de transactions dispose de deux paramètres, max_standby_archive_delay et max_standby_streaming_delay, qui définissent le délai maximum autorisé pour appliquer les enregistrements. Les requêtes en conflit seront annulées si l'application des enregistrements prend plus de temps que celui défini. Il existe deux paramètres pour que des délais différents puissent être observés suivant le cas : lecture des enregistrements à partir d'un journal archivé (par exemple lors de la restauration initiale à partir d'une sauvegarde ou lors d'un « rattrapage » si le serveur en attente accumulait du retard par rapport au primaire) et lecture des enregistrements à partir de la réplication en flux.

Pour un serveur en attente dont le but principal est la haute-disponibilité, il est préférable de configurer des valeurs assez basses pour les paramètres de délai, de façon à ce que le serveur en attente ne soit pas trop en retard par rapport au serveur primaire à cause des délais suivis à cause des requêtes exécutées sur le serveur en attente. Par contre, si le serveur en attente doit exécuter des requêtes longues, alors une valeur haute, voire infinie, du délai pourrait être préférable. Néanmoins, gardez en tête qu'une requête mettant du temps à s'exécuter pourrait empêcher les autres requêtes de voir les modifications récentes sur le serveur primaire si elle retarde l'application des enregistrements de journaux de transactions.

Une fois que le délai spécifié par `max_standby_archive_delay` ou `max_standby_streaming_delay` a été dépassé, toutes les requêtes en conflit seront annulées. Ceci résulte habituellement en une erreur d'annulation, bien que certains cas, comme un `DROP DATABASE`, peuvent occasionner l'arrêt complet de la connexion. De plus, si le conflit intervient sur un verrou détenu par une transaction en attente, la session en conflit sera terminée (ce comportement pourrait changer dans le futur).

Les requêtes annulées peuvent être ré-exécutées immédiatement (après avoir commencé une nouvelle transaction, bien sûr). Comme l'annulation des requêtes dépend de la nature des enregistrements dans le journal de transactions, une requête annulée pourrait très bien réussir si elle est de nouveau exécutée.

Gardez en tête que les paramètres de délai sont comparés au temps passé depuis que la donnée du journal de transactions a été reçue par le serveur en attente. Du coup, la période de grâce accordée aux requêtes n'est jamais supérieure au paramètre de délai, et peut être considérablement inférieure si le serveur en attente est déjà en retard suite à l'attente de la fin de l'exécution de requêtes précédentes ou suite à son impossibilité de conserver le rythme d'une grosse mise à jour.

La raison la plus fréquente des conflits entre les requêtes en lecture seule et le rejeu des journaux de transactions est le « nettoyage avancé ». Habituellement, PostgreSQL permet le nettoyage des

anciennes versions de lignes quand aucune transaction ne peut les voir pour s'assurer du respect des règles de MVCC. Néanmoins, cette règle peut seulement s'appliquer sur les transactions exécutées sur le serveur primaire. Donc il est possible que le nettoyage effectué sur le primaire supprime des versions de lignes toujours visibles sur une transaction exécutée sur le serveur en attente.

Les utilisateurs expérimentés peuvent noter que le nettoyage des versions de ligne ainsi que le gel des versions de ligne peuvent potentiellement avoir un conflit avec les requêtes exécutées sur le serveur en attente. L'exécution d'un `VACUUM FREEZE` manuel a de grandes chances de causer des conflits, y compris sur les tables sans lignes mises à jour ou supprimées.

Les utilisateurs doivent s'attendre à ce que les tables fréquemment mises à jour sur le serveur primaire seront aussi fréquemment la cause de requêtes annulées sur le serveur en attente. Dans un tel cas, le paramétrage d'une valeur finie pour `max_standby_archive_delay` ou `max_standby_streaming_delay` peut être considéré comme similaire à la configuration de `statement_timeout`.

Si le nombre d'annulations de requêtes sur le serveur en attente est jugé inadmissible, quelques solutions existent. La première option est de définir la variable `hot_standby_feedback` qui permet d'empêcher les conflits liés au nettoyage opéré par la commande `VACUUM` en lui interdisant de nettoyer les lignes récemment supprimées. Si vous le faites, vous devez noter que cela retardera le nettoyage des versions de lignes mortes sur le serveur primaire, ce qui pourrait résulter en une fragmentation non désirée de la table. Néanmoins, cette situation ne sera pas meilleure si les requêtes du serveur en attente s'exécutaient directement sur le serveur primaire. Vous avez toujours le bénéfice de l'exécution sur un serveur distant. Si des serveurs secondaires se connectent et se déconnectent fréquemment, vous pourriez vouloir faire des ajustements pour gérer la période durant laquelle `hot_standby_feedback` n'est pas renvoyé. Par exemple, vous pouvez considérer l'augmentation de `max_standby_archive_delay` pour que les requêtes ne soient pas annulées rapidement par des conflits avec le journal de transactions d'archive durant les périodes de déconnexion. Vous pouvez également considérer l'augmentation de `max_standby_streaming_delay` pour éviter des annulations rapides par les nouvelles données de flux de transaction après la reconnexion.

Une autre option revient à augmenter vacuum_defer_cleanup_age sur le serveur primaire, pour que les lignes mortes ne soient pas nettoyées aussi rapidement que d'habitude. Cela donnera plus de temps aux requêtes pour s'exécuter avant d'être annulées sur le serveur en attente, sans voir à configurer une valeur importante de `max_standby_streaming_delay`. Néanmoins, il est difficile de garantir une fenêtre spécifique de temps d'exécution avec cette approche car `vacuum_defer_cleanup_age` est mesuré en nombre de transactions sur le serveur primaire.

Le nombre de requêtes annulées et le motif de cette annulation peut être visualisé avec la vue système `pg_stat_database_conflicts` sur le serveur secondaire. La vue système `pg_stat_database` contient aussi des informations synthétiques sur ce sujet.

Les utilisateurs peuvent contrôler si un message doit être écrit dans les traces du serveur lorsque le rejeu des journaux de transactions est en conflit depuis plus longtemps que `deadlock_timeout`. Ce comportement est contrôlé par le paramètre log_recovery_conflict_waits.

27.4.3. Aperçu pour l'administrateur

Si `hot_standby` est positionné à `on` dans `postgresql.conf` (valeur par défaut) et qu'un fichier `standby.signal` est présent, le serveur fonctionnera en mode Hot Standby. Toutefois, il pourrait s'écouler du temps avant que les connections en Hot Standby soient autorisées, parce que le serveur n'acceptera pas de connexions tant que la récupération n'aura pas atteint un point garantissant un état cohérent permettant aux requêtes de s'exécuter. Pendant cette période, les clients qui tentent de se connecter seront rejetés avec un message d'erreur. Pour confirmer que le serveur a démarré, vous pouvez soit tenter de vous connecter en boucle, ou rechercher ces messages dans les journaux du serveur:

```
LOG:  entering standby mode
```

```
... puis, plus loin ...

LOG:  consistent recovery state reached
LOG:  database system is ready to accept read-only connections
```

L'information sur la cohérence est enregistrée une fois par checkpoint sur le primaire. Il n'est pas possible d'activer le mode Hot Standby si on lit des WAL générés durant une période pendant laquelle `wal_level` n'était pas positionné à `replica` ou `logical` sur le primaire. L'arrivée à un état cohérent peut aussi être retardée si ces deux conditions se présentent:

- Une transaction en écriture a plus de 64 sous-transactions
- Des transactions en écriture ont une durée très importante

Si vous effectuez du log shipping par fichier (*warm standby*), vous pourriez devoir attendre jusqu'à l'arrivée du prochain fichier de WAL, ce qui pourrait être aussi long que le paramètre `archive_timeout` du primaire.

Certains paramètres déterminent la taille de la mémoire partagée pour le suivi des identifiants de transaction, des verrous, et des transactions préparées. Ces structures mémoires partagées ne peuvent pas être plus petites sur le secondaire que sur le primaire pour s'assurer que le secondaire ne tombera pas à court de mémoire partagée pendant la récupération. Par exemple, si le primaire avait utilisé des transactions préparées mais que le secondaire n'avait pas alloué de mémoire partagée pour suivre ces transactions préparées, alors la récupération ne pourrait reprendre avant que la configuration du secondaire ne soit adaptée. Les paramètres concernés sont:

- `max_connections`
- `max_prepared_transactions`
- `max_locks_per_transaction`
- `max_wal_senders`
- `max_worker_processes`

La façon la plus simple pour éviter que cela ne devienne un problème est d'avoir tous ces paramètres définis sur les serveurs secondaires à des valeurs égales ou supérieures que celles du primaire. C'est pourquoi, si vous voulez augmenter ces valeurs, vous devez le faire d'abord sur tous les serveurs secondaires avant de le faire sur le serveur primaire. De la même façon, si vous voulez diminuer ces valeurs, vous devez d'abord le faire sur le serveur primaire, puis sur tous les serveurs secondaires. Gardez également à l'esprit que lorsqu'un secondaire est promu, il devient alors la nouvelle référence des valeurs de ces paramètres pour les secondaires qui s'y raccrochent. De ce fait, pour éviter que cela ne devienne un problème lors d'une bascule (*switchover* ou *failover*), il est recommandé de conserver ces paramètres identiques sur tous les serveurs secondaires.

Les journaux de transactions tracent le changement de ces paramètres sur le serveur primaire. Si un serveur hot standby rejoue un journal de transactions qui indique que la valeur actuelle sur le primaire est plus élevée que la sienne, un message d'avertissement sera écrit dans les traces du serveur et le rejeu mis en pause. Par exemple :

```
WARNING:  hot standby is not possible because of insufficient
 parameter settings
DETAIL:  max_connections = 80 is a lower setting than on the
 primary server, where its value was 100.
LOG:  recovery has paused
DETAIL:  If recovery is unpaused, the server will shut down.
```

```
HINT:  You can then restart the server after making the necessary
 configuration changes.
```

A ce stade, les paramètres du secondaire doivent être ajustés et l'instance redémarrée avant que le rejeu ne puisse reprendre. Si le secondaire n'est pas en mode hot standby alors, lorsqu'il rencontrera un changement de paramètre incompatible, il s'éteindra immédiatement sans pause, puisqu'il serait inutile de rester actif.

Il est important que l'administrateur sélectionne le paramétrage approprié pour max_standby_archive_delay et max_standby_streaming_delay. Le meilleur choix varie les priorités. Par exemple, si le serveur a comme tâche principale d'être un serveur de haute-disponibilité, alors il est préférable d'avoir une configuration assez basse, voire à zéro, de ces paramètres. Si le serveur en attente est utilisé comme serveur supplémentaire pour des requêtes du type décisionnel, il sera acceptable de mettre les paramètres de délai à des valeurs allant jusqu'à plusieurs heures, voire même -1 (cette valeur signifiant qu'il est possible d'attendre que les requêtes se terminent d'elles-même).

Les "hint bits" (bits d'indices) écrits sur le primaire ne sont pas journalisés en WAL, il est donc probable que les hint bits soient réécrits sur le secondaire. Ainsi, le serveur secondaire fera toujours des écritures disques même si tous les utilisateurs sont en lecture seule ; aucun changement ne se produira sur les données elles mêmes. Les utilisateurs écriront toujours les fichiers temporaires pour les gros tris et re-génèreront les fichiers d'information relcache, il n'y a donc pas de morceau de la base qui soit réellement en lecture seule en mode hot standby. Notez aussi que les écritures dans des bases distantes en utilisant le module dblink, et d'autres opérations en dehors de la base s'appuyant sur des fonctions PL seront toujours possibles, même si la transaction est en lecture seule localement.

Les types suivants de commandes administratives ne sont pas acceptées durant le mode de récupération:

- Langage de Définition de Données (LDD ou DDL) : comme `CREATE INDEX`
- Droits et propriété : `GRANT`, `REVOKE`, `REASSIGN`
- Commandes de maintenance : `ANALYZE`, `VACUUM`, `CLUSTER`, `REINDEX`

Notez encore une fois que certaines de ces commandes sont en fait autorisées durant les transactions en "lecture seule" sur le primaire.

Par conséquent, vous ne pouvez pas créer d'index supplémentaires qui existeraient uniquement sur le secondaire, ni des statistiques qui n'existeraient que sur le secondaire. Si ces commandes administratives sont nécessaires, elles doivent être exécutées sur le primaire, et ces modifications se propageront à terme au secondaire.

`pg_cancel_backend()` et `pg_terminate_backend()` fonctionneront sur les processus utilisateurs, mais pas sur les processus de démarrage, qui effectuent la récupération. `pg_stat_activity` ne montre pas les transactions de récupération comme actives. Ainsi, `pg_prepared_xacts` est toujours vide durant la récupération. Si vous voulez traiter des transactions préparées douteuses, interrogez `pg_prepared_xacts` sur le primaire, et exécutez les commandes pour résoudre le problème à cet endroit ou résolvez-les après la fin de la restauration.

`pg_locks` affichera les verrous posés par les processus, comme en temps normal. `pg_locks` affiche aussi une transaction virtuelle gérée par le processus de démarrage qui possède tous les `AccessExclusiveLocks` posés par les transactions rejouées par la récupération. Notez que le processus de démarrage n'acquiert pas de verrou pour effectuer les modifications à la base, et que par conséquent les verrous autre que `AccessExclusiveLocks` ne sont pas visibles dans `pg_locks` pour le processus de démarrage; ils sont simplement censés exister.

Le plugin Nagios check_pgsql fonctionnera, parce que les informations simples qu'il vérifie existent. Le script de supervision check_postgres fonctionnera aussi, même si certaines valeurs retournées pourraient être différentes ou sujettes à confusion. Par exemple, la date de dernier vacuum ne sera pas mise à jour, puisqu’aucun vacuum ne se déclenche sur le secondaire. Les vacuums s'exécutant sur le primaire envoient toujours leurs modifications au secondaire.

Les options de contrôle des fichiers de WAL ne fonctionneront pas durant la récupération, comme `pg_backup_start`, `pg_switch_wal`, etc...

Les modules à chargement dynamique fonctionnent, comme `pg_stat_statements`.

Les verrous consultatifs fonctionnent normalement durant la récupération, y compris en ce qui concerne la détection des verrous mortels (deadlocks). Notez que les verrous consultatifs ne sont jamais tracés dans les WAL, il est donc impossible pour un verrou consultatif sur le primaire ou le secondaire d'être en conflit avec la ré-application des WAL. Pas plus qu'il n'est possible d'acquérir un verrou consultatif sur le primaire et que celui-ci initie un verrou consultatif similaire sur le secondaire. Les verrous consultatifs n'ont de sens que sur le serveur sur lequel ils sont acquis.

Les systèmes de réplications à base de triggers tels que Slony, Londiste et Bucardo ne fonctionneront pas sur le secondaire du tout, même s'ils fonctionneront sans problème sur le serveur primaire tant que les modifications ne sont pas envoyées sur le serveur secondaire pour y être appliquées. Le rejeu de WAL n'est pas à base de triggers, vous ne pouvez donc pas utiliser le secondaire comme relais vers un système qui aurait besoin d'écritures supplémentaires ou utilise des triggers.

Il n'est pas possible d'assigner de nouveaux OID, bien que des générateurs d'UUID puissent tout de même fonctionner, tant qu'ils n'ont pas besoin d'écrire un nouveau statut dans la base.

À l'heure actuelle, la création de table temporaire n'est pas autorisée durant les transactions en lecture seule, certains scripts existants pourraient donc ne pas fonctionner correctement. Cette restriction pourrait être levée dans une version ultérieure. Il s'agit à la fois d'un problème de respect des standards et d'un problème technique.

`DROP TABLESPACE` ne peut réussir que si le tablespace est vide. Certains utilisateurs pourraient utiliser de façon active le tablespace via leur paramètre `temp_tablespaces`. S'il y a des fichiers temporaires dans le tablespace, toutes les requêtes actives sont annulées pour s'assurer que les fichiers temporaires sont supprimés, afin de supprimer le tablespace et de continuer l'application des WAL.

Exécuter `DROP DATABASE` ou `ALTER DATABASE ... SET TABLESPACE` sur le serveur primaire générera un enregistrement dans les journaux de transactions qui causera la déconnexion de tous les utilisateurs actuellement connectés à cette base de données. Cette action survient immédiatement, quelque soit la valeur du paramètre `max_standby_streaming_delay`. Notez que `ALTER DATABASE ... RENAME` ne déconnecte pas les utilisateurs qui, dans la plupart des cas, ne s'en apercevront pas. Cela peut néanmoins semer de la confusion pour un programme qui dépendrait du nom de la base.

En fonctionnement normal (pas en restauration), si vous exécutez `DROP USER` ou `DROP ROLE` pour un rôle ayant l'attribut LOGIN alors que cet utilisateur est toujours connecté alors rien ne se produit pour cet utilisateur connecté -- il reste connecté. L'utilisateur ne peut toutefois pas se reconnecter. Ce comportement est le même en récupération, un `DROP USER` sur le primaire ne déconnecte donc pas cet utilisateur sur le secondaire.

Le système de statistiques cumulatives est actif pendant la restauration. Tous les parcours, lectures, blocs, utilisations d'index, etc. seront enregistrés normalement sur le standby. Néanmoins, le rejeu des journaux de transactions n'incrémentera pas les compteurs spécifiques des relations et des bases. Le rejeu n'incrémentera pas les colonnes de pg_stat_all_tables (comme n_tup_ins), pas plus que les lectures et écritures réalisées par le processus de démarrage ne seront tracées dans les vues pg_statio, ou que les colonnes associées de pg_stat_database ne seront incrémentées.

Autovacuum n'est pas actif durant la récupération, il démarrera normalement à la fin de la récupération.

Les processus d'écriture en arrière plan (checkpointer et background writer) sont actifs durant la restauration. Le processus checkpointer process effectuera les restartpoints (similaires aux checkpoints sur le primaire) et le processus background writer réalisera les activités normales de nettoyage de blocs. Ceci peut inclure la mise à jour des information de hint bit des données du serveur secondaire. La commande `CHECKPOINT` est acceptée pendant la récupération, bien qu'elle déclenche un restartpoint et non un checkpoint.

27.4.4. Référence des paramètres de Hot Standby

De nombreux paramètres ont été mentionnés ci-dessus dans Section 27.4.2 et Section 27.4.3.

Sur le primaire, les paramètres wal_level et vacuum_defer_cleanup_age peuvent être utilisés. max_standby_archive_delay et max_standby_streaming_delay n'ont aucun effet sur le primaire.

Sur le serveur en attente, les paramètres hot_standby, max_standby_archive_delay et max_standby_streaming_delay peuvent être utilisés. vacuum_defer_cleanup_age n'a pas d'effet tant que le serveur reste dans le mode standby, mais deviendra important quand le serveur en attente deviendra un serveur primaire.

27.4.5. Avertissements

Il y a plusieurs limitations au Hot Standby. Elles peuvent et seront probablement résolues dans des versions ultérieures:

- Une connaissance complète des transactions en cours d'exécution est nécessaire avant de pouvoir déclencher des instantanés. Des transactions utilisant un grand nombre de sous-transactions (à l'heure actuelle plus de 64) retarderont le démarrage des connexions en lecture seule jusqu'à complétion de la plus longue transaction en écriture. Si cette situation se produit, des messages explicatifs seront envoyés dans la trace du serveur.

- Des points de démarrage valides pour les requêtes du secondaire sont générés à chaque checkpoint sur le primaire. Si le secondaire est éteint alors que le primaire est déjà éteint, il est tout à fait possible ne pas pouvoir repasser en Hot Standby tant que le primaire n'aura pas été redémarré, afin qu'il génère de nouveaux points de démarrage dans les journaux WAL. Cette situation n'est pas un problème dans la plupart des situations où cela pourrait se produire. Généralement, si le primaire est éteint et plus disponible, c'est probablement en raison d'un problème sérieux qui va de toutes façons forcer la conversion du secondaire en primaire. Et dans des situations où le primaire est éteint intentionnellement, la procédure standard est également de promouvoir le secondaire.

- À la fin de la récupération, les `AccessExclusiveLocks` possédés par des transactions préparées nécessiteront deux fois le nombre d'entrées normal dans la table de verrous. Si vous pensez soit exécuter un grand nombre de transactions préparées prenant des `AccessExclusiveLocks`, ou une grosse transaction prenant beaucoup de `AccessExclusiveLocks`, il est conseillé d'augmenter la valeur de `max_locks_per_transaction`, peut-être jusqu'à une valeur double de celle du serveur primaire. Vous n'avez pas besoin de prendre ceci en compte si votre paramètre `max_prepared_transactions` est 0.

- Il n'est pas encore possible de passer une transaction en mode d'isolation sérialisable tout en supportant le hot standby (voir Section 13.2.3 et Section 13.4.1 pour plus de détails). Une tentative de modification du niveau d'isolation d'une transaction à sérialisable en hot standby générera une erreur.

Chapitre 28. Superviser l'activité de la base de données

Un administrateur de bases de données se demande fréquemment : « Que fait le système en ce moment ? » Ce chapitre explique la façon de le savoir.

Plusieurs outils sont disponibles pour surveiller l'activité de la base de données et pour analyser les performances. Une grande partie de ce chapitre concerne la description du système de statistiques cumulatives de PostgreSQL mais personne ne devrait négliger les programmes de surveillance Unix standards tels que `ps`, `top`, `iostat` et `vmstat`. De plus, une fois qu'une requête peu performante a été identifiée, des investigations supplémentaires pourraient être nécessaires en utilisant la commande `EXPLAIN` de PostgreSQL. La Section 14.1 discute de `EXPLAIN` et des autres méthodes pour comprendre le comportement d'une seule requête.

28.1. Outils Unix standard

Sur la plupart des plateformes Unix, PostgreSQL modifie son titre de commande reporté par `ps` de façon à ce que les processus serveur individuels puissent être rapidement identifiés. Voici un affichage d'exemple :

```
$ ps auxww | grep ^postgres
postgres  15551  0.0  0.1  57536  7132 pts/0    S    18:02   0:00
 postgres -i
postgres  15554  0.0  0.0  57536  1184 ?        Ss   18:02   0:00
 postgres: background writer
postgres  15555  0.0  0.0  57536   916 ?        Ss   18:02   0:00
 postgres: checkpointer
postgres  15556  0.0  0.0  57536   916 ?        Ss   18:02   0:00
 postgres: walwriter
postgres  15557  0.0  0.0  58504  2244 ?        Ss   18:02   0:00
 postgres: autovacuum launcher
postgres  15582  0.0  0.0  58772  3080 ?        Ss   18:04   0:00
 postgres: joe runbug 127.0.0.1 idle
postgres  15606  0.0  0.0  58772  3052 ?        Ss   18:07   0:00
 postgres: tgl regression [local] SELECT waiting
postgres  15610  0.0  0.0  58772  3056 ?        Ss   18:07   0:00
 postgres: tgl regression [local] idle in transaction
```

(L'appel approprié de `ps` varie suivant les différentes plateformes, de même que les détails affichés. Cet exemple est tiré d'un système Linux récent.) Le premier processus affiché ici est le processus serveur primaire. Les arguments affichés pour cette commande sont les mêmes qu'à son lancement. Les quatre processus suivants sont des processus en tâche de fond lancés automatiquement par le processus primaire (le processus « autovacuum launcher » ne sera pas présent si vous avez configuré le système pour qu'il n'exécute pas l'autovacuum). Chacun des autres processus est un processus serveur gérant une connexion cliente. Tous ces processus restants initialisent l'affichage de la ligne de commande sous la forme :

`postgres: `*`utilisateur base_de_données hôte activité`*

L'utilisateur, la base de données et les éléments de l'hôte (client) restent identiques pendant toute la vie de connexion du client mais, l'indicateur d'activité change. L'activité pourrait être `idle` (c'est-à-dire en attente d'une commande du client), `idle in transaction` (en attente du client à l'intérieur d'un bloc de `BEGIN`/`COMMIT`) ou un nom de commande du type `SELECT`. De plus, `waiting` est ajouté si le processus serveur est en attente d'un verrou détenu par une autre session. Dans l'exemple ci-dessus, nous pouvons supposer que le processus 15606 attend que le processus 15610 finisse sa

transaction, et par conséquent libère un verrou (le processus 15610 doit être celui qui bloque car il n'y a aucune autre session active. Dans des cas plus compliqués, il serait nécessaire de regarder dans la vue système `pg_locks` pour déterminer qui est en train de bloquer qui.)

Si cluster_name a été configuré, le nom de l'instance figurera également dans la sortie :

```
$ psql -c 'SHOW cluster_name'
 cluster_name
--------------
 server1
(1 row)

$ ps aux|grep server1
postgres   27093  0.0  0.0  30096  2752 ?        Ss   11:34   0:00
 postgres: server1: background writer
...
```

Si vous avez désactivé update_process_title, alors l'indicateur d'activité n'est pas mis à jour ; le titre du processus est configuré une seule fois quand un nouveau processus est lancé. Sur certaines plateformes, ceci permet d'économiser du temps. Sur d'autres, cette économie est insignifiante.

> **Astuce**
>
> Solaris requiert une gestion particulière. Vous devez utiliser `/usr/ucb/ps` plutôt que `/bin/ps`. Vous devez aussi utiliser deux options `w` et non pas seulement une. En plus, votre appel original de la commande `postgres` doit avoir un affichage de statut dans `ps` plus petit que celui fourni par les autres processus serveur. Vous devez donc faire ces trois opérations sinon l'affichage de `ps` pour chaque processus serveur sera la ligne de commande originale de `postgres`.

28.2. Le système de statistiques cumulatives

Le *système de statistiques cumulatives* de PostgreSQL récupère et fournit des informations sur l'activité du serveur. Actuellement, l'accès aux tables et index est comptabilisé en terme d'accès aux lignes et aux blocs. Le nombre total de lignes de chaque table et les informations sur les opérations de nettoyage et de calcul de statistiques sont aussi récupérées pour chaque table. Si activé, les appels aux fonctions des utilisateurs et le temps total passé dans chacune sont aussi comptabilisés.

PostgreSQL est également capable de renvoyer des informations dynamiques en temps réel sur ce qu'il se passe exactement dans le système, comme la commande exacte en cours d'exécution par d'autres processus serveur et les autres connexions qui existent dans le système. Cette fonctionnalité est indépendante du système de statistiques cumulatives.

28.2.1. Configuration de la récupération de statistiques

Comme la récupération de statistiques ajoute un temps supplémentaire à l'exécution de la requête, le système peut être configuré pour récupérer ou non des informations. Ceci est contrôlé par les paramètres de configuration qui sont normalement initialisés dans `postgresql.conf` (voir Chapitre 20 pour plus de détails sur leur initialisation).

Le paramètre track_activities active la collecte d'informations sur la commande en cours d'exécution pour n'importe quel processus serveur.

Le paramètre track_counts contrôle si les statistiques cumulatives sont récupérées pour les accès aux tables et index.

Le paramètre track_functions active le calcul de statistiques sur l'utilisation des fonctions définies par l'utilisateur.

Le paramètre track_io_timing active la collecte des temps de lecture et d'écriture de blocs.

Le paramètre track_wal_io_timing active la collecte des temps d'écriture des journaux de transactions.

Normalement, ces paramètres sont configurés dans `postgresql.conf` de façon à ce qu'ils s'appliquent à tous les processus serveur, mais il est possible de les activer/désactiver sur des sessions individuelles en utilisant la commande SET (pour empêcher les utilisateurs ordinaires de cacher leur activité à l'administrateur, seuls les superutilisateurs sont autorisés à modifier ces paramètres avec `SET`).

Les statistiques cumulatives sont placées en mémoire partagée. Chaque processus PostgreSQL récupère les statistiques localement, puis met à jour les données partagées à des intervalles adéquats. Quand un serveur, y compris un secondaire physique, s'arrête proprement, une copie permanente des données statistiques est enregistrée dans le sous-répertoire `pg_stat`, pour que les statistiques survivent aux redémarrages du serveur. Par contre, en cas de démarrage après un arrêt sale (autrement dit, après un arrêt immédiat, un crash du serveur, un démarrage suite à la restauration d'une sauvegarde de base), tous les compteurs statistiques sont réinitialisés.

28.2.2. Visualiser les statistiques

Plusieurs vues prédéfinies, listées à Tableau 28.1, sont disponibles pour montrer l'état courant du système. Il existe aussi plusieurs autres vues, listées à Tableau 28.2, qui montrent les statistiques accumulées. De manière alternative, il est possible de créer des vues personnalisées qui utilisent les fonctions statistiques sous-jacentes, comme discuté à Section 28.2.24.

Lors de l'utilisation des vues et fonctions des statistiques cumulatives pour superviser les données collectées, il est important de savoir que l'information n'est pas mise à jour instantanément. Chaque processus serveur individuel vide les statistiques accumulées en mémoire partagée avant d'être de nouveau en attente, mais cela n'arrive pas plus fréquemment qu'une fois toutes les `PGSTAT_MIN_INTERVAL` millisecondes (1 seconde sauf si cette constante est modifiée avant la compilation du serveur) ; donc une requête ou une transaction toujours en cours n'affecte pas les totaux affichés et l'information est de ce fait en retard par rapport à l'activité en cours. Néanmoins, l'information de la requête en cours récupérée par `track_activities` est toujours à jour.

Un autre point important est que, lorsqu'un processus serveur se voit demander d'afficher une des statistiques accumulées, les valeurs accédées sont mises en cache jusqu'à la fin de la transaction courante dans la configuration par défaut. Donc les statistiques afficheront une information statique tout au long de la vue de la transaction. De façon similaire, l'information sur les requêtes en cours de toutes les sessions est récupérée quand cette information est demandée pour la première fois à l'intérieur d'une transaction, et la même information sera affichée tout au long de la transaction. Ceci est une fonctionnalité, et non pas un bug, car cela vous permet de réaliser plusieurs requêtes sur les statistiques et de corréler les résultats sans vous soucier que les nombres changent entre temps. Lors de l'analyse interactive des statistiques ou avec des requêtes coûteuses, le delta du temps entre les accès aux statistiques individuelles peut amener à des valeurs fausses significativement dans les statistiques en cache. Pour minimiser cela, `stats_fetch_consistency` peut être configuré à `snapshot`, au prix d'une utilisation mémoire accrue pour meetre en cache des données statistiques inutiles. Par contre, s'il est connu que les statistiques ne sont accédées qu'une seule fois, mettre en cache les statistiques accédées n'est pas nécessaire et peut être évité en configurant `stats_fetch_consistency` à `none`. Vous pouvez exécuter `pg_stat_clear_snapshot()` pour annuler l'image des statistiques de la transaction en cours ou les valeurs en cache, si applicable. La prochaine utilisation des informations statistiques causera (en mode snapshot) la construction d'un nouveau snapshot ou (en mode cache) la mise en cache des statistiques accédées.

Une transaction peut aussi voir ses propres statistiques (pas encore vidées dans les statistiques en mémoire partagée) dans les vues `pg_stat_xact_all_tables`, `pg_stat_xact_sys_tables`, `pg_stat_xact_user_tables` et

`pg_stat_xact_user_functions`. Ces nombres n'agissent pas comme indiqués ci-dessus ; ils sont au contraire mis à jour en continue tout au long de la transaction.

Certaines des informations des vues statistiques dynamiques montrées dans Tableau 28.1 sont à accès restreint. Les utilisateurs ordinaires peuvent seulement voir toutes les informations sur leur propres sessions (les sessions appartenant à un rôle dont ils sont membres). Les lignes des autres sessions auront de nombreuses colonnes à NULL. Notez néanmoins que l'existence d'une session et ses propriétés générales, tel que l'utilisateur et la base de données, sont visibles à tous les utilisateurs. Les superutilisateurs et les rôles disposant des droits du rôle interne `pg_read_all_stats` (voir aussi Section 22.5) peuvent accéder à toutes les informations sur les sessions.

Tableau 28.1. Vues statistiques dynamiques

Nom de la vue	Description
`pg_stat_activity`	Une ligne par processus serveur, montrant les informations liées à l'activité courante du processus, comme l'état et la requête en cours. Voir `pg_stat_activity` pour plus de détails.
`pg_stat_replication`	Une ligne par processus d'envoi de WAL, montrant les statistiques sur la réplication vers le serveur standby connecté au processus. Voir `pg_stat_replication` pour les détails.
`pg_stat_wal_receiver`	Seulement une ligne, affichant des statistiques sur le récepteur WAL à partir du serveur ayant ce récepteur. Voir `pg_stat_wal_receiver` pour les détails.
`pg_stat_recovery_prefetch`	Only one row, showing statistics about blocks prefetched during recovery. See `pg_stat_recovery_prefetch` for details.
`pg_stat_subscription`	Au moins une ligne par souscription, affichant des informations sur les processus workers de souscription. Voir `pg_stat_subscription` pour plus de détails.
`pg_stat_ssl`	Une ligne par connexion (régulière et de réplication), montrant des informations sur le chiffrement SSL utilisé pour ces connexions. Voir `pg_stat_ssl` pour les détails.
`pg_stat_gssapi`	Une ligne par connexion (standard et réplication), montrant des informations sur l'authentification et le chiffrement GSSAPI utilisés sur cette connexion. Voir `pg_stat_gssapi` pour les détails.
`pg_stat_progress_analyze`	Une ligne pour chaque connexion (incluant celles des processus *autovacuum worker*) exécutant un `ANALYZE`, pour indiquer le progrès actuel. Voir Section 28.4.1.
`pg_stat_progress_create_index`	Une ligne par chaque processus serveur exécutant un `CREATE INDEX` ou un `REINDEX`, affichant la progression actuelle. Voir Section 28.4.2.
`pg_stat_progress_vacuum`	Une ligne pour chaque processus (incluant les processus autovacuum worker) exécutant un `VACUUM`, affichant le progrès en cours. Voir Section 28.4.3.
`pg_stat_progress_cluster`	Une ligne pour chaque processus serveur exécutant `CLUSTER` ou `VACUUM FULL`, affichant le progrès en cours. Voir Section 28.4.4.
`pg_stat_progress_basebackup`	Une ligne pour chaque processus walsender envoyant une sauvegarde de base, en affichant le progrès actuel. Voir Section 28.4.5.
`pg_stat_progress_copy`	Une ligne pour chaque processus en cours d'exécution de `COPY`, affichant le progrès actuel. Voir Section 28.4.6.

Tableau 28.2. Vues sur les statistiques récupérées

Nom de la vue	Description
`pg_stat_archiver`	Une seule ligne, montrant les statistiques sur l'activité du processus d'archivage des journaux

Nom de la vue	Description
	de transactions. Voir `pg_stat_archiver` pour les détails.
`pg_stat_bgwriter`	Une seule ligne, montrant les statistiques d'activité du processus d'écriture d'arrière-plan. Voir `pg_stat_bgwriter` pour plus de détails.
`pg_stat_wal`	Une seule ligne, montrant les statistiques d'activité WAL. Voir `pg_stat_wal` pour plus de détails.
`pg_stat_database`	Une ligne par base de données, montrant les statistiques globales des bases. Voir `pg_stat_database` pour plus de détails.
`pg_stat_database_conflicts`	Une ligne par base de données, montrant les statistiques au niveau de la base concernant les requêtes annulées à cause de conflits avec les serveurs standby en restauration. Voir `pg_stat_database_conflicts` pour plus de détails.
`pg_stat_all_tables`	Une ligne par table de la base de données courante, montrant les statistiques d'accès de chaque table spécifiquement. Voir `pg_stat_all_tables` pour plus de détails.
`pg_stat_sys_tables`	Identique à `pg_stat_all_tables`, sauf que seules les tables systèmes sont affichées
`pg_stat_user_tables`	Identique à `pg_stat_all_tables`, sauf que seules les tables utilisateurs sont affichées
`pg_stat_xact_all_tables`	Similaire à `pg_stat_all_tables`, mais décompte les actions prises dans la transaction en cours (qui ne sont *pas* encore pris en compte dans la vue `pg_stat_all_tables` et les vues du même type). Les colonnes correspondant au nombre de lignes vivantes et mortes, ainsi que celles pour les actions du VACUUM et de l'ANALYZE ne sont pas présentes dans cette vue
`pg_stat_xact_sys_tables`	Identique à `pg_stat_xact_all_tables`, sauf que seules les tables systèmes sont affichées
`pg_stat_xact_user_tables`	Identique à `pg_stat_xact_all_tables`, sauf que seules les tables utilisateurs sont affichées
`pg_stat_all_indexes`	Une ligne par index de la base de données courante, montrant les statistiques d'accès de chaque index spécifiquement. Voir `pg_stat_all_indexes` pour plus de détails.
`pg_stat_sys_indexes`	Identique à `pg_stat_all_indexes`, sauf que seules les tables systèmes sont affichées
`pg_stat_user_indexes`	Identique à `pg_stat_all_indexes`, sauf que seules les tables utilisateurs sont affichées
`pg_statio_all_tables`	Une ligne par table de la base de données courante, montrant les statistiques d'entrées/sorties de chaque table spécifiquement. Voir

Nom de la vue	Description
	`pg_statio_all_tables` pour plus de détails.
`pg_statio_sys_tables`	Identique à `pg_statio_all_tables`, sauf que seules les tables systèmes sont affichées
`pg_statio_user_tables`	Identique à `pg_statio_all_tables`, sauf que seules les tables utilisateur sont affichées
`pg_statio_all_indexes`	Une ligne par index de la base de données courante, montrant les les statistiques d'entrées/sorties de chaque index spécifiquement. Voir `pg_statio_all_indexes` pour plus de détails.
`pg_statio_sys_indexes`	Identique à `pg_statio_all_indexes`, sauf que seuls les index systèmes sont affichés
`pg_statio_user_indexes`	Identique à `pg_statio_all_indexes`, sauf que seuls les index utilisateur sont affichés
`pg_statio_all_sequences`	Une ligne par séquence de la base de données courante, montrant les statistiques d'entrées/sorties de chaque séquence spécifiquement. Voir `pg_statio_all_sequences` pour plus de détails.
`pg_statio_sys_sequences`	Identique à `pg_statio_all_sequences`, sauf que seules les séquences système sont affichées (actuellement, aucune séquence système n'est définie, donc cette vue est toujours vide)
`pg_statio_user_sequences`	Identique à `pg_statio_all_sequences`, sauf que seules les séquences utilisateur sont affichées
`pg_stat_user_functions`	Une ligne par fonction suivie, montrant les statistiques d'exécution de cette fonction. Voir `pg_stat_user_functions` pour plus de détails.
`pg_stat_xact_user_functions`	Similaire à `pg_stat_user_functions`, mais compte seulement les appels pendant la transaction en cours (qui ne sont *pas* encore inclus dans `pg_stat_user_functions`)
`pg_stat_slru`	Une ligne par SLRU, affichant les statistiques des opérations. Voir `pg_stat_slru` pour les détails.
`pg_stat_replication_slots`	Une ligne par slot de réplication, affichant les statistiques de l'utilisation du slot de réplication. Voir `pg_stat_replication_slots` pour les détails.
`pg_stat_subscription_stats`	One row per subscription, showing statistics about errors. See `pg_stat_subscription_stats` for details.

Les statistiques par index sont particulièrement utiles pour déterminer les index utilisés et leur efficacité.

Les vues pg_statio_ sont principalement utiles pour déterminer l'efficacité du cache tampon. Quand le nombre de lectures disques réelles est plus petit que le nombre de récupérations valides par le tampon, alors le cache satisfait la plupart des demandes de lecture sans faire appel au noyau. Néanmoins, ces statistiques ne nous donnent pas l'histoire complète : à cause de la façon dont PostgreSQL gère les entrées/sorties disque, les données qui ne sont pas dans le tampon de PostgreSQL pourraient toujours résider dans le tampon d'entrées/sorties du noyau et pourraient, du coup, être toujours récupérées sans nécessiter une lecture physique. Les utilisateurs intéressés pour obtenir des informations plus détaillées sur le comportement des entrées/sorties dans PostgreSQL sont invités à utiliser les vues statistiques de PostgreSQL avec les outils du système d'exploitation permettant une vue de la gestion des entrées/sorties par le noyau.

28.2.3. pg_stat_activity

La vue pg_stat_activity aura une ligne par processus serveur, montrant des informations liées à l'activité courante de ce processus.

Tableau 28.3. Vue pg_stat_activity

Type Description
datid oid OID de la base de données auquel ce processus serveur est connecté
datname name Nom de la base de données auquel ce processus serveur est connecté
pid integer Identifiant du processus serveur
leader_pid integer Identifiant du processus du leader pour une requête parallélisée si ce si ce processus est un worker. NULL si ce processus est le leader ou ne participe pas à une requête parallélisée.
usesysid oid OID de l'utilisateur connecté à ce processus serveur
usename name Nom de l'utilisateur connecté à ce processus serveur
application_name text Nom de l'application connectée à ce processus serveur
client_addr inet Adresse IP du client pour ce processus serveur. Si ce champ est vide, cela indique soit que le client est connecté via un socket Unix sur la machine serveur soit qu'il s'agit d'un processus interne tel qu'autovacuum.
client_hostname text Nom d'hôte du client connecté, comme reporté par une recherche DNS inverse sur client_addr. Ce champ ne sera rempli que pour les connexions IP, et seulement quand log_hostname est activé.
client_port integer Numéro de port TCP que le client utilise pour communiquer avec le processus serveur, ou -1 si un socket Unix est utilisé.
backend_start timestamp with time zone Heure de démarrage du processus. Pour les processus backends, c'est l'heure où le client s'est connecté au serveur.
xact_start timestamp with time zone

Type Description
Heure de démarrage de la transaction courante du processus, ou NULL si aucune transaction n'est active. Si la requête courante est la première de sa transaction, cette colonne a la même valeur que la colonne `query_start`.
`query_start` `timestamp with time zone` Heure à laquelle la requête active a été démarrée, ou si `state` ne vaut pas `active`, quand la dernière requête a été lancée.
`state_change` `timestamp with time zone` Heure à laquelle l'état (`state`) a été modifié en dernier
`wait_event_type` `text` Type de l'événement pour lequel le processus est en attente sinon NULL. Voir Tableau 28.4.
`wait_event` `text` Nom de l'événement d'attente si le processus est en attente, NULL dans le cas contraire. Voir Tableau 28.5 jusqu'à Tableau 28.13.
`state` `text` État général du processus serveur. Les valeurs possibles sont : • `active` : le processus serveur exécute une requête. • `idle` : le processus serveur est en attente d'une commande par le client. • `idle in transaction` : le processus serveur est en transaction, mais n'est pas en train d'exécuter une requête. • `idle in transaction (aborted)` : l'état est similaire à `idle in transaction`, à la différence qu'une des instructions de la transaction a généré une erreur. • `fastpath function call` : le processus serveur exécute une fonction fast-path. • `disabled` : cet état est affiché si track_activities est désactivé pour ce processus serveur.
`backend_xid` `xid` Identifiant de transaction de haut niveau de ce processus, si disponible.
`backend_xmin` `xid` L'horizon `xmin` de ce processus.
`query_id` `bigint` L'identifiant de la requête la plus récente du processus. Si `state` vaut `active`, alors cette colonne indique l'identifiant de la requête en cours d'exécution. Pour les autres états, cette colonne indique l'identifiant de la dernière requête exécutée par le processus. Les identifiants de requêtes ne sont pas calculés par défaut, ainsi cette colonne peut être nulle sauf si le paramètre compute_query_id est activé ou si un module tiers calculant les identifiants de requêtes est configuré.
`query` `text` Texte de la requête la plus récente pour ce processus serveur. Si `state` vaut `active`, alors ce champ affiche la requête en cours d'exécution. Dans tous les autres cas, il affichera la dernière requête à avoir été exécutée. Par défaut, le texte de la requête est tronqué à 1024 octets. Cette valeur peut être modifiée avec le paramètre track_activity_query_size.
`backend_type` `text` Type du processus actuel. Les types possibles sont `autovacuum launcher`, `autovacuum worker`, `logical replication launcher`, `logical replication worker`, `parallel worker`, `background writer`, `client`

Type	Description
	`backend`, `checkpointer`, `archiver`, `startup`, `walreceiver`, `walsender` et `walwriter`. De plus, les background workers enregistrés par les extensions pourraient avoir des types supplémentaires.

> **Note**
>
> Les colonnes `wait_event` et `state` sont indépendantes. Si un processus serveur est dans l'état `active`, il pourrait, ou non, être en attente (`waiting`) d'un événement. Si l'état est `active` et si `wait_event` est différent de NULL, cela signifie qu'une requête est en cours d'exécution, mais que cette exécution est bloquée quelque part dans le système.

Tableau 28.4. Types de Wait Event

Type de Wait Event	Description
`Activity`	Le processus serveur est en attente. Ce type d'événement indique un processus attendant une activité dans sa boucle de traitement principal. `wait_event` identifie le point d'attente spécifique ; voir Tableau 28.5.
`BufferPin`	Le processus serveur est en attente d'un accès exclusif à un tampon de données. Les attentes de ce type peuvent être prolongées si un autre processus détient un curseur ouvert qui a lu des données du tampon en question. Voir Tableau 28.6.
`Client`	Le processus serveur est en attente d'activité sur un socket connecté à une application utilisateur. De ce fait, le serveur s'attend à une action indépendante de ses processus internes. `wait_event` identifie le point d'attente spécifique ; voir Tableau 28.7.
`Extension`	Le processus serveur est en attente d'une condition définie par une extension. Voir Tableau 28.8.
`IO`	Le processus serveur est en attente de la fin d'une opération I/O. `wait_event` identifie le point d'attente spécifique ; voir Tableau 28.9.
`IPC`	Le processus serveur est en attente d'une interaction avec un autre processus serveur. `wait_event` identifie le point d'attente spécifique ; voir Tableau 28.10.
`Lock`	Le processus serveur est un attente d'un verrou lourd. Les verrous lourds, aussi connus sous le nom de verrous du gestionnaire ou simplement verrous, protègent principalement les objets SQL comme les tables. Néanmoins, ils sont aussi utilisés pour assurer l'exclusion mutuelle pour certaines opérations internes comme l'agrandissement d'une relation. `wait_event` identifie le type de verrou en attente ; voir Tableau 28.11.

Type de Wait Event	Description
LWLock	Le processus serveur est en attente d'un verrou léger. La plupart de ces types de verrou protègent une structure de données spécifique en mémoire partagée. wait_event contient un nom identifiant le but de ce verrou léger. (Certains verrous ont des noms spécifiques ; les autres font partie d'un groupe de verrous, chacun ayant le même but.) Voir Tableau 28.12.
Timeout	Le processus serveur est en attente de l'expiration d'un délai. wait_event identifie le point d'attente spécifique ; voir Tableau 28.13.

Tableau 28.5. Wait Events de type Activity

Wait Event Activity	Description
ArchiverMain	En attente dans la boucle principale du processus archiver.
AutoVacuumMain	En attente dans la boucle principale du processus autovacuum launcher.
BgWriterHibernate	En attente du processus background writer, en hibernation.
BgWriterMain	En attente dans la boucle principale du processus background writer.
CheckpointerMain	En attente dans la boucle principale du processus checkpointer.
LogicalApplyMain	En attente dans la boucle principale du processus logical replication apply.
LogicalLauncherMain	En attente dans la boucle principale du processus logical replication launcher.
RecoveryWalStream	En attente dans la boucle principale du processus startup pour l'arrivée des WAL durant la réplication en flux.
SysLoggerMain	En attente dans la boucle principale du processus syslogger.
WalReceiverMain	En attente dans la boucle principale du processus WAL receiver.
WalSenderMain	En attente dans la boucle principale du processus WAL sender.
WalWriterMain	En attente dans la boucle principale du processus WAL writer.

Tableau 28.6. Wait Events de type BufferPin

Wait Event BufferPin	Description
BufferPin	En attente de l'acquisition d'un verrou exclusif sur un tampon.

Tableau 28.7. Wait Events de type `Client`

Wait Event `Client`	Description
`ClientRead`	En attente d'une lecture de données provenant du client.
`ClientWrite`	En attente d'écriture de données vers le client.
`GSSOpenServer`	En attente de lecture de données provenant du client lors de l'établissement d'une session GSSAPI.
`LibPQWalReceiverConnect`	En attente de WAL receiver pour établir une connexion vers un serveur distant.
`LibPQWalReceiverReceive`	En attente de WAL receiver pour recevoir des données provenant d'un serveur distant.
`SSLOpenServer`	En attente de SSL lors d'une tentative de connexion.
`WalSenderWaitForWAL`	En attente de l'envoi des WAL au processus WAL sender.
`WalSenderWriteData`	En attente de toute activité pendant le traitement des réponses provenant du WAL receiver dans le processus WAL sender.

Tableau 28.8. Wait Events de type `Extension`

Wait Event `Extension`	Description
`Extension`	En attente d'une extension.

Tableau 28.9. Wait Events de type `IO`

Wait Event `IO`	Description
`BaseBackupRead`	En attente que la sauvegarde de base lise un fichier.
`BaseBackupSync`	En attente que les données écrites par une sauvegarde de base atteignent un stockage durable.
`BaseBackupWrite`	En attente que la sauvegarde de base écrive un fichier.
`BufFileRead`	En attente d'une lecture d'un fichier en cache.
`BufFileWrite`	En attente d'une écriture dans un fichier en cache.
`BufFileTruncate`	En attente de la troncature d'un fichier en cache.
`ControlFileRead`	En attente d'une lecture à partir du fichier `pg_control`.
`ControlFileSync`	En attente que le fichier `pg_control` atteigne un stockage stable.
`ControlFileSyncUpdate`	En attente qu'une mise à jour du fichier `pg_control` atteigne un stockage stable.
`ControlFileWrite`	En attente d'une écriture dans le fichier `pg_control`.
`ControlFileWriteUpdate`	En attente d'une écriture pour mettre à jour le fichier `pg_control`.

Wait Event `IO`	**Description**
`CopyFileRead`	En attente d'une lecture lors d'une opération de copie de fichier.
`CopyFileWrite`	En attente d'une écriture lors d'une opération de copie de fichier.
`DSMFillZeroWrite`	En attente du remplissage d'un fichier de mémoire partagée dynamique avec des zéros.
`DataFileExtend`	En attente de l'agrandissement d'un fichier de données.
`DataFileFlush`	En attente que le fichier de données atteigne un stockage stable.
`DataFileImmediateSync`	En attente d'une synchronisation immédiate d'un fichier de données vers un stockage stable.
`DataFilePrefetch`	En attente d'une prélecture asynchrone à partir d'un fichier de données.
`DataFileRead`	En attente d'une lecture à partir d'un fichier de données.
`DataFileSync`	En attente de modifications dans un fichier de données pour atteindre un stockage stable.
`DataFileTruncate`	En attente de la troncature d'un fichier de données.
`DataFileWrite`	En attente de l'écriture d'un fichier de données.
`LockFileAddToDataDirRead`	En attente d'une lecture lors de l'ajout d'une ligne dans le fichier de verrouillage du répertoire des données.
`LockFileAddToDataDirSync`	En attente que les données atteignent un stockage stable lors de l'ajout d'une ligne dans le fichier de verrouillage du répertoire des données.
`LockFileAddToDataDirWrite`	En attente d'une écriture lors de l'ajout d'une ligne dans le fichier de verrouillage du répertoire des données.
`LockFileCreateRead`	En attente d'une lecture lors de la création du fichier de verrouillage du répertoire des données.
`LockFileCreateSync`	En attente des données pour atteindre un stockage stable lors de la création du fichier de verrouillage du répertoire des données.
`LockFileCreateWrite`	En attente d'une écriture lors de la création du fichier de verrouillage du répertoire des données.
`LockFileReCheckDataDirRead`	En attente d'une lecture lors de la vérification du fichier de verrouillage du répertoire des données.
`LogicalRewriteCheckpointSync`	En attente que les correspondances de réécriture logique atteignent un stockage stable lors d'un checkpoint.
`LogicalRewriteMappingSync`	En attente que les données de correspondance atteignent un stockage stable lors d'une réécriture logique.
`LogicalRewriteMappingWrite`	En attente d'une écriture des données de correspondance lors d'une réécriture logique.

Wait Event `IO`	**Description**
`LogicalRewriteSync`	En attente que les correspondances de réécriture logique atteignent un stockage stable.
`LogicalRewriteTruncate`	En attente de la troncature de données de correspondance lors d'une réécriture logique.
`LogicalRewriteWrite`	En attente d'une écriture des correspondances de réécriture logique.
`RelationMapRead`	En attente d'une lecture du fichier de carte des relations.
`RelationMapSync`	En attente que le fichier de carte des relations atteigne un stockage stable.
`RelationMapWrite`	En attente d'une écriture dans le fichier de carte des relations.
`ReorderBufferRead`	En attente d'une lecture lors de la gestion des tampons réordonnés.
`ReorderBufferWrite`	En attente d'une écriture lors de la gestion des tampons réordonnés.
`ReorderLogicalMappingRead`	En attente d'une lecture d'une correspondance logique lors de la gestion des tampons réordonnés.
`ReplicationSlotRead`	En attente d'une lecture d'un fichier de contrôle d'un slot de réplication.
`ReplicationSlotRestoreSync`	En attente que le fichier de contrôle du slot de réplication atteigne un stockage stable pour le restaurer en mémoire.
`ReplicationSlotSync`	En attente que le fichier de contrôle du slot de réplication atteigne un stockage stable.
`ReplicationSlotWrite`	En attente de l'écriture d'un fichier de contrôle du slot de réplication.
`SLRUFlushSync`	En attente que les données SLRU atteignent un stockage stable lors d'un checkpoint ou d'un arrêt de base.
`SLRURead`	En attente d'une lecture d'une page SLRU.
`SLRUSync`	En attente que les données SLRU atteignent un stockage stable suivant l'écriture d'une page.
`SLRUWrite`	En attente d'une écriture d'une page SLRU.
`SnapbuildRead`	En attente d'une lecture d'une image d'un catalogue historique sérialisé.
`SnapbuildSync`	En attente qu'une image d'un catalogue historique sérialisé atteigne un stockage stable.
`SnapbuildWrite`	En attente d'une écriture d'une image d'un catalogue historique sérialisé.
`TimelineHistoryFileSync`	En attente qu'un fichier d'historique des timelines soit reçu via la réplication en flux pour atteindre un stockage stable.
`TimelineHistoryFileWrite`	En attente de l'écriture d'un fichier d'historique des timelines reçu via la réplication en flux.
`TimelineHistoryRead`	En attente de la lecture d'un fichier d'historique des timelines.

Wait Event IO	Description
TimelineHistorySync	En attente qu'un fichier d'historique des timelines nouvellement créé atteigne un stockage stable.
TimelineHistoryWrite	En attente de l'écriture d'un fichier d'historique des timelines nouvellement créé.
TwophaseFileRead	En attente de la lecture d'un fichier d'état pour le 2PC.
TwophaseFileSync	En attente que le fichier d'état pour le 2PC atteigne un stockage stable.
TwophaseFileWrite	En attente d'une écriture d'un fichier d'état pour le 2PC.
VersionFileSync	En attente que le fichier version atteigne un stockage stable lors de la création d'une base.
VersionFileWrite	En attente d'une écriture du fichier version lors de la création d'une base.
WALBootstrapSync	En attente que le WAL atteigne un stockage stable lors du bootstrapping.
WALBootstrapWrite	En attente d'un écriture d'une page WAL lors du bootstrapping.
WALCopyRead	En attente d'une lecture lors de la création d'un nouveau segment WAL en copiant un WAL existant.
WALCopySync	En attente d'un nouveau segment WAL copié un WAL existant pour atteindre un stockage stable.
WALCopyWrite	En attente d'une écriture lors de la création d'un nouveau segment WAL en en copiant un existant.
WALInitSync	En attente que le fichier WAL nouvellement initialisé atteigne un stockage stable.
WALInitWrite	En attente d'une écriture lors de l'initialisation d'un nouveau fichier WAL.
WALRead	En attente d'une lecture à partir d'un fichier WAL.
WALSenderTimelineHistoryRead	En attente d'une lecture à partir d'un fichier d'historique des timelines lors qu'une commande de timeline d'un walsender.
WALSync	En attente qu'un fichier WAL atteigne un stockage stable.
WALSyncMethodAssign	En attente que les données atteignent un stockage stable lors de l'affectation d'une nouvelle méthode de synchronisation des WAL.
WALWrite	En attente d'une écriture dans un fichier WAL.

Tableau 28.10. Wait Events de type IPC

Wait Event IPC	Description
AppendReady	En attente que les nœuds enfants d'un nœud Append soit prêt.

Wait Event `IPC`	**Description**
`ArchiveCleanupCommand`	Waiting for archive_cleanup_command to complete.
`ArchiveCommand`	Waiting for archive_command to complete.
`BackendTermination`	En attente de la fin d'un autre processus.
`BackupWaitWalArchive`	En attente que les fichiers WAL requis pour une sauvegarde soient correctement archivés.
`BgWorkerShutdown`	En attente de l'arrêt du background worker.
`BgWorkerStartup`	En attente du démarrage du background worker.
`BtreePage`	En attente que le numéro de page requis pour continuer un parcours parallélisé d'un index B-tree soit disponible.
`BufferIO`	En attente que les opérations I/O se terminent.
`CheckpointDone`	En attente de la fin d'un checkpoint.
`CheckpointStart`	En attente du démarrage d'un checkpoint.
`ExecuteGather`	En attente de l'activité d'un processus fils lors de l'exécution du nœud `Gather`.
`HashBatchAllocate`	En attente qu'un participant d'un Parallel Hash alloue une table de hachage.
`HashBatchElect`	En attente de l'élection d'un participant d'un Parallel Hash pour allouer une table de hachage.
`HashBatchLoad`	En attente que les autres participants d'un Parallel Hash finissent de charger une table de hachage.
`HashBuildAllocate`	En attente qu'un participant élu d'un Parallel Hash alloue la table de hachage initiale.
`HashBuildElect`	En attente de l'élection d'un participant à un Parallel Hash pour allouer la table de hachage.
`HashBuildHashInner`	En attente que les autres participants à un Parallel Hash finissent le hachage de la relation interne.
`HashBuildHashOuter`	En attente que les autres participants à un Parallel Hash finissent le partitionnement de la relation externe.
`RegisterSyncRequest`	Attente lors de l'envoi des demandes de synchronisation au checkpointer car la queue de demandes est remplie.
`HashGrowBatchesAllocate`	En attente qu'un participant élu à un Parallel Hash alloue plus de batches.
`HashGrowBatchesDecide`	En attente de l'élection d'un participant à un Parallel Hash pour décider de l'agrandissement futur du batch.
`HashGrowBatchesElect`	En attente de l'élection d'un participant à un Parallel Hash pour participer à l'allocation d'un plus grand nombre de batches.
`HashGrowBatchesFinish`	En attente de l'élection d'un participant à un Parallel Hash pour décider de l'agrandissement futur du batch.

Wait Event `IPC`	**Description**
`HashGrowBatchesRepartition`	En attente que les autres participants d'un Parallel Hash terminent le repartitionement.
`HashGrowBucketsAllocate`	En attente qu'un participant élu à un Parallel Hash fissent l'allocation d'un plus grand nombre de buckets.
`HashGrowBucketsElect`	En attente de l'élection d'un participant à un Parallel Hash pour allouer plus de buckets.
`HashGrowBucketsReinsert`	En attente que les autres participants d'un Parallel Hash finissent d'insérer des lignes dans les nouveaux buckets.
`LogicalSyncData`	En attente qu'un serveur distant de réplication logique envoie les données pour la synchronisation initiale de la table.
`LogicalSyncStateChange`	En attente qu'un serveur distant de réplication logique change l'état.
`MessageQueueInternal`	En attente qu'un autre processus soit attaché à une queue de messages partagé.
`MessageQueuePutMessage`	En attente de l'écriture d'un message de protocole dans une queue de messages partagé.
`MessageQueueReceive`	En attente de la réception d'octets à partir d'une queue de messages partagé.
`MessageQueueSend`	En attente de l'envoi d'octets vers une queue de messages partagé.
`ParallelBitmapScan`	En attente que le parcours parallélisé de bitmap soit initialisé.
`ParallelCreateIndexScan`	En attente que les workers parallélisés d'un `CREATE INDEX` finissent le parcours de la table.
`ParallelFinish`	En attente que les workers parallélisés finissent le traitement.
`ProcArrayGroupUpdate`	En attente que le leader du groupe efface l'identifiant de transaction à la fin d'une transaction.
`ProcSignalBarrier`	En attente qu'un événement *barrier* soit traité par tous les processus serveurs.
`Promote`	En attente de la promotion du standby.
`RecoveryConflictSnapshot`	En attente de la résolution du conflit de restauration pour un nettoyage par VACUUM.
`RecoveryConflictTablespace`	En attente de la résolution du conflit de restauration suite à la suppression d'un tablespace.
`RecoveryEndCommand`	Waiting for recovery_end_command to complete.
`RecoveryPause`	En attente de la remise en route de la restauration.
`ReplicationOriginDrop`	En attente que l'origine de réplication devienne inactive, pour qu'elle puisse être supprimée.

Wait Event IPC	Description
ReplicationSlotDrop	En attente que le slot de réplication devienne inactif pour qu'il puisse être supprimé.
RestoreCommand	Waiting for restore_command to complete.
SafeSnapshot	En attente de l'obtention d'une image valide pour une transaction READ ONLY DEFERRABLE.
SyncRep	En attente de la confirmation d'un serveur distant pour une réplication synchrone.
WalReceiverExit	En attente que le processus wal receiver quitte.
WalReceiverWaitStart	En attente que le processus de démarrage envoie les données d'initialisation pour la réplication en flux.
XactGroupUpdate	En attente du leader pour la mise à jour du statut de transaction à la fin d'une transaction.

Tableau 28.11. Wait Events de type Lock

Wait Event Lock	Description
advisory	En attente de l'acquisition d'un verrou informatif.
extend	En attente de l'agrandissement d'une relation.
frozenid	En attente de la mise à jour de pg_database.datfrozenxid et pg_database.datminmxid.
object	En attente de l'acquisition d'un verrou sur un objet de la base qui n'est pas une relation.
page	En attente de l'acquisition d'un verrou sur une page d'une relation.
relation	En attente de l'acquisition d'un verrou sur une relation.
spectoken	En attente de l'acquisition d'un verrou d'insertion spéculatif.
transactionid	En attente de la fin d'une transaction.
tuple	En attente de l'acquisition d'un verrou sur une ligne.
userlock	En attente de l'acquisition d'un verrou utilisateur.
virtualxid	En attente de l'acquisition d'un verrou sur un identifiant de transaction virtuel.

Tableau 28.12. Wait Events de type LWLock

Wait Event LWLock	Description
AddinShmemInit	En attente de la gestion d'une allocation d'espace de l'extension dans la mémoire partagée.
AutoFile	En attente de la mise à jour du fichier postgresql.auto.conf.
Autovacuum	En attente de la lecture ou de la mise à jour de l'état courant des workers de l'autovacuum.

Wait Event LWLock	Description
AutovacuumSchedule	En attente de l'assurance qu'une table sélectionnée pour l'autovacuum a toujours besoin d'être traité.
BackgroundWorker	En attente de la lecture ou de la mise à jour de l'état du background worker.
BtreeVacuum	En attente de la lecture ou de la mise à jour des informations liées au vacuum pour un index B-tree.
BufferContent	En attente de l'accès à une page de données en mémoire.
BufferMapping	En attente de l'association d'un bloc de données avec un tampon dans le cache.
CheckpointerComm	En attente de la gestion des demandes de fsync.
CommitTs	En attente de la lecture ou la mise à jour de la dernière valeur pour l'horodatage de la validation d'une transaction.
CommitTsBuffer	En attente des I/O sur un tampon SLRU d'horodatage de validation de transaction.
CommitTsSLRU	En attente de l'accès au cache SLRU d'horodatage de validation de transaction.
ControlFile	En attente de la lecture ou de la mise à jour du fichier pg_control ou de la création d'un nouveau fichier WAL.
DynamicSharedMemoryControl	En attente de lecture ou de la mise à jour des informations d'allocation de mémoire partagée dynamique.
LockFastPath	En attente de la lecture ou mise à jour des informations du verrou *fast-path* d'un processus.
LockManager	En attente de la lecture ou mise à jour des informations sur les verrous « heavyweight ».
LogicalRepWorker	En attente de la lecture ou de la mise à jour de l'état des workers de réplication logique.
MultiXactGen	En attente de la lecture ou de la mise à jour de l'état des identifiants partagés de multi-transactions.
MultiXactMemberBuffer	En attente des I/O sur un tampon SLRU pour les membres multixact.
MultiXactMemberSLRU	En attente de l'accès au cache SLRU des membres multixact.
MultiXactOffsetBuffer	En attente des I/O sur un tampon SLRU des décalages multixact.
MultiXactOffsetSLRU	En attente de l'accès au cache SLRU des décalages multixact.
MultiXactTruncation	En attente de la lecture ou de la troncature des informations multixact.
NotifyBuffer	En attente des I/O sur un tampon SLRU pour les messages NOTIFY.

Wait Event LWLock	Description
NotifyQueue	En attente de la lecture ou de la mise à jour des messages NOTIFY.
NotifyQueueTail	En attente de la mise à jour de la limite sur le stockage des messages NOTIFY.
NotifySLRU	En attente de l'accçs à un cache SLRU des messages NOTIFY.
OidGen	En attente de l'allocation d'un nouvel OID.
OldSnapshotTimeMap	En attente de la lecture ou de la mise à jour des informations sur le contrôle des images.
ParallelAppend	En attente de la sélection du prochain sous-plan dans l'exécution d'un nœud Parallel Append.
ParallelHashJoin	En attente de la synchronisation des workers lors de l'exécution d'un plan Parallel Hash Join.
ParallelQueryDSA	En attente de l'allocation de la mémoire partagée dynamique pour une requête parallélisée.
PerSessionDSA	En attente de l'allocation de la mémoire partagée dynamique pour une requête parallélisée.
PerSessionRecordType	En attente de l'accès aux informations d'une requête parallélisée sur les types composites.
PerSessionRecordTypmod	En attente de l'accès aux informations d'une requête parallélisée sur les modificateurs de type qui identifient les types de record anonymes.
PerXactPredicateList	En attente de l'accès à la liste des verrous de prédicat détenus par la transaction sérialisable actuelle lors d'une requête parallélisée.
PredicateLockManager	En attente de l'accès aux informations de verrou de prédicat utilisées par les transactions sérialisables.
ProcArray	En attente de l'accès aux structures de données partagées (typiquement, pour obtenir une image ou pour récupérer l'identifiant de transaction d'une session).
RelationMapping	En attente de la lecture ou de la mise à jour d'un fichier pg_filenode.map (utilisé pour tracer les affectations de numéros de fichiers de certains catalogues systèmes).
RelCacheInit	En attente de la lecture ou de la mise à jour du fichier d'initialisation du cache des relations, pg_internal.init.
ReplicationOrigin	En attente de la création, de la suppression ou de l'utilisation d'une origine de réplication.
ReplicationOriginState	En attente de la lecture ou la mise à jour de la progression d'une origine de réplication.
ReplicationSlotAllocation	En attente de l'allocation ou de la libération d'un slot de réplication.
ReplicationSlotControl	En attente de la lecture ou de la mise à jour d'un état d'un slot de réplication.
ReplicationSlotIO	En attente des I/O sur un slot de réplication.

Wait Event LWLock	Description
SerialBuffer	En attente des I/O sur un tampon SLRU pour la gestion des conflits des transactions sérialisables.
SerializableFinishedList	En attente de l'accès à la liste des transactions sérialisées terminées.
SerializablePredicateList	En attente de l'accès à la liste des verrous de prédicat détenus par les transactions sérialisées.
PgStatsDSA	Waiting for stats dynamic shared memory allocator access
PgStatsHash	Waiting for stats shared memory hash table access
PgStatsData	Waiting for shared memory stats data access
SerializableXactHash	En attente de la lecture ou de la mise à jour des informations sur les transactions sérialisables.
SerialSLRU	En attente de l'accès au cache SLRU de gestion des conflits pour les transactions sérialisées.
SharedTidBitmap	En attente de l'accès à une bitmap partagé de TID lors d'un parcours d'index bitmap parallélisé.
SharedTupleStore	En attente de l'accès à un stockage de ligne partagé lors d'une requête parallélisée.
ShmemIndex	En attente de la recherche ou de l'allocation d'espace en mémoire partagée.
SInvalRead	En attente de la récupération de messages en provenance de la queue d'invalidation des catalogues partagées.
SInvalWrite	En attente de l'ajout d'un message dans la queue d'invalidation des catalogues partagées.
SubtransBuffer	En attente des I/O sur un tampon SLRU d'une sous-transaction.
SubtransSLRU	En attente de l'accès à un cache SLRU de sous-transaction.
SyncRep	En attente de la lecture ou de la mise à jour d'informations sur l'état de la réplication synchrone.
SyncScan	En attente de la sélection de l'emplacement de démarrage d'un parcours synchronisé de table.
TablespaceCreate	En attente de la création ou suppression d'un tablespace.
TwoPhaseState	En attente de la lecture ou de la mise à jour de l'état des transactions préparées.
WALBufMapping	En attente du remplacement d'un bloc dans le cache des WAL.
WALInsert	En attente de l'insertion de données WAL dans un tampon mémoire.
WALWrite	En attente que les tampons WAL soient écrits sur disque.

Wait Event LWLock	Description
WrapLimitsVacuum	En attente de la mise à jour des limites sur la consommation des identifiants de transaction et des identifiants multixact.
XactBuffer	En attente des I/O sur un tampon SLRU de statut de transaction.
XactSLRU	En attente de l'accès au cache SLRU de statut de transaction.
XactTruncation	En attente de l'exécution de pg_xact_status ou de la mise à jour du plus ancien identifiant de transaction disponible.
XidGen	En attente de l'allocation d'un nouvel identifiant de transaction.

> **Note**
>
> Les extensions peuvent ajouter des types LWLock à la liste affichée dans le Tableau 28.12. Dans certains cas, le nom affecté par une extension ne sera pas disponible dans tous les processus serveurs ; donc un *wait event* de type LWLock pourrait être rapporté comme simplement une « extension » plutôt que comme le nom affecté par l'extension.

Tableau 28.13. Wait Events de type Timeout

Wait Event Timeout	Description
BaseBackupThrottle	En attente pendant la sauvegarde de base lors de la limitation de l'activité.
CheckpointWriteDelay	En attente entre les écritures d'un checkpoint.
PgSleep	En attente suite à un appel à pg_sleep ou à une fonction du même type.
RecoveryApplyDelay	En attente de l'application des WAL lors de la restauration à cause du paramétrage d'un délai.
RecoveryRetrieveRetryInterval	En attente de la restauration quand les données WAL ne sont pas disponibles quelque soit la source (pg_wal, archive ou flux).
VacuumDelay	En attente de la fin d'un délai du VACUUM après dépassement d'un délai.
VacuumTruncate	Waiting to acquire an exclusive lock to truncate off any empty pages at the end of a table vacuumed.

Voici un exemple pour voir les *wait events* :

```
SELECT pid, wait_event_type, wait_event FROM pg_stat_activity WHERE
 wait_event is NOT NULL;

 pid  | wait_event_type | wait_event
------+-----------------+------------
 2540 | Lock            | relation
 6644 | LWLock          | ProcArray
(2 rows)
```

28.2.4. pg_stat_replication

La vue pg_stat_replication contiendra une ligne par processus d'envoi de WAL, montrant des statistiques sur la réplication avec le serveur standby connecté au processus. Seuls les serveurs standby directement connectés sont listés ; aucune information n'est disponible concernant les serveurs standby en aval.

Tableau 28.14. Vue pg_stat_replication

Type Description
pid integer Identifiant du processus d'envoi des WAL
usesysid oid OID de l'utilisateur connecté à ce processus
usename name Nom de l'utilisateur connecté à ce processus
application_name text Nom de l'application qui est connectée à ce processus
client_addr inet Adresse IP du client connecté à ce processus. Si ce champ est NULL, ceci signifie que le client est connecté via un socket Unix sur la machine serveur.
client_hostname text Nom de l'hôte du client connecté, comme renvoyé par une recherche DNS inverse sur client_addr. Ce champ sera uniquement non NULL pour les connexions IP, et seulement si log_hostname est activé.
client_port integer Numéro du port TCP que le client utilise pour la communication avec ce processus, ou -1 si un socket Unix est utilisée.
backend_start timestamp with time zone Heure à laquelle ce processus a été démarré, exemple, lorsque le client s'est connecté à ce processus expéditeur de WALs.
backend_xmin xid L'horizon xmin de ce serveur standby renvoyé par hot_standby.
state text État courant du processus walsender. Les valeurs possibles sont : • startup : Le processus walsender est en cours de démarrage. • catchup : Le secondaire connecté au processus walsender est en cours de rattrapage du primaire. • streaming : Ce processus walsender envoie les modifications au serveur secondaire connecté depuis que ce dernier a rattrapé le primaire. • backup : Ce processus walsender est en train d'envoyer une sauvegarde. • stopping : Ce processus walsender est en cours d'arrêt.
sent_lsn pg_lsn La position de la dernière transaction envoyée sur cette connexion
write_lsn pg_lsn La position de la dernière transaction écrite sur disque par ce serveur standby
flush_lsn pg_lsn

Type Description
La position de la dernière transaction vidée sur disque par ce serveur standby
`replay_lsn` `pg_lsn` La position de la dernière transaction rejouée dans la base de données par ce serveur standby
`write_lag` `interval` Durée passée entre le vidage local des WAL récents et la réception de notification que ce serveur secondaire les a bien écrites (mais pas encore vidées ou appliquées). Ceci peut être utilisé pour mesurer le délai que le niveau `remote_write` de `synchronous_commit` coûterait lors de la validation si ce serveur était configuré comme un serveur secondaire synchrone.
`flush_lag` `interval` Durée passée entre le vidage local des WAL récents et la réception de notification que ce serveur secondaire les a bien écrites et vidées sur disque (mais pas encore appliquées). Ceci peut être utilisé pour mesurer le délai que le niveau `on` de `synchronous_commit` coûterait lors de la validation si ce serveur était configuré comme un serveur secondaire synchrone.
`replay_lag` `interval` Durée passée entre le vidage local des WAL récents et la réception de notification que ce serveur secondaire les a bien écrites, vidées sur disque et appliquées. Ceci peut être utilisé pour mesurer le délai que le niveau `remote_apply` de `synchronous_commit` coûterait lors de la validation si ce serveur était configuré comme un serveur secondaire synchrone.
`sync_priority` `integer` Priorité de ce serveur standby pour être choisi comme le serveur `standby` synchrone dans une réplication synchrone basée sur la priorité. Ceci n'a pas d'effet sur une réplication synchrone basée sur un quorum.
`sync_state` `text` État synchrone de ce serveur standby. Les valeurs possibles sont : • `async` : Ce serveur standby est asynchrone. • `potential` : Ce serveur standby est maintenant asynchrone, mais peut potentiellement devenir synchrone si un des synchrones échoue. • `sync` : Ce serveur standby est synchrone. • `quorum` : Ce serveur standby est considéré comme un candidat dans les standbys avec quorum.
`reply_time` `timestamp with time zone` Horodatage d'envoi du dernier message de réponse reçu par le standby

Les délais rapportés dans la vue `pg_stat_replication` sont des mesures de temps prises pour l'écriture, le vidage sur disque et le rejeu des données récentes des WAL et pour que le serveur d'envoi soit mis au courant. Ces durées représentent le délai de validation qui a été (ou aurait été) introduit par chaque niveau de validation synchrone si le serveur distant était configuré comme un standby synchrone. Pour un standby asynchrone, la colonne `replay_lag` renvoie une approximation du délai avant que les transactions récentes deviennent visibles aux requêtes. Si le serveur standby a complètement rattrapé le serveur d'envoi et qu'il n'y a plus d'activité en écriture (donc plus de nouveaux enregistrements dans les journaux de transactions), les délais mesurés le plus récemment continueront à être affichés pendant un court instant, puis seront mis à NULL.

Les délais fonctionnent automatiquement pour la réplication physique. Les plugins de décodage logique pourraient émettre des messages de trace. S'ils ne le font pas, le mécanisme de trace affichera simplement une valeur NULL.

Note

Les délais rapportés ne sont pas des prédictions du temps pris par le serveur standby pour rattraper le serveur d'envoi en constatant le taux actuel de rejeu. Un tel système afficherait des temps similaires alors que de nouveaux journaux de transactions seraient générés, mais différeraient lorsque le serveur deviendrait inactif. En particulier, quand le serveur standby a complètement rattrapé le serveur d'envoi, `pg_stat_replication` affiche le temps pris pour écrire, vider sur disque et rejouer l'emplacement de l'enregistrement le plus récemment rapporté plutôt que zéro comme certains utilisateurs pourraient s'y attendre. Ceci est cohérent avec le but de mesurer les délais de la validation synchrone et de la visibilité des transactions pour les transactions récentes en écriture. Pour réduire la confusion pour les utilisateurs s'attendant à un autre modèle de retard, les colonnes de retard sont réinitialisées à NULL après un court moment sur un système entièrement à jour et complètement inactif. Les systèmes de supervision devraient choisir s'ils souhaitent représenter ces colonnes comme des données manquantes, des données à zéro, ou continuer à afficher la dernière valeur connue.

28.2.5. `pg_stat_replication_slots`

La vue `pg_stat_replication_slots` contiendra une ligne par slot de réplication logique, montrant les statistiques de son utilisation.

Tableau 28.15. `pg_stat_replication_slots` View

Type de la colonne **Description**
`slot_name` `text` Un identifiant, unique à l'instance, pour le slot de réplication
`spill_txns` `bigint` Nombre de transactions déversées sur disque une fois que la mémoire utilisée par le décodage logique pour décoder les changements depuis les WAL, a dépassé `logical_decoding_work_mem`. Le compteur est incrémenté pour à la fois les transactions de plus haut niveau et les sous-transactions.
`spill_count` `bigint` Nombre de fois où des transactions sont déversées sur disque pendant le décodage des changements des WAL pour ce slot. Ce compteur est incrémenté chaque fois qu'une transaction est déversée, et une même transaction peut déverser plusieurs fois.
`spill_bytes` `bigint` Quantité de données de transaction décodées déversées sur disque pendant le décodage des changements des WAL pour ce slot. Ce compteur et ceux liés au déversement sur disque peuvent être utilisés pour mesurer les E/S qui se produisent lors du décodage logique et permettre d'optimiser `logical_decoding_work_mem`.
`stream_txns` `bigint` Nombre de transactions en cours envoyées au plugin de sortie de décodage logique après que la mémoire utilisée par le décodage logique pour décoder les changements depuis les WAL pour ce slot a dépassé `logical_decoding_work_mem`. Le flux de réplication ne fonctionne qu'avec les transactions de plus haut niveau (les sous-transactions ne sont pas envoyées indépendemment), ainsi le compteur n'est pas incrémenté pour les sous-transactions.
`stream_countbigint` Nombre de fois ou des transactions en cours sont envoyées au plugin de sortie de décodage logique lors du décodage des changements depuis les WAL pour ce slot. Ce compteur est incrémenté chaque fois qu'une transaction est envoyée, et la même transaction peut être envoyée plusieurs fois.

Type de la colonne Description
`stream_bytes` `bigint` Quantité de données de transactions décodées pour les transactions en cours envoyées par flux au plugin de sortie de décodage logique pendant que des changements sont décodés pour ce slot depuis les WAL. Ce compteur et les compteurs similaires liés pour ce slot peuvent être utilisés pour optimiser `logical_decoding_work_mem`.
`total_txns` `bigint` Nombre de transactions décodées envoyées au plugin de sortie de décodage pour ce slot. Cela ne comprend que les transactions de plus haut niveau pour ce slot, et ce compteur n'est pas incrémenté pour les sous-transactions. Notez que cela inclut les transactions qui sont envoyées par flux et/ou déversées sur disque.
`total_bytes` `bigint` Quantité de données de transactions décodées pour envoyer les transactions au plugin de sortie de décodage pendant que les changements sont décodés depuis les WAL pour ce slot. Notez que cela inclut les transactions qui sont envoyées par flux et/ou déversées sur disque.
`stats_reset` `timestamp with time zone` Date où ces statistiques ont été remises à zéro.

28.2.6. `pg_stat_wal_receiver`

La vue `pg_stat_wal_receiver` contiendra seulement une ligne, affichant les statistiques du walreceiver du serveur de connexion.

Tableau 28.16. Vue `pg_stat_wal_receiver`

Type Description
`pid` `integer` Identifiant du processus de réception des enregistrements de transaction
`status` `text` Statut d'activité du processus walreceiver
`receive_start_lsn` `pg_lsn` Première position dans le journal de transaction utilisée quand walreceiver a été démarré
`receive_start_tli` `integer` Première ligne de temps utilisée quand walreceiver a été démarré
`written_lsn` `pg_lsn` Dernier emplacement dans les journaux de transactions ayant déjà été reçu et écrit sur disque, mais non vidé du cache. Ceci ne doit pas être utilisé pour des vérifications sur l'intégrité des données.
`flushed_lsn` `pg_lsn` Dernier emplacement dans les journaux de transactions ayant déjà été reçu et écrit sur disque, et le cache vidé, la valeur initiale de ce champ étant le premier emplacement utilisé lorsque le processus walreceiver a été démarré.
`received_tli` `integer` Numéro de la ligne de temps de la dernière position des journaux de transactions, déjà reçue et écrite sur disque, la valeur initiale de ce champ étant la ligne de temps de la première position dans les journaux de transactions utilisée lors du démarrage du walreceiver
`last_msg_send_time` `timestamp with time zone` Horodatage d'envoi du dernier message reçu à partir du walsender

Type Description
`last_msg_receipt_time` `timestamp with time zone` Horodatage de la réception du dernier message à partir du walsender
`latest_end_lsn` `pg_lsn` Dernière position de transaction reportée par le walsender associé
`latest_end_time` `timestamp with time zone` Horodatage de la dernière position de transaction reportée par le walsender associé
`slot_name` `text` Nom du slot de réplication utilisé par ce walreceiver
`sender_host` `text` Hôte de l'instance PostgreSQL auquel ce processus « wal receiver » est connecté. Il peut s'agir d'un nom d'hôte, d'une adresse `IP` ou d'un chemin d'accès à un répertoire si la connexion se fait via un socket Unix (dans ce dernier cas, il est facile de le distinguer car il s'agira toujours d'un chemin absolu débutant par le caractère (`/`).)
`sender_port` `integer` Numéro de port de l'instance PostgreSQL auquel wal receiver est connecté.
`conninfo` `text` Chaîne de connexion utilisée par ce wal receiver, les informations sensibles au niveau sécurité sont cachés.

28.2.7. `pg_stat_recovery_prefetch`

The `pg_stat_recovery_prefetch` view will contain only one row. The columns `wal_distance`, `block_distance` and `io_depth` show current values, and the other columns show cumulative counters that can be reset with the `pg_stat_reset_shared` function.

Tableau 28.17. `pg_stat_recovery_prefetch` View

Column Type Description
`stats_reset` `timestamp with time zone` Time at which these statistics were last reset
`prefetch` `bigint` Number of blocks prefetched because they were not in the buffer pool
`hit` `bigint` Number of blocks not prefetched because they were already in the buffer pool
`skip_init` `bigint` Number of blocks not prefetched because they would be zero-initialized
`skip_new` `bigint` Number of blocks not prefetched because they didn't exist yet
`skip_fpw` `bigint` Number of blocks not prefetched because a full page image was included in the WAL
`skip_rep` `bigint` Number of blocks not prefetched because they were already recently prefetched
`wal_distance` `int` How many bytes ahead the prefetcher is looking
`block_distance` `int` How many blocks ahead the prefetcher is looking
`io_depth` `int` How many prefetches have been initiated but are not yet known to have completed

28.2.8. pg_stat_subscription

La vue pg_stat_subscription contiendra une ligne par souscription du worker principal (avec le PID NULL si le processus worker n'est pas en cours d'exécution), et des lignes supplémentaires pour les workers gérant la copie initiale de données des tables souscrites.

Tableau 28.18. Vue pg_stat_subscription

Type Description
subid oid OID de la souscription
subname name Nom de la souscription
pid integer Identifiant du processus worker de la souscription
relid oid OID de la relation que le processus worker synchronise ; NULL pour le processus worker apply principal
received_lsn pg_lsn Dernier emplacement de journal de transactions reçu, la valeur initiale de ce champ étant 0
last_msg_send_time timestamp with time zone Horodatage d'envoi du dernier message reçu à partir du walsender original
last_msg_receipt_time timestamp with time zone Horodatage de réception du dernier message reçu du walsender original
latest_end_lsn pg_lsn Dernier emplacement des journaux de transactions rapporté par le walsender original
latest_end_time timestamp with time zone Horodatage du dernier emplacement de journal de transactions rapporté par le walsender original

28.2.9. pg_stat_subscription_stats

The pg_stat_subscription_stats view will contain one row per subscription.

Tableau 28.19. pg_stat_subscription_stats View

Column Type Description
subid oid OID of the subscription
subname name Name of the subscription
apply_error_count bigint Number of times an error occurred while applying changes
sync_error_count bigint Number of times an error occurred during the initial table synchronization
stats_reset timestamp with time zone Time at which these statistics were last reset

28.2.10. pg_stat_ssl

La vue `pg_stat_ssl` contiendra une ligne par backend ou processus d'envoi de WAL, montrant des statistiques sur l'usage de SSL dans cette connexion. Elle peut être jointe à `pg_stat_activity` ou `pg_stat_replication` sur la colonne `pid` pour obtenir plus de détails sur la connexion.

Tableau 28.20. Vue `pg_stat_ssl`

Type Description
`pid` `integer` ID du processus backend ou du processus d'envoi de WAL
`ssl` `boolean` True si SSL est utilisé dans cette connexion
`version` `text` Version de SSL utilisée, ou NULL si SSL n'est pas utilisé pour cette connexion
`cipher` `text` Nom du chiffrement SSL utilisé, ou NULL si SSL n'est pas utilisé pour cette connexion
`bits` `integer` Nombre de bits dans l'algorithme de chiffrement utilisé, ou NULL si SSL n'est pas utilisé pour cette connexion
`client_dn` `text` Champ Distinguished Name (DN) utilisé par le certificat du client, ou NULL si aucun certificat client n'a été fourni ou si SSL n'est pas utilisé pour cette connexion. Ce champ est tronqué si le champ DN est plus long que `NAMEDATALEN` (64 caractères dans une compilation standard).
`client_serial` `numeric` Numéro de série du certificat client ou NULL si aucun certificat n'a été fourni ou si le SSL n'est pas utilisé dans cette connexion. La combinaison du numéro de série du certification et de l'émetteur du certificat identifie de façon unique un certificat (sauf si l'émetteur réutilise par erreur les numéros de série).
`issuer_dn` `text` DN de l'émetteur du certificat client ou NULL si aucun certificat client n'a été fourni ou si le SSL n'est pas utilisé dans cette connexion. Ce champ est tronqué comme `client_dn`.

28.2.11. `pg_stat_gssapi`

La vue `pg_stat_gssapi` contient une ligne par processus serveur, affichant des informations sur l'utilisation de GSSAPI pour cette connexion. Elle peut être jointe à `pg_stat_activity` ou `pg_stat_replication` sur la colonne `pid` pour obtenir plus de détails sur la connexion.

Tableau 28.21. Vue `pg_stat_gssapi`

Type Description
`pid` `integer` Identifiant du processus serveur
`gss_authenticated` `boolean` True si l'authentification GSSAPI a été utilisée pour cette connexion
`principal` `text` Principal utilisé pour authentifier cette connexion, ou NULL si GSSAPI n'a pas été utilisé pour authentifier cette connexion. Ce champ est tronqué si le principal est plus long que `NAMEDATALEN` (64 caractères dans une construction standard).
`encrypted` `boolean` True si le chiffrement GSSAPI est utilisé avec cette connexion

28.2.12. pg_stat_archiver

La vue pg_stat_archiver aura toujours une seule ligne contenant les données du processus d'archivage de l'instance.

Tableau 28.22. Vue pg_stat_archiver

Type Description
archived_count bigint Nombre de journaux de transactions archivés avec succès
last_archived_wal text Nom du plus récent journal de transaction archivé avec succès
last_archived_time timestamp with time zone Horodatage de la plus récente opération d'archivage réussie
failed_count bigint Nombre d'échecs d'archivage de journaux de transactions
last_failed_wal text Nom du journal de transactions correspondant au plus récent archivage échoué
last_failed_time timestamp with time zone Horodatage de la plus récente opération d'archivage échouée
stats_reset timestamp with time zone Horodatage de la dernière réinitialisation de ces statistiques

Normally, WAL files are archived in order, oldest to newest, but that is not guaranteed, and does not hold under special circumstances like when promoting a standby or after crash recovery. Therefore it is not safe to assume that all files older than last_archived_wal have also been successfully archived.

28.2.13. pg_stat_bgwriter

La vue pg_stat_bgwriter aura toujours une ligne unique, contenant les données globales de l'instance.

Tableau 28.23. Vue pg_stat_bgwriter

Type Description
checkpoints_timed bigint Nombre de checkpoints planifiés ayant été effectués
checkpoints_req bigint Nombre de checkpoints demandés ayant été effectués
checkpoint_write_time double precision Temps total passé dans la partie des checkpoints où les fichiers sont écrits sur disque, en millisecondes.
checkpoint_sync_time double precision Temps total passé dans la partie des checkpoints où les fichiers sont synchronisés sur le disque, en millisecondes.
buffers_checkpoint bigint Nombre de tampons écrits durant des checkpoints
buffers_clean bigint Nombre de tampons écrits par le processus background writer (processus d'écriture en tâche de fond)

Type Description
`maxwritten_clean` `bigint` Nombre de fois que le processus background writer a arrêté son parcours de nettoyage pour avoir écrit trop de tampons
`buffers_backend` `bigint` Nombre de tampons écrits directement par un processus serveur
`buffers_backend_fsync` `bigint` Nombre de fois qu'un processus serveur a du exécuter son propre appel à `fsync` (normalement le processus background writer gère ces appels même quand le processus serveur effectue sa propre écriture)
`buffers_alloc` `bigint` Nombre de tampons alloués
`stats_reset` `timestamp with time zone` Dernière fois que ces statistiques ont été réinitialisées

28.2.14. `pg_stat_wal`

La vue `pg_stat_wal` a toujours une ligne, contenant les données sur l'activité des journaux de transaction de l'instance.

Tableau 28.24. `pg_stat_wal` View

Type de la colonne Description
`wal_records` `bigint` Nombre total d'enregistrements générés dans les journaux de transaction
`wal_fpi` `bigint` Nombre total d'images de pages complètes générées dans les journaux de transactions
`wal_bytes` `numeric` Quantité totale de journaux de transactions générés en octets.
`wal_buffers_full` `bigint` Nombre de fois où des données de journaux de transactions ont été écrites sur disque parce que le cache des journaux de transactions était rempli
`wal_write` `bigint` Nombre de fois où les tampons de journaux de transactions ont été écrits sur disque via des appels à `XLogWrite`. Voir Section 30.5 pour plus d'informations sur la fonction interne de journaux de transactions `XLogWrite`.
`wal_sync` `bigint` Nombre de fois où les fichiers des journaux de transactions ont été synchronisés sur disque via des appels à `issue_xlog_fsync` (si fsync est à `on` et wal_sync_method vaut soit `fdatasync`, `fsync` soit `fsync_writethrough`), sinon zéro. Voir Section 30.5 pour plus d'informations sur la fonction interne des journaux de transaction `issue_xlog_fsync`.
`wal_write_time` `double precision` Temps total dépensé à l'écriture du cache des journaux de transactions sur disque via des appels à `XLogWrite`, en millisecondes (si track_wal_io_timing est activé, sinon zéro). Cela inclut le temps de synchronisation quand `wal_sync_method` vaut soit `open_datasync` soit `open_sync`.
`wal_sync_time` `double precision` Temps total dépensé à synchroniser les fichiers des journaux de transactions vers le disque via des appels à `issue_xlog_fsync`, en millisecondes (si

Type de la colonne Description
`track_wal_io_timing` est activé, et `fsync` vaut `on`, et `wal_sync_method` vaut soit `fdatasync`, soit `fsync` soit `fsync_writethrough`). Sinon zéro.
`stats_reset` `timestamp with time zone` Date où ces statistiques ont été remises à zéro

28.2.15. `pg_stat_database`

La vue `pg_stat_database` ne contiendra qu'une ligne pour chaque base de données dans l'instance, plus une pour les objets partagés, montrant ses statistiques globales.

Tableau 28.25. Vue `pg_stat_database`

Type Description
`datid` `oid` OID d'une base de données, ou 0 pour les objets partagés
`datname` `name` Nom de cette base de données, or `NULL` pour les objets partagés
`numbackends` `integer` Nombre de processus serveur actuellement connectés à cette base de données, ou `NULL` pour les objets partagés. C'est la seule colonne de cette vue qui renvoie une valeur reflétant l'état actuel ; toutes les autres colonnes renvoient les valeurs accumulées depuis la dernière réinitialisation
`xact_commit` `bigint` Nombre de transactions de cette base de données qui ont été validées
`xact_rollback` `bigint` Nombre de transactions de cette base de données qui ont été annulées
`blks_read` `bigint` Nombre de blocs disques lus dans cette base de données
`blks_hit` `bigint` Nombre de fois que des blocs disques étaient déjà dans le cache tampon, et qu'il n'a donc pas été nécessaire de les lire sur disque (cela n'inclut que les accès dans le cache tampon de PostgreSQL, pas dans le cache de fichiers du système d'exploitation).
`tup_returned` `bigint` Number of live rows fetched by sequential scans and index entries returned by index scans in this database
`tup_fetched` `bigint` Number of live rows fetched by index scans in this database
`tup_inserted` `bigint` Nombre de lignes insérées par des requêtes dans cette base de données
`tup_updated` `bigint` Nombre de lignes mises à jour par des requêtes dans cette base de données
`tup_deleted` `bigint` Nombre de lignes supprimées par des requêtes dans cette base de données
`conflicts` `bigint` Nombre de requêtes annulées à cause de conflits avec la restauration dans cette base de données. (Les conflits n'arrivent que sur des serveurs de standby ; voir `pg_stat_database_conflicts` pour plus de détails.)
`temp_files` `bigint`

Type Description
Nombre de fichiers temporaires créés par des requêtes dans cette base de données. Tous les fichiers temporaires sont comptabilisés, quel que soit la raison de la création du fichier temporaire (par exemple, un tri ou un hachage) et quel que soit la valeur du paramètre log_temp_files.
`temp_bytes` `bigint` Quantité totale de données écrites dans des fichiers temporaires par des requêtes dans cette base de données. Tous les fichiers temporaires sont comptabilisés, quel que soit la raison de la création de ce fichier temporaire, et de la valeur du paramètre log_temp_files.
`deadlocks` `bigint` Nombre de verrous mortels détectés dans cette base de données
`checksum_failures` `bigint` Nombre d'échecs pour la vérification des sommes de contrôle détectés sur cette base de données (ou sur un objet partagé), ou NULL si les sommes de contrôle ne sont pas activées.
`checksum_last_failure` `timestamp with time zone` Horodatage du dernier échec de vérification de somme de contrôle pour un bloc détecté dans cette base (ou sur un objet partagé), ou NULL si les sommes de contrôle ne sont pas activés.
`blk_read_time` `double precision` Temps passé à lire des blocs de donnée dans des fichiers par des processus serveur dans cette base de données, en millisecondes (si track_io_timing est activé, sinon zéro)
`blk_write_time` `double precision` Temps passé à écrire des blocs de données dans des fichiers par les processus serveur dans cette base de données, en millisecondes (si track_io_timing est activé, sinon zéro)
`session_time` `double precision` Temps passé à écrire par les sessions de cette base de données, en millisecondes (notez que les statistiques sont seulement mises à jour quand l'état d'une session change, ainsi si des sessions ont été inactives longtemps, ce temps d'inactivité ne sera pas inclus)
`active_time` `double precision` Temps passé à exécuter des expressions SQL dans cette base de données, en millisecondes (ceci correspond à l'état `actif` ainsi qu'à l'`appel de fonction fastpath` dans `pg_stat_activity`)
`idle_in_transaction_time` `double precision` Temps passé inactif dans une transaction dans cette base de données, en millisecondes (cela correspond aux états `inactif dans une transaction` (« idle in transaction ») et `inactif dans une transaction (annulé)` (« idle in transaction (aborted) ») dans `pg_stat_activity`)
`sessions` `bigint` Nombre total de sessions établies dans cette base de données
`sessions_abandoned` `bigint` Nombre de sessions dans cette base de données qui ont été terminées parce que la connexion au client a été perdue
`sessions_fatal` `bigint` Nombre de sessions dans cette base de données qui ont été terminées par des erreurs fatales
`sessions_killed` `bigint` Nombre de sessions dans cette base de données qui ont été terminées par l'intervention d'un opérateur
`stats_reset` `timestamp with time zone`

Type Description
Dernière fois que ces statistiques ont été réinitialisées

28.2.16. pg_stat_database_conflicts

> **Avertissement**
>
> Utiliser `pg_stat_reset()` réinitialise aussi les compteurs que l'autovacuum utilise pour déterminer quand déclencher une opération VACUUM ou une opération ANALYZE. Réinitialiser ces compteurs peut empêcher l'autovacuum de réaliser un travail pourtant nécessaire, ce qui entrainerait comme conséquence une fragmentation des tables ou des statistiques obsolètes sur les données des tables. Un ANALYZE sur la base est recommandé après avoir réinitialisé les statistiques.

La vue `pg_stat_database_conflicts` contiendra une ligne par base de données, montrant des statistiques au niveau de chaque base de données concernant les requêtes annulées survenant à cause de conflits avec la restauration sur des serveurs standby. Cette vue contiendra seulement des informations sur les serveurs standby, dans la mesure où aucun conflit ne survient sur les serveurs primaires.

Tableau 28.26. Vue pg_stat_database_conflicts

Type Description
`datid` `oid` OID de la base de données
`datname` `name` Nom de cette base de données
`confl_tablespace` `bigint` Nombre de requêtes dans cette base de données qui ont été annulées suite à la suppression de tablespaces
`confl_lock` `bigint` Nombre de requêtes dans cette base de données qui ont été annulées suite à des délais dépassés sur des verrouillages
`confl_snapshot` `bigint` Nombre de requêtes dans cette base de données qui ont été annulées à cause d'instantanés trop vieux
`confl_bufferpin` `bigint` Nombre de requêtes dans cette base de données qui ont été annulées à cause de tampons verrouillés
`confl_deadlock` `bigint` Nombre de requêtes dans cette base de données qui ont été annulées à cause de `deadlocks`

28.2.17. pg_stat_all_tables

La vue `pg_stat_all_tables` contiendra une ligne par table dans la base de données courante (incluant les tables TOAST), montrant les statistiques d'accès pour cette table spécifiquement. Les vues `pg_stat_user_tables` et `pg_stat_sys_tables` contiennent les mêmes informations, mais filtrent respectivement les tables utilisateurs et les tables systèmes.

Tableau 28.27. Vue `pg_stat_all_tables`

Type Description
`relid` `oid` OID d'une table
`schemaname` `name` Nom du schéma dans lequel se trouve cette table
`relname` `name` Nom de cette table
`seq_scan` `bigint` Nombre de parcours séquentiels initiés sur cette table
`seq_tup_read` `bigint` Nombre de lignes vivantes rapportées par des parcours séquentiels
`idx_scan` `bigint` Nombre de parcours d'index initiés sur cette table
`idx_tup_fetch` `bigint` Nombre de lignes vivantes rapportées par des parcours d'index
`n_tup_ins` `bigint` Nombre de lignes insérées
`n_tup_upd` `bigint` Nombre de lignes mises à jour (y compris les lignes mises à jour par HOT)
`n_tup_del` `bigint` Nombre de lignes supprimées
`n_tup_hot_upd` `bigint` Nombre de lignes mises à jour par HOT (c'est-à-dire sans mises à jour d'index nécessaire)
`n_live_tup` `bigint` Nombre estimé de lignes vivantes
`n_dead_tup` `bigint` Nombre estimé de lignes mortes
`n_mod_since_analyze` `bigint` Nombre estimé de lignes modifiées depuis le dernier ANALYZE sur cette table
`n_ins_since_vacuum` `bigint` Nombre estimé de lignes insérées depuis le dernier VACUUM sur cette table
`last_vacuum` `timestamp with time zone` Dernière fois qu'une opération VACUUM manuelle a été faite sur cette table (sans compter `VACUUM FULL`)
`last_autovacuum` `timestamp with time zone` Dernière fois que le démon autovacuum a exécuté une opération VACUUM sur cette table
`last_analyze` `timestamp with time zone` Dernière fois qu'une opération ANALYZE a été lancée manuellement sur cette table
`last_autoanalyze` `timestamp with time zone` Dernière fois que le démon autovacuum a exécuté une opération ANALYZE sur cette table
`vacuum_count` `bigint` Nombre de fois qu'une opération VACUUM manuelle a été lancée sur cette table (sans compter `VACUUM FULL`)
`autovacuum_count` `bigint`

Type Description
Nombre de fois que le démon autovacuum a exécuté une opération VACUUM manuelle
`analyze_count` `bigint` Nombre de fois qu'une opération ANALYZE manuelle a été lancée sur cette table
`autoanalyze_count` `bigint` Nombre de fois que le démon autovacuum a exécuté une opération ANALYZE sur cette table

28.2.18. `pg_stat_all_indexes`

La vue `pg_stat_all_indexes` contiendra une ligne pour chaque index dans la base de données courante, montrant les statistiques d'accès sur cet index spécifiquement. Les vues `pg_stat_user_indexes` et `pg_stat_sys_indexes` contiennent la même information, mais sont filtrées pour ne montrer respectivement que les index utilisateurs et les index système.

Tableau 28.28. Vue `pg_stat_all_indexes`

Type Description
`relid` `oid` OID de la table pour cet index
`indexrelid` `oid` OID de cet index
`schemaname` `name` Nom du schéma dans lequel se trouve cet index
`relname` `name` Nom de la table pour cet index
`indexrelname` `name` Nom de cet index
`idx_scan` `bigint` Nombre de parcours d'index initiés par cet index
`idx_tup_read` `bigint` Nombre d'entrées d'index retournées par des parcours sur cet index
`idx_tup_fetch` `bigint` Nombre de lignes vivantes de la table rapportées par des simples parcours d'index utilisant cet index

Les index peuvent être utilisés avec un simple parcours d'index, un parcours d'index « bitmap » ou l'optimiseur. Dans un parcours de bitmap, les sorties de plusieurs index peuvent être combinées avec des règles AND ou OR, c'est pourquoi il est difficile d'associer des lectures de lignes individuelles de la table avec des index spécifiques quand un parcours de bitmap est utilisé. Par conséquent, un parcours de bitmap incrémente le(s) valeur(s) de `pg_stat_all_indexes`. `idx_tup_read` pour le(s) index qu'il utilise, et incrémente la valeur de `pg_stat_all_tables`. `idx_tup_fetch` pour la table, mais il n'affecte pas `pg_stat_all_indexes`.`idx_tup_fetch`. L'optimiseur accède également aux index pour vérifier si des constantes fournies sont en dehors des plages de valeurs enregistrées par les statistiques de l'optimiseur car celles-ci peuvent ne pas être à jour.

> **Note**
>
> Les valeurs de `idx_tup_read` et `idx_tup_fetch` peuvent être différentes même sans aucune utilisation de parcours de bitmap, car `idx_tup_read` comptabilise les entrées d'index récupérées de cet index alors que `idx_tup_fetch` comptabilise le nombre de lignes

vivantes rapportées de la table. Le second sera moindre si des lignes mortes ou pas encore validées sont rapportées en utilisant l'index, ou si des lectures de lignes de la table sont évitées grâce à un parcours d'index seul.

28.2.19. `pg_statio_all_tables`

La vue `pg_statio_all_tables` contiendra une ligne pour chaque table dans la base de données courante (en incluant les tables TOAST), montrant les statistiques d'entrées/sorties de chaque table spécifiquement. Les vues `pg_statio_user_tables` et `pg_statio_sys_tables` contiennent la même information, mais sont filtrées pour ne montrer respectivement que les tables utilisateurs et les tables système.

Tableau 28.29. Vue `pg_statio_all_tables`

Type Description
`relid` `oid` OID d'une table
`schemaname` `name` Nom du schéma dans lequel se trouve cette table
`relname` `name` Nom de cette table
`heap_blks_read` `bigint` Nombre de blocs disque lus hors cache pour cette table
`heap_blks_hit` `bigint` Nombre de blocs disque lus dans le cache pour cette table
`idx_blks_read` `bigint` Nombre de blocs disque lus hors cache pour tous les index de cette table
`idx_blks_hit` `bigint` Nombre de tampons lus dans le cache pour tous les index de cette table
`toast_blks_read` `bigint` Nombre de blocs disque lus sur la partie TOAST de cette table (si présente)
`toast_blks_hit` `bigint` Nombre de tampons récupérés sur la partie TOAST de cette table (si présente)
`tidx_blks_read` `bigint` Nombre de blocs disque lus sur les index de la partie TOAST de cette table (si présente)
`tidx_blks_hit` `bigint` Nombre de tampons récupérés sur les index de la partie TOAST de cette table (si présente)

28.2.20. `pg_statio_all_indexes`

La vue `pg_statio_all_indexes` contiendra une ligne pour chaque index dans la base de données courante, montrant les statistiques d'entrées/sorties sur chaque index spécifiquement. Les vues `pg_statio_user_indexes` et `pg_statio_sys_indexes` contiennent la même information, mais sont filtrées pour ne montrer respectivement que les tables utilisateur et tables système.

Tableau 28.30. Vue `pg_statio_all_indexes`

Type Description
`relid` `oid`

Type Description
OID de la table pour cet index
`indexrelid` `oid` OID de cet index
`schemaname` `name` Nom du schéma dans lequel se trouve cet index
`relname` `name` Nom de la table pour cet index
`indexrelname` `name` Nom de cet index
`idx_blks_read` `bigint` Nombre de blocs disque lus pour cet index
`idx_blks_hit` `bigint` Nombre de tampons récupérés sur cet index

28.2.21. `pg_statio_all_sequences`

La vue `pg_statio_all_sequences` contiendra une ligne pour chaque séquence dans la base de données courante, montrant les statistiques d'entrées/sorties pour chaque séquence spécifiquement.

Tableau 28.31. Vue `pg_statio_all_sequences`

Type Description
`relid` `oid` OID de cette séquence
`schemaname` `name` Nom du schéma dans lequel se trouve cette séquence
`relname` `name` Nom de cette séquence
`blks_read` `bigint` Nombre de blocs disque lus pour cette séquence
`blks_hit` `bigint` Nombre de tampons récupérés pour cette séquence

28.2.22. `pg_stat_user_functions`

La vue `pg_stat_user_functions` contiendra une ligne pour chaque fonction suivie, montrant les statistiques d'exécution de cette fonction. Le paramètre track_functions contrôle exactement quelles fonctions sont suivies.

Tableau 28.32. Vue `pg_stat_user_functions`

Type Description
`funcid` `oid` OID de cette fonction
`schemaname` `name` Nom du schéma dans lequel se trouve cette fonction
`funcname` `name` Nom de cette fonction

Type Description
`calls` `bigint` Nombre de fois que cette fonction a été appelée
`total_time` `double precision` Temps total passé dans cette fonction ainsi que dans toutes les autres fonctions appelées par elle, en millisecondes
`self_time` `double precision` Temps total passé dans cette fonction seule, sans inclure les autres fonctions appelées par elle, en millisecondes

28.2.23. `pg_stat_slru`

PostgreSQL accède à certaines informations stockées sur disque via des caches *SLRU* (*simple least-recently-used*). La vue `pg_stat_slru` contiendra une ligne pour chaque cache SLRU tracé, affichant des statistiques sur l'accès aux pages cachées.

Tableau 28.33. Vue `pg_stat_slru`

Type Description
`name` `text` Nom du cache SLRU
`blks_zeroed` `bigint` Nombre de blocs initialisés à zéro
`blks_hit` `bigint` Nombre de fois que les blocs disques ont été trouvés dans le cache SLRU, et donc qu'une lecture disque n'était pas nécessaire (ceci inclut seulement les lectures dans le cache SLRU, pas les lectures dans le cache du système d'exploitation)
`blks_read` `bigint` Nombre de blocs disques lus à partir de ce cache SLRU
`blks_written` `bigint` Nombre de blocs disques écrit dans ce cache SLRU
`blks_exists` `bigint` Nombre de blocs dont l'existence a été vérifiée dans ce cache SLRU
`flushes` `bigint` Nombre de vidages de données modifiées pour ce cache SLRU
`truncates` `bigint` Nombre de troncatures pour ce cache SLRU
`stats_reset` `timestamp with time zone` Horodatage de la dernière réinitialisation de ces statistiques

28.2.24. Fonctions statistiques

Une autre façon de regarder les statistiques peut être mise en place en écrivant des requêtes utilisant les mêmes fonctions d'accès sous- jacentes utilisées par les vues standards montrées au-dessus. Pour des détails comme les noms de fonction, veuillez consulter les définitions de ces vues standards. (Par exemple, dans psql vous pouvez utiliser `\d+ pg_stat_activity`.) Les fonctions d'accès pour les statistiques par base de données prennent comme argument un OID pour identifier sur quelle base de données travailler. Les fonctions par table et par index utilisent un OID de table ou d'index. Les fonctions pour les statistiques par fonctions utilisent un OID de fonction. Notez que seuls les tables, index et fonctions dans la base de données courante peuvent être vus avec ces fonctions.

Les fonctions supplémentaires liées au système de statistiques cumulatives sont listées dans Tableau 28.34.

Tableau 28.34. Fonctions supplémentaires de statistiques

Fonction Description
`pg_backend_pid` () → `integer` Identifiant du processus serveur gérant la session courante.
`pg_stat_get_xact_blocks_fetched` (`oid`) → `bigint` Renvoie le nombre de demandes de lecture de bloc pour la table ou l'index dans la transaction en cours. Ce nombre soustrait à `pg_stat_get_xact_blocks_hit` donne le nombre d'appels à la fonction noyau `read()` ; le nombre de lectures physiques réelles est généralement plus basse grâce au cache au niveau noyau.
`pg_stat_get_xact_blocks_hit` (`oid`) → `bigint` Renvoie le nombre de demandes de lecture de bloc pour la table ou l'index dans la transaction en cours, trouvé dans le cache (donc ne déclenchant pas les appels à la fonction noyau `read()`).
`pg_stat_get_activity` (`integer`) → `setof record` Retourne un enregistrement d'informations sur le processus serveur du PID spécifié, ou un enregistrement pour chaque processus serveur actif dans le système si `NULL` est spécifié. Les champs retournés sont des sous-ensembles de ceux dans la vue `pg_stat_activity`.
`pg_stat_get_snapshot_timestamp` () → `timestamp with time zone` Returns the timestamp of the current statistics snapshot, or NULL if no statistics snapshot has been taken. A snapshot is taken the first time cumulative statistics are accessed in a transaction if `stats_fetch_consistency` is set to `snapshot`
`pg_stat_clear_snapshot` () → `void` Discards the current statistics snapshot or cached information.
`pg_stat_reset` () → `void` Réinitialise à 0 tous les compteurs statistiques pour la base de données actuelle. Cette fonction est restreinte aux superutilisateurs par défaut, mais d'autres utilisateurs peuvent avoir le droit EXECUTE pour exécuter la fonction.
`pg_stat_reset_shared` (`text`) → `void` Réinitialise à 0 les compteurs statistiques de l'instance, suivant l'argument. L'argument peut valoir `bgwriter` pour réinitialiser tous les compteurs affichés dans la vue `pg_stat_bgwriter`, `archiver` pour réinitialiser tous les compteurs affichés dans la vue `pg_stat_archiver`, `wal` to reset all the counters shown in the `pg_stat_wal` view or `recovery_prefetch` to reset all the counters shown in the `pg_stat_recovery_prefetch` view. Cette fonction est restreinte aux superutilisateurs par défaut, mais d'autres utilisateurs peuvent avoir le droit EXECUTE pour exécuter la fonction.
`pg_stat_reset_single_table_counters` (`oid`) → `void` Réinitialise à 0 les statistiques pour une table ou un index or shared across all databases in the cluster to zero. dans la base de données courante. Cette fonction est restreinte aux superutilisateurs par défaut, mais d'autres utilisateurs peuvent avoir le droit EXECUTE pour exécuter la fonction.
`pg_stat_reset_single_function_counters` (`oid`) → `void` Réinitialise à 0 les statistiques pour une fonction dans la base de données courante. Cette fonction est restreinte aux superutilisateurs par défaut, mais d'autres utilisateurs peuvent avoir le droit EXECUTE pour exécuter la fonction.

Fonction Description
`pg_stat_reset_slru ( text ) → void` Réinitialise à 0 les statistiques à un cache SLRU, ou pour toutes les SLRU de l'instance. Si l'argument est NULL, tous les compteurs affichés dans la vue `pg_stat_slru` sont réinitialisés pour tous les caches SLRU. L'argument fait partie de `CommitTs`, `MultiXactMember`, `MultiXactOffset`, `Notify`, `Serial`, `Subtrans` ou `Xact` pour réinitialiser les compteurs pour cette seule entrée. Si l'argument est `other` (ou tout nom non reconnu), alors les compteurs de tous les autres caches SLRU, comme ceux définis par des extensions, sont réinitialisés. Cette fonction est restreinte aux superutilisateurs par défaut, mais d'autres utilisateurs peuvent avoir le droit EXECUTE pour exécuter la fonction.
`pg_stat_reset_replication_slot ( text ) → void` Remet à zéro les statistiques du slot de réplication définies par l'argument. Si l'argument est `NULL`, réinitialise les statistiques pour tous les slots de réplication. Cette fonction est restreinte aux superutilisateurs par défaut, mais d'autres utilisateurs peuvent avoir le droit EXECUTE pour exécuter la fonction.
`pg_stat_reset_subscription_stats ( oid ) → void` Resets statistics for a single subscription shown in the `pg_stat_subscription_stats` view to zero. If the argument is `NULL`, reset statistics for all subscriptions. This function is restricted to superusers by default, but other users can be granted EXECUTE to run the function.

`pg_stat_get_activity`, la fonction sous-jacente de la vue `pg_stat_activity`, retourne un ensemble d'enregistrements contenant toute l'information disponible sur chaque processus serveur. Parfois il peut être plus pratique de n'obtenir qu'un sous-ensemble de cette information. Dans ces cas-là, un ensemble plus vieux de fonctions d'accès aux statistiques par processus serveur peut être utilisé ; celle-ci sont montrées dans Tableau 28.35. Ces fonctions d'accès utilisent un numéro d'identifiant du processus serveur, qui va de un au nombre de processus serveur actuellement actifs. La fonction `pg_stat_get_backend_idset` fournit une manière pratique de générer une ligne pour chaque processus serveur actif pour appeler ces fonctions. Par exemple, pour montrer les PID et requêtes en cours de tous les processus serveur :

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Tableau 28.35. Fonctions statistiques par processus serveur

Fonction Description
`pg_stat_get_backend_idset () → setof integer` Ensemble de numéros de processus serveur actuellement actifs (de 1 jusqu'au nombre de processus serveur actifs)
`pg_stat_get_backend_activity ( integer ) → text` Texte de la requête la plus récente de ce processus serveur
`pg_stat_get_backend_activity_start ( integer ) → timestamp with time zone` Heure à laquelle la requête la plus récente a été démarrée
`pg_stat_get_backend_client_addr ( integer ) → inet`

Fonction Description
Adresse IP du client connecté à ce processus serveur
`pg_stat_get_backend_client_port ( integer ) → integer` Numéro de port TCP que le client utilise pour communiquer
`pg_stat_get_backend_dbid ( integer ) → oid` OID de la base de données auquel ce processus serveur est connecté
`pg_stat_get_backend_pid ( integer ) → integer` Identifiant du processus serveur
`pg_stat_get_backend_start ( integer ) → timestamp with time zone` Heure à laquelle ce processus a été démarré
`pg_stat_get_backend_userid ( integer ) → oid` OID de l'utilisateur connecté à ce processus serveur
`pg_stat_get_backend_wait_event_type ( integer ) → text` Nom du type d'événement d'attente si le processus est actuellement en attente, NULL sinon. Voir Tableau 28.4 pour les détails.
`pg_stat_get_backend_wait_event ( integer ) → text` Nom de l'événement d'attente si le processus est actuellement en attente, NULL sinon. Voir Tableau 28.4 pour les détails.
`pg_stat_get_backend_xact_start ( integer ) → timestamp with time zone` Heure à laquelle la transaction courante a été démarrée

28.3. Visualiser les verrous

Un autre outil utile pour surveiller l'activité des bases de données est la table système `pg_locks`. Elle permet à l'administrateur système de visualiser des informations sur les verrous restant dans le gestionnaire des verrous. Par exemple, cette fonctionnalité peut être utilisée pour :

- Visualiser tous les verrous en cours, tous les verrous sur les relations d'une base de données particulière ou tous les verrous détenus par une session PostgreSQL particulière.
- Déterminer la relation de la base de données disposant de la plupart des verrous non autorisés (et qui, du coup, pourraient être une source de contention parmi les clients de la base de données).
- Déterminer l'effet de la contention des verrous sur les performances générales des bases de données, ainsi que l'échelle dans laquelle varie la contention sur le trafic de la base de données.

Les détails sur la vue `pg_locks` apparaissent dans la Section 54.12. Pour plus d'informations sur les verrous et la gestion des concurrences avec PostgreSQL, référez-vous au Chapitre 13.

28.4. Rapporter la progression

PostgreSQL a la possibilité de rapporter la progression de certaines commandes lors de leur exécution. Actuellement, les seules commandes supportant un rapport de progression sont `ANALYZE`, `CLUSTER`, `CREATE INDEX`, `VACUUM`, `COPY`, et base_backup (i.e., commande de réplication que pg_basebackup exécute pour réaliser une sauvegarde de base). Ceci pourrait être étendu dans le futur.

28.4.1. Rapporter la progression d'ANALYZE

Quand ANALYZE est en cours d'exécution, la vue pg_stat_progress_analyze contiendra une ligne pour chaque processus qui exécute actuellement cette commande. Les tables ci-dessous décrivent les informations rapportées et fournissent des informations sur leur interprétation.

Tableau 28.36. Vue pg_stat_progress_analyze

Type Description
pid integer Identifiant du processus serveur.
datid oid OID de la base de données où ce processus est connecté.
datname name Nom de la base de données où ce processus est connecté.
relid oid OID de la table en cours de traitement par ANALYZE.
phase text Phase de traitement courante. Voir Tableau 28.37.
sample_blks_total bigint Nombre total de blocs de la table à traiter.
sample_blks_scanned bigint Nombre de blocs parcourus.
ext_stats_total bigint Nombre de statistiques étendues.
ext_stats_computed bigint Nombre de statistiques étendues calculées. Ce compteur avance seulement quand la phase est computing extended statistics.
child_tables_total bigint Nombre de tables enfants.
child_tables_done bigint Nombre de tables enfants parcourues. Ce compteur avance seulement quand la phase est acquiring inherited sample rows.
current_child_table_relid oid OID de la table enfant en cours de traitement. Ce champ est seulement valide quand la phase est acquiring inherited sample rows.

Tableau 28.37. Phases ANALYZE

Phase	Description
initializing	La commande prépare le début du traitement de la table. Cette phase est normalement très brève.
acquiring sample rows	La command est en cours de parcours de la table indiquée par relid pour obtenir les lignes de l'échantillon.
acquiring inherited sample rows	La command est en cours de parcours des tables enfants pour obtenir les lignes de l'échantillon. Les colonnes child_tables_total, child_tables_done et current_child_table_relid contiennent les informations de progression pour cette phase.
computing statistics	La command calcule les statistiques à partir des lignes de l'échantillon obtenu lors du parcours de la table.
computing extended statistics	La commande calcule les statistiques étendues à partir des lignes de l'échantillon obtenu lors du parcours de la table.

Phase	Description
`finalizing analyze`	La commande met à jour `pg_class`. Quand cette phase est terminée, `ANALYZE` terminera.

> **Note**
>
> Notez que quand `ANALYZE` est exécutée sur une table partitionnée,toutes ces partitions sont traitées récursivement. Dans ce cas, la progression d'`ANALYZE` est rapportée tout d'abord pour la table parent alors que les statistiques d'héritage sont récupérées, suivies par celles de chaque partition.

28.4.2. Rapporter la progression du CREATE INDEX

Quand un `CREATE INDEX` ou un `REINDEX` est en cours d'exécution, la vue `pg_stat_progress_create_index` contient une ligne pour chaque processus serveur en train de créer des index. Les tables ci-dessous décrivent les informations rapportées et fournissent des informations sur leur interprétation.

Tableau 28.38. Vue `pg_stat_progress_create_index`

Type Description
`pid` `integer` Identifiant du processus serveur.
`datid` `oid` OID de la base de données de connexion du processus.
`datname` `name` Nom de la base de données de connexion du processus.
`relid` `oid` OID de la table liée à l'index en cours de création.
`index_relid` `oid` OID de l'index en cours de création ou de réindexation. Lors d'un `CREATE INDEX` non concurrent, cette colonne vaut 0.
`command` `text` La commande en cours d'exécution : `CREATE INDEX`, `CREATE INDEX CONCURRENTLY`, `REINDEX` ou `REINDEX CONCURRENTLY`.
`phase` `text` Phase en cours de traitement pour la création de l'index. Voir Tableau 28.39.
`lockers_total` `bigint` Nombre total de processus bloquant à attendre, si applicable.
`lockers_done` `bigint` Nombre de processus bloquant déjà attendus.
`current_locker_pid` `bigint` Identifiant du processus bloquant actuellement.
`blocks_total` `bigint` Nombre total de blocs à traiter dans la phase en cours.
`blocks_done` `bigint` Nombre de blocs déjà traités dans la phase en cours.
`tuples_total` `bigint`

Type Description
Nombre total de lignes à traiter dans la phase en cours.
`tuples_done` `bigint` Nombre de lignes déjà traitées dans la phase en cours.
`partitions_total` `bigint` Lors de la création d'un index sur une table partitionnée, cette colonne est configurée au nombre total de partitions sur lesquels l'index est créé. Ce champs vaut `0` lors d'un `REINDEX`.
`partitions_done` `bigint` Lors de la création d'un index sur une table partitionnée, cette colonne est configurée au nombre total de partitions sur lesquels l'index a déjà été créé.

Tableau 28.39. Phases du CREATE INDEX

Phase	Description
`initializing`	`CREATE INDEX` ou `REINDEX` prépare la création de l'index. Cette phase est normalement très brève.
`waiting for writers before build`	`CREATE INDEX CONCURRENTLY` ou `REINDEX CONCURRENTLY` attend que les transactions avec des verrous en écriture qui peuvent voir la table se finissent Cette phase est ignorée quand elle n'est pas en mode concurrent. Les colonnes `lockers_total`, `lockers_done` et `current_locker_pid` contiennent les informations de progression sur cette phase.
`building index`	L'index est en cours de construction par le code spécifique de la méthode d'accès. Dans cette phase, les méthodes d'accès qui supportent les rapports de progression remplissent eux-même les données de progression, et la sous-phase est indiquée dans cette colonne. Typiquement, `blocks_total` et `blocks_done` contiendront les données de progression, ainsi que `tuples_total` et `tuples_done` potentiellement.
`waiting for writers before validation`	`CREATE INDEX CONCURRENTLY` ou `REINDEX CONCURRENTLY` est en attente de la fin des transactions avec verrous en écriture pouvant potentiellement écrire dans la table. Cette phase est ignorée quand elle n'est pas en mode concurrent. Les colonnes `lockers_total`, `lockers_done` et `current_locker_pid` contiennent les informations de progression pour cette phase.
`index validation: scanning index`	`CREATE INDEX CONCURRENTLY` parcourt l'index pour trouver les lignes qui ont besoin d'être validées. Cette phase est ignorée quand elle n'est pas en mode concurrent. Les colonnes `blocks_total` (configurée à la taille totale de l'index) et `blocks_done` contiennent les informations de progression pour cette phase.

Phase	Description
`index validation: sorting tuples`	`CREATE INDEX CONCURRENTLY` trie la sortie de la phase de parcours de l'index.
`index validation: scanning table`	`CREATE INDEX CONCURRENTLY` parcourt la table pour valider les enregistrements d'index collectés dans les deux phases précédentes. Cette phase est ignorée quand elle n'est pas en mode concurrent. Les colonnes `blocks_total` (configurée à la taille totale de la table) et `blocks_done` contiennent les informations de progression pour cette phase.
`waiting for old snapshots`	`CREATE INDEX CONCURRENTLY` ou `REINDEX CONCURRENTLY` est en attente que les transactions pouvant potentiellement voir la table relâchent leur snapshot. Cette phase est ignorée quand elle n'est pas en mode concurrent. Les colonnes `lockers_total`, `lockers_done` et `current_locker_pid` contiennent les informations de progression pour cette phase.
`waiting for readers before marking dead`	`REINDEX CONCURRENTLY` est en attente de la fin des transactions avec verrous en lecture sur la table, avant de marquer l'ancien index comme mort. Cette phase est ignorée quand elle n'est pas en mode concurrent. Les colonnes `lockers_total`, `lockers_done` et `current_locker_pid` contiennent les informations de progression pour cette phase.
`waiting for readers before dropping`	`REINDEX CONCURRENTLY` est en attente de la fin des transactions avec verrous en lecture sur la table, avant de supprimer l'ancien index. Cette phase est ignorée quand elle n'est pas en mode concurrent. Les colonnes `lockers_total`, `lockers_done` et `current_locker_pid` contiennent les informations de progression pour cette phase.

28.4.3. Rapporter la progression du VACUUM

La vue `pg_stat_progress_vacuum` contient une ligne pour chaque processus serveur (incluant les processus autovacuum worker) en train d'exécuter un `VACUUM`. Les tableaux ci-dessous décrivent les informations rapportées et fournissent des informations sur leur interprétation. La progression des commandes `VACUUM FULL` est rapportée par `pg_stat_progress_cluster` parce que le `VACUUM FULL` comme le `CLUSTER` réécrivent la table, alors qu'un `VACUUM` simple la modifie directement. Voir Section 28.4.4.

Tableau 28.40. Vue `pg_stat_progress_vacuum`

Type Description
`pid` `integer` Identifiant (PID) du processus serveur.
`datid` `oid` OID de la base de données où est connecté ce processus serveur.
`datname` `name`

Type Description
Nom de la base de données où est connecté ce processus serveur.
`relid` `oid` OID de la table nettoyée par le VACUUM.
`phase` `text` Phase actuelle du vacuum. Voir Tableau 28.41.
`heap_blks_total` `bigint` Nombre total de blocs de la table. Ce nombre est récupéré au début du parcours. Des blocs peuvent être ajoutés par la suie, mais ne seront pas (et n'ont pas besoin d'être) visités par ce VACUUM.
`heap_blks_scanned` `bigint` Nombre de blocs parcourus dans la table. Comme la carte de visibilité est utilisée pour optimiser les parcours, certains blocs seront ignorés sans inspection ; les blocs ignorés sont inclus dans ce total, pour que ce nombre puisse devenir égal à `heap_blks_total` quand le nettoyage se termine. Ce compteur avance seulement quand la phase est `scanning heap`.
`heap_blks_vacuumed` `bigint` Nombre de blocs nettoyés dans la table. Sauf si la table n'a pas d'index, ce compteur avance seulement quand la phase est `vacuuming heap`. Les blocs qui ne contiennent aucune ligne morte sont ignorés, donc le compteur pourrait parfois avancer par de larges incréments.
`index_vacuum_count` `bigint` Nombre de cycles de nettoyage d'index réalisés.
`max_dead_tuples` `bigint` Nombre de lignes mortes que nous pouvons stocker avant d'avoir besoin de réaliser un cycle de nettoyage d'index, basé sur maintenance_work_mem.
`num_dead_tuples` `bigint` Nombre de lignes mortes récupérées depuis le dernier cycle de nettoyage d'index.

Tableau 28.41. Phases du VACUUM

Phase	Description
`initializing`	VACUUM se prépare à commencer le parcours de la table. Cette phase est habituellement très rapide.
`scanning heap`	VACUUM parcourt la table. Il va défragmenter chaque bloc si nécessaire et potentiellement réaliser un gel des lignes. La colonne `heap_blks_scanned` peut être utilisée pour surveiller la progression du parcours.
`vacuuming indexes`	VACUUM est en train de nettoyer les index. Si une table a des index, ceci surviendra au moins une fois par vacuum, après le parcours complet de la table. Cela pourrait arriver plusieurs fois par vacuum si maintenance_work_mem (ou, dans le cas de l'autovacuum, autovacuum_work_mem s'il est configuré) n'est pas suffisamment important pour y enregistrer le nombre de lignes mortes trouvées.
`vacuuming heap`	VACUUM est en train de nettoyer la table. Nettoyer la table est différent du parcours de

Phase	Description
	la table, et survient après chaque phase de nettoyage d'index. Si `heap_blks_scanned` est inférieur à `heap_blks_total`, le système retournera à parcourir la table après la fin de cette phase. Sinon, il commencera le nettoyage des index une fois cette phase terminée.
`cleaning up indexes`	`VACUUM` est en train de nettoyer les index. Ceci survient après que la table ait été entièrement parcourue et que le vacuum des index et de la table soit terminé.
`truncating heap`	`VACUUM` est en cours de troncage de la table pour pouvoir redonner au système d'exploitation les pages vides en fin de relation. Ceci survient après le nettoyage des index.
`performing final cleanup`	`VACUUM` réalise le nettoyage final. Durant cette phase, `VACUUM` nettoiera la carte des espaces libres, mettra à jour les statistiques dans `pg_class`, et rapportera les statistiques au système de statistiques cumulatives. Une fois cette phase terminée, `VACUUM` se terminera.

28.4.4. Rapporter la progression de CLUSTER

Quand `CLUSTER` ou `VACUUM FULL` est en cours d'exécution, la vue `pg_stat_progress_cluster` contiendra une ligne pour chaque processus serveur en train d'exécuter une de ces deux commandes. Les tables ci-dessous décrivent les informations rapportées et fournissent des informations sur leur interprétation.

Tableau 28.42. Vue `pg_stat_progress_cluster`

Type Description
`pid` `integer` Identifiant du processus serveur.
`datid` `oid` OID de la base de données de connexion du processus.
`datname` `name` Nom de la base de données de connexion du processus.
`relid` `oid` OID de la table en cours de traitement.
`command` `text` La commande exécutée. Soit `CLUSTER` soit `VACUUM FULL`.
`phase` `text` Phase de traitement actuelle. Voir Tableau 28.43.
`cluster_index_relid` `oid` Si la table est parcourue avec un index, OID de l'index utilisé. Sinon 0.
`heap_tuples_scanned` `bigint` Nombre de lignes parcourues. Ce compteur s'incrémente seulement quand la phase est `seq scanning heap`, `index scanning heap` ou `writing new heap`.
`heap_tuples_written` `bigint` Nombre de lignes écrites. Ce compteur s'incrémente seulement quand la phase est `seq scanning heap`, `index scanning heap` ou `writing new heap`.

Type Description
heap_blks_total bigint Nombre total de blocs dans la table. Ce nombre est rapporté au début de seq scanning heap.
heap_blks_scanned bigint Nombre de blocs parcourus. Ce compteur s'incrément seulement quand la phase est seq scanning heap.
index_rebuild_count bigint Nombre d'index reconstruit. Ce compteur s'incrémente seulement quand la phase est rebuilding index.

Tableau 28.43. Phases de CLUSTER et VACUUM FULL

Phase	Description
initializing	La commande prépare le début du parcours de la table. Cette phase est normalement très brève.
seq scanning heap	La commande parcourt la table en utilisant un parcours séquentiel.
index scanning heap	CLUSTER parcourt la table en utilisant un parcours d'index.
sorting tuples	CLUSTER trie les lignes.
writing new heap	CLUSTER est en cours d'écriture du nouveau fichier de la table.
swapping relation files	La commande bascule les fichiers nouvellement construit en place.
rebuilding index	La commande reconstruit les index.
performing final cleanup	La commande réalise le nettoyage final. Quand cette phase est terminée, CLUSTER ou VACUUM FULL terminera.

28.4.5. Rapporter la progression de la sauvegarde de base

Quand une application comme pg_basebackup réalise une sauvegarde de base, la vue pg_stat_progress_basebackup contiendra une ligne pour chaque processus walsender en cours d'exécution de la commande de réplication BASE_BACKUP et d'envoi de la sauvegarde. Les tables ci-dessous décrivent les informations qui seront rapportées et fournissent des informations sur leur interprétation.

Tableau 28.44. Vue pg_stat_progress_basebackup

Type Description
pid integer Identifiant du processus walsender.
phase text Phase en cours de traitement. Voir Tableau 28.45.
backup_total bigint Quantité totale de données à envoyer. Ceci est estimé et rapporté au début de la phase streaming database files. Notez que c'est uniquement une approximation car

Type Description
la base de données pourrait changer pendant la phase `streaming database files` et les journaux de transactions pourraient être inclus dans la sauvegarde après coup. C'est toujours la même valeur que `backup_streamed` une fois la quantité de données envoyées dépasse la taille totale estimée. Si l'estimation est désactivée dans total size. Si l'estimation est désactivée dans pg_basebackup (grâce à l'option `--no-estimate-size`), cette colonne vaut `NULL`.
`backup_streamed` `bigint` Quantité de données envoyées. Ce compteur avance seulement quand la phase est `streaming database files` ou `transferring wal files`.
`tablespaces_total` `bigint` Nombre total de tablespaces à envoyer.
`tablespaces_streamed` `bigint` Nombre de tablespaces envoyées. Ce compteur avance seulement quand la phase est `streaming database files`.

Tableau 28.45. Phases de la sauvegarde de base

Phase	Description
`initializing`	Le processus walsender prépare le début de la sauvegarde. Cette phase est normalement très brève.
`waiting for checkpoint to finish`	Le processus walsender est en cours d'exécution de `pg_backup_start` pour initialiser une sauvegarde de base, et en attente que le checkpoint de début de sauvegarde se finisse.
`estimating backup size`	Le processus walsender estime actuellement la quantité totale des fichiers de la base à envoyer comme une sauvegarde de base.
`streaming database files`	Le processus walsender est en cours d'envoi les fichiers de la base comme sauvegarde de base.
`waiting for wal archiving to finish`	Le processus walsender exécute actuellement le `pg_backup_stop` pour terminer la sauvegarde, et est en attente de tous les journaux de transactions requis pour que la sauvegarde de base soit correctement archivée. Si l'option `--wal-method=none` ou l'option `--wal-method=stream` est spécifié avec pg_basebackup, la sauvegarde se terminera une fois cette phase terminée.
`transferring wal files`	Le processus walsender est en cours de transfert des journaux de transactions générés lors de la sauvegarde. Cette phase survient après la phase `waiting for wal archiving to finish` si l'option `--wal-method=fetch` est indiquée à pg_basebackup. La sauvegarde se terminera quand cette phase sera terminée.

28.4.6. COPY Progress Reporting

Lorsque `COPY` est en cours d'exécution, la vue `pg_stat_progress_copy` contient alors une ligne pour chaque processus serveur qui exécute à l'instant une commande `COPY`. La table ci-dessous décrit les informations qui sont affichées et fournit des informations sur comment les interpréter.

Tableau 28.46. Vue `pg_stat_progress_copy`

Type de la colonne Description
`pid` `integer`

Type de la colonne Description
ID du processus serveur.
`datid` `oid` OID de la base de données sur laquelle le processus serveur est connecté.
`datname` `name` Nom de la base de données sur laquelle le processus serveur est connecté.
`relid` `oid` OID de la table sur laquelle la commande `COPY` s'exécute. Cette valeur vaut `0` si la copie se fait depuis une requête `SELECT`.
`command` `text` La commande qui est exécutée : `COPY FROM`, ou `COPY TO`.
`type` `text` Le type d'entrée/sortie à partir duquel les données sont lues ou écrites : `FILE`, `PROGRAM`, `PIPE` (pour `COPY FROM STDIN` et `COPY TO STDOUT`), ou `CALLBACK` (utilisée par exemple durant la synchronisation initiale de table dans la réplication logique).
`bytes_processed` `bigint` Nombre d'octets déjà traités par la commande `COPY`.
`bytes_total` `bigint` Taille du fichier source pour la commande `COPY FROM` en octets. La valeur vaut `0` si cette information n'est pas disponible.
`tuples_processed` `bigint` Nombre de lignes déjà traitées par la commande `COPY`.
`tuples_excluded` `bigint` Nombre de lignes non traitées car exclues par la clause `WHERE` de la commande `COPY`.

28.5. Traces dynamiques

PostgreSQL fournit un support pour les traces dynamiques du serveur de bases de données. Ceci permet l'appel à un outil externe à certains points du code pour tracer son exécution.

Un certain nombre de sondes et de points de traçage sont déjà insérés dans le code source. Ces sondes ont pour but d'être utilisées par des développeurs et des administrateurs de base de données. Par défaut, les sondes ne sont pas compilées dans PostgreSQL ; l'utilisateur a besoin de préciser explicitement au script configure de rendre disponible les sondes.

Actuellement, l'outil DTrace[1] est supporté. Il est disponible sur Solaris, macOS, FreeBSD, NetBSD et Oracle Linux. Le projet SystemTap[2] fournit un équivalent DTrace et peut aussi être utilisé. Le support d'autres outils de traces dynamiques est possible théoriquement en modifiant les définitions des macros dans `src/include/utils/probes.h`.

28.5.1. Compiler en activant les traces dynamiques

Par défaut, les sondes ne sont pas disponibles, donc vous aurez besoin d'indiquer explicitement au script configure de les activer dans PostgreSQL. Pour inclure le support de DTrace, ajoutez `--enable-dtrace` aux options de configure. Lire Section 17.4 pour plus d'informations.

[1] https://en.wikipedia.org/wiki/DTrace
[2] https://sourceware.org/systemtap/

28.5.2. Sondes disponibles

Un certain nombre de sondes standards sont fournies dans le code source, comme indiqué dans Tableau 28.47. Tableau 28.48 précise les types utilisés dans les sondes. D'autres peuvent être ajoutées pour améliorer la surveillance de PostgreSQL.

Tableau 28.47. Sondes disponibles pour DTrace

Nom	Paramètres	Aperçu
`transaction-start`	`(LocalTransactionId)`	Sonde qui se déclenche au lancement d'une nouvelle transaction. arg0 est l'identifiant de transaction
`transaction-commit`	`(LocalTransactionId)`	Sonde qui se déclenche quand une transaction se termine avec succès. arg0 est l'identifiant de transaction
`transaction-abort`	`(LocalTransactionId)`	Sonde qui se déclenche quand une transaction échoue. arg0 est l'identifiant de transaction
`query-start`	`(const char *)`	Sonde qui se déclenche lorsque le traitement d'une requête commence. arg0 est la requête
`query-done`	`(const char *)`	Sonde qui se déclenche lorsque le traitement d'une requête se termine. arg0 est la requête
`query-parse-start`	`(const char *)`	Sonde qui se déclenche lorsque l'analyse d'une requête commence. arg0 est la requête
`query-parse-done`	`(const char *)`	Sonde qui se déclenche lorsque l'analyse d'une requête se termine. arg0 est la requête
`query-rewrite-start`	`(const char *)`	Sonde qui se déclenche lorsque la ré-écriture d'une requête commence. arg0 est la requête
`query-rewrite-done`	`(const char *)`	Sonde qui se déclenche lorsque la ré-écriture d'une requête se termine. arg0 est la requête
`query-plan-start`	`()`	Sonde qui se déclenche lorsque la planification d'une requête commence
`query-plan-done`	`()`	Sonde qui se déclenche lorsque la planification d'une requête se termine
`query-execute-start`	`()`	Sonde qui se déclenche lorsque l'exécution d'une requête commence
`query-execute-done`	`()`	Sonde qui se déclenche lorsque l'exécution d'une requête se termine
`statement-status`	`(const char *)`	Sonde qui se déclenche à chaque fois que le

Nom	Paramètres	Aperçu
		processus serveur met à jour son statut dans `pg_stat_activity.status`. arg0 est la nouvelle chaîne de statut
`checkpoint-start`	`(int)`	Sonde qui se déclenche quand un point de retournement commence son exécution. arg0 détient les drapeaux bit à bit utilisés pour distinguer les différents types de points de retournement, comme un point suite à un arrêt, un point immédiat ou un point forcé
`checkpoint-done`	`(int, int, int, int, int)`	Sonde qui se déclenche quand un point de retournement a terminé son exécution (les sondes listées après se déclenchent en séquence lors du traitement d'un point de retournement). arg0 est le nombre de tampons mémoires écrits. arg1 est le nombre total de tampons mémoires. arg2, arg3 et arg4 contiennent respectivement le nombre de journaux de transactions ajoutés, supprimés et recyclés
`clog-checkpoint-start`	`(bool)`	Sonde qui se déclenche quand la portion CLOG d'un point de retournement commence. arg0 est true pour un point de retournement normal, false pour un point de retournement suite à un arrêt
`clog-checkpoint-done`	`(bool)`	Sonde qui se déclenche quand la portion CLOG d'un point de retournement commence. arg0 a la même signification que pour `clog-checkpoint-start`
`subtrans-checkpoint-start`	`(bool)`	Sonde qui se déclenche quand la portion SUBTRANS d'un point de retournement commence. arg0 est true pour un point de retournement normal, false pour un point de retournement suite à un arrêt
`subtrans-checkpoint-done`	`(bool)`	Sonde qui se déclenche quand la portion SUBTRANS d'un point de retournement se termine. arg0 a la même signification que pour `subtrans-checkpoint-start`

Nom	Paramètres	Aperçu
`multixact-checkpoint-start`	`(bool)`	Sonde qui se déclenche quand la portion MultiXact d'un point de retournement commence. arg0 est true pour un point de retournement normal, false pour un point de retournement suite à un arrêt
`multixact-checkpoint-done`	`(bool)`	Sonde qui se déclenche quand la portion MultiXact d'un point de retournement se termine. arg0 a la même signification que pour `multixact-checkpoint-start`
`buffer-checkpoint-start`	`(int)`	Sonde qui se déclenche quand la portion d'écriture de tampons d'un point de retournement commence. arg0 contient les drapeaux bit à bit pour distinguer différents types de point de retournement comme le point après arrêt, un point immédiat, un point forcé
`buffer-sync-start`	`(int, int)`	Sonde qui se déclenche quand nous commençons d'écrire les tampons modifiés pendant un point de retournement (après identification des tampons qui doivent être écrits). arg0 est le nombre total de tampons. arg1 est le nombre de tampons qui sont modifiés et n'ont pas besoin d'être écrits
`buffer-sync-written`	`(int)`	Sonde qui se déclenche après chaque écriture d'un tampon lors d'un point de retournement. arg0 est le numéro d'identifiant du tampon
`buffer-sync-done`	`(int, int, int)`	Sonde qui se déclenche quand tous les tampons modifiés ont été écrits. arg0 est le nombre total de tampons. arg1 est le nombre de tampons réellement écrits par le processus de point de retournement. arg2 est le nombre attendu de tampons à écrire (arg1 de `buffer-sync-start`); toute différence reflète d'autres processus écrivant des tampons lors du point de retournement
`buffer-checkpoint-sync-start`	`()`	Sonde qui se déclenche une fois les tampons modifiés écrits par le noyau et avant de commencer à lancer des requêtes fsync

Nom	Paramètres	Aperçu
`buffer-checkpoint-done`	`()`	Sonde qui se déclenche après la fin de la synchronisation des tampons sur le disque
`twophase-checkpoint-start`	`()`	Sonde qui se déclenche quand la portion deux-phases d'un point de retournement est commencée
`twophase-checkpoint-done`	`()`	Sonde qui se déclenche quand la portion deux-phases d'un point de retournement est terminée
`buffer-read-start`	`(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)`	Sonde qui se déclenche quand la lecture d'un tampon commence. arg0 et arg1 contiennent les numéros de fork et de bloc de la page (arg1 vaudra -1 s'il s'agit de demande d'extension de la relation). arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation. arg5 est l'identifiant du processus moteur qui a créé la relation temporaire pour un tampon local ou `InvalidBackendId` (-1) pour un tampon partagé. arg6 est true pour une demande d'extension de la relation, false pour une lecture ordinaire
`buffer-read-done`	`(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool, bool)`	Sonde qui se déclenche quand la lecture d'un tampon se termine. arg0 et arg1 contiennent les numéros de fork et de bloc de la page (arg1 contient le numéro de bloc du nouveau bloc ajouté s'il s'agit de demande d'extension de la relation). arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation. arg5 est l'identifiant du processus moteur qui a créé la relation temporaire pour un tampon local ou `InvalidBackendId` (-1) pour un tampon partagé. arg6 est true pour une demande d'extension de la relation, false pour une lecture ordinaire. arg7 est true si la tampon était

Nom	Paramètres	Aperçu
		disponible en mémoire, false sinon
`buffer-flush-start`	`(ForkNumber, BlockNumber, Oid, Oid, Oid)`	Sonde qui se déclenche avant de lancer une demande d'écriture pour un bloc partagé. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation
`buffer-flush-done`	`(ForkNumber, BlockNumber, Oid, Oid, Oid)`	Sonde qui se déclenche quand une demande d'écriture se termine. (Notez que ceci ne reflète que le temps passé pour fournir la donnée au noyau ; ce n'est habituellement pas encore écrit sur le disque.) Les arguments sont identiques à ceux de `buffer-flush-start`
`buffer-write-dirty-start`	`(ForkNumber, BlockNumber, Oid, Oid, Oid)`	Sonde qui se déclenche quand un processus serveur commence à écrire un tampon modifié sur disque. Si cela arrive souvent, cela implique que shared_buffers est trop petit ou que les paramètres de contrôle de bgwriter ont besoin d'un ajustement.) arg0 et arg1 contiennent les numéros de fork et de bloc de la page. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation
`buffer-write-dirty-done`	`(ForkNumber, BlockNumber, Oid, Oid, Oid)`	Sonde qui se déclenche quand l'écriture d'un tampon modifié est terminé. Les arguments sont identiques à ceux de `buffer-write-dirty-start`
`wal-buffer-write-dirty-start`	`()`	Sonde qui se déclenche quand un processus serveur commence à écrire un tampon modifié d'un journal de transactions parce qu'il n'y a plus d'espace disponible dans le cache des journaux de transactions. (Si cela arrive souvent, cela implique que wal_buffers est trop petit.)

Nom	Paramètres	Aperçu
`wal-buffer-write-dirty-done`	`()`	Sonde qui se déclenche quand l'écriture d'un tampon modifié d'un journal de transactions est terminée
`wal-insert`	`(unsigned char, unsigned char)`	Sonde qui se déclenche quand un enregistrement est inséré dans un journal de transactions. arg0 est le gestionnaire de ressource (rmid) pour l'enregistrement. arg1 contient des informations supplémentaires
`wal-switch`	`()`	Sonde qui se déclenche quand une bascule du journal de transactions est demandée
`smgr-md-read-start`	`(ForkNumber, BlockNumber, Oid, Oid, Oid, int)`	Sonde qui se déclenche au début de la lecture d'un bloc d'une relation. arg0 et arg1 contiennent les numéros de fork et de bloc de la page. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation. arg5 est l'identifiant du processus moteur qui a créé la relation temporaire pour un tampon local ou `InvalidBackendId` (-1) pour un tampon partagé
`smgr-md-read-done`	`(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)`	Sonde qui se déclenche à la fin de la lecture d'un bloc. arg0 et arg1 contiennent les numéros de fork et de bloc de la page. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation. arg5 est l'identifiant du processus moteur qui a créé la relation temporaire pour un tampon local ou `InvalidBackendId` (-1) pour un tampon partagé. arg6 est le nombre d'octets réellement lus alors que arg7 est le nombre d'octets demandés (s'il y a une différence, il y a un problème)
`smgr-md-write-start`	`(ForkNumber, BlockNumber, Oid, Oid, Oid, int)`	Sonde qui se déclenche au début de l'écriture d'un bloc dans une relation. arg0 et arg1

Nom	Paramètres	Aperçu
		contiennent les numéros de fork et de bloc de la page. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation. arg5 est l'identifiant du processus moteur qui a créé la relation temporaire pour un tampon local ou `InvalidBackendId` (-1) pour un tampon partagé
`smgr-md-write-done`	`(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)`	Sonde qui se déclenche à la fin de l'écriture d'un bloc. arg0 et arg1 contiennent les numéros de fork et de bloc de la page. arg2, arg3 et arg4 contiennent respectivement l'OID du tablespace, de la base de données et de la relation identifiant ainsi précisément la relation. arg5 est l'identifiant du processus moteur qui a créé la relation temporaire pour un tampon local ou `InvalidBackendId` (-1) pour un tampon partagé. arg6 est le nombre d'octets réellement écrits alors que arg7 est le nombre d'octets demandés (si ces nombres sont différents, cela indique un problème)
`sort-start`	`(int, bool, int, int, bool, int)`	Sonde qui se déclenche quand une opération de tri est démarré. arg0 indique un tri de la table, de l'index ou d'un datum. arg1 est true si on force les valeurs uniques. arg2 est le nombre de colonnes clés. arg3 est le nombre de ko de mémoire autorisé pour ce travail. arg4 est true si un accès aléatoire au résultat du tri est requis arg5 indique serial si `0`, parallel worker si `1`, ou parallel leader si `2`.
`sort-done`	`(bool, long)`	Sonde qui se déclenche quand un tri est terminé. arg0 est true pour un tri externe, false pour un tri interne. arg1 est le nombre de blocs disque utilisés pour un tri externe, ou le nombre de ko de mémoire utilisés pour un tri interne

Nom	Paramètres	Aperçu
lwlock-acquire	(char *, LWLockMode)	Sonde qui se déclenche quand un LWLock a été acquis. arg0 est la tranche de LWLock. arg1 est le mode de verrou attendu, soit exclusif soit partagé.
lwlock-release	(char *)	Sonde qui se déclenche quand un LWLock a été relâché (mais notez que tout processus en attente n'a pas encore été réveillé). arg0 est la tranche de LWLock.
lwlock-wait-start	(char *, LWLockMode)	Sonde qui se déclenche quand un LWLock n'était pas immédiatement disponible et qu'un processus serveur a commencé à attendre la disponibilité du verrou. arg0 est la tranche de LWLock. arg1 est le mode de verrou attendu, soit exclusif soit partagé.
lwlock-wait-done	(char *, LWLockMode)	Sonde qui se déclenche quand un processus serveur n'est plus en attente d'un LWLock (il n'a pas encore le verrou). arg0 est la tranche de LWLock. arg1 est le mode de verrou attendu, soit exclusif soit partagé.
lwlock-condacquire	(char *, LWLockMode)	Sonde qui se déclenche quand un LWLock a été acquis avec succès malgré le fait que l'appelant ait demandé de ne pas attendre. arg0 est la tranche de LWLock. arg1 est le mode de verrou attendu, soit exclusif soit partagé.
lwlock-condacquire-fail	(char *, LWLockMode)	Sonde qui se déclenche quand un LWLock, demandé sans attente, n'est pas accepté. arg0 est la tranche de LWLock. arg1 est le mode de verrou attendu, soit exclusif soit partagé.
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	Sonde qui se déclenche quand une demande d'un gros verrou (*lmgr lock*) a commencé l'attente parce que le verrou n'était pas disponible. arg0 à arg3 sont les champs identifiant l'objet en cours de verrouillage. arg4 indique le type d'objet à verrouiller. arg5 indique le type du verrou demandé
lock-wait-done	(unsigned int, unsigned int,	Sonde qui se déclenche quand une demande d'un gros verrou

Nom	Paramètres	Aperçu
	`unsigned int, unsigned int, unsigned int, LOCKMODE)`	(*lmgr lock*) a fini d'attendre (c'est-à-dire que le verrou a été accepté). Les arguments sont identiques à ceux de lock-wait-start
`deadlock-found`	`()`	Sonde qui se déclenche quand un verrou mortel est trouvé par le détecteur

Tableau 28.48. Types définis utilisés comme paramètres de sonde

Type	Definition
`LocalTransactionId`	`unsigned int`
`LWLockMode`	`int`
`LOCKMODE`	`int`
`BlockNumber`	`unsigned int`
`Oid`	`unsigned int`
`ForkNumber`	`int`
`bool`	`unsigned char`

28.5.3. Utiliser les sondes

L'exemple ci-dessous montre un script DTrace pour l'analyse du nombre de transactions sur le système, comme alternative à l'interrogation régulière de `pg_stat_database` avant et après un test de performance :

```
#!/usr/sbin/dtrace -qs

 postgresql$1:::transaction-start
 {
       @start["Start"] = count();
       self->ts  = timestamp;
 }

 postgresql$1:::transaction-abort
 {
       @abort["Abort"] = count();
 }

 postgresql$1:::transaction-commit
 /self->ts/
 {
       @commit["Commit"] = count();
       @time["Total time (ns)"] = sum(timestamp - self->ts);
       self->ts=0;
 }
```

À son exécution, le script de l'exemple D donne une sortie comme :

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
 ^C
```

```
Start                                    71
Commit                                   70
Total time (ns)                  2312105013
```

> **Note**
>
> SystemTap utilise une notation différente de DTrace pour les scripts de trace, même si les points de trace sont compatibles. Il est intéressant de noter que, lorsque nous avons écrit ce texte, les scripts SystemTap doivent référencer les noms des sondes en utilisant des tirets bas doubles à la place des tirets simples. Il est prévu que les prochaines versions de SystemTap corrigent ce problème.

Vous devez vous rappeler que les programmes DTrace doivent être écrits soigneusement, sinon les informations récoltées pourraient ne rien valoir. Dans la plupart des cas où des problèmes sont découverts, c'est l'instrumentation qui est erronée, pas le système sous-jacent. En discutant des informations récupérées en utilisant un tel système, il est essentiel de s'assurer que le script utilisé est lui-aussi vérifié et discuter.

28.5.4. Définir de nouvelles sondes

De nouvelles sondes peuvent être définies dans le code partout où le développeur le souhaite bien que cela nécessite une nouvelle compilation. Voici les étapes nécessaires pour insérer de nouvelles sondes :

1. Décider du nom de la sonde et des données nécessaires pour la sonde
2. Ajoutez les définitions de sonde dans `src/backend/utils/probes.d`
3. Inclure `pg_trace.h` s'il n'est pas déjà présent dans le module contenant les points de sonde, et insérer les macros `TRACE_POSTGRESQL` aux emplacements souhaités dans le code source
4. Recompiler et vérifier que les nouvelles sondes sont disponibles

Exemple : Voici un exemple d'ajout d'une sonde pour tracer toutes les nouvelles transactions par identifiant de transaction.

1. La sonde sera nommée `transaction-start` et nécessite un paramètre de type `LocalTransactionId`
2. Ajout de la définition de la sonde dans `src/backend/utils/probes.d` :

   ```
   probe transaction__start(LocalTransactionId);
   ```

   Notez l'utilisation du double tiret bas dans le nom de la sonde. Dans un script DTrace utilisant la sonde, le double tiret bas doit être remplacé par un tiret, donc `transaction-start` est le nom à documenter pour les utilisateurs.
3. Au moment de la compilation, `transaction__start` est converti en une macro appelée `TRACE_POSTGRESQL_TRANSACTION_START` (notez que les tirets bas ne sont plus doubles ici), qui est disponible en incluant le fichier `pg_trace.h`. Il faut ajouter l'appel à la macro aux bons emplacements dans le code source. Dans ce cas, cela ressemble à :

   ```
   TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
   ```

4. Après une nouvelle compilation et l'exécution du nouveau binaire, il faut vérifier que la nouvelle sonde est disponible en exécutant la commande DTrace suivante. Vous deviez avoir cette sortie :

```
# dtrace -ln transaction-start
   ID    PROVIDER          MODULE           Fonction NAME
18705 postgresql49878     postgres     StartTransactionCommand
transaction-start
18755 postgresql49877     postgres     StartTransactionCommand
transaction-start
18805 postgresql49876     postgres     StartTransactionCommand
transaction-start
18855 postgresql49875     postgres     StartTransactionCommand
transaction-start
18986 postgresql49873     postgres     StartTransactionCommand
transaction-start
```

Il faut faire attention à d'autres choses lors de l'ajout de macros de trace dans le code C :

- Vous devez faire attention au fait que les types de données indiqués pour les paramètres d'une sonde correspondent aux types de données des variables utilisées dans la macro. Dans le cas contraire, vous obtiendrez des erreurs de compilation.

- Sur la plupart des plateformes, si PostgreSQL est construit avec `--enable-dtrace`, les arguments pour une macro de trace seront évalués à chaque fois que le contrôle passe dans la macro, *même si aucun traçage n'est réellement en cours*. Cela a généralement peu d'importance si vous rapportez seulement les valeurs de quelques variables locales, mais faites bien attention à l'utilisation de fonctions coûteuses. Si vous devez le faire, pensez à protéger la macro avec une vérification pour vous assurer que la trace est bien activée :

```
if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
    TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
```

  Chaque macro de trace a une macro `ENABLED` correspondante.

Chapitre 29. Surveiller l'utilisation des disques

Ce chapitre explique comment surveiller l'utilisation que fait PostgreSQL des disques.

29.1. Déterminer l'utilisation des disques

Chaque table possède un fichier principal (*heap*) dans lequel la majorité des données sont stockées. Si la table a des colonnes potentiellement grandes en taille, il pourrait aussi y avoir un fichier TOAST associé à la table. Ce fichier permet de stocker les valeurs trop larges pour tenir dans la table principale (voir la Section 73.2). Si la table TOAST existe, un index valide lui est associé. Des index peuvent également être associés à la table de base. Chaque table ou index est stocké dans un fichier distinct -- voire plus si la taille du fichier dépasse 1 Go. Les conventions de nommage de ces fichiers sont décrites dans la Section 73.1.

L'espace disque peut être surveillé de trois façons différentes : en utilisant les fonctions SQL listées dans Tableau 9.94, en utilisant le module oid2name ou en inspectant manuellement les catalogues système. Les fonctions SQL sont les plus simples à utiliser et sont généralement recommandées. Le reste de cette section montre comment le faire en inspectant les catalogues système.

Sur une base de données récemment « nettoyée » (`VACUUM`) ou « analysée » (`ANALYZE`), psql permet de lancer des requêtes pour voir l'occupation disque d'une table :

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE
 relname = 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

Chaque page a une taille de 8 ko, typiquement. (Rappelez-vous que `relpages` est seulement mis à jour par `VACUUM`, `ANALYZE` et quelques commandes DDL telles que `CREATE INDEX`.) Le chemin du fichier n'a d'intérêt que si vous voulez examiner directement le fichier de la table.

Pour connaître l'espace disque utilisé par les tables TOAST, on utilise une requête similaire à la suivante :

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM pg_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

On peut aussi facilement afficher la taille des index :

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

       relname     | relpages
-------------------+----------
 customer_id_index |       26
```

Les tables et les index les plus volumineux sont repérés à l'aide de la requête suivante :

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

29.2. Panne pour disque saturé

La tâche la plus importante d'un administrateur de base de données, en ce qui concerne la surveillance des disques, est de s'assurer que les disques n'arrivent pas à saturation. Un disque de données plein ne corrompt pas les données mais peut empêcher toute activité. S'il s'agit du disque contenant les fichier WAL, une alerte PANIC et un arrêt du serveur peuvent survenir.

S'il n'est pas possible de libérer de la place sur le disque, il faut envisager le déplacement de quelques fichiers vers d'autres systèmes de fichiers à l'aide des *tablespaces*. Voir la Section 23.6 pour plus d'informations.

> **Astuce**
>
> Certains systèmes de fichiers réagissent mal à proximité des limites de remplissage. Il est donc préférable de ne pas attendre ce moment pour réagir.

Si le système supporte les quotas disque par utilisateur, la base de données est alors soumise au quota de l'utilisateur qui exécute le serveur de base de données. Dépasser ce quota a les mêmes conséquences néfastes qu'un disque plein.

Chapitre 30. Fiabilité et journaux de transaction

Ce chapitre explique comment les journaux de transaction sont utilisés pour obtenir des traitements efficaces et fiables.

30.1. Fiabilité

La fiabilité est une propriété importante de tout système de base de données sérieux. PostgreSQL fait tout son possible pour garantir un fonctionnement fiable. Un des aspects de la fiabilité est de stocker toutes les données validées par une transaction dans un espace non volatile, insensible aux coupures de courant, aux bogues du système d'exploitation et aux problèmes matériels (sauf en cas de problème sur l'espace non volatile, bien sûr). Pour cela, il est habituellement suffisant d'écrire avec succès les données sur le stockage permanent de l'ordinateur (disque dur ou un équivalent). En fait, même si un ordinateur est irrémédiablement hors d'usage, si le disque dur survit, ces données peuvent être déplacées vers un autre ordinateur au matériel similaire et toutes les transactions validées resteront intactes.

Bien que forcer l'enregistrement des données périodiquement sur le disque semble être une opération simple, ce n'est pas le cas. Comme les disques durs sont beaucoup plus lents que la mémoire principale et les processeurs, plusieurs niveaux de cache existent entre la mémoire principale de l'ordinateur et les disques. Tout d'abord, il y a le tampon cache du système d'exploitation, qui met en cache les blocs disque fréquemment utilisés et combine les écritures. Heureusement, tous les systèmes d'exploitation fournissent aux applications un moyen de forcer les écritures du cache vers le disque et PostgreSQL utilise ces fonctions (voir le paramètre wal_sync_method pour en ajuster le fonctionnement).

Ensuite, il peut y avoir un cache dans le contrôleur du disque dur ; ceci est assez courant sur les cartes contrôleur RAID. Certains de ces caches sont *write-through*, ce qui signifie que les écritures sont envoyées au disque dès qu'elles arrivent. D'autres sont *write-back*, ce qui veut dire que les données sont envoyées au lecteur un peu plus tard. De tels caches peuvent être un danger car la mémoire cache du contrôleur du disque est volatile elle perdra donc son contenu à la prochaine coupure de courant. Des cartes contrôleur de meilleure qualité ont des caches *avec batterie* (BBU), c'est-à-dire que la carte dispose d'une batterie qui maintient l'alimentation du cache en cas de coupure de courant. Une fois celui-ci revenu, les données seront écrites sur les disques durs.

Et enfin, la plupart des disques durs ont des caches. Certains sont *write-through* alors que d'autres sont *write-back*, et les mêmes problèmes se posent pour les caches *write-back* des disques que pour ceux de contrôleurs. Les disques IDE et SATA grand public en particulier sont susceptibles d'avoir des caches « write-back » qui ne survivront pas à une perte de courant. De nombreux SSD sont aussi dotés de caches « write-back » volatiles.

Ces caches peuvent généralement être désactivés. Néanmoins, la méthode pour le faire dépend du système d'exploitation et du type de disque :

- Sur Linux, on peut interroger les disques IDE et SATA avec la commande `hdparm -I` ; le cache en écriture est activé si une étoile (`*`) se trouve derrière le texte `Write cache`. `hdparm -W 0` peut être utilisé pour désactiver le cache en écriture. Les disques SCSI peuvent être vérifiés en utilisant sdparm[1]. Utilisez `sdparm --get=WCE` pour voir si le cache en écriture est activé et `sdparm --clear=WCE` pour le désactiver.

- Sur FreeBSD, les disques IDE peuvent être interrogés avec `camcontrol` et le cache en écriture désactivé avec `hw.ata.wc=0` dans le fichier de configuration `/boot/loader.conf` ; les disques SCSI peuvent être interrogés avec `camcontrol identify`, et le cache en écriture peut être consulté et modifié en utilisant `sdparm` quand cette commande est disponible.

[1] http://sg.danny.cz/sg/sdparm.html

- Sur Solaris, le cache disque en écriture est contrôlé par `format -e`. (Le système de fichiers Solaris ZFS est sûr même quand le cache disque en écriture est activé car il exécute ses propres commandes de vidage du cache.)

- Sur Windows, si `wal_sync_method` vaut `open_datasync` (la valeur par défaut), le cache en écriture peut être désactivé en décochant `My Computer\Open\`*`disk drive`*`\Properties\Hardware\Properties\Policies\Enable write caching on the disk`. Ou bien configurez `wal_sync_method` à `fsync` ou `fsync_writethrough` pour désactiver le cache en écriture.

- Sur macOS, le cache en écriture peut être évité en configurant `wal_sync_method` à `fsync_writethrough`.

Les disques SATA récents (ceux compatibles ATAPI-6 ou supérieurs) proposent une commande pour vider le cache sur le disque (`FLUSH CACHE EXT`) alors que les disques SCSI proposent depuis longtemps une commande similaire, `SYNCHRONIZE CACHE`. Ces commandes ne sont pas directement accessibles à PostgreSQL, mais certains systèmes de fichiers (comme ZFS, ext4) peuvent les utiliser pour vider les données sur disque pour les disques au cache en écriture activé. Malheureusement, ces systèmes de fichiers se comportent de façon non optimale avec des contrôleurs disque équipés de batterie (BBU, acronyme de *Battery-Backed Unit*). Dans ce type de configuration, la commande de synchronisation force l'écriture de toutes les données comprises dans le cache sur les disques, éliminant ainsi tout l'intérêt d'un cache protégé par une batterie. Vous pouvez lancer l'outil pg_test_fsync, disponible dans le code source de PostgreSQL, pour vérifier si vous êtes affecté. Si vous l'êtes, les améliorations de performance du cache BBU peuvent être de nouveaux obtenues en désactivant les barrières d'écriture dans la configuration du système de fichiers ou en reconfigurant le contrôleur de disque, si cela est possible. Si les barrières d'écriture sont désactivées, assurez-vous que la batterie reste active. Une batterie défectueuse peut être une cause de perte de données. Il reste à espérer que les concepteurs de systèmes de fichiers et de contrôleurs disque finissent par corriger ce comportement gênant.

Quand le système d'exploitation envoie une demande d'écriture au système de stockage, il ne peut pas faire grand-chose pour s'assurer que les données sont arrivées dans un espace de stockage réellement non volatil. C'est plutôt la responsabilité de l'administrateur de s'assurer que tous les composants de stockage garantissent l'intégrité des données et des métadonnées du système de fichier. Évitez les contrôleurs disques ne disposant pas de caches protégés par batterie. Au niveau du disque, désactivez le cache « write-back » si le disque ne peut garantir que les données seront écrites avant un arrêt. Si vous utilisez des disques SSD, sachez que beaucoup n'honorent pas les commandes de vidage de cache par défaut. Vous pouvez tester la fiabilité du comportement du système disque en utilisant `diskchecker.pl`[2].

Un autre risque de perte de données est dû aux opérations d'écriture sur les plateaux du disque. Les plateaux sont divisés en secteurs de 512 octets généralement. Chaque opération de lecture ou écriture physique traite un secteur entier. Quand une demande d'écriture arrive au lecteur, sa taille peut être un multiple de 512 octets (PostgreSQL écrit généralement 8192 octets, soit 16 secteurs, à la fois) et le processus d'écriture peut échouer à cause d'une coupure de courant à tout moment, ce qui implique que certains octets peuvent être écrits et les autres perdus. Pour se prévenir contre ce type d'échec, PostgreSQL écrit périodiquement des pages complètes dans le stockage permanent des journaux de transactions *avant* de modifier la page réelle sur disque. Ainsi, lors d'une récupération après un arrêt brutal, PostgreSQL peut restaurer les pages écrites partiellement à partir des journaux de transactions. Si vous avez un système de fichiers qui vous protège contre les écritures de pages incomplètes (par exemple ZFS), vous pouvez désactiver la création des images de page en utilisant le paramètre full_page_writes. Les contrôleurs disques disposant d'une batterie (BBU pour *Battery-Backed Unit*) n'empêchent pas les écritures de pages partielles sauf s'ils garantissent que les données sont écrites par pages complètes de 8 ko.

PostgreSQL protège aussi de certaines corruptions de données au niveau du support de stockage qui peuvent se produire suite à des erreurs au niveau matériel ou des problèmes d'usure au fil du temps, comme la lecture ou l'écriture de données invalides.

[2] https://brad.livejournal.com/2116715.html

- Dans un journal de transaction, chaque enregistrement est protégé par une somme de contrôle CRC-32C (32 bits) qui nous dit si le contenu de l'enregistrement est correct. La valeur CRC est définie à l'écriture de chaque enregistrement dans le journal et vérifiée durant une récupération après arrêt brutal, la récupération d'une archive et la réplication.

- Par défaut, les pages de données ne disposent pas de sommes de contrôle, mais les images des pages complètes stockées dans les enregistrements des journaux de transactions seront protégées. Voir initdb pour les détails sur l'activation des sommes de contrôle sur les pages de données.

- Les structures de données internes comme `pg_xact`, `pg_subtrans`, `pg_multixact`, `pg_serial`, `pg_notify`, `pg_stat`, `pg_snapshots` n'ont pas de sommes de contrôle, et leurs pages de données ne sont pas protégées par les écritures de pages complètes. Cependant, lorsque de telles structures de données sont persistantes, les enregistrements des journaux de transactions sont écrits de manière à ce que les modifications récentes puissent être rapidement reconstruites lors d'une restauration après incident, et ces enregistrements sont protégés comme décrit plus haut.

- Les fichiers d'état de `pg_twophase` sont protégés chacun par une somme de contrôle CRC-32C.

- Les fichiers de données temporaires utilisés lors de grosses requêtes SQL pour le tri, la matérialisation ou des résultats intermédiaires ne font pas actuellement l'objet d'une somme de contrôle, et la modification de ces fichiers n'est pas non plus consignée dans les enregistrements des journaux de transactions.

PostgreSQL ne protége pas contre les erreurs mémoires et il est supposé que vous travaillez avec de la RAM avec correction d'erreur (ECC) aux standards de l'industrie, ou une meilleure protection.

30.2. Sommes de contrôle pour les données

Par défaut, les blocs de données ne sont pas protégées par des sommes de contrôle, mais ceci peut être activé en option pour une instance. Une fois activé, chaque bloc de données inclut une somme de contrôle qui est mise à jour quand la page est écrite et vérifiée chaque fois que le bloc est lu. Seuls les blocs de données sont protégés par des sommes de contrôle ; les structures de données internes et les fichiers temporaires ne le sont pas.

Les sommes de contrôle peuvent être activées quand l'instance est initialisée en utilisant initdb. Elles peuvent aussi être activées ou désactivées plus tard lors d'une opération où l'instance est arrêtée. Les sommes de contrôle sont activées ou désactivées au niveau de l'instance et ne peuvent pas activées spécifiquement pour certaines bases ou tables.

L'état actuel des sommes de contrôle dans l'instance peut être vérifié en regardant la valeur de la variable de configuration en lecture seule data_checksums, par exemple en exécutant la commande SQL `SHOW data_checksums`.

Lors d'une tentative de récupération suite à une corruption de blocs, il pourrait être nécessaire de contourner la protection par somme de contrôle. Pour cela, utilisez temporairement le paramètre de configuration ignore_checksum_failure.

30.2.1. Vérification hors ligne des sommes de contrôle

L'outil pg_checksums peut être utilisé pour activer ou désactiver les sommes de contrôle, ainsi que pour vérifier ces sommes, toujours sur une instance arrêtée.

30.3. Write-Ahead Logging (WAL)

Write-Ahead Logging (ou WAL, pour « écriture de journaux en amont ») est une méthode courante pour garantir l'intégrité des données. Une description détaillée peut être trouvée dans la plupart des

livres sur le traitement transactionnel, sinon tous. En résumé, le concept central du WAL est de ne modifier les fichiers de données (donc les tables et les index) qu'après que les changements ont été journalisés, c'est-à-dire qu'après que les enregistrements décrivant ces changements ont été écrits sur du stockage permanent. Si nous suivons cette procédure, nous n'avons pas besoin d'écrire les pages de données vers le disque à chaque validation de transaction car nous savons que, dans l'éventualité d'une défaillance, nous serons capables de récupérer la base de données en utilisant le journal : chaque changement qui n'a pas été appliqué aux pages de données peut être ré-exécuté depuis les enregistrements du journal. (Ceci est une récupération *roll-forward*, aussi connue sous le nom de REDO).

> **Astuce**
>
> Comme les journaux de transactions permettent de restaurer le contenu des fichiers de base de données après un arrêt brutal, un système de fichiers journalisé n'est pas nécessaire pour stocker avec fiabilité les fichiers de données ou les journaux de transactions. En fait, le surcroît de travail lié à la journalisation peut réduire les performances, tout spécialement si la journalisation cause l'écriture des *données* sur disque. Heureusement, l'écriture des données lors de la journalisation peut souvent être désactivée avec une option de montage du système de fichiers, par exemple `data=writeback` sur un système de fichiers Linux ext3. Par contre, les systèmes de fichiers journalisés accélèrent le redémarrage après un arrêt brutal.

Utiliser les journaux de transactions permet de réduire de façon significative le nombre d'écritures sur le disque : seul le journal a besoin d'être écrit sur le disque pour garantir qu'une transaction a été validée, il n'y a pas besoin d'écrire dans chaque fichier de données modifié par la transaction. Ce journal est écrit séquentiellement, ainsi le coût de vidage sur disque du journal est largement moindre que le coût d'écriture des pages de données. Ceci est tout spécialement vrai pour les serveurs gérant beaucoup de petites transactions touchant différentes parties du stockage de données. De plus, quand le serveur traite beaucoup de petites transactions en parallèle, un `fsync` du journal des transactions peut suffire pour enregistrer plusieurs transactions.

Les journaux de transactions rendent possible le support de sauvegarde en ligne et de récupération à un moment donné dans le temps, comme décrit dans la Section 26.3. En archivant les journaux de transaction, nous permettons un retour à tout instant couvert par les données disponibles dans les journaux de transaction : nous installons simplement une ancienne sauvegarde physique de la base de données et nous rejouons les journaux de transactions jusqu'au moment désiré. Qui plus est, la sauvegarde physique n'a pas besoin d'être une image instantanée de l'état de la base de données -- si elle a été faite pendant une longue période de temps, alors rejouer les journaux de transactions pour cette période corrigera toute incohérence interne.

30.4. Validation asynchrone (Asynchronous Commit)

La *validation asynchrone* est une option qui permet aux transactions de se terminer plus rapidement. Le risque encouru est de perdre les transactions les plus récentes dans le cas où le serveur s'arrête brutalement. Dans beaucoup d'applications, le compromis est acceptable.

Comme le décrit la section précédente, la validation d'une transaction est habituellement *synchrone* : le serveur attend que les enregistrements des journaux de transaction soient bien sauvegardés sur le stockage permanent avant d'informer le client du succès de l'opération. Le client a donc la garantie qu'une transaction validée est stockée de façon sûre, donc même en cas d'arrêt brutal immédiatement après. Néanmoins, pour les petites transactions, ce délai est une partie importante de la durée totale d'exécution de la transaction. Sélectionner le mode de validation asynchrone signifie que le serveur annonce le succès de l'opération dès que la transaction est terminée logiquement, donc avant que les enregistrements du journal de transaction que cette transaction a générés ne soient réellement stockés sur disque. Ceci peut apporter une accélération importante pour les petites transactions.

La validation asynchrone introduit le risque des pertes de données. Il existe un petit délai entre le moment où le rapport de la fin d'une transaction est envoyé au client et celui où la transaction est réellement enregistrée (c'est-à-dire le moment où le résultat de cette transaction ne pourra pas être perdu même en cas d'arrêt brutal du serveur). Du coup, la validation asynchrone ne devrait pas être utilisée si le client se base sur le fait que la transaction est enregistrée de façon sûre. Par exemple, une banque ne devra pas utiliser la validation asynchrone pour une transaction enregistrant la sortie d'argent d'un distributeur bancaire. Dans de nombreux autres scénarios, comme la trace d'événements, il n'y a pas besoin de garantie forte de ce type.

Le risque pris avec l'utilisation de la validation asynchrone concerne la perte de données, pas la corruption de données. Si le serveur s'arrête brutalement, il redémarrera en rejouant les journaux de transaction jusqu'au dernier enregistrement vidé sur disque. La base sera donc restaurée dans un état cohérent mais toutes les transactions qui n'auront pas été enregistrées sur disque n'y apparaîtront pas. L'effet immédiat est donc la perte des toutes dernières transactions. Comme les transactions sont rejouées dans l'ordre de validation, aucune incohérence ne sera introduite -- par exemple, si la transaction B fait des modifications sur les effets d'une précédente transaction A, il n'est pas possible que les effets de A soient perdus alors que les effets de B sont préservés.

L'utilisateur peut sélectionner le mode de validation de chaque transaction ; il est donc possible d'avoir en même temps des transactions validées en synchrone et en asynchrone. Cela offre une grande flexibilité dans le choix entre la performance et la certitude de la durabilité des transactions. Le mode de validation est contrôlé par le paramètre utilisateur synchronous_commit, qui peut être modifié comme tout autre paramètre utilisateur. Le mode utilisé pour une transaction dépend de la valeur de `synchronous_commit` quand démarre la validation de la transaction.

Certaines commandes, par exemple `DROP TABLE`, sont forcées en mode synchrone quelle que soit la valeur du paramètre `synchronous_commit`. Ceci a pour but de s'assurer de la cohérence entre le système de fichiers du serveur et l'état logique de la base de données. Les commandes gérant la validation en deux phases (*two-phase commit*), comme `PREPARE TRANSACTION`, sont aussi toujours synchrones.

Si la base de données s'arrête brutalement dans la fenêtre de vulnérabilité entre une validation asynchrone et l'écriture des enregistrements dans le journal des transactions, les modifications réalisées lors de cette transaction *seront* perdues. Cette fenêtre est limitée car un processus en tâche de fond (le « wal writer ») écrit sur le disque les enregistrements non écrits des journaux de transaction toutes les wal_writer_delay millisecondes. La fenêtre de vulnérabilité s'étend en réalité à trois fois `wal_writer_delay` car le processus d'écriture des journaux de transaction est conçu pour favoriser l'écriture de pages complètes lors des périodes de grosses activités.

> **Attention**
>
> Un arrêt en mode immédiat est équivalent à un arrêt brutal et causera du coup la perte des validations asynchrones.

La validation asynchrone fournit un comportement différent de la simple désactivation de fsync. `fsync` est un paramètre pour le serveur entier qui modifie le comportement de toutes les transactions. Il désactive dans PostgreSQL toute la logique qui tente de synchroniser les écritures dans les différentes parties de la base, et donc un arrêt brutal (plus précisément un arrêt brutal du matériel ou du système d'exploitation, pas un plantage de PostgreSQL lui-même) pourrait résulter en une corruption arbitraire de l'état de la base de données. Dans de nombreux scénarios, la validation asynchrone fournit la majorité des améliorations de performances obtenues par la désactivation de `fsync`, mais sans le risque de la corruption de données.

commit_delay semble aussi très similaire à la validation asynchrone mais il s'agit en fait d'une méthode de validation synchrone (en fait, `commit_delay` est ignoré lors d'une validation asynchrone). `commit_delay` provoque un délai avant la mise à jour sur disque du WAL d'une transaction, dans l'espoir que l'opération profite à d'autres transactions commitées à peu près au même moment. Ce

paramètre peut être vu comme le moyen d'augmenter la fenêtre de temps durant laquelle chaque transaction peut participer à un même vidage sur disque, pour amortir son coût sur plusieurs transactions.

30.5. Configuration des journaux de transaction

Il y a plusieurs paramètres de configuration associés aux journaux de transaction qui affectent les performances de la base. Cette section explique leur utilisation. Consultez Chapitre 20 pour des détails sur la mise en place de ces paramètres.

Dans la séquence des transactions, les *checkpoints* (ou *points de contrôle*) sont des points qui garantissent que les fichiers de données des tables et des index ont été mis à jour avec toutes les informations enregistrées dans le journal avant le checkpoint. Au moment du checkpoint, toutes les pages de données modifiées (*dirty*) sont écrites sur le disque et une entrée spéciale, pour le checkpoint, est écrite dans le journal. (Les modifications étaient déjà envoyées dans les journaux de transactions.) En cas de défaillance, la procédure de récupération recherche le dernier enregistrement d'un checkpoint (enregistrement connu sous le nom de « redo log ») pour déterminer le point des journaux à partir duquel il devra lancer l'opération de REDO. Toute modification effectuée sur les fichiers de données avant ce point a la garantie d'avoir été enregistrée sur disque. Du coup, après un checkpoint, tous les segments représentant des journaux de transaction précédant celui contenant le « redo record » ne sont plus nécessaires et peuvent être soit recyclés, soit supprimés (quand l'archivage des journaux de transaction est activé, ces derniers doivent être archivés avant d'être recyclés ou supprimés).

Un checkpoint doit écrire toutes les pages de données modifiées sur disque, ce qui peut causer une charge disque importante. Pour cette raison, l'activité d'un checkpoint est diluée de façon à ce que les entrées/sorties disque commencent au début du checkpoint et se terminent avant le démarrage du checkpoint suivant ; ceci minimise la dégradation des performances lors des checkpoints.

Le processus checkpointer lance automatiquement un checkpoint de temps en temps. Il démarre toutes les checkpoint_timeout secondes ou si max_wal_size risque d'être dépassé, suivant ce qui arrive en premier. La configuration par défaut de ces deux paramètres est, respectivement, 5 minutes et 1 Go. Si aucun enregistrement WAL n'a été écrit depuis le dernier checkpoint, il n'y en aura pas de nouveaux, même si la durée `checkpoint_timeout` est dépassée. (Si l'archivage des WAL est en place et que vous voulez définir une limite basse correspondant à la fréquence d'archivage des fichiers de manière à limiter la perte potentielle de données, vous devez ajuster le paramètre archive_timeout plutôt que les paramètres affectant les checkpoints.) Il est aussi possible de forcer un checkpoint en utilisant la commande SQL `CHECKPOINT`.

La réduction de `checkpoint_timeout` et/ou `max_wal_size` provoque des checkpoints plus fréquents. Cela permet une récupération plus rapide après arrêt brutal puisqu'il y aura moins d'écritures à refaire. Cependant, il faut équilibrer cela avec le coût d'écritures plus fréquentes des pages modifiées. Si full_page_writes est configuré (ce qui est le défaut), il reste un autre facteur à considérer. Pour s'assurer de la cohérence des pages de données, la première modification d'une page de données après chaque checkpoint résulte dans la journalisation du contenu de la page en entier. Dans ce cas, un intervalle de checkpoints plus petit augmentera le volume d'écriture des journaux de transaction, annulant en partie l'intérêt d'utiliser cet intervalle plus petit et générant de toute façon plus d'entrées/sorties au niveau disque.

Les checkpoints sont assez coûteux, tout d'abord parce qu'ils écrivent tous les tampons modifiés jusqu'à ce moment, et ensuite parce qu'ils génèrent un trafic supplémentaire dans les journaux de transaction, comme indiqué ci-dessus. Du coup, il est conseillé de configurer les paramètres des checkpoints assez haut pour qu'ils ne surviennent pas trop fréquemment. Pour une vérification rapide de l'adéquation de vos paramètres, vous pouvez configurer le paramètre checkpoint_warning. Si les checkpoints se déclenchent à moins de `checkpoint_warning` secondes d'intervalle, un message est affiché dans les journaux applicatifs du serveur recommandant d'accroître `max_wal_size`. L'apparition occasionnelle d'un tel message ne doit pas vous alarmer mais, s'il apparaît souvent, alors les paramètres

de contrôle devraient être augmentés. Les opérations en masse, comme les transferts importants de données via `COPY`, peuvent faire apparaître nombre de ces messages d'avertissement si vous n'avez pas configuré `max_wal_size` avec une valeur suffisamment haute.

Pour éviter de saturer les entrées/sorties avec de très nombreuses écritures de pages modifiées, leur écriture pendant le checkpoint est étalée sur une période de temps. Elle est contrôlée par checkpoint_completion_target, donné comme une fraction de l'intervalle des checkpoints(configuré en utilisant `checkpoint_timeout`). Le taux d'entrées/sorties est ajusté pour que le checkpoint se termine quand la fraction indiquée de `checkpoint_timeout` secondes s'est écoulée, ou avant que `max_wal_size` soit dépassé, selon ce qui arrivera en premier. Avec la valeur par défaut de 0,9, on peut s'attendre à ce que PostgreSQL termine chaque checkpoint un peu avant le prochain checkpoint planifié (à environ 90% de la durée du dernier checkpoint). Ceci étale les I/O autant que possible pour que la charge en I/O des checkpoints soit lissée pendant tout l'intervalle des checkpoints. L'inconvénient est que prolonger les checkpoints affecte le temps de redémarrage après crash parce qu'un plus grand nombre de segments WAL devront être conservés pour une possible utilisation après crash. Un utilisateur inquiet par la durée requise pour la restauration après crash pourrait vouloir réduire `checkpoint_timeout` pour que les checkpoints surviennent plus fréquemment, tout en étalant les I/O sur l'intervalle des checkpoints. Alternativement, `checkpoint_completion_target` pourrait être réduit mais cela résulterait en des moments avec des I/O plus intenses (lors du checkpoint) et des moments avec des I/O moins intenses (après la fin du checkpoint mais avant le début du suivant). De ce fait, cela n'est pas recommandé. Bien que `checkpoint_completion_target` puisse être configuré aussi haut que 1,0, il est typiquement recommandé de le configurer pas plus haut que 0,9 (la valeur par défaut) car les checkpoints ont quelques autres activités en dehors de l'écriture du cache modifié. Une valeur de 1,0 peut résulter en checkpoints qui ne se terminent pas à temps, ce qui entraînerait des baisses de performance à cause de variations inattendues dans le nombre de journaux nécessaires.

Sur les plateformes Linux et POSIX, checkpoint_flush_after permet de forcer le système d'exploitation à vider sur disque les pages écrites par un checkpoint après qu'un nombre configurable d'octets a été écrit. Sinon ces pages pourraient rester dans le cache disque du système d'exploitation, provoquant un blocage quand `fsync` est exécuté à la fin d'un checkpoint. Cette configuration aide souvent à réduire la latence des transactions mais il peut aussi avoir un effet inverse sur les performances, particulièrement pour des charges supérieures à shared_buffers mais plus petites que le cache disque du système d'exploitation.

Le nombre de fichiers de segments WAL dans le répertoire `pg_wal` dépend des paramètres `min_wal_size`, `max_wal_size` et de la quantité de WAL générée lors des cycles de checkpoints précédents. Quand les anciens fichiers de segments ne sont plus nécessaires, ils sont supprimés ou recyclés (c'est-à-dire renommés pour devenir les segments suivants selon les numéros de la séquence). Si, à cause d'un bref pic du débit des WAL, `max_wal_size` est dépassé, les fichiers inutiles seront supprimés jusqu'à ce que le système revienne sous cette limite. En-dessous de cette limite, le système recycle suffisamment de fichiers WAL pour couvrir le besoin estimé jusqu'au checkpoint suivant, et supprime le reste. L'estimation est basée sur une moyenne glissante du nombre de fichiers WAL utilisés dans les cycles de checkpoint précédents. Elle est augmentée immédiatement si l'utilisation en cours dépasse l'estimation, pour correspondre aux pics d'utilisation plutôt qu'à l'utilisation moyenne, jusqu'à un certain point. `min_wal_size` définit un nombre minimum de fichiers WAL recyclés pour une utilisation future, même si le système est inutilisé et que l'estimation suggère que peu de WAL sont nécessaires.

Indépendamment de `max_wal_size`, les wal_keep_size méga-octets les plus récents des fichiers WAL et un fichier WAL supplémentaire sont conservés en permanence. De plus, si l'archivage est activé, les anciens segments ne sont ni supprimés ni recyclés jusqu'à la réussite de leur archivage. Si l'archivage des WAL n'est pas assez rapide pour tenir le rythme de la génération des WAL, ou si `archive_command` ou `archive_library` échoue de manière répétée, les anciens fichiers WAL s'accumuleront dans le répertoire `pg_wal` jusqu'à ce que ce problème soit résolu. Un serveur standby lent ou en échec, et qui utilise un slot de réplication, aura le même effet (voir Section 27.2.6).

En mode de restauration d'archive et en mode standby, le serveur réalise périodiquement des *restartpoints* (points de redémarrage), qui sont similaires aux checkpoints lors du fonctionnement

normal : le serveur force l'écriture de son état sur disque, met à jour le fichier `pg_control` pour indiquer que les données déjà traitées des journaux de transactions n'ont plus besoin d'être parcourues de nouveau, puis recycle les anciens journaux de transactions trouvés dans le répertoire `pg_wal`. Les restartpoints ne peuvent être réalisés plus fréquemment que les checkpoints du primaire car les restartpoints peuvent seulement être réalisés aux enregistrements de checkpoint. Un restartpoint est déclenché lorsqu'un enregistement de checkpoint est atteint si un minimum de `checkpoint_timeout` secondes se sont écoulées depuis le dernier restartpoint, ou si la taille totale des journaux de transactions va dépasser `max_wal_size`. Néanmoins, à cause de ces limitations sur quand un restartpoint peut être effectué, `max_wal_size` est souvent dépassé lors d'une restauration jusqu'à au plus un cycle de checkpoint de journaux (`max_wal_size` n'est de toute façon jamais une limite en dur, vous devriez donc toujours laisser plein d'espace pour éviter de manquer d'espace disque).

Il existe deux fonctions WAL internes couramment utilisées : `XLogInsertRecord` et `XLogFlush`. `XLogInsertRecord` est utilisée pour placer une nouvelle entrée à l'intérieur des tampons WAL en mémoire partagée. S'il n'y a plus d'espace pour la nouvelle entrée, `XLogInsertRecord` devra écrire (autrement dit, déplacer dans le cache du noyau) quelques tampons WAL remplis. Ceci n'est pas souhaitable parce que `XLogInsertRecord` est utilisée à chaque modification bas niveau de la base (par exemple, lors de l'insertion d'une ligne) quand un verrou exclusif est posé sur les pages de données affectées, et l'opération doit donc être aussi rapide que possible. Pire encore, écrire des tampons WAL peut aussi forcer la création d'un nouveau journal, ce qui prend encore plus de temps. Normalement, les tampons WAL doivent être écrits et vidés par un appel à `XLogFlush` fait, la plupart du temps, au moment de la validation d'une transaction pour assurer que les entrées de la transaction sont écrites vers un stockage permanent. Sur les systèmes avec une importante écriture de journaux, les requêtes de `XLogFlush` peuvent ne pas arriver assez souvent pour empêcher `XLogInsert` d'avoir à écrire lui-même. Sur de tels systèmes, on devrait augmenter le nombre de tampons WAL en modifiant le paramètre wal_buffers. Quand full_page_writes est configuré et que le système est très occupé, configurer `wal_buffers` avec une valeur plus importante aide à lisser les temps de réponse dans la période suivant immédiatement chaque checkpoint.

Le paramètre commit_delay définit combien de micro-secondes un processus maître d'un groupe de commit va s'endormir après avoir obtenu un verrou avec `XLogFlush`, pendant que les autres processus du groupe vont s'ajouter à la queue derrière le maître. Ce délai permet aux autres processus serveur d'ajouter leurs enregistrements de commit aux buffers WAL, pour qu'ils soient tous écrits par un éventuel vidage sur disque du maître. Il n'y aura pas d'endormissement si fsync n'est pas activé, ou si moins de commit_siblings autres sessions sont actuellement dans une transaction active ; ce mécanisme évite l'endormissement quand il est improbable que d'autres sessions valident bientôt leurs transactions. Il est à noter que, sur certaines plateformes, la résolution d'une requête d'endormissement est de dix millisecondes, ce qui implique que toute valeur comprise entre 1 et 10000 pour le paramètre `commit_delay` aura le même effet. Notez aussi que, sur certaines plateformes, les opérations d'endormissement peuvent être légèrement plus longues que ce qui a été demandé par le paramètre.

Comme l'objet de `commit_delay` est de permettre d'amortir le coût de chaque opération de vidage sur disque sur plusieurs transactions concurrentes (potentiellement au prix de la latence des transactions), il est nécessaire de quantifier ce coût pour choisir intelligemment la valeur de ce paramètre. Plus le coût est élevé, plus `commit_delay` sera efficace au sein d'un débit de transactions croissant, jusqu'à un certain point. Le programme pg_test_fsync peut être utilisé pour mesurer le temps moyen en microsecondes que prend une seule opération de vidage de WAL. La moitié du temps moyen rapporté par ce programme pour une mise à jour d'une simple opération d'écriture de 8 ko est la valeur la plus souvent recommandée comme point de départ de l'optimisation d'une charge particulière. Bien que l'ajustement de la valeur de `commit_delay` soit particulièrement utile lorsque les journaux WAL sont stockés sur des disques à latence élevée, le gain peut aussi être significatif sur les supports de stockage avec des temps de synchronisation très rapides, comme les SSD ou les grappes RAID avec des caches en écriture dotés de batterie ; mais dans tous les cas, cela doit être testé avec une charge représentative de la réalité. Des valeurs plus élevées de `commit_siblings` peuvent être utilisées dans ce cas, alors que de petites valeurs de `commit_siblings` sont souvent utiles sur des supports de grande latence. À noter qu'il est possible qu'une valeur trop élevée de `commit_delay` augmente la latence des transactions à un tel point que le débit des transactions en souffre.

Lorsque commit_delay est défini à zéro (il s'agit de la valeur par défaut), il est toujours possible qu'un regroupement de commits se produise, mais chaque groupe ne consistera qu'en sessions atteignant le moment de l'enregistrement de commit pendant le laps de temps où la précédente opération de vidage (s'il y en a) opère. Avec un grand nombre de clients, un « effet tunnel » (*gangway effect*) a tendance à se produire, et ainsi les effets du regroupement de commits deviennent significatifs même lorsque commit_delay est à zéro, et dans ce cas commit_delay devient inutile. Définir commit_delay n'est utile que quand (1) il existe des transactions concurrentes, et (2) le débit est limité dans une certaine mesure par la vitesse de commit ; mais, dans le cas d'un temps de latence du disque élevé, ce paramètre peut augmenter efficacement le flux de transaction avec seulement deux clients (c'est-à-dire un unique client qui valide, et une transaction sœur).

Le paramètre wal_sync_method détermine comment PostgreSQL demande au noyau de forcer les mises à jour des journaux de transaction sur le disque. Toutes les différentes options devraient être identiques en terme de fiabilité, à l'exception de fsync_writethrough, qui peut parfois forcer une écriture du cache disque même quand d'autres options ne le font pas. Néanmoins, connaître l'option la plus rapide est assez dépendant de la plateforme. Vous pouvez tester les vitesses des différentes options en utilisant le programme pg_test_fsync. Notez que ce paramètre est ignoré si fsync a été désactivé.

Activer le paramètre de configuration wal_debug (à supposer que PostgreSQL ait été compilé avec le support de ce paramètre) permet d'enregistrer chaque appel WAL à XLogInsertRecord et XLogFlush dans les journaux applicatifs du serveur. Cette option pourrait être remplacée par un mécanisme plus général dans le futur.

Il existe deux fonctions internes pour écrire des données WAL sur disque : XLogWrite et issue_xlog_fsync. Quand track_wal_io_timing est activé, le nombre total de fois où XLogWrite écrit et issue_xlog_fsync synchronise des données WAL sur disque est compté comme, respectivement, wal_write_time et wal_sync_time dans pg_stat_wal. XLogWrite est normalement appelé par XLogInsertRecord (quand il n'y a plus d'espace pour le nouvel enregistrement dans le cachedes WAL), XLogFlush et le walwriter, pour écrire le cache WAL sur disque et appeler issue_xlog_fsync. issue_xlog_fsync est normalement appelé par XLogWrite pour synchroniser les fichiers WAL sur disque. Si wal_sync_method vaut soit open_datasync soit open_sync, une opération d'écriture dans XLogWrite garantit de synchroniser les données des WAL écrites sur disque et issue_xlog_fsync ne fait rien. Si wal_sync_method vaut soit fdatasync, soit fsync, soit fsync_writethrough, les opérations en écriture déplacent le cache WAL dans le cache du noyau et issue_xlog_fsync les synchronise sur disque. Quelle que soit la configuration de track_wal_io_timing, le nombre de fois où XLogWrite écrit et que issue_xlog_fsync synchronise les données WAL sur disque sont aussi comptées dans, respectivement, wal_write et wal_sync de la vue pg_stat_wal.

Le paramètre recovery_prefetch peut être utilisé pour réduite les durées d'attente en entrée/sortie pendant une restauration en indiquant au niveau d'initier des lecteurs de blocs disques qui seront bientôt nécessaires mais qui ne sont pas actuellement dans le cache disque de PostgreSQL. Les paramètres maintenance_io_concurrency et wal_decode_buffer_size limitent, respectivement, la concurrence et la distance sur les lectures en avance. Par défaut, c'est configuré à try, ce qui active la fonctionnalité sur les systèmes où posix_fadvise est disponible.

30.6. Vue interne des journaux de transaction

Le mécanisme WAL est automatiquement activé ; aucune action n'est requise de la part de l'administrateur, sauf s'assurer que l'espace disque requis par les journaux de transaction est présent et que tous les réglages nécessaires sont faits (voir la Section 30.5).

Les enregistrements WAL sont ajoutés aux journaux WAL, enregistrement après enregistrement. La position d'insertion est donnée par le *Log Sequence Number* (LSN, pour numéro de séquence de journal) qui est un décalage d'octets (*offset*) au sein des journaux de transactions, qui s'incrémente de manière monotone à chaque enregistrement. Les valeurs du LSN sont renvoyées en tant que type de données pg_lsn. Les valeurs peuvent être comparées pour calculer le volume de données WAL les séparant, permettant ainsi de mesurer l'avancement de la réplication et de la restauration.

Les journaux de transactions sont un ensemble de fichiers stockés dans le répertoire `pg_wal` sous celui des données, chacun d'une taille de 16 Mo normalement (cette taille pouvant être modifiée en modifiant l'option `--wal-segsize` d'initdb). Chaque fichier est divisé en pages de généralement 8 ko (cette taille pouvant être modifiée avec l'option `--with-wal-blocksize` de configure). Les en-têtes d'une entrée de journal sont décrites dans `access/xlogrecord.h` ; le contenu d'une entrée dépend du type de l'événement qui est enregistré. Les fichiers sont nommés suivant des nombres continûment incrémentés, commençant par `000000010000000000000001`. Les nombres ne bouclent pas, mais cela prendra beaucoup, beaucoup de temps pour épuiser le stock de nombres disponibles.

Il est avantageux que les journaux soient situés sur un autre disque que celui des fichiers principaux de la base de données. Cela peut se faire en déplaçant le répertoire `pg_wal` vers un autre emplacement (serveur arrêté, bien sûr) et en créant dans le répertoire principal de données un lien symbolique de l'emplacement original vers le nouveau.

Le but de WAL est de s'assurer que le journal est écrit avant de modifier les enregistrements de la base, mais cela peut être mis en échec par des disques qui rapportent une écriture réussie au noyau quand, en fait, ils ont seulement mis en cache les données et ne les ont pas encore stockées sur le disque. Une coupure de courant dans ce genre de situation peut mener à une corruption irrécupérable des données. Les administrateurs devraient s'assurer que les disques contenant les journaux de transaction de PostgreSQL ne produisent pas ce genre de faux rapports. (Voir Section 30.1.)

Après qu'un checkpoint a été fait et le journal vidé sur disque, la position du checkpoint est sauvegardée dans le fichier `pg_control`. Donc, au début de la récupération, le serveur lit en premier `pg_control` et ensuite l'entrée du checkpoint ; ensuite, il opère le REDO en progressant à partir de la position du journal indiquée dans l'entrée du checkpoint. Parce que l'ensemble du contenu des pages de données est sauvegardé dans le journal à la première modification de page après un checkpoint (en supposant que full_page_writes n'est pas désactivé), toutes les pages modifiées depuis le checkpoint seront restaurées dans un état cohérent.

Pour gérer le cas où `pg_control` est corrompu, nous devrions permettre le parcours des segments de journaux existants en ordre inverse -- du plus récent au plus ancien -- pour trouver le dernier checkpoint. Ceci n'a pas encore été implémenté. `pg_control` est assez petit (moins d'une page disque) pour ne pas être sujet aux problèmes d'écriture partielle et, au moment où ceci est écrit, il n'y a eu aucun rapport de défaillance d'une base due uniquement à l'incapacité à lire `pg_control`. Donc, bien qu'étant théoriquement un point faible, `pg_control` ne semble pas être un problème en pratique.

Chapitre 31. Réplication logique

La réplication logique est une méthode permettant de répliquer des données au niveau objet ainsi que les modifications apportées à ces objets, ceci basé sur leur identité de réplication (habituellement la clé primaire). L'utilisation du terme de « réplication logique » est faite en opposition à la réplication physique, cette dernière utilisant l'adresse exacte des blocs couplée avec une réplication octet par octet. PostgreSQL supporte ces deux méthodes, référez-vous à l'article Chapitre 27. La réplication logique permet un contrôle fin des données au niveau de la réplication et de la sécurité.

La réplication logique utilise un système de *publication* / *abonnement* avec un ou plusieurs *abonnés* qui s'abonnent à une ou plusieurs *publications* d'un nœud particulier. Les abonnés récupèrent les données des publications auxquelles ils sont abonnés et peuvent éventuellement renvoyer ces informations, ce qui permet un système de réplication en cascade dans le cas de configurations plus complexes.

La réplication logique d'une table commence en général en prenant un instantané des données sur la base publiée et en le copiant vers la base abonnée. Une fois cette étape réalisée, les changements sur la base publiée sont envoyés à la base abonnée en temps réel. La base abonnée applique les modifications dans le même ordre qu'elles auront été réalisées de façon à ce que la cohérence transactionnelle soit garantie pour les publications d'un seul abonnement. Cette méthode de réplication porte parfois le nom de réplication transactionnelle.

Les cas typiques d'utilisation de la réplication logique peuvent être les suivants :

- Envoyer immédiatement les changements réalisés sur une base de données, ou sur un sous-ensemble de ces données, de façon incrémentale à une base de données abonnée;
- Déclencher des triggers pour des changements spécifiques lorsqu'ils apparaissent sur la base de données abonnée ;
- Réaliser la consolidation de plusieurs bases de données au sein d'une seule (par exemple pour répondre à des problématiques analytiques) ;
- Répliquer entre des versions majeures différentes de PostgreSQL ;
- Répliquer des instances PostgreSQL sur des plateformes différentes (par exemple de Linux à Windows) ;
- Donner accès à des données répliquées à différents groupes d'utilisateurs ;
- Partager un sous-ensemble de données entre plusieurs bases de données.

Une base de données abonnée se comporte comme n'importe quelle autre base de données d'une instance PostgreSQL et peut être utilisée comme base de données de publication pour d'autres base de données en lui définissant ses propres publications. Lorsque la base abonnée est considérée comme une base en lecture seule par l'application, il ne va pas y avoir de problèmes de conflits. D'un autre côté, s'il y a des écritures provenant soit de l'application soit d'un autre abonnement sur le même ensemble de tables, des conflits peuvent survenir.

31.1. Publication

Une *publication* peut être définie sur n'importe quel serveur primaire de réplication physique. Le nœud sur laquelle la publication est définie est nommé *publieur*. Une publication est un ensemble de modifications générées par une table ou un groupe de tables et peut aussi être défini comme un ensemble de modifications ou un ensemble de réplication. Chaque publication existe au sein d'une seule base de données.

Les publications sont différenciées du schéma et n'ont pas d'impact sur la manière dont la base est accédée. Chaque table peut être ajoutée à différentes publications au besoin. Actuellement, les

publications ne contiennent que les tables et toutes les tables d'un schéma. Les objets doivent être ajoutés explicitement, sauf si la publication a été créée pour toutes les tables (`ALL TABLES`).

Les publications peuvent choisir de limiter les changements qu'elles produisent avec n'importe quelle combinaison de `INSERT`, `UPDATE`, `DELETE` et `TRUNCATE`, ceci d'une façon similaire à l'activation de triggers en fonction d'un certain type d'événement. Par défaut, tous les types d'opération sont répliqués. Ces spécifications de publication s'appliquent seulement pour les opérations DML ; elles n'affectent pas la copie initiale de synchronisation des données. (Les filtres de ligne n'ont pas d'effet pour la commande `TRUNCATE`. Voir Section 31.3.)

Une table publiée doit avoir une « identité de réplication » configurée pour être capable de répliquer des opérations `UPDATE` et `DELETE`, pour que les lignes appropriées à modifier ou supprimer puissent être identifiées du côté de l'abonné. Par défaut, il s'agit de la clé primaire, si elle existe. Un autre index d'unicité (avec quelques prérequis supplémentaires) peut aussi être configuré du côté de l'abonné. Si la table n'a pas de clé convenable, alors elle peut être configurée pour l'identité de réplicat « full », ce qui signifie que la ligne entière devient la clé. Néanmoins, ceci est très inefficace et devrait seulement être utilisé si aucune autre solution n'est disponible. Si une identité de réplication est différente de « full » du côté du publieur, une identité de réplication comprenant les mêmes colonnes, ou moins de colonnes, peut aussi être configuré du côté de l'abonné. Voir `REPLICA IDENTITY` pour les détails sur la configuration de l'identité de réplication. Si une table sans identité de réplication est ajoutée à une publication qui réplique les opérations `UPDATE` ou `DELETE`, alors les opérations `UPDATE` ou `DELETE` suivantes causeront une erreur sur le publieur. Les opérations `INSERT` peuvent se réaliser quelle que soit l'identité de réplication.

Chaque publication peut avoir plusieurs abonnés.

Une publication est créée en utilisant la commande `CREATE PUBLICATION` et peut ensuite être modifiée ou supprimée en utilisant la commande correspondante.

Les tables individuelles peuvent être ajoutées ou supprimées dynamiquement en utilisant `ALTER PUBLICATION`. Les opérations `ADD TABLE` et `DROP TABLE` sont toutes les deux transactionnelles ; de ce fait, une table va commencer ou arrêter de répliquer dans le bon instantané seulement une fois que la transaction a été validée.

31.2. Abonnement

Un *abonnement* est le côté aval de la réplication logique. Le nœud où un abonnement a été défini est nommé *abonné*. Un abonnement définit la connexion à une autre base de données et un ensemble de publications (une ou plus) auxquelles l'abonné veut souscrire.

La base de données abonnée se comporte comme n'importe quelle base de données d'une instance PostgreSQL et peut être utilisée comme éditeur pour d'autres bases de données en définissant ses propres publications.

Un nœud abonné peut avoir plusieurs abonnements si besoin. Il est possible de définir plusieurs abonnements entre une même paire publieur - abonné. Dans ce cas, il faut faire attention à ce que les objets des publications auxquelles l'abonné a souscrit ne se chevauchent pas.

Chaque abonnement recevra les changements par un slot de réplication (voir Section 27.2.6). Des slots de réplications supplémentaires peuvent être nécessaires pour la synchronisation initiale des données d'une table contenant des données pré-existantes mais ils seront supprimés à la fin de la synchronisation des données.

Un abonnement de réplication logique peut être réalisé sur un serveur secondaire pour de la réplication synchrone (voir Section 27.2.8). Le nom du serveur secondaire correspond par défaut au nom de l'abonnement. Un nom alternatif peut être indiqué avec le paramètre `application_name` dans les informations de connexion à l'abonnement.

Les abonnements sont sauvegardés par `pg_dump` si l'utilisateur courant a des droits de superutilisateur. Si ce n'est pas le cas, un message d'avertissement est renvoyé et les abonnements ne

sont pas sauvegardés. En effet, les informations d'abonnements contenues dans `pg_subscription` ne sont pas consultables par des utilisateurs dotés de droits moins importants.

Un abonnement est ajouté en utilisant `CREATE SUBSCRIPTION`. Il peut être arrêté/repris à n'importe quel moment en utilisant la commande `ALTER SUBSCRIPTION`, et il peut être supprimé par la commande `DROP SUBSCRIPTION`.

Quand un abonnement est supprimé puis recréé, les informations de synchronisation sont perdues. Cela signifie que les données doivent être resynchronisées ensuite.

La définition d'un schéma n'est pas répliquée, et les tables publiées doivent exister sur la base abonnée. Seules des tables standards peuvent accueillir des données répliquées. Par exemple, il n'est pas pas possible de répliquer dans une vue.

La correspondance entre les tables du publieur et de l'abonné est réalisée en utilisant le nom entièrement qualifié de la table. La réplication entre des tables portant un nom différent sur la base abonnée n'est pas supportée.

La correspondance sur les colonnes d'une table se fait aussi par nom. L'ordre des colonnes dans la table sur le serveur abonné ne correspond pas forcément à l'ordre sur le serveur publieur. Les types de données n'ont pas non plus besoin de correspondre, à partir du moment où la représentation textuelle de la donnée peut être convertie vers le type de données cible. Par exemple, vous pouvez répliquer depuis une colonne de type `integer` vers une colonne de type `bigint`. La table cible peut aussi avoir des colonnes supplémentaires non fournies par la table publiée. Ce type de colonne sera rempli avec la valeur par défaut fournie dans la définition de la table cible.

31.2.1. Gestion des slots de réplication

Comme présenté plus tôt, chaque abonnement (actif) reçoit les changements depuis un slot de réplication du serveur distant (publication).

Des slots de synchronisation de tables supplémentaires sont normalement temporaires, créés en interne pour réaliser la synchronisation initiale des tables et supprimés automatiquement quand elles ne sont plus nécessaires. Ces slots de synchronisation de table ont des noms générés automatiquement : « `pg_%u_sync_%u_%llu` » (paramètres *`oid`* de la souscription, *`relid`* de la table, *`sysid`* pour l'identifiant du système).

Normalement, le slot de réplication distant est créé automatiquement en utilisant la commande `CREATE SUBSCRIPTION` et il est supprimé automatiquement en utilisant la commande `DROP SUBSCRIPTION`. Dans certaines situations, il peut être utile ou nécessaire de manipuler les abonnements ainsi que les slots de réplication sous-jacents de façon séparées. Voici quelques exemples :

- Lorsqu'en créant un abonnement, le slot de réplication correspondant existe déjà. Dans ce cas, l'abonnement peut être créé en utilisant l'option `create_slot = false` pour réaliser l'association avec le slot existant ;

- Lorsqu'en créant un abonnement, le serveur distant n'est pas disponible ou dans un état indéfini. Dans ce cas, l'abonnement peut être créé en utilisant l'option `connect = false`. Le serveur distant ne sera alors jamais contacté. C'est la méthode utilisée par pg_dump. Le slot de réplication distant devra alors être créé manuellement avant que l'abonnement ne puisse être activé ;

- Lorsqu'on supprime un abonnement et que le slot de réplication doit être conservé, par exemple lorsqu'une base abonnée est déplacée vers un serveur différent et sera activée depuis cette nouvelle localisation. Dans ce cas, il faut dissocier le slot de réplication de l'abonnement correspondant en utilisant la commande `ALTER SUBSCRIPTION` avant de supprimer l'abonnement ;

- Lorsque l'on supprime un abonnement et que le serveur distant n'est pas joignable. Dans ce cas, il faut aussi dissocier le slot de réplication de l'abonnement correspondant en utilisant

`ALTER SUBSCRIPTION` avant de supprimer l'abonnement. Si l'instance distante n'existe plus, aucune action supplémentaire n'est nécessaire. Si, par contre, l'instance distante est simplement temporairement injoignable, le slot de réplication (et tout slot de synchronisation de table restant) devrait être supprimé manuellement, sinon l'instance va persévérer à conserver ses fichiers WAL jusqu'à saturation de l'espace disque disponible. Ces cas doivent être traités avec beaucoup de précautions.

31.2.2. Exemples

Créer des tables tests sur le publieur.

```
test_pub=# CREATE TABLE t1(a int, b text, PRIMARY KEY(a));
CREATE TABLE
test_pub=# CREATE TABLE t2(c int, d text, PRIMARY KEY(c));
CREATE TABLE
test_pub=# CREATE TABLE t3(e int, f text, PRIMARY KEY(e));
CREATE TABLE
```

Créer les mêmes tables sur l'abonné.

```
test_sub=# CREATE TABLE t1(a int, b text, PRIMARY KEY(a));
CREATE TABLE
test_sub=# CREATE TABLE t2(c int, d text, PRIMARY KEY(c));
CREATE TABLE
test_sub=# CREATE TABLE t3(e int, f text, PRIMARY KEY(e));
CREATE TABLE
```

Insérer des données sur les tables du côté publieur.

```
test_pub=# INSERT INTO t1 VALUES (1, 'one'), (2, 'two'), (3,
 'three');
INSERT 0 3
test_pub=# INSERT INTO t2 VALUES (1, 'A'), (2, 'B'), (3, 'C');
INSERT 0 3
test_pub=# INSERT INTO t3 VALUES (1, 'i'), (2, 'ii'), (3, 'iii');
INSERT 0 3
```

Créer les publications pour les tables. Les publications `pub2` et `pub3a` interdisent certaines opérations `publish`. La publication `pub3b` a un filtre de lignes (voir Section 31.3).

```
test_pub=# CREATE PUBLICATION pub1 FOR TABLE t1;
CREATE PUBLICATION
test_pub=# CREATE PUBLICATION pub2 FOR TABLE t2 WITH (publish =
 'truncate');
CREATE PUBLICATION
test_pub=# CREATE PUBLICATION pub3a FOR TABLE t3 WITH (publish =
 'truncate');
CREATE PUBLICATION
test_pub=# CREATE PUBLICATION pub3b FOR TABLE t3 WHERE (e > 5);
CREATE PUBLICATION
```

Créer les abonnements (souscriptions) pour les publications. La souscription `sub3` s'abonne à `pub3a` et `pub3b`. Toutes les souscriptions copieront les données initiales par défaut.

```
test_sub=# CREATE SUBSCRIPTION sub1
test_sub-# CONNECTION 'host=localhost dbname=test_pub
 application_name=sub1'
test_sub-# PUBLICATION pub1;
CREATE SUBSCRIPTION
test_sub=# CREATE SUBSCRIPTION sub2
test_sub-# CONNECTION 'host=localhost dbname=test_pub
 application_name=sub2'
test_sub-# PUBLICATION pub2;
CREATE SUBSCRIPTION
test_sub=# CREATE SUBSCRIPTION sub3
test_sub-# CONNECTION 'host=localhost dbname=test_pub
 application_name=sub3'
test_sub-# PUBLICATION pub3a, pub3b;
CREATE SUBSCRIPTION
```

Observez que les données initiales des tables sont copiées, quelque soit l'opération `publish` de la publication.

```
test_sub=# SELECT * FROM t1;
 a |   b
---+-------
 1 | one
 2 | two
 3 | three
(3 rows)

test_sub=# SELECT * FROM t2;
 c | d
---+---
 1 | A
 2 | B
 3 | C
(3 rows)
```

De plus, comme la copie initiale de données ignore l'opération `publish` et comme la publication `pub3a` n'a pas de filtre de lignes, cela signifie que la table `t3` copiée contient toutes les lignes même quand elles ne correspondent pas au filtre de lignes de la publication `pub3b`.

```
test_sub=# SELECT * FROM t3;
 e |  f
---+-----
 1 | i
 2 | ii
 3 | iii
(3 rows)
```

Insérer plus de données dans les tables du côté publieur.

```
test_pub=# INSERT INTO t1 VALUES (4, 'four'), (5, 'five'), (6,
 'six');
INSERT 0 3
test_pub=# INSERT INTO t2 VALUES (4, 'D'), (5, 'E'), (6, 'F');
INSERT 0 3
test_pub=# INSERT INTO t3 VALUES (4, 'iv'), (5, 'v'), (6, 'vi');
INSERT 0 3
```

Maintenant, les données du côté publieur ressemblent à ceci :

```
test_pub=# SELECT * FROM t1;
 a |   b
---+-------
 1 | one
 2 | two
 3 | three
 4 | four
 5 | five
 6 | six
(6 rows)

test_pub=# SELECT * FROM t2;
 c | d
---+---
 1 | A
 2 | B
 3 | C
 4 | D
 5 | E
 6 | F
(6 rows)

test_pub=# SELECT * FROM t3;
 e |  f
---+-----
 1 | i
 2 | ii
 3 | iii
 4 | iv
 5 | v
 6 | vi
(6 rows)
```

Observez que, durant la réplication normale, les opérations `publish` appropriées sont utilisées. Cela signifie que les publications `pub2` et `pub3a` ne répliquent pas les opérations `INSERT`. De plus, la publication `pub3b` répliquera seulement les données correspondant au filtre de lignes de `pub3b`. Maintenant, les données du côté abonné ressemblent à ceci :

```
test_sub=# SELECT * FROM t1;
 a |   b
---+-------
 1 | one
 2 | two
 3 | three
 4 | four
 5 | five
 6 | six
(6 rows)

test_sub=# SELECT * FROM t2;
 c | d
---+---
 1 | A
 2 | B
 3 | C
```

```
(3 rows)

test_sub=# SELECT * FROM t3;
 e |  f
---+-----
 1 | i
 2 | ii
 3 | iii
 6 | vi
(4 rows)
```

31.3. Filtres de lignes

Par défaut, toutes les données de toutes les tables publiées seront répliquées aux abonnés appropriés. Les données répliquées peuvent être réduites en utilisant un *filtre de lignes*. Un utilisateur pourrait choisir d'utiliser des filtres de lignes pour des raisons de comportement, de sécurité ou de performance. Si une table publiée configure un filtre de lignes, une ligne est répliquée seulement si ses données satisfont l'expression du filtre de lignes. Cela permet à un ensemble de tables d'être répliquées partiellement. Le filtre de lignes est défini par table. Utilisez une clause `WHERE` après le nom de la table pour chaque table publiée dont les données doivent être filtrées. La clause `WHERE` doit être entourée de parenthèses. Voir CREATE PUBLICATION pour les détails.

31.3.1. Règles des filtres de lignes

Les filtres de lignes sont appliqués *avant* de publier les changements. Si le filtre de ligne est évalué à `false` ou `NULL`, alors la ligne n'est pas répliquée. L'expression de la clause `WHERE` est évaluée avec le même rôle utilisé pour la connexion de réplication (soit le rôle indiqué dans la clause `CONNECTION` de l'instruction CREATE SUBSCRIPTION). Les filtres de ligne n'ont pas deffet sur la commande `TRUNCATE`.

31.3.2. Restrictions de l'expression

La clause `WHERE` autorise uniquement des expressions simples. Elle ne peut pas contenir de fonctions, opérateurs, types et collations définis par les utilisateurs, des références aux colonnes système ou à des fonctions internes non immutables.

Si une publication publie des opérations `UPDATE` ou `DELETE`, la clause `WHERE` du filtre de lignes doit contenir uniquement les colonnes couvertes par l'identité de réplica (voir `REPLICA IDENTITY`). Si une publication publie uniquement des opérations `INSERT`, la clause `WHERE` du filtre de ligne peut utiliser toute colonne.

31.3.3. Transformations d'un UPDATE

À chaque fois qu'un `UPDATE` est traité, l'expression du filtre de lignes est évaluée pour l'ancienne et la nouvelle ligne (autrement dit, en utilisant les données avant et après la mise à jour). Si les deux évaluations valent `true`, les modifications de l'`UPDATE` sont répliquées. Si les deux évaluations valent `false`, les modifications ne sont pas répliquées. Si seulement une des deux évaluations vaut true, la commande `UPDATE` est transformée en `INSERT` ou `DELETE`, pour éviter toute incohérence de données. La ligne sur l'abonné doit refléter ce qui est défini par l'expression de filtre de lignes sur le publieur.

Si l'ancienne ligne satisfait l'expression du filtre de lignes (elle a été envoyée sur l'abonné) contrairement à la nouvelle ligne, alors, d'un point de vue de la cohérence des données, l'ancienne ligne devrait être supprimée de l'abonné. Donc la commande `UPDATE` est transformée en commande `DELETE`.

Si l'ancienne ligne ne satisfait pas l'expression du filtre de lignes (elle n'a pas été envoyée sur l'abonné) contrairement à la nouvelle ligne, alors, d'un point de vue de la cohérence des données, la nouvelle ligne doit être ajoutée à l'abonné. Donc la commande `UPDATE` est transformée en commande `INSERT`.

Tableau 31.1 résume les transformations appliquées.

Tableau 31.1. Résumé des transformations d'un `UPDATE`

Ancienne ligne	Nouvelle ligne	Transformation
pas de correspondance	pas de correspondance	ne réplique pas
pas de correspondance	correspondance	`INSERT`
correspondance	pas de correspondance	`DELETE`
correspondance	correspondance	`UPDATE`

31.3.4. Tables partitionnées

Si la publication contient une table partitionnée, le paramètre de publication `publish_via_partition_root` détermine le filtre de lignes à utiliser. Si `publish_via_partition_root` vaut `true`, le filtre de ligne de la *table partitionnée racine* est utilisé. Si `publish_via_partition_root` vaut `false` (ce qui est le cas par défaut), le filtre de lignes de chaque *partition* est utilisé.

31.3.5. Synchronisation initiale des données

Si la souscription requiert une copie des données pré-existantes de la table et qu'une publication contient des clauses `WHERE`, seules les données qui satisfont les expressions des filtres de ligne sont copiées sur l'abonné.

Si la souscription a plusieurs publications dans lesquelles une table a été publiée avec différentes clauses `WHERE`, les lignes qui satisfont *une* des expressions seront copiées. Voir Section 31.3.6 pour les détails.

> **Avertissement**
>
> Comme la synchronisation initiale des données ne prend pas en compte le paramètre `publish` lors de la copie des données pré-existantes de la table, certaines lignes pourraient être copiées alors qu'elles n'auraient pas été répliquées avec des instructions DML. Référez-vous à Section 31.7.1, et à Section 31.2.2 pour des exemples.

> **Note**
>
> Si l'abonné est d'une version antérieure à la 15, la copie des données pré-existantes n'utilise pas les filtres de ligne même s'ils sont définis dans la publication. Ceci est dû au fait que les anciennes versions peuvent seulement copier les données de la table entière.

31.3.6. Combiner plusieurs filtres de lignes

Si la souscription a plusieurs publications pour lesquelles la même table a été publiée avec des filtres de lignes différents (pour la même opération `publish`, ces expressions sont assemblées avec un OU logique, pour que les lignes satisfaisant au moins *une* expression seront répliquées. Cela signifie que les autres filtres de lignes pour la même table deviennent redondants si :

- Une des publications n'a pas de filtres de lignes ;
- Une des publications a été créée en utilisant `FOR ALL TABLES`. Cette clause n'autorise pas les filtres de lignes ;
- Une des publications a été créée en utilisant `FOR ALL TABLES IN SCHEMA` et la table appartient au schéma référencé. Cette clause n'autorise pas les filtres de lignes.

31.3.7. Exemples

Créer quelques tables à utiliser dans les exemples suivants.

```
test_pub=# CREATE TABLE t1(a int, b int, c text, PRIMARY KEY(a,c));
CREATE TABLE
test_pub=# CREATE TABLE t2(d int, e int, f int, PRIMARY KEY(d));
CREATE TABLE
test_pub=# CREATE TABLE t3(g int, h int, i int, PRIMARY KEY(g));
CREATE TABLE
```

Créer quelques publications. La publication `p1` a une table (`t1`) et cette table a un filtre de lignes. La publication `p2` a deux tables. La table `t1` n'a pas de filtre de lignes, et la table `t2` a un filtre de lignes. La publication `p3` a deux tables, qui ont toutes les deux un filtre de lignes.

```
test_pub=# CREATE PUBLICATION p1 FOR TABLE t1 WHERE (a > 5 AND c =
 'NSW');
CREATE PUBLICATION
test_pub=# CREATE PUBLICATION p2 FOR TABLE t1, t2 WHERE (e = 99);
CREATE PUBLICATION
test_pub=# CREATE PUBLICATION p3 FOR TABLE t2 WHERE (d = 10), t3
 WHERE (g = 10);
CREATE PUBLICATION
```

`psql` peut être utilisé pour afficher les expressions de filtres de lignes (si défini) pour chaque publication.

```
test_pub=# \dRp+
                               Publication p1
  Owner   | All tables | Inserts | Updates | Deletes | Truncates |
 Via root
----------+------------+---------+---------+---------+-----------
+----------
 postgres | f          | t       | t       | t       | t         |
 f
Tables:
    "public.t1" WHERE ((a > 5) AND (c = 'NSW'::text))

                               Publication p2
  Owner   | All tables | Inserts | Updates | Deletes | Truncates |
 Via root
----------+------------+---------+---------+---------+-----------
+----------
 postgres | f          | t       | t       | t       | t         |
 f
Tables:
    "public.t1"
    "public.t2" WHERE (e = 99)
```

```
                                Publication p3
  Owner   | All tables | Inserts | Updates | Deletes | Truncates |
 Via root
----------+------------+---------+---------+---------+-----------
+----------
 postgres | f          | t       | t       | t       | t         |
 f
Tables:
    "public.t2" WHERE (d = 10)
    "public.t3" WHERE (g = 10)
```

`psql` peut être utilisé pour afficher les expressions des filtres de lignes (si défini) pour chaque table. La table `t1` est membre de deux publications, mais a un filtre de lignes uniquement dans `p1`. La table `t2` est membre de deux publications, et a un filtre de lignes différent sur chacune.

```
test_pub=# \d t1
                Table "public.t1"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 a      | integer |           | not null |
 b      | integer |           |          |
 c      | text    |           | not null |
Indexes:
    "t1_pkey" PRIMARY KEY, btree (a, c)
Publications:
    "p1" WHERE ((a > 5) AND (c = 'NSW'::text))
    "p2"

test_pub=# \d t2
                Table "public.t2"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 d      | integer |           | not null |
 e      | integer |           |          |
 f      | integer |           |          |
Indexes:
    "t2_pkey" PRIMARY KEY, btree (d)
Publications:
    "p2" WHERE (e = 99)
    "p3" WHERE (d = 10)

test_pub=# \d t3
                Table "public.t3"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 g      | integer |           | not null |
 h      | integer |           |          |
 i      | integer |           |          |
Indexes:
    "t3_pkey" PRIMARY KEY, btree (g)
Publications:
    "p3" WHERE (g = 10)
```

Sur le nœud abonné, créer une table `t1` avec la même définition que celle du publieur, et créer aussi la souscription `s1` vers la publication `p1`.

```
test_sub=# CREATE TABLE t1(a int, b int, c text, PRIMARY KEY(a,c));
CREATE TABLE
test_sub=# CREATE SUBSCRIPTION s1
test_sub-# CONNECTION 'host=localhost dbname=test_pub
 application_name=s1'
test_sub-# PUBLICATION p1;
CREATE SUBSCRIPTION
```

Insérer quelques lignes. Seules les lignes satisfaisant la clause `t1 WHERE` de la publication `p1` sont répliquées.

```
test_pub=# INSERT INTO t1 VALUES (2, 102, 'NSW');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (3, 103, 'QLD');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (4, 104, 'VIC');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (5, 105, 'ACT');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (6, 106, 'NSW');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (7, 107, 'NT');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (8, 108, 'QLD');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES (9, 109, 'NSW');
INSERT 0 1

test_pub=# SELECT * FROM t1;
 a |  b  |  c
---+-----+-----
 2 | 102 | NSW
 3 | 103 | QLD
 4 | 104 | VIC
 5 | 105 | ACT
 6 | 106 | NSW
 7 | 107 | NT
 8 | 108 | QLD
 9 | 109 | NSW
(8 rows)


test_sub=# SELECT * FROM t1;
 a |  b  |  c
---+-----+-----
 6 | 106 | NSW
 9 | 109 | NSW
(2 rows)
```

Mettre à jour quelques données, où les valeurs de l'ancienne et de la nouvelle ligne satisfont la clause `t1 WHERE` de la publication `p1`. La commande `UPDATE` réplique les changements de façon normale.

```
test_pub=# UPDATE t1 SET b = 999 WHERE a = 6;
UPDATE 1

test_pub=# SELECT * FROM t1;
 a |  b  |  c
```

```
---+-----+-----
 2 | 102 | NSW
 3 | 103 | QLD
 4 | 104 | VIC
 5 | 105 | ACT
 7 | 107 | NT
 8 | 108 | QLD
 9 | 109 | NSW
 6 | 999 | NSW
(8 rows)


test_sub=# SELECT * FROM t1;
 a |  b  |  c
---+-----+-----
 9 | 109 | NSW
 6 | 999 | NSW
(2 rows)
```

Mettre à jour des données de façon à ce que les valeurs de l'ancienne ligne ne satisfont pas la clause `t1 WHERE` de la publication `p1`, mais que les valeurs de la nouvelle ligne la satisfont. La commande `UPDATE` est transformée en une commande `INSERT` et les changements sont répliqués. Voir la nouvelle ligne sur l'abonné.

```
test_pub=# UPDATE t1 SET a = 555 WHERE a = 2;
UPDATE 1

test_pub=# SELECT * FROM t1;
  a  |  b  |  c
-----+-----+-----
   3 | 103 | QLD
   4 | 104 | VIC
   5 | 105 | ACT
   7 | 107 | NT
   8 | 108 | QLD
   9 | 109 | NSW
   6 | 999 | NSW
 555 | 102 | NSW
(8 rows)


test_sub=# SELECT * FROM t1;
  a  |  b  |  c
-----+-----+-----
   9 | 109 | NSW
   6 | 999 | NSW
 555 | 102 | NSW
(3 rows)
```

Mettre à jour des données de façon à ce que les valeurs de l'ancienne ligne satisfont la clause `t1 WHERE` de la publication `p1`, mais que les valeurs de la nouvelle ligne ne la satisfont pas. La commande `UPDATE` est transformée en une commande `DELETE` et les changements sont répliqués. Voir la nouvelle ligne supprimée sur l'abonné.

```
test_pub=# UPDATE t1 SET c = 'VIC' WHERE a = 9;
UPDATE 1
```

```
test_pub=# SELECT * FROM t1;
  a  |  b  |  c
-----+-----+-----
   3 | 103 | QLD
   4 | 104 | VIC
   5 | 105 | ACT
   7 | 107 | NT
   8 | 108 | QLD
   6 | 999 | NSW
 555 | 102 | NSW
   9 | 109 | VIC
(8 rows)
```

```
test_sub=# SELECT * FROM t1;
  a  |  b  |  c
-----+-----+-----
   6 | 999 | NSW
 555 | 102 | NSW
(2 rows)
```

Les exemples suivants montrent comment le paramètre de publication `publish_via_partition_root` détermine si le filtre de ligne de la table parent ou enfant sera utilisé dans le cas de tables partitionnées.

Créer une table partitionnée sur le publieur.

```
test_pub=# CREATE TABLE parent(a int PRIMARY KEY) PARTITION BY
 RANGE(a);
CREATE TABLE
test_pub=# CREATE TABLE child PARTITION OF parent DEFAULT;
CREATE TABLE
```

Créer les mêmes tables sur l'abonné.

```
test_sub=# CREATE TABLE parent(a int PRIMARY KEY) PARTITION BY
 RANGE(a);
CREATE TABLE
test_sub=# CREATE TABLE child PARTITION OF parent DEFAULT;
CREATE TABLE
```

Créer une publication `p4`, puis abonnez-vous à cette publication. Le paramètre de publication `publish_via_partition_root` est configuré à true. Il existe des filtres de lignes définis sur la table partitionnée (`parent`), et sur sa partition (`child`).

```
test_pub=# CREATE PUBLICATION p4 FOR TABLE parent WHERE (a < 5),
 child WHERE (a >= 5)
test_pub-# WITH (publish_via_partition_root=true);
CREATE PUBLICATION
```

```
test_sub=# CREATE SUBSCRIPTION s4
test_sub-# CONNECTION 'host=localhost dbname=test_pub
 application_name=s4'
test_sub-# PUBLICATION p4;
CREATE SUBSCRIPTION
```

Insérer quelques valeurs directement dans les tables `parent` et `child`. Elles se répliquent en utilisant le filtre de lignes de `parent` (parce que `publish_via_partition_root` vaut true).

```
test_pub=# INSERT INTO parent VALUES (2), (4), (6);
INSERT 0 3
test_pub=# INSERT INTO child VALUES (3), (5), (7);
INSERT 0 3

test_pub=# SELECT * FROM parent ORDER BY a;
 a
---
 2
 3
 4
 5
 6
 7
(6 rows)


test_sub=# SELECT * FROM parent ORDER BY a;
 a
---
 2
 3
 4
(3 rows)
```

Répéter le même test, mais avec une valeur différent pour `publish_via_partition_root`. Le paramètre de publication `publish_via_partition_root` est configuré à false. Un filtre de lignes est défini sur la partition (`child`).

```
test_pub=# DROP PUBLICATION p4;
DROP PUBLICATION
test_pub=# CREATE PUBLICATION p4 FOR TABLE parent, child WHERE (a
 >= 5)
test_pub-# WITH (publish_via_partition_root=false);
CREATE PUBLICATION


test_sub=# ALTER SUBSCRIPTION s4 REFRESH PUBLICATION;
ALTER SUBSCRIPTION
```

Faire les insertions sur le publieur de la même façon qu'avant. Elles se répliquent en utilisant le filtre de lignes de `child` (parce que `publish_via_partition_root` vaut false).

```
test_pub=# TRUNCATE parent;
TRUNCATE TABLE
test_pub=# INSERT INTO parent VALUES (2), (4), (6);
INSERT 0 3
test_pub=# INSERT INTO child VALUES (3), (5), (7);
INSERT 0 3

test_pub=# SELECT * FROM parent ORDER BY a;
 a
---
```

```
 2
 3
 4
 5
 6
 7
(6 rows)

test_sub=# SELECT * FROM child ORDER BY a;
 a
---
 5
 6
 7
(3 rows)
```

31.4. Listes de colonnes

Chaque publication peut indiquer en option aux abonnés les colonnes à répliquer pour chaque table. La table du côté abonné doit avoir au moins toutes les colonnes publiées. Si aucune liste de colonnes n'est indiquée, alors toutes les colonnes du côté du publieur sont répliquées. Voir CREATE PUBLICATION pour les détails sur la syntaxe.

Le choix des colonnes peut être basé sur des raisons de comportement ou de performance. Néanmoins, ne vous basez pas sur cette fonctionnalité pour de la sécurité : un abonné mal intentionné est capable d'obtenir des données des colonnes qui ne sont pas spécifiquement publiées. D'un point de vue sécurité, des protections peuvent être mises en place du côté du publieur.

Si aucune liste de colonne n'est indiquée, toutes les colonnes ajoutées ultérieurement à la table sont automatiquement répliquées. Cela signifie qu'avoir une liste de colonnes qui nomme toutes les colonnes n'aboutit pas au même comportement que ne pas avoir de liste de colonnes.

Une liste de colonnes peut contenir seulement des références de colonnes. L'ordre des colonnes dans la liste n'est pas préservé.

Préciser une liste de colonnes n'est pas possible quand la publication publie aussi toutes les tables d'un schéma (clause `FOR TABLES IN SCHEMA`).

Pour les tables partitionnées, le paramètre de publication `publish_via_partition_root` détermine quel liste de colonnes utiliser. Si `publish_via_partition_root` vaut `true`, la liste de colonnes utilisée est celle de la table partitionnée. Sinon, si `publish_via_partition_root` vaut `false` (valeur par défaut), la liste de colonnes utilisée est celle de la partition concernée.

Si une publication publie des opérations `UPDATE` ou `DELETE`, toute liste de colonnes doit inclure les colonnes d'identité de réplica de la table (voir `REPLICA IDENTITY`). Si une publication publie seulement des opérations `INSERT`, alors la liste de colonnes peut omettre les colonnes d'identité de réplica.

Les listes de colonnes n'ont pas d'effet sur la commande `TRUNCATE`.

Lors de la synchronisation initiale des données, seules les colonnes publiées sont copiées. Néanmoins, si l'abonné est d'une version antérieure à la 15, alors toutes les colonnes dans la table sont copiées lors de la synchronisation initiale des données, ignorant en cela les liste de colonnes.

Attention : Combiner des listes de plusieurs publications

Actuellement, il n'est pas possible qu'une souscription soit entreprise auprès de plusieurs publications quand la même table a été publiée avec des listes de colonnes différentes.

CREATE SUBSCRIPTION interdit la création de telles souscriptions mais il est toujours possible d'arriver dans cette situation par l'ajout ou la modification de listes de colonnes du côté publication une fois que la souscription a été créée.

Ceci signifie que la modification de liste de colonnes sur les publications déjà souscrites peut amener à des erreurs du côté souscripteur.

Si ce problème affecte une souscription, la seule façon de reprendre la réplication est d'ajuster une des listes de colonnes côté publication pour que les listes correspondent ; puis soit de créer de nouveau la souscription, soit utiliser `ALTER SUBSCRIPTION ... DROP PUBLICATION` pour supprimer une des publications problématiques et l'ajouter de nouveau après.

31.4.1. Exemples

Créer une table `t1` à utiliser dans l'exemple suivant.

```
test_pub=# CREATE TABLE t1(id int, a text, b text, c text, d text,
 e text, PRIMARY KEY(id));
CREATE TABLE
```

Créer une publication `p1`. Une liste de colonnes est définie pour la table `t1` pour réduire le nombre de colonnes qui seront répliquées. Notez que l'ordre des noms de colonnes dans la liste de colonnes n'a pas d'importance.

```
test_pub=# CREATE PUBLICATION p1 FOR TABLE t1 (id, b, a, d);
CREATE PUBLICATION
```

`psql` peut être utilisé pour afficher les listes de colonnes (si définies) pour chaque publication.

```
test_pub=# \dRp+
                                Publication p1
  Owner   | All tables | Inserts | Updates | Deletes | Truncates |
 Via root
----------+------------+---------+---------+---------+-----------
+----------
 postgres | f          | t       | t       | t       | t         |
 f
Tables:
    "public.t1" (id, a, b, d)
```

`psql` peut être utilisé pour afficher les listes de colonnes (si définies) pour chaque table.

```
test_pub=# \d t1
                Table "public.t1"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | integer |           | not null |
 a      | text    |           |          |
 b      | text    |           |          |
 c      | text    |           |          |
 d      | text    |           |          |
 e      | text    |           |          |
Indexes:
```

```
    "t1_pkey" PRIMARY KEY, btree (id)
Publications:
    "p1" (id, a, b, d)
```

Sur le nœud abonné, créer une table `t1` qui maintenant a seulement besoin d'un sous-ensemble des colonnes qui sont sur la table côté publieur `t1`, et crée aussi l'abonnement `s1` qui cible la publication `p1`.

```
test_sub=# CREATE TABLE t1(id int, b text, a text, d text, PRIMARY
 KEY(id));
CREATE TABLE
test_sub=# CREATE SUBSCRIPTION s1
test_sub-# CONNECTION 'host=localhost dbname=test_pub
 application_name=s1'
test_sub-# PUBLICATION p1;
CREATE SUBSCRIPTION
```

Sur le nœud publieur, insérer quelques lignes dans la table `t1`.

```
test_pub=# INSERT INTO t1 VALUES(1, 'a-1', 'b-1', 'c-1', 'd-1',
 'e-1');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES(2, 'a-2', 'b-2', 'c-2', 'd-2',
 'e-2');
INSERT 0 1
test_pub=# INSERT INTO t1 VALUES(3, 'a-3', 'b-3', 'c-3', 'd-3',
 'e-3');
INSERT 0 1
test_pub=# SELECT * FROM t1 ORDER BY id;
 id |  a  |  b  |  c  |  d  |  e
----+-----+-----+-----+-----+-----
  1 | a-1 | b-1 | c-1 | d-1 | e-1
  2 | a-2 | b-2 | c-2 | d-2 | e-2
  3 | a-3 | b-3 | c-3 | d-3 | e-3
(3 rows)
```

Seules les données de la liste de colonnes de la publication `p1` sont répliquées.

```
test_sub=# SELECT * FROM t1 ORDER BY id;
 id |  b  |  a  |  d
----+-----+-----+-----
  1 | b-1 | a-1 | d-1
  2 | b-2 | a-2 | d-2
  3 | b-3 | a-3 | d-3
(3 rows)
```

31.5. Conflits

La réplication logique se comporte de la même manière pour les opérations DML dans le sens où les données seront mises à jour même si la modification a été faite en local sur la base abonnée. Si les données entrantes entrainent des violations de contrainte d'intégrité, la réplication s'arrête. Cela sera référencé comme un *conflit*. Lorsque l'on réplique des opérations `UPDATE` ou `DELETE`, les données manquantes ne produiront pas de conflit et des opérations de la sorte seront simplement évitées.

Les opérations de réplication logique sont réalisées avec les droits du propriétaire de la souscription. Les échecs de droit sur les tables cibles causeront des conflits de réplication, tout autant que l'activation

de politiques de sécurité au niveau ligne sur des tables cibles et pour lesquels le propriétaire de la souscription est sujet des politiques de sécurité, sans chercher si une politique rejetterait habituellement les opérations INSERT, UPDATE, DELETE ou TRUNCATE en cours de réplication. Cette restriction sur les politiques de sécurité niveau ligne pourrait être supprimée dans une prochaine version de PostgreSQL.

Lorsqu'un conflit entraine une erreur, cela stoppe la réplication ; Le conflit devra être résolu manuellement par un utilisateur. Des informations détaillées concernant le conflit seront disponibles dans les journaux applicatifs de l'instance abonnée.

La résolution peut être réalisée, soit en changeant les données ou les droits sur la base abonnée pour qu'elles ne soient plus en conflit avec les données entrantes ou en évitant les transactions qui sont en conflit avec les données existantes. Quand un conflit produit une erreur, la réplication ne peut pas continuer et le processus worker de la réplication logique émet un message du type suivant dans les journaux applicatifs de l'abonné :

```
ERROR:  duplicate key value violates unique constraint "test_pkey"
DETAIL:  Key (c)=(1) already exists.
CONTEXT:  processing remote data for replication origin "pg_16395"
 during "INSERT" for replication target relation "public.test" in
 transaction 725 finished at 0/14C0378
```

Le LSN de la transaction qui contient le changement violant la contrainte et le nom d'origine de réplication peuvent être trouvés à partir du journal applicatif du serveur (LSN 0/14C0378 et origine de réplication pg_16395 dans le cas ci-dessus). La transaction qui a produit le conflit peut être ignorée en utilisant l'instruction ALTER SUBSCRIPTION ... SKIP avec le LSN final (LSN 0/14C0378). Le LSN final pourrait être un LSN sur lequel la transaction est validée ou préparée sur le publieur. La transaction peut aussi être ignorée en appelant la fonction pg_replication_origin_advance(). Avant d'utiliser cette fonction, la souscription doit être désactivée temporairement soit par ALTER SUBSCRIPTION ... DISABLE soit en utilisant l'option disable_on_error de la souscription. Ensuite, vous pouvez utiliser la fonction pg_replication_origin_advance() avec *node_name* (pg_16395) et le LSN suivant du LSN final (0/14C0379). La position actuelle des origines peut être trouvée dans la vue système pg_replication_origin_status. Merci de noter qu'ignorer la transaction complète inclut d'ignorer des changements qui pourraient ne pas violer des contraintes. Ceci peut rendre l'abonné incohérent très facilement.

31.6. Restrictions

La réplication logique souffre actuellement des restrictions ou des fonctionnalités manquantes suivantes. Elles pourraient être adressées dans les prochaines versions.

- La structure de la base de données et les commandes DDL ne sont pas répliquées. Le schéma initial peut être copié à la main en utilisant la commande pg_dump --schema-only. Les modifications de schéma suivantes auront besoin d'être synchronisées manuellement. (Notez, néanmoins, qu'il n'est pas nécessaire que les schémas soient strictement identiques des deux côtés.) La réplication logique est robuste quand il y a des modifications de schéma dans une base de données. Quand le schéma est changé sur le publieur et les données répliquées commencent à arriver sur l'abonné mais ne correspondent pas à la structure de la table, la réplication renverra une erreur jusqu'à ce que le schéma soit mis à jour. Dans de nombreux cas, les erreurs intermittentes peuvent être évitées en appliquant des modifications de schéma à l'abonné en premier ;

- Les données des séquences ne sont pas répliquées. Les données des colonnes de type serial et des colonnes identité, gérées par des séquences, seront bien sûr répliquées comme faisant partie de la table, mais la séquence elle-même affichera toujours la valeur de démarrage sur l'abonné. Si l'abonné est utilisé comme une base de données en lecture seule, alors cela ne devrait pas être un problème. Néanmoins, s'il est nécessaire de faire un bascule (*switchover* ou *failover* sur la base de

données abonnée, alors les séquences auront besoin d'être mises à jour à leur dernières valeurs, soit en copiant les données courantes du publieur (peut-être en utilisant `pg_dump`), soit en déterminant une valeur suffisamment haute à partir des données de la table ;

- La réplication des commandes `TRUNCATE` est supportée mais il est nécessaire de prêter attention lors de l'utilisation de cette commande sur des groupes de tables connectés par des clés étrangères. Lors de la réplication d'une action truncate, l'abonné tronquera le même groupe de tables tronquées sur le publieur, qu'elles soient spécifiées explicitement ou implicitement (grâce à la clause `CASCADE`), moins les tables qui ne font pas partie de la souscription. Ceci fonctionnera correctement si toutes les tables affectées font partie de la même souscription. Cependant, si certaines tables à tronquer ont des clés étrangères vers des tables qui ne font pas partie de la même souscription, alors l'application de l'action truncate échouera sur le serveur abonné ;

- Les Large Objects (voir Chapitre 35) ne sont pas répliqués. Il n'y a pas de contournement pour ça, en dehors d'enregistrer les données dans des tables normales ;

- La réplication est seulement supportée par les tables, y compris les tables partitionnées. Toute tentative de répliquer d'autres types de relation, comme les vues, les vues matérialisées ou les tables externes, résultera en une erreur ;

- Lors de la réplication entre tables partitionnées, la réplication actuelle a pour origine, par défaut, les partitions filles sur le publieur, donc les partitions sur le publieur doivent exister aussi sur l'abonné en tant que tables cibles valides. Elles peuvent être soit des partitions filles elles-mêmes, soit de nouveau sous-partitionnées, soit des tables indépendantes. Les publications peuvent aussi spécifier les changements à répliquer en utilisant l'identité et le schéma de la table racine partitionnée au lieu de chaque partition individuelle à l'origine des changements (voir `CREATE PUBLICATION`).

31.7. Architecture

La réplication logique démarre en copiant un instantané des données sur la base de publication. Une fois cette étape réalisée, les modifications sur la base de publication sont envoyées à la base de données abonnée au fil de l'eau. La base abonnée applique les modifications sur les données dans l'ordre dans lequel les validations ont été effectuées sur la base éditeur de manière à ce que la cohérence transactionnelle soit respectée pour les publications vis à vis de tous les abonnements.

La réplication se construit de façon similaire à la réplication physique en flux (*Streaming Replication*, voir Section 27.2.5). Ceci est implémenté par les processus « walsender » et « apply ». Le processus walsender démarre le décodage logique (décrit dans la section Section 55.5) des fichiers WAL et charge le plugin de décodage logique standard (`pgoutput`). Ce plugin transforme les changements lus depuis les fichiers WAL vers le protocole de réplication logique (voir Section 55.5) et filtre les données en fonction des spécificités des publications. Les données sont envoyées au fil de l'eau au processus apply, qui met en relation les données vers les tables locales et applique les changements individuels au moment où ils sont reçus, dans le bon ordre transactionnel.

Le processus apply sur l'instance de la base abonnée fonctionne toujours avec le paramètre `session_replication_role` défini à la valeur `replica`. Ceci signifie que, par défaut, les triggers et règles ne se déclencheront pas sur un abonné. Les utilisateurs peuvent choisir en option d'activer les triggers et les règles sur une table en utilisant la commande `ALTER TABLE` et les clauses `ENABLE TRIGGER` et `ENABLE RULE`.

Le processus apply de la réplication logique déclenche actuellement des triggers de ligne, et non pas des triggers de requêtes. Néanmoins, la synchronisation initiale des tables est implémentée comme une commande `COPY`, ce qui peut déclencher les triggers `INSERT` en mode ligne et requête.

31.7.1. Instantané initial

Les données initiales présentes dans des tables abonnées sont photographiées et copiées dans une instance parallèle qui utilise un type particulier de processus apply. Ce processus va créer son propre

slot de réplication et copier les données existantes. Dès que la copie est terminée, le contenu de la table deviendra visible aux autres processus. Une fois les données existantes copiées, le processus passe en mode de synchronisation, qui assure que la table est amenée vers un état synchronisé avec le processus apply principal, ceci en transférant toutes les modifications survenues pendant la copie initiale des données, réalisée avec le système de réplication logique standard. Lors de cette phase de synchronisation, les changements sont appliqués et validés dans le même ordre que sur le publieur. Une fois la synchronisation terminée, le contrôle de la réplication de la table est rendu au processus apply principal et la réplication continue telle quelle.

> **Note**
>
> Le paramètre de publication `publish` affecte uniquement les opérations DML qui seront répliquées. La synchronisation initiale des données ne prend pas ce paramètre en compte lors de la copie des données existantes de la table.

31.8. Supervision

Puisque la réplication logique est basée sur une architecture similaire à la réplication physique en flux, la supervision d'une instance publication est similaire à la supervision d'une instance primaire dans la réplication physique (voir Section 27.2.5.2).

Les informations des abonnements sont consultables dans la vue `pg_stat_subscription`. Cette vue contient une ligne pour chaque processus d'abonnement. Un abonnement peut avoir zéro ou plusieurs processus abonnés, selon son état.

Normalement il y a un seul processus apply démarré pour un abonnement actif. Un abonnement désactivé ou une publication effondrée n'aura pas de ligne dans cette vue. Si la synchronisation initiale d'une table est en cours, il y aura des processus supplémentaires pour les tables en cours de synchronisation.

31.9. Sécurité

Un utilisateur capable de modifier le schéma des tables côté souscription peut exécuter un code arbitraire en tant que le rôle propriétaire de la souscription qui modifie ces tables. Limitez le propriétaire et le droit `TRIGGER` sur ces tables aux rôles de confiance. Néanmoins, si des utilisateurs sans confiance peuvent créer des tables, utilisez seulement des publications qui listent explicitement les tables. Autrement dit, créez une souscription `FOR ALL TABLES` ou `FOR TABLES IN SCHEMA` uniquement quand les superutilisateurs ont confiance en tous les utilisateurs autorisés à créer une table non temporaire sur le publieur ou sur l'abonné.

Le rôle utilisée pour la réplication doit avoir l'attribut `REPLICATION` (ou être un superutilisateur). Si le rôle ne dispose pas des attributs `SUPERUSER` et `BYPASSRLS`, les politiques de sécurité niveau ligne du publieur peuvent s'exécuter. Si le rôle n'a pas confiance en tous les propriétaires de tables, incluez `options=-crow_security=off` dans la chaîne de connexion ;: si un propriétaire de table ajoute ensuite une politique de sécurité de ligne, cette configuration imposera un arrêt de la réplication plutôt qu'une exécution de la politique. L'accès de ce rôle à l'instance doit avoir été déclaré dans `pg_hba.conf` et ce rôle doit avoir l'attribut `LOGIN`.

Le nom du plugin en sortie utilisé par la connexion de réplication doit être inclus dans le paramètre output_plugin_libraries du serveur. (Pour les abonnés, le nom du plugin qui est utilisé est `pgoutput`.) Les super-utilisateurs peuvent modifier la liste de confiance par-connexion, en incluant `options=-coutput_plugin_libraries=...` dans la chaîne de connexion.

Pour être capable de copier les données originales de la table, le rôle utilisé pour la connexion de réplication doit avoir le droit `SELECT` sur une table publiée (ou être un superutilisateur).

Pour créer une publication, l'utilisateur doit avoir le droit CREATE pour la base de données.

Pour ajouter des tables à une publication, l'utilisateur doit être propriétaire de ces tables. Pour ajouter toutes les tables d'un schéma dans une publication, l'utilisateur doit avoir l'attribut SUPERUSER. Pour créer une publication qui publie toutes les tables ou toutes les tables d'un schéma automatiquement, l'utilisateur doit avoir l'attribut SUPERUSER.

Pour créer un abonnement, l'utilisateur doit avoir les droits de superutilisateur.

Le processus apply lié à un abonnement tournera sur la base de données locale avec les droits d'un propriétaire de l'abonnement.

Sur le publieur, les droits sont vérifiés uniquement au démarrage de la connexion de réplication et ne sont pas vérifiés de nouveau à chaque fois qu'un enregistrement de changement est lu.

Sur l'abonné, les droits du propriétaire de la souscription sont vérifiés à chaque application d'une transaction. Si un processus worker est en cours de traitement pour appliquer une transaction au moment où le propriétaire de la souscription est changé dans une transaction concurrente, l'application de la transaction en cours continuera sous les droits de l'ancien propriétaire.

31.10. Paramètres de configuration

La réplication logique requiert de nombreuses configurations pour fonctionner.

Du côté de l'éditeur, `wal_level` doit être positionné à `logical`, et `max_replication_slots` doit être positionné au minimum au nombre d'abonnements que l'on va connecter, plus quelques-uns que l'on réservera pour les synchronisations des tables. Le paramètre `max_wal_senders` devrait être positionné au minimum à la même valeur que `max_replication_slots` en plus du nombre de réplicats physiques qui pourraient être connectés au même moment.

`max_replication_slots` doit aussi être configuré sur l'abonné. Il doit être configuré à au moins la valeur du nombre de souscriptions qui seront ajoutés par le souscripteur, plus une réserve pour la synchronisation des tables. `max_logical_replication_workers` doit être configuré à au moins la valeur du nombre de souscriptions, plus, là-aussi, une réserve pour la synchronisation des tables. En supplément, le paramètre `max_worker_processes` peut devoir être ajusté pour s'accorder au nombre de processus de réplication, (`max_logical_replication_workers` + `1`). Notez que certaines extensions et les requêtes parallélisées prennent elles aussi des unités de la réserve de `max_worker_processes`.

31.11. Démarrage rapide

En premier, définissez les options de configurations dans le fichier `postgresql.conf` :

```
wal_level = logical
```

La valeur par défaut des autres paramètres est suffisante pour une mise en place de base.

Le fichier `pg_hba.conf` doit être mis à jour pour autoriser la réplication (les valeurs dépendent de la configuration réelle de votre réseau et de l'utilisateur dont vous disposerez pour vous connecter) :

```
host     all     repuser     0.0.0.0/0     md5
```

Ensuite sur la base du publieur :

```
CREATE PUBLICATION mypub FOR TABLE users, departments;
```

Et sur la base abonnée :

```
CREATE SUBSCRIPTION mysub CONNECTION 'dbname=foo host=bar
 user=repuser' PUBLICATION mypub;
```

Les instructions précédentes vont démarrer le processus de réplication, qui va réaliser la synchronisation initiale du contenu des tables `users` et `departments` et qui commencera ensuite à répliquer les changements de manière incrémentale sur ces tables.

Chapitre 32. JIT (compilation à la volée)

Ce chapitre décrit ce qu'est le JIT (compilation à la volée), et comment le configurer dans PostgreSQL.

32.1. Qu'est-ce que le JIT ?

La compilation à la volée (ou JIT pour *Just-in-Time Compilation*) est le processus de transformation de l'évaluation d'un programme interprété en un programme natif, et ce pendant l'exécution. Par exemple, au lieu d'utiliser un code généraliste pouvant évaluer des expressions SQL arbitraires pour évaluer un prédicat SQL particulier comme `WHERE a.col = 3`, il est possible de générer une fonction spécifique à cette expression et qui peut être exécutée nativement par le CPU, apportant une accélération.

PostgreSQL sait procéder à une compilation JIT grâce à LLVM[1] s'il a été compilé avec `--with-llvm` (voir `--with-llvm`).

Consultez `src/backend/jit/README` pour plus de détails.

32.1.1. JIT Opérations accélérées

Actuellement, l'implémentation JIT de PostgreSQL supporte l'accélération de l'évaluation d'expression et du décodage d'enregistrement. Plusieurs autres opérations pourraient être accélérées dans le futur.

L'évaluation d'expression est utilisée pour évaluer les clauses `WHERE`, les listes de colonnes, les agrégats et les projections. Elle peut être accélérée en générant du code spécifique à chaque cas.

Le décodage d'enregistrement est le processus de transformation d'un enregistrement sur disque (voir Section 73.6.1) dans sa représentation en mémoire. Il peut être accéléré en créant une fonction spécifique au format de la table et au nombre de colonnes extraites.

32.1.2. Inclusion

PostgreSQL est très extensible et permet de définir de nouveaux types de données, fonctions, opérateurs et autres objets de base de données ; voir Chapitre 38. En fait, ceux intégrés sont implémentés avec à peu près les mêmes mécanismes. Cette extensibilité a un surcoût, par exemple à cause des appels de fonction (voir Section 38.3). Pour réduire ce surcoût, la compilation JIT peut intégrer le corps des petites fonctions dans les expressions qui les utilisent. Cela permet d'optimiser un pourcentage significatif du surcoût.

32.1.3. Optimisation

LLVM permet d'optimiser le code généré. Certaines optimisations sont suffisamment peu coûteuses pour être accomplies à chaque utilisation du JIT, alors que d'autres n'ont de bénéfice que pour les requêtes durant plus longtemps. Voir pour plus de détails sur les optimisations. [2]

32.2. Quand utiliser le JIT ?

La compilation JIT bénéficie surtout aux requêtes de longue durée et limitées par le processeur. Ce seront souvent des requêtes analytiques. Pour les requêtes courtes, le surcoût apporté par la compilation JIT sera souvent supérieur au temps qu'elle permet de gagner.

[1] https://llvm.org/
[2] https://llvm.org/docs/passes.html#transform-passes

Pour déterminer si la compilation JIT doit être utilisée, le coût total estimé d'une requête (voir Chapitre 75 et Section 20.7.2) est utilisé. Le coût estimé de la requête sera comparé à la configuration du paramètre jit_above_cost. Si le coût est supérieur, une compilation JIT sera opérée. Deux décisions supplémentaires sont encore nécessaires. Premièrement, si le coût estimé est plus important que la configuration de jit_inline_above_cost, les petites fonctions et opérateurs utilisés dans la requête seront intégrés. Ensuite, si le coût est plus important que la valeur de jit_optimize_above_cost, les optimisations coûteuses sont appliquées pour améliorer le code généré. Chacune de ses options augmente la surcharge de la compilation JIT mais peut réduire considérablement la durée d'exécution.

Ces décisions basées sur les coûts seront réalisées au moment de la planification, pas au moment de l'exécution. Ceci signifie que, quand des instructions préparées sont utilisées et qu'un plan générique est utilisé (voir PREPARE), les valeurs des paramètres de configuration en effet au moment de la préparation contrôlent les décisions, pas les valeurs des paramètres au moment de l'exécution.

> **Note**
>
> Si jit est à `off`, ou si aucune implémentation du JIT n'est disponible (par exemple parce que le serveur a été compilé sans `--with-llvm`), le JIT ne sera pas opéré, même s'il est considéré comme bénéfique selon les critères ci-dessus. Placer jit à `off` prend effet au moment de la planification comme de l'exécution.

EXPLAIN peut être utilisé pour voir si le JIT est utilisé ou pas. Par exemple, voici une requête n'utilisant pas le JIT :

```
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate  (cost=16.27..16.29 rows=1 width=8) (actual
 time=0.303..0.303 rows=1 loops=1)
   ->  Seq Scan on pg_class  (cost=0.00..15.42 rows=342 width=4)
 (actual time=0.017..0.111 rows=356 loops=1)
 Planning Time: 0.116 ms
 Execution Time: 0.365 ms
(4 rows)
```

Étant donné le coût de la planification, il est parfaitement raisonnable que le JIT ne soit pas utilisé, son coût aurait été supérieur au temps potentiellement épargné. Ajuster les limites de coût amèneront son utilisation :

```
=# SET jit_above_cost = 10;
SET
=# EXPLAIN ANALYZE SELECT SUM(relpages) FROM pg_class;
                                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate  (cost=16.27..16.29 rows=1 width=8) (actual
 time=6.049..6.049 rows=1 loops=1)
   ->  Seq Scan on pg_class  (cost=0.00..15.42 rows=342 width=4)
 (actual time=0.019..0.052 rows=356 loops=1)
 Planning Time: 0.133 ms
 JIT:
   Functions: 3
   Options: Inlining false, Optimization false, Expressions true,
 Deforming true
```

```
  Timing: Generation 1.259 ms, Inlining 0.000 ms, Optimization
0.797 ms, Emission 5.048 ms, Total 7.104 ms
Execution Time: 7.416 ms
```

Comme on le voit ici, le JIT a été utilisé, mais pas l'intégration et l'optimisation coûteuse. Si jit_inline_above_cost et jit_optimize_above_cost étaient abaissés comme jit_above_cost, cela changerait.

32.3. Configuration

La variable de configuration jit détermine si la compilation JIT est activée ou désactivée. S'il est activé, les variables de configuration jit_above_cost, jit_inline_above_cost et jit_optimize_above_cost déterminent si la compilation JIT est réalisée pour une requête et combien d'effort est dépensé pour le faire.

jit_provider détermine l'implémentation JIT utilisée. Il est rarement requis de le modifier. Voir Section 32.4.2.

Dans un but de développement et de débogage, quelques paramètres de configuration supplémentaires existent, comme décrit dans Section 20.17.

32.4. Extensibilité

32.4.1. Support de l'intégration pour les extensions

L'implémentation JIT de PostgreSQL peut intégrer le corps des fonctions de type `C` et `internal`, ainsi que les opérateurs basés sur ces fonctions. Voir Section 32.1.2. Pour le faire pour les fonctions au sein des extensions, la définition de ces fonctions doit être disponible. En utilisant PGXS pour construire une extension pour un serveur compilé avec le support JIT LLVM, les fichiers nécessaires seront automatiquement installés.

Les fichiers adéquats doivent être installés dans `$pkglibdir/bitcode/$extension/` et un résumé de ceux-ci dans `$pkglibdir/bitcode/$extension.index.bc`, où `$pkglibdir` est le répertoire retourné par `pg_config --pkglibdir` et `$extension` le nom (*basename*) de la librairie partagée de l'extension.

> **Note**
>
> Pour les fonctions construites dans PostgreSQL même, le bitcode est installé dans `$pkglibdir/bitcode/postgres`.

32.4.2. Fournisseur JIT interchangeable

PostgreSQL fournit une implémentation du JIT basée sur LLVM. L'interface au fournisseur du JIT est ouverte et le fournisseur peut être changé sans recompiler (bien qu'actuellement, le processus de construction fournit seulement des données de support pour LLVM). Le fournisseur actif est choisi par le GUC jit_provider.

32.4.2.1. Interface du fournisseur JIT

Un fournisseur JIT est chargé en chargeant dynamiquement la librairie partagée indiquée. Pour la situer, le chemin de recherche de librairie habituel est utilisé. Pour fournir les callbacks du fournisseur JIT et pour indiquer que la librairie est bien un fournisseur JIT, cette dernière doit fournir une fonction

C nommée `_PG_jit_provider_init`. À cette fonction est passée une structure qui doit être remplie avec les pointeurs des fonctions callback pour les différentes actions.

```
struct JitProviderCallbacks
{
    JitProviderResetAfterErrorCB reset_after_error;
    JitProviderReleaseContextCB release_context;
    JitProviderCompileExprCB compile_expr;
};

extern void _PG_jit_provider_init(JitProviderCallbacks *cb);
```

Chapitre 33. Tests de régression

Les tests de régression composent un ensemble exhaustif de tests pour l'implémentation SQL dans PostgreSQL. Ils testent les opérations SQL standards ainsi que les fonctionnalités étendues de PostgreSQL.

33.1. Lancer les tests

Les tests de régression peuvent être lancés sur un serveur déjà installé et fonctionnel ou en utilisant une installation temporaire à l'intérieur du répertoire de construction. De plus, ils peuvent être lancés en mode « parallèle » ou en mode « séquentiel ». Le mode séquentiel lance les scripts de test en série, alors que le mode parallèle lance plusieurs processus serveur pour paralléliser l'exécution des groupes de tests. Les tests parallèles permettent de s'assurer du bon fonctionnement des communications interprocessus et du verrouillage.

33.1.1. Exécuter les tests sur une installation temporaire

Pour lancer les tests de régression en parallèle après la construction, mais avant l'installation, il suffit de saisir

```
make check
```

dans le répertoire de premier niveau (on peut aussi se placer dans le répertoire `src/test/regress` et y lancer la commande). Au final, la sortie devrait ressembler à quelque chose comme

```
=====================
 All 193 tests passed.
=====================
```

ou une note indiquant l'échec des tests. Voir la Section 33.2 avant de supposer qu'un « échec » représente un problème sérieux.

Comme cette méthode de tests exécute un serveur temporaire, cela ne fonctionnera pas si vous avez construit le serveur en tant que root, étant donné que le serveur ne démarre pas en tant que root. La procédure recommandée est de ne pas construire en tant que root ou de réaliser les tests après avoir terminé l'installation.

Si vous avez configuré PostgreSQL pour qu'il s'installe dans un emplacement où existe déjà une ancienne installation de PostgreSQL et que vous lancez `make check` avant d'installer la nouvelle version, vous pourriez trouver que les tests échouent parce que les nouveaux programmes essaient d'utiliser les bibliothèques partagées déjà installées (les symptômes typiques sont des plaintes concernant des symboles non définis). Si vous souhaitez lancer les tests avant d'écraser l'ancienne installation, vous devrez construire avec `configure --disable-rpath`. Néanmoins, il n'est pas recommandé d'utiliser cette option pour l'installation finale.

Les tests de régression en parallèle lancent quelques processus avec votre utilisateur. Actuellement, le nombre maximum est de vingt scripts de tests en parallèle, ce qui signifie 40 processus : il existe un processus serveur, un psql et habituellement un processus parent pour le psql de chaque script de tests. Si votre système force une limite par utilisateur sur le nombre de processus, assurez-vous que cette limite est d'au moins 50, sinon vous pourriez obtenir des échecs hasardeux dans les tests en parallèle. Si vous ne pouvez pas augmenter cette limite, vous pouvez diminuer le degré de parallélisme en initialisant le paramètre `MAX_CONNECTIONS`. Par exemple,

```
make MAX_CONNECTIONS=10 check
```

ne lance pas plus de dix tests en même temps.

33.1.2. Exécuter les tests sur une installation existante

Pour lancer les tests après l'installation (voir le Chapitre 17), initialisez un répertoire de données et lancez le serveur comme expliqué dans le Chapitre 19, puis lancez

```
make installcheck
```

ou pour un test parallèle

```
make installcheck-parallel
```

Les tests s'attendront à contacter le serveur sur l'hôte local et avec le numéro de port par défaut, sauf en cas d'indication contraire avec les variables d'environnement `PGHOST` et `PGPORT`. Les tests seront exécutés dans une base de données nommée `regression` ; toute base de données existante de même nom sera supprimée.

Les tests créent aussi de façon temporaire des objets globaux, comme les rôles, les tablespaces et les abonnements. Ces objets auront des noms commençant avec `regress_`. Attention à l'utilisation du mode `installcheck` dans une installation qui a de vrais objets globaux nommés de cette manière.

33.1.3. Suites supplémentaires de tests

Les commandes `make check` et `make installcheck` exécutent seulement les tests de régression internes qui testent des fonctionnalités internes du serveur PostgreSQL. Les sources contiennent aussi des suites supplémentaires de tests, la plupart ayant à voir avec des fonctionnalités supplémentaires comme les langages optionnels de procédures.

Pour exécuter toutes les suites de tests applicables aux modules qui ont été sélectionnés à la construction, en incluant les tests internes, tapez une des commandes suivantes dans le répertoire principal de construction :

```
make check-world
make installcheck-world
```

Ces commandes exécutent les tests en utilisant, respectivement, un serveur temporaire ou un serveur déjà installé, comme expliqué précédemment pour `make check` et `make installcheck`. Les autres considérations sont identiques à celles expliquées précédemment pour chaque méthode. Notez que `make check-world` construit une instance séparée (répertoire de données temporaire) pour chaque module testé, donc cela réclame plus de temps et d'espace disque que `make installcheck-world`.

Sur une machine moderne avec plusieurs CPU et sans limites au niveau du système d'exploitation, il est possible d'aller bien plus vite avec le parallélisme. La solution utilisée par de nombreux développeurs de PostgreSQL revient en fait à exécuter tous les tests ainsi :

```
make check-world -j8 >/dev/null
```

avec une limite au niveau de la valeur de l'option `-j` proche du nombre de CPU disponibles. Supprimer stdout élimine une sortie verbeuse qui n'est pas particulièrement intéressante quand on veut simplement vérifier le succès de l'opération. (En cas d'échec, les messages sur stderr sont généralement suffisant pour déterminer ce qu'il faut vérifier avec précision.)

Autrement, vous pouvez exécuter les suites individuelles de tests en tapant `make check` ou `make installcheck` dans le sous-répertoire approprié du répertoire de construction. Gardez en tête que

`make installcheck` suppose que vous avez installé les modules adéquats, pas seulement le serveur de base.

Les tests supplémentaires pouvant être demandés de cette façon incluent :

- Les tests de régression pour les langages optionnels de procédures stockées. Ils sont situés dans `src/pl`.
- Les tests de régression pour les modules `contrib`, situés dans `contrib`. Tous les modules `contrib` n'ont pas forcément des suites de tests.
- Les tests de régression pour les bibliothèques d'interface, situées dans `src/interfaces/libpq/test` et `src/interfaces/ecpg/test`.
- Les tests pour les méthodes d'authentification supportés nativement, situés dans `src/test/authentication`. (Voir ci-dessous pour des tests supplémentaires relatifs à l'authentification.)
- Les tests simulant le comportement de sessions concurrentes, situés dans `src/test/isolation`.
- Les tests de la restauration après crash et de la réplication physique, situés dans `src/test/recovery`.
- Les tests pour la réplication logique, situés dans `src/test/subscription`.
- Les tests des programmes clients, situés dans `src/bin`.

Lors de l'utilisation du mode `installcheck`, ces tests créeront et supprimeront des bases de données de test dont les noms contiennent `regression`, par exemple `pl_regression` ou `contrib_regression`. Il faut faire bien attention à ne pas utiliser le mode `installcheck` avec une installation qui contient des bases de données importantes nommées de cette façon.

Certains des suites de tests supplémentaires utilisent l'infrastructure TAP expliquée dans Section 33.4. Les tests TAP sont seulement exécutés si PostgreSQL a été configuré avec l'option `--enable-tap-tests`. Cela est recommandé pour le développement, mais peut être omis s'il n'y a pas d'installation Perl appropriée.

Certains tests ne sont pas exécutés par défaut, soit parce qu'ils ne sont pas sécurisés pour fonctionner sur un système multi-utilisateurs, soit parce qu'ils nécessitent un logiciel spécifique. Vous pouvez décider quels seront les suites de test à exécuter lors de l'execution de `make` par son paramètrage ou par l'affectation d'une configuration à la variable d'environnement `PG_TEST_EXTRA` dans une liste séparée par des espaces, par exemple :

```
make check-world PG_TEST_EXTRA='kerberos ldap ssl'
```

Les valeurs suivantes sont actuellement prises en charge :

`kerberos`

Exécute la suite de tests présent dans `src/test/kerberos`. Cela nécessite une installation de MIT Kerberos et ouvre les sockets d'écoute TCP/IP.

`ldap`

Exécute la suite de tests présent dans `src/test/ldap`. Cela nécessite une installation de OpenLDAP et ouvre les sockets d'écoute TCP/IP.

`ssl`

Exécute la suite de tests présent dans `src/test/ssl`. Ceci ouvre les sockets d'écoute TCP/IP.

`wal_consistency_checking`

Utilise `wal_consistency_checking=all` lors de l'exécution de certains tests sous `src/test/recovery`. Désactivé par défaut car il est gourmand en ressources.

Les tests pour les fonctionnalités qui ne sont pas prises en charge par la configuration de construction actuelle ne sont pas exécutés même si elles sont mentionnées dans `PG_TEST_EXTRA`.

De plus, il existe des tests dans `src/test/modules` qui seront exécutés par `make check-world` mais pas par `make installcheck-world`. Ceci est dû au fait qu'ils installent des extensions qui ne sont pas de production ou qui ont des effets de bord considérés non désirables pour une installation en production. Vous pouvez utiliser `make install` et `make installcheck` dans un de ces sous-répertoires si vous le souhaitez mais il n'est pas recommandé de la faire sur un serveur autre qu'un serveur de tests.

33.1.4. Locale et encodage

Par défaut, les tests sur une installation temporaire utilisent la locale définie dans l'environnement et l'encodage de la base de données correspondante est déterminé par `initdb`. Il peut être utile de tester différentes locales en configurant les variables d'environnement appropriées. Par exemple :

```
make check LANG=C
make check LC_COLLATE=en_US.utf8 LC_CTYPE=fr_CA.utf8
```

Pour des raisons d'implémentation, configurer `LC_ALL` ne fonctionne pas dans ce cas. Toutes les autres variables d'environnement liées à la locale fonctionnent.

Lors d'un test sur une installation existante, la locale est déterminée par l'instance existante et ne peut pas être configurée séparément pour un test.

Vous pouvez aussi choisir l'encodage de la base explicitement en configurant la variable `ENCODING`. Par exemple :

```
make check LANG=C ENCODING=EUC_JP
```

Configurer l'encodage de la base de cette façon n'a un sens que si la locale est C. Dans les autres cas, l'encodage est choisi automatiquement à partir de la locale. Spécifier un encodage qui ne correspond pas à la locale donnera une erreur.

L'encodage de la base de données peut être configuré pour des tests sur une installation temporaire ou existante, bien que, dans ce dernier cas, il doit être compatible avec la locale d'installation.

33.1.5. Paramétrages personnalisés du serveur

Un paramétrage personnalisé du serveur peut être configuré dans la variable d'environnement `PGOPTIONS` pour les tests qui suivront (pour les paramètres qui le permettent) :

```
make check PGOPTIONS="-c force_parallel_mode=regress -c
 work_mem=50MB"
```

Lors de l'exécution sur une installation temporaire, un paramétrage personnalisé peut aussi être configuré en fournissant un fichier `postgresql.conf` déjà configuré :

```
echo 'log_checkpoints = on' > test_postgresql.conf
```

```
echo 'work_mem = 50MB' >> test_postgresql.conf
make check EXTRA_REGRESS_OPTS="--temp-config=test_postgresql.conf"
```

Ceci peut être utile pour activer des traces supplémentaires, pour ajuster les limites en ressource ou pour activer des vérifications supplémentaires à l'exécution, telles que debug_discard_caches.

33.1.6. Tests supplémentaires

La suite interne de tests de régression contient quelques fichiers de tests qui ne sont pas exécutés par défaut, car ils pourraient dépendre de la plateforme ou prendre trop de temps pour s'exécuter. Vous pouvez les exécuter ou en exécuter d'autres en configurant la variable `EXTRA_TESTS`. Par exemple, pour exécuter le test `numeric_big` :

```
make check EXTRA_TESTS=numeric_big
```

33.2. Évaluation des tests

Quelques installations de PostgreSQL proprement installées et totalement fonctionnelles peuvent « échouer » sur certains des tests de régression à cause de certains points spécifiques à la plateforme comme une représentation de nombres à virgules flottantes ou « message wording ». Les tests sont actuellement évalués en utilisant une simple comparaison `diff` avec les sorties générées sur un système de référence, donc les résultats sont sensibles aux petites différences système. Quand un test est rapporté comme « échoué », toujours examiner les différences entre les résultats attendus et ceux obtenus ; vous pourriez très bien trouver que les différences ne sont pas significatives. Néanmoins, nous nous battons toujours pour maintenir des fichiers de références précis et à jour pour toutes les plateformes supportées de façon à ce que tous les tests puissent réussir.

Les sorties actuelles des tests de régression sont dans les fichiers du répertoire `src/test/regress/results`. Le script de test utilise `diff` pour comparer chaque fichier de sortie avec les sorties de référence stockées dans le répertoire `src/test/regress/expected`. Toutes les différences sont conservées pour que vous puissiez les regarder dans `src/test/regress/regression.diffs`. (Lors de l'exécution d'une suite de tests en dehors des tests internes, ces fichiers doivent apparaître dans le sous-répertoire adéquat, mais pas `src/test/regress`.)

Si vous n'aimez pas les options utilisées par défaut pour la commande `diff`, configurez la variable d'environnement `PG_REGRESS_DIFF_OPTS`. Par exemple `PG_REGRESS_DIFF_OPTS='-c'` (ou vous pouvez lancer `diff` vous-même, si vous préférez).

Si, pour certaines raisons, une plateforme particulière génère un « échec » pour un test donné mais qu'une revue de la sortie vous convaint que le résultat est valide, vous pouvez ajouter un nouveau fichier de comparaison pour annuler le rapport d'échec pour les prochains lancements du test. Voir la Section 33.3 pour les détails.

33.2.1. Différences dans les messages d'erreurs

Certains des tests de régression impliquent des valeurs en entrée intentionnellement invalides. Les messages d'erreur peuvent provenir soit du code de PostgreSQL soit des routines système de la plateforme hôte. Dans ce dernier cas, les messages pourraient varier entre plateformes mais devraient toujours refléter des informations similaires. Ces différences dans les messages résulteront en un échec du test de régression qui pourrait être validé après vérification.

33.2.2. Différences au niveau des locales

Si vous lancez des tests sur un serveur initialisé avec une locale autre que C, alors il pourrait y avoir des différences dans les ordres de tris. La suite de tests de régression est initialisée pour gérer ce problème en fournissant des fichiers de résultats alternatifs qui gèrent ensemble un grand nombre de locales.

Pour exécuter les tests dans une locale différente lors de l'utilisation de la méthode d'installation temporaire, passez les variables d'environnement relatives à la locale sur la ligne de commande de `make`, par exemple :

```
make check LANG=de_DE.utf8
```

(Le pilote de tests des régressions déconfigure `LC_ALL`, donc choisir la locale par cette variable ne fonctionne pas.) Pour ne pas utiliser de locale, vous devez soit déconfigurer toutes les variables d'environnement relatives aux locales (ou les configurer à `C`) ou utiliser une option spéciale :

```
make check NO_LOCALE=1
```

Lors de l'exécution des tests sur une installation existante, la configuration de la locale est déterminée d'après l'installation existante. Pour la modifier, initialiser le cluster avec une locale différente en passant les options appropriées à `initdb`.

En général, il est conseillé d'essayer l'exécution des tests de régression dans la configuration de locale souhaitée pour l'utilisation en production, car cela testera aussi les portions de code relatives à l'encodage et à la locale qui pourront être utilisées en production. Suivant l'environnement du système d'exploitation, vous pourrez obtenir des échecs, mais vous saurez au moins le comportement à attendre sur la locale lorsque vous utiliserez vos vraies applications.

33.2.3. Différences au niveau des dates/heures

La plupart des résultats date/heure sont dépendants de l'environnement de zone horaire. Les fichiers de référence sont générés pour la zone horaire `America/Los_Angeles`, et il y aura des échecs apparents si les tests ne sont pas lancés avec ce paramétrage de fuseau horaire. Le pilote des tests de régression initialise la variable d'environnement `PGTZ` à `America/Los_Angeles` ce qui nous assure normalement de bons résultats.

33.2.4. Différences sur les nombres à virgules flottantes

Quelques tests impliquent des calculs sur des nombres flottants à 64 bits (`double precision`) à partir de colonnes de tables. Des différences dans les résultats appliquant des fonctions mathématiques à des colonnes `double precision` ont été observées. Les tests de `float8` et `geometry` sont particulièrement sensibles aux différences entre plateformes, voire aux différentes options d'optimisation des compilateurs. L'œil humain est nécessaire pour déterminer la véritable signification de ces différences, habituellement situées après la dixième décimale.

Certains systèmes affichent moins zéro comme `-0` alors que d'autres affichent seulement `0`.

Certains systèmes signalent des erreurs avec `pow()` et `exp()` différemment suivant le mécanisme attendu du code de PostgreSQL.

33.2.5. Différences dans l'ordre des lignes

Vous pourriez voir des différences dans lesquelles les mêmes lignes sont affichées dans un ordre différent de celui qui apparaît dans le fichier de référence. Dans la plupart des cas, ce n'est pas à strictement parlé un bogue. La plupart des scripts de tests de régression ne sont pas assez stricts pour utiliser un `ORDER BY` sur chaque `SELECT` et, du coup, l'ordre des lignes pourrait ne pas être correctement défini suivant la spécification SQL. En pratique, comme nous sommes avec les mêmes requêtes sur les mêmes données avec le même logiciel, nous obtenons habituellement le même résultat

sur toutes les plateformes et le manque d'`ORDER BY` n'est pas un problème. Quelques requêtes affichent des différences d'ordre entre plateformes. Lors de tests avec un serveur déjà installé, les différences dans l'ordre des lignes peuvent aussi être causées par un paramètrage des locales à une valeur différente de C ou par un paramètrage personnalisé, comme des valeurs personnalisées de `work_mem` ou du coût du planificateur.

Du coup, si vous voyez une différence dans l'ordre, vous n'avez pas à vous inquiéter sauf si la requête possède un `ORDER BY` que votre résultat ne respecte pas. Néanmoins, rapportez tout de même ce problème que nous ajoutions un `ORDER BY` à cette requête pour éliminer les faux « échecs » dans les versions suivantes.

Vous pourriez vous demander pourquoi nous n'ordonnons pas toutes les requêtes des tests de régression explicitement pour supprimer ce problème une fois pour toutes. La raison est que cela rendrait les tests de régression moins utiles car ils tendraient à exercer des types de plans de requêtes produisant des résultats ordonnés à l'exclusion de celles qui ne le font pas.

33.2.6. Profondeur insuffisante de la pile

Si les tests d'`erreurs` se terminent avec un arrêt brutal du serveur pendant la commande `select infinite_recurse()`, cela signifie que la limite de la plateforme pour la taille de pile du processus est plus petite que le paramètre max_stack_depth ne l'indique. Ceci est corrigeable en exécutant le postmaster avec une limite pour la taille de pile plus importante (4 Mo est recommandé avec la valeur par défaut de `max_stack_depth`). Si vous n'êtes pas capables de le faire, une alternative est de réduire la valeur de `max_stack_depth`.

Sur les plateformes supportant `getrlimit()`, le serveur devrait choisir automatiquement une valeur sûre pour `max_stack_depth` ; donc, à moins de surcharger manuellement ce paramètre, un échec de ce type est un bug à reporter.

33.2.7. Test « random »

Le script de tests `random` a pour but de produire des résultats aléatoires. Dans de très rares cas, ceci fait échouer random aux tests de régression. Saisir :

```
diff results/random.out expected/random.out
```

ne devrait produire au plus que quelques lignes différentes. Cela est normal et ne devient préoccupant que si les tests random échouent en permanence lors de tests répétés

33.2.8. Paramètres de configuration

Lors de l'exécution de tests contre une installation existante, certains paramètres configurés à des valeurs spécifiques pourraient causer l'échec des tests. Par exemple, modifier des paramètres comme `enable_seqscan` ou `enable_indexscan` pourrait être la cause de changements de plan affectant le résultat des tests qui utilisent `EXPLAIN`.

33.3. Fichiers de comparaison de variants

Comme certains de ces tests produisent de façon inhérente des résultats dépendants de l'environnement, nous avons fourni des moyens de spécifier des fichiers résultats alternatifs « attendus ». Chaque test de régression peut voir plusieurs fichiers de comparaison affichant les résultats possibles sur différentes plateformes. Il existe deux mécanismes indépendants pour déterminer quel fichier de comparaison est utilisé pour chaque test.

Le premier mécanisme permet de sélectionner les fichiers de comparaison suivant des plateformes spécifiques. Le fichier de correspondance `src/test/regress/resultmap` définit le fichier de comparaison à utiliser pour chaque plateforme. Pour éliminer les tests « échoués » par erreur pour une

plateforme particulière, vous choisissez ou vous créez un fichier variant de résultat, puis vous ajoutez une ligne au fichier `resultmap`.

Chaque ligne du fichier de correspondance est de la forme

```
nomtest:sortie:modeleplateform=fichiercomparaison
```

Le nom de tests est juste le nom du module de tests de régression particulier. La valeur en sortie indique le fichier à vérifier. Pour les tests de régression standards, c'est toujours `out`. La valeur correspond à l'extension de fichier du fichier en sortie. Le modèle de plateforme est un modèle dans le style des outils Unix `expr` (c'est-à-dire une expression rationnelle avec une ancre implicite `^` au début). Il est testé avec le nom de plateforme affiche par `config.guess`. Le nom du fichier de comparaison est le nom de base du fichier de comparaison substitué.

Par exemple : il manque à certains systèmes une fonction `strtof` qui fonctionne, pour laquelle notre contournement cause des erreurs d'arrondis dans le test de régression `float4`. Du coup, nous fournissons un fichier de comparaison variable, `float4-misrounded-input.out`, qui inclut les résultats attendus sur ces systèmes. Pour faire taire les messages d'« échec » erronés sur les plateformes HP-UX 10, `resultmap` inclut

```
+float4:out:hppa.*-hp-hpux10.*=float4-misrounded-input.out
```

qui se déclenche sur toute machine où la sortie de `config.guess` correspond à `hppa.*-hp-hpux10.*`. D'autres lignes dans `resultmap` sélectionnent le fichier de comparaison variable pour les autres plateformes si c'est approprié.

Le second mécanisme de sélection des fichiers de comparaison variants est bien plus automatique : il utilise simplement la « meilleure correspondance » parmi les différents fichiers de comparaison fournis. Le script pilote des tests de régression considère le fichier de comparaison standard pour un test, *`nomtest`*`.out`, et les fichiers variants nommés *`nomtest_chiffre`*`.out` (où *`chiffre`* est un seul chiffre compris entre `0` et `9`). Si un tel fichier établit une correspondance exacte, le test est considéré réussi ; sinon, celui qui génère la plus petite différence est utilisé pour créer le rapport d'échec. (Si `resultmap` inclut une entrée pour le test particulier, alors le *`nomtest`* de base est le nom de substitut donné dans `resultmap`.)

Par exemple, pour le test `char`, le fichier de comparaison `char.out` contient des résultats qui sont attendus dans les locales `C` et `POSIX`, alors que le fichier `char_1.out` contient des résultats triés comme ils apparaissent dans plusieurs autres locales.

Le mécanisme de meilleure correspondance a été conçu pour se débrouiller avec les résultats dépendant de la locale mais il peut être utilisé dans toute situation où les résultats des tests ne peuvent pas être prédits facilement à partir de la plateforme seule. Une limitation de ce mécanisme est que le pilote test ne peut dire quelle variante est en fait « correcte » dans l'environnement en cours ; il récupèrera la variante qui semble le mieux fonctionner. Du coup, il est plus sûr d'utiliser ce mécanisme seulement pour les résultats variants que vous voulez considérer comme identiquement valides dans tous les contextes.

33.4. Tests TAP

Différents tests, en particulier les tests des programmes clients sous `src/bin`, utilisent les outils TAP de Perl et sont exécutés en utilisant le programme de tests Perl appelé `prove`. Les programmes de test clients situés dans `src/bin` utilisent les outils Perl TAP et sont exécutés par `prove`. Il est possible de passer des options en ligne de commande à `prove` en positionnant la variable `make PROVE_FLAGS`, par exemple :

```
make -C src/bin check PROVE_FLAGS='--timer'
```

Voir la page de manuel de `prove` pour plus d'information.

La variable PROVE_TESTS de la commande make peut être utilisée pour définir une liste de chemins relatifs séparés par des espaces blancs, vers le Makefile appelant prove pour lancer le sous-ensemble spécifié de tests à la place de la valeur par défaut t/*.pl. Par exemple :

```
make check PROVE_TESTS='t/001_test1.pl t/003_test3.pl'
```

Les tests TAP nécessitent le module IPC::Run. Ce module est disponible depuis CPAN ou un paquet du système d'exploitation. Ils requièrent aussi que PostgreSQL soit configuré avec l'option --enable-tap-tests.

En général, les tests TAP testeront les exécutables dans une arborescence d'installation préalable si vous exécutez make installcheck, ou construirons une nouvelle arborescence d'installation locale à partir des sources courants si vous dites make check. Dans tous les cas, ils initialiseront une instance locale (répertoire de données) et exécuteront temporairement un serveur dessus. Certains de ces tests s'exécutent avec plus d'un serveur. De ce fait, ces tests peuvent être très consommateurs en ressource.

Il est important de réaliser que les tests TAP démarreront les serveurs de test même quand vous utilisez make installcheck ; ceci est contraire à l'infrastructure traditionnelle de tests (donc non TAP) qui s'attend à utiliser un serveur de tests déjà en cours d'exécution. Certains sous-répertoires de PostgreSQL contiennent à la fois des tests dans le style traditionnel et d'autres dans le style TAP, ceci signifiant que make installcheck produira un mix de résultats provenant de serveurs temporaires et d'un serveur de tests déjà en cours d'exécution.

33.5. Examen de la couverture des tests

Le code source de PostgreSQL peut être compilé avec des informations supplémentaire sur la couverture des tests, pour qu'il devienne possible d'examiner les parties du code couvertes par les tests de régression ou par toute suite de tests exécutée avec le code. Cette fonctionnalité est supportée en compilant avec GCC et nécessite les programmes gcov et lcov.

La suite typique de commandes ressemble à ceci :

```
./configure --enable-coverage ... OTHER OPTIONS ...
make
make check # or other test suite
make coverage-html
```

Puis pointez votre navigateur HTML vers coverage/index.html.

Si vous n'avez pas lcov ou préférez une sortie texte par rapport à un rapport HTML, vous pouvez exécuter

```
make coverage
```

au lieu de make coverage-html, qui produira des fichiers de sortie .gcov pour chaque fichier source concerné par le test. (make coverage et make coverage-html surchargeront les fichiers de l'autre, donc les mixer pourrait apporter de la confusion.)

Vous pouvez exécuter différents tests avant de faire la rapport de couverture ; le nombre d'exécutions va s'accumuler. Si vous voulez réinitialiser le nombre d'exécutions entre vos différents tests, exécutez :

```
make coverage-clean
```

Vous pouvez exécuter la commande `make coverage-html` ou `make coverage` dans un sous-répertoire si vous voulez un rapport de couverture pour seulement une partie du code.

Utilisez `make distclean` pour tout nettoyer une fois terminé.

Partie IV. Interfaces client

Cette partie décrit les interfaces de programmation client distribuées avec PostgreSQL. Chaque chapitre peut être lu indépendamment. Noter qu'il y a beaucoup d'autres interfaces de programmation pour des clients, chacune distribuée séparément avec sa propre documentation. Les lecteurs de cette partie doivent être familiers de l'utilisation des requêtes SQL de manipulation et d'interrogation d'une base (voir la Partie II), et bien sûr du langage de programmation de l'interface utilisée.

Table des matières

34. libpq -- Bibliothèque C 941
34.1. Fonctions de contrôle de connexion à la base de données 941
34.1.1. Chaînes de connexion 949
34.1.2. Mots clés de la chaîne de connexion 951
34.2. Fonctions de statut de connexion 959
34.3. Fonctions d'exécution de commandes 965
34.3.1. Fonctions principales 965
34.3.2. Récupérer l'information dans le résultat des requêtes 974
34.3.3. Récupérer d'autres informations de résultats 978
34.3.4. Échapper les chaînes dans les commandes SQL 979
34.4. Traitement des commandes asynchrones 982
34.5. Mode pipeline 987
34.5.1. Utiliser le mode pipeline 987
34.5.2. Fonctions associées avec le mode pipeline 989
34.5.3. Quand utiliser le mode pipeline 991
34.6. Récupérer le résultats des requêtes ligne par ligne 991
34.7. Annuler des requêtes en cours d'exécution 992
34.8. Interface rapide (Fast Path) 993
34.9. Notification asynchrone 994
34.10. Fonctions associées à la commande COPY 995
34.10.1. Fonctions d'envoi de données pour COPY 996
34.10.2. Fonctions de réception des données de COPY 997
34.10.3. Fonctions obsolètes pour COPY 998
34.11. Fonctions de contrôle 1000
34.12. Fonctions diverses 1002
34.13. Traitement des messages 1006
34.14. Système d'événements 1007
34.14.1. Types d'événements 1008
34.14.2. Procédure de rappel de l'événement 1010
34.14.3. Fonctions de support des événements 1010
34.14.4. Exemple d'un événement 1012
34.15. Variables d'environnement 1014
34.16. Fichier de mots de passe 1016
34.17. Fichier des services de connexion 1017
34.18. Recherche LDAP des paramètres de connexion 1017
34.19. Support de SSL 1018
34.19.1. Vérification par le client du certificat serveur 1019
34.19.2. Certificats des clients 1020
34.19.3. Protection fournie dans les différents modes 1021
34.19.4. Utilisation des fichiers SSL 1022
34.19.5. Initialisation de la bibliothèque SSL 1022
34.20. Comportement des programmes threadés 1023
34.21. Construire des applications avec libpq 1024
34.22. Exemples de programmes 1025
35. Large Objects 1037
35.1. Introduction 1037
35.2. Fonctionnalités de l'implantation 1037
35.3. Interfaces client 1037
35.3.1. Créer un « Large Object » 1038
35.3.2. Importer un « Large Object » 1038
35.3.3. Exporter un « Large Object » 1039
35.3.4. Ouvrir un « Large Object » existant 1039
35.3.5. Écrire des données dans un « Large Object » 1040
35.3.6. Lire des données à partir d'un « Large Object » 1040
35.3.7. Recherche dans un « Large Object » 1040

35.3.8. Obtenir la position de recherche d'un « Large Object » 1041
35.3.9. Tronquer un « Large Object » 1041
35.3.10. Fermer un descripteur de « Large Object » 1042
35.3.11. Supprimer un objet large 1042
35.4. Fonctions du côté serveur 1042
35.5. Programme d'exemple 1043
36. ECPG -- SQL embarqué en C 1049
36.1. Le Concept 1049
36.2. Gérer les Connexions à la Base de Données 1050
36.2.1. Se Connecter au Serveur de Base de Données 1050
36.2.2. Choisir une connexion 1051
36.2.3. Fermer une Connexion 1053
36.3. Exécuter des Commandes SQL 1054
36.3.1. Exécuter des Ordres SQL 1054
36.3.2. Utiliser des Curseurs 1055
36.3.3. Gérer les Transactions 1055
36.3.4. Requêtes préparées 1056
36.4. Utiliser des Variables Hôtes 1057
36.4.1. Aperçu 1057
36.4.2. Sections Declare 1057
36.4.3. Récupérer des Résultats de Requêtes 1058
36.4.4. Correspondance de Type 1059
36.4.5. Manipuler des Types de Données SQL Non-Primitives 1067
36.4.6. Indicateurs 1072
36.5. SQL Dynamique 1072
36.5.1. Exécuter des Ordres SQL Dynamiques sans Jeu de Donnée 1072
36.5.2. Exécuter une requête avec des paramètres d'entrée 1073
36.5.3. Exécuter une Requête avec un Jeu de Données 1073
36.6. Librairie pgtypes 1074
36.6.1. Chaîne de caractères 1074
36.6.2. Le type numeric 1075
36.6.3. Le Type date 1078
36.6.4. Le Type timestamp 1082
36.6.5. Le Type interval 1086
36.6.6. Le Type decimal 1087
36.6.7. errno Valeurs de pgtypeslib 1088
36.6.8. Constantes Spéciales de pgtypeslib 1089
36.7. Utiliser les Zones de Descripteur 1089
36.7.1. Zones de Descripteur SQL nommées 1089
36.7.2. Zones de Descripteurs SQLDA 1092
36.8. Gestion des Erreurs 1103
36.8.1. Mettre en Place des Callbacks 1103
36.8.2. sqlca 1105
36.8.3. SQLSTATE contre SQLCODE 1107
36.9. Directives de Préprocesseur 1110
36.9.1. Inclure des Fichiers 1110
36.9.2. Les Directives define et undef 1111
36.9.3. Directives ifdef, ifndef, elif, else, et endif 1112
36.10. Traiter des Programmes en SQL Embarqué 1113
36.11. Fonctions de la Librairie 1114
36.12. Large Objects 1114
36.13. Applications C++ 1116
36.13.1. Portée des Variable Hôtes 1116
36.13.2. Développement d'application C++ avec un Module Externe en C 1118
36.14. Commandes SQL Embarquées 1120
36.15. Mode de Compatibilité Informix 1146
36.15.1. Types supplémentaires 1147
36.15.2. Ordres SQL Embarqués Supplémentaires/Manquants 1147

36.15.3. Zones de Descripteurs SQLDA Compatibles Informix 1147
36.15.4. Fonctions Additionnelles 1151
36.15.5. Constantes Supplémentaires 1161
36.16. Mode de compatibilité Oracle 1162
36.17. Fonctionnement Interne 1163
37. Schéma d'information 1166
37.1. Le schéma 1166
37.2. Types de données 1166
37.3. information_schema_catalog_name 1167
37.4. administrable_role_authorizations 1167
37.5. applicable_roles 1167
37.6. attributes 1168
37.7. character_sets 1170
37.8. check_constraint_routine_usage 1171
37.9. check_constraints 1172
37.10. collations 1172
37.11. collation_character_set_applicability 1172
37.12. column_column_usage 1173
37.13. column_domain_usage 1173
37.14. column_options 1174
37.15. column_privileges 1174
37.16. column_udt_usage 1175
37.17. columns 1175
37.18. constraint_column_usage 1179
37.19. constraint_table_usage 1179
37.20. data_type_privileges 1180
37.21. domain_constraints 1180
37.22. domain_udt_usage 1181
37.23. domains 1181
37.24. element_types 1183
37.25. enabled_roles 1186
37.26. foreign_data_wrapper_options 1186
37.27. foreign_data_wrappers 1186
37.28. foreign_server_options 1187
37.29. foreign_servers 1187
37.30. foreign_table_options 1188
37.31. foreign_tables 1188
37.32. key_column_usage 1189
37.33. parameters 1189
37.34. referential_constraints 1191
37.35. role_column_grants 1192
37.36. role_routine_grants 1192
37.37. role_table_grants 1193
37.38. role_udt_grants 1194
37.39. role_usage_grants 1194
37.40. routine_column_usage 1195
37.41. routine_privileges 1196
37.42. routine_routine_usage 1196
37.43. routine_sequence_usage 1197
37.44. routine_table_usage 1198
37.45. routines 1198
37.46. schemata 1203
37.47. sequences 1203
37.48. sql_features 1204
37.49. sql_implementation_info 1205
37.50. sql_parts 1205
37.51. sql_sizing 1205
37.52. table_constraints 1206

37.53. table_privileges 1207
37.54. tables 1207
37.55. transforms 1208
37.56. triggered_update_columns 1209
37.57. triggers 1209
37.58. udt_privileges 1211
37.59. usage_privileges 1211
37.60. user_defined_types 1212
37.61. user_mapping_options 1214
37.62. user_mappings 1214
37.63. view_column_usage 1215
37.64. view_routine_usage 1215
37.65. view_table_usage 1216
37.66. views 1216

Chapitre 34. libpq -- Bibliothèque C

libpq est l'interface de programmation pour les applications C avec PostgreSQL. libpq est un ensemble de fonctions permettant aux programmes clients d'envoyer des requêtes au serveur PostgreSQL et de recevoir les résultats de ces requêtes.

libpq est aussi le moteur sous-jacent de plusieurs autres interfaces de programmation de PostgreSQL, comme ceux écrits pour C++, Perl, Python, Tcl et ECPG. Donc, certains aspects du comportement de libpq seront importants pour vous si vous utilisez un de ces paquetages. En particulier, la Section 34.15, la Section 34.16 et la Section 34.19 décrivent le comportement que verra l'utilisateur de toute application utilisant libpq.

Quelques petits programmes sont inclus à la fin de ce chapitre (Section 34.22) pour montrer comment écrire des programmes utilisant libpq. Il existe aussi quelques exemples complets d'applications libpq dans le répertoire `src/test/examples` venant avec la distribution des sources.

Les programmes clients utilisant libpq doivent inclure le fichier d'en-tête `libpq-fe.h` et doivent être lié avec la bibliothèque libpq.

34.1. Fonctions de contrôle de connexion à la base de données

Les fonctions suivantes concernent la réalisation d'une connexion avec un serveur PostgreSQL. Un programme peut avoir plusieurs connexions ouvertes sur des serveurs à un même moment (une raison de la faire est d'accéder à plusieurs bases de données). Chaque connexion est représentée par un objet `PGconn`, obtenu avec la fonction `PQconnectdb`, `PQconnectdbParams` ou `PQsetdbLogin`. Notez que ces fonctions renverront toujours un pointeur d'objet non nul, sauf peut-être dans un cas de manque de mémoire pour l'allocation de l'objet `PGconn`. La fonction `PQstatus` doit être appelée pour vérifier le code retour pour une connexion réussie avant de lancer des requêtes via l'objet de connexion.

> **Avertissement**
>
> Si des utilisateurs non dignes de confiance ont accès à une base de données qui n'a pas adopté une méthode sécurisée d'utilisation des schémas, commencez chaque session en supprimant du `search_path` les schémas accessibles par tout le monde. Il est possible de configurer le paramètre `options` à la valeur `-csearch_path=`. Sinon, il est possible d'exécuter `PQexec(`*`conn`*`, "SELECT pg_catalog.set_config('search_path', '', false)")` tout de suite après la connexion. Cette considération n'est pas spécifique à la libpq ; elle s'applique à chaque interface permettant d'exécuter des commandes SQL arbitraires.

> **Avertissement**
>
> Sur Unix, la création d'un processus via l'appel système fork() avec des connexions libpq ouvertes peut amener à des résultats imprévisibles car les processus parent et enfants partagent les même sockets et les mêmes ressources du système d'exploitation. Pour cette raison, un tel usage n'est pas recommandé, alors qu'exécuter un `exec` à partir du processus enfant pour charger un nouvel exécutable est sûr.

`PQconnectdbParams`

Établit une nouvelle connexion au serveur de base de données.

```
PGconn *PQconnectdbParams(const char * const *keywords,
                          const char * const *values,
                          int expand_dbname);
```

Cette fonction ouvre une nouvelle connexion à la base de données en utilisant les paramètres à partir des deux tableaux terminés par un `NULL`. Le premier, `keywords`, est défini comme un tableau de chaînes, chacune étant un mot-clé. Le second, `values`, donne la valeur pour chaque mot-clé. Contrairement à `PQsetdbLogin` ci-dessous, l'ensemble des paramètres peut être étendu sans changer la signature de la fonction donc son utilisation (ou ses versions non bloquantes, à savoir `PQconnectStartParams` et `PQconnectPoll`) est recommandée pour les nouvelles applications.

Les mots clés actuellement reconnus sont listés dans Section 34.1.2.

Les tableaux fournis peuvent être vides pour utiliser tous les paramètres par défaut ou peuvent contenir une ou plusieurs configurations. Elles doivent correspondre en longueur. Le traitement s'arrêtera à la première entrée `NULL` dans le tableau `keywords`. De plus, si l'entrée `values` associée à une entrée `keywords` non `NULL` est `NULL` ou une chaîne vide, cette entrée est ignorée et le traitement continue avec la prochaine paire d'entrées des tableaux.

Quand `expand_dbname` est différent de zéro, la valeur pour le premier mot-clé *`dbname`* est testé pour vérifier s'il s'agit d'une *chaîne de connexion*. Dans ce cas, elle sera « éclatée » dans les paramètres individuels de connexion. La valeur est considérée être une chaîne de connexion, plutôt qu'un nom de base, si elle contient un signe égal (`=`) ou si elle commence avec un désignateur de schéma URI. (Vous trouverez plus de détails sur les formats de chaîne de connexion dans Section 34.1.1.) Seule la première occurrence de *`dbname`* est traitée de cette façon ; tout paramètre *`dbname`* supplémentaire est traité comme un simple nom de base.

En général, les tableaux de paramètres sont traités du début à la fin. Si un mot clé est répété, la dernière valeur (non `NULL` ou vide) est utilisée. Cette règle s'applique en particulier quand un mot clé trouvé dans la chaîne de connexion est en conflit avec un mot clé apparaissant dans le tableau `keywords`. De ce fait, le développeur peut déterminer si les entrées du tableau peuvent surcharger ou être surchargées par des valeurs prises dans la chaîne de connexion. Les entrées du tableau apparaissant avant une entrée *`dbname`* éclatée peuvent être surchargées par les champs de la chaîne de connexion et, à leur tour,ces champs peuvent être surchargés par des entrées du tableau apparaissant après *`dbname`* (mais, encore une fois, seulement si ces entrées fournissent des valeurs non vides).

Après avoir traité toutes les entrées de tableau et toute chaîne de connexion éclatée, tous les paramètres de connexion restants non configurés sont remplis avec leur valeurs par défaut. Si une variable d'environnement d'un paramètre non configuré (voir Section 34.15) est configuré, sa valeur est utilisée. Si la variable d'environnement n'est pas configurée, alors la valeur par défaut interne du paramètre est utilisée.

`PQconnectdb`

Établit une nouvelle connexion à un serveur de bases de données.

```
PGconn *PQconnectdb(const char *conninfo);
```

Cette fonction ouvre une nouvelle connexion à la base de données en utilisant les paramètres pris à partir de la chaîne `conninfo`.

La chaîne passée peut être vide pour utiliser tous les paramètres par défaut ou elle peut contenir un ou plusieurs paramètres, séparés par des espaces blancs. Elle peut aussi contenir une URI. Voir Section 34.1.1 pour les détails.

`PQsetdbLogin`

Crée une nouvelle connexion sur le serveur de bases de données.

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

C'est le prédécesseur de `PQconnectdb` avec un ensemble fixe de paramètres. Cette fonction a les mêmes fonctionnalités sauf que les paramètres manquants seront toujours initialisés avec leur valeurs par défaut. Écrire `NULL` ou une chaîne vide pour un de ces paramètres fixes dont vous souhaitez utiliser la valeur par défaut.

Si *`dbName`* contient un signe `=` ou a un préfixe URI de connexion valide, il est pris pour une chaîne *`conninfo`* exactement de la même façon que si elle était passée à `PQconnectdb`, et le reste des paramètres est ensuite appliqué comme spécifié dans `PQconnectdbParams`.

`pgtty` n'est plus utilisé et toute valeur passée sera ignorée.

`PQsetdb`

Crée une nouvelle connexion sur le serveur de bases de données.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

C'est une macro faisant appel à `PQsetdbLogin` avec des pointeurs nuls pour les paramètres *`login`* et *`pwd`*. Elle est fournie pour la compatibilité avec les très vieux programmes.

`PQconnectStartParams`
`PQconnectStart`
`PQconnectPoll`

Crée une connexion au serveur de bases de données d'une façon non bloquante.

```
PGconn *PQconnectStartParams(const char * const *keywords,
                             const char * const *values,
                              int expand_dbname);

        PGconn *PQconnectStart(const char *conninfo);

        PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

Ces trois fonctions sont utilisées pour ouvrir une connexion au serveur de bases de données d'une façon telle que le thread de votre application n'est pas bloqué sur les entrées/sorties distantes en demandant la connexion. Le but de cette approche est que l'attente de la fin des entrées/sorties peut se faire dans la boucle principale de l'application plutôt qu'à l'intérieur de `PQconnectdbParams` ou `PQconnectdb`, et donc l'application peut gérer des opérations en parallèle à d'autres activités.

Avec PQconnectStartParams, la connexion à la base de données est faite en utilisant les paramètres à partir des tableaux keywords et values, et contrôlée par expand_dbname, comme décrit dans PQconnectdbParams.

Avec PQconnectStart, la connexion à la base de données est faite en utilisant les paramètres provenant de la chaîne conninfo comme décrit ci-dessus pour PQconnectdb.

Ni PQconnectStartParams ni PQconnectStart ni PQconnectPoll ne bloqueront, aussi longtemps qu'un certain nombre de restrictions est respecté :

- Le paramètre hostaddr doit être utilisé de façon appropriée pour empêcher l'exécution de requêtes DNS. Voir la documentation de ce paramètre sur Section 34.1.2 pour plus de détails.
- Si vous appelez PQtrace, assurez-vous que l'objet de flux dans lequel vous enregistrez les traces ne bloquera pas.
- Assurez-vous que le socket soit dans l'état approprié avant d'appeler PQconnectPoll, comme décrit ci-dessous.

Pour commencer une demande de connexion non bloquante, appelez PQconnectStart ou PQconnectStartParams. Si le résultat est nul, alors libpq a été incapable d'allouer une nouvelle structure PGconn. Sinon, un pointeur valide vers une structure PGconn est renvoyé (bien qu'il ne représente pas encore une connexion valide vers la base de données). Au retour de PQconnectStart, appelez PQstatus(conn). Si le résultat vaut CONNECTION_BAD, la tentative de connexion a déjà échoué, généralement à cause de paramètres de connexion invalides.

Si PQconnectStart ou PQconnectStartParams réussit, la prochaine étape est d'appeler souvent libpq de façon à ce qu'il continue la séquence de connexion. Utilisez PQsocket(conn) pour obtenir le descripteur de socket sous la connexion à la base de données. (Attention, ne supposez pas que la socket reste identique pour les différents appels à PQconnectPoll.) Du coup, une boucle : si le dernier retour de PQconnectPoll(conn) est PGRES_POLLING_READING, attendez que la socket soit prête pour lire (comme indiqué par select(), poll() ou une fonction système similaire). Puis, appelez de nouveau PQconnectPoll(conn). En revanche, si le dernier retour de PQconnectPoll(conn) est PGRES_POLLING_WRITING, attendez que la socket soit prête pour écrire, puis appelez de nouveau PQconnectPoll(conn). À la première itération, si vous avez encore à appeler PQconnectPoll, continuez comme s'il avait renvoyé PGRES_POLLING_WRITING. Continuez cette boucle jusqu'à ce que PQconnectPoll(conn) renvoie PGRES_POLLING_FAILED, indiquant que la procédure de connexion a échoué ou PGRES_POLLING_OK, indiquant le succès de la procédure de connexion.

À tout moment pendant la connexion, le statut de cette connexion peut être vérifié en appelant PQstatus. Si le résultat est CONNECTION_BAD, alors la procédure de connexion a échoué ; si, au contraire, elle renvoie CONNECTION_OK, alors la connexion est prête. Ces deux états sont détectables à partir de la valeur de retour de PQconnectPoll, décrite ci-dessus. D'autres états pourraient survenir lors (et seulement lors) d'une procédure de connexion asynchrone. Ils indiquent l'état actuel de la procédure de connexion et pourraient être utiles pour fournir un retour à l'utilisateur. Ces statuts sont :

CONNECTION_STARTED

Attente de la connexion à réaliser.

CONNECTION_MADE

Connexion OK ; attente d'un envoi.

CONNECTION_AWAITING_RESPONSE

Attente d'une réponse du serveur.

`CONNECTION_AUTH_OK`

Authentification reçue ; attente de la fin du lancement du moteur.

`CONNECTION_SSL_STARTUP`

Négociation du cryptage SSL.

`CONNECTION_SETENV`

Négociation des paramètres de l'environnement.

`CONNECTION_CHECK_WRITABLE`

Vérification que la connexion est capable de gérer des transactions en écriture.

`CONNECTION_CONSUME`

En train de traiter les messages de réponse restants sur la connexion.

Notez que, bien que ces constantes resteront (pour maintenir une compatibilité), une application ne devrait jamais se baser sur l'apparition de ces états dans un ordre particulier, ou leur survenance tout court, ou sur le fait que le statut fait partie de ces valeurs documentées. Une application pourrait faire quelque chose comme ça :

```
switch(PQstatus(conn))
{
    case CONNECTION_STARTED:
        feedback = "Connexion en cours...";
        break;

    case CONNECTION_MADE:
        feedback = "Connecté au serveur...";
        break;
.
.
.
    default:
        feedback = "Connexion...";
}
```

Le paramètre de connexion `connect_timeout` est ignoré lors de l'utilisation `PQconnectPoll` ; c'est de la responsabilité de l'application de décider quand une période de temps excessive s'est écoulée. Sinon, `PQconnectStart` suivi par une boucle `PQconnectPoll` est équivalent à `PQconnectdb`.

Notez que quand `PQconnectStart` ou `PQconnectStartParams` renvoient un pointeur non NULL, vous devez appeler `PQfinish` quand vous en avez fini pour libérer la structure et tout bloc mémoire qui y est associé. Ceci doit être fait même si la tentative de connexion échoue ou est abandonnée.

PQconndefaults

Renvoie les options de connexion par défaut.

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
```

```
   char   *keyword;   /* Mot clé de l'option */
   char   *envvar;    /* Nom de la variable d'environnement
équivalente */
   char   *compiled;  /* Valeur par défaut interne */
   char   *val;       /* Valeur actuelle de l'option ou NULL */
   char   *label;     /* Label du champ pour le dialogue de
connexion */
   char   *dispchar;  /* Indique comment afficher ce champ
                         dans un dialogue de connexion. Les
valeurs sont :
                         ""        Affiche la valeur entrée
sans modification
                         "*"       Champ de mot de passe -
cache la valeur
                         "D"       Option de débogage - non
affiché par défaut
                      */
   int     dispsize;  /* Taille du champ en caractère pour le
dialogue */
} PQconninfoOption;
```

Renvoie un tableau d'options de connexion. Ceci pourrait être utilisé pour déterminer toutes les options possibles de `PQconnectdb` et leur valeurs par défaut. La valeur de retour pointe vers un tableau de structures `PQconninfoOption` qui se termine avec une entrée utilisant un pointeur nul pour `keyword`. Le pointeur null est renvoyé si la mémoire n'a pas pu être allouée. Notez que les valeurs par défaut actuelles (champs `val`) dépendront des variables d'environnement et d'autres contextes. Un fichier de service manquant ou non valide sera ignoré de manière silencieuse. Les demandeurs doivent traiter les données des options de connexion en lecture seule.

Après le traitement du tableau d'options, libérez-le en le passant à la fonction `PQconninfoFree`. Si cela n'est pas fait, un petit groupe de mémoire est perdu à chaque appel de `PQconndefaults`.

`PQconninfo`

Renvoie les options de connexion utilisées par une connexion en cours.

```
PQconninfoOption *PQconninfo(PGconn *conn);
```

Renvoie un tableau des options de connexion. Cette fonction peut être utilisée pour déterminer les valeurs de toutes les options de `PQconnectdb` qui ont été utilisées pour se connecter au serveur. La valeur renvoyée pointe vers un tableau de structures `PQconninfoOption` qui se termine avec une entrée possédant un pointeur `keyword` nul. Toutes les notes ci-dessus pour `PQconndefaults` s'appliquent aussi au résultat de `PQconninfo`.

`PQconninfoParse`

Renvoie les options de connexions analysées d'après la chaîne de connexion fournie.

```
PQconninfoOption *PQconninfoParse(const char *conninfo,
char **errmsg);
```

Analyse une chaîne de connexion et renvoie les options résultantes dans un tableau ; renvoie `NULL` si un problème a été détecté avec la chaîne de connexion. Ceci peut être utilisé pour déterminer les options de `PQconnectdb` dans la chaîne de connexion fournie. La valeur de retour pointe

vers un tableau de structures PQconninfoOption et termine avec une entrée ayant un pointeur keyword nul.

Toutes les options légales seront présentes dans le tableau en résultat mais le PQconninfoOption pour toute option absente de la chaîne de connexion aura sa valeur (val) configurée à NULL ; les valeurs par défaut ne sont pas utilisées.

Si errmsg n'est pas NULL, alors *errmsg est configuré à NULL en cas de succès et sinon à un message d'erreur (alloué via un appel à malloc) expliquant le problèm. (Il est aussi possible pour *errmsg d'être configuré à NULL et la fonction de renvoyer NULL ; cela indique un cas de mémoire épuisée.)

Après avoir traité le tableau des options, libérez-le en le passant à PQconninfoFree. Si ce n'est pas fait, de la mémoire sera perdu à chaque appel à PQconninfoParse. Réciproquement, si une erreur survient et que errmsg n'est pas NULL, assurez-vous de libérer la chaîne d'erreur en utilisant PQfreemem.

PQfinish

Ferme la connexion au serveur. Libère aussi la mémoire utilisée par l'objet PGconn.

```
void PQfinish(PGconn *conn);
```

Notez que même si la connexion au serveur a échoué (d'après l'indication de PQstatus), l'application devrait appeler PQfinish pour libérer la mémoire utilisée par l'objet PGconn. Le pointeur PGconn ne doit pas être encore utilisé après l'appel à PQfinish.

PQreset

Réinitialise le canal de communication avec le serveur.

```
void PQreset(PGconn *conn);
```

Cette fonction fermera la connexion au serveur et tentera l'établissement d'une nouvelle connexion au même serveur en utilisant tous les paramètres utilisés précédemment. Ceci pourrait être utile en cas de récupération après une perte de connexion.

PQresetStart
PQresetPoll

Réinitialise le canal de communication avec le serveur d'une façon non bloquante.

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

Ces fonctions fermeront la connexion au serveur et tenteront d'établir une nouvelle connexion sur le même serveur, en utilisant tous les paramètres précédemment utilisés. Ceci peut être utile pour revenir à un état normal après une erreur si une connexion est perdue. Ces fonctions diffèrent de PQreset (ci-dessus) dans le fait qu'elles agissent d'une façon non bloquante. Ces fonctions souffrent des mêmes restrictions que PQconnectStartParams, PQconnectStart et PQconnectPoll.

Pour lancer une réinitialisation de la connexion, exécutez PQresetStart. Si cette fonction 0, la réinitialisation a échoué. Si elle renvoie 1, récupérez le résultat de la réinitialisation en utilisant PQresetPoll exactement de la même façon que vous auriez créé la connexion en utilisant PQconnectPoll.

`PQpingParams`

`PQpingParams` renvoie le statut du serveur. Elle accepte les mêmes paramètres de connexion que ceux de la fonction `PQconnectdbParams`, décrite ci-dessus. Néanmoins, il n'est pas nécessaire de fournir les bons nom d'utilisateur, mot de passe, ou nom de base de données pour obtenir le statut du serveur. Néanmoins, si des valeurs incorrectes sont fournies, le serveur tracera une tentative échouée de connexion.

```
PGPing PQpingParams(const char * const *keywords,
            const char * const *values,
            int expand_dbname);
```

La fonction renvoie une des valeurs suivantes :

`PQPING_OK`

Le serveur est en cours d'exécution et semble accepter les connexions.

`PQPING_REJECT`

Le serveur est en cours d'exécution mais est dans un état qui interdit les connexions (démarrage, arrêt, restauration après crash).

`PQPING_NO_RESPONSE`

Le serveur n'a pas pu être contacté. Cela pourrait indiquer que le serveur n'est pas en cours d'exécution ou qu'il y a un problème avec les paramètres de connexion donnés (par exemple un mauvais numéro de port). Cela peut aussi indiquer un problème de connexion réseau (par exemple un pare-feu qui bloque la demande de connexion).

`PQPING_NO_ATTEMPT`

Aucune tentative n'a été faite pour contacter le serveur à cause des paramètres fournis erronnés ou à cause d'un problème au niveau client (par exemple un manque mémoire).

`PQping`

`PQping` renvoie l'état du serveur. Elle accepte les mêmes paramètres de connexion que ceux de la fonction `PQconnectdb`, décrite ci-dessus. Néanmoins, il n'est pas nécessaire de fournir les bons nom d'utilisateur, mot de passe, ou nom de base de données pour obtenir le statut du serveur; toutefois, si des valeurs incorrectes sont fournies, le serveur tracera une tentative de connexion en échec.

```
PGPing PQping(const char *conninfo);
```

Les valeurs de retour sont les mêmes que pour `PQpingParams`.

`PQsetSSLKeyPassHook_OpenSSL`

`PQsetSSLKeyPassHook_OpenSSL` permet à une application de passer outre la gestion par défaut des fichiers clés de certificats client de la libpq en utilisant sslpassword ou une invite interactive.

```
void PQsetSSLKeyPassHook_OpenSSL(PQsslKeyPassHook_OpenSSL_type
 hook);
```

L'application passe un pointeur a une fonction callback avec la signature :

```
int callback_fn(char *buf, int size, PGconn *conn);
```

que `libpq` appelera *à la place du* gestionnaire de la fonction par défaut `PQdefaultSSLKeyPassHook_OpenSSL`. Le callback devra déterminer le mot de passe pour la clé et la copier dans le résultat tampon `buf` de taille `size`. La chaîne dans *`buf`* doit être terminée par null. Le callback doit retourner la longueur du mot de passe stockée dans *`buf`* en excluant le caractère terminateur nul. En cas d'échec, le callback devra affecter `buf[0] = '\0'` et retourner 0. Voir `PQdefaultSSLKeyPassHook_OpenSSL` dans le code source de `libpq` pour un exemple.

Si l'utilisateur spécifie un emplacement explicite de la clef, son chemin sera dans `conn->sslkey` quand le callback est appelé. Il sera vide si le chemin de clé par défaut est utilisé. Pour les clés qui sont spécifiques au moteur, celui-ci est responsable de leurs implémentations qu'elles utilisent le callback mot de passe OpenSSL ou qu'elles définissent leur propre gestionnaire.

Le callback de l'application peut choisir de déléguer les cas non gérés à `PQdefaultSSLKeyPassHook_OpenSSL`, ou l'appeler en premier et essayer autre chose si elle retourne 0, ou complétement la passer outre.

Le callback *ne doit pas* éviter le contrôle de flux normal avec quelques exceptions, `longjmp(...)`, etc. Il doit retourner normallement.

`PQgetSSLKeyPassHook_OpenSSL`

`PQgetSSLKeyPassHook_OpenSSL` retourne le point d'ancrage de la fonction de mot de passe de clé de certificat client courant, ou `NULL` si aucun n'a été affecté.

```
PQsslKeyPassHook_OpenSSL_type PQgetSSLKeyPassHook_OpenSSL(void);
```

34.1.1. Chaînes de connexion

Plusieurs fonctions de la bibliothèque libpq analysent une chaîne donnée par l'utilisateur pour obtenir les paramètres de connexion. Deux formats sont acceptés pour ces chaînes : les chaînes mot clé/valeur et les URI. Les URI respectent généralement la RFC 3986[1], sauf que les chaînes de connexions multi hôtes sont autorisées, comme décrit ci-dessous.

34.1.1.1. Chaînes de connexion clé/valeur

Dans le format mot clé/valeur, chaque configuration de paramètre se présente sous la forme *`mot clé = valeur`* avec des espaces entre les paramètres. Les espaces autour du signe égal sont optionnels. Pour écrire une valeur vide ou une valeur contenant des espaces, il est nécessaires de l'entourer de guillemets simples, par exemple `clé = 'une valeur'`. Les guillemets simples et les antislashs compris dans une valeur doivent être échappés par un antislash, comme ceci `\'` et ceci `\\`.

Exemple :

```
host=localhost port=5432 dbname=mabase connect_timeout=10
```

Les mots clés reconnus pour les paramètres sont listés dans Section 34.1.2.

[1] https://datatracker.ietf.org/doc/html/rfc3986

34.1.1.2. URI de connexion

La forme générale pour une URI de connexion est :

```
      postgresql://[utilisateur[:mot_de_passe]@][hote][:port][,...]
[/nom_base][?param1=valeur1&...]
      postgresql://[spec_utilisateur@][spec_hote][/nom_base]
[?spec_param]

où spec_utilisateur vaut :

utilisateur[:motdepasse]

et spec_hote vaut :

[hote][:port][,...]

et spec_param vaut :

nom=valeur[&...]
```

Le désignateur d'URI peut être soit `postgresql://` soit `postgres://`. Chacune des parties restantes de l'URI est optionnelle. Les exemples suivants montrent des syntaxes valides pour l'URI :

```
 postgresql://
 postgresql://localhost
 postgresql://localhost:5433
 postgresql://localhost/ma_base
 postgresql://utilisateur@localhost
 postgresql://utilisateur:secret@localhost
 postgresql://autre@localhost/autre_base?
connect_timeout=10&application_name=mon_appli
 postgresql://host1:123,host2:456/somedb?
target_session_attrs=any&application_name=myapp
```

Les valeurs qui apparaîtraient normalement dans la partie hiérarchique de l'URI peuvent être aussi données sous la forme de paramètres nommés. Par exemple :

```
 postgresql:///ma_base?host=localhost&port=5433
```

Tous les paramètres nommés doivent correspondre aux mots clés listés dans Section 34.1.2, sauf que, pour la compatibilité avec les URI des connexions JDBC, les parties avec of `ssl=true` sont traduites en `sslmode=require`.

L'URI de connexion doit être encodée avec percent-encoding[2] si elle inclut des symboles avec des significations spéciales dans toutes les parties. Voici un exemple où un signe égal (`=`) est remplacé avec `%3D` et un caractère espace avec `%20` :

```
postgresql://user@localhost:5433/mydb?options=-c
%20synchronous_commit%3Doff
```

[2] https://datatracker.ietf.org/doc/html/rfc3986#section-2.1

La partie host peut être soit un nom d'hôte soit une adresse IP. Pour indiquer une adresse IPv6, il est nécessaire de l'englober dans des crochets :

```
postgresql://[2001:db8::1234]/database
```

La partie host est interprétée de la façon décrite pour le paramètre host. En particulier, une connexion par socket de domaine Unix est choisi si la partie host est vide ou ressemble à un nom de chemin absolu. Dans tous les autres cas, une connexion TCP/IP est démarrée. Cependant, notez que le slash est un caractère réservé dans la partie hiérarchique de l'URI. Donc, pour indiquer un répertoire non standard pour la socket de domaine Unix, il faut soit omettre d'indiquer la partie host dans l'URI et l'indiquer en tant que paramètre nommé, soit encoder le chemin dans le composant host de l'URI :

```
postgresql:///dbname?host=/var/lib/postgresql
postgresql://%2Fvar%2Flib%2Fpostgresql/dbname
```

Il est possible de spécifier plusieurs composants hôte, chacun avec un port optionnel, dans une seule URI. Une URI de la forme `postgresql://host1:port1,host2:port2,host3:port3/` est équivalent à une chaîne de connexion de la forme `host=host1,host2,host3 port=port1,port2,port3`. Comme indiqué plus en détails plus bas, chaque hôte sera testé à son tour jusqu'à ce qu'une connexion soit établie avec succès.

34.1.1.3. Spécifier plusieurs hôtes

Il est possible de spécifier plusieurs hôtes où se connecter. Ils sont essayés dans l'ordre donné. Dans un format clé/valeur, les options `host`, `hostaddr`, and `port` acceptent des listes de valeurs séparées par une virgule. Le même nombre d'éléments doit être donné dans chaque option qui est spécifiée pour que le premier élément dans `hostaddr` au premier nom d'hôte, le second `hostaddr` correspond au second nom d'hôte, et ainsi de suite. Seule exception pour l'option `port`, si un seul port est spécifié, il est utilisé pour tous les hôtes.

Dans le format de connexion URI, vous pouvez lister plusieurs paires `hote:port` séparées par des virgules dans le composant `host` de l'URI.

Quel que soit le format, un simple nom d'hôte peut aussi se traduire en plusieurs adresses réseau. Un exemple habituel est un hôte qui a à la fois une adresse IPv4 et une adresse IPv6.

Quand plusieurs hôtes sont indiqués ou quand un nom d'hôte est converti en plusieurs adresses, tous les hôtes et adresses seront essayés dans l'ordre jusqu'à ce qu'une connexion réussisse. Si aucun des hôtes ne peut être atteint, la connexion échoue. Si la connexion réussit mais que l'authentification échoue, les autres hôtes de la liste ne seront pas testés.

Si un fichier de mots de passe est utilisé, vous pouvez avoir plusieurs mots de passe pour des hôtes différents. Toutes les autres options de connexion sont identiques pour chaque hôte de la liste. Par exemple, il n'est pas possible d'indiquer un nom d'utilisateur différent pour les différents hôtes.

34.1.2. Mots clés de la chaîne de connexion

Les mots clés actuellement reconnus sont :

`host`

Nom de l'hôte où se connecter. Si un nom d'hôte ressemble à un nom de chemin absolu, il spécifie une communication par domaine Unix plutôt qu'une communication TCP/IP ; la valeur est le nom du répertoire où le fichier socket est stocké. (Sur Unix, un chemin absolu commence par un slash. Sur windows, les chemins débutant par la lettre du lecteur sont aussi reconnus.) Si le nom de l'#ote commence avec un `@`, il est pris pour un socket de domaine Unix dans l'espace de

nom abstrait (actuellement supporté sur Linux et Windows). Par défaut, quand `host` n'est pas spécifié ou est vide, il s'agit d'une communication par socket de domaine Unix dans `/tmp` (ou tout autre répertoire de socket spécifié lors de la construction de PostgreSQL). Sur Windows et sur les machines sans socket de domaine Unix, la valeur par défaut est de se connecter à `localhost`.

Une liste de noms d'hôtes séparés par des virgules est aussi acceptée, auquel cas chaque nom d'hôte est testé dans l'ordre ; un élément vide dans la liste implique le comportement par défaut comme décrit plus haut. Voir Section 34.1.1.3 pour les détails.

`hostaddr`

Adresse IP numérique de l'hôte de connexion. Elle devrait être au format d'adresse standard IPv4, par exemple `172.28.40.9`. Si votre machine supporte IPv6, vous pouvez aussi utiliser ces adresses. La communication TCP/IP est toujours utilisée lorsqu'une chaîne non vide est spécifiée pour ce paramètre. Si ce paramètre n'est pas spécifié, la valeur pour `host` sera résolue pour trouver l'adresse IP correspondante -- ou, si `host` spécifie une adresse IP, cette valeur sera utilisée directement.

Utiliser `hostaddr` permet à l'application d'éviter une résolution de nom, ce qui pourrait être important dans les applications avec des contraintes de temps. Cependant, un nom d'hôte est requis pour les méthodes d'authentification GSSAPI ou SSPI, ainsi que pour la vérification de certificat SSL en `verify-full`. Les règles suivantes sont observées :

- Si `host` est indiqué sans `hostaddr`, une résolution du nom de l'hôte est lancée. (Si vous utilisez `PQconnectPoll`, la recherche survient quand `PQconnectPoll` considère le nom d'hôte pour la première fois, et cela pourrait être la cause d'un blocage de `PQconnectPoll` pendant une bonne durée de temps.)

- Si `hostaddr` est indiqué sans `host`, la valeur de `hostaddr` fournit l'adresse réseau de l'hôte. La tentative de connexion échouera si la méthode d'authentification nécessite un nom d'hôte.

- Si `host` et `hostaddr` sont spécifiés, la valeur de `hostaddr` donne l'adresse réseau de l'hôte. La valeur de `host` est ignorée sauf si la méthode d'authentification la réclame, auquel cas elle sera utilisée comme nom d'hôte.

Notez que l'authentification a de grandes chances d'échouer si `host` n'est pas le nom du serveur à l'adresse réseau `hostaddr`. Et quand `host` et `hostaddr` sont tous deux spécifiés, seul `host` est utilisé pour identifier la connexion dans un fichier de mots de passe (voir la Section 34.16).

Une liste d'adresses `hostaddr` séparées par des virgules est aussi acceptée, auquel cas chaque hôte est essayé dans cet ordre. Un élément vide dans la liste mène à l'utilisation du nom d'hôte correspondant, ou, s'il est vide aussi, de la valeur par défaut. Voir Section 34.1.1.3 pour plus de détails.

Sans un nom ou une adresse d'hôte, libpq se connectera en utilisant un socket local de domaine Unix. Sur Windows et sur des machines sans sockets de domaine Unix, il tentera une connexion sur `localhost`.

`port`

Numéro de port pour la connexion au serveur ou extension du nom de fichier pour des connexions de domaine Unix. Si plusieurs hôtes sont indiquées dans les paramètres `host` ou `hostaddr`, ce paramètre spécifie une liste de ports séparés par des virgules et de même taille que la liste des hôtes, ou bien il peut préciser un seul port à tester pour tous les hôtes. Une chaîne vide, ou un élément vide de la liste, indique le numéro de port par défaut établi à la compilation de PostgreSQL.

`dbname`

Nom de la base de données. Par défaut, la même que le nom utilisateur. Dans certains contextes, la valeur est vérifiée pour les formats étendus ; voir Section 34.1.1 pour plus d'informations.

`user`

Nom de l'utilisateur PostgreSQL qui se connecte. Par défaut, il s'agit du même nom que l'utilisateur système lançant l'application.

`password`

Mot de passe à utiliser si le serveur demande une authentification par mot de passe.

`passfile`

Spécifie le nom du fichier utilisé pour stocker les mots de passe (voir Section 34.16). La valeur par défaut est `~/.pgpass`, ou `%APPDATA%\postgresql\pgpass.conf` sur Microsoft Windows. (Aucune erreur n'est levée si le fichier n'existe pas.)

`channel_binding`

Cette option contrôle l'utilisation cliente de la liaison de canal. Une valeur `require` signifie que la connexion doit toujours employer la liaison de canal, `prefer` signifie que le client doit choisir la liaison de canal si disponible, et `disable` empêche l'utilisation de liaison de canal. Le défaut est `prefer` si PostgreSQL est compilé avec le support SSL.

La liaison de canal est une méthode permettant au serveur de s'authentifier vers un client. Elle est seulement supportée sur des connexions SSL avec PostgreSQL 11 ou des versions ultérieures utilisant la méthode d'authentification `SCRAM`.

`connect_timeout`

Temps d'attente maximum lors d'une connexion, en secondes (écrit sous la forme d'un entier décimal, par exemple `10`). La valeur zéro, une valeur négative ou sans valeur indique une attente infinie. Le délai minimal autorisé est 2 secondes, de ce fait la valeur `1` est interprétée comme `2`. Ce délai s'applique séparément pour chaque nom d'hôte ou adresse IP. Par exemple, si vous indiquez deux hôtes et que le paramètre `connect_timeout` vaut 5, chaque hôte sera en timeout si aucune connexion n'est réalisée en 5 secondes, donc le temps total passé à attendre une connexion peut monter jusqu'à 10 secondes.

`client_encoding`

Ceci configure le paramètre `client_encoding` pour cette connexion. En plus des valeurs acceptées par l'option correspondante du serveur, vous pouvez utiliser `auto` pour déterminer le bon encodage à partir de la locale courante du client (variable d'environnement `LC_CTYPE` sur les systèmes Unix).

`options`

Spécifie les options en ligne de commande à envoyer au serveur à l'exécution. Par exemple, en le configurant à `-c geqo=off`, cela configure la valeur de la session pour le paramètre `geqo` à `off`. Les espaces à l'intérieur de cette chaîne sont considérés comme séparateurs d'arguments, sauf si ils sont échappés avec le caractère d'échappement `\` ; écrivez `\\` pour obtenir le caractère d'échappement lui-même. Pour une discussion détaillée des options disponibles, voir Chapitre 20.

`application_name`

Précise une valeur pour le paramètre de configuration application_name.

`fallback_application_name`

Indique une valeur de secours pour le paramètre de configuration application_name. Cette valeur sera utilisée si aucune valeur n'est donnée à `application_name` via un paramètre de connexion ou la variable d'environnement `PGAPPNAME`. L'indication d'un nom de secours est utile pour les programmes outils génériques qui souhaitent configurer un nom d'application par défaut mais permettent sa surcharge par l'utilisateur.

`keepalives`

Contrôle si les paramètres TCP keepalives côté client sont utilisés. La valeur par défaut est de 1, signifiant ainsi qu'ils sont utilisés. Vous pouvez le configurer à 0, ce qui aura pour effet de les désactiver si vous n'en voulez pas. Ce paramètre est ignoré pour les connexions réalisées via un socket de domaine Unix.

`keepalives_idle`

Contrôle le nombre de secondes d'inactivité après lequel TCP doit envoyer un message keepalive au server. Une valeur de zéro utilise la valeur par défaut du système. Ce paramètre est ignoré pour les connexions réalisées via un socket de domaine Unix ou si les paramètres keepalives sont désactivés. Ce paramètre est uniquement supporté sur les systèmes où les options `TCP_KEEPIDLE` ou une option socket équivalente sont disponibles et sur Windows ; pour les autres systèmes, ce paramètre n'a pas d'effet.

`keepalives_interval`

Contrôle le nombre de secondes après lequel un message TCP keepalive doit être retransmis si le serveur ne l'a pas acquitté. Une valeur de zéro utilise la valeur par défaut du système. Ce paramètre est uniquement supporté sur les systèmes où l'option `TCP_KEEPINTVL` ou une option socket équivalente est disponible et sur Windows ; pour les autres systèmes, ce paramètre n'a pas d'effet.

`keepalives_count`

Contrôle le nombre de messages TCP keepalive pouvant être perdus avant que la connexion du client au serveur ne soit considérée comme perdue. Une valeur de zéro utilise la valeur par défaut du système. Ce paramètre est uniquement supporté sur les systèmes où l'option `TCP_KEEPCNT` ou une option socket équivalente est disponible et sur Windows ; pour les autres systèmes, ce paramètre n'a pas d'effet.

`tcp_user_timeout`

Contrôle le nombre de millisecondes durant lesquelles les données transmises peuvent rester non acquittées avant que la connexion soit fermée de force. Une valeur à zéro indique d'utiliser la valeur par défaut du système. Ce paramètre est ignoré pour les connexions initiées via socket de domaine Unix. Il est seulement pris en charge sur les systèmes où `TCP_USER_TIMEOUT` est disponible ; sur les autres systèmes, il n'a pas d'effet.

`replication`

Cette option détermine si la connexion doit utiliser le protocole de réplication au lieu du protocole normal. C'est ce qu'utilisent les connexions de réplication de PostgreSQL, ainsi que des outils comme pg_basebackup, mais il peut aussi être utilisé par des applications tierces. Pour une description du protocole de réplication, consulter Section 55.4

Les valeurs suivantes, non sensibles à la casse, sont supportées :

`true`, `on`, `yes`, `1`

La connexion passe en mode réplication physique.

`database`

La connexion passe en mode réplication logique, se connectant à la base spécifiée par le paramètre `dbname`.

`false`, `off`, `no`, `0`

On utilise la connexion habituelle, ce qui est le comportement par défaut.

En mode réplication physique ou logique, seul le protocole simple peut être utilisé.

`gssencmode`

Cette option détermine si une connexion TCP/IP sécurisée avec GSS sera négociée avec le serveur, et si oui avec quelle priorité. Il y a trois modes :

`disable`

essaie uniquement une connexion non chiffrée avec GSSAPI.

`prefer` (par défaut)

si des informations d'authentification GSSAPI sont présentes (c'est-à-dire, dans un cache d'authentification), essaie d'abord une connexion chiffrée avec GSSAPI ; si elle échoue ou qu'il n'y a pas les informations d'authentification, essaie une connexion non chiffrée avec GSSAPI. C'est le comportement par défaut quand PostgreSQL a été compilé avec le support de GSSAPI.

`require`

essaie uniquement une connexion chiffrée avec GSSAPI.

`gssencmode` est ignoré pour les connexions par socket du domaine Unix. Si PostgreSQL est compilé sans le support de GSSAPI, l'utilisation de l'option `require` provoquera une erreur, alors que `prefer` sera acceptée mais, dans ce cas, libpq n'essaiera pas d'initier une connexion chiffrée avec GSSAPI.

`sslmode`

Cette option détermine si ou avec quelle priorité une connexion TCP/IP SSL sécurisée sera négociée avec le serveur. Il existe six modes :

`disable`

essaie seulement une connexion non SSL

`allow`

essaie en premier lieu une connexion non SSL ; si cette tentative échoue, essaie une connexion SSL

`prefer` (par défaut)

essaie en premier lieu une connexion SSL ; si cette tentative échoue, essaie une connexion non SSL

`require`

essaie seulement une connexion SSL. Si un certificat racine d'autorité (CA) est présent, vérifie le certificat de la même façon que si `verify-ca` était spécifié

`verify-ca`

essaie seulement une connexion SSL et vérifie que le certificat client est créé par une autorité de certification (CA) de confiance

`verify-full`

essaie seulement une connexion SSL, vérifie que le certificat client est créé par un CA de confiance et que le nom du serveur correspond bien à celui du certificat

Voir Section 34.19 pour une description détaillée du fonctionnement de ces options.

`sslmode` est ignoré pour la communication par socket de domaine Unix. Si PostgreSQL est compilé sans le support de SSL, l'utilisation des options `require`, `verify-ca` et `verify-full` causera une erreur, alors que les options `allow` et `prefer` seront acceptées, bien qu'en fait libpq n'essaiera pas de négocier une connexion SSL.

Notez que si le chiffrement GSSAPI est possible, il sera utilisé de préférence au chiffrement SSL, quelque soit la valeur de `sslmode`. Pour forcer l'utilisation du chiffrement SSL dans un environnement qui dispose d'une infrastructure GSSAPI fonctionnelle (tel qu'un serveur Kerberos), configurez aussi `gssencmode` à `disable`.

`requiressl`

Cette option est obsolète et remplacée par l'option `sslmode`.

Si initialisée à 1, une connexion SSL au serveur est requise (ce qui est équivalent à un `sslmode require`). libpq refusera alors de se connecter si le serveur n'accepte pas une connexion SSL. Si initialisée à 0 (la valeur par défaut), libpq négociera le type de connexion avec le serveur (équivalent à un `sslmode prefer`). Cette option est seulement disponible si PostgreSQL est compilé avec le support SSL.

`sslcompression`

Si ce paramètre vaut 1, les données envoyées sur des connexions SSL seront compressées. S'il vaut 0, la compression sera désactivée. Le défaut est 0. Ce paramètre est ignoré pour une connexion sans SSL.

De nos jours, la compression SSL est considérée non sûre et son usage n'est plus recommandé. OpenSSL 1.1.0 désactive la compression par défaut, et de nombreux systèmes d'exploitation la désactive aussi dans des versions précédentes, donc activer ce paramètre n'aura aucun effet si le serveur n'accepte pas la compression. PostgreSQL 14 désactive totalement la compression dans le moteur.

Si la sécurité n'est pas un souci majeur, la compression peut améliorer le débit si le réseau est le goulot d'étranglement. Désactiver la compression peut améliorer le temps de réponse et le débit si le processeur est le facteur limitant.

`sslcert`

Ce paramètre indique le nom du fichier du certificat SSL client, remplaçant le fichier par défaut, `~/.postgresql/postgresql.crt`. Ce paramètre est ignoré si la connexion n'utilise pas SSL.

`sslkey`

Ce paramètre indique l'emplacement de la clé secrète utilisée pour le certificat client. Il peut soit indiquer un nom de fichier qui sera utilisé à la place du fichier `~/.postgresql/postgresql.key` par défaut, soit indiquer une clé obtenue par un « moteur » externe (les moteurs sont des modules chargeables d'OpenSSL). La spécification d'un moteur externe devrait consister en un nom de moteur et un identifiant de clé spécifique au moteur, les deux séparés par une virgule. Ce paramètre est ignoré si la connexion n'utilise pas SSL.

`sslpassword`

Ce paramètre définit le mot de passe pour la clé secrète spécifiée dans `sslkey`, permettant aux clés de certificat privé du client d'être stockées sous format chiffré sur disque même quand la saisie interactive de passphrase n'est pas pratique.

Spécifier ce paramètre avec toute valeur non vide supprime l'invite de saisie `Enter PEM pass phrase:` que OpenSSL émet par défaut quand une clé de certificat client chiffrée est fournie à `libpq`.

Si la clé est non chiffrée, ce paramètre est ignoré. Le paramètre n'a pas d'effet sur les clés spécifiée par les moteurs OpenSSL à moins que le moteur utilise le mécanisme de callback de mot de passe OpenSSL pour les invites de saisie.

Il n'y a pas de variable d'environment équivalente pour cette option, et aucune possibilité pour la déterminer dans `.pgpass`. Elle peut être utilisée dans un fichier de service de définition de connexion. Les utilisateurs avec des usages plus sophistiqués devraient considérer l'utilisation des moteurs OpenSSL et outils comme PKCS#11 ou les périphériques de déchargement de crypto USB.

`sslrootcert`

Ce paramètre indique le nom d'un fichier contenant le ou les certificats de l'autorité de certification SSL (CA). Si le fichier existe, le certificat du serveur sera vérifié. La signature devra appartenir à une de ces autorités. La valeur par défaut est `~/.postgresql/root.crt`.

`sslcrl`

Ce paramètre indique le nom du fichier de la liste de révocation du certificat SSL serveur. Les certificats listés dans ce fichier, s'il existe, seront rejetés lors d'une tentative d'authentification avec le certificat du serveur. Si ni sslcrl ni sslcrldir ne sont configurés, ce paramètre est pris pour `~/.postgresql/root.crl`.

`sslcrldir`

Ce paramètre indique le nom du répertoire de la liste de révocation des certificats SSL serveurs (CRL). Les certificats listés dans les fichiers de ce répertoire, s'il existe, seront rejetés lors de tentative d'authentification du certificat du serveur.

Le répertoire doit être préparé avec la commande OpenSSL `openssl rehash` ou `c_rehash`. Voir sa documentation pour les détails.

`sslcrl` et `sslcrldir` peuvent être indiqués ensemble.

`sslsni`

Si ce paramètre est configuré à A (sa valeur par défaut), libpq configure l'extension TLS « Server Name Indication » (SNI) sur les connexions SSL. En configurant ce paramètre à 0, il est désactivé.

Server Name Indication peut être utilisé par les proxies SSL pour router les connexions sans avoir à déchiffrer le flux SSL. (Notez que cela requiert un proxy comprenant le système de handshake du protocole PostgreSQL, pas n'importe quel proxy SSL.) Néanmoins, SNI fait que le nom d'hôte de destination apparaît en clair sur le trafic réseau, donc il pourrait être indésirable dans certains cas.

`requirepeer`

Ce paramètre indique le nom d'utilisateur du serveur auprès du système d'exploitation, par exemple `requirepeer=postgres`. Lors d'une connexion par socket de domaine Unix, si ce paramètre est configuré, le client vérifie au début de la connexion que le processus tourne sous le nom d'utilisateur indiqué ; dans le cas contraire, la connexion échoue avec une erreur. Ce paramètre peut être utilisé pour fournir une authentification serveur similaire à celle disponible pour les certificats SSL avec les connexions TCP/IP. (Notez que, si la socket de domaine Unix est dans `/tmp` ou tout espace autorisé en écriture pour tout le monde, n'importe quel utilisateur peut y mettre un serveur en écoute. Utilisez ce paramètre pour vous assurer que vous êtes connecté à un serveur exécuté par un utilisateur de confiance.) Cette option est seulement supportée par les plateformes où la méthode d'authentification `peer` est disponible ; voir Section 21.9.

`ssl_min_protocol_version`

Ce paramètre spécifie la version minimale de protocole SSL/TLS pour permettre la connexion. Les valeurs valides sont `TLSv1`, `TLSv1.1`, `TLSv1.2` et `TLSv1.3`. Les protocoles supportés

dépendent de la version utilisée de OpenSSL, les plus anciennes versions ne supportant pas les versions de protocole les plus modernes. S'il n'est pas indiqué, la valeur par défaut `TLSv1.2`, ce qui satisfait les bonnes pratiques de l'industrie lors de l'écriture de ce document.

`ssl_max_protocol_version`

Ce paramètre spécifie la version maximale de protocole SSL/TLS pour permettre la connexion. Les valeurs valides sont `TLSv1`, `TLSv1.1`, `TLSv1.2` et `TLSv1.3`. Les protocoles supportés dépendent de la version utilisée de OpenSSL, les anciennes versions ne supportant pas les versions les plus modernes du protocole. S'il n'est pas affecté, ce paramètre est ignoré et la connexion utilisera la version maximale du protocole définie par le backend, si elle existe. Affecter une version maximale du protocole est principalement utile pour tester ou si certains composants ont des problèmes à utiliser un protocole plus récent.

`krbsrvname`

Nom du service Kerberos à utiliser lors de l'authentification avec GSSAPI. Il doit correspondre avec le nom du service spécifié dans la configuration du serveur pour que l'authentification Kerberos puisse réussir. (Voir aussi la Section 21.6.) La valeur par défaut est normalement `postgres`, mais cela peut être changé lors de la compilation de PostgreSQL avec l'option `--with-krb-srvnam` de configure. Dans la plupart des environnements, ce paramètre n'a jamais besoin d'être modifié. Certaines implémentations Kerberos nécessitent un nom de service différent, par exemple Microsoft Active Directory qui réclame que le nom de service soit en majuscules (`POSTGRES`).

`gsslib`

Bibliothèque GSS à utiliser pour l'authentification GSSAPI. Ce paramètre est actuellement ignoré, sauf sur les versions Windows qui incluent la prise en charge de GSSAPI et de SSPI. Dans ce cas, configurer ce paramètre à `gssapi` pour amener la libpq à utiliser la bibliothèque GSSAPI pour l'authentification au lieu de SSPI par défaut.

`service`

Nom du service à utiliser pour des paramètres supplémentaires. Il spécifie un nom de service dans `pg_service.conf` contenant des paramètres de connexion supplémentaires. Ceci permet aux applications de spécifier uniquement un nom de service pour que les paramètres de connexion puissent être maintenus de façon centralisée. Voir Section 34.17.

`target_session_attrs`

Cette option détermine si la session doit avoir certaines propriétés pour être acceptable. C'est typiquement utilisé en combinaison avec plusieurs noms d'hôte pour sélectionner la première alternative acceptable parmi différents hôtes. Il existe six modes :

`any` (default)

toute connexion réussie est acceptable

`read-write`

la session doit accepter les transactions en lecture/écriture par défaut (c'est-à-dire que le serveur ne doit pas être en mode hot standby et le paramètre `default_transaction_read_only` doit être à `off`)

`read-only`

la session ne doit pas accepter les transactions en lecture/écriture par défaut (l'inverse)

`primary`

le serveur ne doit pas être en mode hot standby

`standby`

le serveur doit être en mode hot standby

`prefer-standby`

essaie tout d'abord de trouver un serveur secondaire, mais aucun des hôtes listés n'est un serveur secondaire, alors tente de nouveau dans le mode `any`

34.2. Fonctions de statut de connexion

Ces fonctions sont utilisées pour interroger le statut d'un objet de connexion existant.

> **Astuce**
>
> Les développeurs d'application libpq devraient être attentif à maintenir l'abstraction `PGconn`. Utilisez les fonctions d'accès décrites ci-dessous pour obtenir le contenu de `PGconn`. Référencer les champs internes de `PGconn` en utilisant `libpq-int.h` n'est pas recommandé parce qu'ils sont sujets à modification dans le futur.

Les fonctions suivantes renvoient les valeurs des paramètres utilisés pour la connexion. Ces valeurs sont fixes pour la durée de vie de la connexion. Si une chaîne de connexion multi-hôtes est utilisée, les valeurs de `PQhost`, `PQport` et `PQpass` peuvent changer si une nouvelle connexion est établie en utilisant le même objet `PGconn`. Les autres valeurs sont figées pour la durée de vie de l'objet `PGconn`.

`PQdb`

Renvoie le nom de la base de données de la connexion.

```
char *PQdb(const PGconn *conn);
```

`PQuser`

Renvoie le nom d'utilisateur utilisé pour la connexion.

```
char *PQuser(const PGconn *conn);
```

`PQpass`

Renvoie le mot de passe utilisé pour la connexion.

```
char *PQpass(const PGconn *conn);
```

`PQpass` retournera soit le mot de passe spécifié dans les paramètres de connexion, soit, s'il n'y en avait pas, le mot de passe obtenu depuis le fichier de mots de passe. Dans ce dernier cas, si plusieurs hôtes étaient spécifiés dans les paramètres de connexion, il n'est pas possible de se fier au résultat de `PQpass` jusqu'à l'établissement de la connexion. Le statut de la connexion peut être vérifié avec la fonction `PQstatus`.

`PQhost`

Renvoie le nom d'hôte du serveur utilisé pour la connexion. Cela peut être un nom d'hôte, une adresse IP ou un chemin de répertoire si la connexion est réalisée via un socket Unix. (Le cas du chemin se distingue par le fait que ce sera toujours un chemin absolu commençant par `/`.)

```
char *PQhost(const PGconn *conn);
```

Si les paramètres de connexion `host` and `hostaddr` sont tous les deux précisés, alors `PQhost` retournera `host`. Si seul `hostaddr` a été spécifié, c'est cela qui est retourné. Si plusieurs hôtes sont spécifiés dans les paramètres de connexion, `PQhost` retourne l'hôte à qui l'on s'est effectivement connecté.

`PQhost` retourne `NULL` si l'argument *`conn`* est `NULL`. Sinon, s'il y a une erreur en déterminant l'`host` (peut-être que la connexion n'a pas été complètement établie ou qu'il y a eu une erreur), il retourne une chaîne vide.

Si plusieurs hôtes ont été spécifiés dans les paramètres de connexion, il n'est pas possible de se baser sur le résultat de `PQhost` avant l'établissement de la connexion. Le statut de la connexion peut être vérifié avec la fonction `PQstatus`.

`PQhostaddr`

Renvoie l'adresse IP du serveur de la connexion active. Ca peut être l'adresse résultant de la résolution d'un nom d'hôte, ou une adresse IP fournie par le paramètre `hostaddr`.

```
char *PQhostaddr(const PGconn *conn);
```

`PQhostaddr` renvoie `NULL` si l'argument *`conn`* est `NULL`. Sinon, si une erreur survient en produisant l'information sur l'hôte (peut-être parce que la connexion n'a pas été complètement établie ou qu'elle a subi une erreur), renvoie une chaîne vide.

`PQport`

Renvoie le numéro de port utilisé pour la connexion active.

```
char *PQport(const PGconn *conn);
```

Si de multiples ports étaient spécifiés dans les paramètres de connexion, `PQport` renvoie le port auquel on est effectivement connecté.

`PQport` retourne `NULL` si l'argument *`conn`* est `NULL`. Sinon, s'il y a une erreur en déterminant le `port` (peut-être que la connexion n'a pas été complètement établie ou qu'il y a eu une erreur), il retourne une chaîne vide.

Si plusieurs ports ont été spécifiés dans les paramètres de connexion, il n'est pas possible de se baser sur le résultat de `PQport` avant l'établissement de la connexion. Le statut de la connexion peut être vérifié avec la fonction `PQstatus`.

`PQtty`

Cette fonction ne fait plus rien, mais elle reste disponible pour des raisons de compatibilité ascendante. La fonction renvoie toujours une chaîne vide ou `NULL` si l'argument *`conn`* vaut `NULL`.

```
char *PQtty(const PGconn *conn);
```

`PQoptions`

Renvoie les options en ligne de commande passées lors de la demande de connexion.

```
char *PQoptions(const PGconn *conn);
```

Les fonctions suivantes renvoient des données de statut qui peuvent changer suite à des opérations sur l'objet PGconn.

PQstatus

Renvoie l'état de la connexion.

```
ConnStatusType PQstatus(const PGconn *conn);
```

Le statut peut prendre un certain nombre de valeurs. Néanmoins, deux seulement ne concernent pas les procédures de connexion asynchrone : CONNECTION_OK et CONNECTION_BAD. Une bonne connexion à la base de données a l'état CONNECTION_OK. Une tentative de connexion ayant échoué est signalée par le statut CONNECTION_BAD. D'habitude, un état OK le restera jusqu'à PQfinish mais un échec dans les communications peut résulter en un statut changeant prématurément en CONNECTION_BAD. Dans ce cas, l'application peut essayer de rattraper la situation en appelant PQreset.

Voir l'entrée de PQconnectStartParams, PQconnectStart et de PQconnectPoll à propos des autres codes de statut qui pourraient être renvoyés.

PQtransactionStatus

Renvoie l'état actuel de la transaction du serveur.

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

Le statut peut être PQTRANS_IDLE (actuellement inactif), PQTRANS_ACTIVE (une commande est en cours), PQTRANS_INTRANS (inactif, dans un bloc de transaction valide) ou PQTRANS_INERROR (inactif, dans un bloc de transaction échoué). PQTRANS_UNKNOWN est rapporté si la connexion est mauvaise. PQTRANS_ACTIVE n'est rapporté que si une requête a été envoyée au serveur et n'est pas encore terminée.

PQparameterStatus

Recherche la valeur en cours d'un paramètre du serveur.

```
const char *PQparameterStatus(const PGconn *conn, const char
 *paramName);
```

Certaines valeurs de paramètres sont rapportées par le serveur automatiquement ou lorsque leur valeurs changent. PQparameterStatus peut être utilisé pour interroger ces paramètres. Il renvoie la valeur en cours d'un paramètre s'il est connu et NULL si le paramètre est inconnu.

Les paramètres renvoyées par la version actuelle incluent server_version, server_encoding, client_encoding, application_name, default_transaction_read_only, in_hot_standby, is_superuser, session_authorization, datestyle, IntervalStyle, TimeZone, integer_datetimes et standard_conforming_strings. (server_encoding, TimeZone et integer_datetimes n'étaient pas renvoyés dans les versions antérieures à la 8.0 ; standard_conforming_strings n'était pas renvoyé dans les versions antérieures à la 8.1 ; IntervalStyle n'était pas renvoyé dans les versions antérieures à la 8.4 ; application_name n'était pas renvoyé dans les versions antérieures à la 9.0 ; default_transaction_read_only et in_hot_standby n'étaient pas renvoyés dans les versions antérieures à la 14.). Notez que server_version, server_encoding et integer_datetimes ne peuvent pas changer après le lancement du serveur.

Si aucune valeur n'est indiquée pour `standard_conforming_strings`, les applications peuvent considérer qu'elle vaut `off`, c'est-à-dire que les antislashs sont traités comme des échappements dans les chaînes littérales. De plus, la présence de ce paramètre peut être pris comme une indication que la syntaxe d'échappement de chaîne (`E'...'`) est acceptée.

Bien que le pointeur renvoyé est déclaré `const`, il pointe en fait vers un stockage mutable associé à la structure `PGconn`. Il est déconseillé de supposer que le pointeur restera valide pour toutes les requêtes.

`PQprotocolVersion`

Interroge le protocole interface/moteur lors de son utilisation.

```
int PQprotocolVersion(const PGconn *conn);
```

Les applications peuvent utiliser ceci pour déterminer si certaines fonctionnalités sont supportées. Actuellement, les seules valeurs possible sont 3 (protocole 3.0) ou zéro (mauvaise connexion). La version du protocole ne changera pas après la fin du lancement de la connexion mais il pourrait théoriquement changer lors d'une réinitialisation de la connexion. Le protocole 3.0 est supporté par les versions 7.4 et suivantes de PostgreSQL.

`PQserverVersion`

Renvoie un entier représentant la version du moteur.

```
int PQserverVersion(const PGconn *conn);
```

Les applications peuvent utiliser cette fonction pour déterminer la version du serveur de bases de données où ils sont connectés Le résultat est obtenu en multipliant le numéro de version majeure de la bibliothèque par 10000 et en ajoutant le numéro de version mineure. Par exemple, la version 10.1 renverra 100001, et la version 11.0 renverra 110000. Zéro est renvoyé si la connexion est mauvaise.

Avant la version majeure 10, PostgreSQL utilisait des numéros de version en trois parties, pour lesquelles les deux premières parties représentaient la version majeure. Pour ces versions, `PQlibVersion` utilise deux chiffres pour chaque partie. Par exemple, la version 9.1.5 renverra 90105, et la version 9.2.0 renverra 90200.

De ce fait, pour déterminer la compatibilité de certaines fonctionnalités, les applications devrait diviser le résultat de `PQserverVersion` par 100, et non pas par 10000, pour déterminer le numéro de version majeure logique. Dans toutes les versions, seuls les deux derniers chiffres diffèrent entre des versions mineures (versions correctives).

`PQerrorMessage`

Renvoie le dernier message d'erreur généré par une opération sur la connexion.

```
char *PQerrorMessage(const PGconn* conn);
```

Pratiquement toutes les fonctions libpq initialiseront un message pour `PQerrorMessage` en cas d'échec. Notez que, par convention dans libpq, un résultat non vide de `PQerrorMessage` peut courir sur plusieurs lignes et finira par un retour chariot. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand le pointeur `PGconn` associé sera passé à `PQfinish`. La chaîne résultante n'est pas supposée rester la même pendant les opérations sur la structure `PGconn`.

`PQsocket`

Obtient le descripteur de fichier du socket de la connexion au serveur. Un descripteur valide sera plus grand ou égal à 0 ; un résultat de -1 indique qu'aucune connexion au serveur n'est actuellement

ouverte (ceci ne changera pas lors du fonctionnement habituel, mais pourrait changer lors de la mise en place de la connexion ou lors d'une réinitialisation).

```
int PQsocket(const PGconn *conn);
```

`PQbackendPID`

Renvoie l'identifiant du processus (PID) du serveur gérant cette connexion.

```
int PQbackendPID(const PGconn *conn);
```

Le PID du processus backend est utile pour des raisons de débogage et pour la comparaison avec les messages `NOTIFY` (qui incluent le PID du processus serveur lançant la notification). Notez que le PID appartient à un processus exécuté sur l'hôte du serveur de bases de données et non pas sur l'hôte local !

`PQconnectionNeedsPassword`

Renvoie true (1) si la méthode d'authentification de la connexion nécessite un mot de passe, mais qu'aucun n'est disponible. Renvoie false (0) sinon.

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

Cette fonction peut être utilisée après un échec de connexion pour décider s'il faut demander un mot de passe à l'utilisateur.

`PQconnectionUsedPassword`

Renvoie true (1) si la méthode d'authentification de la connexion a utilisé un mot de passe. Renvoie false (0) sinon.

```
int PQconnectionUsedPassword(const PGconn *conn);
```

Cette fonction peut être utilisée après une connexion, réussie ou en échec, pour détecter si le serveur demande un mot de passe.

Les fonctions ci-dessous renvoient des informations relatives à SSL. Cette information ne change généralement pas après qu'une connexion soit établie.

`PQsslInUse`

Renvoie true (1) si la connexion utilise SSL, false (0) dans le cas contraire.

```
int PQsslInUse(const PGconn *conn);
```

`PQsslAttribute`

Renvoie des informations relatives à SSL à propos de la connexion.

```
const char *PQsslAttribute(const PGconn *conn, const char
 *attribute_name);
```

La liste des attributs disponibles varie en fonction de la bibliothèque SSL utilisée, et du type de la connexion. Renvoie NULL si la connexion n'utilise pas SSL ou si le nom de l'attribut spécifié n'est pas défini pour la bibliothèque utilisée.

Les attributs suivants sont communément disponibles :

`library`

Nom de l'implémentation SSL utilisée. (À ce jour seul `"OpenSSL"` est implémenté)

`protocol`

Version de SSL/TLS utilisée. Les valeurs courantes sont `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`, mais une implémentation peut renvoyer d'autres chaînes si d'autres protocoles sont utilisés.

`key_bits`

Nombre de bits de la clef utilisée par l'algorithme de chiffrement.

`cipher`

Le nom raccourci de la suite cryptographique utilisée, par exemple `"DHE-RSA-DES-CBC3-SHA"`. Les noms sont spécifiques à chaque implémentation.

`compression`

Renvoie "on" si la compression SSL est utilisée, sinon renvoie "off".

Comme cas particulier, l'attribut `library` peut être requêté sans connexion en passant NULL à l'argument `conn`. Le résultat sera le nom de la bibliothèque SSL par défaut ou NULL si libpq a été compilé sans le support SSL. (Avant PostgreSQL version 15, passer NULL pour l'argument `conn` résultait toujours en un NULL. Les programmes clients qui ont besoin de différencier entrer les implémentations nouvelle et ancienne dans ce cas peuvent vérifier la macro `LIBPQ_HAS_SSL_LIBRARY_DETECTION`.)

`PQsslAttributeNames`

Renvoie un tableau des noms d'attribut SSL disponibles. Le tableau est terminé par un pointeur NULL.

```
const char * const * PQsslAttributeNames(const PGconn *conn);
```

`PQsslStruct`

Renvoie un pointeur sur un objet SSL qui est dépendant de l'implémentation et qui décrit la connexion. Renvoie NULL si la connexion n'est pas chiffrée ou si le type demandé d'un objet n'est pas disponible à partir de l'implémentation SSL de la connexion.

```
void *PQsslStruct(const PGconn *conn, const char *struct_name);
```

Là où les structures disponibles dépendent de l'implémentation SSL utilisée. Pour OpenSSL, il y a une structure, disponible sous le nom `OpenSSL`, qui renvoie un pointeur sur la structure `SSL` d'OpenSSL. Un exemple de code utilisant cette fonction pourrait être :

```
 #include <libpq-fe.h>
```

```
#include <openssl/ssl.h>

...

    SSL *ssl;

    dbconn = PQconnectdb(...);
    ...

    ssl = PQsslStruct(dbconn, "OpenSSL");
    if (ssl)
    {
        /* utilisez les fonctions OpenSSL pour accéder à ssl */
    }
```

Cette structure peut être utilisée pour vérifier les niveaux de chiffrement, les certificats du serveur, etc. Référez-vous à la documentation d'OpenSSL pour des informations sur cette structure.

`PQgetssl`

Renvoie la structure SSL utilisée dans la connexion, ou NULL si SSL n'est pas utilisé.

```
void *PQgetssl(const PGconn *conn);
```

Cette fonction est équivalente à `PQsslStruct(conn, "OpenSSL")`. Elle ne devrait pas être utilisée dans les nouvelles applications, car la structure renvoyée est spécifique à OpenSSL et ne sera pas disponible si une autre implémentation SSL est utilisée. Pour vérifier si une connexion utilise SSL, appelez plutôt `PQsslInUse`, et, pour plus de détails à propos de la connexion, utilisez `PQsslAttribute`.

34.3. Fonctions d'exécution de commandes

Une fois la connexion au serveur de la base de données établie avec succès, les fonctions décrites ici sont utilisées pour exécuter les requêtes SQL et les commandes.

34.3.1. Fonctions principales

`PQexec`

Soumet une commande au serveur et attend le résultat.

```
PGresult *PQexec(PGconn *conn, const char *command);
```

Renvoie un pointeur `PGresult` ou peut-être un pointeur NULL. Un pointeur non NULL sera généralement renvoyé sauf dans des conditions particulières comme un manque de mémoire ou lors d'erreurs sérieuses telles que l'incapacité à envoyer la commande au serveur. La fonction `PQresultStatus` devrait être appelée pour vérifier le code retour pour toute erreur (incluant la valeur d'un pointeur NULL, auquel cas il renverra `PGRES_FATAL_ERROR`). Utilisez `PQerrorMessage` pour obtenir plus d'informations sur l'erreur.

La chaîne de la commande peut inclure plusieurs commandes SQL (séparées par des points virgules). Les requêtes multiples envoyées dans un unique appel à `PQexec` sont exécutées dans une seule transaction, sauf si des commandes `BEGIN`/`COMMIT` explicites sont incluses dans la chaîne de requête pour la diviser en de multiples transactions. (Voir Section 55.2.2.1 pour plus de détails sur

comment le serveur traite les chaînes multi-requêtes.) Notez néanmoins que la structure `PGresult` renvoyée décrit seulement le résultat de la dernière commande exécutée à partir de la chaîne. Si une des commandes échoue, l'exécution de la chaîne s'arrête et le `PGresult` renvoyé décrit la condition d'erreur.

`PQexecParams`

Soumet une commande au serveur et attend le résultat, avec la possibilité de passer des paramètres séparément du texte de la commande SQL.

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

`PQexecParams` est identique à `PQexec` mais offre des fonctionnalités supplémentaires : des valeurs de paramètres peuvent être spécifiées séparément de la chaîne de commande et les résultats de la requête peuvent être demandés soit au format texte soit au format binaire.

Voici les arguments de la fonction :

`conn`

L'objet connexion où envoyer la commande.

`command`

La chaîne SQL à exécuter. Si les paramètres sont utilisés, ils sont référencés dans la chaîne avec `$1`, `$2`, etc.

`nParams`

Le nombre de paramètres fournis ; il s'agit de la longueur des tableaux *`paramTypes[]`*, *`paramValues[]`*, *`paramLengths[]`* et *`paramFormats[]`*. (Les pointeurs de tableau peuvent être `NULL` quand *`nParams`* vaut zéro.)

`paramTypes[]`

Spécifie, par OID, les types de données à affecter aux symboles de paramètres. Si *`paramTypes`* est `NULL` ou si tout élément spécifique du tableau est zéro, le serveur infère un type de donnée pour le symbole de paramètre de la même façon qu'il le ferait pour une chaîne littérale sans type.

`paramValues[]`

Spécifie les vraies valeurs des paramètres. Un pointeur NULL dans ce tableau signifie que le paramètre correspondant est NULL ; sinon, le pointeur pointe vers une chaîne texte terminée par un octet nul (pour le format texte) ou vers des données binaires dans le format attendu par le serveur (pour le format binaire).

`paramLengths[]`

Spécifie les longueurs des données réelles des paramètres du format binaire. Il est ignoré pour les paramètres NULL et les paramètres de format texte. Le pointeur du tableau peut être NULL quand il n'y a pas de paramètres binaires.

paramFormats[]

Spécifie si les paramètres sont du texte (place un 0 dans la ligne du tableau pour le paramètre correspondant) ou binaire (place un 1 dans la ligne du tableau pour le paramètre correspondant). Si le pointeur du tableau est nul, alors tous les paramètres sont présumés être des chaînes de texte.

Les valeurs passées dans le format binaire nécessitent de connaître la représentation interne attendue par le processus backend. Par exemple, les entiers doivent être passés dans l'ordre réseau pour les octets. Passer des valeurs `numeric` requiert de connaître le format de stockage du serveur, comme implémenté dans `src/backend/utils/adt/numeric.c::numeric_send()` et `src/backend/utils/adt/numeric.c::numeric_recv()`.

resultFormat

Indiquez zéro pour obtenir les résultats dans un format texte et un pour les obtenir dans un format binaire. (Il n'est actuellement pas possible d'obtenir des formats différents pour des colonnes de résultats différentes bien que le protocole le permette.)

Le principal avantage de `PQexecParams` sur `PQexec` est que les valeurs de paramètres peuvent être séparées à partir de la chaîne de commande, évitant ainsi le besoin de guillemets et d'échappements, toujours pénibles et sources d'erreurs.

Contrairement à `PQexec`, `PQexecParams` autorise au plus une commande SQL dans une chaîne donnée (il peut y avoir des points-virgules mais pas plus d'une commande non vide). C'est une limitation du protocole sous-jacent mais cela a quelque utilité comme défense supplémentaire contre les attaques par injection de SQL.

> **Astuce**
>
> Spécifier les types de paramètres via des OID est difficile, surtout si vous préférez ne pas coder en dur des valeurs OID particulières dans vos programmes. Néanmoins vous pouvez éviter de le faire, même dans des cas où le serveur lui-même ne peut pas déterminer le type du paramètre ou choisit un type différent de celui que vous voulez. Dans le texte de la commande SQL, ajoutez une conversion explicite au symbole de paramètre pour indiquer le type de données que vous enverrez. Par exemple :
>
> ```
> SELECT * FROM ma_table WHERE x = $1::bigint;
> ```
>
> Ceci impose le traitement du paramètre `$1` en tant que `bigint` alors que, par défaut, il se serait vu affecté le même type que `x`. Forcer la décision du type de paramètre, soit de cette façon soit en spécifiant l'OID du type numérique, est fortement recommandé lors de l'envoi des valeurs des paramètres au format binaire car le format binaire a moins de redondance que le format texte et, du coup, il y a moins de chance que le serveur détecte une erreur de correspondance de type pour vous.

`PQprepare`

Soumet une requête pour créer une instruction préparée avec les paramètres donnés et attends la fin de son exécution.

```
PGresult *PQprepare(PGconn *conn,
         const char *stmtName,
         const char *query,
         int nParams,
```

```
            const Oid *paramTypes);
```

PQprepare crée une instruction préparée pour une exécution ultérieure avec PQexecPrepared. Cette fonction autorise les commandes à être exécutées de façon répétée sans être analysées et planifiées à chaque fois ; voir PREPARE pour les détails.

La fonction crée une instruction préparée nommée *stmtName* à partir de la chaîne *query*, qui ne doit contenir qu'une seule commande SQL. *stmtName* peut être "" pour créer une instruction non nommée, auquel cas toute instruction non nommée déjà existante est automatiquement remplacée par cette dernière sinon une erreur sera levée si le nom de l'instruction est déjà définie dans la session en cours. Si des paramètres sont utilisés, ils sont référencés dans la requête avec $1, $2, etc. *nParams* est le nombre de paramètres pour lesquels des types sont prédéfinis dans le tableau *paramTypes[]* (le pointeur du tableau pourrait être NULL quand *nParams* vaut zéro). *paramTypes[]* spécifie les types de données à affecter aux symboles de paramètres par leur OID. Si *paramTypes* est NULL ou si un élément particulier du tableau vaut zéro, le serveur affecte un type de données au symbole du paramètre de la même façon qu'il le ferait pour une chaîne littérale non typée. De plus, la requête peut utiliser des symboles de paramètre avec des nombres plus importants que *nParams* ; les types de données seront aussi inférés pour ces symboles. (Voir PQdescribePrepared pour un moyen de trouver les types de données inférés.)

Comme avec PQexec, le résultat est normalement un objet PGresult dont le contenu indique le succès ou l'échec côté serveur. Un résultat NULL indique un manque de mémoire ou une incapacité à envoyer la commande. Utilisez PQerrorMessage pour obtenir plus d'informations sur de telles erreurs.

Les instructions préparées avec PQexecPrepared peuvent aussi être créées en exécutant les instructions SQL PREPARE. De plus, bien qu'il n'y ait aucune fonction libpq pour supprimer une instruction préparée, l'instruction SQL DEALLOCATE peut être utilisée dans ce but.

PQexecPrepared

Envoie une requête pour exécuter une instruction séparée avec les paramètres donnés, et attend le résultat.

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

PQexecPrepared est identique à PQexecParams mais la commande à exécuter est spécifiée en nommant l'instruction préparée précédemment au lieu de donner une chaîne de requête. Cette fonctionnalité permet aux commandes utilisées de façon répétée d'être analysées et planifiées seulement une fois plutôt que chaque fois qu'ils sont exécutés. L'instruction doit avoir été préparée précédemment dans la session en cours.

Les paramètres sont identiques à PQexecParams, sauf que le nom d'une instruction préparée est donné au lieu d'une chaîne de requête et que le paramètre *paramTypes[]* n'est pas présent (il n'est pas nécessaire car les types des paramètres de l'instruction préparée ont été déterminés à la création).

PQdescribePrepared

Soumet une requête pour obtenir des informations sur l'instruction préparée indiquée et attend le retour de la requête.

```
PGresult *PQdescribePrepared(PGconn *conn, const char
 *stmtName);
```

`PQdescribePrepared` permet à une application d'obtenir des informations sur une instruction préparée précédente.

`stmtName` peut être `""` ou `NULL` pour référencer l'instruction non nommée. Sinon, ce doit être le nom d'une instruction préparée existante. En cas de succès, un `PGresult` est renvoyé avec le code retour `PGRES_COMMAND_OK`. Les fonctions `PQnparams` et `PQparamtype` peuvent utiliser ce `PGresult` pour obtenir des informations sur les paramètres de l'instruction préparée, et les fonctions `PQnfields`, `PQfname`, `PQftype`, etc. fournissent des informations sur les colonnes résultantes (s'il y en a) de l'instruction.

`PQdescribePortal`

Soumet une requête pour obtenir des informations sur le portail indiqué et attend le retour de la requête.

```
PGresult *PQdescribePortal(PGconn *conn, const char
 *portalName);
```

`PQdescribePortal` permet à une application d'obtenir des informations sur un portail précédemment créé. (libpq ne fournit pas d'accès direct aux portails mais vous pouvez utiliser cette fonction pour inspecter les propriétés d'un curseur créé avec la commande SQL `DECLARE CURSOR`.) `PQdescribePortal` est seulement supporté dans les connexions en protocole 3.0 et ultérieurs ; il échouera lors de l'utilisation du protocole 2.0.

`portalName` peut être `""` ou `NULL` pour référencer un portail sans nom. Sinon, il doit correspondre au nom d'un portail existant. En cas de succès, un `PGresult` est renvoyé avec le code de retour `PGRES_COMMAND_OK`. Les fonctions `PQnfields`, `PQfname`, `PQftype`, etc. peuvent utiliser ce `PGresult` pour obtenir des informations sur les colonnes résultantes (s'il y en a) du portail.

La structure `PGresult` encapsule le résultat renvoyé par le serveur. Les développeurs d'applications libpq devraient faire attention au maintien de l'abstraction de `PGresult`. Utilisez les fonctions d'accès ci-dessous pour obtenir le contenu de `PGresult`. Évitez de référencer directement les champs de la structure `PGresult` car ils sont sujets à changements dans le futur.

`PQresultStatus`

Renvoie l'état du résultat d'une commande.

```
ExecStatusType PQresultStatus(const PGresult *res);
```

`PQresultStatus` peut renvoyer une des valeurs suivantes :

`PGRES_EMPTY_QUERY`

La chaîne envoyée au serveur était vide.

`PGRES_COMMAND_OK`

Fin avec succès d'une commande ne renvoyant aucune donnée.

`PGRES_TUPLES_OK`

Fin avec succès d'une commande renvoyant des données (telle que `SELECT` ou `SHOW`).

PGRES_COPY_OUT

Début de l'envoi (à partir du serveur) d'un flux de données.

PGRES_COPY_IN

Début de la réception (sur le serveur) d'un flux de données.

PGRES_BAD_RESPONSE

La réponse du serveur n'a pas été comprise.

PGRES_NONFATAL_ERROR

Une erreur non fatale (une note ou un avertissement) est survenue.

PGRES_FATAL_ERROR

Une erreur fatale est survenue.

PGRES_COPY_BOTH

Lancement du transfert de données Copy In/Out (vers et à partir du serveur). Cette fonctionnalité est seulement utilisée par la réplication en flux, ce statut ne devrait donc pas apparaître dans les applications ordinaires.

PGRES_SINGLE_TUPLE

La structure PGresult contient une seule ligne de résultat provenant de la commande courante. Ce statut n'intervient que lorsque le mode ligne-à-ligne a été sélectionné pour cette requête (voir Section 34.6).

PGRES_PIPELINE_SYNC

La structure PGresult représente un point de synchronisation dans le mode pipeline, réclamé par PQpipelineSync. Ce statut survient seulement quand le mode pipeline a été sélectionné.

PGRES_PIPELINE_ABORTED

La structure PGresult représente un pipeline qui a reçu une erreur du serveur. PQgetResult doit être appelé de façon répété et, chaque fois, il renverra ce code de statut jusqu'à la fin du pipeline courant, auquel cas il renverra PGRES_PIPELINE_SYNC et le traitement normal pourra recommencer.

Si le statut du résultat est PGRES_TUPLES_OK ou PGRES_SINGLE_TUPLE, alors les fonctions décrites ci-dessous peuvent être utilisées pour récupérer les lignes renvoyées par la requête. Notez qu'une commande SELECT qui récupère zéro ligne affichera toujours PGRES_TUPLES_OK. PGRES_COMMAND_OK est pour les commandes qui ne peuvent jamais renvoyer de lignes (INSERT ou UPDATE sans une clause RETURNING, etc.). Une réponse PGRES_EMPTY_QUERY pourrait indiquer un bogue dans le logiciel client.

Un résultat de statut PGRES_NONFATAL_ERROR ne sera jamais renvoyé directement par PQexec ou d'autres fonctions d'exécution de requêtes ; les résultats de ce type sont passés à l'exécuteur de notifications (voir la Section 34.13).

PQresStatus

Convertit le type énuméré renvoyé par PQresultStatus en une constante de type chaîne décrivant le code d'état. L'appelant ne devrait pas libérer le résultat.

```
char *PQresStatus(ExecStatusType status);
```

`PQresultErrorMessage`

Renvoie le message d'erreur associé avec la commande ou une chaîne vide s'il n'y a pas eu d'erreurs.

```
char *PQresultErrorMessage(const PGresult *res);
```

S'il y a eu une erreur, la chaîne renvoyée incluera un retour chariot en fin. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand le pointeur `PGresult` associé sera passé à `PQclear`.

Immédiatement après un appel à `PQexec` ou `PQgetResult`, `PQerrorMessage` (sur la connexion) renverra la même chaîne que `PQresultErrorMessage` (sur le résultat). Néanmoins, un `PGresult` conservera son message d'erreur jusqu'à destruction, alors que le message d'erreur de la connexion changera avec les opérations suivantes. Utilisez `PQresultErrorMessage` quand vous voulez connaître le statut associé avec un `PGresult` particulier ; utilisez `PQerrorMessage` lorsque vous souhaitez connaître le statut à partir de la dernière opération sur la connexion.

`PQresultVerboseErrorMessage`

Renvoie une version reformatée du message d'erreur associé avec un objet `PGresult`.

```
char *PQresultVerboseErrorMessage(const PGresult *res,
                                  PGVerbosity verbosity,
                                  PGContextVisibility
 show_context);
```

Dans certaines situations, un client pourrait vouloir une version plus détaillée d'une erreur déjà rapportée. `PQresultVerboseErrorMessage` couvre ce besoin en traitant le message tel qu'il aurait été produit par `PQresultErrorMessage` si la configuration souhaitée de la verbosité était activée pour la connexion au moment où l'objet `PGresult` indiqué a été généré. Si le `PGresult` ne correspond pas une erreur, « PGresult is not an error result » est renvoyé à la place. La chaîne renvoyée inclut un retour à la ligne en fin de chaîne.

Contrairement à la plupart des autres fonctions d'extraction de données à partir d'un objet `PGresult`, le résultat de cette fonction est une chaîne tout juste allouée. L'appelant doit la libérer en utilisant `PQfreemem()` quand la chaîne n'est plus nécessaire.

Un NULL en retour est possible s'il n'y a pas suffisamment de mémoire.

`PQresultErrorField`

Renvoie un champ individuel d'un rapport d'erreur.

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

`fieldcode` est un identifiant de champ d'erreur ; voir les symboles listés ci-dessous. `NULL` est renvoyé si `PGresult` n'est pas un résultat d'erreur ou d'avertissement, ou n'inclut pas le champ spécifié. Les valeurs de champ n'incluront normalement pas un retour chariot en fin. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand le pointeur `PGresult` associé sera passé à `PQclear`.

Les codes de champs suivants sont disponibles :

`PG_DIAG_SEVERITY`

La sévérité ; le contenu du champ peut être `ERROR`, `FATAL` ou `PANIC` dans un message d'erreur, ou `WARNING`, `NOTICE`, `DEBUG`, `INFO` ou `LOG` dans un message de notification, ou une traduction localisée d'un de ceux-ci. Toujours présent.

`PG_DIAG_SEVERITY_NONLOCALIZED`

La sévérité ; le contenu du champ peut être `ERROR`, `FATAL` ou `PANIC` (dans un message d'erreur), ou `WARNING`, `NOTICE`, `DEBUG`, `INFO` ou `LOG` (dans un message de notification). C'est identique au champ `PG_DIAG_SEVERITY` sauf que le contenu n'est jamais traduit. Il est présent uniquement dans les rapports générés par les versions 9.6 et ultérieurs de PostgreSQL.

`PG_DIAG_SQLSTATE`

Le code SQLSTATE de l'erreur. Ce code identifie le type d'erreur qui est survenu ; il peut être utilisé par des interfaces utilisateur pour des opérations spécifiques (telles que la gestion des erreurs) en réponse à une erreur particulière de la base de données. Pour une liste des codes SQLSTATE possibles, voir l'Annexe A. Ce champ n'est pas localisable et est toujours présent.

`PG_DIAG_MESSAGE_PRIMARY`

Le message d'erreur principal, compréhensible par un humain (typiquement sur une ligne). Toujours présent.

`PG_DIAG_MESSAGE_DETAIL`

Détail : un message d'erreur secondaire et optionnel proposant plus d'informations sur le problème. Pourrait courir sur plusieurs lignes.

`PG_DIAG_MESSAGE_HINT`

Astuce : une suggestion supplémentaire sur ce qu'il faut faire suite à ce problème. Elle a pour but de différer du détail car elle offre un conseil (potentiellement inapproprié) plutôt que des faits établis. Pourrait courir sur plusieurs lignes.

`PG_DIAG_STATEMENT_POSITION`

Une chaîne contenant un entier décimal indiquant une position du curseur d'erreur comme index dans la chaîne d'instruction originale. Le premier caractère a l'index 1 et les positions sont mesurées en caractères, et non en octets.

`PG_DIAG_INTERNAL_POSITION`

Ceci est défini de la même façon que le champ `PG_DIAG_STATEMENT_POSITION` mais est utilisé quand la position du curseur fait référence à une commande générée en interne plutôt qu'une soumise par le client. Le champ `PG_DIAG_INTERNAL_QUERY` apparaîtra toujours quand ce champ apparaît.

`PG_DIAG_INTERNAL_QUERY`

Le texte d'une commande générée en interne et échouée. Ce pourrait être, par exemple, une requête SQL lancée par une fonction PL/pgSQL.

`PG_DIAG_CONTEXT`

Une indication du contexte dans lequel l'erreur est apparue. Actuellement, cela inclut une trace de la pile d'appels des fonctions du langage procédural actif et de requêtes générées en interne. La trace a une entrée par ligne, la plus récente au début.

`PG_DIAG_SCHEMA_NAME`

Si l'erreur était associée à un objet spécifique de la base de données, nom du schéma contenant cet objet.

`PG_DIAG_TABLE_NAME`

Si l'erreur était associée à une table spécifique, nom de cette table. (Fait référence au champ du nom du schéma pour le nom du schéma de la table.)

`PG_DIAG_COLUMN_NAME`

Si l'erreur était associée à une colonne spécifique d'une table, nom de cette colonne. (Fait référence aux champs de noms du schéma et de la table pour identifier la table.)

`PG_DIAG_DATATYPE_NAME`

Si l'erreur était associée à un type de données spécifique, nom de ce type de données. (Fait référence au champ du nom du schéma pour le schéma du type de données.)

`PG_DIAG_CONSTRAINT_NAME`

Si l'erreur était associée à une contrainte spécifique, nom de cette contrainte. Cela fait référence aux champs listés ci-dessus pour la table ou le domaine associé. (Dans ce cadre, les index sont traités comme des contraintes, même s'ils n'ont pas été créés avec la syntaxe des contraintes.)

`PG_DIAG_SOURCE_FILE`

Le nom du fichier contenant le code source où l'erreur a été rapportée.

`PG_DIAG_SOURCE_LINE`

Le numéro de ligne dans le code source où l'erreur a été rapportée.

`PG_DIAG_SOURCE_FUNCTION`

Le nom de la fonction dans le code source où l'erreur a été rapportée.

> **Note**
>
> Les champs pour les noms du schéma, de la table, de la colonne, du type de données et de la contrainte sont fournis seulement pour un nombre limité de types d'erreurs ; voir Annexe A. Ne supposez pas que la présence d'un de ces champs garantisse la présence d'un autre champ. Les sources d'erreurs du moteur observent les relations notées ci-dessus mais les fonctions utilisateurs peuvent utiliser ces champs d'une autre façon. Dans la même idée, ne supposez pas que ces champs indiquent des objets encore existants dans la base de données courante.

Le client est responsable du formatage des informations affichées suivant ses besoins ; en particulier, il doit supprimer les longues lignes si nécessaires. Les caractères de retour chariot apparaissant dans les champs de message d'erreur devraient être traités comme des changements de paragraphes, pas comme des changements de lignes.

Les erreurs générées en interne par libpq auront une sévérité et un message principal mais aucun autre champ.

Notez que les champs d'erreurs sont seulement disponibles pour les objets `PGresult`, et non pas pour les objets `PGconn` ; il n'existe pas de fonction `PQerrorField`.

`PQclear`

Libère le stockage associé avec un `PGresult`. Chaque résultat de commande devrait être libéré via `PQclear` lorsqu'il n'est plus nécessaire.

```
void PQclear(PGresult *res);
```

Vous pouvez conserver un objet `PGresult` aussi longtemps que vous en avez besoin ; il ne part pas lorsque vous lancez une nouvelle commande, même pas si vous fermez la connexion. Pour vous en débarrasser, vous devez appeler `PQclear`. En cas d'oubli, le résultat sera une fuite de mémoire dans votre application.

34.3.2. Récupérer l'information dans le résultat des requêtes

Ces fonctions sont utilisées pour extraire des informations provenant d'un objet `PGresult` représentant un résultat valide pour une requête (statut `PGRES_TUPLES_OK` ou `PGRES_SINGLE_TUPLE`). Elles peuvent aussi être utilisés pour extraire des informations à partir d'une opération Describe réussie : le résultat d'un Describe a les mêmes informations de colonnes qu'une exécution réelle de la requête aurait fournie, mais avec zéro ligne. Pour les objets avec d'autres valeurs de statut, ces fonctions agiront comme si le résultat avait zéro ligne et zéro colonne.

`PQntuples`

Renvoie le nombre de lignes (enregistrements, ou *tuples*) du résultat de la requête. (Notez que les objets `PGresult` sont limités à au plus `INT_MAX` lignes, donc un résultat de type `int` est suffisant.)

```
int PQntuples(const PGresult *res);
```

`PQnfields`

Renvoie le nombre de colonnes (champs) de chaque ligne du résultat de la requête.

```
int PQnfields(const PGresult *res);
```

`PQfname`

Renvoie le nom de la colonne associé avec le numéro de colonne donnée. Les numéros de colonnes commencent à zéro. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand le pointeur `PGresult` associée est passée à `PQclear`.

```
char *PQfname(const PGresult *res,
                      int column_number);
```

`NULL` est renvoyé si le numéro de colonne est en dehors de la plage.

`PQfnumber`

Renvoie le numéro de colonne associé au nom de la colonne donné.

```
int PQfnumber(const PGresult *res,
                     const char *column_name);
```

-1 est renvoyé si le nom donné ne correspond à aucune colonne.

Le nom donné est traité comme un identifiant dans une commande SQL, c'est-à-dire qu'il est mis en minuscule sauf s'il est entre des guillemets doubles. Par exemple, pour le résultat de la requête suivante :

```
SELECT 1 AS FOO, 2 AS "BAR";
```

nous devons obtenir les résultats suivants :

```
PQfname(res, 0)              foo
      PQfname(res, 1)              BAR
      PQfnumber(res, "FOO")        0
      PQfnumber(res, "foo")        0
      PQfnumber(res, "BAR")        -1
      PQfnumber(res, "\"BAR\"")    1
```

PQftable

Renvoie l'OID de la table à partir de laquelle la colonne donnée a été récupérée. Les numéros de colonnes commencent à 0.

```
Oid PQftable(const PGresult *res,
                    int column_number);
```

`InvalidOid` est renvoyé si le numéro de colonne est en dehors de la plage, ou si la colonne spécifiée n'est pas une simple référence à une colonne de table. Vous pouvez requêter la table système `pg_class` pour déterminer exactement quelle table est référencée.

Le type `Oid` et la constante `InvalidOid` seront définis lorsque vous incluerez le fichier d'en-tête libpq. Ils auront le même type entier.

PQftablecol

Renvoie le numéro de colonne (dans sa table) de la colonne correspondant à la colonne spécifiée de résultat de la requête. Les numéros de colonne du résultat commencent à 0, mais les colonnes de table ont des numéros supérieurs à zéro.

```
int PQftablecol(const PGresult *res,
                     int column_number);
```

Zéro est renvoyé si le numéro de colonne est en dehors de la plage, ou si la colonne spécifiée n'est pas une simple référence à une colonne de table, ou lors de l'utilisation d'un protocole antérieur à la version 3.0.

PQfformat

Renvoie le code de format indiquant le format de la colonne donné. Les numéros de colonnes commencent à zéro.

```
int PQfformat(const PGresult *res,
                     int column_number);
```

Le code de format zéro indique une représentation textuelle des données, alors qu'un code un indique une représentation binaire (les autres codes sont réservés pour des définitions futures).

`PQftype`

Renvoie le type de données associé avec le numéro de colonne donné. L'entier renvoyé est le numéro OID interne du type. Les numéros de colonnes commencent à zéro.

```
Oid PQftype(const PGresult *res,
                   int column_number);
```

Vous pouvez lancer des requêtes sur la table système `pg_type` pour obtenir les noms et propriétés des différents types de données. Les OID des types de données intégrés sont définis dans le fichier `catalog/pg_type_d.h` du répertoire `include` de l'installation de PostgreSQL.

`PQfmod`

Renvoie le modificateur de type de la colonne associée avec le numéro de colonne donné. Les numéros de colonnes commencent à zéro.

```
int PQfmod(const PGresult *res,
                  int column_number);
```

L'interprétation des valeurs du modificateur est spécifique au type ; typiquement elles indiquent la précision ou les limites de taille. La valeur -1 est utilisée pour indiquer « aucune information disponible ». La plupart des types de données n'utilisent pas les modificateurs, auquel cas la valeur est toujours -1.

`PQfsize`

Renvoie la taille en octets de la colonne associée au numéro de colonne donné. Les numéros de colonnes commencent à 0.

```
int PQfsize(const PGresult *res,
                   int column_number);
```

`PQfsize` renvoie l'espace alloué pour cette colonne dans une ligne de la base de données, en d'autres termes la taille de la représentation interne du serveur du type de données (en conséquence ce n'est pas réellement utile pour les clients). Une valeur négative indique que les types de données ont une longueur variable.

`PQbinaryTuples`

Renvoie 1 si `PGresult` contient des données binaires et 0 s'il contient des données texte.

```
int PQbinaryTuples(const PGresult *res);
```

Cette fonction est obsolète (sauf dans le cas d'une utilisation en relation avec `COPY`) car un seul `PGresult` peut contenir du texte dans certaines colonnes et des données binaires dans d'autres. `PQfformat` est à préférer. `PQbinaryTuples` renvoie 1 seulement si toutes les colonnes du résultat sont dans un format binaire (format 1).

`PQgetvalue`

Renvoie la valeur d'un seul champ d'une seule ligne d'un `PGresult`. Les numéros de lignes et de colonnes commencent à zéro. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand le pointeur `PGresult` associé sera passé à `PQclear`.

```
char* PQgetvalue(const PGresult *res,
                         int row_number,
```

```
                                int column_number);
```

Pour les données au format texte, la valeur renvoyée par `PQgetvalue` est une représentation au format chaîne de caractères terminée par un octet nul de la valeur du champ. Pour les données au format binaire, la valeur dans la représentation binaire est déterminée par le type de la donnée, fonctions `typsend` et `typreceive`. (La valeur est en fait suivie d'un octet zéro dans ce cas aussi, mais ce n'est pas réellement utile car la valeur a des chances de contenir d'autres valeurs NULL embarquées).

Une chaîne vide est renvoyée si la valeur du champ est NULL. Voir `PQgetisnull` pour distinguer les valeurs NULL des chaînes vides.

Le pointeur renvoyé par `PQgetvalue` pointe vers le stockage qui fait partie de la structure `PGresult`. Personne ne devrait modifier les données vers lesquelles il pointe, et tout le monde devrait copier les données dans un autre stockage explicitement si elles doivent être utilisées après la durée de vie de la structure `PGresult`.

`PQgetisnull`

Teste un champ pour savoir s'il est NULL. Les numéros de lignes et de colonnes commencent à 0.

```
int PQgetisnull(const PGresult *res,
                        int row_number,
                        int column_number);
```

Cette fonction renvoie 1 si le champ est nul et 0 s'il contient une valeur non NULL (notez que `PQgetvalue` renverra une chaîne vide, et non pas un pointeur NULL, pour un champ NULL).

`PQgetlength`

Renvoie la longueur réelle de la valeur d'un champ en octet. Les numéros de lignes et de colonnes commencent à 0.

```
int PQgetlength(const PGresult *res,
                       int row_number,
                       int column_number);
```

C'est la longueur réelle des données pour cette donnée en particulier, c'est-à-dire la taille de l'objet pointé par `PQgetvalue`. Pour le format textuel, c'est identique à `strlen()`. Pour le format binaire, c'est une information essentielle. Notez que l'on ne devrait *pas* se fier à `PQfsize` pour obtenir la taille réelle des données.

`PQnparams`

Renvoie le nombre de paramètres d'une instruction préparée.

```
int PQnparams(const PGresult *res);
```

Cette fonction est utile seulement pour inspecter le résultat de `PQdescribePrepared`. Pour les autres types de requêtes, elle renverra zéro.

`PQparamtype`

Renvoie le type de donnée du paramètre indiqué dans l'instruction. Les numéros des paramètres commencent à 0.

```
Oid PQparamtype(const PGresult *res, int param_number);
```

Cette fonction est utile seulement pour inspecter le résultat de `PQdescribePrepared`. Pour les autres types de requêtes, elle renverra zéro.

`PQprint`

Affiche toutes les lignes et, optionnellement, les noms des colonnes dans le flux de sortie spécifié.

```
void PQprint(FILE* fout,      /* flux de sortie */
                  const PGresult *res,
                  const PQprintOpt *po);

       typedef struct
       {
           pqbool  header;      /* affiche les en-têtes des
 champs et le nombre de
                                   lignes */
           pqbool  align;       /* aligne les champs */
           pqbool  standard;    /* vieux format (mort) */
           pqbool  html3;       /* affiche les tables en HTML
 */
           pqbool  expanded;    /* étend les tables */
           pqbool  pager;       /* utilise le paginateur pour
 la sortie si nécessaire
                                   */
           char    *fieldSep;   /* séparateur de champ */
           char    *tableOpt;   /* attributs des éléments de
 table HTML */
           char    *caption;    /* titre de la table HTML */
           char    **fieldName; /* Tableau terminé par un NULL
 des noms de remplacement
                                   des champs */
       } PQprintOpt;
```

Cette fonction était auparavant utilisée par psql pour afficher les résultats des requêtes mais ce n'est plus le cas. Notez qu'elle assume que les données sont dans un format textuel.

34.3.3. Récupérer d'autres informations de résultats

Ces fonctions sont utilisées pour extraire d'autres informations des objets `PGresult`.

`PQcmdStatus`

Renvoie l'état de la commande de l'instruction SQL qui a généré le `PGresult`.

```
char * PQcmdStatus(PGresult *res);
```

D'habitude, c'est juste le nom de la commande mais elle peut inclure des données supplémentaires comme le nombre de lignes traitées. L'appelant ne devrait pas libérer directement le résultat. Il sera libéré quand le pointeur `PGresult` associée sera passé à `PQclear`.

`PQcmdTuples`

Renvoie le nombre de lignes affectées par la commande SQL.

```
char * PQcmdTuples(PGresult *res);
```

Cette fonction renvoie une chaîne contenant le nombre de lignes affectées par l'instruction SQL qui a généré `PGresult`. Cette fonction ne peut être utilisée qu'après l'exécution d'une instruction `SELECT`, `CREATE TABLE AS`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `MOVE`, `FETCH` ou `COPY`, ou un `EXECUTE` d'une instruction préparée contenant une instruction `INSERT`, `UPDATE`, `DELETE` ou `MERGE`. Si la commande qui a généré `PGresult` était autre chose, `PQcmdTuples` renverrait directement une chaîne vide. L'appelant ne devrait pas libérer la valeur de retour directement. Elle sera libérée quand le pointeur `PGresult` associée sera passé à `PQclear`.

`PQoidValue`

Renvoie l'OID de la ligne insérée, si la commande SQL était un `INSERT` qui a inséré exactement une ligne dans une table comprenant des OID, ou un `EXECUTE` d'une requête préparée contenant une instruction `INSERT` convenable. Sinon cette fonction renvoie `InvalidOid`. Cette fonction renverra aussi `InvalidOid` si la table touchée par l'instruction `INSERT` ne contient pas d'OID.

```
Oid PQoidValue(const PGresult *res);
```

`PQoidStatus`

Cette fonction est obsolète. Utilisez plutôt `PQoidValue`. De plus, elle n'est pas compatible avec les threads. Elle renvoie une chaîne contenant l'OID de la ligne insérée alors que `PQoidValue` renvoie la valeur de l'OID.

```
char * PQoidStatus(const PGresult *res);
```

34.3.4. Échapper les chaînes dans les commandes SQL

`PQescapeLiteral`

```
        char *PQescapeLiteral(PGconn *conn, const char *str,
 size_t length);
```

`PQescapeLiteral` échappe une chaîne pour l'utiliser dans une commande SQL. C'est utile pour insérer des données comme des constantes dans des commandes SQL. Certains caractères, comme les guillemets et les antislashs, doivent être traités avec des caractères d'échappement pour éviter qu'ils ne soient traités d'après leur signification spéciale par l'analyseur SQL. `PQescapeLiteral` réalise cette opération.

`PQescapeLiteral` renvoie une version échappée du paramètre *`str`* dans une mémoire allouée avec `malloc()`. Cette mémoire devra être libérée en utilisant `PQfreemem()` quand le résultat ne sera plus utile. Un octet zéro final n'est pas requis et ne doit pas être compté dans *`length`*. (Si un octet zéro est découvert avant le traitement de *`length`* octets, `PQescapeLiteral` s'arrête au zéro ; ce comportement est celui de `strncpy`.) Les caractères spéciaux de la chaîne en retour ont été remplacés pour qu'ils puissent être traités correctement par l'analyseur de chaînes de PostgreSQL. Un octet zéro final est aussi ajouté. Les guillemets simples qui doivent entourer les chaînes littérales avec PostgreSQL sont inclus dans la chaîne résultante.

En cas d'erreur, `PQescapeLiteral` renvoie `NULL` et un message adéquat est stocké dans l'objet *`conn`*.

> **Astuce**
>
> Il est particulièrement important de faire un échappement propre lors de l'utilisation de chaînes provenant d'une source qui n'est pas de confiance. Sinon, il existe un risque de sécurité : vous vous exposez à une attaque de type « injection SQL » avec des commandes SQL non voulues injectées dans votre base de données.

Notez qu'il n'est pas nécessaire ni correct de faire un échappement quand une valeur est passée en tant que paramètre séparé dans `PQexecParams` ou ses routines sœurs.

`PQescapeIdentifier`

```
char *PQescapeIdentifier(PGconn *conn, const char *str,
size_t length);
```

`PQescapeIdentifier` échappe une chaîne pour qu'elle puisse être utilisé en tant qu'identifiant SQL, par exemple pour le nom d'une table, d'une colonne ou d'une fonction. C'est utile quand un identifiant fourni par un utilisateur pourrait contenir des caractères spéciaux, qui sinon ne seraient pas interprétés comme faisant partie de l'identifiant par l'analyseur SQL, ou lorsque l'identifiant pourrait contenir des caractères en majuscule, dont la casse doit être préservée.

`PQescapeIdentifier` renvoie une version du paramètre *`str`* échappée comme doit l'être un identifiant SQL, dans une mémoire allouée avec `malloc()`. Cette mémoire doit être libérée en utilisant `PQfreemem()` quand le résultat n'est plus nécessaire. Un octet zéro final n'est pas nécessaire et ne doit pas être comptabilisé dans *`length`*. (Si un octet zéro est trouvé avant le traitement de *`length`* octets, `PQescapeIdentifier` s'arrête au zéro ; ce comportement est celui de `strncpy`.) Les caractères spéciaux de la chaîne en retour ont été remplacés pour que ce dernier soit traité proprement comme un identifiant SQL. Un octet zéro final est aussi ajouté. La chaîne de retour sera aussi entourée de guillemets doubles.

En cas d'erreur, `PQescapeIdentifier` renvoie `NULL` et un message d'erreur adéquat est stocké dans l'objet *`conn`*.

> **Astuce**
>
> Comme avec les chaînes littérales, pour empêcher les attaques par injection SQL, les identifiants SQL doivent être échappés lorsqu'ils proviennent d'une source non sûre.

`PQescapeStringConn`

```
size_t PQescapeStringConn (PGconn *conn,
                           char *to, const char *from,
size_t length,
                           int *error);
```

`PQescapeStringConn` > échappe les chaînes littérales de la même façon que `PQescapeLiteral` . Contrairement à `PQescapeLiteral` , l'appelant doit fournir un tampon d'une taille appropriée. De plus, `PQescapeStringConn` > n'ajoute pas de guillemets simples autour des chaînes littérales de PostgreSQL ; elles doivent être ajoutées dans la commande SQL où ce résultat sera inséré. Le paramètre *`from`* pointe vers le premier caractère de la chaîne à échapper, et le paramètre *`length`* précise le nombre d'octets contenus dans cette chaîne. Un octet

zéro final n'est pas nécessaire et ne doit pas être comptabilisé dans *length*. (Si un octet zéro est trouvé avant le traitement de *length* octets, PQescapeStringConn s'arrête au zéro ; ce comportement est celui de strncpy.) *to* doit pointer vers un tampon qui peut contenir au moins un octet de plus que deux fois la valeur de *length*, sinon le comportement de la fonction est indéfini. Le comportement est indéfini si les chaînes *to* et *from* se recouvrent.

Si le paramètre *error* est différent de NULL, alors *error est configuré à zéro en cas de succès, et est différent de zéro en cas d'erreur. Actuellement, les seules conditions permettant une erreur impliquent des encodages multi-octets dans la chaîne source. La chaîne en sortie est toujours générée en cas d'erreur mais on peut s'attendre à ce que le serveur la rejette comme une chaîne malformée. En cas d'erreur, un message adéquat est stocké dans l'objet *conn*, que *error* soit NULL ou non.

PQescapeStringConn renvoie le nombre d'octets écrits dans *to*, sans inclure l'octet zéro final.

PQescapeString

PQescapeString est une version ancienne et obsolète de PQescapeStringConn .

```
        size_t PQescapeString (char *to, const char *from,
 size_t length);
```

La seule différence avec PQescapeStringConn tient dans le fait que PQescapeString n'a pas de paramètres *conn* et *error*. À cause de cela, elle ne peut ajuster son comportement en fonction des propriétés de la connexion (comme l'encodage des cnécssairearactères) et du coup, *elle pourrait fournir des résultats erronés*. De plus, elle ne peut pas renvoyer de conditions d'erreur.

PQescapeString peut être utilisé en toute sécurité avec des programmes client utilisant une seule connexion PostgreSQL à la fois (dans ce cas, il peut trouver ce qui l'intéresse « en arrière-plan »). Dans d'autres contextes, c'est un risque en terme de sécurité. Cette fonction devrait être évitée et remplacée par PQescapeStringConn .

PQescapeByteaConn

Échappe des données binaires à utiliser à l'intérieur d'une commande SQL avec le type bytea. Comme avec PQescapeStringConn , c'est seulement utilisé pour insérer des données directement dans une chaîne de commande SQL.

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                 const unsigned char *from,
                                 size_t from_length,
                                 size_t *to_length);
```

Certaines valeurs d'octets *doivent* être échappées lorsqu'elles font partie d'un littéral bytea dans une instruction SQL. PQescapeByteaConn échappe les octets en utilisant soit un codage hexadécimal soit un échappement avec des antislashs. Voir Section 8.4 pour plus d'informations.

Le paramètre *from* pointe sur le premier octet de la chaîne à échapper et le paramètre *from_length* donne le nombre d'octets de cette chaîne binaire (un octet zéro final n'est ni nécessaire ni compté). Le paramètre *to_length* pointe vers une variable qui contiendra la longueur de la chaîne échappée résultante. Cette longueur inclut l'octet zéro de terminaison.

PQescapeByteaConn renvoie une version échappée du paramètre *from* dans la mémoire allouée avec malloc(). Cette mémoire doit être libérée avec PQfreemem lorsque le résultat n'est plus nécessaire. Tous les caractères spéciaux de la chaîne de retour sont remplacés de façon à ce qu'ils puissent être traités proprement par l'analyseur de chaînes littérales de PostgreSQL et

par la fonction d'entrée bytea. Un octet zéro final est aussi ajouté. Les guillemets simples qui encadrent les chaînes littérales de PostgreSQL ne font pas partie de la chaîne résultante.

En cas d'erreur, un pointeur NULL est renvoyé et un message d'erreur adéquat est stocké dans l'objet *conn*. Actuellement, la seule erreur possible est une mémoire insuffisante pour stocker la chaîne résultante.

PQescapeBytea

PQescapeBytea est une version ancienne et obsolète de PQescapeByteaConn.

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

La seule différence avec PQescapeByteaConn est que PQescapeBytea ne prend pas de paramètre PGconn. De ce fait, PQescapeBytea ne peut être utilisé en toute sécurité que dans des programmes qui n'utilisent qu'une seule connexion PostgreSQL à la fois (dans ce cas, il peut trouver ce dont il a besoin « en arrière-plan »). Elle *pourrait donner des résultats erronés* si elle est utilisé dans des programmes qui utilisent plusieurs connexions de bases de données (dans ce cas, utilisez PQescapeByteaConn).

PQunescapeBytea

Convertit une représentation de la chaîne en donnés binaires -- l'inverse de PQescapeBytea. Ceci est nécessaire lors de la récupération de données bytea en format texte, mais pas lors de sa récupération au format binaire.

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t
 *to_length);
```

Le paramètre *from* pointe vers une chaîne de telle qu'elle pourrait provenir de PQgetvalue lorsque la colonne est de type bytea. PQunescapeBytea convertit cette représentation de la chaîne en sa représentation binaire. Elle renvoie un pointeur vers le tampon alloué avec malloc(), ou NULL en cas d'erreur, et place la taille du tampon dans *to_length*. Le résultat doit être libéré en utilisant PQfreemem lorsqu'il n'est plus nécessaire.

Cette conversion n'est pas l'inverse exacte de PQescapeBytea car la chaîne n'est pas supposée être "échappée" tel qu'elle est renvoyée par PQgetvalue. Cela veut dire, notamment, qu'il n'y a pas besoin de réfléchir à la mise entre guillemets de la chaîne, et donc pas besoin d'un paramètre PGconn.

34.4. Traitement des commandes asynchrones

La fonction PQexec est adéquate pour soumettre des commandes aux applications standards, synchrones. Néanmoins, elle a quelques défauts pouvant être d'importance à certains utilisateurs :

- PQexec attend que la commande se termine. L'application pourrait avoir du travail ailleurs (comme le rafraîchissement de l'interface utilisateur), auquel cas elle ne voudra pas être bloquée en attente de la réponse.
- Comme l'exécution de l'application cliente est suspendue en attendant le résultat, il est difficile pour l'application de décider qu'elle voudrait annuler la commande en cours (c'est possible avec un gestionnaire de signaux mais pas autrement).

- `PQexec` ne peut renvoyer qu'une structure `PGresult`. Si la chaîne de commande soumise contient plusieurs commandes SQL, toutes les structures `PGresult` sont annulées par `PQexec`, sauf la dernière.

- `PQexec` récupère toujours le résultat entier de la commande, le mettant en cache dans une seule structure `PGresult`. Bien que cela simplifie la logique de la gestion des erreurs pour l'application, cela peut ne pas se révéler pratique pour les résultats contenant de nombreuses lignes.

Les applications qui n'apprécient pas ces limitations peuvent utiliser à la place les fonctions sous-jacentes à partir desquelles `PQexec` est construit : `PQsendQuery` et `PQgetResult`. Il existe aussi `PQsendQueryParams`, `PQsendPrepare`, `PQsendQueryPrepared`, `PQsendDescribePrepared` et `PQsendDescribePortal`, pouvant être utilisées avec `PQgetResult` pour dupliquer les fonctionnalités de respectivement `PQexecParams`, `PQprepare` , `PQexecPrepared`, `PQdescribePrepared` et `PQdescribePortal`.

`PQsendQuery`

Soumet une commande au serveur sans attendre le(s) résultat(s). 1 est renvoyé si la commande a été correctement envoyée et 0 dans le cas contraire (auquel cas, utilisez la fonction `PQerrorMessage` pour obtenir plus d'informations sur l'échec).

```
int PQsendQuery(PGconn *conn, const char *command);
```

Après un appel réussi à `PQsendQuery`, appelez `PQgetResult` une ou plusieurs fois pour obtenir les résultats. `PQsendQuery` ne peut pas être appelé de nouveau (sur la même connexion) tant que `PQgetResult` ne renvoie pas de pointeur NULL, indiquant que la commande a terminé.

Dans le mode pipeline, Cette fonction est interdite.

`PQsendQueryParams`

Soumet une commande et des paramètres séparés au serveur sans attendre le(s) résultat(s).

```
int PQsendQueryParams(PGconn *conn,
                      const char *command,
                      int nParams,
                      const Oid *paramTypes,
                      const char * const *paramValues,
                      const int *paramLengths,
                      const int *paramFormats,
                      int resultFormat);
```

Ceci est équivalent à `PQsendQuery` sauf que les paramètres de requêtes peuvent être spécifiés séparément à partir de la chaîne de requête. Les paramètres de la fonction sont gérés de façon identique à `PQexecParams`.

`PQsendPrepare`

Envoie une requête pour créer une instruction préparée avec les paramètres donnés et redonne la main sans attendre la fin de son exécution.

```
int PQsendPrepare(PGconn *conn,
   const char *stmtName,
   const char *query,
   int nParams,
   const Oid *paramTypes);
```

Ceci est la version asynchrone de `PQprepare` : elle renvoie 1 si elle a été capable d'envoyer la requête, 0 sinon. Après un appel terminé avec succès, appelez `PQgetResult` pour déterminer

si le serveur a créé avec succès l'instruction préparée. Les paramètres de la fonction sont gérés de façon identique à PQprepare .

PQsendQueryPrepared

Envoie une requête pour exécuter une instruction préparée avec des paramètres donnés sans attendre le(s) résultat(s).

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

Ceci est similaire à PQsendQueryParams mais la commande à exécuter est spécifiée en nommant une instruction préparée précédente au lieu de donner une chaîne contenant la requête. Les paramètres de la fonction sont gérés de façon identique à PQexecPrepared.

PQsendDescribePrepared

Soumet une requête pour obtenir des informations sur l'instruction préparée indiquée sans attendre sa fin.

```
        int PQsendDescribePrepared(PGconn *conn, const char
 *stmtName);
```

Ceci est la version asynchrone de PQdescribePrepared : elle renvoie 1 si elle a été capable d'envoyer la requête, 0 dans le cas contraire. Après un appel réussi, appelez PQgetResult pour obtenir les résultats. Les paramètres de la fonction sont gérés de façon identique à PQdescribePrepared.

PQsendDescribePortal

Soumet une requête pour obtenir des informations sur le portail indiqué sans attendre la fin de la commande.

```
        int PQsendDescribePortal(PGconn *conn, const char
 *portalName);
```

Ceci est la version asynchrone de PQdescribePortal : elle renvoie 1 si elle a été capable d'envoyer la requête, 0 dans le cas contraire. Après un appel réussi, appelez PQgetResult pour obtenir les résultats. Les paramètres de la fonction sont gérés de façon identique à PQdescribePortal.

PQgetResult

Attend le prochain résultat d'un appel précédent à PQsendQuery, PQsendQueryParams, PQsendPrepare, PQsendQueryPrepared, PQsendDescribePrepared, PQsendDescribePortal, ou PQpipelineSync, et le renvoie. Un pointeur NULL est renvoyé quand la commande est terminée et qu'il n'y aura plus de résultats.

```
PGresult *PQgetResult(PGconn *conn);
```

`PQgetResult` doit être appelé de façon répétée jusqu'à ce qu'il retourne un pointeur NULL indiquant que la commande s'est terminée. (Si appelée à un moment où aucune commande n'est active, `PQgetResult` renverra juste immédiatement un pointeur NULL). Chaque résultat non NULL provenant de `PQgetResult` devrait être traité en utilisant les mêmes fonctions d'accès à `PGresult` que celles précédemment décrites. N'oubliez pas de libérer chaque objet résultat avec `PQclear` quand vous en avez terminé. Notez que `PQgetResult` bloquera seulement si la commande est active et que les données nécessaires en réponse n'ont pas encore été lues par `PQconsumeInput`.

Dans le mode pipeline, `PQgetResult` retournera normalement jusqu'à l'arrivée d'une erreur ; pour les requêtes suivantes envoyées après celle qui a causé l'erreur jusqu'au point suivant de synchronisation (ce dernier exclus), un résultat spécial de type `PGRES_PIPELINE_ABORTED` sera renvoyé, et un pointeur nul sera renvoyé après cela. Quand le pointde synchronisation du pipeline est atteint, un résultat de type `PGRES_PIPELINE_SYNC` sera renvoyé. Le résultat de la requête suivant le point de synchronisation suit immédiatement (autrement dit aucun pointeur nul n'est renvoyé après le point de synchronisation).

> **Note**
>
> Même quand `PQresultStatus` indique une erreur fatale, `PQgetResult` doit être appelé jusqu'à ce qu'il renvoie un pointeur NULL pour permettre à libpq de traiter l'information sur l'erreur correctement.

Utiliser `PQsendQuery` et `PQgetResult` résout un des problèmes de `PQexec` : si une chaîne de commande contient plusieurs commandes SQL, les résultats de ces commandes peuvent être obtenus individuellement (ceci permet une simple forme de traitement en parallèle : le client peut gérer les résultats d'une commande alors que le serveur travaille sur d'autres requêtes de la même chaîne de commandes).

Une autre fonctionnalité fréquemment demandée, pouvant être obtenue avec `PQsendQuery` et `PQgetResult` est la récupération d'un gros résultat une ligne à la fois. Ceci est discuté dans Section 34.6.

Néanmoins, appeler `PQgetResult` causera toujours un blocage du client jusqu'à la fin de la prochaine commande SQL. Ceci est évitable en utilisant proprement deux fonctions supplémentaires :

`PQconsumeInput`

Si l'entrée est disponible à partir du serveur, la consomme.

```
int PQconsumeInput(PGconn *conn);
```

`PQconsumeInput` renvoie normalement 1 indiquant « aucune erreur », mais renvoie zéro s'il y a eu une erreur (auquel cas `PQerrorMessage` peut être consulté). Notez que le résultat ne dit pas si des données ont été récupérées en entrée. Après avoir appelé `PQconsumeInput`, l'application devrait vérifier `PQisBusy` et/ou `PQnotifies` pour voir si leur état a changé.

`PQconsumeInput` peut être appelée même si l'application n'est pas encore préparée à traiter un résultat ou une notification. La fonction lira les données disponibles et les sauvegardera dans un tampon indiquant ainsi qu'une lecture d'un `select()` est possible. L'application peut donc utiliser `PQconsumeInput` pour effacer la condition `select()` immédiatement, puis examiner les résultats à loisir.

`PQisBusy`

Renvoie 1 si une commande est occupée, c'est-à-dire que `PQgetResult` bloque en attendant une entrée. Un zéro indique que `PQgetResult` peut être appelé avec l'assurance de ne pas bloquer.

```
int PQisBusy(PGconn *conn);
```

`PQisBusy` ne tentera pas lui-même de lire les données à partir du serveur ; du coup, `PQconsumeInput` doit être appelé d'abord ou l'état occupé ne prendra jamais fin.

Une application typique de l'utilisation de ces fonctions aura une boucle principale utilisant `select()` ou `poll()` pour attendre que toutes les conditions soient remplies. Une des conditions sera la disponibilité des données à partir du serveur, ce qui signifie des données lisibles pour `select()` sur le descripteur de fichier identifié par `PQsocket`. Lorsque la boucle principale détecte la disponibilité de données, elle devrait appeler `PQconsumeInput` pour lire l'en-tête. Elle peut ensuite appeler `PQisBusy` suivi par `PQgetResult` si `PQisBusy` renvoie false (0). Elle peut aussi appeler `PQnotifies` pour détecter les messages `NOTIFY` (voir la Section 34.9).

Un client qui utilise `PQsendQuery`/`PQgetResult` peut aussi tenter d'annuler une commande en cours de traitement par le serveur ; voir la Section 34.7. Mais quelle que soit la valeur renvoyée par `PQcancel` , l'application doit continuer avec la séquence normale de lecture du résultat en utilisant `PQgetResult`. Une annulation réussie causera simplement une fin plus rapide de la commande.

En utilisant les fonctions décrites ci-dessus, il est possible d'éviter le blocage pendant l'attente de données du serveur. Néanmoins, il est toujours possible que l'application se bloque en attendant l'envoi vers le serveur. C'est relativement peu fréquent mais cela peut arriver si de très longues commandes SQL ou données sont envoyées (mais c'est bien plus probable si l'application envoie des données via `COPY IN`). Pour éviter cette possibilité et parvenir à des opérations de bases de données totalement non bloquantes, les fonctions supplémentaires suivantes peuvent être utilisées.

`PQsetnonblocking`

Initialise le statut non bloquant de la connexion.

```
int PQsetnonblocking(PGconn *conn, int arg);
```

Initialise l'état de la connexion à non bloquant si *`arg`* vaut 1 et à bloquant si *`arg`* vaut 0. Renvoie 0 si OK, -1 en cas d'erreur.

Dans l'état non bloquant, les appels réussis à `PQsendQuery`, `PQputline`, `PQputnbytes`, `PQputCopyData` et `PQendcopy` ne bloqueront pas ; leurs modifications seront stockées dans un tampon local de sortie jusqu'à ce qu'elles soient vidées sur disque. Les appels échoués renverront une erreur et devront être tentés de nouveau.

Notez que `PQexec` n'honore pas le mode non bloquant ; s'il est appelé, il agira d'une façon bloquante malgré tout.

`PQisnonblocking`

Renvoie le statut bloquant de la connexion à la base de données.

```
int PQisnonblocking(const PGconn *conn);
```

Renvoie 1 si la connexion est en mode non bloquant, 1 dans le cas contraire.

`PQflush`

Tente de vider les données des queues de sortie du serveur. Renvoie 0 en cas de succès (ou si la queue d'envoi est vide), -1 en cas d'échec quelle que soit la raison ou 1 s'il a été incapable d'envoyer encore toutes les données dans la queue d'envoi (ce cas arrive seulement si la connexion est non bloquante).

```
int PQflush(PGconn *conn);
```

Après avoir envoyé une commande ou des données dans une connexion non bloquante, appelez `PQflush`. S'il renvoie 1, attendez que la socket devienne prête en lecture ou en écriture. Si elle est prête en écriture, appelez de nouveau `PQflush`. Si elle est prête en lecture, appelez `PQconsumeInput`, puis appelez `PQflush`. Répétez jusqu'à ce que `PQflush` renvoie 0. (Il est nécessaire de vérifier si elle est prête en lecture, et de vidanger l'entrée avec `PQconsumeInput` car le serveur peut bloquer en essayant d'envoyer des données, par exemple des messages NOTICE, et ne va pas lire nos données tant que nous n'avons pas lu les siennes.) Une fois que `PQflush` renvoie 0, attendez que la socket soit disponible en lecture, puis lisez la réponse comme décrit ci-dessus.

34.5. Mode pipeline

Le mode pipeline de libpq permet aux applications d'envoyer une requête sans avoir à lire le résultat de la requête précédemment envoyée. En utilisant le mode pipeline, un client attendra moins le serveur car plusieurs requêtes/résultats peuvent être envoyés/reçus dans une seule transaction réseau.

Alors que le mode pipeline offre une amélioration significative des performances, les clients en écriture utilisant le mode pipeline sont plus complexes car cela implique de gérer une queue de requêtes en cours et de trouver quel résultat correspond à quelle requête dans la queue.

Le mode pipeline consomme généralement aussi plus de mémoire sur le client et le serveur, bien qu'une gestion attentionnée et agressive de la queue d'envoi/réception peut mitiger cela. Ceci s'applique que la connexion soit ou non en mode blocage ou non blocage.

Alors que l'API de pipeline de libpq a été introduite dans PostgreSQL 14, c'est une fonctionnalité du côté client qui ne requiert pas de support spécial côté serveur, et fonctionne avec tout serveur qui supporte le protocole de requêtes étendu v3. Pour plus d'informations, voir Section 55.2.4.

34.5.1. Utiliser le mode pipeline

Pour exécuter des pipelines, l'application doit basculer la connexion en mode pipeline, ce qui se fait avec `PQenterPipelineMode`. `PQpipelineStatus` peut être utilisé pour tester si le mode pipeline est actif. Dans le mode pipeline, seules les opérations asynchrones qui utilisent le protocole de requête étendue sont autorisées, les chaînes de commandes contenant plusieurs commandes SQL et `COPY` sont interdits. Utiliser les fonctions d'exécution de commandes synchrones telles que `PQfn`, `PQexec`, `PQexecParams`, `PQprepare`, `PQexecPrepared`, `PQdescribePrepared`, `PQdescribePortal`, renvoie une erreur. `PQsendQuery` est aussi interdite parce qu'elle utilise le protocole de requête simple. Une fois que toutes les commandes envoyées ont eu leurs résultats traités, l'application peut retourner dans le mode sans pipeline avec `PQexitPipelineMode`.

> **Note**
>
> Il est préférable d'utiliser le mode pipeline avec libpq dans le mode sans blocage. Si utilisé en mode blocage, il est possible qu'un deadlock client/serveur survienne. [3]

34.5.1.1. Exécuter des requêtes

Après être entré en mode pipeline, l'application envoie les requêtes en utilisant `PQsendQueryParams` ou un des équivalents pour requêtes préparées `PQsendQueryPrepared`. Ces requêtes sont mises en queue côté client jusqu'au vidage du serveur ; ceci survient quand

[3] Le client bloquera les tentatives d'envoyer des requêtes au serveur, mais le serveur bloquera en essayant d'envoyer les résultats au client des requêtes qu'il a déjà traité. Ceci survient seulement quand le client envoie suffisamment de requêtes pour remplir à la fois son tampon en sortie et le tampon de réception du serveur avant qu'il bascule pour traiter les entrées du serveur, mais il est difficile de prédire exactement quand cela arrivera.

PQpipelineSync est utilisé pour établir un point de synchronisation dans le pipeline ou quand PQflush est appelé. Les fonctions PQsendPrepare, PQsendDescribePrepared et PQsendDescribePortal fonctionnent aussi en mode pipeline. Le traitement des résultats est décrit ci-dessous.

Le serveur exécute les requêtes et envoie les résultats dans l'ordre dans lequel le client les a envoyé. Le serveur commencera immédiatement l'exécution des commandes dans le pipeline. Notez que les résultats sont mis en tampon du côté du serveur ; le serveur vide ce tampon quand un point de synchronisation est établi avec PQpipelineSync ou quand PQsendFlushRequest est appelé. Si l'exécution d'une requête rencontre une erreur, le serveur annule la transaction en cours et n'exécute pas toute autre commande dans la queue jusqu'au prochain point de synchronisation ; un résultat PGRES_PIPELINE_ABORTED est produit pour chacune de ces commandes. (Ceci reste vrai même si les commandes dans le pipeline vont annuler la transaction.) Le traitement des requêtes continue après le point de synchronisation.

Il est possible qu'une opération dépende des résultats d'une opération précédente ; par exemple, une requête pourrait définir une table que la prochaine requête du même pipeline utilise. De façon similaire, une application pourrait créer une requête préparée nommée et l'exécuter dans les requêtes suivantes du même pipeline.

34.5.1.2. Traitement des résultats

Pour traiter le résultat d'une requête dans un pipeline, l'application appelle PQgetResult de façon répété et gère chaque résultat jusqu'à ce que PQgetResult renvoie null. Le résultat de la prochaine requête dans le pipeline sera alors récupéré en utilisant de nouveau PQgetResult et le cycle sera répété. L'application gère les résultats de requête individuelle comme d'habitude. Quand les résultats de toutes les requêtes du pipeline ont été renvoyés, PQgetResult renvoie un résultat contenant la valeur de statut PGRES_PIPELINE_SYNC

Le client peut choisir de différer le traitement des résultats jusqu'à ce que le pipeline complet ait été envoyé ou de continuer en envoyant d'autres requêtes dans le pipeline ; voir Section 34.5.1.4.

Pour entrer en mode single-row, appelez PQsetSingleRowMode avant de récupérer les résultats avec PQgetResult. Cette sélection de mode est efficace seulement pour la requête en cours de traitement. Pour plus d'informations sur l'utilisation de PQsetSingleRowMode, référez-vous à Section 34.6.

PQgetResult se comporte comme pour le traitement asynchrone normal, sauf qu'il peut contenir les nouveaux types PGresult, à savoir PGRES_PIPELINE_SYNC et PGRES_PIPELINE_ABORTED. PGRES_PIPELINE_SYNC est indiqué exactement une fois pour chaque PQpipelineSync au point correspondant dans le pipeline. PGRES_PIPELINE_ABORTED est émis à la place d'un résultat de requête normal pour la première erreur et tous les résultats suivants jusqu'au prochain PGRES_PIPELINE_SYNC ; voir Section 34.5.1.3.

PQisBusy, PQconsumeInput, etc opèrent normalement lors du traitement ds résultats du pipeline. En particulier, un appel à PQisBusy au milieu d'un pipeline renvoie 0 si les résultats de toutes les requêtes lancées jusqu'à maintenant ont été consommés.

libpq ne fournit pas d'informations à l'application quant à la requête en cours de traitement (sauf que PQgetResult renvoie null pour indiquer que nous commençons l'envoi des résultats de la requête suivante). L'application doit garder la trace de l'ordre dans lequel elle a envoyé les requête, pour les associer avec les résultats correspondants. Les applications utiliseront typiquement une machine à état ou une queue FIFO pur ça.

34.5.1.3. Gestion des erreurs

Du point de vue du client, après que PQresultStatus renvoie PGRES_FATAL_ERROR, le pipeline est considéré comme annulé. PQresultStatus renverra un résultat

PGRES_PIPELINE_ABORTED pour chaque opération restant en queue dans un pipeline annulé. Le résultat pour PQpipelineSync est rapporté comme PGRES_PIPELINE_SYNC pour signaler la fin du pipeline annulé et le retour au traitement normal des résultats.

Le client *doit* traiterles résultats avec PQgetResult lors de la récupération après erreur.

Si le pipeline utilisait une transaction implicite, alors les opérations qui se sont déjà exécutées sont annulées et les opérations en queue après l'opération annulée sont complètement ignorées. Le même comportement reste si le pipeline commence et valide une seule transaction explicite (la première requête est BEGIN et la dernière est COMMIT) sauf que la session reste dans un état de transaction annulée à la fin du pipeline. Si un pipeline contient *plusieurs transactions explicites*, toutes les transactions validées avant l'erreur restent validées, la transaction actuellement en cours est annulée, et toutes les opérations suivantes sont complètement ignorées, ceci incluant les transactions suivantes. Si unpoint de synchronisation du pipeline survient avec un bloc de transaction explicite dans un état annulé, le pipeline suivant deviendra immédiatement annulé sauf si la commande suivante place la transaction en mode normal avec ROLLBACK.

> **Note**
>
> Le client ne doit pas supposer que le travail est validé quand il *envoie* un COMMIT -- seulment quand le résultat correspondant est reçu pour confirmer que la validation est terminée. Comme des erreurs arrivent en asynchrone, l'application a besoin d'être capable de recommencer depuis le dernier changement validé *reçu* et renvoie le travail effectué après ce point si quelque chose ne s'est pas bien passé.

34.5.1.4. Interleaving Result Processing and Query Dispatch

Pour éviter des verrous sur des pipelines importants, le client devrait être structuré autour d'une boucle d'événements non bloquante en utilisant les possibilités du système d'exploitation, tel que select, poll, WaitForMultipleObjectEx, etc.

L'application doit généralement maintenir une queue de travail restant à déployer et une queue de travail qui a déjà été déployée mais dont les résultats n'ont pas encore été traités. Quand le socket est accessible en écriture, il pourrait déployer plus de travail. Quand le socket est lisible, il pourrait lire les résultats et les traiter, les faisant correspondre à la prochaine entrée dans la queue de résultats correspondante. Suivant la mémoire disponible, les résultats provenant du socket doivent être lus fréquemment : il n'est pas nécessaire d'attendre la fin du pipeline pour lire les résultats. Les pipelines doivent être placés en unités logiquesde travail, habituellement (mais pas nécessairement) une transaction par pipeline. Il n'est pas besoin de quitter le mode pipeline et de ré-entrer dans ce mode entre les pipelines ou d'autre la fin d'un pipeline avant d'envoyer le suivant.

Un exemple utilisant select() et une simple machine à état pour trace le travail envoyé et reçu se trouve dans src/test/modules/libpq_pipeline/libpq_pipeline.c à partir de la distribution des sources de PostgreSQL.

34.5.2. Fonctions associées avec le mode pipeline

PQpipelineStatus

Renvoie le statut actuel du mode pipeline de la connexion libpq.

```
PGpipelineStatus PQpipelineStatus(const PGconn *conn);
```

PQpipelineStatus peut renvoyer une des valeurs suivantes :

`PQ_PIPELINE_ON`

La connexion libpq est en mode pipeline.

`PQ_PIPELINE_OFF`

La connexion libpq n'est *pas* en mode pipeline.

`PQ_PIPELINE_ABORTED`

La connexion libpq est en mode pipeline et une erreur est survenue lors du traitement du pipeline actuel. Le drapeau d'annulation est effacé quand `PQgetResult` renvoie un résultat de type `PGRES_PIPELINE_SYNC`.

`PQenterPipelineMode`

Cause l'entrée en mode pipeline de la connexion si elle est actuellement inactive ou déjà en mode pipeline.

```
int PQenterPipelineMode(PGconn *conn);
```

Renvoie 1 en cas de succès. Renvoie 0 et n'a pas d'effet si la connection est actuellement active, par exemple elle a un résultat prêt ou elle est en attente d'entrée du serveur, etc. Cette fonction n'envoie rien au serveur, elle modifie juste l'état de la connexion libpq.

`PQexitPipelineMode`

Cause la sortie du mode pipeline pour la connexion si elle est en mode pipeline avec une queue vide et aucun résultat en attente.

```
int PQexitPipelineMode(PGconn *conn);
```

Renvoie 1 en cas de succès. Renvoie 1 et ne fait aucune action si la connexion n'est pas en mode pipeline. Si l'instruction en cours n'a pas terminé son traitement ou si `PQgetResult` n'a pas été appelé pour récupérer les résultats de toutes les requêtes envoyées précédemment, renvoie 0 (auquel cas, utilisez `PQerrorMessage` pour obtenir plus d'informations sur l'échec).

`PQpipelineSync`

Marque un point de synchronisation dans un pipeline en envoyant un message sync et en vidant le tampon d'envoi. Ceci sert comme délimiteur d'une transaction implicite et d'un point de restauration après erreur ;voir Section 34.5.1.3.

```
int PQpipelineSync(PGconn *conn);
```

Renvoie 1 en cas de succès. Renvoie 0 si la connexion n'est pas dans le mode pipeline ou est en cours d'envoi d'un message sync en échec.

`PQsendFlushRequest`

Envoie une requête au serveur pour vider son tampon en sortie.

```
int PQsendFlushRequest(PGconn *conn);
```

Renvoie 1 en cas de succès. Renvoie 0 en cas d'échec.

Le serveur vide son tampon en sortie automatiquement comme conséquence de l'appel à `PQpipelineSync` ou pour toute requête quand la connexion n'utilise pas le mode pipeline ;

cette fonction est utile pour forcer le serveur à vider son tampon en sortie dans le mode pipeline sans établir de point de synchronisation. Notez que la requête n'est pas elle-même vidée du serveur automatiquement ; utilisez `PQflush` si nécessaire.

34.5.3. Quand utiliser le mode pipeline

Tout comme le mode de requête asynchrone, il n'y a pas de surcharge significative pour les performances lors de l'utilisation du mode pipeline. Il augmente la complexité de l'application cliente, et une grande attention est requise pour éviter les deadlocks client/serveur, mais le mode pipeline peut offrir des améliorations considérables des performances, en échange d'une utilisation mémoire accrue.

Le mode pipeline est plus utile quand le serveur est distant, autrement dit quand la latence réseau (« durée du ping ») est importante, et aussi quand plusieurs petites opérations sont en cours d'exécution en succession rapide. Il y a moins d'intérêt à utiliser les commandes en pipeline quand chaque requête prend plusieurs fois le temps des aller/retours client/serveur pour s'exécuter. Une exécution d'une opération de 100 instructions sur un serveur avec des aller/retour à 300 ms va prendre 30 secondes de latence réseau sans pipeline. Avec le pipeline, celle pourrait prendre 0,3 seconde à attendre les résultats du serveur.

Utilisez les commandes de pipeline quand votre application fait plein de petites opérations `INSERT`, `UPDATE` et `DELETE` qui ne peuvent pas facilement être transformées en opérations sur des ensembles ou en une opération `COPY`.

Le mode pipeline n'est pas utile quand l'information d'une opération est requise par le client pour produire la prochaine opération. Dans de tels cas, le client devra introduire un point de synchronisation et attendre un aller/retour complet client/serveur pour obtenir le résultat nécessaire. Néanmoins, il est souvent possible d'ajuster la conception du client pour échanger les informations requises côté serveur. Les cycles lecture/modification/écriture sont tout spécialement de bon candidat ; par exemple :

```
BEGIN;
SELECT x FROM mytable WHERE id = 42 FOR UPDATE;
-- result: x=2
-- client adds 1 to x:
UPDATE mytable SET x = 3 WHERE id = 42;
COMMIT;
```

pourrait être réalisé plus efficacement avec :

```
UPDATE mytable SET x = x + 1 WHERE id = 42;
```

Envoyer dans un pipeline est moins utile et plus complexe quand un seul pipeline contient plusieurs transactions (voir Section 34.5.1.3).

34.6. Récupérer le résultats des requêtes ligne par ligne

D'habitude, libpq récupère le résultat complet d'une commande SQL et la renvoie à l'application sous la forme d'une seule structure `PGresult`. Ce peut être impraticable pour les commandes renvoyant un grand nombre de lignes. Dans de tels cas, les applications peuvent utiliser `PQsendQuery` et `PQgetResult` dans le *mode ligne-à-ligne*. Dans ce mode, les lignes du résultat sont renvoyées à l'application une par une, au fur et à mesure qu'elles sont reçues du serveur.

Pour entrer dans le mode ligne-à-ligne, appelez `PQsetSingleRowMode` immédiatement après un appel réussi à `PQsendQuery` (ou une fonction similaire). Cette sélection de mode ne fonctionne que pour la requête en cours d'exécution. Puis appelez `PQgetResult` de façon répétée, jusqu'à

ce qu'elle renvoit NULL, comme documenté dans Section 34.4. Si la requête renvoit des lignes, elles sont renvoyées en tant qu'objets `PGresult` individuels, qui ressemblent à des résultats de requêtes standards en dehors du fait qu'elles ont le code de statut `PGRES_SINGLE_TUPLE` au lieu de `PGRES_TUPLES_OK`. Après la dernière ligne, ou immédiatement si la requête ne renvoit aucune ligne, un objet à zéro ligne avec le statut `PGRES_TUPLES_OK` est renvoyé ; c'est le signal qu'aucune autre ligne ne va arriver. (Notez cependant qu'il est toujours nécessaire de continuer à appeler `PQgetResult` jusqu'à ce qu'elle renvoit NULL.) Tous les objets `PGresult` contiendront les mêmes données de description de lignes (noms de colonnes, types, etc.) qu'un objet `PGresult` standard aurait pour cette requête. Chaque objet doit être libéré avec la fonction `PQclear` comme d'ordinaire.

Lors de l'utilisation du mode pipeline, le mode single-row doit être activé pour chaque requête dans le pipeline avant de récupérer les résultats pour cette requête avec `PQgetResult`. Voir Section 34.5 pour plus d' informations.

`PQsetSingleRowMode`

Sélectionne le mode ligne simple pour la requête en cours d'exécution.

```
int PQsetSingleRowMode(PGconn *conn);
```

Cette fonction peut seulement être appelée immédiatement après `PQsendQuery` ou une de ses fonctions sœurs, avant toute autre opération sur la connexion comme `PQconsumeInput` ou `PQgetResult`. Si elle est appelée au bon moment, la fonction active le mode ligne-à-ligne pour la requête en cours et renvoit 1. Sinon, le mode reste inchangé et la fonction renvoit 0. Dans tous les cas, le mode retourne à la normale après la fin de la requête en cours.

> **Attention**
>
> Lors du traitement d'une requête, le serveur peut renvoyer quelques lignes puis rencontrer une erreur, causant l'annulation de la requête. D'ordinaire, la bibliothèque partagée libpq jette ces lignes et renvoit une erreur. Avec le mode ligne-à-ligne, des lignes ont déjà pu être envoyées à l'application. Du coup, l'application verra quelques objets `PGresult` de statut `PGRES_SINGLE_TUPLE` suivis par un objet de statut `PGRES_FATAL_ERROR`. Pour un bon comportement transactionnel, l'application doit être conçue pour invalider ou annuler tout ce qui a été fait avec les lignes précédemment traitées si la requête finit par échouer.

34.7. Annuler des requêtes en cours d'exécution

Une application client peut demander l'annulation d'une commande qui est toujours en cours d'exécution par le serveur en utilisant les fonctions décrites dans cette section.

`PQgetCancel`

Crée une structure de données contenant les informations nécessaires à l'annulation d'une commande lancée sur une connexion particulière à la base de données.

```
PGcancel *PQgetCancel(PGconn *conn);
```

`PQgetCancel` crée un objet `PGcancel` à partir d'un objet connexion `PGconn`. Il renverra `NULL` si le paramètre *`conn`* donné est `NULL` ou est une connexion invalide. L'objet `PGcancel`

est une structure opaque qui n'a pas pour but d'être accédé directement par l'application ; elle peut seulement être passée à `PQcancel` ou `PQfreeCancel`.

`PQfreeCancel`

Libère une structure de données créée par `PQgetCancel`.

```
void PQfreeCancel(PGcancel *cancel);
```

`PQfreeCancel` libère un objet donné par `PQgetCancel`.

`PQcancel`

Demande que le serveur abandonne l'exécution de la commande en cours.

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

La valeur renvoyée est 1 si la demande d'annulation a été correctement envoyée et 0 sinon. Si non, *`errbuf`* contient un message d'erreur expliquant pourquoi. *`errbuf`* doit être un tableau de caractères d'une taille de *`errbufsize`* octets (la taille recommandée est de 256 octets).

Un envoi réussi ne garantit pas que la demande aura un quelconque effet. Si l'annulation est réelle, la commande en cours terminera plus tôt et renverra une erreur. Si l'annulation échoue (disons, parce que le serveur a déjà exécuté la commande), alors il n'y aura aucun résultat visible.

`PQcancel` peut être invoqué de façon sûre par le gestionnaire de signaux si *`errbuf`* est une variable locale dans le gestionnaire de signaux. L'objet `PGcancel` est en lecture seule en ce qui concerne `PQcancel`, pour qu'il puisse aussi être appelé à partir d'un thread séparé de celui manipulant l'objet `PGconn`.

`PQrequestCancel`

`PQrequestCancel` est une variante obsolète de `PQcancel`.

```
int PQrequestCancel(PGconn *conn);
```

Demande au serveur l'abandon du traitement de la commande en cours d'exécution. Elle opère directement sur l'objet `PGconn` et, en cas d'échec, stocke le message d'erreur dans l'objet `PGconn` (d'où il peut être récupéré avec `PQerrorMessage`). Bien qu'il s'agisse de la même fonctionnalité, cette approche n'est pas sûre dans les programmes multi-threads et les gestionnaires de signaux car il est possible que la surcharge du message d'erreur de `PGconn` gênera l'opération en cours sur la connexion.

34.8. Interface rapide (Fast Path)

PostgreSQL fournit une interface rapide (*Fast Path*) pour des appels de fonctions simples au serveur.

> **Avertissement**
>
> Cette interface n'est pas sûr et ne devrait pas être utilisée. Quand *`result_is_int`* est configuré à `0`, `PQfn` pourrait écrire des données au delà de la fin de *`result_buf`*, que le buffer est suffisamment de place ou pas pour le nombre d'octets demandé. De plus, elle est obsolète car vous pourriez réaliser les mêmes choses avec des performances similaires et plus de fonctionnalités en initialisant une instruction préparée pour définir l'appel de fonction.

Puis, exécuter l'instruction avec une transmission binaire des paramètres et des substitutions de résultats pour un appel de fonction à chemin rapide.

La fonction `PQfn` demande l'exécution d'une fonction du serveur via l'interface de chemin rapide :

```
PGresult* PQfn(PGconn* conn,
                   int fnid,
                   int *result_buf,
                   int *result_len,
                   int result_is_int,
                   const PQArgBlock *args,
                   int nargs);

    typedef struct
    {
        int len;
        int isint;
        union
        {
            int *ptr;
            int integer;
        } u;
    } PQArgBlock;
```

L'argument *`fnid`* est l'OID de la fonction à exécuter. *`args`* et *`nargs`* définissent les paramètres à passer à la fonction ; ils doivent correspondre à la liste d'arguments déclarés de la fonction. Quand le champ *`isint`* d'une structure est vrai, la valeur de *`u.integer`* est envoyée au serveur en tant qu'entier de la longueur indiquée (qui doit être 2 ou 4 octets) ; les permutations d'octets adéquates sont opérées. Quand *`isint`* est faux, le nombre d'octets indiqué sur *`*u.ptr`* est envoyé sans traitement ; les données doivent être dans le format attendu par le serveur pour la transmission binaire du type de données de l'argument de la fonction. (La déclaration de *`u.ptr`* en tant que type `int *` est historique ; il serait préférable de la considérer comme un `void *`.) *`result_buf`* pointe vers le tampon dans lequel placer le code de retour de la fonction. L'appelant doit avoir alloué suffisamment d'espace pour stocker le code de retour (il n'y a pas de vérification !). La longueur effective du résultat en octet sera renvoyée dans l'entier pointé par *`result_len`*. Si un résultat sur un entier de 2 ou 4 octets est attendu, initialisez *`result_is_int`* à 1, sinon initialisez-le à 0. Initialiser *`result_is_int`* à 1 fait que libpq permute les octets de la valeur si nécessaire, de façon à ce que la bonne valeur `int` soit délivrée pour la machine cliente ; notez qu'un entier sur quatre octets est fourni dans *`*result_buf`* pour chaque taille de résultat autorisée. Quand *`result_is_int`* vaut 0, la chaîne d'octets au format binaire envoyée par le serveur est renvoyée non modifiée. (Dans ce cas, il est préférable de considérer *`result_buf`* comme étant du type `void *`.)

`PQfn` renvoie toujours un pointeur `PGresult` valide, avec un statut `PGRES_COMMAND_OK` en cas de succès et `PGRES_FATAL_ERROR` si un problème a été rencontré. L'état du résultat devrait être vérifié avant que le résultat ne soit utilisé. Le demandeur est responsable de la libération de la structure `PGresult` avec `PQclear` lorsque celle-ci n'est plus nécessaire.

Pour passer un argument NULL à la fonction, configurez le champ *`len`* de cette structure à `-1` ; les champs *`isint`* et *`u`* sont alors hors sujet.

Si la fonction renvoie NULL, *`*result_len`* est configuré à `-1`, et *`*result_buf`* n'est pas modifié.

Notez qu'il n'est pas possible de gérer des ensembles de résultats en utilisant cette interface. De plus, la fonction doit être une fonction standard, par une fonction d'agrégat ou une fonction de fenêtrage.

34.9. Notification asynchrone

PostgreSQL propose des notifications asynchrones via les commandes LISTEN et NOTIFY. Une session cliente enregistre son intérêt dans un canal particulier avec la commande LISTEN (et peut arrêter son écoute avec la commande UNLISTEN). Toutes les sessions écoutant un canal particulier seront notifiées de façon asynchrone lorsqu'une commande NOTIFY avec ce nom de canal sera exécutée par une session. Une chaîne de « charge » peut être renseignée pour fournir des données supplémentaires aux processus en écoute.

Les applications libpq soumettent les commandes LISTEN, UNLISTEN et NOTIFY comme des commandes SQL ordinaires. L'arrivée des messages NOTIFY peut être détectée ensuite en appelant PQnotifies.

La fonction PQnotifies renvoie la prochaine notification à partir d'une liste de messages de notification non gérés reçus à partir du serveur. Il renvoie un pointeur NULL s'il n'existe pas de notification en attente. Une fois qu'une notification est renvoyée à partir de PQnotifies, elle est considérée comme étant gérée et sera supprimée de la liste des notifications.

```
PGnotify* PQnotifies(PGconn *conn);

   typedef struct pgNotify
   {
       char *relname;              /* nom du canal de la
 notification */
       int  be_pid;                /* ID du processus serveur
 notifiant */
       char *extra;                /* chaîne de charge pour la
 notification */
   } PGnotify;
```

Après avoir traité un objet PGnotify renvoyé par PQnotifies, assurez-vous de libérer le pointeur PQfreemem. Il est suffisant de libérer le pointeur PGnotify ; les champs relname et extra ne représentent pas des allocations séparées (le nom de ces champs est historique ; en particulier, les noms des canaux n'ont pas besoin d'être liés aux noms des relations.)

Exemple 34.2 donne un programme d'exemple illustrant l'utilisation d'une notification asynchrone.

PQnotifies ne lit pas réellement les données à partir du serveur ; il renvoie simplement les messages précédemment absorbés par une autre fonction de libpq. Dans les anciennes versions de libpq, la seule façon de s'assurer une réception à temps des messages NOTIFY consistait à soumettre constamment des commandes de soumission, même vides, puis de vérifier PQnotifies après chaque PQexec. Bien que ceci fonctionnait, cela a été abandonné car un gaspillage de ressources.

Une meilleure façon de vérifier les messages NOTIFY lorsque vous n'avez pas de commandes utiles à exécuter est d'appeler PQconsumeInput puis de vérifier PQnotifies. Vous pouvez utiliser select() pour attendre l'arrivée des données à partir du serveur, donc sans utiliser de CPU sauf lorsqu'il y a quelque chose à faire (voir PQsocket pour obtenir le numéro du descripteur de fichiers à utiliser avec select()). Notez que ceci fonctionnera que vous soumettiez les commandes avec PQsendQuery/PQgetResult ou que vous utilisiez simplement PQexec. Néanmoins, vous devriez vous rappeler de vérifier PQnotifies après chaque PQgetResult ou PQexec pour savoir si des notifications sont arrivées pendant le traitement de la commande.

34.10. Fonctions associées à la commande COPY

Dans PostgreSQL, la commande COPY a des options pour lire ou écrire à partir de la connexion réseau utilisée par libpq. Les fonctions décrites dans cette section autorisent les applications à prendre avantage de cette capacité en apportant ou en consommant les données copiées.

Le traitement complet est le suivant. L'application lance tout d'abord la commande SQL COPY via PQexecPQexec ou une des fonctions équivalentes. La réponse à ceci (s'il n'y a pas d'erreur dans la commande) sera un objet PGresult avec un code de retour PGRES_COPY_OUT ou PGRES_COPY_IN (suivant la direction spécifiée pour la copie). L'application devrait alors utiliser les fonctions de cette section pour recevoir ou transmettre des lignes de données. Quand le transfert de données est terminé, un autre objet PGresult est renvoyé pour indiquer le succès ou l'échec du transfert. Son statut sera PGRES_COMMAND_OK en cas de succès et PGRES_FATAL_ERROR si un problème a été rencontré. À ce point, d'autres commandes SQL peuvent être exécutées via PQexec (il n'est pas possible d'exécuter d'autres commandes SQL en utilisant la même connexion tant que l'opération COPY est en cours).

Si une commande COPY est lancée via PQexec dans une chaîne qui pourrait contenir d'autres commandes supplémentaires, l'application doit continuer à récupérer les résultats via PQgetResult après avoir terminé la séquence COPY. C'est seulement quand PQgetResult renvoie NULL que vous pouvez être certain que la chaîne de commandes PQexec est terminée et qu'il est possible de lancer d'autres commandes.

Les fonctions de cette section devraient seulement être exécutées pour obtenir un statut de résultat PGRES_COPY_OUT ou PGRES_COPY_IN à partir de PQexec ou PQgetResult.

Un objet PGresult gérant un de ces statuts comporte quelques données supplémentaires sur l'opération COPY qui commence. Les données supplémentaires sont disponibles en utilisant les fonctions qui sont aussi utilisées en relation avec les résultats de requêtes :

PQnfields

Renvoie le nombre de colonnes (champs) à copier.

PQbinaryTuples

0 indique que le format de copie complet est textuel (lignes séparées par des retours chariots, colonnes séparées par des caractères de séparation, etc). 1 indique que le format de copie complet est binaire. Voir COPY pour plus d'informations.

PQfformat

Renvoie le code de format (0 pour le texte, 1 pour le binaire) associé avec chaque colonne de l'opération de copie. Les codes de format par colonne seront toujours zéro si le format de copie complet est textuel, mais le format binaire supporte à la fois des colonnes textuelles et des colonnes binaires (néanmoins, avec l'implémentation actuelle de COPY, seules les colonnes binaires apparaissent dans une copie binaire donc pour le moment les formats par colonnes correspondent toujours au format complet).

34.10.1. Fonctions d'envoi de données pour COPY

Ces fonctions sont utilisées pour envoyer des données lors d'un COPY FROM STDIN. Elles échoueront si elles sont appelées alors que la connexion ne se trouve pas dans l'état COPY_IN.

PQputCopyData

Envoie des données au serveur pendant un état COPY_IN.

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

Transmet les données de COPY dans le tampon spécifié (*buffer*), sur *nbytes* octets, au serveur. Le résultat vaut 1 si les données ont été placées dans la queue, zéro si elles n'ont pas été

placées dans la queue à cause de tampons pleins (cela n'arrivera qu'en mode non bloquant) ou -1 si une erreur s'est produite. (Utilisez PQerrorMessage pour récupérer des détails si la valeur de retour vaut -1. Si la valeur vaut zéro, attendez qu'il soit prêt à écrire et ré-essayez).

L'application peut diviser le flux de données de COPY dans des tampons de taille adéquate. Les limites des tampons n'ont pas de signification sémantique lors de l'envoi. Le contenu du flux de données doit correspondre au format de données attendu par la commande COPY ; voir COPY pour des détails.

PQputCopyEnd

Envoie une indication de fin de transfert au serveur lors de l'état COPY_IN.

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

Termine l'opération COPY_IN avec succès si *errormsg* est NULL. Si *errormsg* n'est pas NULL alors COPY est passé en échec, avec la chaîne pointée par *errormsg* comme message d'erreur. (Mais ne pas supposer que ce message d'erreur précis proviendra du serveur car le serveur pourrait avoir déjà échoué sur la commande COPY pour des raisons qui lui sont propres).

Le résultat est 1 si le message de terminaison a été envoyé ; ou en mode non bloquant, cela peut seulement indiquer que le message de terminaison a été correctement mis en file d'attente. (En mode non bloquant, pour être certain que les données ont été correctement envoyées, vous devriez ensuite attendre que le mode écriture soit disponible puis appeler PQflush, à répéter jusqu'à ce que 0 soit renvoyé). Zéro indique que la fonction n'a pas pu mettre en file d'attente le message de terminaison à cause d'une file pleine ; ceci ne peut survenir qu'en mode non bloquant. (Dans ce cas, attendez que le mode écriture soit disponible puis rappelez à nouveau la fonction PQputCopyEnd). Si une erreur physique survient, -1 est renvoyé ; vous pouvez alors appeler PQerrorMessage pour avoir plus de détails sur l'erreur.

Après un appel réussi à PQputCopyEnd, appelez PQgetResult pour obtenir le statut de résultat final de la commande COPY. Vous pouvez attendre que le résultat soit disponible de la même façon. Puis, retournez au fonctionnement normal.

34.10.2. Fonctions de réception des données de COPY

Ces fonctions sont utilisées pour recevoir des données lors d'un COPY TO STDOUT. Elles échoueront si elles sont appelées alors que la connexion n'est pas dans l'état COPY_OUT

PQgetCopyData

Reçoit des données à partir du serveur lors d'un état COPY_OUT.

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

Tente d'obtenir une autre ligne de données du serveur lors d'une opération COPY. Les données ne sont renvoyées qu'une ligne à la fois ; si seulement une ligne partielle est disponible, elle n'est pas renvoyée. Le retour d'une ligne avec succès implique l'allocation d'une portion de mémoire pour contenir les données. Le paramètre *buffer* ne doit pas être NULL. **buffer* est initialisé pour pointer vers la mémoire allouée ou vers NULL au cas où aucun tampon n'est renvoyé. Un tampon résultat non NULL devra être libéré en utilisant PQfreemem lorsqu'il ne sera plus utile.

Lorsqu'une ligne est renvoyée avec succès, la valeur de retour est le nombre d'octets de la donnée dans la ligne (et sera donc toujours supérieur à zéro). La chaîne renvoyée est toujours terminée

par un octet nul bien que ce ne soit utile que pour les COPY textuels. Un résultat zéro indique que la commande COPY est toujours en cours mais qu'aucune ligne n'est encore disponible (ceci est seulement possible lorsque *async* est vrai). Un résultat -1 indique que COPY a terminé. Un résultat -2 indique qu'une erreur est survenue (consultez PQerrorMessage pour en connaître la raison).

Lorsque *async* est vraie (différent de zéro), PQgetCopyData ne bloquera pas en attente d'entrée ; il renverra zéro si COPY est toujours en cours mais qu'aucune ligne n'est encore disponible. (Dans ce cas, attendez qu'il soit prêt à lire puis appelez PQconsumeInput avant d'appeler PQgetCopyData de nouveau). Quand *async* est faux (zéro), PQgetCopyData bloquera tant que les données ne seront pas disponibles ou tant que l'opération n'aura pas terminée.

Après que PQgetCopyData a renvoyé -1, appelez PQgetResult pour obtenir le statut de résultat final de la commande COPY. On peut attendre la disponibilité de ce résultat comme d'habitude. Puis, retournez aux opérations habituelles.

34.10.3. Fonctions obsolètes pour COPY

Ces fonctions représentent d'anciennes méthodes de gestion de COPY. Bien qu'elles fonctionnent toujours, elles sont obsolètes à cause de leur pauvre gestion des erreurs, des méthodes inadéquates de détection d'une fin de transmission, et du manque de support des transferts binaires et des transferts non bloquants.

PQgetline

Lit une ligne de caractères terminée par un retour chariot (transmis par le serveur) dans un tampon de taille *length*.

```
int PQgetline(PGconn *conn,
                      char *buffer,
                      int length);
```

Cette fonction copie jusqu'à *length*-1 caractères dans le tampon et convertit le retour chariot en un octet nul. PQgetline renvoie EOF à la fin de l'entrée, 0 si la ligne entière a été lue et 1 si le tampon est complet mais que le retour chariot à la fin n'a pas encore été lu.

Notez que l'application doit vérifier si un retour chariot est constitué de deux caractères \., ce qui indique que le serveur a terminé l'envoi des résultats de la commande COPY. Si l'application peut recevoir des lignes de plus de *length*-1 caractères, une attention toute particulière est nécessaire pour s'assurer qu'elle reconnaisse la ligne \. correctement (et ne confond pas, par exemple, la fin d'une longue ligne de données pour une ligne de terminaison).

PQgetlineAsync

Lit une ligne de données COPY (transmise par le serveur) dans un tampon sans blocage.

```
int PQgetlineAsync(PGconn *conn,
                           char *buffer,
                           int bufsize);
```

Cette fonction est similaire à PQgetline mais elle peut être utilisée par des applications qui doivent lire les données de COPY de façon asynchrone, c'est-à-dire sans blocage. Après avoir lancé la commande COPY et obtenu une réponse PGRES_COPY_OUT, l'application devrait appeler PQconsumeInput et PQgetlineAsync jusqu'à ce que le signal de fin des données soit détecté.

Contrairement à PQgetline, cette fonction prend la responsabilité de détecter la fin de données.

À chaque appel, PQgetlineAsync renverra des données si une ligne de données complète est disponible dans le tampon d'entrée de libpq. Sinon, aucune ligne n'est renvoyée jusqu'à l'arrivée du reste de la ligne. La fonction renvoie -1 si le marqueur de fin de copie des données a été reconnu, 0 si aucune donnée n'est disponible ou un nombre positif indiquant le nombre d'octets renvoyés. Si -1 est renvoyé, l'appelant doit ensuite appeler PQendcopy puis retourner aux traitements habituels.

Les données renvoyées ne seront pas étendues au-delà de la limite de la ligne. Si possible, une ligne complète sera retournée en une fois. Mais si le tampon offert par l'appelant est trop petit pour contenir une ligne envoyée par le serveur, alors une ligne de données partielle sera renvoyée. Avec des données textuelles, ceci peut être détecté en testant si le dernier octet renvoyé est \n ou non (dans un COPY binaire, l'analyse réelle du format de données COPY sera nécessaire pour faire la détermination équivalente). La chaîne renvoyée n'est pas terminée par un octet nul. (Si vous voulez ajouter un octet nul de terminaison, assurez-vous de passer un *bufsize* inférieur de 1 par rapport à l'espace réellement disponible).

PQputline

Envoie une chaîne terminée par un octet nul au serveur. Renvoie 0 si tout va bien et EOF s'il est incapable d'envoyer la chaîne.

```
int PQputline(PGconn *conn,
                      const char *string);
```

Le flux de données de COPY envoyé par une série d'appels à PQputline a le même format que celui renvoyé par PQgetlineAsync, sauf que les applications ne sont pas obligées d'envoyer exactement une ligne de données par appel à PQputline ; il est correct d'envoyer une ligne partielle ou plusieurs lignes par appel.

> **Note**
>
> Avant le protocole 3.0 de PostgreSQL, il était nécessaire pour l'application d'envoyer explicitement les deux caractères \. comme ligne finale pour indiquer au serveur qu'elle a terminé l'envoi des données du COPY. Bien que ceci fonctionne toujours, cette méthode est obsolète et la signification spéciale de \. pourrait être supprimée dans une prochaine version. Il est suffisant d'appeler PQendcopy après avoir envoyé les vraies données.

PQputnbytes

Envoie une chaîne non terminée par un octet nul au serveur. Renvoie 0 si tout va bien et EOF s'il n'a pas été capable d'envoyer la chaîne.

```
int PQputnbytes(PGconn *conn,
                        const char *buffer,
                        int nbytes);
```

C'est exactement comme PQputline, sauf que le tampon de données n'a pas besoin d'être terminé avec un octet nul car le nombre d'octets envoyés est spécifié directement. Utilisez cette procédure pour envoyer des données binaires.

PQendcopy

Se synchronise avec le serveur.

```
int PQendcopy(PGconn *conn);
```

Cette fonction attend que le serveur ait terminé la copie. Elle devrait indiquer soit le moment où la dernière chaîne a été envoyée au serveur en utilisant `PQputline`, soit le moment où la dernière chaîne a été reçue du serveur en utilisant `PQgetline`. Si ce n'est pas fait, le serveur renverra un « out of sync » (perte de synchronisation) au client. Au retour de cette fonction, le serveur est prêt à recevoir la prochaine commande SQL. Le code de retour 0 indique un succès complet et est différent de zéro dans le cas contraire (utilisez `PQerrorMessage` pour récupérer des détails sur l'échec).

Lors de l'utilisation de `PQgetResult`, l'application devrait répondre à un résultat `PGRES_COPY_OUT` en exécutant `PQgetline` de façon répétée, suivi par un `PQendcopy` une fois la ligne de terminaison aperçue. Il devrait ensuite retourner à la boucle `PQgetResult` jusqu'à ce que `PQgetResult` renvoie un pointeur NULL. De façon similaire, un résultat `PGRES_COPY_IN` est traité par une série d'appels à `PQputline` suivis par un `PQendcopy`, ensuite retour à la boucle `PQgetResult`. Cet arrangement vous assurera qu'une commande `COPY` intégrée dans une série de commandes SQL sera exécutée correctement.

Les anciennes applications sont susceptibles de soumettre un `COPY` via `PQexec` et supposent que la transaction est faite après un `PQendcopy`. Ceci fonctionnera correctement seulement si `COPY` est la seule commande SQL dans la chaîne de commandes.

34.11. Fonctions de contrôle

Ces fonctions contrôlent divers détails du comportement de libpq.

`PQclientEncoding`

Renvoie l'encodage client.

```
int PQclientEncoding(const PGconn *conn);
```

Notez qu'il renvoie l'ID de l'encodage, pas une chaîne symbolique telle que `EUC_JP`. Renvoie -1 en cas d'échec. Pour convertir un ID d'encodage en nom, vous pouvez utiliser :

```
char *pg_encoding_to_char(int encoding_id);
```

`PQsetClientEncoding`

Configure l'encodage client.

```
int PQsetClientEncoding(PGconn *conn, const char
*encoding);
```

`conn` est la connexion au serveur, et *`encoding`* est l'encodage que vous voulez utiliser. Si la fonction initialise l'encodage avec succès, elle renvoie 0, sinon -1. L'encodage en cours pour cette connexion peut être déterminé en utilisant `PQclientEncoding` .

`PQsetErrorVerbosity`

Détermine la verbosité des messages renvoyés par `PQerrorMessage` et `PQresultErrorMessage`.

```
typedef enum
       {
           PQERRORS_TERSE,
           PQERRORS_DEFAULT,
           PQERRORS_VERBOSE,
           PQERRORS_SQLSTATE
       } PGVerbosity;

       PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity
 verbosity);
```

`PQsetErrorVerbosity` initialise le mode de verbosité, renvoyant le paramétrage précédant de cette connexion. Dans le mode *TERSE*, les messages renvoyés incluent seulement la sévérité, le texte principal et la position ; ceci tiendra normalement sur une seule ligne. Le mode *DEFAULT* produit des messages qui incluent ces champs ainsi que les champs détail, astuce ou contexte (ils pourraient être sur plusieurs lignes). Le mode *VERBOSE* inclut tous les champs disponibles. Le mode *SQLSTATE* inclut seulement la sévérité de l'erreur et le code d'erreur `SQLSTATE`, s'il est disponible (dans le cas contraire, la sortie est identique au mode *TERSE*).

Modifier la verbosité n'affecte pas les messages disponibles à partir d'objets `PGresult` déjà existants, seulement ceux créés après. (Mais voyez `PQresultVerboseErrorMessage` si vous voulez afficher une erreur précédente avec une verbosité différente).

`PQsetErrorContextVisibility`

Détermine la gestion des champs `CONTEXT` dans les messages renvoyées par `PQerrorMessage` et `PQresultErrorMessage`.

```
typedef enum
{
    PQSHOW_CONTEXT_NEVER,
    PQSHOW_CONTEXT_ERRORS,
    PQSHOW_CONTEXT_ALWAYS
} PGContextVisibility;

PGContextVisibility PQsetErrorContextVisibility(PGconn *conn,
 PGContextVisibility show_context);
```

`PQsetErrorContextVisibility` configure le mode d'affichage du contexte, renvoyant la précédente configuration de la connexion. Ce mode contrôle si le champ `CONTEXT` est inclus dans les messages. Le mode *NEVER* n'inclut jamais `CONTEXT`, alors que *ALWAYS* l'inclut en permanence s'il est disponible. Dans le mode par défaut, *ERRORS*, les champs `CONTEXT` sont inclus seulement pour les messages d'erreur, et non pas pour les messages d'informations et d'avertissements. (Cependant, si le paramètre verbosity est *TERSE* ou *SQLSTATE*, les champs `CONTEXT` sont omis quelque soit le contexte de mode d'affichage.)

La modification de ce mode n'affecte pas les messages disponibles à partir des objets `PGresult` déjà existants, seulement ceux créés après. (Cependant, voyez `PQresultVerboseErrorMessage` si vous voulez afficher une erreur précédente avec un mode d'affichage différent.)

`PQtrace`

Active la trace de la communication entre client et serveur vers un fichier de débogage.

```
void PQtrace(PGconn *conn, FILE *stream);
```

Chaque ligne consiste en : un horodatage optionnel, un indicateur de direction(`F` pour les messages du client au serveur ou `B` pour les messages du serveur au client), la longueur du message, le type du message et le contenu du message. Les champs autre que le contenu du message (horodatage, direction, longueur et type) sont séparés par une tabulation. Le contenu du message est séparé par un espace. Les chaînes du protocole sont entourées de guillemets doubles alors que les chaînes utilisées comme valeurs de données sont entourées de guillemets simples. Les caractères non affichables sont affichés sous la forme d'échappement hexadécimaux. Plus d'informations spécifique au type de message sont disponibles dans Section 55.7.

> **Note**
>
> Sur Windows, si la bibliothèque libpq et une application sont compilées avec des options différentes, cet appel de fonction fera planter l'application car la représentation interne des pointeurs `FILE` diffère. Spécifiquement, les options multi-threaded/single-threaded, release/debug et static/dynamic devraient être identiques pour la bibliothèque et les applications qui l'utilisent.

`PQsetTraceFlags`

Contrôle le comportement des traces sur la communication client/serveur.

```
void PQsetTraceFlags(PGconn *conn, int flags);
```

`flags` contient des bits décrivant le mode opérationnel des traces. Si `flags` contient `PQTRACE_SUPPRESS_TIMESTAMPS`, alors l'horodatage n'est pas inclus lors de l'affichage de chaque message. Si `flags` contient `PQTRACE_REGRESS_MODE`, alors certains champs sont modifiés lors de la sortie de chaque message, comme par exemple l'OID de l'objet, pour rendre la sortie plus agréable à utiliser dans des environnements de tests. Cette fonction doit être appelée après l'appel à `PQtrace`.

`PQuntrace`

Désactive les traces commencées avec `PQtrace`.

```
void PQuntrace(PGconn *conn);
```

34.12. Fonctions diverses

Comme toujours, certaines fonctions ne sont pas catégorisables.

`PQfreemem`

Libère la mémoire allouée par libpq.

```
void PQfreemem(void *ptr);
```

Libère la mémoire allouée par libpq, particulièrement `PQescapeByteaConn`, `PQescapeBytea`, `PQunescapeBytea`, et `PQnotifies`. Il est particulièrement important que cette fonction, plutôt que `free()`, soit utilisée sur Microsoft Windows. Ceci est dû au fait qu'allouer de la mémoire dans une DLL et la relâcher dans l'application ne marche que si les

drapeaux multi-thread/mon-thread, release/debug et static/dynamic sont les mêmes pour la DLL et l'application. Sur les plateformes autres que Microsoft Windows, cette fonction est identique à la fonction `free()` de la bibliothèque standard.

`PQconninfoFree`

Libère les structures de données allouées par `PQconndefaults` ou `PQconninfoParse`.

```
void PQconninfoFree(PQconninfoOption *connOptions);
```

Un simple appel à `PQfreemem` ne suffira pas car le tableau contient des références à des chaînes complémentaires.

`PQencryptPasswordConn`

Prépare la forme chiffrée du mot de passe PostgreSQL.

```
char *PQencryptPasswordConn(PGconn *conn, const char
*passwd, const char *user, const char *algorithm);
```

Cette fonction est utilisée par les applications clientes qui souhaitent envoyer des commandes comme `ALTER USER joe PASSWORD 'passe'`. Une bonne pratique est de ne pas envoyer le mot de passe en clair dans une telle commande car le mot de passe serait exposé dans les journaux, les affichages d'activité et ainsi de suite. À la place, utilisez cette fonction pour convertir le mot de passe sous forme chiffrée avant de l'envoyer.

Les arguments *`passwd`* et *`user`* sont le mot de passe en clair et le nom SQL de l'utilisateur correspondant. *`algorithm`* spécifie l'algorithme de chiffrement à utiliser pour chiffrer le mot de passe. Pour le moment, les algorithmes supportés sont `md5` et `scram-sha-256` (`on` et `off` sont également acceptés comme des alias pour `md5`, pour compatibilité avec les versions des anciens serveurs). Veuillez noter que le support de `scram-sha-256` a été introduit dans la version 10 de PostgreSQL, et ne fonctionnera pas correctement avec des versions de serveur plus ancienne. Si *`algorithm`* est `NULL`, cette fonction demandera au serveur la valeur actuelle du réglage password_encryption. Cela peut être bloquant, et échouera si la transaction courante est annulée ou si la connexion est occupée à effectuer une autre requête. Si vous souhaitez utiliser l'algorithme par défaut du serveur mais que vous voulez éviter un blocage, vérifiez vous-même `password_encryption` avant d'appeler `PQencryptPasswordConn`, et fournissez cette valeur pour *`algorithm`*.

La valeur retournée est une chaîne allouée par `malloc`. L'appelant peut partir du principe que la chaîne ne contient pas de caractères spéciaux qui nécessiteraient un échappement. Utilisez `PQfreemem` pour libérer le résultat quand vous avez fini de l'utiliser. En cas d'erreur, `NULL` est retourné, et un message d'erreur adéquat est stocké dans l'objet de connexion.

`PQencryptPassword`

Prépare la version chiffrée en md5 du mot de passe PostgreSQL.

```
char *PQencryptPassword(const char *passwd, const char *user);
```

`PQencryptPassword` est une version ancienne et obsolète de `PQencryptPasswordConn`. La différence est que `PQencryptPassword` ne nécessite pas d'objet de connexion, et que l'algorithme de chiffrement utilisé est toujours `md5`.

PQmakeEmptyPGresult

Construit un objet PGresult vide avec le statut indiqué.

```
PGresult *PQmakeEmptyPGresult(PGconn *conn,
ExecStatusType status);
```

C'est une fonction interne de la libpq pour allouer et initialiser un objet PGresult vide. Cette fonction renvoie NULL si la mémoire n'a pas pu être allouée. Elle est exportée car certaines applications trouveront utiles de générer elles-mêmes des objets de résultat (tout particulièrement ceux avec des statuts d'erreur). Si *conn* n'est pas NULL et que *status* indique une erreur, le message d'erreur courant de la connexion indiquée est copié dans PGresult. De plus, si *conn* n'est pas NULL, toute procédure d'événement enregistrée dans la connexion est copiée dans le PGresult. (Elles n'obtiennent pas d'appels PGEVT_RESULTCREATE, mais jetez un œil à PQfireResultCreateEvents.) Notez que PQclear devra être appelé sur l'objet, comme pour un PGresult renvoyé par libpq lui-même.

PQfireResultCreateEvents

Déclenche un événement PGEVT_RESULTCREATE (voir Section 34.14) pour chaque procédure d'événement enregistrée dans l'objet PGresult. Renvoie autre chose que zéro en cas de succès, zéro si une des procédures d'événement échoue.

```
int PQfireResultCreateEvents(PGconn *conn, PGresult
*res);
```

L'argument conn est passé aux procédures d'événement mais n'est pas utilisé directement. Il peut être NULL si les procédures de l'événement ne l'utilisent pas.

Les procédures d'événements qui ont déjà reçu un événement PGEVT_RESULTCREATE ou PGEVT_RESULTCOPY pour cet objet ne sont pas déclenchées de nouveau.

La raison principale pour séparer cette fonction de PQmakeEmptyPGresult est qu'il est souvent approprié de créer un PGresult et de le remplir avec des données avant d'appeler les procédures d'événement.

PQcopyResult

Fait une copie de l'objet PGresult. La copie n'est liée en aucune façon au résultat source et PQclear doit être appelée quand la copie n'est plus nécessaire. Si la fonction échoue, NULL est renvoyé.

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

Cela n'a pas pour but de faire une copie exacte. Le résultat renvoyé a toujours le statut PGRES_TUPLES_OK, et ne copie aucun message d'erreur de la source. (Néanmoins, elle copie la chaîne de statut de commande.) L'argument *flags* détermine ce qui est copié. C'est un OR bit à bit de plusieurs drapeaux. PG_COPYRES_ATTRS indique la copie des attributs du résultat source (définition des colonnes). PG_COPYRES_TUPLES indique la copie des lignes du résultat source. (Cela implique de copier aussi les attributs.) PG_COPYRES_NOTICEHOOKS indique la copie des gestionnaires de notification du résultat source. PG_COPYRES_EVENTS indique la copie des événements du résultat source. (Mais toute instance de données associée avec la source n'est pas copiée.) Les procédures événement reçoivent les événements PGEVT_RESULTCOPY.

`PQsetResultAttrs`

Initialise les attributs d'un objet `PGresult`.

```
int PQsetResultAttrs(PGresult *res, int numAttributes,
PGresAttDesc *attDescs);
```

Les *`attDescs`* fournis sont copiés dans le résultat. Si le pointeur *`attDescs`* est `NULL` ou si *`numAttributes`* est inférieur à 1, la requête est ignorée et la fonction réussit. Si *`res`* contient déjà les attributs, la fonction échouera. Si la fonction échoue, la valeur de retour est zéro. Si la fonction réussit, la valeur de retour est différente de zéro.

`PQsetvalue`

Initialise la valeur d'un champ d'une ligne d'un objet `PGresult`.

```
int PQsetvalue(PGresult *res, int tup_num, int field_num,
char *value, int len);
```

La fonction fera automatiquement grossir le tableau de lignes internes des résultats, si nécessaire. Néanmoins, l'argument *`tup_num`* doit être inférieur ou égal à `PQntuples`, ceci signifiant que la fonction peut seulement faire grossir le tableau des lignes une ligne à la fois. Mais tout champ d'une ligne existante peut être modifié dans n'importe quel ordre. Si une valeur à *`field_num`* existe déjà, elle sera écrasée. Si *`len`* vaut 1 ou si *`value`* est `NULL`, la valeur du champ sera configurée à la valeur SQL `NULL`. *`value`* est copié dans le stockage privé du résultat, donc n'est plus nécessaire après le retour de la fonction. Si la fonction échoue, la valeur de retour est zéro. Dans le cas contraire, elle a une valeur différente de zéro.

`PQresultAlloc`

Alloue un stockage supplémentaire pour un objet `PGresult`.

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

Toute mémoire allouée avec cette fonction sera libérée quand *`res`* sera effacé. Si la fonction échoue, la valeur de retour vaut `NULL`. Le résultat est garanti d'être correctement aligné pour tout type de données, comme pour un `malloc`.

`PQresultMemorySize`

Renvoie le nombre d'octets alloués pour un objet `PGresult`.

```
size_t PQresultMemorySize(const PGresult *res);
```

La valeur est la somme de tous les appels `malloc` associés avec l'objet `PGresult`, c'est-à-dire tout l'espace qui peut être libéré par `PQclear`. Cette information est utile pour gérer la consommation mémoire.

`PQlibVersion`

Renvoie la version de libpq en cours d'utilisation.

```
int PQlibVersion(void);
```

Le résultat de cette fonction peut être utilisé pour déterminer, à l'exécution, si certaines fonctionnalités spécifiques sont disponibles dans la version chargée de libpq. Par exemple, cette fonction peut être utilisée pour déterminer les options de connexions disponibles pour `PQconnectdb`.

Le résultat est obtenu en multipliant le numéro de version majeure de la bibliothèque par 10000 et en ajoutant le numéro de version mineure. Par exemple, la version 10.1 renverra 100001, et la version 11.0 renverra 110000.

Avant la version majeure 10, PostgreSQL utilisait des numéros de version en trois parties, pour lesquelles les deux premières parties représentaient la version majeure. Pour ces versions, `PQlibVersion` utilise deux chiffres pour chaque partie. Par exemple, la version 9.1.5 renverra 90105, et la version 9.2.0 renverra 90200.

De ce fait, pour déterminer la compatibilité de certaines fonctionnalités, les applications devraient diviser le résultat de `PQlibVersion` par 100, et non pas par 10000, pour déterminer le numéro de version majeure logique. Dans toutes les versions, seuls les deux derniers chiffres diffèrent entre des versions mineures (versions correctives).

> **Note**
>
> Cette fonction apparaît dans PostgreSQL 9.1, donc elle ne peut pas être utilisée pour détecter des fonctionnalités des versions précédentes car l'appeler créera une dépendance sur la version 9.1 et les versions ultérieures.

34.13. Traitement des messages

Les messages de note et d'avertissement générés par le serveur ne sont pas renvoyés par les fonctions d'exécution des requêtes car elles n'impliquent pas d'échec dans la requête. À la place, elles sont passées à la fonction de gestion des messages et l'exécution continue normalement après le retour du gestionnaire. La fonction par défaut de gestion des messages affiche le message sur `stderr` mais l'application peut surcharger ce comportement en proposant sa propre fonction de gestion.

Pour des raisons historiques, il existe deux niveaux de gestion de messages, appelés la réception des messages et le traitement. Pour la réception, le comportement par défaut est de formater le message et de passer une chaîne au traitement pour affichage. Néanmoins, une application qui choisit de fournir son propre receveur de messages ignorera typiquement la couche d'envoi de messages et effectuera tout le travail au niveau du receveur.

La fonction `PQsetNoticeReceiver` initialise ou examine le receveur actuel de messages pour un objet de connexion. De la même façon, `PQsetNoticeProcessor` initialise ou examine l'émetteur actuel de messages.

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

    PQnoticeReceiver
    PQsetNoticeReceiver(PGconn *conn,
                        PQnoticeReceiver proc,
                        void *arg);

    typedef void (*PQnoticeProcessor) (void *arg, const char
 *message);

    PQnoticeProcessor
```

```
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

Chacune de ces fonctions reçoit le pointeur de fonction du précédent receveur ou émetteur de messages et configure la nouvelle valeur. Si vous fournissez un pointeur de fonction NULL, aucune action n'est réalisée mais le pointeur actuel est renvoyé.

Quand un message de note ou d'avertissement est reçu du serveur ou généré de façon interne par libpq, la fonction de réception du message est appelée. Le message lui est passé sous la forme d'un `PGresult` `PGRES_NONFATAL_ERROR` (ceci permet au receveur d'extraire les champs individuels en utilisant `PQresultErrorField` ou d'obtenir le message complet préformaté en utilisant `PQresultErrorMessage` ou `PQresultVerboseErrorMessage`). Le même pointeur void passé à `PQsetNoticeReceiver` est aussi renvoyé (ce pointeur peut être utilisé pour accéder à un état spécifique de l'application si nécessaire).

Le receveur de messages par défaut extrait simplement le message (en utilisant `PQresultErrorMessage`) et le passe au système de traitement du message.

Ce dernier est responsable de la gestion du message de note ou d'avertissement, fourni au format texte. La chaîne texte du message est passée (avec un retour chariot final), plus un pointeur sur void identique à celui passé à `PQsetNoticeProcessor` (ce pointeur est utilisé pour accéder à un état spécifique de l'application si nécessaire).

Le traitement des messages par défaut est simplement :

```
static void
    defaultNoticeProcessor(void * arg, const char * message)
    {
        fprintf(stderr, "%s", message);
    }
```

Une fois que vous avez initialisé un receveur ou une fonction de traitement des messages, vous devez vous attendre à ce que la fonction soit appelée aussi longtemps que l'objet `PGconn` ou qu'un objet `PGresult` réalisé à partir de celle-ci existent. À la création d'un `PGresult`, les pointeurs de gestion courants de `PGconn` sont copiés dans `PGresult` pour une utilisation possible par des fonctions comme `PQgetvalue`.

34.14. Système d'événements

Le système d'événements de libpq est conçu pour notifier les gestionnaires d'événements enregistrés de l'arrivée d'événements intéressants de la libpq, comme par exemple la création ou la destruction d'objets `PGconn` et `PGresult`. Un cas d'utilisation principal est de permettre aux applications d'associer leur propres données avec un `PGconn` ou un `PGresult` et de s'assurer que les données soient libérées au bon moment.

Chaque gestionnaire d'événement enregistré est associé avec deux types de données, connus par libpq comme des pointeurs opaques, c'est-à-dire `void *`. Il existe un pointeur *passthrough* fourni par l'application quand le gestionnaire d'événements est enregistré avec un `PGconn`. Le pointeur passthrough ne change jamais pendant toute la durée du `PGconn` et des `PGresult` générés grâce à lui ; donc s'il est utilisé, il doit pointer vers des données à longue vie. De plus, il existe un pointeur de *données instanciées*, qui commence à NULL dans chaque objet `PGconn` et `PGresult`. Ce pointeur peut être manipulé en utilisant les fonctions `PQinstanceData`, `PQsetInstanceData`, `PQresultInstanceData` et `PQresultSetInstanceData`. Notez que, contrairement au pointeur passthrough, les `PGresult` n'héritent pas automatiquement des données instanciées d'un `PGconn`. libpq ne sait pas vers quoi pointent les pointeurs passthrough et de données instanciées, et n'essaiera hamais de les libérer -- cela tient de la responsabilité du gestionnaire d'événements.

34.14.1. Types d'événements

L'enum `PGEventId` précise tous les types d'événements gérés par le système d'événements. Toutes ses valeurs ont des noms commençant avec `PGEVT`. Pour chaque type d'événement, il existe une structure d'informations sur l'événement, précisant les paramètres passés aux gestionnaires d'événement. Les types d'événements sont :

`PGEVT_REGISTER`

L'événement d'enregistrement survient quand `PQregisterEventProc` est appelé. C'est le moment idéal pour initialiser toute structure `instanceData` qu'une procédure d'événement pourrait avoir besoin. Seul un événement d'enregistrement sera déclenché par gestionnaire d'événement sur une connexion. Si la procédure échoue (renvoie zéro), l'enregistrement est annulé.

```
typedef struct
{
    PGconn *conn;
} PGEventRegister;
```

Quand un événement `PGEVT_REGISTER` est reçu, le pointeur *`evtInfo`* doit être converti en un `PGEventRegister *`. Cette structure contient un `PGconn` qui doit être dans le statut `CONNECTION_OK` ; garanti si `PQregisterEventProc` est appelé juste après avoir obtenu un bon `PGconn`. Lorsqu'elle renvoie un code d'erreur, le nettoyage doit être réalisé car aucun événement `PGEVT_CONNDESTROY` ne sera envoyé.

`PGEVT_CONNRESET`

L'événement de réinitialisation de connexion est déclenché après un `PQreset` ou un `PQresetPoll`. Dans les deux cas, l'événement est seulement déclenché si la ré-initialisation est réussie. La valeur de retour de la procédure événement est ignoré dans les versions 15 et ultérieures de PostgreSQL. Avec les versions précédentes, il est important de renvoyer le succès (différent de zéro), sinon la connexion sera annulée.

```
typedef struct
{
    PGconn *conn;
} PGEventConnReset;
```

Quand un événement `PGEVT_CONNRESET` est reçu, le pointeur *`evtInfo`* doit être converti en un `PGEventConnReset *`. Bien que le `PGconn` a été réinitialisé, toutes les données de l'événement restent inchangées. Cet événement doit être utilisé pour ré-initialiser/recharger/re-requêter tout `instanceData` associé. Notez que même si la procédure d'événement échoue à traiter `PGEVT_CONNRESET`, elle recevra toujours un événement `PGEVT_CONNDESTROY` à la fermeture de la connexion.

`PGEVT_CONNDESTROY`

L'événement de destruction de la connexion est déclenché en réponse à `PQfinish`. Il est de la responsabilité de la procédure de l'événement de nettoyer proprement ses données car libpq n'a pas les moyens de gérer cette mémoire. Un échec du nettoyage amènera des fuites de mémoire.

```
typedef struct
{
```

```
    PGconn *conn;
} PGEventConnDestroy;
```

Quand un événement PGEVT_CONNDESTROY est reçu, le pointeur *evtInfo* doit être converti en un PGEventConnDestroy *. Cet événement est déclenché avant que PQfinish ne réalise d'autres nettoyages. La valeur de retour de la procédure est ignorée car il n'y a aucun moyen d'indiquer un échec de PQfinish. De plus, un échec de la procédure ne doit pas annuler le nettoyage de la mémoire non désirée.

PGEVT_RESULTCREATE

L'événement de création de résultat est déclenché en réponse à l'utilisation d'une fonction d'exécution d'une requête, par exemple PQgetResult. Cet événement sera déclenché seulement après la création réussie du résultat.

```
typedef struct
{
    PGconn *conn;
    PGresult *result;
} PGEventResultCreate;
```

Quand un événement PGEVT_RESULTCREATE est reçu, le pointeur *evtInfo* doit être converti en un PGEventResultCreate *. Le paramètre *conn* est la connexion utilisée pour générer le résultat. C'est le moment idéal pour initialiser tout instanceData qui doit être associé avec le résultat. Si une procédure d'événement échoue (renvoie zéro), cet événement de procédure sera ignoré pour la durée de vie restante du résultat ; autrement dit, il ne recevra pas les événements PGEVT_RESULTCOPY ou PGEVT_RESULTDESTROY pour ce résultat ou les résultats copiés à partir de lui.

PGEVT_RESULTCOPY

L'événement de copie du résultat est déclenché en réponse à un PQcopyResult. Cet événement se déclenchera seulement une fois la copie terminée. Seules les procédures qui ont géré avec succès l'événement PGEVT_RESULTCREATE ou PGEVT_RESULTCOPY pour le résultat source recevront les événements PGEVT_RESULTCOPY.

```
typedef struct
{
    const PGresult *src;
    PGresult *dest;
} PGEventResultCopy;
```

Quand un événement PGEVT_RESULTCOPY est reçu, le pointeur *evtInfo* doit être converti en un PGEventResultCopy *. Le résultat *src* correspond à ce qui a été copié alors que le résultat *dest* correspond à la destination. Cet événement peut être utilisé pour fournir une copie complète de instanceData, ce que PQcopyResult ne peut pas faire. Si une procédure d'événement échoue (renvoie zéro), cet événement de procédure sera ignoré pour la durée de vie restante du résultat ; autrement dit, il ne recevra pas les événements PGEVT_RESULTCOPY ou PGEVT_RESULTDESTROY pour ce résultat ou les résultats copiés à partir de lui.

PGEVT_RESULTDESTROY

L'événement de destruction de résultat est déclenché en réponse à la fonction PQclear. Il est de la responsabilité de l'événement de nettoyer proprement les données de l'événement car libpq

n'a pas la capacité de gérer cette mémoire. Si le nettoyage échoue, cela sera la cause de pertes mémoire.

```
typedef struct
{
    PGresult *result;
} PGEventResultDestroy;
```

Quand un événement `PGEVT_RESULTDESTROY` est reçu, le pointeur *`evtInfo`* doit être converti en un `PGEventResultDestroy *`. Cet événement est déclenché avant que `PQclear` ne puisse faire de nettoyage. La valeur de retour de la procédure est ignorée car il n'existe aucun moyen d'indiquer un échec à partir de `PQclear`. De plus, un échec de la procédure ne doit pas annuler le nettoyage de la mémoire non désirée.

34.14.2. Procédure de rappel de l'événement

`PGEventProc`

`PGEventProc` est une définition de type pour un pointeur vers une procédure d'événement, c'est-à-dire la fonction utilisateur appelée pour les événements de la libpq. La signature d'une telle fonction doit être :

```
int eventproc(PGEventId evtId, void *evtInfo, void
*passThrough)
```

Le paramètre *`evtId`* indique l'événement `PGEVT` qui est survenu. Le pointeur *`evtInfo`* doit être converti vers le type de structure approprié pour obtenir plus d'informations sur l'événement. Le paramètre *`passThrough`* est le pointeur fourni à `PQregisterEventProc` quand la procédure de l'événement a été enregistrée. La fonction doit renvoyer une valeur différente de zéro en cas de succès et zéro en cas d'échec.

Une procédure d'événement particulière peut être enregistrée une fois seulement pour un `PGconn`. Ceci est dû au fait que l'adresse de la procédure est utilisée comme clé de recherche pour identifier les données instanciées associées.

> **Attention**
>
> Sur Windows, les fonctions peuvent avoir deux adresses différentes : une visible de l'extérieur de la DLL et une visible de l'intérieur. Il faut faire attention que seule une de ces adresses soit utilisée avec les fonctions d'événement de la libpq, sinon une confusion en résultera. La règle la plus simple pour écrire du code fonctionnel est de s'assurer que les procédures d'événements sont déclarées `static`. Si l'adresse de la procédure doit être disponible en dehors de son propre fichier source, il faut exposer une fonction séparée pour renvoyer l'adresse.

34.14.3. Fonctions de support des événements

`PQregisterEventProc`

Enregistre une procédure de rappel pour les événements avec libpq.

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
                        const char *name, void
*passThrough);
```

Une procédure d'événement doit être enregistrée une fois pour chaque PGconn pour lequel vous souhaitez recevoir des événements. Il n'existe pas de limite, autre que la mémoire, sur le nombre de procédures d'événements qui peuvent être enregistrées avec une connexion. La fonction renvoie une valeur différente de zéro en cas de succès, et zéro en cas d'échec.

L'argument *proc* sera appelé quand se déclenchera un événement libpq. Son adresse mémoire est aussi utilisée pour rechercher instanceData. L'argument *name* est utilisé pour faire référence à la procédure d'événement dans les messages d'erreur. Cette valeur ne peut pas être NULL ou une chaîne de longueur nulle. La chaîne *name* est copiée dans PGconn, donc ce qui est passé n'a pas besoin d'exister longtemps. Le pointeur *passThrough* est passé à *proc* à chaque arrivée d'un événement. Cet argument peut être NULL.

PQsetInstanceData

Initialise avec *data* l'instanceData de la connexion *conn* pour la procédure *proc*. Cette fonction renvoie zéro en cas d'échec et autre chose en cas de réussite. (L'échec est seulement possible si *proc* n'a pas été correctement enregistré dans *conn*.)

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc,
void *data);
```

PQinstanceData

Renvoie l'instanceData de la connexion *conn* associée au *proc* ou NULL s'il n'y en a pas.

```
void *PQinstanceData(const PGconn *conn, PGEventProc
proc);
```

PQresultSetInstanceData

Initialise avec *data* l'instanceData du résultat pour la procédure *proc*. Cette fonction renvoie zéro en cas d'échec et autre chose en cas de réussite. (L'échec est seulement possible si *proc* n'a pas été correctement enregistré dans le résultat.)

```
int PQresultSetInstanceData(PGresult *res, PGEventProc
proc, void *data);
```

Attention, aucun stockage représenté par *data* ne sera pris en compte par PQresultMemorySize, à moins qu'il est alloué en utilisant PQresultAlloc. (Faire ceci est recommandé car cela élimine le besoin de libérer le stockage explicitement quand le résultat est supprimé.)

PQresultInstanceData

Renvoie l'instanceData du résultat associé à *proc* ou NULL s'il n'y en a pas.

```
void *PQresultInstanceData(const PGresult *res,
PGEventProc proc);
```

34.14.4. Exemple d'un événement

Voici un exemple d'une gestion de données privées associée aux connexions et aux résultats de la libpq.

```
/* en-tête nécessaire pour les événements de la libpq (note :
 inclut libpq-fe.h) */
#include <libpq-events.h>

/* la donnée instanciée : instanceData */
typedef struct
{
    int n;
    char *str;
} mydata;

/* PGEventProc */
static int myEventProc(PGEventId evtId, void *evtInfo, void
 *passThrough);

int
main(void)
{
    mydata *data;
    PGresult *res;
    PGconn *conn =
        PQconnectdb("dbname=postgres options=-csearch_path=");

    if (PQstatus(conn) != CONNECTION_OK)
    {
        /* PQerrorMessage's result includes a trailing newline */
        fprintf(stderr, "%s", PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }

    /* appelée une fois pour toute connexion qui doit recevoir des
 événements.
     * Envoit un PGEVT_REGISTER à myEventProc.
     */
    if (!PQregisterEventProc(conn, myEventProc, "mydata_proc",
 NULL))
    {
        fprintf(stderr, "Cannot register PGEventProc\n");
        PQfinish(conn);
        return 1;
    }

    /* la connexion instanceData est disponible */
    data = PQinstanceData(conn, myEventProc);

    /* Envoit un PGEVT_RESULTCREATE à myEventProc */
    res = PQexec(conn, "SELECT 1 + 1");

    /* le résultat instanceData est disponible */
    data = PQresultInstanceData(res, myEventProc);
```

```
    /* Si PG_COPYRES_EVENTS est utilisé, envoit un PGEVT_RESULTCOPY
 à myEventProc */
    res_copy = PQcopyResult(res, PG_COPYRES_TUPLES |
 PG_COPYRES_EVENTS);

    /* le résultat instanceData est disponible si PG_COPYRES_EVENTS
 a été
     * utilisé lors de l'appel à PQcopyResult.
     */
    data = PQresultInstanceData(res_copy, myEventProc);

    /* Les deux fonctions de nettoyage envoient PGEVT_RESULTDESTROY
 à myEventProc */
    PQclear(res);
    PQclear(res_copy);

    /* Envoit un PGEVT_CONNDESTROY à myEventProc */
    PQfinish(conn);

    return 0;
}

static int
myEventProc(PGEventId evtId, void *evtInfo, void *passThrough)
{
    switch (evtId)
    {
        case PGEVT_REGISTER:
        {
            PGEventRegister *e = (PGEventRegister *)evtInfo;
            mydata *data = get_mydata(e->conn);

            /* associe des données spécifiques de l'application
 avec la connexion */
            PQsetInstanceData(e->conn, myEventProc, data);
            break;
        }

        case PGEVT_CONNRESET:
        {
            PGEventConnReset *e = (PGEventConnReset *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            if (data)
              memset(data, 0, sizeof(mydata));
            break;
        }

        case PGEVT_CONNDESTROY:
        {
            PGEventConnDestroy *e = (PGEventConnDestroy *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            /* libère les données instanciées car la connexion est
 en cours de destruction */
            if (data)
              free_mydata(data);
            break;
```

```
        }

        case PGEVT_RESULTCREATE:
        {
            PGEventResultCreate *e = (PGEventResultCreate
*)evtInfo;
            mydata *conn_data = PQinstanceData(e->conn,
myEventProc);
            mydata *res_data = dup_mydata(conn_data);

            /* associe des données spécifiques à l'application avec
les résultats (copié de la connexion) */
            PQresultSetInstanceData(e->result, myEventProc,
res_data);
            break;
        }

        case PGEVT_RESULTCOPY:
        {
            PGEventResultCopy *e = (PGEventResultCopy *)evtInfo;
            mydata *src_data = PQresultInstanceData(e->src,
myEventProc);
            mydata *dest_data = dup_mydata(src_data);

            /* associe des données spécifiques à l'application avec
les résultats (copié d'un résultat) */
            PQresultSetInstanceData(e->dest, myEventProc,
dest_data);
            break;
        }

        case PGEVT_RESULTDESTROY:
        {
            PGEventResultDestroy *e = (PGEventResultDestroy
*)evtInfo;
            mydata *data = PQresultInstanceData(e->result,
myEventProc);

            /* libère les données instanciées car le résultat est
en cours de destruction */
            if (data)
              free_mydata(data);
            break;
        }

        /* unknown event id, just return TRUE. */
        default:
            break;
    }

    return TRUE; /* event processing succeeded */
}
```

34.15. Variables d'environnement

Les variables d'environnement suivantes peuvent être utilisées pour sélectionner des valeurs par défaut pour les paramètres de connexion, valeurs qui seront utilisées par `PQconnectdb`, `PQsetdbLogin` et `PQsetdb` si aucune valeur n'est directement précisée par le code appelant. Elles sont utiles pour éviter de coder en dur les informations de connexion à la base de données dans les applications clients, par exemple.

- `PGHOST` se comporte de la même façon que le paramètre de configuration host.
- `PGHOSTADDR` se comporte de la même façon que le paramètre de configuration hostaddr. Elle peut être initialisée à la place ou en plus de `PGHOST` pour éviter la charge supplémentaire d'une résolution DNS.
- `PGPORT` se comporte de la même façon que le paramètre de configuration port.
- `PGDATABASE` se comporte de la même façon que le paramètre de configuration dbname.
- `PGUSER` se comporte de la même façon que le paramètre de configuration user.
- `PGPASSWORD` se comporte de la même façon que le paramètre de configuration password. L'utilisation de cette variable d'environnement n'est pas recommandée pour des raisons de sécurité, car certains systèmes d'exploitation autorisent les utilisateurs autres que root à voir les variables d'environnement du processus via ps) ; à la place, envisagez l'utilisation d'un fichier de mots de passe (voir la Section 34.16.
- `PGPASSFILE` se comporte de la même façon que le paramètre de connexion passfile.
- `PGCHANNELBINDING` se comporte de la même façon que le paramètre de connexion channel_binding.
- `PGSERVICE` se comporte de la même façon que le paramètre de configuration service.
- `PGSERVICEFILE` indique le nom du fichier service de connexion par utilisateur. Sa valeur par défaut est `~/.pg_service.conf`, sauf sur Microsoft Windows où elle est `%APPDATA%\postgresql\.pg_service.conf`.
- `PGOPTIONS` se comporte de la même façon que le paramètre de configuration options.
- `PGAPPNAME` se comporte de la même façon que le paramètre de connexion application_name.
- `PGSSLMODE` se comporte de la même façon que le paramètre de configuration sslmode.
- `PGREQUIRESSL` se comporte de la même façon que le paramètre de configuration requiressl. Cette variable d'environnement est obsolète, remplacée par la variable `PGSSLMODE` si les deux variables sont initialisées, `PGREQUIRESSL` est ignoré.
- `PGSSLCOMPRESSION` se comporte de la même façon que le paramètre de connexion sslcompression.
- `PGSSLCERT` se comporte de la même façon que le paramètre de configuration sslcert.
- `PGSSLKEY` se comporte de la même façon que le paramètre de configuration sslkey.
- `PGSSLROOTCERT` se comporte de la même façon que le paramètre de configuration sslrootcert.
- `PGSSLCRL` se comporte de la même façon que le paramètre de configuration sslcrl.
- `PGSSLCRLDIR` se comporte de la même façon que le paramètre de connexion sslcrldir.
- `PGSSLSNI` se comporte de la même façon que le paramètre de connexion sslsni.
- `PGREQUIREPEER` se comporte de la même façon que le paramètre de connexion requirepeer.
- `PGSSLMINPROTOCOLVERSION` se comporte de la même façon que le paramètre de connexion ssl_min_protocol_version.

- PGSSLMAXPROTOCOLVERSION se comporte de la même façon que le paramètre de connexion ssl_max_protocol_version.
- PGKRBSRVNAME se comporte de la même façon que le paramètre de configuration krbsrvname.
- PGGSSENCMODE se comporte de la même façon que le paramètre de connexion gssencmode.
- PGGSSLIB se comporte de la même façon que le paramètre de configuration gsslib.
- PGCONNECT_TIMEOUT se comporte de la même façon que le paramètre de configuration connect_timeout.
- PGCLIENTENCODING se comporte de la même façon que le paramètre de connexion client_encoding.
- PGTARGETSESSIONATTRS se comporte de la même façon que le paramètre de connexion target_session_attrs.

Les variables d'environnement peuvent être utilisées pour spécifier le comportement par défaut de chaque session PostgreSQL (voir aussi les commandes ALTER ROLE et ALTER DATABASE pour modifier le comportement par défaut par utilisateur ou par base de données).

- PGDATESTYLE initialise le style par défaut de la représentation de la date et de l'heure (équivalent à `SET datestyle TO ...`).
- PGTZ initialise le fuseau horaire par défaut (équivalent à `SET timezone TO ...`).
- PGGEQO initialise le mode par défaut pour l'optimiseur génétique de requêtes (équivalent à `SET geqo TO ...`).

Référez-vous à la commande SQL SET pour plus d'informations sur des valeurs correctes pour ces variables d'environnement.

Les variables d'environnement suivantes déterminent le comportement interne de libpq ; elles surchargent les valeurs par défaut issues de la compilation.

- PGSYSCONFDIR configure le répertoire contenant le fichier `pg_service.conf` et, peut-être dans une future version, d'autres fichiers de configuration globaux au système.
- PGLOCALEDIR configure le répertoire contenant les fichiers `locale` pour l'internationalisation des messages.

34.16. Fichier de mots de passe

Le fichier `.pgpass`, situé dans le répertoire personnel de l'utilisateur est un fichier contenant les mots de passe à utiliser si la connexion requiert un mot de passe (et si aucun mot de passe n'a été spécifié). Sur Microsoft Windows, le fichier est nommé `%APPDATA%\postgresql\pgpass.conf` (où %APPDATA% fait référence au sous-répertoire Application Data du profil de l'utilisateur). De manière alternative, le fichier de mots de passe peut être spécifié en utilisant le paramètre de connexion passfile ou la variable d'environnement PGPASSFILE.

Ce fichier devra être composé de lignes au format suivant (une ligne par connexion) :

nom_hote:*port*:*database*:*nomutilisateur*:*motdepasse*

(Vous pouvez ajouter un rappel en commentaire dans le fichier en copiant cette ligne et en la précédant d'un dièse (#).) Chacun des quatre premiers champs peut être une valeur littérale ou *, qui correspond à tout. La première ligne correspondant aux paramètres de connexion sera utilisée (du coup, placez les entrées les plus spécifiques en premier lorsque vous utilisez des jokers). Si une entrée doit contenir : ou \, échappez ce caractère avec \. Le nom de l'hôte est rapproché du paramètre de connexion `host` s'il est spécifié, sinon au paramètre `hostaddr` si spécifié. Si ni l'un ni l'autre ne sont fournis,

l'hôte `localhost` sera alors recherché. L'hôte `localhost` est également recherché si la connexion est une socket de domaine Unix et que le paramètre `host` correspond au répertoire par défaut de la libpq. Sur un serveur secondaire, un champ `database` à `replication` correspond aux connexions de réplication par flux au serveur maître. À part cela, le champ `database` est d'une utilité limitée, puisque les utilisateurs ont le même mot de passe pour toutes les bases de données de l'instance.

Sur les systèmes Unix, les droits sur un fichier de mots de passe doivent interdire l'accès au groupe et au reste du monde ; faites-le avec une commande comme `chmod 0600 ~/.pgpass`. Si les droits sont moins stricts, le fichier sera ignoré. Sur Microsoft Windows, il est supposé que le fichier est stocké dans un répertoire qui est sécurisé, donc aucune vérification des droits n'est effectuée.

34.17. Fichier des services de connexion

Le fichier des services de connexion permet d'associer des paramètres de connexion à un nom de service unique. Ce nom de service peut ensuite être spécifié par une chaîne de connexion libpq et les paramétrages associés seront utilisés. On peut donc modifier les paramètres de connexion sans avoir à recompiler l'application utilisant la libpq. Le nom de service peut aussi être spécifié en utilisant la variable d'environnement `PGSERVICE`.

Les noms de service peuvent être définis soit comme un fichier par utilisateur soit comme un fichier du système. Si le même nom de service existe dans les deux cas, le fichier utilisateur a priorité. Par défaut, le fichier service de l'utilisateur s'appelle `~/.pg_service.conf` ; ce nom et ce chemin peuvent être surchargés en configurant la variable d'environnement `PGSERVICEFILE`. Le fichier du système est nommé `pg_service.conf`. Par défaut, il est recherché dans le répertoire `etc` de l'installation PostgreSQL (utilisez `pg_config --sysconfdir` pour identifier précisément ce répertoire). Il est aussi possible de configurer uniquement le nom du répertoire en utilisant la variable d'environnement `PGSYSCONFDIR`.

Chaque fichier service utilise le format des « fichiers INI » où le nom de section et les paramètres sont des paramètres de connexion ; voir Section 34.1.2 pour une liste. Par exemple :

```
# comment
[mabase]
host=unhote
port=5433
user=admin
```

`share/pg_service.conf.sample` est fourni comme fichier d'exemple dans le répertoire d'installation de PostgreSQL.

Les paramètres de connexion obtenus à partir d'un fichier service sont combinés avec les paramètres obtenus autrement. La configuration réalisée par un fichier service surcharge les variables d'environnement correspondantes et peut elle-même être surchargée avec une valeur indiquée directement dans la chaîne de connexion. Par exemple, en utilisant le fichier service ci-dessus, une chaîne de connexion `service=mydb  port=5434` utilisera l'hôte `somehost`, le port `5434`, l'utilisateur `admin`, et les autres paramètres tels qu'ils sont configurés par les variables d'environnement ou leurs valeurs par défaut.

34.18. Recherche LDAP des paramètres de connexion

Si libpq a été compilé avec le support de LDAP (option `--with-ldap` du script `configure`), il est possible de récupérer les options de connexion comme `host` ou `dbname` via LDAP à partir d'un serveur central. L'avantage en est que, si les paramètres de connexion d'une base évolue, l'information de connexion n'a pas à être modifiée sur toutes les machines clientes.

La recherche LDAP des paramètres de connexion utilise le fichier service `pg_service.conf` (voir Section 34.17). Une ligne dans `pg_service.conf` commençant par `ldap://` sera reconnue comme une URL LDAP et une requête LDAP sera exécutée. Le résultat doit être une liste de paires `motclé = valeur` qui sera utilisée pour configurer les options de connexion. L'URL doit être conforme à la RFC 1959[4] et être de la forme :

```
   ldap://
[hôte[:port]]/base_recherche?attribut?étendue_recherche?filtre
```

où *`hôte`* vaut par défaut `localhost` et *`port`* vaut par défaut 389.

Le traitement de `pg_service.conf` se termine après une recherche LDAP réussie, mais continue si le serveur LDAP ne peut pas être contacté. Cela fournit un moyen de préciser d'autres URL LDAP pointant vers d'autres serveurs LDAP, des paires classiques `motclé = valeur` ou les options de connexion par défaut. Si dans ce cas vous préférez avoir un message d'erreur, ajoutez une ligne syntaxiquement incorrecte après l'URL LDAP.

À titre d'exemple, une entrée LDAP créée à partir du fichier LDIF suivant

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:device
cn:mydatabase
description:host=dbserver.mycompany.com
description:port=5439
description:dbname=mydb
description:user=mydb_user
description:sslmode=require
```

peut être retrouvée avec l'URL LDAP suivante :

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?description?one?
(cn=mydatabase)
```

Dans le fichier de service, vous pouvez mélanger des entrées standards avec des recherches LDAP. Voici un exemple complet d'un bloc dans `pg_service.conf` :

```
    # seuls l'hôte et le port sont stockés dans LDAP,
    # spécifiez explicitement le nom de la base et celui de
 l'utilisateur
    [customerdb]
    dbname=clients
    user=utilisateurappl
    ldap://ldap.acme.com/cn=serveur,cn=hosts?pgconnectinfo?base?
(objectclass=*)
```

34.19. Support de SSL

[4] https://datatracker.ietf.org/doc/html/rfc1959

PostgreSQL dispose d'un support natif des connexions SSL pour chiffrer les connexions client/serveur en utilisant TLS pour améliorer ainsi la sécurité. Voir la Section 19.9 pour des détails sur la fonctionnalité SSL côté serveur.

libpq lit le fichier de configuration système d'OpenSSL. Par défaut, ce fichier est nommé `openssl.cnf` et est placé dans le répertoire indiqué par `openssl version -d`. Cette valeur par défaut peut être surchargée en configurant la variable d'environnement `OPENSSL_CONF` avec le nom du fichier de configuration souhaité.

34.19.1. Vérification par le client du certificat serveur

Par défaut, PostgreSQL ne vérifie pas le certificat du serveur. Cela signifie qu'il est possible de se faire passer pour le serveur final (par exemple en modifiant un enregistrement DNS ou en prenant l'adresse IP du serveur) sans que le client ne le sache. Pour empêcher cette usurpation (*spoofing*), le client doit être capable de vérifier l'identité du serveur via une chaîne de confiance. Une chaîne de confiance est établie en plaçant sur un ordinateur le certificat racine (auto-signé) d'une autorité de certification (CA), et sur un autre ordinateur un certificat feuille *signé* avec le certificat racine. Il est aussi possible d'utiliser un certificat « intermédiaire » signé par le certificat racine et qui signe des certificats feuilles.

Pour permettre au client de vérifier l'identité du serveur, placez un certificat racine sur le client et un certificat feuille signé par le certificat racine sur le serveur. Pour permettre au serveur de vérifier l'identité du client, placez un certificat racine sur le serveur et un certificat feuille signé par le certificat racine sur le client. Un ou plusieurs certificats intermédiaires (habituellement stockés avec le certificat feuille) peuvent aussi être utilisés pour lier le certificat feuille au certificat racine.

Une fois qu'une chaîne de confiance a été établie, il existe deux façons pour le client de valider le certificat feuille envoyé par le serveur. Si le paramètre `sslmode` est configuré à `verify-ca`, libpq vérifiera qu'il peut faire confiance au serveur en vérifiant la chaîne des certificats jusqu'au certificat racine stocké sur le client. Si `sslmode` est configuré à `verify-full`, libpq va *aussi* vérifier que le nom d'hôte du serveur correspond au nom stocké dans le certificat du serveur. La connexion SSL échouera si le certificat du serveur n'établit pas ces correspondances. La connexion SSL échouera si le certificat du serveur ne peut pas être vérifié. `verify-full` est recommandé pour les environnements les plus sensibles à la sécurité.

En mode `verify-full`, le nom de l'hôte est mis en correspondance avec le ou les attributs `Subject Alternative Name` (SAN) du certificat, ou avec l'attribut `Common Name` si aucun SAN de type `dNSName` n'est présent. Si le nom du certificat débute avec le caractère étoile (`*`), l'étoile sera traitée comme un métacaractère qui correspondra à tous les caractères *à l'exception* du point. Cela signifie que le certificat ne pourra pas correspondre à des sous-domaines. Si la connexion se fait en utilisant une adresse IP au lieu d'un nom d'hôte, l'adresse IP sera vérifiée (sans faire de résolution DNS) avec les SAN de type `iPAddress` ou `dNSName`. Si aucun SAN `iPAddress` n'est présent et qu'aucun SAN `dNSName` correspondant n'est présent, l'adresse IP de l'hôte est vérifié avec l'attribut Common Name.

> **Note**
>
> Pour la compatibilité avec les anciennes versions de PostgreSQL, l'adresse IP de l'hôte est vérifiée d'une façon différente à partir de RFC 6125[5]. L'adresse IP de l'hôte est toujours vérifiée avec avec les SAN `dNSName` ainsi qu'avec les SAN `iPAddress` et peut être vérifiée avec l'attribut Common Name si aucun SAN n'existe.

Pour permettre la vérification du certificat du serveur, un ou plusieurs certificats racines doivent être placés dans le fichier `~/.postgresql/root.crt` du répertoire personnel de l'utilisateur (sur Windows, le fichier est nommé `%APPDATA%\postgresql\root.crt`). Les certificats

[5] https://datatracker.ietf.org/doc/html/rfc6125

intermédiaires doivent aussi être ajoutés au fichier s'ils sont nécessaires pour lier la chaîne de certificats envoyée par le serveur aux certificats racines stockés sur le client.

Les entrées de la liste de révocation des certificats (CRL) sont aussi vérifiées si le fichier `~/.postgresql/root.crl` existe (`%APPDATA%\postgresql\root.crl` sur Microsoft Windows).

L'emplacement du certificat racine et du CRL peuvent être changés avec les paramètres de connexion `sslrootcert` et `sslcrl`, ou les variables d'environnement `PGSSLROOTCERT` et `PGSSLCRL`. `sslcrldir` ou la variable d'environnement `PGSSLCRLDIR` peuvent aussi être utilisées pour indiquer un répertoire contenant les fichiers CRL.

> **Note**
>
> Par compatibilité avec les anciennes versions de PostgreSQL, si un certificat racine d'autorité existe, le comportement de `sslmode=require` sera identique à celui de `verify-ca`. Cela signifie que le certificat du serveur est validé par l'autorité de certification. Il ne faut pas se baser sur ce comportement. Les applications qui ont besoin d'une validation du certificat doivent toujours utiliser `verify-ca` ou `verify-full`.

34.19.2. Certificats des clients

Si le serveur tente de vérifier l'identité du client en réclamant le certificat feuille du client, libpq enverra lse certificats stockés dans le fichier `~/.postgresql/postgresql.crt` du répertoire personnel de l'utilisateur. Les certificats doivent former une chaîne jusqu'au certificat racine de confiance du serveur. Un fichier de clé privé correspondant `~/.postgresql/postgresql.key` doit aussi être présent. Sur Microsoft Windows, ces fichiers sont nommés `%APPDATA%\postgresql \postgresql.crt` et `%APPDATA%\postgresql\postgresql.key`. L'emplacement des fichiers certificat et clé peut être surchargé par les paramètres de connexion `sslcert` et `sslkey`, ou par les variables d'environnement `PGSSLCERT` et `PGSSLKEY`.

Sur les systèmes Unix, les droits sur le fichier de clé privée ne doit pas permettre l'accès au monde et au groupe ; vous pouvez vous en assurer avec une commande telle que `chmod 0600 ~/.postgresql/postgresql.key`. Il est aussi possible de rendre root propriétaire deu fichier et d'avoir le droit d'accès pour le groupe (autrement dit, les droits `0640`). Cette configuration est prévue pour les installations où les fichiers certificat et clé sont gérés par le système d'exploitation. L'utilisateur de libpq devra alors devenir membre du group qui a accès à ces fichiers certificat et clé. (Sur Microsoft Windows, aucune vérification n'est effectuée sur les droits des fichiers car le répertoire `%APPDATA%\postgresql` est supposé sécurisé.)

Le premier certificat dans `postgresql.crt` doit être le certificat du client parce qu'il doit correspondre à la clé privée du client. Les certificats « intermédiaires » peuvent être ajoutés au fichier en option -- faire ainsi permet d'éviter d'avoir à stocker les certificats intermédiaires sur le serveur (ssl_ca_file).

Le certificat et la clé doit être en format PEM ou ASN.1 DER.

La clé peut être stockée en clair dans le texte ou chiffrée avec un passphrase utilisant un algorithme supporté par OpenSSL, comme AES-128. Si la clé est stockée chiffrée, alors la passphrase doit être fournie dans l'option de connexion sslpassword. Si une clé chiffrée est fournie et que l'option `sslpassword` est absente ou vide, une invite de saisie de mot de passe interactive par OpenSSL s'affichera avec `Enter PEM pass phrase:` si un TTY est présent. Les applications peuvent passer outre l'invite du certificat client et la gestion du paramètre `sslpassword` en fournissant leur propre callback pour la clé du mot de passe, voir `PQsetSSLKeyPassHook_OpenSSL`.

Pour des instructions sur la création de certificats, voir Section 19.9.5.

34.19.3. Protection fournie dans les différents modes

Les différentes valeurs du paramètre `sslmode` fournissent différents niveaux de protection. SSL peut fournir une protection contre trois types d'attaques différentes :

Écoute clandestine (*eavesdropping*)

Si une tierce partie peut examiner le trafic réseau entre le client et le serveur, elle peut lire à la fois les informations de connexion (dont le nom de l'utilisateur et son mot de passe) ainsi que les données transmises SSL utilise le chiffrement pour empêcher cela.

Homme du milieu (MITM, *Man in the middle*)

Si une tierce partie peut modifier les données transitant entre le client et le serveur, il peut prétendre être le serveur et, du coup, voir et modifier les données *y compris si elles sont chiffrées*. La tierce partie peut ensuite renvoyer les informations de connexion et les données au serveur d'origine, rendant impossible à ce dernier la détection de l'attaque. Les vecteurs habituels pour parvenir à ce type d'attaque sont l'empoisonnement des DNS (*DNS poisoning*) et le détournement d'adresses (*address hijacking*), où le client est dirigé vers un autre serveur que celui attendu. Il existe encore plusieurs autres méthodes pour accomplir ceci. SSL utilise la vérification des certificats pour l'empêcher, en authentifiant le serveur auprès du client.

Usurpation d'identité

Si une tierce partie peut prétendre être un client autorisé, il peut tout simplement accéder aux données auquel il n'a pas droit. Typiquement, cela peut arriver avec une gestion incorrecte des mots de passe. SSL utilise les certificats clients pour empêcher ceci, en s'assurant que seuls les propriétaires de certificats valides peuvent accéder au serveur.

Pour qu'une connexion soit sûre, l'utilisation de SSL doit être configurée *sur le client et sur le serveur* avant que la connexion ne soit effective. Si elle n'est configurée que sur le serveur, le client pourrait envoyer des informations sensibles (comme les mots de passe) avant de savoir que le serveur exige une haute sécurité. Dans libpq, les connexions sécurisées peuvent être garanties en configurant le paramètre `sslmode` à `verify-full` ou `verify-ca`, et en fournissant au système un certificat racine à vérifier. Ceci est analogue à l'utilisation des URL `https` pour la navigation web chiffrée.

Une fois que le serveur est authentifié, le client peut envoyer des données sensibles. Cela signifie que, jusqu'à ce point, le client n'a pas besoin de savoir si les certificats seront utilisés pour l'authentification ne le spécifier que dans la configuration du serveur est donc sûr.

Toutes les options SSL impliquent une charge supplémentaire sous forme de chiffrement et d'échange de clés. Il y a donc un compromis à trouver entre performance et sécurité. Tableau 34.1 illustre les risques que les différentes valeurs de `sslmode` cherchent à protéger, et ce que cela apporte en sécurité et fait perdre en performances.

Tableau 34.1. Description des modes SSL

`sslmode`	Protection contre l'écoute clandestine	Protection contre l'attaque MITM	Remarques
`disable`	Non	Non	Peu m'importe la sécurité, et je ne veux pas le coût apporté par le chiffrement.
`allow`	Peut-être	Non	Peu m'importe la sécurité, mais je vais accepter le coût du chiffrement si le serveur insiste.
`prefer`	Peut-être	Non	Peu m'importe le chiffrement, mais j'accepte le coût du chiffrement si le serveur le supporte.

sslmode	Protection contre l'écoute clandestine	Protection contre l'attaque MITM	Remarques
require	Oui	Non	Je veux chiffrer mes données, et j'en accepte le coût. Je fais confiance au réseau pour me garantir que je me connecterai toujours au serveur que je veux.
verify-ca	Oui	Dépend de la politique de la CA	Je veux chiffrer mes données, et j'en accepte le coût. Je veux être sûr que je me connecte à un serveur en qui j'ai confiance.
verify-full	Oui	Oui	Je veux chiffrer mes données, et j'en accepte le coût. Je veux être sûr que je me connecte à un serveur en qui j'ai confiance et que c'est bien celui que j'ai indiqué.

La différence entre `verify-ca` et `verify-full` dépend de la politique de la CA racine. Si une CA publique est utilisé, `verify-ca` permet les connexions à un serveur que *quelqu'un d'autre* a pu enregistrer pour cette CA. Dans ce cas, `verify-full` devrait toujours être utilisé. Si une CA locale est utilisée, voire un certificat auto-signé, utiliser `verify-ca` fournit souvent suffisamment de protection.

La valeur par défaut pour `sslmode` est `prefer`. Comme l'indique la table ci-dessus, cela n'a pas de sens d'un point de vue de la sécurité, et ne promet, si elle possible, qu'un surcoût en terme de performance. Cette valeur est fournie par défaut uniquement pour la compatibilité descendante, et n'est pas recommandée pour les déploiements de serveurs nécessitant de la sécurité.

34.19.4. Utilisation des fichiers SSL

Tableau 34.2 résume les fichiers liés à la configuration de SSL sur le client.

Tableau 34.2. Utilisation des fichiers SSL libpq/client

Fichier	Contenu	Effet
`~/.postgresql/postgresql.crt`	certificat client	envoyé au serveur
`~/.postgresql/postgresql.key`	clé privée du client	prouve le certificat client envoyé par l'utilisateur ; n'indique pas que le propriétaire du certificat est de confiance
`~/.postgresql/root.crt`	autorités de confiance	vérifie que le certificat du serveur est signé par une autorité de confiance
`~/.postgresql/root.crl`	certificats révoqués par les autorités de confiance	le certificat du serveur ne doit pas être sur cette liste

34.19.5. Initialisation de la bibliothèque SSL

Si votre application initialise les bibliothèques `libssl` et/ou `libcrypto` et que libpq est construit avec le support de SSL, vous devez appeler la fonction `PQinitOpenSSL` pour indiquer à libpq que les bibliothèques `libssl` et/ou `libcrypto` ont été initialisées par votre application, de façon à ce que libpq n'initialise pas elle aussi ces bibliothèques. Néanmoins, ceci n'est pas nécessaire lors

de l'utilisation d'OpenSSL version 1.1.0 ou supérieure, car les initialisations dupliquées ne sont plus problématiques.

`PQinitOpenSSL`

Permet aux applications de sélectionner les bibliothèques de sécurité à initialiser.

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

Quand *`do_ssl`* est différent de zéro, libpq initialisera la bibliothèque OpenSSL avant d'ouvrir une connexion à la base de données. Quand *`do_crypto`* est différent de zéro, la bibliothèque `libcrypto` sera initialisée. Par défaut (si `PQinitOpenSSL` n'est pas appelé), les deux bibliothèques sont initialisées. Quand le support de SSL n'est pas intégré, cette fonction est présente mais ne fait rien.

Si votre application utilise et initialise soit OpenSSL soit `libcrypto`, vous *devez* appeler cette fonction avec des zéros pour les paramètres appropriés avant d'ouvrir la première connexion à la base de données. De plus, assurez-vous que vous avez fait cette initialisation avant d'ouvrir une connexion à la base de données.

`PQinitSSL`

Permet aux applications de sélectionner les bibliothèques de sécurité à initialiser.

```
void PQinitSSL(int do_ssl);
```

Cette fonction est équivalente à `PQinitOpenSSL(do_ssl, do_ssl)`. C'est suffisant pour les applications qui initialisent à la fois OpenSSL et `libcrypto` ou aucune des deux.

`PQinitSSL` est présente depuis PostgreSQL 8.0, alors que `PQinitOpenSSL` a été ajoutée dans PostgreSQL 8.4, donc `PQinitSSL` peut être préférée pour les applications qui ont besoin de fonctionner avec les anciennes versions de libpq.

34.20. Comportement des programmes threadés

libpq est réentrante et compatible avec les threads par défaut. Vous pourriez avoir besoin d'utiliser des options de compilation supplémentaires en ligne lorsque vous compilez le code de votre application. Référez-vous à la documentation de votre système pour savoir comment construire des applications avec threads ou recherchez `PTHREAD_CFLAGS` et `PTHREAD_LIBS` dans `src/Makefile.global`. Cette fonction permet d'interroger le statut de compatibilité de libpq avec les threads :

`PQisthreadsafe`

Renvoie le statut de compatibilité avec les threads pour libpq library.

```
int PQisthreadsafe();
```

Renvoie 1 si libpq supporte les threads, 0 dans le cas contraire.

Une restriction : il ne doit pas y avoir deux threads manipulant le même objet `PGconn` à la fois. En particulier, vous ne pouvez pas lancer des commandes concurrentes depuis des threads différents à travers le même objet de connexion (si vous avez besoin de lancer des commandes concurrentes, utilisez plusieurs connexions).

Les objets PGresult sont normalement en lecture seule après leur création et, du coup, peuvent être passés librement entre les threads. Néanmoins, si vous utilisez une des fonctions décrites dans Section 34.12 ou Section 34.14 qui modifient PGresult, il est de votre responsabilité d'éviter des opérations concurrentes sur le même PGresult.

Les fonctions obsolètes PQrequestCancel et PQoidStatus ne gèrent pas les threads et ne devraient pas être utilisées dans des programmes multithreads. PQrequestCancel peut être remplacé par PQcancel. PQoidStatus peut être remplacé par PQoidValue.

Si vous utilisez Kerberos avec votre application (en plus de libpq), vous aurez besoin de verrouiller les appels Kerberos car les fonctions Kerberos ne supportent pas les threads. Voir la fonction PQregisterThreadLock dans le code source de libpq sur comment faire un verrouillage coopératif entre libpq et votre application.

34.21. Construire des applications avec libpq

Pour construire (c'est-à-dire compiler et lier) un programme utilisant libpq, vous avez besoin de faire tout ce qui suit :

- Ajoutez le fichier d'en-tête libpq-fe.h :

  ```
  #include <libpq-fe.h>
  ```

  Si vous ne le faites pas, alors vous obtiendrez normalement des messages d'erreurs similaires à :

  ```
  foo.c: In function `main':
         foo.c:34: `PGconn' undeclared (first use in this function)
         foo.c:35: `PGresult' undeclared (first use in this
   function)
         foo.c:54: `CONNECTION_BAD' undeclared (first use in this
   function)
         foo.c:68: `PGRES_COMMAND_OK' undeclared (first use in this
   function)
         foo.c:95: `PGRES_TUPLES_OK' undeclared (first use in this
   function)
  ```

- Faites pointer votre compilateur sur le répertoire où les fichiers d'en-tête de PostgreSQL ont été installés, en lui fournissant l'option -I*répertoire* (dans certains cas, le compilateur cherchera dans le répertoire en question par défaut, donc vous pouvez omettre cette option). Par exemple, votre ligne de commande de compilation devrait ressembler à ceci :

  ```
  cc -c -I/usr/local/pgsql/include testprog.c
  ```

  Si vous utilisez des makefiles, alors ajoutez cette option à la variable CPPFLAGS :

  ```
  CPPFLAGS += -I/usr/local/pgsql/include
  ```

  S'il y a une chance que votre programme soit compilé par d'autres utilisateurs, alors vous ne devriez pas coder en dur l'emplacement du répertoire. À la place, vous pouvez exécuter l'outil pg_config pour trouver où sont placés les fichiers d'en-tête sur le système local :

  ```
  $ pg_config --includedir
        /usr/local/include
  ```

  Si vous avez installé pkg-config, vous pouvez lancer à la place :

```
$ pkg-config --cflags libpq
-I/usr/local/include
```

Notez que l'option `-I` sera déjà précisée au début du chemin.

Une erreur dans la spécification de la bonne option au compilateur résultera en un message d'erreur tel que

```
testlibpq.c:8:22: libpq-fe.h: No such file or directory
```

- Lors de l'édition des liens du programme final, spécifiez l'option `-lpq` de façon à ce que les bibliothèques libpq soient intégrées, ainsi que l'option `-L`*`répertoire`* pour faire pointer le compilateur vers le répertoire où les bibliothèques libpq résident. (Là encore le compilateur cherchera certains répertoires par défaut). Pour une portabilité maximale, placez l'option `-L` avant l'option `-lpq`. Par exemple :

```
cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -
lpq
```

Vous pouvez aussi récupérer le répertoire des bibliothèques en utilisant `pg_config` :

```
$ pg_config --libdir
       /usr/local/pgsql/lib
```

Ou utiliser de nouveau `pkg-config` :

```
$ pkg-config --libs libpq
-L/usr/local/pgsql/lib -lpq
```

Notez aussi que cela affiche les options complètes, pas seulement le chemin.

Les messages d'erreurs liés à des problèmes de ce style pourraient ressembler à ce qui suit.

```
testlibpq.o: In function `main':
       testlibpq.o(.text+0x60): undefined reference to
 `PQsetdbLogin'
       testlibpq.o(.text+0x71): undefined reference to `PQstatus'
       testlibpq.o(.text+0xa4): undefined reference to
 `PQerrorMessage'
```

Ceci signifie que vous avez oublié `-lpq`.

```
/usr/bin/ld: cannot find -lpq
```

Ceci signifie que vous avez oublié l'option `-L` ou que vous n'avez pas indiqué le bon répertoire.

34.22. Exemples de programmes

Ces exemples (et d'autres) sont disponibles dans le répertoire `src/test/examples` de la distribution des sources.

Exemple 34.1. Premier exemple de programme pour libpq

```
/*
 * src/test/examples/testlibpq.c
 *
 *
 * testlibpq.c
 *
 *      Teste la version C de libpq, la bibliothèque frontend de
 PostgreSQL.
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    int         nFields;
    int         i,
                j;

    /*
     * Si l'utilisateur fournit un paramètre sur la ligne de
 commande,
     * l'utiliser comme une chaîne conninfo ; sinon prendre par
 défaut
     * dbname=postgres et utiliser les variables d'environnement ou
 les
     * valeurs par défaut pour tous les autres paramètres de
 connexion.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Crée une connexion à la base de données */
    conn = PQconnectdb(conninfo);

    /* Vérifier que la connexion au backend a été faite avec succès
 */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }
```

```
   /* Initialise un search path sûr, pour qu'un utilisateur
      malveillant ne puisse prendre le contrôle. */
   res = PQexec(conn,
                "SELECT pg_catalog.set_config('search_path', '',
false)");
   if (PQresultStatus(res) != PGRES_TUPLES_OK)
   {
       fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
       PQclear(res);
       exit_nicely(conn);
   }

   /*
   * Il faut libérer PGresult avec PQclear dès que l'on en a plus
besoin pour
   * éviter les fuites de mémoire.
   */
   PQclear(res);

   /*
    * Notre exemple inclut un curseur, pour lequel il faut que
nous soyons dans
    * un bloc de transaction. On pourrait tout faire dans un seul
PQexec()
    * d'un "select * from pg_database" mais c'est trop trivial
pour faire
    * un bon exemple.
    */

   /* Démarre un bloc de transaction */
   res = PQexec(conn, "BEGIN");
   if (PQresultStatus(res) != PGRES_TUPLES_OK)
   {
       fprintf(stderr, "BEGIN command failed: %s",
PQerrorMessage(conn));
       PQclear(res);
       exit_nicely(conn);
   }
   PQclear(res);

   /*
    * Récupère les lignes de pg_database, catalogue système des
bases de
    * données
    */
   res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from
pg_database");
   if (PQresultStatus(res) != PGRES_COMMAND_OK)
   {
       fprintf(stderr, "DECLARE CURSOR failed: %s",
PQerrorMessage(conn));
       PQclear(res);
       exit_nicely(conn);
   }
   PQclear(res);

   res = PQexec(conn, "FETCH ALL in myportal");
   if (PQresultStatus(res) != PGRES_TUPLES_OK)
```

```
    {
        fprintf(stderr, "FETCH ALL failed: %s",
 PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    /* affiche d'abord les noms des attributs */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* puis affiche les lignes */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* ferme le portail... nous ne cherchons pas s'il y a des
 erreurs... */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* termine la transaction */
    res = PQexec(conn, "END");
    PQclear(res);

    /* ferme la connexion à la base et nettoie */
    PQfinish(conn);

    return 0;
}
```

Exemple 34.2. Deuxième exemple de programme pour libpq

```
/*
 * src/test/examples/testlibpq2.c
 *
 *
 * testlibpq2.c
 *      Teste l'interface de notification asynchrone
 *
 * Démarrez ce programme, puis depuis psql dans une autre fenêtre
 faites
 *   NOTIFY TBL2;
 * Répétez quatre fois pour terminer ce programme.
 *
 * Ou, si vous voulez vous faire plaisir, faites ceci :
 * remplissez une base avec les commandes suivantes
 * (issues de src/test/examples/testlibpq2.sql):
 *
 *   CREATE SCHEMA TESTLIBPQ2;
 *   SET search_path = TESTLIBPQ2;
```

```
 *   CREATE TABLE TBL1 (i int4);
 *
 *   CREATE TABLE TBL2 (i int4);
 *
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
 *
 * Démarrez ce programme, puis depuis psql faites quatre fois :
 *
 *   INSERT INTO TESTLIBPQ2.TBL1 VALUES (10);
 */

#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#ifdef HAVE_SYS_SELECT_H
#include <sys/select.h>
#endif

#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
    int         nnotifies;

    /*
     * Si l'utilisateur fournit un paramètre sur la ligne de
 commande,
     * l'utiliser comme une chaîne conninfo ; sinon prendre par
 défaut
     * dbname=postgres et utiliser les variables d'environnement ou
 les
     * pour tous les autres paramètres de connection.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Se connecte à la base */
```

```
   conn = PQconnectdb(conninfo);

   /* Vérifier que la connexion au backend a été faite avec succès
*/
   if (PQstatus(conn) != CONNECTION_OK)
   {
       fprintf(stderr, "%s", PQerrorMessage(conn));
       exit_nicely(conn);
   }

   /* Initialise un search path sûr, pour qu'un utilisateur
      malveillant ne puisse prendre le contrôle. */
   res = PQexec(conn,
                "SELECT pg_catalog.set_config('search_path', '',
false)");
   if (PQresultStatus(res) != PGRES_COMMAND_OK)
   {
       fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
       PQclear(res);
       exit_nicely(conn);
   }

   /*
   * Il faut libérer PGresult avec PQclear dès que l'on en a plus
besoin pour
   * éviter les fuites de mémoire.
   */
   PQclear(res);

   /*
    * Lance une commande LISTEN pour démarrer des notifications
depuis le
    * NOTIFY.
    */
   res = PQexec(conn, "LISTEN TBL2");
   if (PQresultStatus(res) != PGRES_COMMAND_OK)
   {
       fprintf(stderr, "LISTEN command failed: %s",
PQerrorMessage(conn));
       PQclear(res);
       exit_nicely(conn);
   }

   PQclear(res);

   /* Quitte après avoir reçu quatre notifications. */
   nnotifies = 0;
   while (nnotifies < 4)
   {
       /*
        * Dort jusqu'à ce que quelque chose arrive sur la
connexion. Nous
        * utilisons select(2) pour attendre une entrée, mais vous
pouvez
        * utiliser poll() ou des fonctions similaires.
        */
       int         sock;
       fd_set      input_mask;
```

```
        sock = PQsocket(conn);

        if (sock < 0)
            break;              /* ne devrait pas arriver */

        FD_ZERO(&input_mask);
        FD_SET(sock, &input_mask);

        if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
        {
            fprintf(stderr, "select() failed: %s\n",
 strerror(errno));
            exit_nicely(conn);
        }

        /* Cherche une entrée */
        PQconsumeInput(conn);
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                    "ASYNC NOTIFY of '%s' received from backend PID
 %d\n",
                    notify->relname, notify->be_pid);
            PQfreemem(notify);
            nnotifies++;
            PQconsumeInput(conn);
        }
    }

    fprintf(stderr, "Done.\n");

    /* ferme la connexion à la base et nettoie */
    PQfinish(conn);

    return 0;
}
```

Exemple 34.3. Troisième exemple de programme pour libpq

```
/*
 * src/test/examples/testlibpq3.c
 *
 *
 * testlibpq3.c
 *      Teste des paramètres délicats et des entrées-sorties
 binaires.
 *
 * Avant de lancer ceci, remplissez une base avec les commandes
 suivantes
 * (fournies dans src/test/examples/testlibpq3.sql):
 *
 * CREATE SCHEMA testlibpq3;
 * SET search_path = testlibpq3;
 * SET standard_conforming_strings = ON;
 * CREATE TABLE test1 (i int4, t text, b bytea);
 *
```

```
 * INSERT INTO test1 values (1, 'joe''s place',
 '\000\001\002\003\004');
 * INSERT INTO test1 values (2, 'ho there',
 '\004\003\002\001\000');
 *
 * La sortie attendue est :
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
 * tuple 0: got
 *  i = (4 bytes) 2
 *  t = (8 bytes) 'ho there'
 *  b = (5 bytes) \004\003\002\001\000
 */

#ifdef WIN32
#include <windows.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

/*
 * Cette fonction affiche un résultat qui est la récupération
 * au format binaire d'une table définie dans le commentaire ci-
dessus.
 * Nous l'avons extraite car la fonction main() l'utilise deux
 fois.
 */
static void
show_binary_results(PGresult *res)
{
    int         i,
                j;
    int         i_fnum,
                t_fnum,
                b_fnum;

    /*
```

```
     * Utilise PQfnumber pour éviter de deviner l'ordre des champs
     * dans le résultat
     */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char       *iptr;
        char       *tptr;
        char       *bptr;
        int         blen;
        int         ival;

        /*
         * Récupère les valeurs des champs
         * (on ignore la possibilité qu'ils soient NULL !)
         */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);
        bptr = PQgetvalue(res, i, b_fnum);

        /*
         * La représentation binaire d'INT4 est dans l'ordre
d'octets
         * du réseau (network byte order), qu'il vaut mieux forcer
à
         * l'ordre local.
         */
        ival = ntohl(*((uint32_t *) iptr));

        /*
         * La représentation binaire de TEXT est, hé bien, du
texte,
         * et puisque libpq a été assez sympa pour rajouter un
octet zéro,
         * cela marchera très bien en tant que chaîne C.
         *
         * La représentation binaire de BYTEA est un paquet
d'octets,
         * pouvant incorporer des nulls, donc nous devons faire
attention à
         * la longueur des champs.
         */
        blen = PQgetlength(res, i, b_fnum);

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
               PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
               PQgetlength(res, i, t_fnum), tptr);
        printf(" b = (%d bytes) ", blen);
        for (j = 0; j < blen; j++)
            printf("\\%03o", bptr[j]);
        printf("\n\n");
    }
}
```

```
int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *paramValues[1];
    int         paramLengths[1];
    int         paramFormats[1];
    uint32_t    binaryIntVal;

    /*
     * Si l'utilisateur fournit un paramètre sur la ligne de
 commande,
     * l'utiliser comme une chaîne conninfo ; sinon prendre par
 défaut
     * dbname=postgres et utiliser les variables d'environnement ou
 les
     * valeurs par défaut pour tous les autres paramètres de
 connexion.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Crée une connexion à la base */
    conn = PQconnectdb(conninfo);

    /* Vérifie que la connexion à la base s'est bien déroulée */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /*
    * Il faut libérer PGresult avec PQclear dès que l'on en a plus
 besoin pour
    * éviter les fuites de mémoire.
    */
    res = PQexec(conn, "SET search_path = testlibpq3");
    if (PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    /*
     * Le sujet de ce programme est d'illustrer l'utilisation de
 PQexecParams()
     * avec des paramètres délicats aussi bien que la transmission
 de
     * données binaires.
     *
```

```
    * Ce premier exemple transmet les paramètres en tant que
texte, mais
    * reçoit les résultats en format binaire. Avec des paramètres
    * un peu délicats il n'y a pas besoin de nettoyage fastidieux
en
    * terme de guillemets et d'échappement, même si les données
sont du
    * texte. Notez que nous ne faisons rien de spécial avec les
guillemets
    * dans la valeur du paramètre.
    */

   /* Voici notre paramètre délicat */
   paramValues[0] = "joe's place";

   res = PQexecParams(conn,
                      "SELECT * FROM test1 WHERE t = $1",
                      1,       /* un paramètre */
                      NULL,    /* laissons le backend déduire le
type */
                      paramValues,
                      NULL,    /* pas besoin de la longueur des
paramètres,
                                  c'est du texte */
                      NULL,    /* par défaut tous les paramètres
sont du texte */
                      1);      /* demande le résultat en binaire
*/

   if (PQresultStatus(res) != PGRES_TUPLES_OK)
   {
       fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
       PQclear(res);
       exit_nicely(conn);
   }

   show_binary_results(res);

   PQclear(res);

   /*
    * Dans ce second exemple, on transmet un paramètre entier sous
    * forme binaire, et on récupère à nouveau les paramètres sous
forme
    * binaire.
    *
    * Bien que nous disions à PQexecParams que nous laissons le
backend
    * déduire le type du paramètre, nous forçons la décision en
convertissant
    * le symbole du paramètre dans le texte de la requête. C'est
une bonne
    * précaution quand on envoie des paramètres binaires.
    */

   /* Convertit l'entier "2" dans l'ordre d'octets du réseau */
   binaryIntVal = htonl((uint32_t) 2);
```

```
    /* Met en place les tableaux de paramètres pour PQexecParams */
    paramValues[0] = (char *) &binaryIntVal;
    paramLengths[0] = sizeof(binaryIntVal);
    paramFormats[0] = 1;        /* binary */

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE i = $1::int4",
                       1,       /* un paramètre */
                       NULL,    /* laissons le backend déduire le
 type
                                   du paramètre */
                       paramValues,
                       paramLengths,
                       paramFormats,
                       1);      /* demande des résultats binaires
 */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    show_binary_results(res);

    PQclear(res);

    /* ferme la connexion et nettoie */
    PQfinish(conn);

    return 0;
}
```

Chapitre 35. Large Objects

PostgreSQL dispose de fonctionnalités concernant les *Large Objects*, fournissant un accès de style flux aux données utilisateurs stockées dans une structure spéciale. L'accès en flux est utile pour travailler avec des valeurs de données trop larges pour être manipulées convenablement en entier.

Ce chapitre décrit l'implantation, la programmation et les interfaces du langage de requêtes pour les données de type « Large Objects » dans PostgreSQL. Nous utilisons la bibliothèque C libpq pour les exemples de ce chapitre mais la plupart des interfaces natives de programmation de PostgreSQL supportent des fonctionnalités équivalentes. D'autres interfaces pourraient utiliser l'interface des « Large Objects » en interne pour fournir un support générique des valeurs larges. Ceci n'est pas décrit ici.

35.1. Introduction

Tous les « Large Objects » sont enregistrés dans un seul catalogue système nommé `pg_largeobject`. Chaque « Large Object » a aussi une entrée dans la table système `pg_largeobject_metadata`. Les « Large Objects » peuvent être créés, modifiés et supprimés en utilisant une API de lecture/écriture très similaire à celle des opérations standards sur les fichiers.

PostgreSQL accepte aussi un système de stockage appelé « TOAST » qui stocke automatiquement les valeurs ne tenant pas sur un bloc de la base de données dans une aire de stockage secondaire par table. Ceci rend partiellement obsolète la fonctionnalité des « Large Objects ». Un avantage restant des « Large Objects » est qu'il autorise les valeurs de plus de 4 To en taille alors que les champs TOAST peuvent être d'au plus 1 Go. De plus, lire et mettre à jour des portions d'un « Large Object » se fait très simplement en conservant de bonnes performances alors que la plupart des opérations sur un champ mis dans la partie TOAST demandera une lecture ou une écriture de la valeur totale.

35.2. Fonctionnalités de l'implantation

L'implantation des « Large Objects » les divise en « morceaux » (*chunks* dans la version originale) stockés dans les lignes de la base de données. Un index B-tree garantit des recherches rapides sur le numéro du morceau lors d'accès aléatoires en lecture et écriture.

Les morceaux enregistrés pour un « Large Object » n'ont pas besoin d'être contigües. Par exemple, si une application ouvre un nouveau « Large Object », recherche la position 1000000, et y écrit quelques octets, cela ne résulte pas en l'allocation de 1000000 octets de stockage, mais seulement les morceaux couvrant les octets de données écrites. Néanmoins, une opération de lecture lira des zéros pour tous les emplacements non alloués précédant le dernier morceau existant. Cela correspond au comportement habituel des fichiers « peu alloués » dans les systèmes de fichiers Unix.

À partir de PostgreSQL 9.0, les « Large Objects » ont un propriétaire et un ensemble de droits d'accès pouvant être gérés en utilisant les commandes GRANT et REVOKE. Les droits `SELECT` sont requis pour lire un « Large Object », et les droits `UPDATE` sont requis pour écrire ou tronquer. Seul le propriétaire du « Large Object » ou un superutilisateur peut supprimer, ajouter un commentaire ou modifier le propriétaire d'un « Large Object ». Pour ajuster le comportement en vue de la compatibilité avec les anciennes versions, voir le paramètre lo_compat_privileges.

35.3. Interfaces client

Cette section décrit les possibilités de la bibliothèque d'interface client libpq de PostgreSQL permettant d'accéder aux « Large Objects ». L'interface des « Large Objects » de PostgreSQL est modelée d'après l'interface des systèmes de fichiers Unix avec des analogies pour les fonctions `open`, `read`, `write`, `lseek`, etc.

Toutes les manipulations de « Large Objects » utilisant ces fonctions *doivent* prendre place dans un bloc de transaction SQL car les descripteurs de fichiers des « Large Objects » sont seulement valides pour la durée d'une transaction.

Si une erreur survient lors de l'exécution de ces fonctions, la fonction renverra une valeur autrement impossible, typiquement 0 ou -1. Un message décrivant l'erreur est stocké dans l'objet de connexion et peut être récupéré avec la fonction `PQerrorMessage`.

Les applications clientes qui utilisent ces fonctions doivent inclure le fichier d'en-tête `libpq/libpq-fs.h` et se lier à la bibliothèque libpq.

Les applications clientes ne peuvent pas utiliser ces fonctions quand une connexion libpq est en mode pipeline.

35.3.1. Créer un « Large Object »

La fonction

```
Oid lo_create(PGconn *conn, Oid lobjId);
```

crée un nouveau « Large Object ». L'OID affecté peut être indiqué par le paramètre *`lobjId`* ; dans ce cas, un échec survient si l'OID est déjà utilisé par un autre Large Object. Si *`lobjId`* vaut `InvalidOid` (zéro), alors `lo_create` affecte un OID inutilisé. La valeur de retour est l'OID assigné au nouveau « Large Object » ou `InvalidOid` (zéro) en cas d'erreur.

Un exemple :

```
inv_oid = lo_create(conn, oid_désiré);
```

L'ancienne fonction

```
Oid lo_creat(PGconn *conn, int
    mode);
```

crée aussi un nouveau « Large Object », toujours affecté à un OID inutilisé. La valeur de retour est l'OID qui a été affecté au nouveau « Large Object » ou `InvalidOid`(zéro) en cas d'échec.

Dans les versions 8.1 et ultérieures de PostgreSQL, le *`mode`* est ignoré, pour que `lo_creat` soit exactement équivalent à `lo_create` avec un deuxième argument à zéro. Néanmoins, il y a peu de raisons d'utiliser `lo_creat` sauf si vous avez besoin d'utiliser des serveurs antérieurs à la version 8.1. Pour travailler avec des serveurs si vieux, vous devez utiliser `lo_creat`, et non pas `lo_create`, et vous devez configurer *`mode`* à une valeur parmi `INV_READ`, `INV_WRITE` et `INV_READ | INV_WRITE`. (Ces constantes symboliques sont définies dans le fichier d'en-tête `libpq/libpq-fs.h`.)

Un exemple :

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

35.3.2. Importer un « Large Object »

Pour importer un fichier du système d'exploitation en tant que « Large Object », appelez

```
Oid lo_import(PGconn *conn, const char *filename);
```

`filename` spécifie le nom du fichier à importer comme « Large Object ». Le code de retour est l'OID assigné au nouveau « Large Object » ou `InvalidOid` (zéro) en cas d'échec. Notez que le fichier est

lu par la bibliothèque d'interface du client, pas par le serveur. Donc il doit exister dans le système de fichier du client et être lisible par l'application du client.

La fonction

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid
 lobjId);
```

importe aussi un nouveau « Large Object ». L'OID à affecter peut être indiqué par *`lobjId`* ; dans ce cas, un échec survient si l'OID est déjà utilisé pour un autre « Large Object ». Si *`lobjId`* vaut `InvalidOid` (zéro) alors `lo_import_with_oid` affecte un OID inutilisé (et donc obtient ainsi le même comportement que `lo_import`). La valeur de retour est l'OID qui a été affecté au nouveau « Large Object » ou `InvalidOid` (zéro) en cas d'échec.

`lo_import_with_oid` est apparu avec PostgreSQL 8.4, et utilise en interne `lo_create` qui était une nouveauté de la version 8.1 ; si cette fonction est exécutée sur un serveur en 8.0, voire une version précédente, elle échouera et renverra `InvalidOid`.

35.3.3. Exporter un « Large Object »

Pour exporter un « Large Object » en tant que fichier du système d'exploitation, appelez

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

L'argument *`lobjId`* spécifie l'OID du « Large Object » à exporter et l'argument *`filename`* spécifie le nom du fichier. Notez que le fichier est écrit par la bibliothèque d'interface du client, pas par le serveur. Renvoie 1 en cas de succès, -1 en cas d'échec.

35.3.4. Ouvrir un « Large Object » existant

Pour ouvrir un « Large Object » existant pour lire ou écrire, appelez

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

L'argument *`lobjId`* spécifie l'OID du « Large Object » à ouvrir. Les bits *`mode`* contrôlent si le « Large Object » est ouvert en lecture (`INV_READ`), en écriture(`INV_WRITE`) ou les deux (ces constantes symboliques sont définies dans le fichier d'en-tête `libpq/libpq-fs.h`). `lo_open` renvoie un descripteur (positif) de « Large Object » pour une utilisation future avec `lo_read`, `lo_write`, `lo_lseek`, `lo_lseek64`, `lo_tell`, `lo_tell64`, `lo_truncate`, `lo_truncate64` et `lo_close`. Le descripteur est uniquement valide pour la durée de la transaction en cours. En cas d'échec, -1 est renvoyé.

Actuellement, le serveur ne fait pas de distinction entre les modes `INV_WRITE` et `INV_READ | INV_WRITE` : vous êtes autorisé à lire à partir du descripteur dans les deux cas. Néanmoins, il existe une différence significative entre ces modes et `INV_READ` seul : avec `INV_READ`, vous ne pouvez pas écrire sur le descripteur et la donnée lue à partir de ce dernier, reflètera le contenu du « Large Object » au moment où `lo_open` a été exécuté dans la transaction active, quelles que soient les possibles écritures par cette transaction ou par d'autres. Lire à partir d'un descripteur ouvert avec `INV_WRITE` renvoie des données reflétant toutes les écritures des autres transactions validées ainsi que les écritures de la transaction en cours. Ceci est similaire à la différence de comportement entre les modes de transaction `REPEATABLE READ` et `READ COMMITTED` pour les requêtes SQL `SELECT`.

`lo_open` échouera si le droit `SELECT` n'est pas disponible pour le « Large Object » ou si `INV_WRITE` est indiqué et que le droit `UPDATE` n'est pas disponible. (Avant PostgreSQL 11, ces

vérifications de droit étaient réalisée pour la première vraie lecture ou écriture utilisant ce descripteur.) Ces vérifications de droit peuvent être désactivées avec le paramètre lo_compat_privileges.

Un exemple :

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

35.3.5. Écrire des données dans un « Large Object »

La fonction

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

écrit *len* octets à partir de *buf* (qui doit avoir *len* comme taille) dans le descripteur *fd* du « Large Object ». L'argument *fd* doit provenir d'un appel précédent à la fonction lo_open. Le nombre d'octets réellement écrits est renvoyé (dans l'implémentation actuelle, c'est toujours égal à *len* sauf en cas d'erreur). Dans le cas d'une erreur, la valeur de retour est -1.

Bien que le paramètre *len* est déclaré size_t, cette fonction rejettera les valeurs plus grandes que INT_MAX. En pratique, il est préférable de transférer les données en plusieurs parties d'au plus quelques méga-octets.

35.3.6. Lire des données à partir d'un « Large Object »

La fonction

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

lit jusqu'à *len* octets à partir du descripteur de fichier *fd* dans *buf* (qui doit avoir pour taille *len*). L'argument *fd* doit avoir été fourni par un appel précédent à la fonction lo_open. Le nombre d'octets réellement lus est renvoyé. Cela sera plus petit que *len* si la fin du « Large Object » est atteint avant. Dans le cas d'une erreur, la valeur de retour est -1.

Bien que le paramètre *len* est déclaré size_t, cette fonction rejettera les valeurs plus grandes que INT_MAX. En pratique, il est préférable de transférer les données en plusieurs parties d'au plus quelques méga-octets.

35.3.7. Recherche dans un « Large Object »

Pour modifier l'emplacement courant de lecture ou écriture associé au descripteur d'un « Large Object », on utilise

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

Cette fonction déplace le pointeur d'emplacement courant pour le descripteur du « Large Object » identifié par *fd* au nouvel emplacement spécifié avec le décalage (*offset*). Les valeurs valides pour *whence* sont SEEK_SET (rechercher depuis le début de l'objet), SEEK_CUR (rechercher depuis la position courante) et SEEK_END (rechercher depuis la fin de l'objet). Le code de retour est le nouvel emplacement du pointeur ou -1 en cas d'erreur.

Lors de la gestion de « Large Objects » qui pourraient dépasser 2 Go, utilisez à la place

```
pg_int64 lo_lseek64(PGconn *conn, int fd, pg_int64 offset, int
 whence);
```

Cette fonction a le même comportement que `lo_lseek` mais elle peut accepter un *`offset`* supérieure à 2 Go et/ou rendre un résultat supérieur à 2 Go. Notez que `lo_lseek` échouera si le nouveau pointeur d'emplacement est supérieur à 2 Go.

`lo_lseek64` apparaît avec PostgreSQL 9.3. Si cette fonction est exécutée sur un serveur de version antérieure, elle échouera et renverra -1.

35.3.8. Obtenir la position de recherche d'un « Large Object »

Pour obtenir la position actuelle de lecture ou écriture d'un descripteur de « Large Object », appelez

```
int lo_tell(PGconn *conn, int fd);
```

S'il n'y a pas d'erreur, la valeur renvoyée est -1.

Lors de la gestion de « Large Objects » qui pourraient dépasser 2 Go, utilisez à la place

```
pg_int64 lo_tell64(PGconn *conn, int fd);
```

Cette fonction a le même comportement que `lo_tell` mais elle peut gérer des objets de plus de 2 Go. Notez que `lo_tell` échouera si l'emplacement de lecture/écrire va au-delà des 2 Go.

`lo_tell64` est disponible depuis la version 9.3 de PostgreSQL. Si cette fonction est utilisée sur une ancienne version, elle échouera et renverra -1.

35.3.9. Tronquer un « Large Object »

Pour tronquer un « Large Object » avec une longueur donnée, on utilise

```
int lo_truncate(PGconn *conn, int fd, size_t len);
```

Cette fonction tronque le « Large Object » décrit par *`fd`* avec la longueur *`len`*. L'argument *`fd`* doit avoir été renvoyé par un appel précédent à `lo_open`. Si la valeur du paramètre *`len`* est plus grande que la longueur actuelle du « Large Object », ce dernier est étendu à la longueur spécifiée avec des octets nuls ('\0'). En cas de succès, `lo_truncate` renvoie 0. En cas d'erreur, il renvoie -1.

L'emplacement de lecture/écriture associé avec le descripteur *`fd`* n'est pas modifié.

Bien que le paramètre *`len`* est déclaré `size_t`, `lo_truncate` rejettera toute longueur supérieure à `INT_MAX`.

Lors de la gestion de « Large Objects » qui pourraient dépasser 2 Go, utilisez à la place

```
int lo_truncate64(PGconn *conn, int fd, pg_int64 len);
```

Cette fonction a le même comportement que `lo_truncate` mais elle peut accepter une valeur supérieure à 2 Go pour le paramètre *`len`*.

`lo_truncate` est une nouveauté de PostgreSQL 8.3 ; si cette fonction est également exécuté sur un version plus ancienne du serveur, elle échouera et retournera -1.

`lo_truncate64` est disponible depuis la version 9.3 de PostgreSQL. Si cette fonction est utilisée sur une ancienne version, elle échouera et renverra -1.

35.3.10. Fermer un descripteur de « Large Object »

Un descripteur de « Large Object » peut être fermé en appelant

```
int lo_close(PGconn *conn, int fd);
```

où *`fd`* est un descripteur de « Large Object » renvoyé par `lo_open`. En cas de succès, `lo_close` renvoie zéro. -1 en cas d'échec.

Tous les descripteurs de « Large Objects » restant ouverts à la fin d'une transaction seront automatiquement fermés.

35.3.11. Supprimer un objet large

Pour supprimer un « Large Object » de la base de données, on utilise

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

L'argument *`lobjId`* spécifie l'OID du « Large Object » à supprimer. En cas d'erreur, le code de retour est -1.

35.4. Fonctions du côté serveur

Les fonctions côté serveur conçues pour la manipulation des « Large Objects » en SQL sont listées dans Tableau 35.1.

Tableau 35.1. Fonctions SQL orientées « Large Object »

Fonction **Description** **Exemple(s)**
`lo_from_bytea` (*`loid`* `oid`, *`data`* `bytea`) → `oid` Crée un « Large Object » et stocke des *`data`* dedans. Si le paramètre *`loid`* est zéro, alors le système va choisir un OID libre, sinon cet OID est utilisé (avec une erreur s'il existe déjà un « Large Object » sur cet OID). En cas de succès, l'OID du « Large Object » est renvoyé. `lo_from_bytea(0, '\xffffff00')` → `24528`
`lo_put` (*`loid`* `oid`, *`offset`* `bigint`, *`data`* `bytea`) → `void` Écrit *`data`* en commençant au décalage donné à l'intérieur du « Large Object », ce dernier étant agrandi si nécessaire. `lo_put(24528, 1, '\xaa')` →
`lo_get` (*`loid`* `oid` [, *`offset`* `bigint`, *`length`* `integer`]) → `bytea` Extrait le contenu du « Large Object », ou une sous-chaîne de celui-ci. `lo_get(24528, 0, 3)` → `\xffaaff`

Il existe d'autres fonctions côté serveur correspondant à chacune des fonctions côté client décrites précédemment. En fait, la plupart des fonctions côté client sont simplement des interfaces vers l'équivalent côté serveur. Celles qu'il est possible d'appeler via des commandes SQL sont `lo_creat`, `lo_create`, `lo_create`, `lo_unlink`, `lo_import` et `lo_export`. Voici des exemples de leur utilisation :

```
CREATE TABLE image (
    nom             text,
    donnees         oid
);

SELECT lo_creat(-1);       -- renvoie l'OID du nouveau Large Object

SELECT lo_create(43213);   -- tente de créer le Large Object d'OID
 43213

SELECT lo_unlink(173454);  -- supprime le Large Object d'OID 173454

INSERT INTO image (nom, donnees)
    VALUES ('superbe image', lo_import('/etc/motd'));

INSERT INTO image (nom, donnees)  -- identique à ci-dessus, mais
 précise l'OID à utiliser
    VALUES ('superbe image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.donnees, '/tmp/motd') FROM image
    WHERE nom = 'superbe image';
```

Les fonctions `lo_import` et `lo_export` côté serveur se comportent considérablement différemment de leurs analogues côté client. Ces deux fonctions lisent et écrivent des fichiers dans le système de fichiers du serveur en utilisant les droits du propriétaire du serveur de base de données. Du coup, par défaut, leur utilisation est restreinte aux superutilisateurs PostgreSQL. Au contraire des fonctions côté serveur, les fonctions d'import et d'export côté client lisent et écrivent des fichiers dans le système de fichiers du client en utilisant les droits du programme client. Les fonctions côté client ne nécessitent pas de droits particuliers sur la base, sauf celui de lire et écrire le « Large Object » en question.

> **Attention**
>
> Il est possible de donner le droit d'utiliser les fonctions serveurs `lo_import` et `lo_export` à des utilisateurs standards avec l'instruction GRANT mais les implications sur la sécurité doivent être appréhendées avec attention. Un utilisateur mal intentionné avec de tels droits pourrait facilement les utiliser pour devenir superutilisateur (par exemple en réécrivant les fichiers de configuration du serveur) ou pourrait attaquer le reste du système de fichiers du serveur, sans avoir à se soucier d'obtenir les droits de superutilisateur de la base de données. *L'accès aux rôles ayant de tels droits doit de ce fait être gardé très précautionneusement, tout comme l'accès aux rôles superutilisateur.* Néanmoins, si l'utilisation des fonctions serveurs `lo_import` et `lo_export` sont nécessaires pour certaines tâches de routine, il est plus sûr d'utiliser un rôle avec ces droits qu'un rôle avec les droits complets du superutilisateur car cela aide à réduire le risque de dommages au niveau des erreurs accidentelles.

Les fonctionnalités de `lo_read` et `lo_write` sont aussi disponibles via des appels côté serveur mais les noms des fonctions diffèrent des interfaces côté client du fait qu'elles ne possèdent pas de tiret bas. Vous devez appeler ces fonctions avec `loread` et `lowrite`.

35.5. Programme d'exemple

L'Exemple 35.1 est un programme d'exemple qui montre une utilisation de l'interface des « Large Objects » avec la bibliothèque libpq. Des parties de ce programme disposent de commentaires au bénéfice des utilisateurs. Ce programme est aussi disponible dans la distribution des sources (`src/test/examples/testlo.c`).

Exemple 35.1. Exemple de programme sur les « Large Objects » avec libpq

```
/
*-------------------------------------------------------------------------
 *
 * testlo.c
 *    test utilisant des Large Objects avec libpq
 *
 * Portions Copyright (c) 1996-2022, PostgreSQL Global Development
 Group
 * Portions Copyright (c) 1994, Regents of the University of
 California
 *
 *
 * IDENTIFICATION
 *    src/test/examples/testlo.c
 *
 *-------------------------------------------------------------------------
 */
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE         1024

/*
 * importFile -
 *    importe le fichier "in_filename" dans la base de données
 *    en tant qu'objet "lobjOid"
 *
 */
static Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * ouvre le fichier à lire
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "cannot open unix file\"%s\"\n", filename);
    }

    /*
     * crée le Large Object
```

```
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "cannot create large object");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * lit le fichier Unix écrit dans le fichier inversion
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading \"%s\"",
 filename);
    }

    close(fd);
    lo_close(conn, lobj_fd);

    return lobjId;
}

static void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = '\0';
        fprintf(stderr, ">>> %s";, buf);
        nread += nbytes;
        if (nbytes <= 0)
            break;              /* no more data? */
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

static void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
```

```
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = '\0';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len -
 nwritten);
        nwritten += nbytes;
        if (nbytes <= 0)
        {
            fprintf(stderr, "\nWRITE FAILED!\n");
            break;
        }
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile -
 *    exporte le Large Object "lobjOid" dans le fichier
 * "out_filename"
 *
 */
static void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * ouvre l' « objet » large
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    /*
     * ouvre le fichier à écrire
     */
```

```
    fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "cannot open unix file\"%s\"",
                filename);
    }

    /*
     * lit à partir du fichier inversion et écrit dans le fichier
 Unix
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing \"%s\"",
                    filename);
        }
    }

    lo_close(conn, lobj_fd);
    close(fd);
}

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename
 out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * initialise la connexion
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);
```

```
    /* check to see that the backend connection was successfully
 made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* configure un chemin de recherche toujours sécurisé
     * pour que les utilisateurs avec de mauvaises intentions
     * ne puissent en prendre le contrôle.
     */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '',
 false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "begin");
    PQclear(res);
    printf("importing file \"%s\" ...\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
    if (lobjOid == 0)
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
    else
    {
        printf("\tas large object %u.\n", lobjOid);

        printf("picking out bytes 1000-2000 of the large object
\n");
        pickout(conn, lobjOid, 1000, 1000);

        printf("overwriting bytes 1000-2000 of the large object
 with X's\n");
        overwrite(conn, lobjOid, 1000, 1000);

        printf("exporting large object to file \"%s\" ...\n",
 out_filename);
/*      exportFile(conn, lobjOid, out_filename); */
        if (lo_export(conn, lobjOid, out_filename) < 0)
            fprintf(stderr, "%s\n", PQerrorMessage(conn));
    }

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    return 0;
}
```

Chapitre 36. ECPG -- SQL embarqué en C

Ce chapitre décrit le module de SQL embarqué pour PostgreSQL. Il a été écrit par Linus Tolke (<linus@epact.se>) et Michael Meskes (< meskes@postgresql.org>). Initialement, il a été écrit pour fonctionner avec le C. Il fonctionne aussi avec le C++, mais il ne reconnaît pas encore toutes les syntaxes du C++.

Ce document est assez incomplet. Mais comme l'interface est standardisée, des informations supplémentaires peuvent être trouvées dans beaucoup de documents sur le SQL.

36.1. Le Concept

Un programme SQL embarqué est composé de code écrit dans un langage de programmation ordinaire, dans notre cas le C, mélangé avec des commandes SQL dans des sections spécialement balisées. Pour compiler le programme, le code source (`*.pgc`) passe d'abord dans un préprocesseur pour SQL embarqué, qui le convertit en un programme C ordinaire (`*.c`), afin qu'il puisse ensuite être traité par un compilateur C. (Pour les détails sur la compilation et l'édition de lien dynamique voyez Section 36.10). Les applications ECPG converties appellent les fonctions de la librairie libpq au travers de la librairie SQL embarquée (ecpgli), et communique avec le serveur PostgreSQL au travers du protocole client-serveur normal.

Le SQL embarqué a des avantages par rapport aux autres méthodes de manipulation du SQL dans le code C. Premièrement, il s'occupe du laborieux passage d'information de et vers les variables de votre programme C. Deuxièmement, le code SQL du programme est vérifié à la compilation au niveau syntaxique. Troisièmement, le SQL embarqué en C est supporté par beaucoup d'autres bases de données SQL. L'implémentation PostgreSQL est conçue pour correspondre à ce standard autant que possible, et il est habituellement possible de porter du SQL embarqué d'autres bases SQL vers PostgreSQL assez simplement.

Comme déjà expliqué précédemment, les programmes écrits pour du SQL embarqué sont des programmes C normaux, avec du code spécifique inséré pour exécuter des opérations liées à la base de données. Ce code spécial est toujours de la forme :

```
EXEC SQL ...;
```

Ces ordres prennent, syntaxiquement, la place d'un ordre SQL. En fonction de l'ordre lui-même, ils peuvent apparaître au niveau global ou à l'intérieur d'une fonction.

Les ordres SQL embarqués suivent les règles habituelles de sensibilité à la casse du code SQL, et pas celles du langage C. De plus, elles permettent des commentaires imbriqués comme dans le standard SQL. Néanmoins, la partie C du programme suit le standard C de ne pas accepter des commentaires imbriqués. Les ordres SQL embarqués utilisent des règles SQL, et non pas des règles en C, pour analyser les chaînes placées entre guillemets et les identifiants.(Voir respectivement Section 4.1.2.1 et Section 4.1.1. Notez que ECPG assume que `standard_conforming_strings` vaut `on`.) Bien sûr, la partie de C du programme suit les règles C de mise entre guillemets.

Les sections suivantes expliquent tous les ordres SQL embarqués.

36.2. Gérer les Connexions à la Base de Données

Si des utilisateurs pour lesquels nous n'avons pas confiance ont accès à une base de données qui n'a pas adopté une méthode sécurisée d'usage des schemas, commencez chaque session en supprimant les schémas modifiables par tout le monde du paramètre `search_path`. Par exemple, ajoutez `options=-c search_path=` à *`options`* ou exécutez `EXEC SQL SELECT pg_catalog.set_config('search_path', '', false);` tout de suite après la connexion. Cette considération n'est pas spécifique à ECPG ; elle s'applique à chaque interface permettant d'exécuter des commandes SQL arbitraires.

Cette section explique comment ouvrir, fermer, et changer de connexion à la base.

36.2.1. Se Connecter au Serveur de Base de Données

On se connecte à la base de données avec l'ordre suivant:

```
EXEC SQL CONNECT TO cible [AS nom-connexion] [USER nom-
utilisateur];
```

La *`cible`* peut être spécifiée des façons suivantes:

- *`nomdb`*`[@`*`nomhôte`*`][:`*`port`*`]`
- `tcp:postgresql://`*`nomhôte`*`[:`*`port`*`][/`*`nomdb`*`][?`*`options`*`]`
- `unix:postgresql://nomhôte[:`*`port`*`][/`*`nomdb`*`][?`*`options`*`]`
- une chaine SQL littérale contenant une des formes précédentes
- une référence à une variable caractère contenant une des formes précédentes (voyez les exemples)
- `DEFAULT`

La cible de connexion `DEFAULT` initie une connexion à la base de données par défaut avec l'utilisateur par défaut. Il n'est pas nécessaire de préciser séparément un nom d'utilisateur ou un nom de connexion dans ce cas.

Si vous spécifiez la chaîne de connexion de façon littérale (c'est-à-dire, pas par une chaîne littérale ou une référence à une variable), alors les constituants de la chaîne suivent les règles de grammaire de SQL normal ; par exemple, le *`hostname`* doit être similaire à un ou plusieurs identifiants SQL séparés par des points, et ces identifiants seront insensibles à la casse à moins d'être entre guillemets. Les valeurs de n'importe quelles *`options`* doivent être des identifiants SQL, des entiers, ou des références de variables. Bien sûr, vous pouvez mettre presque n'importe quoi dans un identifiant SQL en le plaçant entre guillemets. En pratique, il y a probablement moins de risque d'erreur d'utiliser une chaîne littérale (entre simples guillemets) ou une référence de variable que d'écrire la chaîne de connexion directement.

Il y a aussi plusieurs façons de spécifier le nom de l'utilisateur :

- *`nomutilisateur`*
- *`nomutilisateur/motdepasse`*
- *`nomutilisateur`* `IDENTIFIED BY` *`motdepasse`*
- *`nomutilisateur`* `USING` *`motdepasse`*

Comme précédemment, les paramètres *nomutilisateur* et *motdepasse* peuvent être un identifiant SQL, une chaîne SQL littérale, ou une référence à une variable caractère.

Si la cible de connexion inclut des *options*, cela consiste en des spécifications *motclé=valeur* séparées par des esperluettes (&). Les mots clés autorisés sont les mêmes que ceux reconnus par libpq (voir Section 34.1.2). Les espaces sont ignorés avant tout *mot-clé* ou *valeur*, mais pas à l'intérieur ou après. Notez qu'il n'existe pas de moyens d'écrire & à l'intérieur d'une *valeur*.

Notez qu'en spécifiant une connexion socket (avec le préfixe `unix:`), le nom d'hôte doit être exactement `localhost`. Pour choisir un répertoire de socket autre que celui par défaut, écrivez le chemin complet du répertoire comme valeur de l'option `host` dans la section *options* de la cible.

Le *nom-connexion* est utilisé pour gérer plusieurs connexions dans un programme. Il peut être omis si le programme n'utilise qu'une connexion. La connexion la plus récemment ouverte devient la connexion courante, qui est utilisée par défaut quand un ordre SQL doit être exécuté (voyez plus bas dans ce chapitre).

Voici quelques exemples d'ordres `CONNECT`:

```
EXEC SQL CONNECT TO mabase@sql.mondomaine.com;

EXEC SQL CONNECT TO tcp:postgresql://sql.mondomaine.com/mabase AS
 maconnexion USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *cible = "mabase@sql.mondomaine.com";
const char *utilisateur = "john";
const char *motdepasse = "secret";
EXEC SQL END DECLARE SECTION;
 ...
EXEC SQL CONNECT TO :cible USER :utilisateur USING :motdepasse;
/* or EXEC SQL CONNECT TO :cible USER :utilisateur/:motdepasse; */
```

Le dernier exemple utilise la fonctionnalité dont on parlait précédemment sous le nom de références de variable. Vous verrez dans les sections finales comment des variables C peuvent être utilisées dans des ordres SQL quand vous les préfixez par deux-points.

Notez que le format de la cible de connexion n'est pas spécifié dans le standard SQL. Par conséquent si vous voulez développer des applications portables, vous pourriez vouloir utiliser quelque chose ressemblant au dernier exemple pour encapsuler la cible de connexion quelque part.

Si des utilisateurs non fiables ont accès à une base de données qui n'a pas adopté un modèle d'utilisation de schémas sécurisés, alors commencez chaque session en retirant les schémas accessibles à l'écriture publique du `search_path`. Par exemple, ajoutez `options=-c search_path=` aux *options*, ou lancez `EXEC SQL SELECT pg_catalog.set_config('search_path', '', false);` après la connexion. Cette considération n'est pas spécifique à ECPG ; elle s'applique à toute interface pour exécuter des commandes SQL arbitraires.

36.2.2. Choisir une connexion

Les ordres des programmes SQL embarqué sont par défaut exécutés dans la connexion courante, c'est à dire la plus récemment ouverte. Si une application a besoin de gérer plusieurs connexions, alors il y a trois façons de le gérer.

La première solution est de choisir explicitement une connexion pour chaque ordre SQL, par exemple:

```
EXEC SQL AT nom-connexion SELECT ...;
```

Cette option est particulièrement appropriée si l'application a besoin d'alterner les accès à plusieurs connexions.

Si votre application utilise plusieurs threads d'exécution, ils ne peuvent pas utiliser une connexion simultanément. Vous devez soit contrôler explicitement l'accès à la connexion (en utilisant des mutexes), ou utiliser une connexion pour chaque thread.

La seconde option est d'exécuter un ordre pour changer de connexion courante. Cet ordre est:

```
EXEC SQL SET CONNECTION nom-connexion;
```

Cette option est particulièrement pratique si de nombreux ordres doivent être exécutés sur la même connexion.

Voici un programme exemple qui gère plusieurs connexions à base de données:

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
    char nomdb[1024];
EXEC SQL END DECLARE SECTION;

int
main()
{
    EXEC SQL CONNECT TO basetest1 AS con1 USER utilisateurtest;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;
    EXEC SQL CONNECT TO basetest2 AS con2 USER utilisateurtest;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;
    EXEC SQL CONNECT TO basetest3 AS con3 USER utilisateurtest;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;

    /* Cette requête serait exécuté dans la dernière base ouverte
 "basetest3". */
    EXEC SQL SELECT current_database() INTO :nomdb;
    printf("courante=%s (devrait être basetest3)\n", nomdb);

    /* Utiliser "AT" pour exécuter une requête dans "basetest2" */
    EXEC SQL AT con2 SELECT current_database() INTO :nomdb;
    printf("courante=%s (devrait être basetest2)\n", nomdb);

    /* Basculer la connexion courante à "basetest1". */
    EXEC SQL SET CONNECTION con1;

    EXEC SQL SELECT current_database() INTO :nomdb;
    printf("courante=%s (devrait être basetest1)\n", nomdb);

    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

Cet exemple devrait produire cette sortie :

```
courante=basetest3 (devrait être basetest3)
courante=basetest2 (devrait être basetest2)
courante=basetest1 (devrait être basetest1)
```

La troisième option est de déclarer un identifiant SQL lié à la connexion, par exemple :

```
EXEC SQL AT connection-name DECLARE statement-name STATEMENT;
EXEC SQL PREPARE statement-name FROM :dyn-string;
```

Une fois que vous avez lié l'identifiant SQL à la connexion, vous exécutez le SQL dynamique sans clause AT. Notez que cette option se comporte comme les directives préprocesseurs, ainsi le lien n'est actif que dans le fichier.

Voici un exemple de programme utilisant cette option :

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
char dbname[128];
char *dyn_sql = "SELECT current_database()";
EXEC SQL END DECLARE SECTION;

int main(){
  EXEC SQL CONNECT TO postgres AS con1;
  EXEC SQL CONNECT TO testdb AS con2;
  EXEC SQL AT con1 DECLARE stmt STATEMENT;
  EXEC SQL PREPARE stmt FROM :dyn_sql;
  EXEC SQL EXECUTE stmt INTO :dbname;
  printf("%s\n", dbname);

  EXEC SQL DISCONNECT ALL;
  return 0;
}
```

Cet exemple produira cette sortie, même si la connexion par défaut est testdb :

```
postgres
```

36.2.3. Fermer une Connexion

Pour fermer une connexion, utilisez l'ordre suivant :

```
EXEC SQL DISCONNECT [connexion];
```

La *connexion* peut être spécifiée des façons suivantes:

- *nom-connexion*
- CURRENT
- ALL

Si aucun nom de connexion n'est spécifié, la connexion courante est fermée.

C'est une bonne pratique qu'une application ferme toujours explicitement toute connexion qu'elle a ouverte.

36.3. Exécuter des Commandes SQL

Toute commande SQL peut être exécutée à l'intérieur d'une application SQL embarquée. Voici quelques exemples montrant comment le faire.

36.3.1. Exécuter des Ordres SQL

Créer une table :

```
EXEC SQL CREATE TABLE truc (nombre integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON truc(nombre);
EXEC SQL COMMIT;
```

Insérer des lignes :

```
EXEC SQL INSERT INTO truc (nombre, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Supprimer des lignes :

```
EXEC SQL DELETE FROM truc WHERE nombre = 9999;
EXEC SQL COMMIT;
```

Mettre à jour des lignes :

```
EXEC SQL UPDATE truc
    SET ascii = 'trucmachin'
    WHERE nombre = 9999;
EXEC SQL COMMIT;
```

Les ordres `SELECT` qui retournent un seul enregistrement peuvent aussi être exécutés en utilisant `EXEC SQL` directement. Pour traiter des jeux de résultats de plusieurs enregistrements, une application doit utiliser un curseur; voyez Section 36.3.2 plus bas. (Exceptionnellement, une application peut récupérer plusieurs enregistrements en une seule fois dans une variable hôte tableau; voyez Section 36.4.4.3.1.)

Select mono-ligne :

```
EXEC SQL SELECT truc INTO :trucmachin FROM table1 WHERE ascii =
 'doodad';
```

De même, un paramètre de configuration peut être récupéré avec la commande `SHOW` :

```
EXEC SQL SHOW search_path INTO :var;
```

Les tokens de la forme `:quelquechose` sont des *variables hôtes*, c'est-à-dire qu'ils font référence à des variables dans le programme C. Elles sont expliquées dans Section 36.4.

36.3.2. Utiliser des Curseurs

Pour récupérer un résultat contenant plusieurs enregistrements, une application doit déclarer un curseur et récupérer chaque enregistrement de ce curseur. Les étapes pour déclarer un curseur sont les suivantes: déclarer le curseur, l'ouvrir, récupérer un enregistrement à partir du curseur, répéter, et finalement le fermer.

Select avec des curseurs :

```
EXEC SQL DECLARE truc_machin CURSOR FOR
    SELECT nombre, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN truc_machin;
EXEC SQL FETCH truc_machin INTO :TrucMachin, MachinChouette;
...
EXEC SQL CLOSE truc_machin;
EXEC SQL COMMIT;
```

Pour plus de détails sur la déclaration d'un curseur, voir DECLARE ; Pour plus de détails sur la récupération des enregistrements depuis un curseur, voir FETCH.

> **Note**
>
> La commande `DECLARE` ne déclenche pas réellement l'envoi d'un ordre au serveur PostgreSQL. Le curseur est ouvert dans le processus serveur (en utilisant la commande `DECLARE`) au moment où la commande `OPEN` est exécutée.

36.3.3. Gérer les Transactions

Dans le mode par défaut, les ordres SQL ne sont validés que quand `EXEC SQL COMMIT` est envoyée. L'interface SQL embarquée supporte aussi l'auto-commit des transactions (de façon similaire au comportement de psql) via l'option de ligne de commande `-t` d'`ecpg` (voyez ecpg) ou par l'ordre `EXEC SQL SET AUTOCOMMIT TO ON`. En mode auto-commit, chaque commande est validée automatiquement sauf si elle se trouve dans un bloc explicite de transaction. Ce mode peut être explicitement désactivé en utilisant `EXEC SQL SET AUTOCOMMIT TO OFF`.

Les commandes suivantes de gestion de transaction sont disponibles :

`EXEC SQL COMMIT`

Valider une transaction en cours.

`EXEC SQL ROLLBACK`

Annuler une transaction en cours.

`EXEC SQL PREPARE TRANSACTION` *`transaction_id`*

Préparer la transaction courante pour une transaction en deux phases.

`EXEC SQL COMMIT PREPARED` *`transaction_id`*

Valide une transaction qui est dans un état préparé.

`EXEC SQL ROLLBACK PREPARED` *`transaction_id`*

Annule une transaction qui est dans un état préparé.

`EXEC SQL SET AUTOCOMMIT TO ON`

Activer le mode auto-commit.

`EXEC SQL SET AUTOCOMMIT TO OFF`

Désactiver le mode auto-commit. C'est la valeur par défaut.

36.3.4. Requêtes préparées

Quand les valeurs à passer à un ordre SQL ne sont pas connues au moment de la compilation, ou que le même ordre SQL va être utilisé de nombreuses fois, les requêtes préparées peuvent être utiles.

L'ordre est préparé en utilisant la commande `PREPARE`. Pour les valeurs qui ne sont pas encore connues, utilisez le substitut « ? »:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database
 WHERE oid = ?";
```

Si un ordre retourne une seule ligne, l'application peut appeler `EXECUTE` après `PREPARE` pour exécuter l'ordre, en fournissant les vraies valeurs pour les substituts avec une clause `USING` :

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
```

Si un ordre retourne plusieurs enregistrements, l'application peut utiliser un curseur déclarés en se servant d'une requête préparée. Pour lier les paramètres d'entrée, le curseur doit être ouvert avec une clause `USING` :

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname FROM pg_database
 WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;

/* Quand la fin du jeu de résultats est atteinte, sortir de la
 boucle while */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
    ...
}
EXEC SQL CLOSE foo_bar;
```

Quand vous n'avez plus besoin de la requête préparée, vous devriez la désallouer :

```
EXEC SQL DEALLOCATE PREPARE nom;
```

Pour plus de détails sur PREPARE, voyez PREPARE. Voyez aussi Section 36.5 pour plus de détails à propos de l'utilisation des substituts et des paramètres d'entrée.

36.4. Utiliser des Variables Hôtes

Dans Section 36.3 vous avez vu comment exécuter des ordres SQL dans un programme SQL embarqué. Certains de ces ordres n'ont utilisé que des valeurs constantes et ne fournissaient pas de moyen pour insérer des valeurs fournies par l'utilisateur dans des ordres ou pour permettre au programme de traiter les valeurs retournées par la requête. Ces types d'ordres ne sont pas très utiles dans des applications réelles. Cette section explique en détail comment faire passer des données entre votre programme en C et les ordres SQL embarqués en utilisant un simple mécanisme appelé *variables hôtes*. Dans un programme SQL embarqué nous considérons que les ordres SQL sont des *invités* dans le code du programme C qui est le *langage hôte*. Par conséquent, les variables du programme C sont appelées *variables hôtes*.

Une autre façon d'échanger des valeurs entre les serveurs PostgreSQL et les applications ECPG est l'utilisation de descripteurs SQL, décrits dans Section 36.7.

36.4.1. Aperçu

Passer des données entre le programme en C et les ordres SQL est particulièrement simple en SQL embarqué. Plutôt que d'avoir un programme qui donne des données dans un ordre SQL, ce qui entraîne des complications variées, comme protéger correctement la valeur, vous pouvez simplement écrire le nom d'une variable C dans un ordre SQL, préfixée par un deux-points. Par exemple :

```
EXEC SQL INSERT INTO unetable VALUES (:v1, 'foo', :v2);
```

Cet ordre fait référence à deux variables C appelées `v1` et `v2` et utilise aussi une chaîne SQL classique, pour montrer que vous n'êtes pas obligé de vous cantonner à un type de données ou à l'autre.

Cette façon d'insérer des variables C dans des ordres SQL fonctionne partout où une expression de valeur est attendue dans un ordre SQL.

36.4.2. Sections Declare

Pour passer des données du programme à la base, par exemple comme paramètres d'une requête, ou pour passer des données de la base vers le programme, les variables C qui sont prévues pour contenir ces données doivent être déclarées dans des sections spécialement identifiées, afin que le préprocesseur SQL embarqué puisse s'en rendre compte.

Cette section commence par :

```
EXEC SQL BEGIN DECLARE SECTION;
```

et se termine par:

```
EXEC SQL END DECLARE SECTION;
```

Entre ces lignes, il doit y avoir des déclarations de variables C normales, comme :

```
int   x = 4;
char  foo[16], bar[16];
```

Comme vous pouvez le voir, vous pouvez optionnellement assigner une valeur initiale à une variable. La portée de la variable est déterminée par l'endroit où se trouve la section de déclaration dans le programme. Vous pouvez aussi déclarer des variables avec la syntaxe suivante, qui crée une section declare implicite :

```
EXEC SQL int i = 4;
```

Vous pouvez avoir autant de sections de déclaration que vous voulez dans un programme.

Ces déclarations sont aussi envoyées dans le fichier produit comme des variables C normales, il n'est donc pas nécessaire de les déclarer une seconde fois. Les variables qui n'ont pas besoin d'être utilisées dans des commandes SQL peuvent être déclarées normalement à l'extérieur de ces sections spéciales.

La définition d'une structure ou d'un union doit aussi être présente dans une section `DECLARE`. Sinon, le préprocesseur ne peut pas traiter ces types, puisuq'il n'en connait pas la définition.

36.4.3. Récupérer des Résultats de Requêtes

Maintenant, vous devriez être capable de passer des données générées par votre programme dans une commande SQL. Mais comment récupérer les résultats d'une requête ? À cet effet, le SQL embarqué fournit certaines variantes spéciales de commandes `SELECT` et `FETCH` habituelles. Ces commandes ont une clause spéciale `INTO` qui spécifie dans quelles variables hôtes les valeurs récupérées doivent être stockées. `SELECT` est utilisé pour une requête qui ne retourne qu'un seul enregistrement, et `FETCH` est utilisé pour une requête qui retourne plusieurs enregistrement, en utilisant un curseur.

Voici un exemple :

```
/*
 * Avec cette table:
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

La clause `INTO` apparaît entre la liste de sélection et la clause `FROM`. Le nombre d'éléments dans la liste SELECT et dans la liste après `INTO` (aussi appelée la liste cible) doivent être égaux.

Voici un exemple utilisant la commande `FETCH` :

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...
```

```
EXEC SQL DECLARE truc CURSOR FOR SELECT a, b FROM test;

 ...

do
{
    ...
    EXEC SQL FETCH NEXT FROM truc INTO :v1, :v2;
    ...
} while (...);
```

Ici, la clause `INTO` apparaît après toutes les clauses normales.

36.4.4. Correspondance de Type

Quand les applications ECPG échangent des valeurs entre le serveur PostgreSQL et l'application C, comme quand elles récupèrent des résultats de requête venant du serveur, ou qu'elles exécutent des ordres SQL avec des paramètres d'entrée, les valeurs doivent être converties entre les types de données PostgreSQL et les types du language hôte (ceux du langage C). Une des fonctionnalités les plus importantes d'ECPG est qu'il s'occupe de cela automatiquement dans la plupart des cas.

De ce point de vue, il y a deux sortes de types de données : des types de données PostgreSQL simples, comme des `integer` et `text`, qui peuvent être lus et écrits directement par l'application. Les autres types PostgreSQL, comme `timestamp` ou `numeric` ne peuvent être accédés qu'à travers des fonctions spéciales de librairie; voyez Section 36.4.4.2.

Tableau 36.1 montre quels types de données de PostgreSQL correspondent à quels types C. Quand vous voulez envoyer ou recevoir une valeur d'un type PostgreSQL donné, vous devriez déclarer une variable C du type C correspondant dans la section declare.

Tableau 36.1. Correspondance Entre les Types PostgreSQL et les Types de Variables C

type de données PostgreSQL	type de variable hôte
`smallint`	`short`
`integer`	`int`
`bigint`	`long long int`
`decimal`	`decimal`[a]
`numeric`	`numeric`[b]
`real`	`float`
`double precision`	`double`
`smallserial`	`short`
`serial`	`int`
`bigserial`	`long long int`
`oid`	`unsigned int`
`character(`*n*`)`, `varchar(`*n*`)`, `text`	`char[`*n*`+1]`, `VARCHAR[`*n*`+1]`
`name`	`char[NAMEDATALEN]`
`timestamp`	`timestamp`[c]
`interval`	`interval`[d]
`date`	`date`[e]

type de données PostgreSQL	type de variable hôte
`boolean`	`bool`[f]
`bytea`	`char *`, `bytea[`*n*`]`

[a]Ce type ne peut être accédé qu'à travers des fonctions spéciales de librairie. Voyez Section 36.4.4.2.
[b]Ce type ne peut être accédé qu'à travers des fonctions spéciales de librairie. Voyez Section 36.4.4.2.
[c]Ce type ne peut être accédé qu'à travers des fonctions spéciales de librairie. Voyez Section 36.4.4.2.
[d]Ce type ne peut être accédé qu'à travers des fonctions spéciales de librairie. Voyez Section 36.4.4.2.
[e]Ce type ne peut être accédé qu'à travers des fonctions spéciales de librairie. Voyez Section 36.4.4.2.
[f]déclaré dans `ecpglib.h` si non natif

36.4.4.1. Manipuler des Chaînes de Caractères

Pour manipuler des types chaînes de caractères SQL, comme `varchar` et `text`, il y a deux façons de déclarer les variables hôtes.

Une façon est d'utiliser `char[]`, un tableau de `char`, qui est la façon la plus habituelle de gérer des données texte en C.

```
EXEC SQL BEGIN DECLARE SECTION;
    char str[50];
EXEC SQL END DECLARE SECTION;
```

Notez que vous devez gérer la longueur vous-même. Si vous utilisez cette variable he comme variable cible d'une requête qui retourne une chaîne de plus de 49 caractères, un débordement de tampon se produira.

L'autre façon est d'utiliser le type VARCHAR, qui est un type spécial fourni par ECPG. La définition d'un tableau de type VARCHAR est convertie dans un `struct` nommé pour chaque variable. Une déclaration comme :

```
VARCHAR var[180];
```

est convertie en:

```
struct varchar_var { int len; char arr[180]; } var;
```

Le membre `arr` contient la chaîne terminée par un octet à zéro. Par conséquent, la variable hôte doit être déclarée avec la longueur incluant le terminateur de chaîne. Le membre `len` stocke la longueur de la chaîne stockée dans `arr` sans l'octet zéro final. Quand une variable hôte est utilisé comme entrée pour une requête, si `strlen` et `len` sont différents, le plus petit est utilisé.

VARCHAR peut être écrit en majuscule ou en minuscule, mais pas dans un mélange des deux.

Les variables hôtes `char` et VARCHAR peuvent aussi contenir des valeurs d'autres types SQL, qui seront stockés dans leur forme chaîne.

36.4.4.2. Accéder à des Types de Données Spéciaux

ECPG contient des types spéciaux qui vous aident interagir facilement avec des types de données spéciaux du serveur PostgreSQL. En particulier, sont supportés les types `numeric`, `decimal`, `date`, `timestamp`, et `interval`. Ces types de données ne peuvent pas être mis de façon utile en correspondance avec des types primitifs du langage hôtes (tels que `int`, `long long int`, ou `char[]`), parce qu'ils ont une structure interne complexe. Les applications manipulent ces types en

déclarant des variables hôtes dans des types spéciaux et en y accédant avec des fonctions de la librairie pgtypes. La librairie pgtypes, décrite en détail dans Section 36.6 contient des fonctions de base pour traiter ces types, afin que vous n'ayez pas besoin d'envoyer une requête au serveur SQL juste pour additionner un interval à un timestamp par exemple.

Les sous-sections suivantes décrivent ces types de données spéciaux. Pour plus de détails à propos des fonctions de librairie pgtype, voyez Section 36.6.

36.4.4.2.1. timestamp, date

Voici une méthode pour manipuler des variables `timestamp` dans l'application hôte ECPG.

Tout d'abord, le programme doit inclure le fichier d'en-tête pour le type `timestamp` :

```
#include <pgtypes_timestamp.h>
```

Puis, déclarez une variable hôte comme type `timestamp` dans la section declare :

```
EXEC SQL BEGIN DECLARE SECTION;
timestamp ts;
EXEC SQL END DECLARE SECTION;
```

Et après avoir lu une valeur dans la variable hôte, traitez la en utilisant les fonctions de la librairie pgtypes. Dans l'exemple qui suit, la valeur `timestamp` est convertie sous forme texte (ASCII) avec la fonction `PGTYPEStimestamp_to_asc()` :

```
EXEC SQL SELECT now()::timestamp INTO :ts;

printf("ts = %s\n", PGTYPEStimestamp_to_asc(ts));
```

Cet exemple affichere des résultats de ce type:

```
ts = 2010-06-27 18:03:56.949343
```

Par ailleurs, le type DATE peut être manipulé de la même façon. Le programme doit inclure `pgtypes_date.h`, déclarer une variable hôte comme étant du type date et convertir une valeur DATE dans sa forme texte en utilisant la fonction `PGTYPESdate_to_asc()`. Pour plus de détails sur les fonctions de la librairie pgtypes, voyez Section 36.6.

36.4.4.2.2. interval

La manipulation du type `interval` est aussi similaire aux types `timestamp` et `date`. Il est nécessaire, par contre, d'allouer de la mémoire pour une valeur de type `interval` de façon explicite. Ou dit autrement, l'espace mémoire pour la variable doit être allouée du tas, et non de la pile.

Voici un programme de démonstration :

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_interval.h>
```

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    interval *in;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;

    in = PGTYPESinterval_new();
    EXEC SQL SELECT '1 min'::interval INTO :in;
    printf("interval = %s\n", PGTYPESinterval_to_asc(in));
    PGTYPESinterval_free(in);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

36.4.4.2.3. numeric, decimal

La manipulation des types `numeric` et `decimal` est similaire au type `interval`: elle requiert de définir d'un pointeur, d'allouer de la mémoire sur le tas, et d'accéder la variable au mouyen des fonctions de librairie pgtypes. Pour plus de détails sur les fonctions de la librairie pgtypes, voyez Section 36.6.

Aucune fonction n'est fournie spécifiquement pour le type `decimal`. Une application doit le convertir vers une variable `numeric` en utilisant une fonction de la librairie pgtypes pour pouvoir le traiter.

Voici un programme montrant la manipulation des variables de type `numeric` et `decimal`.

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_numeric.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    numeric *num;
    numeric *num2;
    decimal *dec;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;

    num = PGTYPESnumeric_new();
    dec = PGTYPESdecimal_new();

    EXEC SQL SELECT 12.345::numeric(4,2), 23.456::decimal(4,2)
 INTO :num, :dec;
```

```
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 0));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 1));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 2));

    /* Convertir le decimal en numeric pour montrer une valeur
 décimale. */
    num2 = PGTYPESnumeric_new();
    PGTYPESnumeric_from_decimal(dec, num2);

    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 0));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 1));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 2));

    PGTYPESnumeric_free(num2);
    PGTYPESdecimal_free(dec);
    PGTYPESnumeric_free(num);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

36.4.4.2.4. bytea

La gestion du type `bytea` est aussi similaire au type VARCHAR. La définition d'un tableau de type `bytea` est convertie en une structure nommée pour chaque variable. Une déclaration comme :

```
bytea var[180];
```

est traduit en :

```
struct bytea_var { int len; char arr[180]; } var;
```

Le membre `arr` accueille des données au format binaire. Il peut même gérer `'\0'` comme faisant partie des données, contrairement à VARCHAR. La donnée est converti de/vers le format hexadécimal et envoyé/reçu par ecpglib.

> **Note**
>
> Une variable `bytea` peut être utilisé uniquement quand bytea_output est configuré à `hex`.

36.4.4.3. Variables Hôtes avec des Types Non-Primitifs

Vous pouvez aussi utiliser des tableaux, typedefs, structs et pointeurs comme variables hôtes.

36.4.4.3.1. Arrays

Il y a deux cas d'utilisations pour des tableaux comme variables hôtes. Le premier est une façon de stocker des chaînes de texte dans des `char[]` ou VARCHAR[], comme expliqué Section 36.4.4.1. Le second cas d'utilisation est de récupérer plusieurs enregistrements d'une requête sans utiliser de curseur. Sans un tableau, pour traiter le résultat d'une requête de plusieurs lignes, il est nécessaire

d'utiliser un curseur et la commande FETCH. Mais avec une variable hôte de type variable, plusieurs enregistrements peuvent être récupérés en une seule fois. La longueur du tableau doit être définie pour pouvoir recevoir tous les enregistrements d'un coup, sans quoi un buffer overflow se produira probablement.

Les exemples suivants parcourent la table système pg_database et montrent tous les OIDs et noms des bases de données disponibles :

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int dbid[8];
    char dbname[8][16];
    int i;
EXEC SQL END DECLARE SECTION;

    memset(dbname, 0, sizeof(char)* 16 * 8);
    memset(dbid, 0, sizeof(int) * 8);

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;

    /* Récupérer plusieurs enregistrements dans des tableaux d'un
 coup. */
    EXEC SQL SELECT oid,datname INTO :dbid, :dbname FROM
 pg_database;

    for (i = 0; i < 8; i++)
        printf("oid=%d, dbname=%s\n", dbid[i], dbname[i]);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

Cet exemple affiche le résultat suivant. (Les valeurs exactes dépendent de votre environnement.)

```
oid=1, dbname=template1
oid=11510, dbname=template0
oid=11511, dbname=postgres
oid=313780, dbname=testdb
oid=0, dbname=
oid=0, dbname=
oid=0, dbname=
```

36.4.4.3.2. Structures

Une structure dont les noms des membres correspondent aux noms de colonnes du résultat d'une requête peut être utilisée pour récupérer plusieurs colonnes d'un coup. La structure permet de gérer plusieurs valeurs de colonnes dans une seule variable hôte.

L'exemple suivant récupère les OIDs, noms, et tailles des bases de données disponibles à partir de la table système pg_database, et en utilisant la fonction pg_database_size(). Dans cet

exemple, une variable structure `dbinfo_t` avec des membres dont les noms correspondent à chaque colonnes du résultat du `SELECT` est utilisée pour récupérer une ligne de résultat sans avoir besoin de mettre plusieurs variables hôtes dans l'ordre `FETCH`.

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
       long long int size;
    } dbinfo_t;

    dbinfo_t dbval;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname,
 pg_database_size(oid) AS size FROM pg_database;
    EXEC SQL OPEN cur1;

    /* quand la fin du jeu de données est atteint, sortir de la
 boucle while */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* Récupérer plusieurs colonnes dans une structure. */
        EXEC SQL FETCH FROM cur1 INTO :dbval;

        /* Afficher les membres de la structure. */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid,
 dbval.datname, dbval.size);
    }

    EXEC SQL CLOSE cur1;
```

Cet exemple montre le résultat suivant. (Les valeurs exactes dépendent du contexte.)

```
oid=1, datname=template1, size=4324580
oid=11510, datname=template0, size=4243460
oid=11511, datname=postgres, size=4324580
oid=313780, datname=testdb, size=8183012
```

Les variables hôtes structures « absorbent » autant de colonnes que la structure a de champs. Des colonnes additionnelles peuvent être assignées à d'autres variables hôtes. Par exemple, le programme ci-dessus pourrait être restructuré comme ceci, avec la variable `size` hors de la structure :

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
    } dbinfo_t;
```

```
    dbinfo_t dbval;
    long long int size;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname,
 pg_database_size(oid) AS size FROM pg_database;
    EXEC SQL OPEN cur1;

    /* quand la fin du jeu de données est atteint, sortir de la
 boucle while */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* Récupérer plusieurs colonnes dans une structure. */
        EXEC SQL FETCH FROM cur1 INTO :dbval, :size;

        /* Afficher les membres de la structure. */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid,
 dbval.datname, size);
    }

    EXEC SQL CLOSE cur1;
```

36.4.4.3.3. Typedefs

Utilisez le mot clé `typedef` pour faire correspondre de nouveaux types aux types existants.

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

Notez que vous pourriez aussi utiliser:

```
EXEC SQL TYPE serial_t IS long;
```

Cette déclaration n'a pas besoin de faire partie d'une section declare.

36.4.4.3.4. Pointeurs

Vous pouvez déclarer des pointeurs vers les types les plus communs. Notez toutefois que vous ne pouvez pas utiliser des pointeurs comme variables cibles de requêtes sans auto-allocation. Voyez Section 36.7 pour plus d'information sur l'auto-allocation.

```
EXEC SQL BEGIN DECLARE SECTION;
    int   *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;
```

36.4.5. Manipuler des Types de Données SQL Non-Primitives

Cette section contient des informations sur comment manipuler des types non-scalaires et des types de données définies au niveau SQL par l'utilisateur dans des applications ECPG. Notez que c'est distinct de la manipulation des variables hôtes des types non-primitifs, décrits dans la section précédente.

36.4.5.1. Tableaux

Les tableaux SQL multi-dimensionnels ne sont pas directement supportés dans ECPG. Les tableaux SQL à une dimension peuvent être placés dans des variables hôtes de type tableau C et vice-versa. Néanmoins, lors de la création d'une instruction, ecpg ne connaît pas le type des colonnes, donc il ne peut pas vérifier si un tableau C est à placer dans un tableau SQL correspondant. Lors du traitement de la sortie d'une requête SQL, ecpg a suffisamment d'informations et, de ce fait, vérifie si les deux sont des tableaux.

Si une requête accède aux *éléments* d'un tableau séparément, cela évite l'utilisation des tableaux dans ECPG. Dans ce cas, une variable hôte avec un type qui peut être mis en correspondance avec le type de l'élément devrait être utilisé. Par exemple, si le type d'une colonne est un tableau d'`integer`, une variable hôte de type `int` peut être utilisée. Par ailleurs, si le type de l'élément est `varchar`, ou `text`, une variable hôte de type `char[]` ou `VARCHAR[]` peut être utilisée.

Voici un exemple. Prenez la table suivante :

```
CREATE TABLE t3 (
    ii integer[]
);

testdb=> SELECT * FROM t3;
     ii
-------------
 {1,2,3,4,5}
(1 row)
```

Le programme de démonstration suivant récupère le 4ème élément du tableau et le stocke dans une variable hôte de type `int` :

```
EXEC SQL BEGIN DECLARE SECTION;
int ii;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii ;
    printf("ii=%d\n", ii);
}
```

```
EXEC SQL CLOSE cur1;
```

Cet exemple affiche le résultat suivant :

```
ii=4
```

Pour mettre en correspondance de multiples éléments de tableaux avec les multiples éléments d'une variable hôte tableau, chaque élément du tableau doit être géré séparément, par exemple :

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4]
 FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1
 INTO :ii_a[0], :ii_a[1], :ii_a[2], :ii_a[3];
    ...
}
```

Notez à nouveau que

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* FAUX */
    EXEC SQL FETCH FROM cur1 INTO :ii_a;
    ...
}
```

ne fonctionnerait pas correctement dans ce cas, parce que vous ne pouvez pas mettre en correspondance une colonne de type tableau et une variable hôte de type tableau directement.

Un autre contournement possible est de stocker les tableaux dans leur forme de représentation texte dans des variables hôtes de type `char[]` ou `VARCHAR[]`. Pour plus de détails sur cette représentation, voyez Section 8.15.2. Notez que cela implique que le tableau ne peut pas être accédé naturellement comme un tableau dans le programme hôte (sans traitement supplémentaire qui transforme la représentation texte).

36.4.5.2. Types Composite

Les types composite ne sont pas directement supportés dans ECPG, mais un contournement simple est possible. Les contournements disponibles sont similaires à ceux décrits pour les tableaux ci-dessus: soit accéder à chaque attribut séparément, ou utiliser la représentation externe en mode chaîne de caractères.

Pour les exemples suivants, soient les types et tables suivants :

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'PostgreSQL') );
```

La solution la plus évidente est d'accéder à chaque attribut séparément. Le programme suivant récupère les données de la table exemple en sélectionnant chaque attribut du type `comp_t` séparément :

```
EXEC SQL BEGIN DECLARE SECTION;
int intval;
varchar textval[33];
EXEC SQL END DECLARE SECTION;

/* Mettre chaque élément de la colonne de type composite dans la
 liste SELECT. */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval,
 (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Récupérer chaque élément du type de colonne composite dans
 des variables hôtes. */
    EXEC SQL FETCH FROM cur1 INTO :intval, :textval;

    printf("intval=%d, textval=%s\n", intval, textval.arr);
}

EXEC SQL CLOSE cur1;
```

Pour améliorer cet exemple, les variables hôtes qui vont stocker les valeurs dans la commande `FETCH` peuvent être rassemblées sous forme de structure, voyez Section 36.4.4.3.2. Pour passer à la structure, l'exemple peut-être modifié comme ci dessous. Les deux variables hôtes, `intval` et `textval`, deviennent membres de `comp_t`, et la structure est spécifiée dans la commande `FETCH`.

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
{
    int intval;
    varchar textval[33];
} comp_t;

comp_t compval;
EXEC SQL END DECLARE SECTION;
```

```
/* Mettre chaque élément de la colonne de type composite dans la
 liste SELECT. */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval,
 (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Mettre toutes les valeurs de la liste SELECT dans une
 structure. */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval,
 compval.textval.arr);
}

EXEC SQL CLOSE cur1;
```

Bien qu'une structure soit utilisée dans la commande `FETCH`, les noms d'attributs dans la clause `SELECT` sont spécifiés un par un. Cela peut être amélioré en utilisant un `*` pour demander tous les attributs de la valeur de type composite.

```
...
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).* FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Mettre toutes les valeurs de la liste SELECT dans une
 structure. */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval,
 compval.textval.arr);
}
...
```

De cette façon, les types composites peuvent être mis en correspondance avec des structures de façon quasi transparentes, alors qu'ECPG ne comprend pas lui-même le type composite.

Et pour finir, il est aussi possible de stocker les valeurs de type composite dans leur représentation externe de type chaîne dans des variables hôtes de type `char[]` ou `VARCHAR[]`. Mais de cette façon, il n'est pas facilement possible d'accéder aux champs de la valeur dans le programme hôte.

36.4.5.3. Types de Base Définis par l'Utilisateur

Les nouveaux types de base définis par l'utilisateur ne sont pas directement supportés par ECPG. Vous pouvez utiliser les représentations externes de type chaîne et les variables hôtes de type `char[]` ou `VARCHAR[]`, et cette solution est en fait appropriée et suffisante pour de nombreux types.

Voici un exemple utilisant le type de données `complex` de l'exemple tiré de Section 38.13. La représentation externe sous forme de chaîne de ce type est `(%lf,%lf)`, qui est définie dans les

fonctions `complex_in()` et `complex_out()`. L'exemple suivant insère les valeurs de type complexe `(1,1)` et `(3,3)` dans les colonnes `a` et `b`, et les sélectionne à partir de la table après cela.

```
EXEC SQL BEGIN DECLARE SECTION;
    varchar a[64];
    varchar b[64];
EXEC SQL END DECLARE SECTION;

    EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)');

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT a, b FROM test_complex;
    EXEC SQL OPEN cur1;

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH FROM cur1 INTO :a, :b;
        printf("a=%s, b=%s\n", a.arr, b.arr);
    }

    EXEC SQL CLOSE cur1;
```

Cet exemple affiche le résultat suivant :

```
a=(1,1), b=(3,3)
```

Un autre contournement est d'éviter l'utilisation directe des types définis par l'utilisateur dans ECPG et à la place créer une fonction ou un cast qui convertit entre le type défini par l'utilisateur et un type primitif que ECPG peut traiter. Notez, toutefois, que les conversions de types, particulièrement les implicites, ne devraient être introduits dans le système de typage qu'avec la plus grande prudence.

Par exemple,

```
CREATE FUNCTION create_complex(r double, i double) RETURNS complex
LANGUAGE SQL
IMMUTABLE
AS $$ SELECT $1 * complex '(1,0')' + $2 * complex '(0,1)' $$;
```

Après cette définition, ce qui suit

```
EXEC SQL BEGIN DECLARE SECTION;
double a, b, c, d;
EXEC SQL END DECLARE SECTION;

a = 1;
b = 2;
c = 3;
d = 4;

EXEC SQL INSERT INTO test_complex VALUES (create_complex(:a, :b),
 create_complex(:c, :d));
```

a le même effet que

```
EXEC SQL INSERT INTO test_complex VALUES ('(1,2)', '(3,4)');
```

36.4.6. Indicateurs

Les exemples précédents ne gèrent pas les valeurs nulles. En fait, les exemples de récupération de données remonteront une erreur si ils récupèrent une valeur nulle de la base. Pour être capable de passer des valeurs nulles à la base ou d'un récupérer, vous devez rajouter une seconde spécification de variable hôte à chaque variable hôte contenant des données. Cette seconde variable est appelée l'*indicateur* et contient un drapeau qui indique si le datum est null, dans quel cas la valeur de la vraie variable hôte est ignorée. Voici un exemple qui gère la récupération de valeurs nulles correctement :

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:

 ...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

La variable indicateur `val_ind` sera zéro si la valeur n'était pas nulle, et sera négative si la valeur était nulle. (Voir Section 36.16 pour activer un comportement spécifique à Oracle.)

L'indicateur a une autre fonction: si la valeur de l'indicateur est positive, cela signifie que la valeur n'est pas nulle, mais qu'elle a été tronquée quand elle a été stockée dans la variable hôte.

Si l'argument `-r no_indicator` est passée au préprocesseur `ecpg`, il fonction dans le mode « no-indicator ». En mode no-indicator, si aucune variable indicator n'est spécifiée, les valeurs nulles sont signalées (en entrée et en sortie) pour les types chaînes de caractère comme des chaînes vides et pour les types integer comme la plus petite valeur possible pour le type (par exemple, `INT_MIN` pour `int`).

36.5. SQL Dynamique

Fréquemment, les ordres SQL particuliers qu'une application doit exécuter sont connus au moment où l'application est écrite. Dans certains cas, par contre, les ordres SQL sont composés à l'exécution ou fournis par une source externe. Dans ces cas, vous ne pouvez pas embarquer les ordres SQL directement dans le code source C, mais il y a une fonctionnalité qui vous permet d'exécuter des ordres SQL que vous fournissez dans une variable de type chaîne.

36.5.1. Exécuter des Ordres SQL Dynamiques sans Jeu de Donnée

La façon la plus simple d'exécuter un ordre SQL dynamique est d'utiliser la commande `EXECUTE IMMEDIATE`. Par exemple :

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

`EXECUTE IMMEDIATE` peut être utilisé pour des ordres SQL qui ne retournent pas de données (par exemple, LDD, `INSERT`, `UPDATE`, `DELETE`). Vous ne pouvez pas exécuter d'ordres qui ramènent des données (par exemple, `SELECT`) de cette façon. La prochaine section décrit comment le faire.

36.5.2. Exécuter une requête avec des paramètres d'entrée

Une façon plus puissante d'exécuter des ordres SQL arbitraires est de les préparer une fois et d'exécuter la requête préparée aussi souvent que vous le souhaitez. Il est aussi possible de préparer une version généralisé d'une requête et d'ensuite en exécuter des versions spécifiques par substitution de paramètres. Quand vous préparez la requête, mettez des points d'interrogation où vous voudrez substituer des paramètres ensuite. Par exemple :

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

Quand vous n'avez plus besoin de la requête préparée, vous devriez la désallouer :

```
EXEC SQL DEALLOCATE PREPARE name;
```

36.5.3. Exécuter une Requête avec un Jeu de Données

Pour exécuter une requête SQL avec une seule ligne de résultat, vous pouvez utiliser `EXECUTE`. Pour enregistrer le résultat, ajoutez une clause `INTO`.

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
```

Une commande `EXECUTE` peut avoir une clause `INTO`, une clause `USING`, les deux, ou aucune.

Si une requête peut ramener plus d'un enregistrement, un curseur devrait être utilisé, comme dans l'exemple suivant. Voyez Section 36.3.2 pour plus de détails à propos des curseurs.)

```
EXEC SQL BEGIN DECLARE SECTION;
char dbaname[128];
char datname[128];
```

```
char *stmt = "SELECT u.usename as dbaname, d.datname "
             "  FROM pg_database d, pg_user u "
             "  WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;
EXEC SQL SELECT pg_catalog.set_config('search_path', '', false);
 EXEC SQL COMMIT;

EXEC SQL PREPARE stmt1 FROM :stmt;

EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH cursor1 INTO :dbaname,:datname;
    printf("dbaname=%s, datname=%s\n", dbaname, datname);
}

EXEC SQL CLOSE cursor1;

EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
```

36.6. Librairie pgtypes

La librairie pgtypes met en correspondance les types de la base de données PostgreSQL avec des équivalents en C qui peuvent être utilisés dans des programmes en C. Elle fournit aussi des fonctions pour effectuer les calculs de base avec ces types en C, c'est à dire, sans l'aide du serveur PostgreSQL. Voyez l'exemple suivant :

```
EXEC SQL BEGIN DECLARE SECTION;
   date date1;
   timestamp ts1, tsout;
   interval iv1;
   char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl
 WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
PGTYPESchar_free(out);
```

36.6.1. Chaîne de caractères

Certaines fonctions comme `PGTYPESnumeric_to_asc` renvoient un pointeur vers une chaîne de caractères fraichement allouée. Ces allocations doivent être libérées avec `PGTYPESchar_free`

plutôt que `free`. (Ceci est seulement important sur Windows où l'allocation et la désallocation de la mémoire ont parfois besoin d'être réalisées par la même bibliothèque.)

36.6.2. Le type numeric

Le type numeric permet de faire des calculs de précision arbitraire. Voyez Section 8.1 pour le type équivalent dans le serveur PostgreSQL. En raison de cette précision arbitraire cette variable doit pouvoir s'étendre et se réduire dynamiquement. C'est pour cela que vous ne pouvez créer des variables numeric que sur le tas, en utilisant les fonctions `PGTYPESnumeric_new` et `PGTYPESnumeric_free`. Le type décimal, qui est similaire mais de précision limitée, peut être créé sur la pile ou sur le tas.

Les fonctions suivantes peuvent être utilisées pour travailler avec le type numeric :

`PGTYPESnumeric_new`

Demander un pointeur vers une variable numérique nouvellement allouée.

```
numeric *PGTYPESnumeric_new(void);
```

`PGTYPESnumeric_free`

Désallouer un type numérique, libérer toute sa mémoire.

```
void PGTYPESnumeric_free(numeric *var);
```

`PGTYPESnumeric_from_asc`

Convertir un type numérique à partir de sa notation chaîne.

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

Les formats valides sont par exemple: `-2`, `.794`, `+3.44`, `592.49E07` or `-32.84e-4`. Si la valeur peut être convertie correctement, un pointeur valide est retourné, sinon un pointeur NULL. À l'heure actuelle ECPG traite toujours la chaine en entier, il n'est donc pas possible pour le moment de stocker l'adresse du premier caractère invalide dans `*endptr`. Vous pouvez sans risque positionner `endptr` à NULL.

`PGTYPESnumeric_to_asc`

Retourne un pointeur vers la chaîne allouée par `malloc` qui contient la représentation chaîne du type numérique `num`.

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

La valeur numérique sera affichée avec `dscale` chiffres décimaux, et sera arrondie si nécessaire. Le résultat doit être libéré avec `PGTYPESchar_free()`.

`PGTYPESnumeric_add`

Ajoute deux variables numériques à une troisième.

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric
 *result);
```

La fonction additionne les variables `var1` et `var2` dans la variable résultat `result`. La fonction retourne 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_sub`

Soustrait deux variables numériques et retourne le résultat dans une troisième.

```
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric
 *result);
```

La fonction soustrait la variable `var2` de la variable `var1`. Le résultat de l'opération est stocké dans la variable `result`. La fonction retourne 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_mul`

Multiplie deux valeurs numeric et retourne le résultat dans une troisième.

```
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric
 *result);
```

La fonction multiplie la variable `var2` de la variable `var1`. Le résultat de l'opération est stocké dans la variable `result`. La fonction retourne 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_div`

Divise deux valeurs numeric et retourne le résultat dans une troisième.

```
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric
 *result);
```

La fonction divise la variable `var2` de la variable `var1`. Le résultat de l'opération est stocké dans la variable `result`. La fonction retourne 0 en cas de succès et -1 en cas d'erreur.

`PGTYPESnumeric_cmp`

Compare deux variables numeric.

```
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
```

Cette fonction compare deux variables numeric. En cas d'erreur, `INT_MAX` est retourné. En cas de réussite, la fonction retourne un des trois résultats suivants :

- 1, si `var1` est plus grand que `var2`
- -1, si `var1` est plus petit que `var2`
- 0, si `var1` et `var2` sont égaux

`PGTYPESnumeric_from_int`

Convertit une variable int en variable numeric.

```
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
```

Cette fonction accepte une variable de type signed int et la stocke dans la variable numeric `var`. La fonction retourne 0 en cas de réussite, et -1 en cas d'échec.

`PGTYPESnumeric_from_long`

Convertit une variable long int en variable numeric.

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric
 *var);
```

Cette fonction accepte une variable de type signed long int et la stocke dans la variable numeric `var`. La fonction retourne 0 en cas de réussite, et -1 en cas d'échec.

`PGTYPESnumeric_copy`

Copie une variable numeric dans une autre.

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

Cette fonction copie la valeur de la variable vers laquelle `src` pointe dans la variable vers laquelle `dst`. Elle retourne 0 en cas de réussite et -1 en cas d'échec.

`PGTYPESnumeric_from_double`

Convertit une variable de type double en variable numeric.

```
int  PGTYPESnumeric_from_double(double d, numeric *dst);
```

Cette fonction accepte une variable de type double et la stocke dans la variable numeric `dst`. La fonction retourne 0 en cas de réussite, et -1 en cas d'échec.

`PGTYPESnumeric_to_double`

Convertit une variable de type numeric en double.

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

Cette fonction convertit la valeur numeric de la variable vers la quelle `nv` pointe vers la variable double vers laquelle `dp` pointe. Elle retourne 0 en cas de réussite et -1 en cas d'échec, les cas de dépassement de capacité inclus. En cas de dépassement, la variable globale `errno` sera positionnée à `PGTYPES_NUM_OVERFLOW` en plus.

`PGTYPESnumeric_to_int`

Convertit une variable de type numeric en int.

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

Cette fonction convertit la valeur numeric de la variable vers la quelle `nv` pointe vers la variable int vers laquelle `ip` pointe. Elle retourne 0 en cas de réussite et -1 en cas d'échec, les cas de dépassement de capacité inclus. En cas de dépassement, la variable globale `errno` sera positionnée à `PGTYPES_NUM_OVERFLOW` en plus.

`PGTYPESnumeric_to_long`

Convertit une variable de type numeric en long.

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

Cette fonction convertit la valeur numeric de la variable vers la quelle `nv` pointe vers la variable long vers laquelle `lp` pointe. Elle retourne 0 en cas de réussite et -1 en cas d'échec, les cas de dépassement de capacité inclus. En cas de dépassement, la variable globale `errno` sera positionnée à `PGTYPES_NUM_OVERFLOW` en plus.

`PGTYPESnumeric_to_decimal`

Convertit une variable de type numeric en decimal.

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

Cette fonction convertit la valeur numeric de la variable vers la quelle `src` pointe vers la variable decimal vers laquelle `dst` pointe. Elle retourne 0 en cas de réussite et -1 en cas d'échec, les cas de dépassement de capacité inclus. En cas de dépassement, la variable globale `errno` sera positionnée à `PGTYPES_NUM_OVERFLOW` en plus.

`PGTYPESnumeric_from_decimal`

Convertit une variable de type decimal en numeric.

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

Cette fonction convertit la valeur decimal de la variable vers la quelle `src` pointe vers la variable numeric vers laquelle `dst` pointe. Elle retourne 0 en cas de réussite et -1 en cas d'échec. Comme le type decimal est implémentée comme une version limitée du type numeric, un dépassement ne peut pas se produire lors de cette conversion.

36.6.3. Le Type date

Le type date en C permet à votre programme de traiter les données type type SQL date. Voyez Section 8.5 pour le type équivalent du serveur PostgreSQL.

Les fonctions suivantes peuvent être utilisées pour travailler avec le type date :

`PGTYPESdate_from_timestamp`

Extraire la partie date d'un timestamp.

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

Cette fonction reçoit un timestamp comme seul argument et retourne la partie date extraite de ce timestamp.

`PGTYPESdate_from_asc`

Convertit une date à partir de sa représentation textuelle.

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

Cette fonction reçoit une chaîne char* C `str` et un pointeur vers une chaîne char* C `endptr`. À l'heure actuelle ECPG traite toujours intégralement la chaîne, et ne supporte donc pas encore l'adresse du premier caractère invalide dans `*endptr`. Vous pouvez positionner `endptr` à NULL sans risque.

Notez que la fonction attend toujours une date au format MDY et qu'il n'y a aucune variable à l'heure actuelle pour changer cela dans ECPG.

Tableau 36.2 montre les formats autorisés en entrée.

Tableau 36.2. Formats d'Entrée Valides pour `PGTYPESdate_from_asc`

Entrée	Sortie
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
1/18/1999	January 18, 1999
01/02/03	February 1, 2003
1999-Jan-08	January 8, 1999
Jan-08-1999	January 8, 1999
08-Jan-1999	January 8, 1999
99-Jan-08	January 8, 1999
08-Jan-99	January 8, 1999
08-Jan-06	January 8, 2006
Jan-08-99	January 8, 1999
19990108	ISO 8601; January 8, 1999
990108	ISO 8601; January 8, 1999
1999.008	year and day of year
J2451187	Julian day
January 8, 99 BC	year 99 before the Common Era

`PGTYPESdate_to_asc`

Retourne la représentation textuelle d'une variable date.

```
char *PGTYPESdate_to_asc(date dDate);
```

La fonction reçoit la date `dDate` comme unique paramètre. Elle retournera la date dans la forme `1999-01-18`, c'est-à-dire le format `YYYY-MM-DD`. Le résultat doit être libéré avec `PGTYPESchar_free()`.

`PGTYPESdate_julmdy`

Extrait les valeurs pour le jour, le mois et l'année d'une variable de type date.

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

La fonction reçoit la date `d` et un pointeur vers un tableau de 3 valeurs entières `mdy`. Le nom de variable indique l'ordre séquentiel: `mdy[0]` contiendra le numéro du mois, `mdy[1]` contiendra le numéro du jour et `mdy[2]` contiendra l'année.

`PGTYPESdate_mdyjul`

Crée une valeur date à partir d'un tableau de 3 entiers qui spécifient le jour, le mois et l'année de la date.

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

Cette fonction reçoit le tableau des 3 entiers (`mdy`) comme premier argument, et son second argument est un pointeur vers la variable de type date devant contenir le résultat de l'opération.

`PGTYPESdate_dayofweek`

Retourne un nombre représentant le jour de la semaine pour une valeur date.

```
int PGTYPESdate_dayofweek(date d);
```

La fonction reçoit la variable date `d` comme seul argument et retourne un entier qui indique le jour de la semaine pour cette date. this date.

- 0 - Dimanche
- 1 - Lundi
- 2 - Mardi
- 3 - Mercredi
- 4 - Jeudi
- 5 - Vendredi
- 6 - Samedi

`PGTYPESdate_today`

Récupérer la date courante.

```
void PGTYPESdate_today(date *d);
```

Cette fonction reçoit un pointeur vers une variable date (`d`) qu'il positionne à la date courante.

`PGTYPESdate_fmt_asc`

Convertir une variable de type date vers sa représentation textuelle en utilisant un masque de formatage.

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char
 *outbuf);
```

La fonction reçoit la date à convertir (`dDate`), le masque de formatage (`fmtstring`) et la chaîne qui contiendra la représentation textuelle de la date (`outbuf`).

En cas de succès, 0 est retourné, et une valeur négative si une erreur s'est produite.

Les littéraux suivants sont les spécificateurs de champs que vous pouvez utiliser:

- `dd` - Le numéro du jour du mois.
- `mm` - Le numéro du mois de l'année.
- `yy` - Le numéro de l'année comme nombre à deux chiffres.
- `yyyy` - Le numéro de l'année comme nombre à quatre chiffres.
- `ddd` - Le nom du jour (abrégé).
- `mmm` - Le nom du mois (abrégé).

Tout autre caractère est recopié tel quel dans la chaîne de sortie.

Tableau 36.3 indique quelques formats possibles. Cela vous donnera une idée de comment utiliser cette fonction. Toutes les lignes de sortie reposent sur la même date : Le 23 novembre 1959.

Tableau 36.3. Formats d'Entrée Valides pour `PGTYPESdate_fmt_asc`

Format	Résultat
`mmddyy`	`112359`
`ddmmyy`	`231159`
`yymmdd`	`591123`
`yy/mm/dd`	`59/11/23`
`yy mm dd`	`59 11 23`
`yy.mm.dd`	`59.11.23`
`.mm.yyyy.dd.`	`.11.1959.23.`
`mmm. dd, yyyy`	`Nov. 23, 1959`
`mmm dd yyyy`	`Nov 23 1959`
`yyyy dd mm`	`1959 23 11`
`ddd, mmm. dd, yyyy`	`Mon, Nov. 23, 1959`
`(ddd) mmm. dd, yyyy`	`(Mon) Nov. 23, 1959`

`PGTYPESdate_defmt_asc`

Utiliser un masque de formatage pour convertir une chaîne de caractère `char*` en une valeur de type date.

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

La fonction reçoit un pointeur vers la valeur de date qui devrait stocker le résultat de l'opération (`d`), le masque de formatage à utiliser pour traiter la date (`fmt`) et la chaîne de caractères char* C contenant la représentation textuelle de la date (`str`). La représentation textuelle doit correspondre au masque de formatage. Toutefois, vous n'avez pas besoin d'avoir une correspondance exacte entre la chaîne et le masque de formatage. La fonction n'analyse qu'en ordre séquentiel et cherche les litéraux `yy` ou `yyyy` qui indiquent la position de l'année, `mm` qui indique la position du mois et `dd` qui indique la position du jour.

Tableau 36.4 indique quelques formats possibles. Cela vous donnera une idée de comment utiliser cette fonction

Tableau 36.4. Formats d'Entrée Valides pour `rdefmtdate`

Format	Chaîne	Résultat
ddmmyy	21-2-54	1954-02-21
ddmmyy	2-12-54	1954-12-02
ddmmyy	20111954	1954-11-20
ddmmyy	130464	1964-04-13
mmm.dd.yyyy	MAR-12-1967	1967-03-12
yy/mm/dd	1954, February 3rd	1954-02-03
mmm.dd.yyyy	041269	1969-04-12
yy/mm/dd	In the year 2525, in the month of July, mankind will be alive on the 28th day	2525-07-28
dd-mm-yy	I said on the 28th of July in the year 2525	2525-07-28
mmm.dd.yyyy	9/14/58	1958-09-14
yy/mm/dd	47/03/29	1947-03-29
mmm.dd.yyyy	oct 28 1975	1975-10-28
mmddyy	Nov 14th, 1985	1985-11-14

36.6.4. Le Type timestamp

Le type timestamp en C permet à vos programmes de manipuler les données du type SQL timestamp. Voyez Section 8.5 pour le type équivalent dans le serveur PostgreSQL.

Les fonctions suivantes peuvent être utilisées pour manipuler le type timestamp :

`PGTYPEStimestamp_from_asc`

Transformer un timestamp de sa représentation texte vers une variable timestamp.

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

La fonction reçoit la chaîne à analyser (`str`) et un pointeur vers un char* C (`endptr`). À l'heure actuelle ECPG traite toujours intégralement la chaîne, et ne supporte donc pas encore l'adresse du premier caractère invalide dans `*endptr`. Vous pouvez positionner `endptr` à NULL sans risque.

La fonction retourne le timestamp identifié en cas de réussite. En cas d'erreur, `PGTYPESInvalidTimestamp` est retourné et `error` est positionné à `PGTYPES_TS_BAD_TIMESTAMP`. Voyez `PGTYPESInvalidTimestamp` pour des informations importantes sur cette valeur.

En général, la chaîne d'entrée peut contenir toute combinaison d'une spécification de date autorisée, un caractère espace et une spécification de temps (time) autorisée. Notez que les timezones ne sont pas supportées par ECPG. Il peut les analyzer mais n'applique aucune calcul

comme le ferait le serveur PostgreSQL par exemple. Les spécificateurs de timezone sont ignorées en silence.

Tableau 36.5 contient quelques exemples pour les chaînes d'entrée.

Tableau 36.5. Formats d'Entrée Valide pour `PGTYPEStimestamp_from_asc`

Entrée	Résultat
`1999-01-08 04:05:06`	`1999-01-08 04:05:06`
`January 8 04:05:06 1999 PST`	`1999-01-08 04:05:06`
`1999-Jan-08 04:05:06.789-8`	`1999-01-08 04:05:06.789 (time zone specifier ignored)`
`J2451187 04:05-08:00`	`1999-01-08 04:05:00 (time zone specifier ignored)`

`PGTYPEStimestamp_to_asc`

Convertit une date vers une chaîne char* C.

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

Cette fonction reçoit le timestamp `tstamp` comme seul argument et retourne une chaîne allouée qui contient la représentation textuelle du timestamp. Le résultat doit être libéré avec `PGTYPESchar_free()`.

`PGTYPEStimestamp_current`

Récupère le timestamp courant.

```
void PGTYPEStimestamp_current(timestamp *ts);
```

Cette fonction récupère le timestamp courant et le sauve dans la variable timestamp vers laquelle `ts` pointe.

`PGTYPEStimestamp_fmt_asc`

Convertit une variable timestamp vers un char* C en utilisant un masque de formatage.

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int
 str_len, char *fmtstr);
```

Cette fonction reçoit un pointeur vers le timestamp à convertir comme premier argument (`ts`), un pointeur vers le tampon de sortie (`output`), la longueur maximale qui a été allouée pour le tampon de sortie (`str_len`) et le masque de formatage à utiliser pour la conversion (`fmtstr`).

En cas de réussite, la fonction retourne 0, et une valeur négative en cas d'erreur.

Vous pouvez utiliser les spécificateurs de format suivant pour le masque de formatage. Les spécificateurs sont les mêmes que ceux utilisés dans la fonction `strftime` de la libc. Tout spécificateur ne correspondant pas à du formatage sera copié dans le tampon de sortie.

- `%A` - est remplacé par la représentation nationale du nom complet du jour de la semaine.

- %a - est remplacé par la représentation nationale du nom abrégé du jour de la semaine.
- %B - est remplacé par la représentation nationale du nom complet du mois.
- %b - est remplacé par la représentation nationale du nom abrégé du mois.
- %C - est remplacé par (année / 100) sous forme de nombre décimal; les chiffres seuls sont précédés par un zéro.
- %c - est remplacé par la représentation nationale de time et date.
- %D - est équivalent à %m/%d/%y.
- %d - est remplacé par le jour du mois sous forme de nombre décimal (01–31).
- %E* %O* - Extensions locales POSIX Les séquences: %Ec %EC %Ex %EX %Ey %EY %Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy sont supposées fournir des représentations alternatives.

  De plus, %OB est implémenté pour représenter des noms de mois alternatifs (utilisé seul, sans jour mentionné).
- %e - est remplacé par le jour du mois comme nombre décimal (1–31) ; les chiffres seuls sont précédés par un blanc.
- %F - est équivalent à %Y-%m-%d.
- %G - est remplacé par une année comme nombre décimal avec le siècle. L'année courante est celle qui contient la plus grande partie de la semaine (Lundi est le premier jour de la semaine).
- %g - est remplacé par la même année que dans %G, mais comme un nombre décimal sans le siècle. (00–99).
- %H - est remplacé par l'heure (horloge sur 24 heures) comme nombre décimal (00–23).
- %h - comme %b.
- %I - est remplacé par l'heure (horloge sur 12 heures) comme nombre décimal(01–12).
- %j - est remplacé par le jour de l'année comme nombre décimal (001–366).
- %k - est remplacé par l'heure (horloge sur 24 heures) comme nombre décimal (0–23) ; les chiffres seuls sont précédés par un blanc.
- %l - est remplacé par l'heure (horloge sur 12 heures) comme nombre décimal (1–12) ; les chiffres seuls sont précédés par un blanc.
- %M - est remplacé par la minute comme nombre décimal (00–59).
- %m - est remplacé par le mois comme nombre décimal (01–12).
- %n - est remplacé par un caractère nouvelle ligne.
- %O* - comme %E*.
- %p - est remplacé par la représentation nationale de « ante meridiem » ou « post meridiem » suivant la valeur appropriée.
- %R - est équivalent à %H:%M.
- %r - est équivalent à %I:%M:%S %p.
- %S - est remplacé par la seconde comme nombre décimal (00–60).

- `%s` - est remplacé par le nombre de secondes depuis l'Epoch, en UTC.
- `%T` - est équivalent à `%H:%M:%S`
- `%t` - est remplacé par une tabulation.
- `%U` - est remplacé par le numéro de la semaine dans l'année (Dimanche est le premier jour de la semaine) comme nombre décimal (00–53).
- `%u` - est remplacé par le jour de la semaine (Lundi comme premier jour de la semaine) comme nombre décimal (1–7).
- `%V` - est remplacé par le numéro de la semaine dans l'année (Lundi est le premier jour de la semaine) comme nombre décimal (01–53). Si l'année contenant le 1er Janvier a 4 jours ou plus dans la nouvelle année, alors c'est la semaine numéro 1; sinon, c'est la dernière semaine de l'année précédente, et la semaine suivante est la semaine 1.
- `%v` - est équivalent à `%e-%b-%Y`.
- `%W` - est remplacé par le numéro de la semaine dans l'année (Lundi est le premier jour de la semaine) comme nombre décimal (00–53).
- `%w` - est remplacé par le jour de la semaine (Dimanche comme premier jour de la semaine) comme nombre décimal (0–6).
- `%X` - est remplacé par la représentation nationale du temps.
- `%x` - est remplacé par la représentation nationale de la date.
- `%Y` - est remplacé par l'année avec le siècle comme un nombre décimal.
- `%y` - est remplacé par l'année sans le siècle comme un nombre décimal (00–99).
- `%Z` - est remplacé par le nom de la zone de temps.
- `%z` - est remplacé par le décalage de la zone de temps par rapport à UTC; un signe plus initial signifie à l'est d'UTC, un signe moins à l'ouest d'UTC, les heures et les minutes suivent avec deux chiffres chacun et aucun délimiteur entre eux (forme commune pour les entêtes de date spécifiés par la RFC 822[1]).
- `%+` - est remplacé par la représentation nationale de la date et du temps.
- `%-*` - extension de la libc GNU. Ne pas faire de padding (bourrage) sur les sorties numériques.
- `$_*` - extension de la libc GNU. Spécifie explicitement l'espace pour le padding.
- `%0*` - extension de la libc GNU. Spécifie explicitement le zéro pour le padding.
- `%%` - est remplacé par `%`.

`PGTYPEStimestamp_sub`

Soustraire un timestamp d'un autre et sauver le résultat dans une variable de type interval.

```
int PGTYPEStimestamp_sub(timestamp *ts1, timestamp *ts2,
 interval *iv);
```

Cette fonction soustrait la variable timestamp vers laquelle pointe `ts2` de la variable de timestamp vers laquelle `ts1` pointe, et stockera le résultat dans la variable interval vers laquelle `iv` pointe.

[1] https://datatracker.ietf.org/doc/html/rfc822

En cas de succès, la fonction retourne 0, et une valeur négative si une erreur s'est produite.

`PGTYPESTimestamp_defmt_asc`

Convertit une valeur timestamp de sa représentation textuelle en utilisant un masque de formatage.

```
int PGTYPESTimestamp_defmt_asc(char *str, char *fmt, timestamp
 *d);
```

Cette fonction reçoit la représentation textuelle d'un timestamp dans la variable `str` ainsi que le masque de formatage à utiliser dans la variable `fmt`. Le résultat sera stocké dans la variable vers laquelle `d` pointe.

Si le masque de formatage `fmt` est NULL, la fonction se rabattra vers le masque de formatage par défaut qui est `%Y-%m-%d %H:%M:%S`.

C'est la fonction inverse de `PGTYPESTimestamp_fmt_asc`. Voyez la documentation à cet endroit pour découvrir toutes les entrées possibles de masque de formatage.

`PGTYPESTimestamp_add_interval`

Ajouter une variable interval à une variable timestamp.

```
int PGTYPESTimestamp_add_interval(timestamp *tin, interval
 *span, timestamp *tout);
```

Cette fonction reçoit un pointeur vers une variable timestamp `tin` et un pointeur vers une variable interval `span`. Elle ajoute l'intervalle au timestapm et sauve le timestamp résultat dans la variable vers laquelle `tout` pointe.

En cas de succès, la fonction retourne 0, et une valeur négative si une erreur s'est produite.

`PGTYPESTimestamp_sub_interval`

Soustrait une variable interval d'une variable timestamp.

```
int PGTYPESTimestamp_sub_interval(timestamp *tin, interval
 *span, timestamp *tout);
```

Cette fonction soustrait la variable interval vers laquelle `span` pointe de la variable timestamp vers laquelle `tin` pointe et sauve le résultat dans la variable vers laquelle `tout` pointe.

En cas de succès, la fonction retourne 0, et une valeur négative si une erreur s'est produite.

36.6.5. Le Type interval

Le type interval en C permet à vos programmes de manipuler des données du type SQL interval. Voyez Section 8.5 pour le type équivalent dans le serveur PostgreSQL.

Les fonctions suivantes peuvent être utilisées pour travailler avec le type interval :

`PGTYPESinterval_new`

Retourne un pointeur vers une variable interval nouvellement allouée.

```
interval *PGTYPESinterval_new(void);
```

`PGTYPESinterval_free`

Libère la mémoire d'une variable interval précédemment allouée.

```
void PGTYPESinterval_free(interval *intvl);
```

`PGTYPESinterval_from_asc`

Convertit un interval à partir de sa représentation textuelle.

```
interval *PGTYPESinterval_from_asc(char *str, char **endptr);
```

Cette fonction traite la chaîne d'entrée `str` et retourne un pointeur vers une variable interval allouée. À l'heure actuelle ECPG traite toujours intégralement la chaîne, et ne supporte donc pas encore l'adresse du premier caractère invalide dans `*endptr`. Vous pouvez positionner `endptr` à NULL sans risque.

`PGTYPESinterval_to_asc`

Convertit une variable de type interval vers sa représentation textuelle.

```
char *PGTYPESinterval_to_asc(interval *span);
```

Cette fonction convertit la variable interval vers laquelle `span` pointe vers un char* C. La sortie ressemble à cet exemple : `@ 1 day 12 hours 59 mins 10 secs`. Le résultat doit être libéré avec `PGTYPESchar_free()`.

`PGTYPESinterval_copy`

Copie une variable de type interval.

```
int PGTYPESinterval_copy(interval *intvlsrc, interval
 *intvldest);
```

Cette fonction copie la variable interval vers laquelle `intvlsrc` pointe vers la variable vers laquelle `intvldest` pointe. Notez que vous devrez allouer la mémoire pour la variable destination auparavant.

36.6.6. Le Type decimal

Le type decimal est similaire au type numeric. Toutefois il est limité à une précision maximale de 30 chiffres significatifs. À l'opposeé du type numeric que ne peut être créé que sur le tas, le type decimal peur être créé soit sur la pile soit sur le tas (au moyen des fonctions `PGTYPESdecimal_new` et `PGTYPESdecimal_free`). Il y a beaucoup d'autres fonctions qui manipulent le type decimal dans le mode de compatibilité Informix décrit dans Section 36.15.

Les fonctions suivantes peut être utilisée pour travailler avec le type decimal et ne sont pas seulement contenues dans la librairie `libcompat`.

`PGTYPESdecimal_new`

Demande un pointeur vers une variable decimal nouvellement allouée.

```
decimal *PGTYPESdecimal_new(void);
```

`PGTYPESdecimal_free`

Libère un type decimal, libère toute sa mémoire.

```
void PGTYPESdecimal_free(decimal *var);
```

36.6.7. errno Valeurs de pgtypeslib

`PGTYPES_NUM_BAD_NUMERIC`

Un argument devrait contenir une variable numeric (ou pointer vers une variable numeric) mais en fait sa représentation en mémoire était invalide.

`PGTYPES_NUM_OVERFLOW`

Un dépassement de capacité s'est produit. Comme le type numeric peut travailler avec une précision quasi-arbitraire, convertir une variable numeric vers d'autres types peut causer un dépassement.

`PGTYPES_NUM_UNDERFLOW`

Un soupassement de capacité s'est produit. Comme le type numeric peut travailler avec une précision quasi-arbitraire, convertir une variable numeric vers d'autres types peut causer un soupassement.

`PGTYPES_NUM_DIVIDE_ZERO`

Il y a eu une tentative de division par zéro.

`PGTYPES_DATE_BAD_DATE`

Une chaîne de date invalide a été passée à la fonction `PGTYPESdate_from_asc`.

`PGTYPES_DATE_ERR_EARGS`

Des arguments invalides ont été passés à la fonction `PGTYPESdate_defmt_asc`.

`PGTYPES_DATE_ERR_ENOSHORTDATE`

Un indicateur invalide a été trouvé dans la chaîne d'entrée par la fonction `PGTYPESdate_defmt_asc`.

`PGTYPES_INTVL_BAD_INTERVAL`

Une chaîne invalide d'interval a été passée à la fonction `PGTYPESinterval_from_asc`, ou une valeur invalide d'interval a été passée à la fonction `PGTYPESinterval_to_asc`.

`PGTYPES_DATE_ERR_ENOTDMY`

Il n'a pas été possible de trouver la correspondance dans l'assignement jour/mois/année de la fonction `PGTYPESdate_defmt_asc`.

`PGTYPES_DATE_BAD_DAY`

Un jour de mois invalide a été trouvé par la fonction `PGTYPESdate_defmt_asc`.

`PGTYPES_DATE_BAD_MONTH`

Une valeur de mois invalide a été trouvée par la fonction `PGTYPESdate_defmt_asc`.

`PGTYPES_TS_BAD_TIMESTAMP`

Une chaîne de timestamp invalide a été passée à la fonction `PGTYPEStimestamp_from_asc`, ou une valeur invalide de timestamp a été passée à la fonction `PGTYPEStimestamp_to_asc`.

`PGTYPES_TS_ERR_EINFTIME`

Une valeur infinie de timestamp a été rencontrée dans un context qui ne peut pas la manipuler.

36.6.8. Constantes Spéciales de pgtypeslib

`PGTYPESInvalidTimestamp`

Une valeur de timestamp représentant un timestamp invalide. C'est retourné par la fonction `PGTYPEStimestamp_from_asc` en cas d'erreur de conversion. Notez qu'en raison de la représentation interne du type de données `timestamp`, `PGTYPESInvalidTimestamp` est aussi un timestamp valide en même temps. Il est positionné à `1899-12-31 23:59:59`. Afin de detecter les erreurs, assurez vous que votre application teste non seulement `PGTYPESInvalidTimestamp` mais aussi `error != 0` après chaque appel à `PGTYPEStimestamp_from_asc`.

36.7. Utiliser les Zones de Descripteur

Une zone de descripteur SQL (SQL Descriptor Area ou SQLDA) est une méthode plus sophistiquée pour traiter le résultat d'un ordre `SELECT`, `FETCH` ou `DESCRIBE`. Une zone de descripteur SQL regroupe les données d'un enregistrement avec ses métadonnées dans une seule structure. Ces métadonnées sont particulièrement utiles quand on exécute des ordres SQL dynamiques, où_la nature des colonnes résultat ne sont pas forcément connues à l'avance. PostgreSQL fournit deux façons d'utiliser des Zones de Descripteur: les Zones de Descripteur SQL nommée et les structures C SQLDA.

36.7.1. Zones de Descripteur SQL nommées

Une zone descripteur SQL nommé est composée d'un entête, qui contient des données concernant l'ensemble du descripteur, et une ou plusieurs zones de descriptions d'objets, qui en fait décrivent chaque colonne de l'enregistrement résultat.

Avant que vous puissiez utiliser une zone de descripteur SQL, vous devez en allouer une :

```
EXEC SQL ALLOCATE DESCRIPTOR identifiant;
```

L'identifiant sert de « nom de variable » de la zone de descripteur. Quand vous n'avez plus besoin du descripteur, vous devriez le désallouer:

```
EXEC SQL DEALLOCATE DESCRIPTOR identifiant;
```

Pour utiliser une zone de descripteur, spécifiez le comme cible de stockage dans une clause `INTO`, à la place d'une liste de variables hôtes :

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

Si le jeu de données retourné est vide, la zone de descripteur contiendra tout de même les métadonnées de la requête, c'est à dire les noms des champs.

Pour les requêtes préparées mais pas encore exécutées, l'ordre `DESCRIBE` peut être utilisé pour récupérer les métadonnées du résultat :

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

Avant PostgreSQL 9.0, le mot clé `SQL` était optionnel, par conséquent utiliser `DESCRIPTOR` et `SQL DESCRIPTOR` produisaent les mêmes zones de descripteur SQL. C'est maintenant obligatoire, et oublier le mot clé `SQL` produit des zones de descripteurs SQLDA, voyez Section 36.7.2.

Dans les ordres `DESCRIBE` et `FETCH`, les mots-clés `INTO` et `USING` peuvent être utilisés de façon similaire : ils produisent le jeu de données et les métadonnées de la zone de descripteur.

Maintenant, comment récupérer les données de la zone de descripteur ? Vous pouvez voir la zone de descripteur comme une structure avec des champs nommés. Pour récupérer la valeur d'un champ à partir de l'entête et le stocker dans une variable hôte, utilisez la commande suivante :

```
EXEC SQL GET DESCRIPTOR name :hostvar = field;
```

À l'heure actuelle, il n'y a qu'un seul champ d'entête défini: *`COUNT`*, qui dit combien il y a de zones de descripteurs d'objets (c'est à dire, combien de colonnes il y a dans le résultat). La variable hôte doit être de type integer. Pour récupérer un champ de la zone de description d'objet, utilisez la commande suivante :

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;
```

`num` peut être un integer literal, ou une variable hôte contenant un integer. Les champs possibles sont :

`CARDINALITY` (integer)

nombres d'enregistrements dans le résultat

`DATA`

objet de donnée proprement dit (par conséquent, le type de données de ce champ dépend de la requête)

`DATETIME_INTERVAL_CODE` (integer)

Quand `TYPE` est `9`, `DATETIME_INTERVAL_CODE` aura une valeur de `1` pour `DATE`, `2` pour `TIME`, `3` pour `TIMESTAMP`, `4` pour `TIME WITH TIME ZONE`, or `5` pour `TIMESTAMP WITH TIME ZONE`.

`DATETIME_INTERVAL_PRECISION` (integer)

non implémenté

`INDICATOR` (integer)

l'indicateur (indique une valeur null ou une troncature de valeur)

`KEY_MEMBER` (integer)

non implémenté

`LENGTH` (integer)

longueur de la donnée en caractères

`NAME` (string)

nom de la colonne

`NULLABLE` (integer)

non implémenté

`OCTET_LENGTH` (integer)

longueur de la représentation caractère de la donnée en octets

`PRECISION` (integer)

précision (pour les types `numeric`)

`RETURNED_LENGTH` (integer)

longueur de la donnée en caractères

`RETURNED_OCTET_LENGTH` (integer)

longueur de la représentation caractère de la donnée en octets

`SCALE` (integer)

échelle (pour le type `numeric`)

`TYPE` (integer)

code numérique du type de données de la colonne

Dans les ordres `EXECUTE`, `DECLARE` and `OPEN`, l'effet des mots clés `INTO` and `USING` est différent. Une zone de descripteur peut aussi être construite manuellement pour fournir les paramètres d'entré pour une requête ou un curseur et `USING SQL DESCRIPTOR` *`name`* est la façon de passer les paramètres d'entrée à une requête paramétrisée. L'ordre pour construire une zone de descripteur SQL est ci-dessous:

```
EXEC SQL SET DESCRIPTOR name VALUE num field = :hostvar;
```

PostgreSQL supporte la récupération de plus d'un enregistrement dans un ordre `FETCH` et les variables hôtes dans ce cas doivent être des tableaux. Par exemple :

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

```
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

36.7.2. Zones de Descripteurs SQLDA

Une zone de descripteur SQLDA est une structure C qui peut aussi être utilisé pour récupérer les résultats et les métadonnées d'une requête. Une structure stocke un enregistrement du jeu de résultat.

```
EXEC SQL include sqlda.h;
sqlda_t         *mysqlda;

EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

Netez que le mot clé `SQL` est omis. Les paragraphes qui parlent des cas d'utilisation de `INTO` and `USING` dans Section 36.7.1 s'appliquent aussi ici, avec un point supplémentaire. Dans un ordre `DESCRIBE` le mot clé `DESCRIPTOR` peut être complètement omis si le mot clé `INTO` est utilisé :

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

Le déroulement général d'un programme qui utilise des SQLDA est:

1. Préparer une requête, et déclarer un curseur pour l'utiliser.
2. Déclarer une SQLDA pour les lignes de résultat.
3. Déclarer une SQLDA pour les paramètres d'entrées, et les initialiser (allocation mémoire, positionnement des paramètres).
4. Ouvrir un curseur avec la SQLDA d'entrée.
5. Récupérer les enregistrements du curseur, et les stocker dans une SQLDA de sortie.
6. Lire les valeurs de la SQLDA de sortie vers les variables hôtes (avec conversion si nécessaire).
7. Fermer le curseur.
8. Libérer la zone mémoire allouée pour la SQLDA d'entrée.

36.7.2.1. Structure de Données SQLDA

Les SQLDA utilisent 3 types de structures de données: `sqlda_t`, `sqlvar_t`, et `struct sqlname`.

> **Astuce**
>
> La structure de la SQLDA de PostgreSQL est similaire à celle de DB2 Universal Database d'IBM, des informations techniques sur la SQLDA de DB2 peuvent donc aider à mieux comprendre celle de PostgreSQL.

36.7.2.1.1. Structure sqlda_t

Le type de structure `sqlda_t` est le type de la SQLDA proprement dit. Il contient un enregistrement. Et deux ou plus `sqlda_t` peuvent être connectées par une liste chaînée par le pointeur du champ

desc_next, représentant par conséquent une collection ordonnée d'enregistrements. Par conséquent, quand deux enregistrements ou plus sont récupérés, l'application peut les lire en suivant le pointeur desc_next dans chaque nœud sqlda_t.

La définition de sqlda_t est :

```
struct sqlda_struct
{
    char            sqldaid[8];
    long            sqldabc;
    short           sqln;
    short           sqld;
    struct sqlda_struct *desc_next;
    struct sqlvar_struct sqlvar[1];
};

typedef struct sqlda_struct sqlda_t;
```

La signification des champs est :

sqldaid

Elle contient la chaîne littérale "SQLDA ".

sqldabc

Il contient la taille de l'espace alloué en octets.

sqln

Il content le nombre de paramètres d'entrée pour une requête paramétrique, dans le cas où il est passé à un ordre OPEN, DECLARE ou EXECUTE utilisant le mot clé USING. Dans le cas où il sert de sortie à un ordre SELECT, EXECUTE ou FETCH statements, sa valeur est la même que celle du champ sqld.

sqld

Il contient le nombre de champs du résultat.

desc_next

Si la requête retourne plus d'un enregistrement, plusieurs structures SQLDA chaînées sont retournées, et desc_next contient un pointeur vers l'élément suivant (enregistrement) de la liste.

sqlvar

C'est le tableau des colonnes du résultat.

36.7.2.1.2. Structure de sqlvar_t

Le type structure sqlvar_t contient la valeur d'une colonne et les métadonnées telles que son type et sa longueur. La définition du type est :

```
struct sqlvar_struct
{
    short          sqltype;
    short          sqllen;
```

```
    char          *sqldata;
    short         *sqlind;
    struct sqlname sqlname;
};

typedef struct sqlvar_struct sqlvar_t;
```

La signification des champs est :

`sqltype`

Contient l'identifiant de type du champ. Pour les valeurs, voyez `enum ECPGttype` dans `ecpgtype.h`.

`sqllen`

Contient la longueur binaire du champ, par exemple 4 octets pour `ECPGt_int`.

`sqldata`

Pointe vers la donnée. Le format de la donnée est décrit dans Section 36.4.4.

`sqlind`

Pointe vers l'indicateur de nullité. 0 signifie non nul, -1 signifie nul. null.

`sqlname`

Le nom du champ.

36.7.2.1.3. Structure struct sqlname

Une structure `struct sqlname` contient un nom de colonne. Il est utilisé comme membre de la structure `sqlvar_t`. La définition de la structure est :

```
#define NAMEDATALEN 64

struct sqlname
{
        short           length;
        char            data[NAMEDATALEN];
};
```

La signification des champs est :

`length`

Contient la longueur du nom du champ.

`data`

Contient le nom du champ proprement dit.

36.7.2.2. Récupérer un jeu de données au moyen d'une SQLDA

Les étapes générales pour récupérer un jeu de données au moyen d'une SQLDA sont :

1. Déclarer une structure `sqlda_t` pour recevoir le jeu de données.

2. Exécuter des commandes FETCH/EXECUTE/DESCRIBE pour traiter une requête en spécifiant la SQLDA déclarée.

3. Vérifier le nombre d'enregistrements dans le résultat en inspectant sqln, un membre de la structure sqlda_t.

4. Récupérer les valeurs de chaque colonne des membres sqlvar[0], sqlvar[1], etc., de la structure sqlda_t.

5. Aller à l'enregistrement suivant (sqlda_t structure) en suivant le pointeur desc_next, un membre de la structure sqlda_t.

6. Répéter l'étape ci-dessus au besoin.

Voici un exemple de récupération d'un jeu de résultats au moyen d'une SQLDA.

Tout d'abord, déclarer une structure sqlda_t pour recevoir le jeu de résultats.

```
sqlda_t *sqlda1;
```

Puis, spécifier la SQLDA dans une commande. Voici un exemple avec une commande FETCH.

```
EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

Faire une boucle suivant la liste chaînée pour récupérer les enregistrements.

```
sqlda_t *cur_sqlda;

for (cur_sqlda = sqlda1;
     cur_sqlda != NULL;
     cur_sqlda = cur_sqlda->desc_next)
{
    ...
}
```

Dans la boucle, faire une autre boucle pour récupérer chaque colonne de données (sqlvar_t) de l'enregistrement.

```
for (i = 0; i < cur_sqlda->sqld; i++)
{
    sqlvar_t v = cur_sqlda->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqllen  = v.sqllen;
    ...
}
```

Pour récupérer une valeur de colonne, vérifiez la valeur de sqltype. Puis, suivant le type de la colonne, basculez sur une façon appropriée de copier les données du champ sqlvar vers une variable hôte.

```
char var_buf[1024];
```

```
switch (v.sqltype)
{
    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ?
 sizeof(var_buf) - 1 : sqllen));
        break;

    case ECPGt_int: /* integer */
        memcpy(&intval, sqldata, sqllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;

    ...
}
```

36.7.2.3. Passer des Paramètres de Requête en Utilisant une SQLDA

La méthode générale pour utiliser une SQLDA pour passer des paramètres d'entrée à une requête préparée sont :

1. Créer une requête préparée (prepared statement)
2. Déclarer une structure sqlda_t comme SQLDA d'entrée.
3. Allouer une zone mémorie (comme structure sqlda_t) pour la SQLDA d'entrée.
4. Positionner (copier) les valeurs d'entrée dans la mémoire allouée.
5. Ouvrir un curseur en spécifiant la SQLDA d'entrée.

Voici un exemple.

D'abord, créer une requête préparée.

```
EXEC SQL BEGIN DECLARE SECTION;
char query[1024] = "SELECT d.oid, * FROM pg_database d,
 pg_stat_database s WHERE d.oid = s.datid AND (d.datname = ? OR
 d.oid = ?)";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :query;
```

Puis, allouer de la mémoire pour une SQLDA, et positionner le nombre de paramètres d'entrée dans `sqln`, une variable membre de la structure<sqlda_t>. Quand deux paramètres d'entrée ou plus sont requis pour la requête préparée, l'application doit allouer de la mémoire supplémentaire qui est calculée par (nombre de paramètres -1) * sizeof<sqlvar_t>. Les exemples affichés ici allouent de l'espace mémoire pour deux paramètres d'entrée.

```
sqlda_t *sqlda2;

sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));

sqlda2->sqln = 2; /* nombre de variables d'entrée */
```

Après l'allocation mémoire, stocker les valeurs des paramètres dans le tableau`sqlvar[]`. (C'est le même tableau que celui qui est utilisé quand la SQLDA reçoit un jeu de résultats.) Dans cet exemple, les paramètres d'entrée sont `"postgres"`, de type chaîne, et `1`, de type integer.

```
sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqllen  = 8;

int intval = 1;
sqlda2->sqlvar[1].sqltype = ECPGt_int;
sqlda2->sqlvar[1].sqldata = (char *) &intval;
sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

En ouvrant un curseur et en spécifiant la SQLDA qui a été positionné auparavant, les paramètres d'entrée sont passés à la requête préparée.

```
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

Et pour finir, après avoir utilisé les SQLDAs d'entrée, la mémoire allouée doit être libérée explicitement, contrairement aux SQLDAs utilisé pour recevoir le résultat d'une requête.

```
free(sqlda2);
```

36.7.2.4. Une application de Démonstration Utilisant SQLDA

Voici un programme de démonstration, qui montre comment récupérer des statistiques d'accès des bases, spécifiées par les paramètres d'entrée, dans les catalogues systèmes.

Cette application joint deux tables systèmes, pg_database et pg_stat_database sur l'oid de la base, et récupère et affiche aussi les statistiques des bases qui sont spécifiées par deux paramètres d'entrées (une base `postgres` et un OID `1`).

Tout d'abord, déclarer une SQLDA pour l'entrée et une SQLDA pour la sortie.

```
EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* un descripteur de sortie */
sqlda_t *sqlda2; /* un descripteur d'entrée  */
```

Puis, se connecter à la base, préparer une requête, et déclarer un curseur pour la requête préparée.

```
int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d,
 pg_stat_database s WHERE d.oid=s.datid AND ( d.datname=? OR
 d.oid=? )";
    EXEC SQL END DECLARE SECTION;
```

```
    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

Puis, mettre des valeurs dans la SQLDA d'entrée pour les paramètres d'entrée. Allouer de la mémoire pour la SQL d'entrée, et positionner le nombre de paramètres d'entrée dans `sqln`. Stocker le type, la valeur et la longueur de la valeur dans `sqltype`, `sqldata` et `sqllen` dans la structure `sqlvar`.

```
    /* Créer une structure SQLDA pour les paramètres d'entrée. */
    sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) +
 sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* number of input variables */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *)&intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

Après avoir positionné la SQLDA d'entrée, ouvrir un curseur avec la SQLDA d'entrée.

```
    /* Ouvrir un curseur avec les paramètres d'entrée. */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

Récupérer les enregistrements dans la SQLDA de sortie à partir du curseur ouvert. (En général, il faut appeler `FETCH` de façon répétée dans la boucle, pour récupérer tous les enregistrements du jeu de données.)

```
    while (1)
    {
        sqlda_t *cur_sqlda;

        /* Assigner le descripteur au curseur  */
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

Ensuite, récupérer les enregistrements du FETCH de la SQLDA, en suivant la liste chaînée de la structure `sqlda_t`.

```
    for (cur_sqlda = sqlda1 ;
         cur_sqlda != NULL ;
         cur_sqlda = cur_sqlda->desc_next)
    {
        ...
```

Lire chaque colonne dans le premier enregistrement. Le nombre de colonnes est stocké dans `sqld`, les données réelles de la première colonne sont stockées dans `sqlvar[0]`, tous deux membres de la structure`sqlda_t`.

```
        /* Afficher toutes les colonnes d'un enregistrement. */
        for (i = 0; i < sqlda1->sqld; i++)
        {
            sqlvar_t v = sqlda1->sqlvar[i];
            char *sqldata = v.sqldata;
            short sqllen  = v.sqllen;

            strncpy(name_buf, v.sqlname.data, v.sqlname.length);
            name_buf[v.sqlname.length] = '\0';
```

Maintenant, la donnée de la colonne est stockée dans la variable `v`. Copier toutes les données dans les variables host, en inspectant `v.sqltype` pour connaître le type de la colonne.

```
            switch (v.sqltype) {
                int intval;
                double doubleval;
                unsigned long long int longlongval;

                case ECPGt_char:
                    memset(&var_buf, 0, sizeof(var_buf));
                    memcpy(&var_buf, sqldata, (sizeof(var_buf) <=
 sqllen ? sizeof(var_buf)-1 : sqllen));
                    break;

                case ECPGt_int: /* integer */
                    memcpy(&intval, sqldata, sqllen);
                    snprintf(var_buf, sizeof(var_buf), "%d",
 intval);
                    break;

                ...

                default:
                    ...
            }

            printf("%s = %s (type: %d)\n", name_buf, var_buf,
 v.sqltype);
        }
```

Fermer le curseur après avoir traité tous les enregistrements, et se déconnecter de la base de données.

```
    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;
```

Le programme dans son entier est visible dans Exemple 36.1.

Exemple 36.1. Programme de Démonstration SQLDA

```
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* descripteur pour la sortie */
sqlda_t *sqlda2; /* descripteur pour l'entrée */

EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d,
 pg_stat_database s WHERE d.oid=s.datid AND ( d.datname=? OR
 d.oid=? )";

    int intval;
    unsigned long long int longlongval;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO uptimedb AS con1 USER uptime;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;

    /* Créer une structure SQLDB pour let paramètres d'entrée */
    sqlda2 = (sqlda_t *)malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* a number of input variables */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *) &intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);

    /* Ouvrir un curseur avec les paramètres d'entrée. */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;

    while (1)
    {
        sqlda_t *cur_sqlda;

        /* Assigner le descripteur au curseur  */
```

```
        EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;

        for (cur_sqlda = sqlda1 ;
             cur_sqlda != NULL ;
             cur_sqlda = cur_sqlda->desc_next)
        {
            int i;
            char name_buf[1024];
            char var_buf[1024];

            /* Afficher toutes les colonnes d'un enregistrement. */
            for (i=0 ; i<cur_sqlda->sqld ; i++)
            {
                sqlvar_t v = cur_sqlda->sqlvar[i];
                char *sqldata = v.sqldata;
                short sqllen  = v.sqllen;

                strncpy(name_buf, v.sqlname.data,
v.sqlname.length);
                name_buf[v.sqlname.length] = '\0';

                switch (v.sqltype)
                {
                    case ECPGt_char:
                        memset(&var_buf, 0, sizeof(var_buf));
                        memcpy(&var_buf, sqldata,
(sizeof(var_buf)<=sqllen ? sizeof(var_buf)-1 : sqllen) );
                        break;

                    case ECPGt_int: /* integer */
                        memcpy(&intval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%d",
intval);
                        break;

                    case ECPGt_long_long: /* bigint */
                        memcpy(&longlongval, sqldata, sqllen);
                        snprintf(var_buf, sizeof(var_buf), "%lld",
longlongval);
                        break;

                    default:
                    {
                        int i;
                        memset(var_buf, 0, sizeof(var_buf));
                        for (i = 0; i < sqllen; i++)
                        {
                            char tmpbuf[16];
                            snprintf(tmpbuf, sizeof(tmpbuf), "%02x
", (unsigned char) sqldata[i]);
                            strncat(var_buf, tmpbuf,
sizeof(var_buf));
                        }
                    }
                        break;
                }
```

```
                printf("%s = %s (type: %d)\n", name_buf, var_buf,
 v.sqltype);
            }

            printf("\n");
        }
    }

    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

L'exemple suivant devrait ressembler à quelque chose comme ce qui suit (des nombres seront différents).

```
oid = 1 (type: 1)
datname = template1 (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = t (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl = {=c/uptime,uptime=CTc/uptime} (type: 1)
datid = 1 (type: 1)
datname = template1 (type: 1)
numbackends = 0 (type: 5)
xact_commit = 113606 (type: 9)
xact_rollback = 0 (type: 9)
blks_read = 130 (type: 9)
blks_hit = 7341714 (type: 9)
tup_returned = 38262679 (type: 9)
tup_fetched = 1836281 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)

oid = 11511 (type: 1)
datname = postgres (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = f (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl =  (type: 1)
datid = 11511 (type: 1)
datname = postgres (type: 1)
```

```
numbackends = 0 (type: 5)
xact_commit = 221069 (type: 9)
xact_rollback = 18 (type: 9)
blks_read = 1176 (type: 9)
blks_hit = 13943750 (type: 9)
tup_returned = 77410091 (type: 9)
tup_fetched = 3253694 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)
```

36.8. Gestion des Erreurs

Cett section explique comment vous pouvez traiter des conditions d'exception et des avertissements dans un programme SQL embarqué. Il y a deux fonctionnalités non-exclusives pour cela.

- Des fonctions de rappel (callbacks) peuvent être configurées pour traiter les conditions d'avertissement et d'erreur en utilisant la commande `WHENEVER`.
- Des informations détaillées à propos de l'erreur ou de l'avertissement peuvent être obtenues de la variable `sqlca`.

36.8.1. Mettre en Place des Callbacks

Une méthode simple pour intercepter des erreurs et des avertissements est de paramétrer des actions spécifiques à exécuter dès qu'une condition particulière se produit. En général :

```
EXEC SQL WHENEVER condition action;
```

`condition` peut être un des éléments suivants :

`SQLERROR`

L'action spécifiée est appelée dès qu'une erreur se produit durant l'exécution d'un ordre SQL.

`SQLWARNING`

L'action spécifiée est appelée dès qu'un avertissement se produit durant l'exécution d'un ordre SQL.

`NOT FOUND`

L'action spécifiée est appelée dès qu'un ordre SQL récupère ou affecte zéro enregistrement. (Cette condition n'est pas une erreur, mais vous pourriez être intéressé par un traitement spécial dans ce cas).

`action` peut être un des éléments suivants:

`CONTINUE`

Cela signifie en fait que la condition est ignorée. C'est le comportement par défaut.

`GOTO` *`label`*
`GO TO` *`label`*

Sauter au label spécifié (en utilisant un ordre `goto` C).

`SQLPRINT`

Affiche un message vers la sortie standard. C'est utile pour des programmes simples ou durant le prototypage. Le détail du message ne peut pas être configuré.

`STOP`

Appelle `exit(1)`, ce qui mettra fin au programme.

`DO BREAK`

Exécuter l'ordre C `break`. Cela ne devrait être utilisé que dans des boucles ou des ordres `switch`.

`DO CONTINUE`

Exécute l'instruction C `continue`. Ceci doit seulement être utilisé dans les instructions de boucle. Son exécution fait que le flot de contrôle est renvoyé au sommet de la boucle.

`CALL` *`name`* `(`*`args`*`)`
`DO` *`name`* `(`*`args`*`)`

Appelle la fonction C spécifiée avec les arguments spécifiés. (Son utilisation est différente des instructions `CALL` et `DO` dans la grammaire habituelle de PostgreSQL.)

Le standard SQL ne fournit que les actions `CONTINUE` et `GOTO` (and `GO TO`).

Voici un exemple de ce que pourriez vouloir utiliser dans un programme simple. Il affichera un message quand un avertissement se produit et tuera le programme quand une erreur se produit:

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

L'ordre `EXEC SQL WHENEVER` est une directive du préprocesseur SQL, pas un ordre SQL. L'action sur erreur ou avertissement qu'il met en place s'applique à tous les ordres SQL embarqués qui apparaissent après le point où le gestionnaire est mis en place, sauf si une autre action a été mise en place pour la même condition entre le premier `EXEC SQL WHENEVER` et l'ordre SQL entrainant la condition, quel que soit le déroulement du programme C. Par conséquent, aucun des extraits des deux programmes C suivants n'aura l'effet escompté :

```
/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}


/*
 * WRONG
```

```
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

36.8.2. sqlca

Pour une gestion plus approfondie des erreurs, l'interface SQL embarquée fournit une variable globale appelée `sqlca` (SQL communication area, ou zone de communication SQL) qui a la structure suivante :

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

(Dans un programme multi-threadé, chaque thread récupère automatiquement sa propre copie de `sqlca`. Ce fonctionnement est similaire à celui de la variable C globale `errno`.) `errno`.)

`sqlca` couvre à la fois les avertissements et les erreurs. Si plusieurs avertissements ou erreurs se produisent durant l'exécution d'un ordre, alors `sqlca` ne contiendra d'informations que sur le dernier.

Si aucune erreur ne s'est produite durant le dernier ordre SQL, `sqlca.sqlcode` vaudra 0 `sqlca.sqlstate` vaudra `"00000"`. Si un avertissement ou erreur s'est produit, alors `sqlca.sqlcode` sera négatif `sqlca.sqlstate` sera différent de `"00000"`. Une valeur positive de `sqlca.sqlcode` indique une condition sans gravité comme le fait que la dernière requête ait retourné zéro enregistrements. `sqlcode` et `sqlstate` sont deux différents schémas de code d'erreur ; les détails sont fournis plus bas.

Si le dernier ordre SQL a réussi, alors `sqlca.sqlerrd[1]` contient l'OID de la ligne traitée, si applicable, et `sqlca.sqlerrd[2]` contient le nombre d'enregistrements traités ou retournés, si applicable à la commande.

En cas d'erreur ou d'avertissement, `sqlca.sqlerrm.sqlerrmc` contiendra une chaine qui décrira une erreur. Le champ `sqlca.sqlerrm.sqlerrml` contiendra la longueur du message d'erreur qui

est stocké dans `sqlca.sqlerrm.sqlerrmc` (le résultat de `strlen()`, par réellement intéressant pour un programmeur C). Notez que certains messages sont trop longs pour tenir dans le tableau de taille fixe `sqlerrmc`; ils seront tronqués.

En cas d'avertissement, `sqlca.sqlwarn[2]` est positionné à `W`. (Dans tous les autres cas, il est positionné à quelque chose de différent de `W`.) Si `sqlca.sqlwarn[1]` est positionné à `W`, alors une valeur a été tronquée quand elle a été stockée dans une variable hôte. `sqlca.sqlwarn[0]` est positionné à `W` si n'importe lequel des autres éléments est positionné pour indiquer un avertissement.

Les champs `sqlcaid`, `sqlabc`, `sqlerrp`, et les éléments restants de `sqlerrd` et `sqlwarn` ne contiennent pour le moment aucune information utile.

La structure `sqlca` n'est pas définie dans le standard SQL, mais est implémentée dans plusieurs autres systèmes de base de données. Les définitions sont similaires dans leur principe, mais si vous voulez écrire des applications portables, vous devriez étudier les différentes implémentations de façon attentive.

Voici un exemple qui combine l'utilisation de `WHENEVER` et de `sqlca`, en affichant le contenu de `sqlca` quand une erreur se produit. Cela pourrait être utile pour déboguer ou prototyper des applications, avant d'installer un gestionnaire d'erreurs plus « user-friendly ».

```
EXEC SQL WHENEVER SQLERROR CALL print_sqlca();

void
print_sqlca()
{
    fprintf(stderr, "==== sqlca ====\n");
    fprintf(stderr, "sqlcode: %ld\n", sqlca.sqlcode);
    fprintf(stderr, "sqlerrm.sqlerrml: %d\n",
 sqlca.sqlerrm.sqlerrml);
    fprintf(stderr, "sqlerrm.sqlerrmc: %s\n",
 sqlca.sqlerrm.sqlerrmc);
    fprintf(stderr, "sqlerrd: %ld %ld %ld %ld %ld %ld\n",
 sqlca.sqlerrd[0],sqlca.sqlerrd[1],sqlca.sqlerrd[2],

 sqlca.sqlerrd[3],sqlca.sqlerrd[4],sqlca.sqlerrd[5]);
    fprintf(stderr, "sqlwarn: %d %d %d %d %d %d %d %d\n",
 sqlca.sqlwarn[0], sqlca.sqlwarn[1], sqlca.sqlwarn[2],

 sqlca.sqlwarn[3], sqlca.sqlwarn[4], sqlca.sqlwarn[5],

 sqlca.sqlwarn[6], sqlca.sqlwarn[7]);
    fprintf(stderr, "sqlstate: %5s\n", sqlca.sqlstate);
    fprintf(stderr, "===============\n");
}
```

Le résultat pourrait ressembler à ce qui suit (ici une erreur due à un nom de table mal saisi) :

```
==== sqlca ====
sqlcode: -400
sqlerrm.sqlerrml: 49
sqlerrm.sqlerrmc: relation "pg_databasep" does not exist on line 38
sqlerrd: 0 0 0 0 0 0
sqlwarn: 0 0 0 0 0 0 0 0
sqlstate: 42P01
===============
```

36.8.3. SQLSTATE contre SQLCODE

Les champs `sqlca.sqlstate` et `sqlca.sqlcode` sont deux schémas qui fournissent des codes d'erreurs. Les deux sont dérivés du standard SQL, mais `SQLCODE` a été marqué comme déprécié dans l'édition SQL-92 du standard, et a été supprimé des éditions suivantes. Par conséquent, les nouvelles applications ont fortement intérêt à utiliser `SQLSTATE`.

`SQLSTATE` est un tableau de cinq caractères. Les cinq caractères contiennent des chiffres ou des lettres en majuscule qui représentent les codes des différentes conditions d'erreur et d'avertissement. `SQLSTATE` a un schéma hiérarchique: les deux premiers caractères indiquent la classe générique de la condition, les trois caractères suivants indiquent la sous-classe de la condition générique. Un état de succès est indiqué par le code `00000`. Les codes `SQLSTATE` sont pour la plupart définis dans le standard SQL. Le serveur PostgreSQL supporte nativement les codes d'erreur `SQLSTATE`; par conséquent, un haut niveau de cohérence entre toutes les applications peut être obtenu en utilisant ce schéma de codes d'erreur. Pour plus d'informations voyez Annexe A.

`SQLCODE`, le schéma d'erreurs déprécié, est un entier simple. Une valeur de 0 indique le succès, une valeur positive indique un succès avec des informations supplémentaires, une valeur négative indique une erreur. Le standard SQL ne définit que la valeur positive +100, qui indique que l'ordre précédent a retourné ou affecté zéro enregistrement, et aucune valeur négative spécifique. par conséquent, ce schéma ne fournit qu'une piètre portabilité et n'a pas de hiérarchie de code d'erreurs. Historiquement, le processeur de SQL embarqué de PostgreSQL a assigné des valeurs spécifiques de `SQLCODE` pour son utilisation propre, qui sont listées ci-dessous avec leur valeur numérique et leur nom symbolique. Rappelez vous qu'ils ne sont pas portables vers d'autres implémentations SQL. Pour simplifier le portage des applications vers le schéma `SQLSTATE`, les valeurs `SQLSTATE` sont aussi listées. Il n'y a pas, toutefois, de correspondance un à un ou un à plusieurs entre les deux schémas (c'est en fait du plusieurs à plusieurs), vous devriez donc consulter la liste globale `SQLSTATE` dans Annexe A au cas par cas.

Voici les valeurs de `SQLCODE` assignées :

0 (`ECPG_NO_ERROR`)

Indique pas d'erreur. (SQLSTATE 00000)

100 (`ECPG_NOT_FOUND`)

C'est un état sans danger indiquant que la dernière commande a récupéré ou traité zéro enregistrements, ou que vous êtes au bout du curseur. (SQLSTATE 02000)

Quand vous bouclez sur un curseur, vous pourriez utiliser ce code comme façon de détecter quand arrêter la boucle, comme ceci:

```
while (1)
{
    EXEC SQL FETCH ... ;
    if (sqlca.sqlcode == ECPG_NOT_FOUND)
        break;
}
```

Mais `WHENEVER NOT FOUND DO BREAK` fait en fait cela en interne, il n'y a donc habituellement aucun avantage à écrire ceci de façon explicite.

-12 (`ECPG_OUT_OF_MEMORY`)

Indique que votre mémoire virtuelle est épuisée. La valeur numérique est définie comme `-ENOMEM`. (SQLSTATE YE001)

-200 (ECPG_UNSUPPORTED)

Indique que le préprocesseur a généré quelque chose que la librairie ne connait pas. Peut-être êtes vous en train d'utiliser des versions incompatibles du préprocesseur et de la librairie. (SQLSTATE YE002)

-201 (ECPG_TOO_MANY_ARGUMENTS)

Cela signifie que la commande a spécifié plus de variables hôte que la commande n'en attendait. (SQLSTATE 07001 or 07002)

-202 (ECPG_TOO_FEW_ARGUMENTS)

Cela signifie que la commande a spécifié moins de variables hôtes que la commande n'en attendait. (SQLSTATE 07001 or 07002)

-203 (ECPG_TOO_MANY_MATCHES)

Cela signifie que la requête a retourné pluiseurs enregistrements mais que l'ordre n'était capable d'en recevoir qu'un (par exemple parce que les variables spécifiées ne sont pas des tableaux. (SQLSTATE 21000)

-204 (ECPG_INT_FORMAT)

La variable hôte est du type `int` et la donnée dans la base de données est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un `int`. La librairie utilise `strtol()` pour cette conversion. (SQLSTATE 42804).

-205 (ECPG_UINT_FORMAT)

La variable hôte est du type `unsigned int` et la donnée dans la base de données est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un `unsigned int`. La librairie utilise `strtoul()` pour cette conversion. (SQLSTATE 42804).

-206 (ECPG_FLOAT_FORMAT)

La variable hôte est du type `float` et la donnée dans la base de données est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un `float`. La librairie utilise `strtod()` pour cette conversion. (SQLSTATE 42804).

-207 (ECPG_NUMERIC_FORMAT)

La variable hôte est du type `numeric` et la donnée dans la base de données est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un `numeric`. (SQLSTATE 42804).

-208 (ECPG_INTERVAL_FORMAT)

La variable hôte est du type `interval` et la donnée dans la base de données est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un `interval`. (SQLSTATE 42804).

-209 (ECPG_DATE_FORMAT)

La variable hôte est du type `date` et la donnée dans la base de données est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un `date`. (SQLSTATE 42804).

-210 (ECPG_TIMESTAMP_FORMAT)

La variable hôte est du type `timestamp` et la donnée dans la base de données est d'un type différent et contient une valeur qui ne peut pas être interprétée comme un `timestamp`. (SQLSTATE 42804).

-211 (`ECPG_CONVERT_BOOL`)

Cela signifie que la variable hôte est de type `bool` et que la donnée dans la base n'est ni `'t'` ni `'f'`. (SQLSTATE 42804)

-212 (`ECPG_EMPTY`)

L'ordre envoyé au serveur PostgreSQL était vide. (Cela ne peut normalement pas arriver dans un programme SQL embarqué, cela pourrait donc laisser supposer une erreur interne.) (SQLSTATE YE002)

-213 (`ECPG_MISSING_INDICATOR`)

Une valeur null a été retournée et aucune variable d'indicateur null n'a été fournie. (SQLSTATE 22002)

-214 (`ECPG_NO_ARRAY`)

Une variable ordinaire a été utilisée à un endroit qui nécessite un tableau. (SQLSTATE 42804)

-215 (`ECPG_DATA_NOT_ARRAY`)

La base a retourné une variable ordinaire à un endroir qui nécessite une variable de tableau. (SQLSTATE 42804)

-216 (`ECPG_ARRAY_INSERT`)

La valeur n'a pas pu être insérée dans le tableau. (SQLSTATE 42804)

-220 (`ECPG_NO_CONN`)

Le programme a essayé d'utiliser une connexion qui n'existe pas. (SQLSTATE 08003)

-221 (`ECPG_NOT_CONN`)

Le programme a essayé d'utiliser une connexion qui existe mais n'est pas ouverte. (C'est une erreur interne.) (SQLSTATE YE002)

-230 (`ECPG_INVALID_STMT`)

L'ordre que vous essayez d'exécuter n'a pas été préparé. (SQLSTATE 26000)

-239 (`ECPG_INFORMIX_DUPLICATE_KEY`)

Erreur de clé en doublon, violation de contrainte unique (mode de compatibilité Informix). (SQLSTATE 23505)

-240 (`ECPG_UNKNOWN_DESCRIPTOR`)

Le descripteur spécifié n'a pas été trouvé. L'ordre que vous essayez d'utiliser n'a pas été préparé. (SQLSTATE 33000)

-241 (`ECPG_INVALID_DESCRIPTOR_INDEX`)

L'index de descripteur spécifié était hors de portée. (SQLSTATE 07009)

-242 (`ECPG_UNKNOWN_DESCRIPTOR_ITEM`)

Un objet de descripteur invalide a été demandé. (C'est une erreur interne.) (SQLSTATE YE002)

-243 (`ECPG_VAR_NOT_NUMERIC`)

Durant l'exécution d'un ordre dynamique, la base a retourné une valeur numeric et la variable hôte n'était pas numeric. (SQLSTATE 07006)

-244 (`ECPG_VAR_NOT_CHAR`)

Durant l'exécution d'un ordre dynamique, la base a retourné une valeur non numeric et la variable hôte était numeric. (SQLSTATE 07006)

-284 (`ECPG_INFORMIX_SUBSELECT_NOT_ONE`)

Un résultat de la sous-requête n'était pas un enregistrement seul (mode de compatibilité Informix). (SQLSTATE 21000)

-400 (`ECPG_PGSQL`)

Une erreur causée par le serveur PostgreSQL. Le message contient le message d'erreur du serveur PostgreSQL.

-401 (`ECPG_TRANS`)

Le serveur PostgreSQL a signalé que nous ne pouvons pas démarrer, valider ou annuler la transaction. (SQLSTATE 08007)

-402 (`ECPG_CONNECT`)

La tentative de connexion à la base n'a pas réussi. (SQLSTATE 08001)

-403 (`ECPG_DUPLICATE_KEY`)

Erreur de clé dupliquée, violation d'une contrainte unique. (SQLSTATE 23505)

-404 (`ECPG_SUBSELECT_NOT_ONE`)

Un résultat de la sous-requête n'est pas un enregistrement unique. (SQLSTATE 21000)

-602 (`ECPG_WARNING_UNKNOWN_PORTAL`)

Un nom de curseur invalide a été spécifié. (SQLSTATE 34000)

-603 (`ECPG_WARNING_IN_TRANSACTION`)

Transaction en cours. (SQLSTATE 25001)

-604 (`ECPG_WARNING_NO_TRANSACTION`)

Il n'y a pas de transaction active (en cours). (SQLSTATE 25P01)

-605 (`ECPG_WARNING_PORTAL_EXISTS`)

Un nom de curseur existant a été spécifié. (SQLSTATE 42P03)

36.9. Directives de Préprocesseur

Plusieurs directives de préprocesseur sont disponibles, qui modifient comment le préprocesseur `ecpg` analyse et traite un fichier.

36.9.1. Inclure des Fichiers

Pour inclure un fichier externe dans votre fichier SQL embarqué, utilisez :

```
EXEC SQL INCLUDE filename;
EXEC SQL INCLUDE <filename>;
EXEC SQL INCLUDE "filename";
```

Le préprocesseur de SQL embarqué recherchera un fichier appelé *filename*.h, le préprocessera, et l'incluera dans la sortie C résultante. En conséquence de quoi, les ordres SQL embarqués dans le fichier inclus seront traités correctement.

Le préprocesseurs ecpg cherchera un fichier dans plusieurs répertoires dans l'ordre suivant :

- répertoire courant
- /usr/local/include
- Le répertoire d'inclusion de PostgreSQL, défini à la compilation (par exemple, /usr/local/pgsql/include)
- /usr/include

Mais quand EXEC SQL INCLUDE "*filename*" est utilisé, seul le répertoire courant est parcouru.

Dans chaque répertoire, le préprocesseur recherchera d'abord le nom de fichier tel que spécifié, et si non trouvé, rajoutera .h au nom de fichier et essaiera à nouveau (sauf si le nom de fichier spécifié a déjà ce suffixe).

Notez que EXEC SQL INCLUDE est *différent* de:

```
#include <filename.h>
```

parce que ce fichier ne serait pas soumis au préprocessing des commandes SQL. Naturellement, vous pouvez continuer d'utiliser la directive C #include pour inclure d'autres fichiers d'entête. files.

> **Note**
>
> Le nom du fichier à inclure est sensible à la casse, même si le reste de la commande EXEC SQL INCLUDE suit les règles normales de sensibilité à la casse de SQL.

36.9.2. Les Directives define et undef

Similaires aux directives #define qui sont connues en C, le SQL embarqué a un concept similaire :

```
EXEC SQL DEFINE name;
EXEC SQL DEFINE name value;
```

Vous pouvez donc définir un nom:

```
EXEC SQL DEFINE HAVE_FEATURE;
```

Et vous pouvez aussi définir des constantes:

```
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```

Utilisez `undef` pour supprimer une définition précédente :

```
EXEC SQL UNDEF MYNUMBER;
```

Bien sûr, vous pouvez continuer d'utiliser les versions C de `#define` et `#undef` dans votre programme SQL embarqué. La différence est le moment où vos valeurs définies sont évalues. Si vous utilisez `EXEC SQL DEFINE` alors la préprocesseur `ecpg` évalue les définition et substitiue les valeurs. Par exemple si vous écrivez :

```
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

alors `ecpg` fera d'emblée la substitution et votre compilateur C ne verra jamais aucun nom ou identifiant `MYNUMBER`. Notez que vous ne pouvez pas utiliser `#define` pour une constante que vous allez utiliser dans une requête SQL embarquée parce que dans ce cas le précomipilateur SQL embarqué n'est pas capable de voir cette déclaration.

Si plusieurs fichiers en entrée sont nommés sur la ligne de commande du pré-processeur `ecpg`, les effets de `EXEC SQL DEFINE` et `EXEC SQL UNDEF` ne se transmettent pas entre fichiers : chaque fichier commence uniquement avec les symboles définis par les options `-D` sur la ligne de commande.

36.9.3. Directives ifdef, ifndef, elif, else, et endif

Vous pouvez utiliser les directives suivantes pour compiler des sections de code sous condition :

`EXEC SQL ifdef` *`nom`*`;`

Vérifie un *`nom`* et traite les lignes suivante si *`nom`* a été créé avec `EXEC SQL define` *`nom`*.

`EXEC SQL ifndef` *`nom`*`;`

Vérifie un *`nom`* et traite les lignes suivantes si *`nom`* *n'a pas* été défini avec `EXEC SQL define` *`nom`*.

`EXEC SQL elif` *`name`*`;`

Commence une section alternative optionnelle après une directive `EXEC SQL ifdef` *`name`* ou `EXEC SQL ifndef` *`name`*. Il peut y avoir autant de sections `elif` que nécessaire. Les lignes suivant un `elif` seront traitées si *`name`* a été défini *et* qu'aucune section précédente pour la même construction `ifdef`/`ifndef`...`endif` n'a été traitée.

`EXEC SQL else;`

Commence une section finale, optionnelle, après une directive `EXEC SQL ifdef` *`name`* ou `EXEC SQL ifndef` *`name`*. Les lignes suivantes seront traitées si aucune section précédente de la même construction `ifdef`/`ifndef`...`endif` n'a été traitée.

`EXEC SQL endif;`

Termine une construction `ifdef`/`ifndef`...`endif`. Les lignes suivantes sont traitées normalement.

Les constructions `ifdef`/`ifndef`...`endif` peuvent être imbriquées, jusqu'à 127 niveaux.

Cet exemple compilera exactement une des trois commandes `SET TIMEZONE` :

```
EXEC SQL ifdef TZVAR;
EXEC SQL SET TIMEZONE TO TZVAR;
EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL endif;
```

36.10. Traiter des Programmes en SQL Embarqué

Maintenant que vous avez une idée de comment rédiger des programmes SQL embarqué en C, vous voudrez probablement savoir comment les compiler. Avant de les compiler, vous passez le fichier dans le préprocesseur C SQL embarqué, qui convertira les ordres SQL que vous avez utilisé vers des appels de fonction spéciaux. Ces fonctions récupèrent des données à partir de leurs arguments, effectuent les commands SQL en utilisant l'interface libpq, et met le résultat dans les arguments spécifiés comme sortie.

Le programme préprocesseur est appelé `ecpg` et fait partie d'une installation normale de PostgreSQL. Les programmes SQL embarqués sont typiquement nommés avec une extension `.pgc`. Si vous avez un fichier de programme appelé `prog1.pgc`, vous pouvez le préprocesseur en appelant simplement :

```
ecpg prog1.pgc
```

Cela créera un fichier appelé `prog1.c`. Si vos fichiers d'entrée ne suivent pas les règles de nommage suggérées, vous pouvez spécifier le fichier de sortie explicitement en utilisant l'option `-o`.

Le fichier préprocessé peut être compilé normalement, par exemple :

```
cc -c prog1.c
```

Les fichiers sources C générés incluent les fichiers d'entête de l'installation PostgreSQL, donc si vous avez installé PostgreSQL à un endroit qui n'est pas recherché par défaut, vous devrez ajouter une option comme `-I/usr/local/pgsql/include` à la ligne de commande de compilation.

Pour lier un programme SQL embarqué, vous aurez besoin d'inclure la librairie `libecpg`, comme ceci :

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

De nouveau, vous pourriez avoir besoin d'ajouter une option comme `-L/usr/local/pgsql/lib` à la ligne de commande.

Vous pouvez utiliser `pg_config` ou `pkg-config` avec le package `libecpg` pour obtenir les chemins de votre installation.

Si vous gérez le processus de compilation d'un projet de grande taille en utilisant make, il serait pratique d'inclure la règle implicite suivante à vos makefiles :

```
ECPG = ecpg
```

```
%.c: %.pgc
        $(ECPG) $<
```

La syntaxe complète de la commande `ecpg` est détaillée dans ecpg.

La librairie ecpg est thread-safe par défaut. Toutefois, vous aurez peut-être besoin d'utiliser des options de ligne de commande spécifiques aux threads pour compiler votre code client.

36.11. Fonctions de la Librairie

La librairie `libecpg` contient principalement des fonctions « cachées » qui sont utilisée pour implémenter les fonctionnalités exprimées par les commandes SQL embarquées. Mais il y a quelques fonctions qui peuvent être appelées directement de façon utile. Notez que cela rendra votre code non-portable.

- `ECPGdebug(int` *`on`*`, FILE *`*`stream`*`)` active les traces de déboggage si appelé avec une valeur différente de 0 en premier argument. La trace contient tous les ordres SQL avec toutes les variables d'entrées insérées, et les résultats du serveur PostgreSQL. Cela peut être très utile quand vous êtes à la recherche d'erreurs dans vos ordres SQL.

  > **Note**
  >
  > Sous Windows, si les librairies ecpg et les applications sont compilées avec des options différentes, cet appel de fonction fera planter l'application parce que la représentation interne des pointeurs `FILE` diffère. En particulier, les options multithreaded/single-threaded, release/debug, et static/dynamic doivent être les mêmes pour la librairie et toutes les applications qui l'utilisent.

- `ECPGget_PGconn(const char *`*`nom_connexion`*`)` retourne le descripteur de connexion à la base de données de la librairie identifié par le nom fourni. Si *`nom_connexion`* est positionné à `NULL`, le descripteur de connexion courant est retourné. Si aucun descripteur de connexion ne peut être identifié, la fonction retourne `NULL`. Le descripteur de connexion retourné peut être utilisé pour appeler toute autre fonction de la libpq, si nécessaire.

  > **Note**
  >
  > C'est une mauvaise idée de manipuler les descripteurs de connexion à la base de donnée faits par ecpg directement avec des routines de libpq.

- `ECPGtransactionStatus(const char *`*`nom_connexion`*`)` retourne l'état de la transaction courante de la connexion identifiée par *`nom_connexion`*. Voyez Section 34.2 et la fonction de la libpq `PQtransactionStatus` pour les détails à propos des codes d'état retournés.

- `ECPGstatus(int` *`lineno`*`, const char*` *`nom_connexion`*`)` retourne vrai si vous êtes connecté à une base et faux sinon. *`nom_connexion`* peut valoir `NULL` si une seule connexion est utilisée.

36.12. Large Objects

Les Large objects ne sont pas supportés directement par ECPG, mais les application ECPG peuvent manipuler des large objects au moyen des fonctions large objects de la libpq, en obtenant l'objet `PGconn` nécessaire par l'appel de la fonction `ECPGget_PGconn`. (Toutefois, l'utilisation directe de

la fonction `ECPGget_PGconn` et la manipulation d'objets `PGconn` devrait être effectuée de façon très prudente, et idéalement pas mélangée avec d'autres appels à la base par ECPG.)

Pour plus de détails à propos de `ECPGget_PGconn`, voyez Section 36.11. Pour les informations sur les fonctions d'interfaçage avec les large objects, voyez Chapitre 35.

Les fonctions large object doivent être appelées dans un bloc de transaction, donc quand autocommit est à off, les commandes `BEGIN` doivent être effectuées explicitement.

Exemple 36.2 montre un programme de démonstration sur les façons de créer, écrire et lire un large object dans une application ECPG.

Exemple 36.2. Programme ECPG Accédant à un Large Object

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <libpq/libpq-fs.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    PGconn     *conn;
    Oid         loid;
    int         fd;
    char        buf[256];
    int         buflen = 256;
    char        buf2[256];
    int         rc;

    memset(buf, 1, buflen);

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;

    conn = ECPGget_PGconn("con1");
    printf("conn = %p\n", conn);

    /* créer */
    loid = lo_create(conn, 0);
    if (loid < 0)
        printf("lo_create() failed: %s", PQerrorMessage(conn));

    printf("loid = %d\n", loid);

    /* test d'écriture */
    fd = lo_open(conn, loid, INV_READ|INV_WRITE);
    if (fd < 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_write(conn, fd, buf, buflen);
    if (rc < 0)
        printf("lo_write() failed\n");
```

```
    rc = lo_close(conn, fd);
    if (rc < 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* read test */
    fd = lo_open(conn, loid, INV_READ);
    if (fd < 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_read(conn, fd, buf2, buflen);
    if (rc < 0)
        printf("lo_read() failed\n");

    rc = lo_close(conn, fd);
    if (rc < 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* vérifier */
    rc = memcmp(buf, buf2, buflen);
    printf("memcmp() = %d\n", rc);

    /* nettoyer */
    rc = lo_unlink(conn, loid);
    if (rc < 0)
        printf("lo_unlink() failed: %s", PQerrorMessage(conn));

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

36.13. Applications C++

ECPG a un support limité pour les applications C++. Cette section décrit certains pièges.

Le préprocesseur `ecpg` prend un fichier d'entrée écrit en C (ou quelque chose qui ressemble à du C) et des commandes SQL embarquées, et convertit les commandes SQL embarquées dans des morceaux de langage, et finalement génère un fichier `.c`. Les déclarations de fichiers d'entête des fonctions de librairie utilisées par les morceaux de langage C que génère `ecpg` sont entourées de blocs `extern "C" { ... }` quand ils sont utilisés en C++, ils devraient donc fonctionner de façon transparente en C++.

En général, toutefois, le préprocesseur `ecpg` ne comprend que le C; il ne gère pas la syntaxe spéciale et les mots réservés du langage C++. Par conséquent, du code SQL embarqué écrit dans du code d'une application C++ qui utilise des fonctionnalités compliquées spécifiques au C++ pourrait ne pas être préprocessé correctement ou pourrait ne pas fonctionner comme prévu.

Une façon sûre d'utiliser du code SQL embarqué dans une application C++ est de cacher les appels à ECPG dans un module C, que le code C++ de l'application appelle pour accéder à la base, et lier ce module avec le reste du code C++. Voyez Section 36.13.2 à ce sujet.

36.13.1. Portée des Variable Hôtes

Le préprocesseur `ecpg` comprend la porté des variables C. Dans le langage C, c'est plutôt simple parce que la portée des variables ne dépend que du bloc de code dans lequel elle se trouve. En C++,

par contre, les variables d'instance sont référencées dans un bloc de code différent de la position de déclaration, ce qui fait que le préprocesseur `ecpg` ne comprendra pas la portée des variables d'instance.

Par exemple, dans le cas suivant, le préprocesseur `ecpg` ne peut pas trouver de déclaration pour la variable `dbname` dans la méthode `test`, une erreur va donc se produire.

```
class TestCpp
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;
}

void Test::test()
{
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

TestCpp::~TestCpp()
{
    EXEC SQL DISCONNECT ALL;
}
```

Ce code génèrera une erreur comme celle qui suit :

```
ecpg test_cpp.pgc
test_cpp.pgc:28: ERROR: variable "dbname" is not declared
```

Pour éviter ce problème de portée, la méthode `test` pourrait être modifiée pour utiliser une variable locale comme stockage intermédiaire. Mais cette approche n'est qu'un mauvais contournement, parce qu'elle rend le code peu élégant et réduit la performance.

```
void TestCpp::test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char tmp[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :tmp;
    strlcpy(dbname, tmp, sizeof(tmp));
```

```
    printf("current_database = %s\n", dbname);
}
```

36.13.2. Développement d'application C++ avec un Module Externe en C

Si vous comprenez ces limitations techniques du préprocesseur `ecpg` en C++, vous arriverez peut-être à la conclusion que lier des objets C et C++ au moment du link pour permettre à des applications C++ d'utiliser les fonctionnalités d'ECPG pourrait être mieux que d'utiliser des commandes SQL embarquées dans du code C++ directement. Cette section décrit un moyen de séparer des commandes SQL embarquées du code d'une application C++ à travers un exemple simple. Dans cet exemple, l'application est implémentée en C++, alors que C et ECPG sont utilisés pour se connecter au serveur PostgreSQL.

Trois types de fichiers devront être créés: un fichier C (`*.pgc`), un fichier d'entête, et un fichier C++ :

`test_mod.pgc`

Un module de routines pour exécuter des commandes SQL embarquées en C. Il sera converti en `test_mod.c` par le préprocesseur.

```
#include "test_mod.h"
#include <stdio.h>

void
db_connect()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;
}

void
db_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

void
db_disconnect()
{
    EXEC SQL DISCONNECT ALL;
}
```

`test_mod.h`

Un fichier d'entête avec les déclarations des fonctions du module C (`test_mod.pgc`). Il est inclus par `test_cpp.cpp`. Ce fichier devra avoir un bloc `extern "C"` autour des déclarations, parce qu'il sera lié à partir d'un module C++.

```
#ifdef __cplusplus
extern "C" {
#endif

void db_connect();
void db_test();
void db_disconnect();

#ifdef __cplusplus
}
#endif
```

`test_cpp.cpp`

Le code principal de l'application, incluant la routine `main`, et dans cet exemple une classe C++.

```
#include "test_mod.h"

class TestCpp
{
  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    db_connect();
}

void
TestCpp::test()
{
    db_test();
}

TestCpp::~TestCpp()
{
    db_disconnect();
}

int
main(void)
{
    TestCpp *t = new TestCpp();

    t->test();
    return 0;
}
```

Pour construire l'application, procédez comme suit. Convertissez `test_mod.pgc` en `test_mod.c` en lançant `ecpg`, et générez `test_mod.o` en compilant `test_mod.c` avec le compilateur C:

```
ecpg -o test_mod.c test_mod.pgc
```

```
cc -c test_mod.c -o test_mod.o
```

Puis, générez `test_cpp.o` en compilant `test_cpp.cpp` avec le compilateur C++:

```
c++ -c test_cpp.cpp -o test_cpp.o
```

Finalement, liez ces objets, `test_cpp.o` et `test_mod.o`, dans un exécutable, en utilisant le compilateur C++:

```
c++ test_cpp.o test_mod.o -lecpg -o test_cpp
```

36.14. Commandes SQL Embarquées

Cette section décrit toutes les commandes SQL qui sont spécifiques au SQL embarqué. Consultez aussi les commandes SQL listées dans Commandes SQL, qui peuvent aussi être utilisée dans du SQL embarqué, sauf mention contraire.

ALLOCATE DESCRIPTOR

ALLOCATE DESCRIPTOR — alloue une zone de descripteur SQL

Synopsis

```
ALLOCATE DESCRIPTOR name
```

Description

`ALLOCATE DESCRIPTOR` alloue une nouvelle zone de descripteur SQL nommée, qui pourra être utilisée pour échanger des données entre le serveur PostgreSQL et le programme hôte.

Les zones de descripteur devraient être libérées après utilisation avec la commande `DEALLOCATE DESCRIPTOR`.

Paramètres

name

Un nom de descripeur SQL, sensible à la casse. Il peut être un identifiant SQL ou une variable hôte.

Exemple

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
```

Compatibilité

`ALLOCATE DESCRIPTOR` est spécifié par le standard SQL.

Voir aussi

DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

CONNECT

CONNECT — établit une connexion à la base de données

Synopsis

```
CONNECT TO connection_target [ AS nom_connexion ]
 [ USER connection_user_name ]
CONNECT TO DEFAULT
CONNECT connection_user_name
DATABASE connection_target
```

Description

La commande `CONNECT` établit une connexion entre le client et le serveur PostgreSQL.

Paramètres

connection_target

connection_target spécifie le serveur cible de la connexion dans une des formes suivantes :

[*database_name*][@*host*][:*port*]

Se connecter par TCP/IP

unix:postgresql://*host* [:*port*] / [*database_name*] [?*connection_option*]

Se connecter par une socket de domaine Unix

tcp:postgresql://*host*[:*port*]/[*database_name*][?*connection_option*]

Se connecter par TCP/IP

constante de type chaîne SQL

contient une valeur d'une des formes précédentes

variable hôte

variable hôte du type `char[]` ou `VARCHAR[]` contenant une valeur d'une des formes précédentes

connection_name

Un identifiant optionnel pour la connexion, afin qu'on puisse y faire référence dans d'autres commandes. Cela peut être un identifiant SQL ou une variable hôte.

connection_user

Le nom d'utilisateur pour une connexion à la base de données.

Ce paramètre peut aussi spécifier un nom d'utilisateur et un mot de passe, en utilisant une des formes *user_name*/*password*, *user_name* `IDENTIFIED BY` *password*, or *user_name* `USING` *password*.

Nom d'utilisateur et mot de passe peuvent être des identifiants SQL, des constantes de type chaîne, ou des variables hôtes.

`DEFAULT`

Utiliser tous les paramètres de connexion par défaut, comme défini par libpq.

Exemples

Voici plusieurs variantes pour spécifier des paramètres de connexion :

```
EXEC SQL CONNECT TO "connectdb" AS main;
EXEC SQL CONNECT TO "connectdb" AS second;
EXEC SQL CONNECT TO "unix:postgresql://200.46.204.71/connectdb" AS
 main USER connectuser;
EXEC SQL CONNECT TO "unix:postgresql://localhost/connectdb" AS main
 USER connectuser;
EXEC SQL CONNECT TO 'connectdb' AS main;
EXEC SQL CONNECT TO 'unix:postgresql://localhost/connectdb' AS main
 USER :user;
EXEC SQL CONNECT TO :db AS :id;
EXEC SQL CONNECT TO :db USER connectuser USING :pw;
EXEC SQL CONNECT TO @localhost AS main USER connectdb;
EXEC SQL CONNECT TO REGRESSDB1 as main;
EXEC SQL CONNECT TO AS main USER connectdb;
EXEC SQL CONNECT TO connectdb AS :id;
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb;
EXEC SQL CONNECT TO connectdb AS main;
EXEC SQL CONNECT TO connectdb@localhost AS main;
EXEC SQL CONNECT TO tcp:postgresql://localhost/ USER connectdb;
EXEC SQL CONNECT TO tcp:postgresql://localhost/connectdb USER
 connectuser IDENTIFIED BY connectpw;
EXEC SQL CONNECT TO tcp:postgresql://localhost:20/connectdb USER
 connectuser IDENTIFIED BY connectpw;
EXEC SQL CONNECT TO unix:postgresql://localhost/ AS main USER
 connectdb;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb AS main
 USER connectuser;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER
 connectuser IDENTIFIED BY "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER
 connectuser USING "connectpw";
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb?
connect_timeout=14 USER connectuser;
```

Voici un programme exemple qui illustre l'utilisation de variables hôtes pour spécifier des paramètres de connexion :

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    char *dbname     = "testdb";    /* nom de la base */
    char *user       = "testuser";  /* nom d'utilisateur pour la
 connexion */
    char *connection = "tcp:postgresql://localhost:5432/testdb";
                                    /* chaîne de connexion */
    char ver[256];                  /* buffer pour contenir la
 chaîne de version */
```

```
EXEC SQL END DECLARE SECTION;

    ECPGdebug(1, stderr);

    EXEC SQL CONNECT TO :dbname USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    EXEC SQL CONNECT TO :connection USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    return 0;
}
```

Compatibilité

`CONNECT` est spécifié dans le standard SQL, mais le format des paramètres de connexion est spécifique à l'implémentation.

Voir aussi

DISCONNECT, SET CONNECTION

DEALLOCATE DESCRIPTOR

DEALLOCATE DESCRIPTOR — désalloue une zone de descripteur SQL

Synopsis

```
DEALLOCATE DESCRIPTOR name
```

Description

`DEALLOCATE DESCRIPTOR` désalloue une zone de descripteur SQL nommée.

Paramètres

name

Le nom du descripteur qui va être désalloué. Il est sensible à la casse. Cela peut-être un identifiant SQL ou une variable hôte.

Exemples

```
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

Compatibilité

`DEALLOCATE DESCRIPTOR` est spécifié dans le standard SQL

See Also

ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

DECLARE

DECLARE — définit un curseur

Synopsis

```
DECLARE nom_curseur [ BINARY ] [ ASENSITIVE | INSENSITIVE ]
 [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT } HOLD ]
 FOR nom_prepare
DECLARE nom_curseur [ BINARY ] [ ASENSITIVE | INSENSITIVE ]
 [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

Description

`DECLARE` déclare un cusreur pour itérer sur le jeu de résultat d'une requête préparée. Cette commande a une sémantique légèrement différente de celle de l'ordre SQL direct `DECLARE`. Là ou ce dernier exécute une requête et prépare le jeu de résultat pour la récupération, cette commande SQL embarquée se contente de déclarer un nom comme « variable de boucle » pour itérer sur le résultat d'une requête; l'exécution réelle se produit quand le curseur est ouvert avec la commande `OPEN`.

Paramètres

`nom_curseur`

Un nom de curseur, sensible à la casse. Cela peut être un identifiant SQL ou une variable hôte.

`nom_prepare`

Le nom de l'une requête préparée, soit comme un identifiant SQL ou comme une variable hôte.

`query`

Une commande SELECT ou VALUES qui fournira les enregistrements que le curseur devra retourner.

Pour la signification des options du curseur, voyez DECLARE.

Exemples

Exemples de déclaration de curseur pour une requête :

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table;
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T;
EXEC SQL DECLARE cur1 CURSOR FOR SELECT version();
```

Un exemple de déclaration de curseur pour une requête préparée :

```
EXEC SQL PREPARE stmt1 AS SELECT version();
EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

Compatibilité

`DECLARE` est spécifié dans le standard SQL.

Voir aussi

OPEN, CLOSE, DECLARE

DECLARE STATEMENT

DECLARE STATEMENT — declare un identifiant d'ordre SQL

Synopsis

```
EXEC SQL [ AT connection_name ] DECLARE statement_name STATEMENT
```

Description

`DECLARE STATEMENT` déclare un identifiant d'ordre SQL. L'identifiant d'ordre SQL peut être associé avec une connexion. Quand l'identifiant est utilisé pour des commandes SQL dynamiques, les commandes sont exécutées en utilisant la connexion associée. L'espace de nommage de la déclaration est une unité précompilée, et de multiples déclaration au même identifiant d'ordre SQL n'est pas permis. Notez que si le précompilateur s'exécute en mode compatible Informix et qu'un ordre SQL est déclaré, alors "database" ne peut être utilisée comme nom de curseur.

Paramètres

`connection_name`

Un nom de connexion à une base de données établie par la commande `CONNECT`.

La clause AT peut être omise, mais de telles commandes n'ont pas de significations.

`statement_name`

Le nom de l'identifiant d'ordre SQL, soit un identifiant SQL ou une variable hôte.

Notes

L'association n'est valide que si la déclaration est physiquement placée en haut de la commande dynamique.

Exemples

```
EXEC SQL CONNECT TO postgres AS con1;
EXEC SQL AT con1 DECLARE sql_stmt STATEMENT;
EXEC SQL DECLARE cursor_name CURSOR FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :dyn_string;
EXEC SQL OPEN cursor_name;
EXEC SQL FETCH cursor_name INTO :column1;
EXEC SQL CLOSE cursor_name;
```

Compatibilité

`DECLARE STATEMENT` est une extension du standard SQL, mais peut être utilisé avec d'autres moteurs de bases de données.

Voir aussi

CONNECT, DECLARE, OPEN

DESCRIBE

DESCRIBE — obtient des informations à propos d'une requête préparée ou d'un jeu de résultat

Synopsis

```
DESCRIBE [ OUTPUT ] nom_prepare USING [ SQL ]
 DESCRIPTOR nom_descripteur
DESCRIBE [ OUTPUT ] nom_prepare INTO [ SQL ]
 DESCRIPTOR nom_descripteur
DESCRIBE [ OUTPUT ] nom_prepare INTO nom_sqlda
```

Description

`DESCRIBE` récupère des informations sur les métadonnées à propos des colonnes de résultat contenues dans une requête préparée, sans déclencher la récupération d'un enregistrement.

Paramètres

nom_prepare

Le nom d'une requête préparée. Cela peut être un identifiant SQL ou une variable hôte.

nom_descripteur

Un nom de descripteur. Il est sensible à la casse. Cela peut être un identifiant SQL ou une variable hôte.

nom_sqlda

Le nom d'une variable SQLDA.

Exemples

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME;
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

Compatibilité

`DESCRIBE` est spécifié dans le standard SQL.

Voir aussi

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

DISCONNECT

DISCONNECT — met fin à une connexion de base de données

Synopsis

```
DISCONNECT nom_connexion
DISCONNECT [ CURRENT ]
DISCONNECT ALL
```

Description

`DISCONNECT` ferme une connexion (ou toutes les connexions) à la base de données.

Paramètres

`nom_connexion`

Une connexion à la base établie par la commande `CONNECT`.

`CURRENT`

Ferme la connexion « courante », qui est soit la connexion ouverte la plus récemment, soit la connexion spécifiée par la commande `SET CONNECTION`. C'est aussi la valeur par défaut si aucun argument n'est donné à la commande `DISCONNECT`.

`ALL`

Ferme toutes les connexions ouvertes.

Exemples

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb AS con3 USER testuser;

    EXEC SQL DISCONNECT CURRENT;  /* ferme con3         */
    EXEC SQL DISCONNECT ALL;      /* ferme con2 and con1 */

    return 0;
}
```

Compatibilité

`DISCONNECT` est spécifié dans le standard SQL.

Voir aussi

CONNECT, SET CONNECTION

EXECUTE IMMEDIATE

EXECUTE IMMEDIATE — prépare et exécute un ordre dynamique

Synopsis

```
EXECUTE IMMEDIATE chaine
```

Description

`EXECUTE IMMEDIATE` prépare et exécute immédiatement un ordre SQL spécifié dynamiquement, sans récupérer les enregistrements du résultat.

Paramètres

chaine

Une chaîne littérale ou une variable hôte contenant l'ordre SQL à exécuter.

Notes

Dans une utilisation typique, la *chaîne* est une variable hôte référençant une chaîne contenant un ordre SQL dynamiquement construit. Le cas d'une chaîne littérale n'est pas très utile ; vous pouvez aussi bien écrire l'ordre SQL directement, sans saisir en plus `EXECUTE IMMEDIATE`.

Si vous utilisez une chaîne littérale, gardez à l'esprit que tout guillemet que vous souhaitez inclure dans l'ordre SQL doit être écrit avec des caractères d'échappement octal (`\042`) et non pas les idiomes habituels C `\"`. En effet, la chaîne est dans une section `EXEC SQL`, donc l'analyseur grammatical ECPG l'analyse suivant les règles SQL et non pas les règles C. Toute barre oblique inverse (« backslash ») sera plus tard gérée en suivant les règles C ; mais `\"` provoque une erreur de syntaxe immédiate parce qu'il est vu comme une fin de litéral.

Exemples

Voici un exemple qui exécute un ordre `INSERT` en utilisant `EXECUTE IMMEDIATE` et une variable hôte appelée `commande`:

```
sprintf(commande, "INSERT INTO test (name, amount, letter) VALUES
 ('db: ''r1'''', 1, 'f')");
EXEC SQL EXECUTE IMMEDIATE :commande;
```

Compatibilité

`EXECUTE IMMEDIATE` est spécifié dans le standard SQL.

GET DESCRIPTOR

GET DESCRIPTOR — récupère des informations d'une zone de descripteur SQL

Synopsis

```
GET DESCRIPTOR nom_descripteur :cvariable
 = element_entete_descripteur
GET DESCRIPTOR nom_descripteur VALUE numero_colonne :cvariable
 = element_descripteur [, ... ]
```

Description

`GET DESCRIPTOR` récupère des informations à propos du résultat d'une requête à partir d'une zone de descripteur SQL et les stocke dans des variables hôtes. Une zone de descripteur est d'ordinaire remplie en utilisant `FETCH` ou `SELECT` avant d'utiliser cette commande pour transférer l'information dans des variables du langage hôte.

Cette commande a deux formes : la première forme récupère les objets de « l'entête » du descripteur, qui s'appliquent au jeu de résultat dans son ensemble. Un exemple est le nombre d'enregistrements. La seconde forme, qui nécessite le nombre de colonnes comme paramètre additionnel, récupère des informations sur une colonne particulière. Par exemple, le type de la colonne, et la valeur réelle de la colonne.

Paramètres

nom_descripteur

Un nom de descripteur.

element_entete_descripteur

Un marqueur identifiant de quel objet de l'entête récupérer l'information. Seul `COUNT`, qui donne le nombre de colonnes dans le résultat, est actuellement supporté.

numero_colonne

Le numéro de la colonne à propos duquel on veut récupérer des informations. Le compte commence à 1.

element_descripteur

Un marqueur identifiant quel élément d'information récupérer d'une colonne. Voir Section 36.7.1 pour une liste d'objets supportés.

cvariable

Une variable hôte qui recevra les données récupérées de la zone de descripteur.

Exemples

Un exemple de récupération du nombre de colonnes dans un résultat:

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
```

Un exemple de récupération de la longueur des données de la première colonne:

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length =
 RETURNED_OCTET_LENGTH;
```

Un exemple de récupération des données de la seconde colonne en tant que chaîne:

```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA;
```

Voici un exemple pour la procédure complète, lors de l'exécution de `SELECT current_database();` et montrant le nombre de colonnes, la longueur de la colonne, et la données de la colonne:

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int  d_count;
    char d_data[1024];
    int  d_returned_octet_length;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL ALLOCATE DESCRIPTOR d;

    /* Déclarer un curseur, l'ouvrir, et assigner un descripteur au
 curseur */
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database();
    EXEC SQL OPEN cur;
    EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;

    /* Récupérer le nombre total de colonnes */
    EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
    printf("d_count                 = %d\n", d_count);

    /* Récupérer la longueur d'une colonne retournée */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length =
 RETURNED_OCTET_LENGTH;
    printf("d_returned_octet_length = %d\n",
 d_returned_octet_length);

    /* Récupérer la colonne retournée en tant que chaîne */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_data = DATA;
    printf("d_data                  = %s\n", d_data);

    /* Fermer */
    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

Quand l'exemple est exécuté, son résultat ressemble à ceci:

```
d_count                  = 1
d_returned_octet_length = 6
d_data                   = testdb
```

Compatibilité

`GET DESCRIPTOR` est spécifié dans le standard SQL.

Voir aussi

ALLOCATE DESCRIPTOR, SET DESCRIPTOR

OPEN

OPEN — ouvre un curseur dynamique

Synopsis

```
OPEN nom_curseur
OPEN nom_curseur USING valeur [, ... ]
OPEN nom_curseur USING SQL DESCRIPTOR nom_descripteur
```

Description

`OPEN` ouvre un curseur et optionnellement lie (bind) les valeurs aux conteneurs (placeholders) dans la déclaration du curseur. Le curseur doit préalablement avoir été déclaré avec la commande `DECLARE`. L'exécution d'`OPEN` déclenche le début de l'exécution de la requête sur le serveur.

Paramètres

`nom_curseur`

Le nom du curseur à ouvrir. Cela peut être un identifiant SQL ou une variable hôte.

`valeur`

Une valeur à lier au placeholder du curseur. Cela peut être une constante SQL, une variable hôte, ou une variable hôte avec indicateur.

`nom_descripteur`

Le nom du descripteur contenant les valeurs à attacher aux placeholders du curseur. Cela peut être un identifiant SQL ou une variable hôte.

Exemples

```
EXEC SQL OPEN a;
EXEC SQL OPEN d USING 1, 'test';
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc;
EXEC SQL OPEN :curname1;
```

Compatibilité

`OPEN` est spécifiée dans le standard SQL.

Voir aussi

DECLARE, CLOSE

PREPARE

PREPARE — prépare un ordre pour son exécution

Synopsis

```
PREPARE nom_préparé FROM chaîne
```

Description

PREPARE prépare l'exécution d'un ordre spécifié dynamiquement sous forme d'une chaîne. C'est différent des ordres SQL directs PREPARE, qui peuvent aussi être utilisés dans des programmes embarqués. La commande EXECUTE peut être utilisée pour exécuter les deux types de requêtes préparées.

Paramètres

nom_prepare

Un identifiant pour la requête préparée.

chaîne

Une chaîne littérale ou une variable hôte contenant un ordre SQL préparable, soit SELECT, INSERT, UPDATE ou DELETE. Utilisez des points d'interrogation (?) pour les valeurs de paramètres à fournir à l'exécution.

Notes

Dans une utilisation type, la *chaîne* est une référence de variable à une chaîne contenant l'ordre SQL dynamiquement construit. Le cas d'une chaîne littérale n'est pas très utile ; vous pouvez tout autant écrire un ordre SQL direct PREPARE.

Si vous utilisez une chaîne littérale, gardez à l'esprit que tout guillemet que vous souhaitez inclure dans l'ordre SQL doit être écrit avec des caractères d'échappement octal (\042) et non pas les idiomes habituels C \". En effet, la chaîne est dans une section EXEC SQL, donc l'analyseur grammatical ECPG l'analyse suivant les règles SQL et non pas les règles C. Toute barre oblique inverse (« backslash ») sera plus tard gérée en suivant les règles C ; mais \" provoque une erreur de syntaxe immédiate parce qu'il est vu comme une fin de litéral.

Exemples

```
char *stmt = "SELECT * FROM test1 WHERE a = ? AND b = ?";

EXEC SQL ALLOCATE DESCRIPTOR outdesc;
EXEC SQL PREPARE foo FROM :stmt;

EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL
 DESCRIPTOR outdesc;
```

Compatibilité

PREPARE est spécifié dans le standard SQL.

Voir aussi

CONNECT, DISCONNECT

SET AUTOCOMMIT

SET AUTOCOMMIT — configure le comportement de l'autocommit pour la session en cours

Synopsis

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

Description

`SET AUTOCOMMIT` configure le comportement de l'autocommit pour la session en cours de la base de données. Par défaut, les programmes SQL embarqués ne sont *pas* en mode autocommit, donc `COMMIT` doit être exécuté explicitement quand il est voulu. Cette commande modifie le mode autocommit pour la session, où chaque requête individuelle est validée implicitement.

Compatibilité

`SET AUTOCOMMIT` est une extension de PostgreSQL ECPG.

SET CONNECTION

SET CONNECTION — sélectionne une connexion de base

Synopsis

```
SET CONNECTION [ TO | = ] nom_connexion
```

Description

`SET CONNECTION` configure la connexion à la base de données « actuelle », qui est celle que toutes les commandes utilisent, sauf en cas de surcharge.

Paramètres

`nom_connexion`

Un nom de connexion établi par la commande `CONNECT`.

`CURRENT`

Configure la connexion comme la connexion actuelle (donc rien n'arrive).

Exemples

```
EXEC SQL SET CONNECTION TO con2;
EXEC SQL SET CONNECTION = con1;
```

Compatibility

`SET CONNECTION` est indiqué dans le standard SQL.

Voir aussi

CONNECT, DISCONNECT

SET DESCRIPTOR

SET DESCRIPTOR — positionne des informations dans une zone de descripteur SQL

Synopsis

```
SET DESCRIPTOR nom_descripteur objet_entete_descripteur = valeur
SET DESCRIPTOR nom_descripteur VALUE numero objet_descripteur
 = valeur [, ...]
```

Description

`SET DESCRIPTOR` remplit une zone de descripteur SQL de valeurs. La zone de descripteur est habituellement utilisée pour lier les paramètres lors d'une exécution de requête préparée

Cette commande a deux formes: la première forme s'applique à l' « entête » du descripteur, qui est indépendant des données spécifiques. La seconde forme assigne des valeurs aux données, identifiées par un numéro.

Paramètres

nom_descripteur

Un nom de descripteur.

objet_entete_descripteur

Un identifiant pour spécifier quelle information de l'entête est concernée. Seul `COUNT`, qui sert à indiquer le nombre de descripteurs, est supporté pour le moment.

number

Le numéro de l'objet du descripteur à modifier. Le compte commence à 1.

objet_descripteur

Un identifiant spécifiant quelle information du descripteur est concernée. Voyez Section 36.7.1 pour une liste des identifiants supportés.

valeur

Une valeur à stocker dans l'objet descripteur. Cela peut être une constante SQL ou une variable hôte.

Exemples

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1;
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val1, DATA =
 'some string';
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA
 = :val2;
```

Compatibilité

`SET DESCRIPTOR` est spécifié dans le standard SQL.

Voir aussi

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

TYPE

TYPE — définit un nouveau type de données

Synopsis

```
TYPE nom_type IS ctype
```

Description

La commande `TYPE` définit un nouveau type C. C'est équivalent à mettre un `typedef` dans une section declare.

Cette commande n'est reconnue que quand `ecpg` est exécutée avec l'option `-c`.

Paramètres

`nom_type`

Le nom du nouveau type. Ce doit être un nom de type valide en C.

`ctype`

Une spécification de type C.

Exemples

```
EXEC SQL TYPE customer IS
    struct
    {
        varchar name[50];
        int     phone;
    };

EXEC SQL TYPE cust_ind IS
    struct ind
    {
        short   name_ind;
        short   phone_ind;
    };

EXEC SQL TYPE c IS char reference;
EXEC SQL TYPE ind IS union { int integer; short smallint; };
EXEC SQL TYPE intarray IS int[AMOUNT];
EXEC SQL TYPE str IS varchar[BUFFERSIZ];
EXEC SQL TYPE string IS char[11];
```

Voici un programme de démonstration qui utilise `EXEC SQL TYPE`:

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;

EXEC SQL TYPE tt IS
    struct
```

```
    {
        varchar v[256];
        int     i;
    };

EXEC SQL TYPE tt_ind IS
    struct ind {
        short   v_ind;
        short   i_ind;
    };

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    tt t;
    tt_ind t_ind;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;

    EXEC SQL SELECT current_database(), 256 INTO :t:t_ind LIMIT 1;

    printf("t.v = %s\n", t.v.arr);
    printf("t.i = %d\n", t.i);

    printf("t_ind.v_ind = %d\n", t_ind.v_ind);
    printf("t_ind.i_ind = %d\n", t_ind.i_ind);

    EXEC SQL DISCONNECT con1;

    return 0;
}
```

La sortie de ce programme ressemble à ceci:

```
t.v = testdb
t.i = 256
t_ind.v_ind = 0
t_ind.i_ind = 0
```

Compatibilité

La commande TYPE est une extension PostgreSQL.

VAR

VAR — définit une variable

Synopsis

```
VAR nomvar IS ctype
```

Description

La commande VAR assigne un nouveau type de données C à une variable hôte. La variable hôte doit être précédemment déclarée dans une section de déclaration.

Paramètres

nomvar

Un nom de variable C.

ctype

Une spécification de type C.

Exemples

```
Exec sql begin declare section;
short a;
exec sql end declare section;
EXEC SQL VAR a IS int;
```

Compatibilité

La commande VAR est une extension PostgreSQL.

WHENEVER

WHENEVER — spécifie l'action à effectuer quand un ordre SQL entraîne le déclenchement d'une classe d'exception

Synopsis

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } action
```

Description

Définit un comportement qui sera appelé dans des cas spéciaux (enregistrements non trouvés, avertissements ou erreurs SQL) dans le résultat de l'exécution SQL.

Paramètres

Voyez Section 36.8.1 pour une description des paramètres.

Exemples

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER NOT FOUND DO CONTINUE;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLWARNING DO warn();
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL WHENEVER SQLERROR CALL print2();
EXEC SQL WHENEVER SQLERROR DO handle_error("select");
EXEC SQL WHENEVER SQLERROR DO sqlnotice(NULL, NONO);
EXEC SQL WHENEVER SQLERROR DO sqlprint();
EXEC SQL WHENEVER SQLERROR GOTO error_label;
EXEC SQL WHENEVER SQLERROR STOP;
```

Une application classique est l'utilisation de `WHENEVER NOT FOUND BREAK` pour gérer le bouclage sur des jeux de résultats:

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '',
 false); EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(),
 'hoge', 256;
    EXEC SQL OPEN cur;

    /* quand la fin du jeu de résultat est atteinte, sortir de la
 boucle */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
```

```
    {
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;
        ...
    }

    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

Compatibilité

`WHENEVER` est spécifié dans le standard SQL, mais la plupart des actions sont des extensions PostgreSQL.

36.15. Mode de Compatibilité Informix

`ecpg` peut être exécuté dans un mode appelé *mode de compatibilité Informix*. Si ce mode est actif, il essaie de se comporté comme s'il était le précompilateur Informix pour Informix E/SQL. En gros, cela va vous permettre d'utiliser le signe dollar au lieu de la primitive `EXEC SQL` pour fournir des commandes SQL embarquées:

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

> **Note**
>
> Il ne doit par y avoir d'espace entre le `$` et la directive de préprocesseur qui le suit, c'est à dire `include`, `define`, `ifdef`, etc. Sinon, le préprocesseur comprendra le mot comme une variable hôte.

Il y a deux modes de compatibilité: `INFORMIX`, `INFORMIX_SE`

Quand vous liez des programmes qui sont dans ce mode de compatibilité, rappelez vous de lier avec `libcompat` qui est fournie avec ECPG.

En plus du liant syntaxique expliqué précédemment, le mode de compatibilité Informix porte d'ESQL vers ECPG quelques fonctions pour l'entrée, la sortie et la transformation des données, ainsi que pour le SQL embarqué.

Le mode de compatibilité Informix est fortement connecté à la librairie pgtypeslib d'ECPG. pgtypeslib met en correspondance les types de données SQL et les types de données du programme hôte C et la plupart des fonctions additionnelles du mode de compatibilité Informix vous permettent de manipuler ces types C des programmes hôtes. Notez toutefois que l'étendue de cette compatibilité est limitée. Il n'essaie pas de copier le comportement d'Informix; il vous permet de faire plus ou mois les mêmes opérations et vou fournit des fonctions qui ont le même nom et ont à la base le même comportement,

mais ce n'est pas un produit de remplacement transparent si vous utilisez Informix à l'heure actuelle. De plus, certains types de données sont différents. Par exemple, les types datetime et interval de PostgreSQL ne savent pas traiter des ranges comme par exemple `YEAR TO MINUTE`, donc vous n'aurez pas de support pour cela dans ECPG non plus.

36.15.1. Types supplémentaires

Le pseudo-type "string" spécifique à Informix pour stocker des chaînes de caractères ajustées à droite est maintenant supporté dans le mode Informix sans avoir besoin d'utiliser `typedef`. En fait, en mode Informix, ECPG refuse de traiter les fichiers sources qui contiennent `typedef untype string;`

```
EXEC SQL BEGIN DECLARE SECTION;
string userid; /* cette variable contient des données ajustées */
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH MYCUR INTO :userid;
```

36.15.2. Ordres SQL Embarqués Supplémentaires/ Manquants

`CLOSE DATABASE`

Cet ordre ferme la connexion courante. En fait, c'est un synonyme du `DISCONNECT CURRENT` d'ECPG:

```
$CLOSE DATABASE;                /* ferme la connexion courante
 */
EXEC SQL CLOSE DATABASE;
```

`FREE nom_curseur`

En raison des différences sur la façon dont ECPG fonctionne par rapport à l'ESQL/C d'Informix (c'est à dire quelles étapes sont purement des transformations grammaticales et quelles étapes s'appuient sur la librairie sous-jacente), il n'y a pas d'ordre `FREE nom_curseur` dans ECPG. C'est parce que, dans ECPG, `DECLARE CURSOR` ne génère pas un appel de fonction à la librairie qui utilise le nom du curseur. Ce qui implique qu'il n'y a pas à gérer les curseurs SQL à l'exécution dans la librairie ECPG, seulement dans le serveur PostgreSQL.

`FREE nom_requete`

`FREE nom_requete` est un synonyme de `DEALLOCATE PREPARE nom_requete`.

36.15.3. Zones de Descripteurs SQLDA Compatibles Informix

Le mode de compatibilité Informix supporte une structure différente de celle décrite dans Section 36.7.2. Voyez ci-dessous:

```
struct sqlvar_compat
{
    short   sqltype;
```

```
    int     sqllen;
    char   *sqldata;
    short  *sqlind;
    char   *sqlname;
    char   *sqlformat;
    short   sqlitype;
    short   sqlilen;
    char   *sqlidata;
    int     sqlxid;
    char   *sqltypename;
    short   sqltypelen;
    short   sqlownerlen;
    short   sqlsourcetype;
    char   *sqlownername;
    int     sqlsourceid;
    char   *sqlilongdata;
    int     sqlflags;
    void   *sqlreserved;
};

struct sqlda_compat
{
    short  sqld;
    struct sqlvar_compat *sqlvar;
    char   desc_name[19];
    short  desc_occ;
    struct sqlda_compat *desc_next;
    void  *reserved;
};

typedef struct sqlvar_compat    sqlvar_t;
typedef struct sqlda_compat     sqlda_t;
```

Les propriétés globales sont:

`sqld`

Le nombre de champs dans le descripteur `SQLDA`.

`sqlvar`

Un pointeur vers les propriétés par champ.

`desc_name`

Inutilisé, rempli d'octets à zéro.

`desc_occ`

La taille de la structure allouée.

`desc_next`

Un pointeur vers la structure SQLDA suivante si le jeu de résultat contient plus d'un enregistrement.

`reserved`

Pointeur inutilisé, contient NULL. Gardé pour la compatibilité Informix.

Les propriétés par champ sont ci-dessous, elles sont stockées dans le tableau `sqlvar`:

`sqltype`

Type du champ. Les constantes sont dans `sqltypes.h`

`sqllen`

La longueur du champ de données.

`sqldata`

Un pointeur vers le champ de données. Ce pointeur est de type `char*`, la donnée pointée par lui est en format binaire. Par exemple:

```
int intval;

switch (sqldata->sqlvar[i].sqltype)
{
    case SQLINTEGER:
        intval = *(int *)sqldata->sqlvar[i].sqldata;
        break;
  ...
}
```

`sqlind`

Un pointeur vers l'indicateur NULL. Si retourné par DESCRIBE ou FETCH alors c'est toujours un pointeur valide. Si utilisé comme valeur d'entrée pour `EXECUTE ... USING sqlda;` alors une valeur de pointeur NULL signifie que la valeur pour ce champ est non nulle. Sinon, un pointeur valide et `sqlitype` doivent être positionnés correctement. Par exemple:

```
if (*(int2 *)sqldata->sqlvar[i].sqlind != 0)
    printf("value is NULL\n");
```

`sqlname`

Le nom du champ. Chaîne terminée par 0.

`sqlformat`

Réservé dans Informix, valeurs de `PQfformat` pour le champ.

`sqlitype`

Type de l'indicateur de données NULL. C'est toujours SQLSMINT quand les données sont retournées du serveur. Quand la `SQLDA` est utilisée pour une requête paramétrique, la donnée est traitée en fonction du type de donnée positionné.

`sqlilen`

Longueur de l'indicateur de données NULL.

`sqlxid`

Type étendu du champ, résultat de `PQftype`.

`sqltypename`
`sqltypelen`
`sqlownerlen`
`sqlsourcetype`
`sqlownername`
`sqlsourceid`
`sqlflags`
`sqlreserved`

Inutilisé.

`sqlilongdata`

C'est égal à `sqldata` si `sqllen` est plus grand que 32nbsp;ko.

Par exemple:

```
EXEC SQL INCLUDE sqlda.h;

    sqlda_t        *sqlda; /* Ceci n'a pas besoin d'être dans la
 DECLARE SECTION embarquée */

    EXEC SQL BEGIN DECLARE SECTION;
    char *prep_stmt = "select * from table1";
    int i;
    EXEC SQL END DECLARE SECTION;

    ...

    EXEC SQL PREPARE mystmt FROM :prep_stmt;

    EXEC SQL DESCRIBE mystmt INTO sqlda;

    printf("# of fields: %d\n", sqlda->sqld);
    for (i = 0; i < sqlda->sqld; i++)
      printf("field %d: \"%s\"\n", sqlda->sqlvar[i]->sqlname);

    EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
    EXEC SQL OPEN mycursor;
    EXEC SQL WHENEVER NOT FOUND GOTO out;

    while (1)
    {
      EXEC SQL FETCH mycursor USING sqlda;
    }

    EXEC SQL CLOSE mycursor;

    free(sqlda); /* La structure principale doit être totalement
 libérée par free()
                  * sqlda and sqlda->sqlvar sont dans une seule
 zone allouée */
```

Pour plus d'informations, voyez l'entête `sqlda.h` et le test de non-régression `src/interfaces/ecpg/test/compat_informix/sqlda.pgc`.

36.15.4. Fonctions Additionnelles

`decadd`

Ajoute deux valeurs décimales.

```
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

La fonction reçoit un poiteur sur la première opérande de type decimal (`arg1`), un pointeur sur la seconde opérande de type decimal (`arg2`) et un pointeur sur la valeur de type decimal qui contiendra la somme (`sum`). En cas de succès, la fonction retourne 0. `ECPG_INFORMIX_NUM_OVERFLOW` est retourné en cas de dépassement et `ECPG_INFORMIX_NUM_UNDERFLOW` en cas de soupassement. -1 est retourné pour les autres échecs et `errno` est positionné au nombre correspondant `errno` de pgtypeslib. pgtypeslib.

`deccmp`

Compare deux variables de type decimal.

```
int deccmp(decimal *arg1, decimal *arg2);
```

La fonction reçoit un pointeur vers la première valeur decimal (`arg1`), un pointeur vers la seconde valeur decimal (`arg2`) et retourne une valeur entière qui indique quelle elle la plus grosse valeur.

- 1, si la valeur pointée par `arg1` est plus grande que celle pointée par `arg2`.
- -1 si la valeur pointée par `arg1` est plus petite que la valeur pointée par `arg2`.
- 0 si les deux valeurs pointées par `arg1` et `arg2` sont égales.

`deccopy`

Copie une valeur decimal.

```
void deccopy(decimal *src, decimal *target);
```

La fonction reçoit un pointeur vers la valeur decimal qui doit être copiée comme premier argument (`src`) et un pointeur vers la structure de type décimale cible (`target`) comme second argument.

`deccvasc`

Convertit une valeur de sa représentation ASCII vers un type decimal.

```
int deccvasc(char *cp, int len, decimal *np);
```

La fonction reçoit un pointeur vers une chaîne qui contient la représentation chaîne du nombre à convertir (`cp`) ainsi que sa longueur `len`. `np` est un pointeur vers la valeur decimal dans laquelle sauver le résultat de l'opération.

Voici quelques formats valides: `-2`, `.794`, `+3.44`, `592.49E07` ou `-32.84e-4`.

La fonction retourne 0 en cas de succès. Si un dépassement ou un soupassement se produisent, `ECPG_INFORMIX_NUM_OVERFLOW` ou `ECPG_INFORMIX_NUM_UNDERFLOW` est retourné.

Si la représentation ASCII n'a pas pu être interprétée, `ECPG_INFORMIX_BAD_NUMERIC` est retourné ou `ECPG_INFORMIX_BAD_EXPONENT` si le problème s'est produit lors de l'analyse de l'exposant.

`deccvdbl`

Convertit une valeur de type double vers une valeur de type decimal.

```
int deccvdbl(double dbl, decimal *np);
```

La fonction reçoit la variable de type double qui devrait être convertie comme premier argument (`dbl`). Comme second argument (`np`), la fonction reçoit un pointeur vers la variable decimal qui recevra le résultat de l'opération.

La fonction retourne 0 en cas de succès et une valeur négative si la conversion a échoué.

`deccvint`

Convertit une valeur de type int vers une valeur de type decimal.

```
int deccvint(int in, decimal *np);
```

La fonction reçoit la variable de type int à convertir comme premier argument (`in`). Comme second argument (`np`), la fonction reçoit un pointeur vers la variable decimal qui recevra le résultat de l'opération.

La fonction retourne 0 en cas de succès et une valeur négative si la conversion a échoué.

`deccvlong`

Convertit une valeur de type long vers une valeur de type decimal.

```
int deccvlong(long lng, decimal *np);
```

La fonction reçoit la variable de type long à convertir comme premier argument (`lng`). Comme second argument (`np`), la fonction reçoit un pointeur vers la variable decimal qui recevra le résultat de l'opération.

La fonction retourne 0 en cas de succès et une valeur négative si la conversion a échoué.

`decdiv`

Divise deux variables de type decimal.

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

La fonction reçoit des pointeurs vers les deux variables qui sont le premier (`n1`) et le second (`n2`) opérandes et calcule `n1`/`n2`. `result` est un pointeur vers la variable qui recevra le résultat de l'opération.

En cas de succès, 0 est retourné, et une valeur négative si la division échoue. En cas de dépassement ou de soupassement, la fonction retourne `ECPG_INFORMIX_NUM_OVERFLOW` ou `ECPG_INFORMIX_NUM_UNDERFLOW` respectivement. Si une tentative de division par zéro se produit, la fonction retourne `ECPG_INFORMIX_NUM_OVERFLOW`.

`decmul`

Multiplie deux valeurs decimal.

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

La fonction reçoit des pointeurs vers les deux variables qui sont le premier (`n1`) et le second (`n2`) opérandes et calcule `n1`/`n2`. `result` est un pointeur vers la variable qui recevra le résultat de l'opération.

En cas de succès, 0 est retourné, et une valeur négative si la division échoue. En cas de dépassement ou de soupassement, la fonction retourne `ECPG_INFORMIX_NUM_OVERFLOW` ou `ECPG_INFORMIX_NUM_UNDERFLOW` respectivement.

`decsub`

Soustrait une valeur decimal d'une autre.

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

La fonction reçoit des pointeurs vers les deux variables qui sont le premier (`n1`) et le second (`n2`) opérandes et calcule `n1`/`n2`. `result` est un pointeur vers la variable qui recevra le résultat de l'opération.

En cas de succès, 0 est retourné, et une valeur négative si la division échoue. En cas de dépassement ou de soupassement, la fonction retourne `ECPG_INFORMIX_NUM_OVERFLOW` ou `ECPG_INFORMIX_NUM_UNDERFLOW` respectivement.

`dectoasc`

Convertit une variable de type decimal vers sa représentation ASCII sous forme de chaîne C char*.

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

La fonction reçoit un pointeur vers une variable de type decimal (`np`) qu'elle convertit vers sa représentation textuelle. `cp` est le tampon qui devra contenir le résultat de l'opération. Le paramètre `right` spécifie combien de chiffres après la virgule doivent être inclus dans la sortie. Le résultat sera arrondi à ce nombre de chiffres décimaux. Positionner `right` à -1 indique que tous les chiffres décimaux disponibles devraient être inclus dans la sortie. Si la longueur du tampon de sortie, qui est indiquée par `len` n'est pas suffisante pour contenir toute la représentation en incluant le caractère NUL final, seul un caractère unique `*` est stocké dans le résultat, et -1 est retourné.

La fonction retourne -1 si le tampon `cp` était trop petit ou `ECPG_INFORMIX_OUT_OF_MEMORY` si plus de mémoire n'était disponible.

`dectodbl`

Convertit une variable de type decimal vers un double.

```
int dectodbl(decimal *np, double *dblp);
```

La fonction reçoit un pointeur vers la valeur decimal à convertir (`np`) et un pointeur vers la variable double qui devra recevoir le résultat de l'opération (`dblp`).

La fonction retourne 0 en cas de succès et une valeur négative si la conversion a échoué.

`dectoint`

Convertit une variable de type decimal vers un integer.

```
int dectoint(decimal *np, int *ip);
```

La fonction reçoit un pointeur vers la valeur decimal à convertir (`np`) et un pointeur vers la variable integer qui devra recevoir le résultat de l'opération (`ip`).

La fonction retourne 0 en cas de succès et une valeur négative si la conversion a échoué. Si un dépassement s'est produit, `ECPG_INFORMIX_NUM_OVERFLOW` est retourné.

Notez que l'implémentation d'ECPG diffère de celle d'Informix. Informix limite un integer entre -32767 et 32767, alors que la limite de l'implémentation d'ECPG dépend de l'architecture (`-INT_MAX .. INT_MAX`).

`dectolong`

Convertit une variable de type decimal vers un long integer.

```
int dectolong(decimal *np, long *lngp);
```

La fonction reçoit un pointeur vers la valeur decimal à convertir (`np`) et un pointeur vers la variable long qui devra recevoir le résultat de l'opération (`lngp`).

La fonction retourne 0 en cas de succès et une valeur négative si la conversion a échoué. Si un dépassement s'est produit, `ECPG_INFORMIX_NUM_OVERFLOW` est retourné.

Notez que l'implémentation d'ECPG diffère de celle d'Informix. Informix limite un integer entre --2,147,483,647 à 2,147,483,647 alors que la limite de l'implémentation d'ECPG dépend de l'architecture (`-LONG_MAX .. LONG_MAX`).

`rdatestr`

Convertit une date vers une chaîne char* C.

```
int rdatestr(date d, char *str);
```

La fonction reçoit deux arguments, le premier est la date à convertir (`d`) et le second est un pointeur vers la chaîne cible. Le format de sortie est toujours `yyyy-mm-dd`, vous aurez donc à allouer au moins 11 octets (en incluant le terminateur NUL) pour la chaîne.

La fonction retourne 0 en cas de succès et une valeur négative si la conversion a échoué.

Notez que l'implémentation d'ECPG diffère de celle de Informix. Dans Informix le format peut être modifié par le positionnement de variable d'enregistrement. Dans ECPG par contre, vous ne pouvez pas changer le format de sortie.

`rstrdate`

Convertit la représentation textuelle d'une date.

```
int rstrdate(char *str, date *d);
```

La fonction reçoit la représentation textuelle d'une date à convertir (`str`) et un pointeur vers une variable de type date (`d`). Cette fonction ne vous permet pas de fournir un masque de formatage. Il utilise le format par défaut d'Informix qui est `mm/dd/yyyy`. En interne, cette fonction est implémentée au travers de `rdefmtdate`. Par conséquent, `rstrdate` n'est pas plus rapide et si vous avez le choix, vous devriez opter pour `rdefmtdate`, qui vous permet de spécifier le masque de formatage explicitement.

La fonction retourne les mêmes valeurs que `rdefmtdate`.

`rtoday`

Récupère la date courante.

```
void rtoday(date *d);
```

La fonction reçoit un poiteur vers une variable de type date (`d`) qu'elle positionne à la date courante.

En interne, cette fonction utilise la fonction `PGTYPESdate_today`.

`rjulmdy`

Extrait les valeurs pour le jour, le mois et l'année d'une variable de type date.

```
int rjulmdy(date d, short mdy[3]);
```

La fonction reçoit la date `d` et un pointeur vers un tableau de 3 entiers courts `mdy`. Le nom de la variable indique l'ordre séquentiel: `mdy[0]` contiendra le numéro du mois, `mdy[1]` contiendra le numéro du jour, et `mdy[2]` contiendra l'année.

La fonction retourne toujours 0 pour le moment.

En interne, cette fonction utilise la fonction `PGTYPESdate_julmdy`.

`rdefmtdate`

Utilise un masque de formatage pour convertir une chaîne de caractère vers une valeur de type date.

```
int rdefmtdate(date *d, char *fmt, char *str);
```

La fonction reçoit un pointeur vers une valeur date qui devra contenir le résultat de l'opération (`d`), le masque de formatage à utiliser pour traiter la date (`fmt`) et la chaîne de caractère char* C qui contient la représentation textuelle de la date (`str`). La représentation textuelle doit correspondre au masque de formatage. La fonction n'analyse qu'en ordre séquentiel et recherche les littéraux `yy` ou `yyyy` qui indiquent la position de l'année, `mm` qui indique la position du mois et `dd` qui indique la position du jour.

La fonction retourne les valeurs suivantes:

- 0 - La fonction s'est terminée avec succès.
- `ECPG_INFORMIX_ENOSHORTDATE` - La date ne contient pas de délimiteur entre le jour, le mois et l'année. Dans ce cas, la chaîne en entrée doit faire exactement 6 ou 8 caractères, mais ce n'est pas le cas.

- ECPG_INFORMIX_ENOTDMY - La chaîne de formatage n'indique pas correctement l'ordre séquentiel de l'année, du mois, et du jour.
- ECPG_INFORMIX_BAD_DAY - La chaîne d'entrée ne contient pas de jour valide.
- ECPG_INFORMIX_BAD_MONTH - La chaîne d'entrée ne contient pas de mois valide.
- ECPG_INFORMIX_BAD_YEAR - La chaîne d'entrée ne contient pas d'année valide.

En interne, cette fonction est implémentée en utilisant la fonction PGTYPESdate_defmt_asc. Voyez la référence à cet endroi pour la table d'exemples.

rfmtdate

Convertit une variable de type date vers sa représentation textuelle en utilisant un masque de formatage.

```
int rfmtdate(date d, char *fmt, char *str);
```

La fonction reçoit une date à convertir (d), le masque de formatage (fmt) et la chaîne qui contiendra la représentation textuelle de la date (str).

La fonction retourne 0 en cas de succès et une valeur négative

En interne, cette fonction utilise la fonction PGTYPESdate_fmt_asc, voyez la référence pour des exemples.

rmdyjul

Crée une valeur date à partir d'un tableau de 3 entiers courts qui spécifient le jour, le mois et l'année de la date.

```
int rmdyjul(short mdy[3], date *d);
```

La fonction reçoit le tableau des 3 entiers courst (mdy) et un pointeur vers une variable de type date qui contiendra le résultat de l'opération.

La fonction retourne toujours 0 à l'heure actuelle.

En interne la fonction est implémentée en utilisant la fonction PGTYPESdate_mdyjul.

rdayofweek

Retourne un nombre représentant le jour de la semaine pour une valeur de date.

```
int rdayofweek(date d);
```

La fonction reçoit la variable date d comme seul argument et retourne un entier qui indique le jour de la semaine pour cette date.

- 0 - Dimanche
- 1 - Lundi
- 2 - Mardi
- 3 - Mercredi

- 4 - Jeudi
- 5 - Vendredi
- 6 - Samedi

En interne, cette fonction est implémentée en utilisant la fonction `PGTYPESdate_dayofweek`.

`dtcurrent`

Récupère le timestamp courant.

```
void dtcurrent(timestamp *ts);
```

La fonction récupère le timestamp courant et l'enregistre dans la variable timestamp vers laquelle `ts` pointe.

`dtcvasc`

Convertit un timestamp de sa représentation textuelle vers une variable timestamp.

```
int dtcvasc(char *str, timestamp *ts);
```

La fonction reçoit la chaîne à traiter (`str`) et un pointeur vers la variable timestamp qui contiendra le résultat de l'opération (`ts`).

La fonction retourne 0 en cas de succès et une valeur négative

En interne, cette fonction utilise la fonction `PGTYPEStimestamp_from_asc`. Voyez la référence pour un tableau avec des exemples de formats.

`dtcvfmtasc`

Convertit un timestamp de sa représentation textuelle vers une variable timestamp en utilisant un masque de formatage.

```
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

La fonction reçoit la chaîne à traiter (`inbuf`), le masque de formatage à utiliser (`fmtstr`) et un pointeur vers la variable timestamp qui contiendra le résultat de l'opération (`dtvalue`).

Cette fonction est implémentée en utilisant la fonction `PGTYPEStimestamp_defmt_asc`. Voyez la documentation à cet endroit pour la liste des spécificateurs de formats qui peuvent être utilisés.

La fonction retourne 0 en cas de succès et une valeur négative

`dtsub`

Soustrait un timestamp d'un autre et retourne une variable de type interval.

```
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

La fonction soustrait la variable timestamp vers laquelle `ts2` pointe de la variable timestamp vers laquelle `ts1` pointe et stockera le résultat dans la variable intervalle vers laquelle `iv` pointe.

En cas de succès, la fonction retourne 0, et une valeur négative si une erreur s'est produite.

`dttoasc`

Convertit une variable timestamp vers une chaîne char* C.

```
int dttoasc(timestamp *ts, char *output);
```

La fonction reçoit un pointeur vers une variable timestamp à convertir (`ts`) et la chaîne qui devra contenir le résultat de l'opération (`output`). Elle convertit `ts` vers sa représentation textuelle comme spécifié par le standard SQL, qui est `YYYY-MM-DD HH:MM:SS`.

En cas de succès, la fonction retourne 0, et une valeur négative si une erreur s'est produite.

`dttofmtasc`

Convertit une variable timestamp vers un char* C en utilisant un masque de formatage.

```
int dttofmtasc(timestamp *ts, char *output, int str_len, char
 *fmtstr);
```

La fonction reçoit un pointeur vers le timestamp à convertir comme premier argument (`ts`), un pointeur vers le tampon de sortie (`output`), la longueur maximale qui a été allouée pour le tampon de sortie (`str_len`) et le masque de formatage à utiliser pour la conversion (`fmtstr`).

En cas de succès, la fonction retourne 0, et une valeur négative si une erreur s'est produite.

En interne, cette fonction utilise la fonction `PGTYPEStimestamp_fmt_asc`. Voyez la référence pour des informations sur les spécifications de masque de formatage qui sont utilisables.

`intoasc`

Convertit une variable interval en chaîne char* C.

```
int intoasc(interval *i, char *str);
```

La fonction reçoit un pointeur vers la variable interval à convertir (`i`) et la chaîne qui contiendra le résultat de l'opération (`str`). Elle convertit `i` vers sa représentation textuelle suivant le standard SQL, qui est `YYYY-MM-DD HH:MM:SS`.

En cas de succès, la fonction retourne 0, et une valeur négative si une erreur s'est produite.

`rfmtlong`

Convertit une valeur long integer vers sa représentation textuelle en utilisant un masque de formatage.

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

La fonction reçoit la valeur long `lng_val`, le masque de formatage `fmt` et un pointeur vers le tampon de sortie `outbuf`. Il convertit la valeur long vers sa représentation textuelle en fonction du masque de formatage.

Le masque de formatage peut être composé des caractères suivants de spécification:

- * (asterisk) - si cette position était blanc sans cela, mettez y un astérisque.
- & (ampersand) - si cette position était blanc sans cela, mettez y un zéro.
- # - transforme les zéros initiaux en blancs.
- < - justifie à gauche le nombre dans la chaîne.
- , (virgule) - Groupe les nombres de 4 chiffres ou plus en groupes de 3 chiffres séparés par des virgules.
- . (point) - Ce caractère sépare la partie entière du nombre de sa partie fractionnaire.
- - (moins) - le signe moins apparaît si le nombre est négatif.
- + (plus) - le signe plus apparaît si le nombre est positif.
- (- ceci remplace le signe moins devant une valeur négative. Le signe moins n'apparaîtra pas.
-) - Ce caractère remplace le signe moins et est affiché après la valeur négative.
- $ - le symbole monétaire.

`rupshift`

Passe une chaîne en majuscule.

```
void rupshift(char *str);
```

La fonction reçoit un pointeur vers une chaîne et convertit tous ses caractères en majuscules.

`byleng`

Retourne le nombre de caractères dans une chaîne sans compter les blancs finaux.

```
int byleng(char *str, int len);
```

La fonction attend une chaîne de longueur fixe comme premier argument (`str`) et sa longueur comme second argument (`len`). Elle retourne le nombre de caractères significatifs, c'est à dire la longueur de la chaîne sans ses blancs finaux.

`ldchar`

Copie une chaîne de longueur fixe vers une chaîne terminée par un NUL.

```
void ldchar(char *src, int len, char *dest);
```

La fonction reçoit la chaîne de longueur fixe à copier (`src`), sa longueur (`len`) et un pointeur vers la mémoire destinataire (`dest`). Notez que vous aurez besoin de réserver au moins `len+1` octets pour la chaine vers laquelle pointe `dest`. Cette fonction copie au plus `len` octets vers le nouvel emplacement (moins si la chaîne source a des blancs finaux) et ajoute le terminateur NUL.

`rgetmsg`

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

Cette fonction existe mais n'est pas implémentée pour le moment!

`rtypalign`

```
int rtypalign(int offset, int type);
```

Cette fonction existe mais n'est pas implémentée pour le moment!

`rtypmsize`

```
int rtypmsize(int type, int len);
```

Cette fonction existe mais n'est pas implémentée pour le moment!

`rtypwidth`

```
int rtypwidth(int sqltype, int sqllen);
```

Cette fonction existe mais n'est pas implémentée pour le moment!

`rsetnull`

Set a variable to NULL.

```
int rsetnull(int t, char *ptr);
```

La fonction reçoit un entier qui indique le type de variable et un pointeur vers la variable elle même, transtypé vers un pointeur char*.

Les types suivants existent :

- `CCHARTYPE` - Pour une variable de type `char` ou `char*`
- `CSHORTTYPE` - Pour une variable de type `short int`
- `CINTTYPE` - Pour une variable de type `int`
- `CBOOLTYPE` - Pour une variable de type `boolean`
- `CFLOATTYPE` - Pour une variable de type `float`
- `CLONGTYPE` - Pour une variable de type `long`
- `CDOUBLETYPE` - Pour une variable de type `double`
- `CDECIMALTYPE` - Pour une variable de type `decimal`
- `CDATETYPE` - Pour une variable de type `date`
- `CDTIMETYPE` - Pour une variable de type `timestamp`

Voici un exemple d'appel à cette fonction:

```
$char c[] = "abc       ";
```

```
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

`risnull`

Teste si une variable est NULL.

```
int risnull(int t, char *ptr);
```

Cette fonction reçoit le type d'une variable à tester (`t`) ainsi qu'un pointeur vers cette variable (`ptr`). Notez que ce dernier doit être transtypé vers un char*. Voyez la fonction `rsetnull` pour une liste de types de variables possibles.

Voici un exemple de comment utiliser cette fonction:

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

36.15.5. Constantes Supplémentaires

Notez que toutes les constantes ici décrivent des erreurs et qu'elles sont toutes définies pour représenter des valeurs négatives. Dans les descriptions des différentes constantes vous pouvez aussi trouver la valeur que les constantes représentent dans l'implémentation actuelle. Toutefois, vous ne devriez pas vous fier à ce nombre. Vous pouvez toutefois vous appuyer sur le faite que toutes sont définies comme des valeurs négatives. values.

`ECPG_INFORMIX_NUM_OVERFLOW`

Les fonctions retournent cette valeur si un dépassement s'est produit dans un calcul. En interne, elle est définie à -1200 (la définition Informix).

`ECPG_INFORMIX_NUM_UNDERFLOW`

Les fonctions retournent cette valeur si un soupassement s'est produit dans un calcul. En interne, elle est définie à -1201 (la définition Informix).

`ECPG_INFORMIX_DIVIDE_ZERO`

Les fonctions retournent cette valeur si une division par zéro a été tentée. En interne, elle est définie à -1202 (la définition Informix).

`ECPG_INFORMIX_BAD_YEAR`

Les fonctions retournent cette valeur si une mauvaise valeur pour une année a été trouvée lors de l'analyse d'une date. En interne elle est définie à -1204 (la définition Informix).

`ECPG_INFORMIX_BAD_MONTH`

Les fonctions retournent cette valeur si une mauvaise valeur pour un mois a été trouvée lors de l'analyse d'une date. En interne elle est définie à -1205 (la définition Informix).

`ECPG_INFORMIX_BAD_DAY`

Les fonctions retournent cette valeur si une mauvaise valeur pour un jour a été trouvée lors de l'analyse d'une date. En interne elle est définie à -1206 (la définition Informix).

`ECPG_INFORMIX_ENOSHORTDATE`

Les fonctions retournent cette valeur si une routine d'analyse a besoin d'une représentation courte de date mais que la chaîne passée n'était pas de la bonne longueur. En interne elle est définie à -1206 (la définition Informix).

`ECPG_INFORMIX_DATE_CONVERT`

Les fonctions retournent cette valeur si une erreur s'est produite durant un formatage de date. En interne, elle est définie à -1210 (la définition Informix).

`ECPG_INFORMIX_OUT_OF_MEMORY`

Les fonctions retournent cette valeur si elles se sont retrouvées à court de mémoire durant leur fonctionnement. En interne, elle est définie à -1211 (la définition Informix).

`ECPG_INFORMIX_ENOTDMY`

Les fonctions retournent cette valeur si la routine d'analyse devait recevoir un masque de formatage (comme `mmddyy`) mai que tous les champs n'étaient pas listés correctement. En interne, elle est définie à -1212 (la définition Informix).

`ECPG_INFORMIX_BAD_NUMERIC`

Les fonctions retournent cette valeur soit parce qu'une routine d'analyse ne peut pas analyser la représentation textuelle d'une valeur numérique parce qu'elle contient des erreurs, soit parce qu'une routine ne peut pas terminer un calcul impliquant des variables numeric parce qu'au moins une des variables numeric est invalide. En interne, elle est définie à -1213 (la définition Informix).

`ECPG_INFORMIX_BAD_EXPONENT`

Les fonctions retournent cette valeur si elles n'ont pas réussi à analyser l'exposant de la représentation textuelle d'une valeur numérique. En interne, elle est définie à -1216 (la définition Informix).

`ECPG_INFORMIX_BAD_DATE`

Les fonctions retournent cette valeur si une chaîne de date invalide leur a été passée. En interne, elle est définie à -1218 (la définition Informix).

`ECPG_INFORMIX_EXTRA_CHARS`

Les fonctions retournent cette valeur si trop de caractères ont été trouvés dans la représentation textuelle d'un format date. En interne, elle est définie à -1264 (la définition Informix).

36.16. Mode de compatibilité Oracle

`ecpg` peut être exécuté dans un *mode de compatibilité Oracle*. Si ce mode est actif, il essaie de se comporter comme si c'était du Pro*C Oracle.

En fait, ce mode change le comportement d'`ecpg` de trois façons :

- Remplit les tableaux de caractères recevant des types chaîne de caractères avec des espaces à la fin pour obtenir la longueur indiquée
- Un octet zéro termine ces tableaux de caractères, et configure la variable indicateur si la troncature survient
- Configure l'indicateur null à `-1` quand les tableaux de caractères reçoivent des types de chaînes de caractères vides

36.17. Fonctionnement Interne

Cette section explique comment ECPG fonctionne en interne. Cette information peut être utile pour comprendre comment utiliser ECPG.

Les quatre premières lignes écrites sur la sortie par `ecpg` sont des lignes fixes. Deux sont des commentaires et deux sont des lignes d'inclusion nécessaires pour s'interfacer avec la librairie. Puis le préprocesseur lit le fichier et écrit la sortie. La plupart du temps, il répète simplement tout dans la sortie.

Quand il voit un ordre `EXEC SQL`, il intervient et le modifie. La commande débute par `EXEC SQL` et se termine par `;`. Tout ce qui se trouve entre deux est traité comme un ordre SQL et analysé pour substitution de variables.

Une substitution de variable se produit quand un symbole commence par un deux-points (`:`). La variable dont c'est le nom est recherchée parmi les variables qui ont été précédemment déclarées dans une section `EXEC SQL DECLARE`.

La fonction la plus importante de la librairie est `ECPGdo`, qui s'occupe de l'exécution de la plupart des commandes. Elle prend un nombre variable d'arguments. Le nombre de ces arguments peut rapidement dépasser la cinquantaine, et nous espérons que cela ne posera de problème sur aucune plateforme.

Les arguments sont:

Un numéro de ligne

C'est le numéro de la ligne originale; c'est utilisé uniquement pour les messages d'erreu.

Une chaîne

C'est la commande SQL à exécuter. Elle est modifiée par les variables d'entrée, c'est à dire les variables qui n'étaient pas connues au moment de la compilation mais qui doivent tout de même faire partie de la commande. Aux endroits où ces variables doivent être positionnées, la chaîne contient des `?`.

Variables d'Entrée

Chaque variable d'entrée entraîne la création de dix arguments. (Voir plus bas.)

`ECPGt_EOIT`

Un `enum` annonçant qu'il n'y a pas de variable d'entrées supplémentaires.

Variables de Sortie

Chaque variable de sortie entraîne la création de dix arguments. (Voir plus bas.) Ces variables sont renseignées par la fonction.

`ECPGt_EORT`

Un `enum` annonçant qu'il n'y a plus de variables.

Pour chaque variable qui fait partie d'une commande SQL, la fonction reçoit dix arguments:

1. Le type sous forme de symbole spécial.
2. Un pointeur vers la valeur ou un pointeur vers le pointeur.
3. La taille de la variable si elle est `char` ou `varchar`.
4. Le nombre d'éléments du tableau (pour les fetch sur tableau).
5. Le décalage vers le prochain élément du tableau (pour les fetch sur tableau).
6. Le type de la variable indicateur sous forme de symbole special.
7. Un pointeur vers la variable indicateur.
8. 0
9. Le nombre d'éléments du tableau d'indicateurs (pour les fetch sur tableau).
10. Le décalage vers le prochain élément du tableau d'indicateurs (pour les fetch sur tableau).

Notez que toutes les commandes SQL ne sont pas traitées de cette façon. Par exemple, un ordre d'ouverture de curseur comme:

```
EXEC SQL OPEN cursor;
```

n'est pas copié vers la sortie. À la place, la commande de curseur `DECLARE` est utilisée à l'endroit de la commande `OPEN` parce qu'elle ouvre effectivement le curseur.

Voici un exemple complet expliquant la sortie du préprocesseur sur un fichier `foo.pgc` (quelques détails pourraient changer en fonction de la version exacte du préprocesseur):

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

est traduit en :

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?
 ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
```

```
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

(L'indentation est ajoutée ici pour améliorer la lisibilité et n'est pas quelque chose que le préprocesseur effectue).

Chapitre 37. Schéma d'information

Le schéma d'information consiste en un ensemble de vues contenant des informations sur les objets définis dans la base de données courante. Le schéma d'information est défini dans le standard SQL et, donc supposé portable et stable -- contrairement aux catalogues système qui sont spécifiques à PostgreSQL et modelés suivant l'implantation. Néanmoins, les vues du schéma d'information ne contiennent pas d'informations sur les fonctionnalités spécifiques à PostgreSQL ; pour cela, on utilise catalogues système et autres vues spécifiques à PostgreSQL.

> **Note**
>
> En demandant des informations sur les contraintes dans la base de données, il est possible qu'une requête conforme au standard s'attendant à ne récupérer qu'une ligne en récupère en fait plusieurs. Ceci est dû au fait que le standard SQL requiert que les noms des contraintes soient uniques dans un même schéma mais PostgreSQL ne force pas cette restriction. Les noms de contraintes créés automatiquement par PostgreSQL évitent les doublons dans le même schéma, mais les utilisateurs peuvent spécifier explicitement des noms existant déjà.
>
> Ce problème peut apparaître lors de la consultation de vues du schéma d'informations, comme par exemple `check_constraint_routine_usage`, `check_constraints`, `domain_constraints` et `referential_constraints`. Certaines autres vues ont des problèmes similaires mais contiennent le nom de la table pour aider à distinguer les lignes dupliquées, par exemple `constraint_column_usage`, `constraint_table_usage`, `table_constraints`.

37.1. Le schéma

Le schéma d'information est lui-même un schéma nommé `information_schema`. Ce schéma existe automatiquement dans toutes les bases de données. Le propriétaire de ce schéma est l'utilisateur initial du cluster. Il a naturellement tous les droits sur ce schéma, dont la possibilité de le supprimer (mais l'espace gagné ainsi sera minuscule).

Par défaut, le schéma d'information n'est pas dans le chemin de recherche des schémas. Il est donc nécessaire d'accéder à tous les objets qu'il contient via des noms qualifiés. Comme les noms de certains objets du schéma d'information sont des noms génériques pouvant survenir dans les applications utilisateur, il convient d'être prudent avant de placer le schéma d'information dans le chemin.

37.2. Types de données

Les colonnes des vues du schéma d'information utilisent des types de données spéciaux, définis dans le schéma d'information. Ils sont définis comme des domaines simples sur des types internes. Vous ne devriez pas utiliser ces types en dehors du schéma d'information, mais les applications doivent pouvoir les utiliser si des sélections sont faites dans le schéma d'information.

Ces types sont :

`cardinal_number`

Un entier non négatif.

`character_data`

Une chaîne de caractères (sans longueur maximale indiquée).

`sql_identifier`

Une chaîne de caractères. Elle est utilisée pour les identifiants SQL, le type de données `character_data` est utilisé pour tout autre type de données texte.

`time_stamp`

Un domaine au-dessus du type `timestamp with time zone`

`yes_or_no`

Un domaine dont le type correspond à une chaîne de caractères, qui contient soit `YES` soit `NO`. C'est utilisé pour représenter des données booléennes (true/false) dans le schéma d'informations. (Le schéma d'informations était inventé avant l'ajout du type `boolean` dans le standard SQL, donc cette convention est nécessaire pour conserver la compatibilité avec le schéma d'informations.)

Chaque colonne du schéma d'information est de l'un des ces cinq types.

37.3. `information_schema_catalog_name`

`information_schema_catalog_name` est une table qui contient en permanence une ligne et une colonne contenant le nom de la base de données courante (catalogue courant dans la terminologie SQL).

Tableau 37.1. Colonnes de `information_schema_catalog_name`

Type de la colonne **Description**
`catalog_name` `sql_identifier` Nom de la base de données contenant ce schéma d'informations

37.4. `administrable_role_authorizations`

La vue `administrable_role_authorizations` identifie tous les rôles pour lesquels l'utilisateur courant possède l'option ADMIN.

Tableau 37.2. Colonnes de `administrable_role_authorizations`

Type de la colonne **Description**
`grantee` `sql_identifier` Nom du rôle pour lequel cette appartenance de rôle a été donnée (peut être l'utilisateur courant ou un rôle différent dans le cas d'appartenances de rôles imbriquées).
`role_name` `sql_identifier` Nom d'un rôle
`role_name` `sql_identifier` Toujours `YES`

37.5. `applicable_roles`

La vue `applicable_roles` identifie tous les rôles dont l'utilisateur courant peut utiliser les droits. Cela signifie qu'il y a certaines chaînes de délégation des droits de l'utilisateur courant au rôle en question. L'utilisateur lui-même est un rôle applicable. L'ensemble de rôles applicables est habituellement utilisé pour la vérification des droits.

Tableau 37.3. Colonnes de `applicable_roles`

Type de la colonne Description
`grantee` `sql_identifier` Nom du rôle à qui cette appartenance a été donnée (peut être l'utilisateur courant ou un rôle différent dans le cas d'appartenances de rôles imbriquées)
`role_name` `sql_identifier` Nom d'un rôle
`is_grantable` `yes_or_no` `YES` si le bénéficiaire a l'option ADMIN sur le rôle, `NO` dans le cas contraire

37.6. `attributes`

La vue `attributes` contient des informations sur les attributs des types de données composites définis dans la base. (La vue ne donne pas d'informations sur les colonnes de table, qui sont quelque fois appelées attributs dans le contexte de PostgreSQL.) Seuls ces attributs sont affichés plutôt que ceux auxquels l'utilisateur courant a accès (s'il est le propriétaire ou a des droits sur le type).

Tableau 37.4. Colonnes de `attributes`

Type de la colonne Description
`udt_catalog` `sql_identifier` Nom de la base contenant le type de données (toujours la base courante)
`udt_schema` `sql_identifier` Nom du schéma contenant le type de données
`udt_name` `sql_identifier` Nom du type de données
`attribute_name` `sql_identifier` Nom de l'attribut
`ordinal_position` `cardinal_number` Position de l'attribut dans le type de données (le décompte commence à 1)
`attribute_default` `character_data` Expression par défaut de l'attribut
`is_nullable` `yes_or_no` `YES` si l'attribut peut être NULL, `NO` dans le cas contraire.
`data_type` `character_data` Type de données de l'attribut s'il s'agit d'un type interne ou `ARRAY` s'il s'agit d'un tableau (dans ce cas, voir la vue `element_types`), sinon `USER-DEFINED` (dans ce cas, le type est identifié dans `attribute_udt_name` et les colonnes associées).
`character_maximum_length` `cardinal_number` Si `data_type` identifie un caractère ou une chaîne de bits, la longueur maximale déclarée ; NULL pour tous les autres types de données ou si aucune longueur maximale n'a été déclarée.
`character_octet_length` `cardinal_number` Si `data_type` identifie un type caractère, la longueur maximale en octets (bytes) d'un datum ; NULL pour tous les autres types de données. La longueur maximale en octets dépend de la longueur maximum déclarée en caractères (voir ci-dessus) et l'encodage du serveur.
`character_set_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible avec PostgreSQL

Type de la colonne **Description**
`character_set_schema` `sql_identifier` S'applique à une fonctionnalité non disponible avec PostgreSQL
`character_set_name` `sql_identifier` S'applique à une fonctionnalité non disponible avec PostgreSQL
`collation_catalog` `sql_identifier` Nom de la base contenant le collationnement de l'attribut (toujours la base de données courante), NULL s'il s'agit du collationnement par défaut ou si le type de données de l'attribut ne peut pas avoir de collationnement
`collation_schema` `sql_identifier` Nom du schéma contenant le collationnement de l'attribut, NULL s'il s'agit du collationnement par défaut ou si le type de données de l'attribut ne peut pas avoir de collationnement
`collation_name` `sql_identifier` Nom du collationnement de l'attribut, NULL s'il s'agit du collationnement par défaut ou si le type de données de l'attribut ne peut pas avoir de collationnement
`numeric_precision` `cardinal_number` Si `data_type` identifie un type numérique, cette colonne contient la précision (déclarée ou implicite) du type pour cet attribut. La précision indique le nombre de chiffres significatifs. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2) comme le précise la colonne `numeric_precision_radix`. Pour tous les autres types de données, cette colonne vaut NULL.
`numeric_precision_radix` `cardinal_number` Si `data_type` identifie un type numérique, cette colonne indique la base d'expression des colonnes `numeric_precision` et `numeric_scale`. La valeur est soit 2 soit 10. Pour tous les autres types de données, cette colonne est NULL.
`numeric_scale` `cardinal_number` Si `data_type` identifie un type numérique exact, cette colonne contient l'échelle (déclarée ou implicite) du type pour cet attribut. L'échelle indique le nombre de chiffres significatifs à droite du point décimal. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2) comme le précise la colonne `numeric_precision_radix`. Pour tous les autres types de données, cette colonne est NULL.
`datetime_precision` `cardinal_number` Si `data_type` identifie une date, une heure, un horodatage ou un interval, cette colonne contient la précision en secondes (déclarée ou implicite) pour cet attribut, c'est-à-dire le nombre de chiffres décimaux suivant le point décimal de la valeur en secondes. Pour tous les autres types de données, cette colonne est NULL.
`interval_type` `character_data` Si `data_type` identifie un type d'intervalle, cette colonne contient la spécification des champs que les intervalles incluent pour cet attribut, par exemple `YEAR TO MONTH`, `DAY TO SECOND`, etc. Si aucune restriction de champs n'est spécifiée (autrement dit, l'intervalle accepte tous les champs) et pour tous les autres types de données, ce champ est NULL.
`interval_precision` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL (voir `datetime_precision` pour la précision en fraction des secondes des attributs du type d'intervalle)
`attribute_udt_catalog` `sql_identifier` Nom de la base dans laquelle le type de données de l'attribut est défini (toujours la base courante)
`attribute_udt_schema` `sql_identifier`

Type de la colonne **Description**
Nom du schéma dans lequel le type de données de l'attribut est défini
`attribute_udt_name` `sql_identifier` Nom du type de données de l'attribut
`scope_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`scope_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`scope_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`maximum_cardinality` `cardinal_number` Toujours NULL car les tableaux ont toujours une cardinalité maximale dans PostgreSQL
`dtd_identifier` `sql_identifier` Un identifiant du descripteur du type de données de l'attribut, unique parmi les descripteurs de types de données de la table. Ceci est principalement utile pour des jointures avec d'autres instances de tels identifiants. (Le format spécifique de l'identifiant n'est pas défini et il n'est pas garanti qu'il reste identique dans les versions futures.)
`is_derived_reference_attribute` `yes_or_no` S'applique à une fonctionnalité non disponible dans PostgreSQL

Voir aussi dans Section 37.17, une vue structurée de façon similaire, pour plus d'informations sur certaines colonnes.

37.7. `character_sets`

La vue `character_sets` identifie les jeux de caractères disponibles pour la base de données courante. Comme PostgreSQL ne supporte pas plusieurs jeux de caractères dans une base de données, cette vue n'en affiche qu'une, celle qui correspond à l'encodage de la base de données.

Les termes suivants sont utilisés dans le standard SQL :

répertoire de caractères (*character repertoire*

> Un ensemble abstrait de caractères, par exemple `UNICODE`, `UCS` ou `LATIN1`. Non exposé en tant qu'objet SQL mais visible dans cette vue.

forme d'encodage de caractères (*character encoding form*)

> Un encodage d'un certain répertoire de caractères. La plupart des anciens répertoires de caractères utilisent seulement un encodage. Du coup, il n'y a pas de noms séparés pour eux (par exemple `LATIN2` est une forme d'encodage applicable au répertoire `LATIN2`). Par contre, Unicode dispose des formats d'encodage `UTF8`, `UTF16`, etc. (ils ne sont pas tous supportés par PostgreSQL). Les formes d'encodage ne sont pas exposés comme un objet SQL mais ils sont visibles dans cette vue.

jeu de caractères (*character set*)

> Un objet SQL nommé qui identifie un répertoire de caractères, un encodage de caractères et un collationnement par défaut. Un jeu de caractères prédéfini aura généralement le même nom qu'une forme d'encodage mais les utilisateurs peuvent définir d'autres noms. Par exemple, le jeu de caractères `UTF8` identifiera typiquement le répertoire des caractères `UCS`, la forme d'encodage `UTF8` et un collationnement par défaut.

Dans PostgreSQL, un « encodage » peut être vu comme un jeu de caractères ou une forme d'encodage des caractères. Ils auront le même nom et il n'y en a qu'un dans une base de données.

Tableau 37.5. Colonnes de `character_sets`

Type de la colonne Description
`character_set_catalog` `sql_identifier` Les jeux de caractères ne sont pas actuellement implémentés comme des objets du schéma, donc cette colonne est NULL.
`character_set_schema` `sql_identifier` Les jeux de caractères ne sont pas actuellement implémentés comme des objets du schéma, donc cette colonne est NULL.
`character_set_name` `sql_identifier` Nom du jeu de caractères, mais affiche actuellement le nom de l'encodage de la base de données
`character_repertoire` `sql_identifier` Répertoire des caractères, affichant `UCS` si l'encodage est `UTF8`, et le nom de l'encodage sinon
`form_of_use` `sql_identifier` Forme d'encodage des caractères, identique à l'encodage de la base de données
`default_collate_catalog` `sql_identifier` Nom de la base de données contenant le collationnement par défaut (toujours la base de données courante si un collationnement est identifié)
`default_collate_schema` `sql_identifier` Nom du schéma contenant le collationnement par défaut
`default_collate_name` `sql_identifier` Nom du collationnement par défaut. Il est identifié comme le collationnement qui correspond aux paramètres `COLLATE` et `CTYPE` pour la base de données courante. S'il n'y a pas de collationnement, cette colonne, le schéma associé et les colonnes du catalogue sont NULL.

37.8. `check_constraint_routine_usage`

La vue `check_constraint_routine_usage` identifie les routines (fonctions et procédures) utilisées par une contrainte de vérification. Seules sont présentées les routines qui appartiennent à un rôle couramment actif.

Tableau 37.6. Colonnes de `check_constraint_routine_usage`

Type de la colonne Description
`constraint_catalog` `sql_identifier` Nom de la base contenant la contrainte (toujours la base courante)
`constraint_schema` `sql_identifier` Nom du schéma contenant la contrainte
`constraint_name` `sql_identifier` Nom de la contrainte
`specific_catalog` `sql_identifier` Nom de la base contenant la fonction (toujours la base courante)
`specific_schema` `sql_identifier` Nom du schéma contenant la fonction
`specific_name` `sql_identifier` Le « nom spécifique » de la fonction. Voir Section 37.45 pour plus d'informations.

37.9. check_constraints

La vue check_constraints contient toutes les contraintes de vérification définies sur une table ou un domaine, possédées par un rôle couramment actif (le propriétaire d'une table ou d'un domaine est le propriétaire de la contrainte).

Tableau 37.7. Colonnes de check_constraints

Type de la colonne Description
constraint_catalog sql_identifier Nom de la base de données contenant la contrainte (toujours la base de données courante)
constraint_schema sql_identifier Nom du schéma contenant la contrainte
constraint_name sql_identifier Nom de la contrainte
constraint_name sql_identifier L'expression de vérification de la contrainte

37.10. collations

La vue collations contient les collationnements disponibles dans la base de données courante.

Tableau 37.8. Colonnes de collations

Type de la colonne Description
collation_catalog sql_identifier Nom de la base de données contenant le collationnement (toujours la base de données courante)
collation_schema sql_identifier Nom du schéma contenant le collationnement
collation_name sql_identifier Nom du collationnement par défaut
pad_attribute character_data Toujours NO PAD (l'alternative PAD SPACE n'est pas supportée par PostgreSQL.)

37.11. collation_character_set_applicability

La vue collation_character_set_applicability identifie les jeux de caractères applicables aux collationnements disponibles. Avec PostgreSQL, il n'existe qu'un jeu de caractères par base de données (voir les explications dans Section 37.7), donc cette vue ne fournit pas beaucoup d'informations utiles.

Tableau 37.9. Colonnes de collation_character_set_applicability

Type de la colonne Description
collation_catalog sql_identifier Nom de la base de données contenant le collationnement (toujours la base de données courante)
collation_schema sql_identifier

Type de la colonne Description
Nom du schéma contenant le collationnement
`collation_name` `sql_identifier` Nom du collationnement par défaut
`character_set_catalog` `sql_identifier` Les jeux de caractères ne sont pas actuellement implémentés comme des objets du schéma, donc cette colonne est NULL.
`character_set_schema` `sql_identifier` Les jeux de caractères ne sont pas actuellement implémentés comme des objets du schéma, donc cette colonne est NULL.
`character_set_name` `sql_identifier` Nom du jeu de caractères

37.12. `column_column_usage`

La vue `column_column_usage` identifie toutes les colonnes générées qui dépendant d'une autre colonne de base dans la même table. Seuls les tables possédées par le rôle actuel sont incluses.

Tableau 37.10. Colonnes de `column_column_usage`

Type de la colonne Description
`table_catalog` `sql_identifier` Nom de la base contenant la table (toujours la base actuelle)
`table_schema` `sql_identifier` Nom du schéma contenant la table
`table_name` `sql_identifier` Nom de la table
`column_name` `sql_identifier` Nom de la colonne de base dont la colonne générée dépend
`dependent_column` `sql_identifier` Nom de la colonne générée

37.13. `column_domain_usage`

La vue `column_domain_usage` identifie toutes les colonnes (d'une table ou d'une vue) utilisant un domaine défini dans la base de données courante et possédé par un rôle couramment actif.

Tableau 37.11. Colonnes de `column_domain_usage`

Type de la colonne Description
`domain_catalog` `sql_identifier` Nom de la base de données contenant le domaine (toujours la base de données courante)
`domain_schema` `sql_identifier` Nom du schéma contenant le domaine
`domain_name` `sql_identifier` Nom du domaine
`table_catalog` `sql_identifier` Nom de la base de données contenant la table (toujours la base de données courante)
`table_schema` `sql_identifier`

Type de la colonne Description
Nom du schéma contenant la table
`table_name` `sql_identifier` Nom de la table
`column_name` `sql_identifier` Nom de la colonne

37.14. `column_options`

La vue `column_options` contient toutes les options définies pour les colonnes des tables étrangères de la base de données courante. Seules sont montrées les tables étrangères auxquelles l'utilisateur courant a accès (soit parce qu'il en est le propriétaire soit parce qu'il dispose de certains droits dessus)

Tableau 37.12. Colonnes de `column_options`

Type de la colonne Description
`table_catalog` `sql_identifier` Nom de la base contenant la table distante (toujours la base de données courante)
`table_schema` `sql_identifier` Nom du schéma contenant la table distante
`table_name` `sql_identifier` Nom de la table distante
`column_name` `sql_identifier` Nom de la colonne
`option_name` `sql_identifier` Nom de l'option
`option_value` `character_data` Valeur de l'option

37.15. `column_privileges`

La vue `column_privileges` identifie tous les droits octroyés sur les colonnes à un rôle couramment actif ou par un rôle couramment actif. Il existe une ligne pour chaque combinaison colonne, donneur (*grantor*) et receveur (*grantee*).

Si un droit a été donné sur une table entière, il s'affichera dans cette vue comme un droit sur chaque colonne, mais seulement pour les types de droits où la granularité par colonne est possible : `SELECT`, `INSERT`, `UPDATE`, `REFERENCES`.

Tableau 37.13. Colonnes de `column_privileges`

Type de la colonne Description
`grantor` `sql_identifier` Nom du rôle ayant accordé le privilège
`grantee` `sql_identifier` Nom du rôle receveur
`table_catalog` `sql_identifier` Nom de la base de données qui contient la table qui contient la colonne (toujours la base de données courante)
`table_schema` `sql_identifier`

Type de la colonne Description
Nom du schéma qui contient la table qui contient la colonne
`table_name` `sql_identifier` Nom de la table qui contient la colonne
`column_name` `sql_identifier` Nom de la colonne
`privilege_type` `character_data` Type de privilège : `SELECT`, `INSERT`, `UPDATE` ou `REFERENCES`
`is_grantable` `yes_or_no` `YES` si le droit peut être accordé, `NO` sinon

37.16. `column_udt_usage`

La vue `column_udt_usage` identifie toutes les colonnes qui utilisent les types de données possédés par un rôle actif. Avec PostgreSQL, les types de données internes se comportent comme des types utilisateur, ils apparaissent aussi ici. Voir aussi la Section 37.17 pour plus de détails.

Tableau 37.14. Colonnes de `column_udt_usage`

Type de la colonne Description
`udt_catalog` `sql_identifier` Nom de la base de données dans laquelle le type de donnée de la colonne (le type sous-jacent du domaine, si applicable) est défini (toujours la base de données courante).
`udt_schema` `sql_identifier` Nom du schéma dans lequel le type de donnée de la colonne (le type sous-jacent du domaine, si applicable) est défini.
`udt_name` `sql_identifier` Nom du type de données de la colonne (le type sous-jacent du domaine, si applicable).
`table_catalog` `sql_identifier` Nom de la base de données contenant la table (toujours la base de données courante).
`table_schema` `sql_identifier` Nom du schéma contenant la table.
`table_name` `sql_identifier` Nom de la table.
`column_name` `sql_identifier` Nom de la colonne.

37.17. `columns`

La vue `columns` contient des informations sur toutes les colonnes de table (et colonnes de vue) de la base. Les colonnes système (`ctid`, etc.) ne sont pas incluses. Seules les colonnes auxquelles l'utilisateur a accès (par propriété ou par privilèges) sont affichées.

Tableau 37.15. Colonnes de `columns`

Type de la colonne Description
`table_catalog` `sql_identifier` Nom de la base de données contenant la table (toujours la base de données courante)
`table_schema` `sql_identifier`

Type de la colonne **Description**
Nom du schéma contenant la table
`table_name` `sql_identifier` Nom de la table
`column_name` `sql_identifier` Nom de la colonne
`ordinal_position` `cardinal_number` Position de la colonne dans la table (la numérotation commençant à 1)
`column_default` `character_data` Expression par défaut de la colonne
`is_nullable` `yes_or_no` `YES` si la colonne est *NULLable* (elle admet une absence de valeur), `NO` dans le cas contraire. La contrainte NOT NULL n'est pas la seule façon de définir qu'une colonne n'est pas *NULLable*.
`data_type` `character_data` Le type de données de la colonne, s'il s'agit d'un type interne ou `ARRAY` s'il s'agit d'un tableau (dans ce cas, voir la vue `element_types`), `USER-DEFINED` dans les autres cas (le type est alors identifié dans `udt_name` et colonnes associées). Si la colonne est fondée sur un domaine, cette colonne est une référence au type sous-jacent du domaine (et le domaine est identifié dans `domain_name` et colonnes associées).
`character_maximum_length` `cardinal_number` Si `data_type` identifie un type chaîne de caractères ou chaîne de bits, la longueur maximale déclarée ; NULL pour tous les autres types de données ou si aucune longueur maximale n'a été déclarée.
`character_octet_length` `cardinal_number` Si `data_type` identifie un type caractère, la longueur maximale en octets (bytes) d'un datum ; NULL pour tous les autres types de données. La longueur maximale en octets dépend de la longueur maximum déclarée en caractères (voir ci-dessus) et l'encodage du serveur.
`numeric_precision` `cardinal_number` Si `data_type` identifie un type numérique, cette colonne contient la précision (déclarée ou implicite) du type pour ce domaine. La précision indique le nombre de chiffres significatifs. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2) comme indiqué dans la colonne `numeric_precision_radix`. Pour tous les autres types de données, la colonne est NULL.
`numeric_precision_radix` `cardinal_number` Si `data_type` identifie un type numérique, cette colonne indique dans quelle base les valeurs des colonnes `numeric_precision` et `numeric_scale` sont exprimées. La valeur est 2 ou 10. Pour tous les autres types de données, cette colonne est NULL.
`numeric_scale` `cardinal_number` Si `data_type` identifie un type numeric exact, cette colonne contient l'échelle (déclarée ou implicite) du type pour ce domaine. L'échelle indique le nombre de chiffres significatifs à la droite du point décimal. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2), comme indiqué dans la colonne `numeric_precision_radix`. Pour tous les autres types de données, cette colonne est NULL.
`datetime_precision` `cardinal_number` Si `data_type` identifie une date, une heure, un horodatage ou un interval, cette colonne contient la précision en secondes (déclarée ou implicite) pour cet attribut, c'est-à-dire le nombre de chiffres décimaux suivant le point décimal de la valeur en secondes. Pour tous les autres types de données, cette colonne est NULL.
`interval_type` `character_data`

Type de la colonne **Description**
Si `data_type` identifie un type d'intervalle, cette colonne contient la spécification des champs que les intervalles incluent pour cette colonne, par exemple `YEAR TO MONTH`, `DAY TO SECOND`, etc. Si aucune restriction de champs n'est spécifiée (autrement dit, l'intervalle accepte tous les champs) et pour tous les autres types de données, ce champ est NULL.
`interval_precision` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL (voir `datetime_precision` pour la précision en fraction des secondes des attributs du type d'intervalle)
`character_set_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`character_set_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`character_set_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`collation_catalog` `sql_identifier` Nom de la base contenant le collationnement de l'attribut (toujours la base de données courante), NULL s'il s'agit du collationnement par défaut ou si le type de données de l'attribut ne peut pas avoir de collationnement
`collation_schema` `sql_identifier` Nom du schéma contenant le collationnement de l'attribut, NULL s'il s'agit du collationnement par défaut ou si le type de données de l'attribut ne peut pas avoir de collationnement
`collation_name` `sql_identifier` Nom du collationnement de l'attribut, NULL s'il s'agit du collationnement par défaut ou si le type de données de l'attribut ne peut pas avoir de collationnement
`domain_catalog` `sql_identifier` Si la colonne a un type domaine, le nom de la base de données où le type est défini (toujours la base de données courante), sinon NULL.
`domain_schema` `sql_identifier` Si la colonne a un type domaine, le nom du schéma où le domaine est défini, sinon NULL.
`domain_name` `sql_identifier` Si la colonne a un type de domaine, le nom du domaine, sinon NULL.
`udt_catalog` `sql_identifier` Nom de la base de données où le type de données de la colonne (le type sous-jacent du domaine, si applicable) est défini (toujours la base de données courante).
`udt_schema` `sql_identifier` Nom du schéma où le type de données de la colonne (le type sous-jacent du domaine, si applicable) est défini.
`udt_name` `sql_identifier` Nom du type de données de la colonne (le type sous-jacent du domaine, si applicable).
`scope_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`scope_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`scope_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`maximum_cardinality` `cardinal_number`

Type de la colonne **Description**
Toujours NULL car les tableaux ont toujours une cardinalité maximale illimitée avec PostgreSQL.
`dtd_identifier` `sql_identifier` Un identifiant du descripteur du type de données de la colonne, unique parmi les descripteurs de type de données contenus dans la table. Ceci est principalement utile pour joindre d'autres instances de ces identifiants. (Le format spécifique de l'identifiant n'est pas défini et rien ne permet d'assurer qu'il restera inchangé dans les versions futures.)
`is_self_referencing` `yes_or_no` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`is_identity` `yes_or_no` Si la colonne est une colonne d'identité, alors `YES`, sinon `NO`.
`identity_generation` `character_data` Si la colonne est une colonne d'identité, alors `ALWAYS` sinon `BY DEFAULT`, reflétant la définition de la colonne.
`identity_start` `character_data` Si la colonne est une colonne identité, alors la valeur de démarrage de la séquence interne, sinon NULL.
`identity_increment` `character_data` Si la colonne est une colonne identité, alors l'incrément de la séquence interne, sinon NULL.
`identity_maximum` `character_data` Si la colonne est une colonne identité, alors la valeur maximale de la séquence interne, sinon NULL.
`identity_minimum` `character_data` Si la colonne est une colonne identité, alors la valeur minimale de la séquence interne, sinon NULL.
`identity_cycle` `yes_or_no` Si la colonne est une colonne identité, alors `YES` si la séquence interne fait un cycle et `NO` dans le cas contraire ; sinon NULL.
`is_generated` `character_data` Si la colonne est une colonne générée, alors `ALWAYS`, sinon `NEVER`.
`generation_expression` `character_data` Si la colonne est une colonne générée, alors l'expression générée, sinon null.
`is_updatable` `yes_or_no` `YES` si la colonne est actualisable, `NO` dans le cas contraire (les colonnes des tables sont toujours modifiables, les colonnes des vues ne le sont pas nécessairement).

Puisqu'en SQL les possibilités de définir les types de données sont nombreuses, et que PostgreSQL offre des possibilités supplémentaires, leur représentation dans le schéma d'information peut s'avérer complexe.

La colonne `data_type` est supposée identifier le type de données interne sous-jacent de la colonne. Avec PostgreSQL, cela signifie que le type est défini dans le schéma du catalogue système `pg_catalog`. Cette colonne est utile si l'application sait gérer les types internes (par exemple, formater les types numériques différemment ou utiliser les données dans les colonnes de précision). Les colonnes `udt_name`, `udt_schema` et `udt_catalog` identifient toujours le type de données sous-jacent de la colonne même si la colonne est basée sur un domaine.

Puisque PostgreSQL traite les types internes comme des types utilisateur, les types internes apparaissent aussi ici. Il s'agit d'une extension du standard SQL.

Toute application conçue pour traiter les données en fonction du type peut utiliser ces colonnes, car, dans ce cas, il importe peu de savoir si la colonne est effectivement fondée sur un domaine. Si la colonne est fondée sur un domaine, l'identité du domaine est stockée dans les colonnes `domain_name`, `domain_schema` et `domain_catalog`. Pour assembler les colonnes avec leurs types de données associés et traiter les domaines comme des types séparés, on peut écrire `coalesce(domain_name, udt_name)`, etc.

37.18. `constraint_column_usage`

La vue `constraint_column_usage` identifie toutes les colonnes de la base de données courante utilisées par des contraintes. Seules sont affichées les colonnes contenues dans une table possédée par un rôle connecté. Pour une contrainte de vérification, cette vue identifie les colonnes utilisées dans l'expression de la vérification. Pour une contrainte de clé étrangère, cette vue identifie les colonnes que la clé étrangère référence. Pour une contrainte d'unicité ou de clé primaire, cette vue identifie les colonnes contraintes.

Tableau 37.16. Colonnes de `constraint_column_usage`

Type de la colonne **Description**
`table_catalog` `sql_identifier` Nom de la base de données contenant la table contenant la colonne utilisée par certaines contraintes (toujours la base de données courante)
`table_schema` `sql_identifier` Nom du schéma contenant la table contenant la colonne utilisée par certaines contraintes
`table_name` `sql_identifier` Nom de la table contenant la colonne utilisée par certaines contraintes
`column_name` `sql_identifier` Nom de la colonne utilisée par certaines contraintes
`constraint_catalog` `sql_identifier` Nom de la base de données contenant la contrainte (toujours la base de données courante)
`constraint_schema` `sql_identifier` Nom du schéma contenant la contrainte
`constraint_name` `sql_identifier` Nom de la contrainte

37.19. `constraint_table_usage`

La vue `constraint_table_usage` identifie toutes les tables de la base de données courante utilisées par des contraintes et possédées par un rôle actuellement activé. (Cela diffère de la vue `table_constraints` qui identifie toutes les contraintes et la table où elles sont définies.) Pour une contrainte de clé étrangère, cette vue identifie la table que la clé étrangère référence. Pour une contrainte d'unicité ou de clé primaire, cette vue identifie simplement la table à laquelle appartient la contrainte. Les contraintes de vérification et les contraintes de non nullité (NOT NULL) ne sont pas incluses dans cette vue.

Tableau 37.17. Colonnes de `constraint_table_usage`

Type de la colonne **Description**
`table_catalog` `sql_identifier` Nom de la base de données contenant la table utilisée par quelques contraintes (toujours la base de données courante)

Type de la colonne Description
`table_schema` `sql_identifier` Nom du schéma contenant la table utilisée par quelque contrainte
`table_name` `sql_identifier` Nom de la table utilisée par quelque contrainte
`constraint_catalog` `sql_identifier` Nom de la base de données contenant la contrainte (toujours la base de données courante)
`constraint_schema` `sql_identifier` Nom du schéma contenant la contrainte
`constraint_name` `sql_identifier` Nom de la contrainte

37.20. `data_type_privileges`

La vue `data_type_privileges` identifie tous les descripteurs de type de données auxquels l'utilisateur a accès, parce qu'il en est le propriétaire ou parce qu'il dispose de quelque droit sur le descripteur. Un descripteur de type de données est créé lorsqu'un type de données est utilisé dans la définition d'une colonne de table, d'un domaine ou d'une fonction (en tant que paramètre ou code de retour). Il stocke alors quelques informations sur l'utilisation du type de données (par exemple la longueur maximale déclarée, si applicable). Chaque descripteur de type de données se voit affecter un identifiant unique parmi les descripteurs de type de données affectés à un objet (table, domaine, fonction). Cette vue n'est probablement pas utile pour les applications, mais elle est utilisée pour définir d'autres vues dans le schéma d'information.

Tableau 37.18. Colonnes de `data_type_privileges`

Type de la colonne Description
`object_catalog` `sql_identifier` Nom de la base de données contenant l'objet décrit (toujours la base de données courante)
`object_schema` `sql_identifier` Nom du schéma contenant l'objet décrit
`object_name` `sql_identifier` Nom de l'objet décrit
`object_type` `character_data` Le type d'objet décrit : fait partie de `TABLE` (le descripteur de type de données concerne une colonne de cette table), `DOMAIN` (le descripteur concerne ce domaine), `ROUTINE` (le descripteur est lié à un type de paramètre ou de code de retour de cette fonction).
`dtd_identifier` `sql_identifier` L'identifiant du descripteur de type de données, unique parmi les descripteurs de type de données pour le même objet.

37.21. `domain_constraints`

La vue `domain_constraints` contient toutes les contraintes appartenant à des domaines définis dans la base de données courante. Seuls sont affichés les contraintes auxquelles l'utilisateur a accès (soit parce qu'il en est le propriétaire, soit parce qu'il possède certains droits dessus).

Tableau 37.19. Colonnes de `domain_constraints`

Type de la colonne Description
`constraint_catalog` `sql_identifier` Nom de la base de données contenant la contrainte (toujours la base de données courante)
`constraint_schema` `sql_identifier` Nom du schéma contenant la contrainte
`constraint_name` `sql_identifier` Nom de la contrainte
`domain_catalog` `sql_identifier` Nom de la base de données contenant le domaine (toujours la base de données courante)
`domain_schema` `sql_identifier` Nom du schéma contenant le domaine
`domain_name` `sql_identifier` Nom du domaine
`is_deferrable` `yes_or_no` `YES` si la vérification de la contrainte peut être différée, `NO` sinon
`initially_deferred` `yes_or_no` `YES` si la vérification de la contrainte, qui peut être différée, est initialement différée, `NO` sinon

37.22. `domain_udt_usage`

La vue `domain_udt_usage` identifie tous les domaines utilisant les types de données possédés par un rôle actif. Sous PostgreSQL, les types de données internes se comportent comme des types utilisateur. Ils sont donc inclus ici.

Tableau 37.20. Colonnes de `domain_udt_usage`

Type de la colonne Description
`udt_catalog` `sql_identifier` Nom de la base de données de définition du type de données domaine (toujours la base de données courante)
`udt_schema` `sql_identifier` Nom du schéma de définition du type de données domaine
`udt_name` `sql_identifier` Nom du type de données domaine
`domain_catalog` `sql_identifier` Nom de la base de données contenant le domaine (toujours la base de données courante)
`domain_schema` `sql_identifier` Nom du schéma contenant le domaine
`domain_name` `sql_identifier` Nom du domaine

37.23. `domains`

La vue `domains` contient tous les *domaines* définis dans la base de données courante. Seuls sont affichés les domaines auxquels l'utilisateur a accès (soit parce qu'il en est le propriétaire, soit parce qu'il possède certains droits dessus).

Tableau 37.21. Colonnes de `domains`

Type de la colonne **Description**
`domain_catalog` `sql_identifier` Nom de la base de données contenant le domaine (toujours la base de données courante)
`domain_schema` `sql_identifier` Nom du schéma contenant le domaine
`domain_name` `sql_identifier` Nom du domaine
`data_type` `character_data` Type de données du domaine s'il s'agit d'un type interne, ou `ARRAY` s'il s'agit d'un tableau (dans ce cas, voir la vue `element_types`), sinon `USER-DEFINED` (dans ce cas, le type est identifié dans `udt_name` et comprend des colonnes associées).
`character_maximum_length` `cardinal_number` Si le domaine a un type caractère ou chaîne de bits, la longueur maximale déclarée ; NULL pour tous les autres types de données ou si aucune longueur maximale n'a été déclarée.
`character_octet_length` `cardinal_number` Si le domaine a un type caractère, la longueur maximale en octets (bytes) d'un datum ; NULL pour tous les autres types de données. La longueur maximale en octets dépend de la longueur maximum déclarée en caractères (voir ci-dessus) et l'encodage du serveur.
`character_set_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`character_set_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`character_set_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`collation_catalog` `sql_identifier` Nom de la base contenant le collationnement du domaine (toujours la base de données courante), NULL s'il s'agit du collationnement par défaut ou si le type de données de l'attribut ne peut pas avoir de collationnement
`collation_schema` `sql_identifier` Nom du schéma contenant le collationnement du domaine, NULL s'il s'agit du collationnement par défaut ou si le type de données du domaine ne peut pas avoir de collationnement
`collation_name` `sql_identifier` Nom du collationnement de la domaine, NULL s'il s'agit du collationnement par défaut ou si le type de données du domaine ne peut pas avoir de collationnement
`numeric_precision` `cardinal_number` Si le domaine a un type numérique, cette colonne contient la précision (déclarée ou implicite) du type de cette colonne. Cette précision indique le nombre de chiffres significatifs. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2), comme indiqué dans la colonne `numeric_precision_radix`. Pour les autres types de données, cette colonne est NULL.
`numeric_precision_radix` `cardinal_number` Si le domaine a un type numérique, cette colonne indique la base des valeurs des colonnes `numeric_precision` et `numeric_scale`. La valeur est soit 2 soit 10. Pour tous les autres types de données, cette colonne est NULL.
`numeric_scale` `cardinal_number` Si le domaine contient un type numeric, cette colonne contient l'échelle (déclarée ou implicite) du type pour cette colonne. L'échelle indique le nombre de chiffres significatifs

Type de la colonne Description
à droite du point décimal. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2), comme indiqué dans la colonne `numeric_precision_radix`. Pour tous les autres types de données, cette colonne est NULL.
`datetime_precision` `cardinal_number` Si le domaine contient un type date, heure ou intervalle, la précision déclarée ; NULL pour les autres types de données ou si la précision n'a pas été déclarée.
`interval_type` `character_data` Si `data_type` identifie un type d'intervalle, cette colonne contient la spécification des champs que les intervalles incluent pour ce domaine, par exemple `YEAR TO MONTH`, `DAY TO SECOND`, etc. Si aucune restriction de champs n'est spécifiée (autrement dit, l'intervalle accepte tous les champs) et pour tous les autres types de données, ce champ est NULL.
`interval_precision` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL (voir `datetime_precision` pour la précision en fraction des secondes des domaines de type d'intervalle)
`domain_default` `character_data` Expression par défaut du domaine
`udt_catalog` `sql_identifier` Nom de la base de données dans laquelle est défini le type de données domaine (toujours la base de données courante)
`udt_schema` `sql_identifier` Nom du schéma où le type de données domaine est défini
`udt_name` `sql_identifier` Nom du type de données domaine
`scope_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`scope_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`scope_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`maximum_cardinality` `cardinal_number` Toujours NULL car les tableaux n'ont pas de limite maximale de cardinalité dans PostgreSQL
`dtd_identifier` `sql_identifier` Un identifiant du descripteur de type de données du domaine, unique parmi les descripteurs de type de données restant dans le domaine (ce qui est trivial car un domaine contient seulement un descripteur de type de données). Ceci est principalement utile pour joindre d'autres instances de tels identifiants (le format spécifique de l'identifiant n'est pas défini et il n'est pas garanti qu'il restera identique dans les versions futures).

37.24. `element_types`

La vue `element_types` contient les descripteurs de type de données des éléments de tableaux. Lorsqu'une colonne de table, un attribut de type composite, un domaine, un paramètre de fonction ou un code de retour de fonction est définie comme un type tableau, la vue respective du schéma d'information contient seulement `ARRAY` dans la colonne `data_type`. Pour obtenir des informations sur le type d'élément du tableau, il est possible de joindre la vue respective avec cette vue. Par exemple, pour afficher les colonnes d'une table avec les types de données et les types d'élément de tableau, si applicable, on peut écrire :

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN
 information_schema.element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE',
 c.dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name,
 e.object_type, e.collection_type_identifier))
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

Cette vue n'inclut que les objets auxquels l'utilisateur courant a accès, parce que propriétaire ou disposant de quelque privilège.

Tableau 37.22. Colonnes de `element_types`

Type de la colonne Description
`object_catalog` `sql_identifier` Nom de la base de données contenant l'objet qui utilise le tableau décrit (toujours la base de données courante)
`object_schema` `sql_identifier` Nom du schéma contenant l'objet utilisant le tableau décrit
`object_name` `sql_identifier` Nom de l'objet utilisant le tableau décrit
`object_type` `character_data` Le type de l'objet utilisant le tableau décrit : il fait partie de `TABLE` (le tableau est utilisé par une colonne de cette table), `USER-DEFINED TYPE` (le tableau est utilisé par un attribut de ce type composite), `DOMAIN` (le tableau est utilisé par ce domaine), `ROUTINE` (le tableau est utilisé par un paramètre ou le type du code de retour de cette fonction).
`collection_type_identifier` `sql_identifier` L'identifiant du descripteur de type de données du tableau décrit. Utilisez cette colonne pour faire une jointure avec les colonnes `dtd_identifier` des autres vues du schéma d'informations.
`data_type` `character_data` Le type de données des éléments du tableau s'il s'agit d'un type interne, sinon `USER-DEFINED` (dans ce cas, le type est identifié comme `udt_name` et dispose de colonnes associées).
`character_maximum_length` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL
`character_octet_length` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL
`character_set_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`character_set_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`character_set_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`collation_catalog` `sql_identifier`

Type de la colonne **Description**
Nom de la base contenant le collationnement du type de l'élément (toujours la base de données courante), NULL s'il s'agit du collationnement par défaut ou si le type de données de l'élément ne peut pas avoir de collationnement
`collation_schema` `sql_identifier` Nom du schéma contenant le collationnement du type de l'élément, NULL s'il s'agit du collationnement par défaut ou si le type de données de l'élément ne peut pas avoir de collationnement
`collation_name` `sql_identifier` Nom du collationnement du type de l'élément, NULL s'il s'agit du collationnement par défaut ou si le type de données de l'élément ne peut pas avoir de collationnement
`numeric_precision` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL
`numeric_precision_radix` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL
`numeric_scale` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL
`datetime_precision` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL
`interval_type` `character_data` Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL
`interval_precision` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données d'éléments de tableau dans PostgreSQL
`domain_default` `character_data` Pas encore implanté
`udt_catalog` `sql_identifier` Nom de la base de données pour lequel le type de données est défini (toujours la base de données courante)
`udt_schema` `sql_identifier` Nom du schéma dans lequel est défini le type de données des éléments
`udt_name` `sql_identifier` Nom du type de données des éléments
`scope_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`scope_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`scope_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`maximum_cardinality` `cardinal_number` Toujours NULL car les tableaux n'ont pas de limite maximale de cardinalité dans PostgreSQL
`dtd_identifier` `sql_identifier` Un identifiant du descripteur de type de données pour cet élément. Ceci n'est actuellement pas utile.

37.25. enabled_roles

La vue enabled_roles identifie les « rôles actuellement actifs ». Les rôles actifs sont définis récursivement comme l'utilisateur courant avec tous les rôles qui ont été donnés aux rôles activés avec l'héritage automatique. En d'autres termes, ce sont les rôles dont l'utilisateur courant est automatiquement membre, par héritage direct ou indirect.

Pour la vérification des permissions, l'ensemble des « rôles applicables » est appliqué, ce qui peut être plus large que l'ensemble des rôles actifs. Il est, de ce fait, généralement préférable d'utiliser la vue applicable_roles à la place de celle-ci ; Voir Section 37.5 pour des détails sur la vue applicable_roles.

Tableau 37.23. Colonnes de enabled_roles

Type de la colonne Description
role_name sql_identifier Nom d'un rôle

37.26. foreign_data_wrapper_options

La vue foreign_data_wrapper_options contient toutes les options définies par les wrappers de données distantes dans la base de données en cours. Seuls les wrappers accessibles par l'utilisateur connecté sont affichés (qu'il soit propriétaire ou qu'il ait des droits dessus).

Tableau 37.24. Colonnes de foreign_data_wrapper_options

Type de la colonne Description
foreign_data_wrapper_catalog sql_identifier Nom de la base de données dans laquelle est défini le wrapper de données distantes (toujours la base de connexion)
foreign_data_wrapper_name sql_identifier Nom du wrapper
option_name sql_identifier Nom d'une option
option_value character_data Valeur de l'option

37.27. foreign_data_wrappers

La vue foreign_data_wrappers contient tous les wrappers de données distantes définis dans le base de données en cours. Seuls sont affichés les wrappers pour lesquels l'utilisateur connecté a accès (qu'il soit propriétaire ou qu'il ait des droits dessus).

Tableau 37.25. Colonnes de foreign_data_wrappers

Type de la colonne Description
foreign_data_wrapper_catalog sql_identifier Nom de la base de données contenant le wrapper de données distantes (toujours la base de données en cours)
foreign_data_wrapper_name sql_identifier

Type de la colonne Description
Nom du wrapper
`authorization_identifier` `sql_identifier` Nom du propriétaire du serveur distant
`library_name` `character_data` Nom du fichier de la bibliothèque implémentant ce wrapper
`foreign_data_wrapper_language` `character_data` Langage utilisé pour implémenter ce wrapper

37.28. `foreign_server_options`

La vue `foreign_server_options` contient toutes les options définies pour les serveurs distants de la base de données en cours. Ne sont affichés que les serveurs distants pour lesquels l'utilisateur connecté a des droits (qu'il soit propriétaire ou qu'il ait quelques droits dessus).

Tableau 37.26. Colonnes de `foreign_server_options`

Type de la colonne Description
`foreign_server_catalog` `sql_identifier` Nom de la base de données contenant le serveur distant (toujours la base de données en cours)
`foreign_server_name` `sql_identifier` Nom du serveur distant
`option_name` `sql_identifier` Nom d'une option
`option_value` `character_data` Valeur de l'option

37.29. `foreign_servers`

La vue `foreign_servers` contient tous les serveurs distants définis dans la base en cours. Ne sont affichés que les serveurs distants pour lesquels l'utilisateur connecté a des droits (qu'il soit propriétaire ou qu'il ait quelques droits dessus).

Tableau 37.27. Colonnes de `foreign_servers`

Type de la colonne Description
`foreign_server_catalog` `sql_identifier` Nom de la base de données dans laquelle ce serveur distant est défini (toujours la base de données en cours)
`foreign_server_name` `sql_identifier` Nom du serveur distant
`foreign_data_wrapper_catalog` `sql_identifier` Nom de la base de données qui contient le wrapper de données distantes utilisé par le serveur distant (toujours la base de données en cours)
`foreign_data_wrapper_name` `sql_identifier` Nom du wrapper de données distantes utilisé par le serveur distant
`foreign_server_type` `character_data`

Type de la colonne Description
Information sur le type de serveur distant, si indiqué lors de la création
`foreign_server_version` `character_data` Information sur la version de serveur distant, si indiqué lors de la création
`authorization_identifier` `sql_identifier` Nom du propriétaire du serveur distant

37.30. `foreign_table_options`

La vue `foreign_table_options` contient toutes les options définies pour les tables distantes de la base de données courante. Seules sont affichées les tables distantes accessibles par l'utilisateur courant (soit parce qu'il en est le propriétaire soit parce qu'il dispose de droits particuliers).

Tableau 37.28. Colonnes de `foreign_table_options`

Type de la colonne Description
`foreign_table_catalog` `sql_identifier` Nom de la base de données qui contient la table distante (toujours la base de données courante)
`foreign_table_schema` `sql_identifier` Nom du schéma contenant la table distante
`foreign_table_name` `sql_identifier` Nom de la table distante
`option_name` `sql_identifier` Nom d'une option
`option_value` `character_data` Valeur de l'option

37.31. `foreign_tables`

La vue `foreign_tables` contient toutes les tables distantes définies dans la base de données courantes. Seules sont affichées les tables distantes accessibles par l'utilisateur courant (soit parce qu'il en est le propriétaire soit parce qu'il dispose de droits particuliers).

Tableau 37.29. Colonnes de `foreign_tables`

Type de la colonne Description
`foreign_table_catalog` `sql_identifier` Nom de la base de données qui contient la table distante (toujours la base de données courante)
`foreign_table_schema` `sql_identifier` Nom du schéma contenant la table distante
`foreign_table_name` `sql_identifier` Nom de la table distante
`foreign_server_catalog` `sql_identifier` Nom de la base de données où le serveur distant est défini (toujours la base de données courante)
`foreign_server_name` `sql_identifier` Nom du serveur distant

37.32. key_column_usage

La vue key_column_usage identifie toutes les colonnes de la base de données courante restreintes par une contrainte unique, clé primaire ou clé étrangère. Les contraintes de vérification ne sont pas incluses dans cette vue. Seules sont affichées les colonnes auxquelles l'utilisateur a accès, parce qu'il est le propriétaire de la table ou qu'il dispose de quelque privilège.

Tableau 37.30. Colonnes de key_column_usage

Type de la colonne Description
constraint_catalog sql_identifier Nom de la base de données contenant la contrainte (toujours la base de données courante)
constraint_schema sql_identifier Nom du schéma contenant la contrainte
constraint_name sql_identifier Nom de la contrainte
table_catalog sql_identifier Nom de la base de données contenant la table contenant la colonne contrainte (toujours la base de données courante)
table_schema sql_identifier Nom du schéma contenant la table contenant la colonne contrainte
table_name sql_identifier Nom de la table contenant la colonne contrainte
column_name sql_identifier Nom de la colonne contrainte
ordinal_position cardinal_number Position ordinale de la colonne dans la clé de contrainte (la numérotation commence à 1)
position_in_unique_constraint cardinal_number Pour une contrainte de type clé étrangère, la position ordinale de la colonne référencée dans sa contrainte d'unicité (la numérotation commence à 1) ; sinon null

37.33. parameters

La vue parameters contient des informations sur les paramètres (arguments) de toutes les fonctions de la base de données courante. Seules sont affichées les fonctions auxquelles l'utilisateur courant a accès, parce qu'il en est le propriétaire ou qu'il dispose de quelque privilège.

Tableau 37.31. Colonnes de parameters

Type de la colonne Description
specific_catalog sql_identifier Nom de la base de données contenant la fonction (toujours la base de données courante)
specific_schema sql_identifier Nom du schéma contenant la fonction
specific_name sql_identifier Le « nom spécifique » de la fonction. Voir la Section 37.45 pour plus d'informations.
ordinal_position cardinal_number Position ordinale du paramètre dans la liste des arguments de la fonction (la numérotation commence à 1)

Type de la colonne **Description**
`parameter_mode` `character_data` `IN` pour les paramètres en entrée, `OUT` pour les paramètres en sortie ou `INOUT` pour les paramètres en entrée/sortie.
`is_result` `yes_or_no` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`as_locator` `yes_or_no` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`parameter_name` `sql_identifier` Nom du paramètre ou NULL si le paramètre n'a pas de nom
`data_type` `character_data` Type de données du paramètre s'il s'agit d'un type interne, ou `ARRAY` s'il s'agit d'un tableau (dans ce cas, voir la vue `element_types`), sinon `USER-DEFINED` (dans ce cas, le type est identifié dans `udt_name` et dispose de colonnes associées).
`character_maximum_length` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL
`character_octet_length` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL
`character_set_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`character_set_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`character_set_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`collation_catalog` `sql_identifier` Toujours NULL car cette information n'est pas appliquée pour configurer les types de données dans PostgreSQL
`collation_schema` `sql_identifier` Toujours NULL car cette information n'est pas appliquée pour configurer les types de données dans PostgreSQL
`collation_name` `sql_identifier` Toujours NULL car cette information n'est pas appliquée pour configurer les types de données dans PostgreSQL
`numeric_precision` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL
`numeric_precision_radix` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL
`numeric_scale` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL
`datetime_precision` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL
`interval_type` `character_data` Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL

Type de la colonne Description
`interval_precision` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL
`udt_catalog` `sql_identifier` Nom de la base de données sur laquelle est défini le paramètre (toujours la base de données courante)
`udt_schema` `sql_identifier` Nom du schéma dans lequel est défini le type de données du paramètre
`udt_name` `sql_identifier` Nom du type de données du paramètre
`scope_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`scope_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`scope_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`maximum_cardinality` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données dans PostgreSQL
`dtd_identifier` `sql_identifier` Un identifiant du descripteur de type de données du paramètre, unique parmi les descripteurs de type de données restant dans la fonction. Ceci est principalement utile pour réaliser une jointure avec les autres instances de tels identifiants (le format spécifique de l'identifiant n'est pas défini et il n'est pas garanti qu'il reste identique dans les prochaines versions).
`parameter_default` `character_data` L'expression par défaut du paramètre, ou NULL si aucune ou si la fonction n'a pas pour propriétaire un des rôles actuellement activés.

37.34. `referential_constraints`

La vue `referential_constraints` contient toutes les contraintes référentielles (clés étrangères) au sein de la base de données courante. Seuls sont affichés les contraintes pour lesquelles l'utilisateur connecté a accès en écriture sur la table référençante (parce qu'il est le propriétaire ou qu'il a d'autres droits que `SELECT`).

Tableau 37.32. Colonnes de `referential_constraints`

Type de la colonne Description
`constraint_catalog` `sql_identifier` Nom de la base de données contenant la contrainte (toujours la base de données courante)
`constraint_schema` `sql_identifier` Nom du schéma contenant la contrainte
`constraint_name` `sql_identifier` Nom de la contrainte
`unique_constraint_catalog` `sql_identifier` Nom de la base de données contenant la contrainte d'unicité ou de clé primaire que la contrainte de clé étrangère référence (toujours la base de données courante)

Type de la colonne Description
`unique_constraint_schema` `sql_identifier` Nom du schéma contenant la contrainte d'unicité ou de clé primaire que la contrainte de clé étrangère référence
`unique_constraint_name` `sql_identifier` Nom de la contrainte d'unicité ou de clé primaire que la contrainte de clé étrangère référence
`match_option` `character_data` Correspondances de la contrainte de clé étrangère : `FULL`, `PARTIAL` ou `NONE`.
`update_rule` `character_data` Règle de mise à jour associée à la contrainte de clé étrangère : `CASCADE`, `SET NULL`, `SET DEFAULT`, `RESTRICT` ou `NO ACTION`.
`delete_rule` `character_data` Règle de suppression associée à la contrainte de clé étrangère : `CASCADE`, `SET NULL`, `SET DEFAULT`, `RESTRICT` ou `NO ACTION`.

37.35. `role_column_grants`

La vue `role_column_grants` identifie tous les privilèges de colonne octroyés pour lesquels le donneur ou le bénéficiaire est un rôle actuellement actif. Plus d'informations sous `column_privileges`. La seule différence réelle entre cette vue et `column_privileges` est que cette vue omet les colonnes qui ont été rendues accessibles à l'utilisateur actuel en utilisant la commande GRANT pour `PUBLIC`.

Tableau 37.33. Colonnes de `role_column_grants`

Type de la colonne Description
`grantor` `sql_identifier` Nom du rôle qui a octroyé le privilège
`grantee` `sql_identifier` Nom du rôle bénéficiaire
`table_catalog` `sql_identifier` Nom de la base de données qui contient la table qui contient la colonne (toujours la base de données courante)
`table_schema` `sql_identifier` Nom du schéma qui contient la table qui contient la colonne
`table_name` `sql_identifier` Nom de la table qui contient la colonne
`column_name` `sql_identifier` Nom de la colonne
`privilege_type` `character_data` Type de privilège : `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES` ou `TRIGGER`
`is_grantable` `yes_or_no` `YES` si le droit peut être transmis, `NO` sinon

37.36. `role_routine_grants`

La vue `role_routine_grants` identifie tous les privilèges de routine octriyés lorsque le donneur ou le bénéficiaire est un rôle actif. Plus d'informations sous `routine_privileges`. La seule

différence réelle entre cette vue et `routine_privileges` est que cette vue omet les colonnes qui ont été rendues accessibles à l'utilisateur actuel en utilisant la commande GRANT pour `PUBLIC`.

Tableau 37.34. Colonnes de `role_routine_grants`

Type de la colonne **Description**
`grantor` `sql_identifier` Nom du rôle qui a octroyé le privilège
`grantee` `sql_identifier` Nom du rôle bénéficiaire
`specific_catalog` `sql_identifier` Nom de la base de données qui contient la fonction (toujours la base de données courante)
`specific_schema` `sql_identifier` Nom du schéma qui contient la fonction
`specific_name` `sql_identifier` Le « nom spécifique » de la fonction. Voir la Section 37.45 pour plus d'informations.
`routine_catalog` `sql_identifier` Nom de la base de données qui contient la fonction (toujours la base de données courante)
`routine_schema` `sql_identifier` Nom du schéma qui contient la fonction
`routine_name` `sql_identifier` Nom de la fonction (peut être dupliqué en cas de surcharge)
`privilege_type` `character_data` Toujours `EXECUTE` (seul type de privilège sur les fonctions)
`is_grantable` `yes_or_no` `YES` si le droit peut être transmis, `NO` sinon

37.37. `role_table_grants`

La vue `role_table_grants` identifie tous les privilèges de tables octroyés lorsque le donneur ou le bénéficiaire est un rôle actif. Plus d'informations sous `table_privileges`. La seule différence réelle entre cette vue et `table_privileges` est que cette vue omet les colonnes qui ont été rendues accessibles à l'utilisateur actuel en utilisant la commande GRANT pour `PUBLIC`.

Tableau 37.35. Colonnes de `role_table_grants`

Type de la colonne **Description**
`grantor` `sql_identifier` Nom du rôle qui a octroyé le privilège
`grantee` `sql_identifier` Nom du rôle bénéficiaire
`table_catalog` `sql_identifier` Nom de la base de données qui contient la table (toujours la base de données courante)
`table_schema` `sql_identifier` Nom du schéma qui contient la table
`table_name` `sql_identifier` Nom de la table

Type de la colonne Description
`privilege_type` `character_data` Type du privilège : `SELECT`, `DELETE`, `INSERT`, `UPDATE`, `REFERENCES` ou `TRIGGER`
`is_grantable` `yes_or_no` `YES` si le droit peut être transmis, `NO` sinon
`with_hierarchy` `yes_or_no` Dans le standard SQL, `WITH HIERARCHY OPTION` est un (sous-)droit séparé autorisant certaines opérations sur la hiérarchie de l'héritage des tables. Dans PostgreSQL, ceci est inclus dans le droit `SELECT`, donc cette colonne affiche `YES` si le droit est `SELECT`, et `NO` sinon.

37.38. `role_udt_grants`

La vue `role_udt_grants` a pour but d'identifier les droits `USAGE` donnés pour des types définis par l'utilisateur pour lesquels celui qui donne le droit et celui qui le reçoit sont des rôles actuellement activés. Plus d'informations sont disponibles dans `udt_privileges`. La seule réelle différence entre cette vue et `udt_privileges` est que cette vue omet les objets qui ont été rendus accessibles à l'utilisateur courant par le biais du pseudo-rôle `PUBLIC`. Comme les types de données n'ont pas vraiment de droits dans PostgreSQL, et disposent seulement d'un droit implicite à `PUBLIC`, cette vue est vide.

Tableau 37.36. Colonnes de `role_udt_grants`

Column Type Description
`grantor` `sql_identifier` Le nom du rôle qui a donné le droit
`grantee` `sql_identifier` Le nom du rôle à qui le droit a été donné
`udt_catalog` `sql_identifier` Nom de la base contenant le type (toujours la base de données courante)
`udt_schema` `sql_identifier` Nom du schéma contenant le type
`udt_name` `sql_identifier` Nom du type
`privilege_type` `character_data` Toujours `TYPE USAGE`
`is_grantable` `yes_or_no` `YES` si le droit peut être donné, `NO` sinon

37.39. `role_usage_grants`

La vue `role_usage_grants` identifie les privilèges d'`USAGE` sur différents types d'objets où le donneur ou le receveur des droits est un rôle actuellement activé. Plus d'informations sous `usage_privileges`. Dans le futur, cette vue pourrait contenir des informations plus utiles. La seule différence réelle entre cette vue et `usage_privileges` est que cette vue omet les colonnes qui ont été rendues accessibles à l'utilisateur actuel en utilisant la commande GRANT pour `PUBLIC`.

Tableau 37.37. Colonnes de `role_usage_grants`

Type de la colonne Description
`grantor` `sql_identifier` Nom du rôle qui a octroyé le privilège
`grantee` `sql_identifier` Nom du rôle bénéficiaire
`object_catalog` `sql_identifier` Nom de la base de données qui contient l'objet (toujours la base de données courante)
`object_schema` `sql_identifier` Nom du schéma qui contient l'objet, if applicable, sinon une chaîne vide
`object_name` `sql_identifier` Nom de l'objet
`object_type` `character_data` `COLLATION`, `DOMAIN`, `FOREIGN DATA WRAPPER`, `FOREIGN SERVER` ou `SEQUENCE`
`privilege_type` `character_data` Toujours `USAGE`
`is_grantable` `yes_or_no` `YES` si le droit peut être transmis, `NO` sinon

37.40. `routine_column_usage`

La vue `routine_column_usage` identifie toutes les colonnes qui sont utilisées par une fonction ou une procédure, soit dans le corps SQL soit dans les expressions par défaut des paramètres. (Ceci fonctionne seulement pour les corps SQL sans guillemets, les corps sans guillemets ou les fonctions dans d'autres langages.) Une colonne est uniquement inclus si sa table a pour propriétaire un rôle actuellement activé.

Tableau 37.38. Colonnes de `routine_column_usage`

Type de colonne Description
`specific_catalog` `sql_identifier` Nom de la base de données contenant la fonction (toujours la base courante)
`specific_schema` `sql_identifier` Nom du schéma contenant la fonction
`specific_name` `sql_identifier` Le « nom spécifique » de la fonction. Voir Section 37.45 pour plus d'informations.
`routine_catalog` `sql_identifier` Nom de la base de données contenant la fonction (toujours la base courante)
`routine_schema` `sql_identifier` Nom du schéma contenant la fonction
`routine_name` `sql_identifier` Nom de la fonction (peut être dupliqué en cas de surcharge)
`table_catalog` `sql_identifier` Nom de la base de données qui contient la table qui est utilisée par la fonction (toujours la base courante)
`table_schema` `sql_identifier` Nom du schéma qui contient la table qui est utilisée par la fonction

Type de colonne Description
`table_name` `sql_identifier` Nom de la table utilisée par la fonction
`column_name` `sql_identifier` Nom de la colonne utilisée par la fonction

37.41. `routine_privileges`

La vue `routine_privileges` identifie tous les droits sur les fonctions à un rôle actuellement activé ou par un rôle actuellement activé. Il existe une ligne pour chaque combinaison fonction, donneur, bénéficiaire.

Tableau 37.39. Colonnes de `routine_privileges`

Type de la colonne Description
`grantor` `sql_identifier` Nom du rôle qui a accordé le privilège
`grantee` `sql_identifier` Nom du rôle bénéficiaire
`specific_catalog` `sql_identifier` Nom de la base de données qui contient la fonction (toujours la base de données courante)
`specific_schema` `sql_identifier` Nom du schéma qui contient la fonction
`specific_name` `sql_identifier` Le « nom spécifique » de la fonction. Voir la Section 37.45 pour plus d'informations.
`routine_catalog` `sql_identifier` Nom de la base de données qui contient la fonction (toujours la base de données courante)
`routine_schema` `sql_identifier` Nom du schéma qui contient la fonction
`routine_name` `sql_identifier` Nom de la fonction (peut être dupliqué en cas de surcharge)
`privilege_type` `character_data` Toujours `EXECUTE` (seul priilège de fonctions)
`is_grantable` `yes_or_no` `YES` si le droit peut être transmis, `NO` sinon

37.42. `routine_routine_usage`

La vue `routine_routine_usage` identifie toutes les fonctions ou procédures utilisées par une autre (ou la même) fonction ou procédure, soit dans le corps SQL ou dans les expressions par défaut de paramètre. (Ceci fonctionne seulement dans le corps SQL sans guillemets, les corps sans guillemets ou les fonctions dans d'autres langages.) Une entrée est incluse ici seulement si la fonction utilisée appartient à un rôle actif maintenant. (Il n'y a aucune restriction sur la fonction appelante.)

Notez que les entrées pour les deux fonctions dans la vue référencent le nom « spécifique » de la routine, même si les noms de colonnes sont utilisées d'une manière incohérente avec les autres vues du schéma d'information liées aux routines. Ceci vient du standard SQL, même si cela est une conception discutable. Voir Section 37.45 pour plus d'information au sujet des noms spécifiques.

Tableau 37.40. Colonnes de `routine_routine_usage`

Type de la colonne Description
`specific_catalog` `sql_identifier` Nom de la base de données qui contient la fonction utilisée (toujours la base courante)
`specific_schema` `sql_identifier` Nom du schéma contenant la fonction utilisée
`specific_name` `sql_identifier` Le « nom spécifique » de la fonction utilisée
`routine_catalog` `sql_identifier` Nom de la base de données qui contient la fonction utilisée par la première fonction (toujours la base courante)
`routine_schema` `sql_identifier` Nom du schéma qui contient la fonction utilisée par la première fonction
`routine_name` `sql_identifier` Le « nom spécifique » de la fonction utilisée par la première fonction.

37.43. `routine_sequence_usage`

La vue `routine_sequence_usage` identifie toutes les séquences qui sont utilisées par une fonction ou une procédure, soit dans le corps SQL soit dans les expressions par défaut des paramètres. (Ceci fonctionne seulement pour les corps SQL sans guillmets, les corps sans guillemets les fonctions dans d'autres langages.) Une séquence est seulement incluse si la séquence est la propriété d'un rôle actuellement actif.

Tableau 37.41. Colonnes de `routine_sequence_usage`

Type de la colonne Description
`specific_catalog` `sql_identifier` Nom de la base de données contenant la fonction (toujours la base courante)
`specific_schema` `sql_identifier` Nom du schéma contenant la fonction
`specific_name` `sql_identifier` Le « nom spécifique » de la fonction. Voir Section 37.45 pour plus d'information
`routine_catalog` `sql_identifier` Nom de la base de données contenant la fonction (toujours la base courante)
`routine_schema` `sql_identifier` Nom du schéma contenant la fonction
`routine_name` `sql_identifier` Nom de la fonction (peut être dupliqué en cas de surcharge)
`schema_catalog` `sql_identifier` Nom de la base de données qui contient la séquence utilisée par la fonction (toujours la base courante)
`sequence_schema` `sql_identifier` Nom du schéma qui contient la séquence utilisée par la fonction
`sequence_name` `sql_identifier` Nom de la séquence utilisée par la fonction

37.44. routine_table_usage

La vue routine_table_usage a pour but d'identifier toutes les tables utilisées dans une fonction ou procédure. Cette information n'est actuellement pas tracée par PostgreSQL.

Tableau 37.42. Colonnes de routine_table_usage

Type de la colonne Description
specific_catalog sql_identifier Nom de la base de données contenant la fonction (toujours la base courante)
specific_schema sql_identifier Nom du schéma contenant la fonction
specific_name sql_identifier Le « nom spécifique » de la fonction. Voir Section 37.45 pour plus d'information
routine_catalog sql_identifier Nom de la base de données contenant la fonction (toujours la base courante)
routine_schema sql_identifier Nom du schéma contenant la fonction
routine_name sql_identifier Nom de la fonction (peut être dupliquée en cas de surcharge)
table_catalog sql_identifier Nom de la base de données qui contient la table utilisée par la fonction (toujours la base courante)
table_schema sql_identifier Nom du schéma qui contient la table utilisée par la fonction
table_name sql_identifier Nom de la table utilisée par la fonction

37.45. routines

La vue routines contient toutes les fonctions de la base de données courante. Seules sont affichées les fonctions auxquelles l'utilisateur courant a accès (qu'il en soit le propriétaire ou dispose de de privilèges).

Tableau 37.43. Colonnes de routines

Type de la colonne Description
specific_catalog sql_identifier Nom de la base de données qui contient la fonction (toujours la base de données courante)
specific_schema sql_identifier Nom du schéma qui contient la fonction
specific_name sql_identifier Le « nom spécifique » de la fonction. Ce nom identifie de façon unique la fonction dans le schéma, même si le nom réel de la fonction est surchargé. Le format du nom spécifique n'est pas défini, il ne devrait être utilisé que dans un but de comparaison avec d'autres instances de noms spécifiques de routines.
routine_catalog sql_identifier Nom de la base de données qui contient la fonction (toujours la base de données courante)

Type de la colonne **Description**
`routine_schema` `sql_identifier` Nom du schéma qui contient la fonction
`routine_name` `sql_identifier` Nom de la fonction (peut être dupliqué en cas de surcharge)
`routine_type` `character_data` `FUNCTION` pour une fonction, `PROCEDURE` pour une procédure
`module_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`module_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`module_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`udt_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`udt_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`udt_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`data_type` `character_data` Type de données de retour de la fonction s'il est interne, `ARRAY` s'il s'agit d'un tableau (dans ce cas, voir la vue `element_types`), sinon `USER-DEFINED` (dans ce cas, le type est identifié dans `type_udt_name` et dispose de colonnes associées). NULL pour une procédure.
`character_maximum_length` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL
`character_octet_length` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL
`character_set_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`character_set_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`character_set_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`collation_catalog` `sql_identifier` Toujours NULL car cette information n'est pas appliquée pour configurer les types de données dans PostgreSQL
`collation_schema` `sql_identifier` Toujours NULL car cette information n'est pas appliquée pour configurer les types de données dans PostgreSQL
`collation_name` `sql_identifier` Toujours NULL car cette information n'est pas appliquée pour configurer les types de données dans PostgreSQL
`numeric_precision` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL
`numeric_precision_radix` `cardinal_number`

Type de la colonne **Description**
Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL
`numeric_scale` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL
`datetime_precision` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL
`interval_type` `character_data` Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL
`interval_precision` `cardinal_number` Toujours NULL car cette information n'est pas appliquée aux types de données renvoyées sous PostgreSQL
`type_udt_catalog` `sql_identifier` Nom de la base de données dans laquelle est défini le type de données de retour de la fonction (toujours la base de données courante). NULL pour une procédure.
`type_udt_schema` `sql_identifier` Nom du schéma dans lequel est défini le type de données de retour de la fonction. NULL pour une procédure.
`type_udt_name` `sql_identifier` Nom du type de données de retour de la fonction. NULL pour une procédure.
`scope_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`scope_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`scope_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`maximum_cardinality` `cardinal_number` Toujours NULL car il n'y a pas de limite maximale à la cardinalité des tableaux dans PostgreSQL
`dtd_identifier` `sql_identifier` Un identifiant du descripteur de type de données du type de données retour, unique parmi les descripteurs de type de données de la fonction. Ceci est principalement utile pour la jointure avec d'autres instances de tels identifiants (le format spécifique de l'identifiant n'est pas défini et il n'est pas certain qu'il restera identique dans les versions futures).
`routine_body` `character_data` Si la fonction est une fonction SQL, alors `SQL`, sinon `EXTERNAL`.
`routine_definition` `character_data` Le texte source de la fonction (NULL si la fonction n'appartient pas à un rôle actif). (Le standard SQL précise que cette colonne n'est applicable que si `routine_body` est `SQL`, mais sous PostgreSQL ce champ contient tout texte source précisé à la création de la fonction.)
`external_name` `character_data` Si la fonction est une fonction C, le nom externe (link symbol) de la fonction ; sinon NULL. (Il s'agit de la même valeur que celle affichée dans `routine_definition`).
`external_language` `character_data` Le langage d'écriture de la fonction
`parameter_style` `character_data`

Type de la colonne **Description**
Toujours `GENERAL` (le standard SQL définit d'autres styles de paramètres qui ne sont pas disponibles avec PostgreSQL).
`is_deterministic` `yes_or_no` Si la fonction est déclarée immuable (déterministe dans le standard SQL), alors `YES`, sinon `NO`. (Les autres niveaux de volatilité disponibles dans PostgreSQL ne peuvent être récupérés via le schéma d'informations).
`sql_data_access` `character_data` Toujours `MODIFIES`, ce qui signifie que la fonction peut modifier les données SQL. Cette information n'est pas utile sous PostgreSQL.
`is_null_call` `yes_or_no` Si la fonction renvoie automatiquement NULL si un de ces arguments est NULL, alors `YES`, sinon `NO`. NULL pour une procédure.
`sql_path` `character_data` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`schema_level_routine` `yes_or_no` Toujours `YES`. (L'opposé serait une méthode d'un type utilisateur, fonctionnalité non disponible dans PostgreSQL).
`max_dynamic_result_sets` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`is_user_defined_cast` `yes_or_no` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`is_implicitly_invocable` `yes_or_no` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`security_type` `character_data` Si la fonction est exécutée avec les droits de l'utilisateur courant, alors `INVOKER`. Si la fonction est exécutée avec les droits de l'utilisateur l'ayant définie, alors `DEFINER`.
`to_sql_specific_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`to_sql_specific_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`to_sql_specific_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`as_locator` `yes_or_no` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`created` `time_stamp` S'applique à une fonctionnalité non disponible dans PostgreSQL
`last_altered` `time_stamp` S'applique à une fonctionnalité non disponible dans PostgreSQL
`new_savepoint_level` `yes_or_no` S'applique à une fonctionnalité non disponible dans PostgreSQL
`is_udt_dependent` `yes_or_no` Actuellement toujours `NO`. `YES` s'applique à une fonctionnalité non disponible dans PostgreSQL.
`result_cast_from_data_type` `character_data` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_as_locator` `yes_or_no` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_char_max_length` `cardinal_number`

Type de la colonne **Description**
S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_char_octet_length` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_char_set_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_char_set_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_char_set_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_collation_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_collation_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_collation_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_numeric_precision` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_numeric_precision_radix` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_numeric_scale` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_datetime_precision` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_interval_type` `character_data` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_interval_precision` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_type_udt_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_type_udt_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_type_udt_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_scope_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_scope_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_scope_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_maximum_cardinality` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`result_cast_dtd_identifier` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL

37.46. schemata

La vue schemata contient tous les schémas de la base de données courante auxquels l'utilisateur courant a accès (soit en étant le propriétaire soit en ayant des privilèges).

Tableau 37.44. Colonnes de schemata

Type de la colonne Description
catalog_name sql_identifier Nom de la base de données dans laquelle se trouve le schéma (toujours la base de données courante)
schema_name sql_identifier Nom du schéma
schema_owner sql_identifier Nom du propriétaire du schéma
default_character_set_catalog sql_identifier S'applique à une fonctionnalité non disponible dans PostgreSQL
default_character_set_schema sql_identifier S'applique à une fonctionnalité non disponible dans PostgreSQL
default_character_set_name sql_identifier S'applique à une fonctionnalité non disponible dans PostgreSQL
sql_path character_data S'applique à une fonctionnalité non disponible dans PostgreSQL

37.47. sequences

La vue sequences contient toutes les séquences définies dans la base courante. Seules sont affichées les séquences auxquelles l'utilisateur courant a accès (qu'il en soit le propriétaire ou dispose de privilèges).

Tableau 37.45. Colonnes de sequences

Type de la colonne Description
sequence_catalog sql_identifier Nom de la base qui contient la séquence (toujours la base en cours)
sequence_schema sql_identifier Nom du schéma qui contient la séquence
sequence_name sql_identifier Nom de la séquence
data_type character_data Type de données de la séquence.
numeric_precision cardinal_number Cette colonne contient la précision (déclarée ou implicite) du type de données de la séquence (voir ci-dessus). La précision indique le nombre de chiffres significatifs. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2), suivant ce qui est indiqué dans la colonne numeric_precision_radix.
numeric_precision_radix cardinal_number Cette colonne indique dans quelle base les valeurs de la colonne numeric_precision et numeric_scale sont exprimées, 2 ou 10.

Type de la colonne Description
`numeric_scale` `cardinal_number` Cette colonne contient l'échelle (déclarée ou implicite) du type de données de la séquence (voir ci-dessus). L'échelle indique le nombre de chiffres significatifs à droite du point décimale. Elle peut être exprimée en décimal (base 10) ou en binaire (base 2), suivant ce qui est indiqué dans la colonne `numeric_precision_radix`.
`start_value` `character_data` La valeur de démarrage de la séquence
`minimum_value` `character_data` La valeur minimale de la séquence
`maximum_value` `character_data` La valeur maximale de la séquence
`increment` `character_data` L'incrément de la séquence
`cycle_option` `yes_or_no` `YES` si la séquence est cyclique, `NO` dans le cas contraire

Notez qu'en accord avec le standard SQL, les valeurs de démarrage, minimale, maximale et d'incrément sont renvoyées en tant que chaînes de caractères.

37.48. `sql_features`

La table `sql_features` contient des informations sur les fonctionnalités officielles définies dans le standard SQL et supportées par PostgreSQL. Ce sont les mêmes informations que celles présentées dans l'Annexe D. D'autres informations de fond y sont disponibles.

Tableau 37.46. Colonnes de `sql_features`

Type de la colonne Description
`feature_id` `character_data` Chaîne identifiant la fonctionnalité
`feature_name` `character_data` Nom descriptif de la fonctionnalité
`sub_feature_id` `character_data` Chaîne identifiant la sous-fonctionnalité ou chaîne de longueur NULL s'il ne s'agit pas d'une sous-fonctionnalité
`sub_feature_name` `character_data` Nom descriptif de la sous-fonctionnalité ou chaîne de longueur NULL s'il ne s'agit pas d'une sous-fonctionnalité
`is_supported` `yes_or_no` `YES` si la fonctionnalité est complètement supportée par la version actuelle de PostgreSQL, `NO` sinon
`is_verified_by` `character_data` Toujours NULL car le groupe de développement PostgreSQL ne réalise pas de tests formels sur la conformité des fonctionnalités
`comments` `character_data` Un commentaire éventuel sur le statut du support de la fonctionnalité

37.49. sql_implementation_info

La table sql_inplementation_info contient des informations sur différents aspects que le standard SQL laisse à la discrétion de l'implantation. Ces informations n'ont de réel intérêt que dans le contexte de l'interface ODBC ; les utilisateurs des autres interfaces leur trouveront certainement peu d'utilité. Pour cette raison, les éléments décrivant l'implantation ne sont pas décrits ici ; ils se trouvent dans la description de l'interface ODBC.

Tableau 37.47. Colonnes de sql_implementation_info

Type de la colonne Description
implementation_info_id character_data Chaîne identifiant l'élément d'information d'implantation
implementation_info_name character_data Nom descriptif de l'élément d'information d'implantation
integer_value cardinal_number Valeur de l'élément d'information d'implantation, ou NULL si la valeur est contenue dans la colonne character_value
character_value character_data Valeur de l'élément d'information d'implantation, ou NULL si la valeur est contenue dans la colonne integer_value
comments character_data Un commentaire éventuel de l'élément d'information d'implantation

37.50. sql_parts

La table sql_parts contient des informations sur les parties du standard SQL supportées par PostgreSQL.

Tableau 37.48. Colonnes de sql_parts

Type de la colonne Description
feature_id character_data Une chaîne d'identification contenant le numéro de la partie
feature_name character_data Nom descriptif de la partie
is_supported yes_or_no YES si cette partie est complètement supportée par la version actuelle de PostgreSQL, NO dans le cas contraire
is_verified_by character_data Toujours NULL, car les développeurs PostgreSQL ne testent pas officiellement la conformité des fonctionnalités
comments character_data Commentaires sur le statut du support de la partie

37.51. sql_sizing

La table sql_sizing contient des informations sur les différentes limites de tailles et valeurs maximales dans PostgreSQL. Ces informations ont pour contexte principal l'interface ODBC ; les

utilisateurs des autres interfaces leur trouveront probablement peu d'utilité. Pour cette raison, les éléments de taille individuels ne sont pas décrits ici ; ils se trouvent dans la description de l'interface ODBC.

Tableau 37.49. Colonnes de `sql_sizing`

Type de la colonne **Description**
`sizing_id` `cardinal_number` Identifiant de l'élément de taille
`sizing_name` `character_data` Nom descriptif de l'élément de taille
`supported_value` `cardinal_number` Valeur de l'élément de taille, ou 0 si la taille est illimitée ou ne peut pas être déterminée, ou NULL si les fonctionnalités pour lesquelles l'élément de taille est applicable ne sont pas supportées
`comments` `character_data` Un commentaire éventuel de l'élément de taille

37.52. `table_constraints`

La vue `table_constraints` contient toutes les contraintes appartenant aux tables possédées par l'utilisateur courant ou pour lesquelles l'utilisateur courant dispose de certains droits différents de SELECT.

Tableau 37.50. Colonnes de `table_constraints`

Type de la colonne **Description**
`constraint_catalog` `sql_identifier` Nom de la base de données qui contient la contrainte (toujours la base de données courante)
`constraint_schema` `sql_identifier` Nom du schéma qui contient la contrainte
`constraint_name` `sql_identifier` Nom de la contrainte
`table_catalog` `sql_identifier` Nom de la base de données qui contient la table (toujours la base de données courante)
`table_schema` `sql_identifier` Nom du schéma qui contient la table
`table_name` `sql_identifier` Nom de la table
`constraint_type` `character_data` Type de contrainte : `CHECK`, `FOREIGN KEY`, `PRIMARY KEY` ou `UNIQUE`
`is_deferrable` `yes_or_no` `YES` si la contrainte peut être différée, `NO` sinon
`initially_deferred` `yes_or_no` `YES` si la contrainte, qui peut être différée, est initialement différée, `NO` sinon
`enforced` `yes_or_no` S'applique à une fonctionnalite non disponible dans PostgreSQL (actuellement, toujours à `YES`)
`nulls_distinct` `yes_or_no`

Type de la colonne Description
Si la contrainte est une contrainte d'unicité, alors YES si la contrainte traite les NULL de façon distincte ou NO si elle traite les NULL de façon non distincte. NULL pour tous les autres types de contraintes.

37.53. table_privileges

La vue table_privileges identifie tous les privilèges accordés, à un rôle actif ou par une rôle actif, sur des tables ou vues. Il y a une ligne par combinaison table, donneur, bénéficiaire.

Tableau 37.51. Colonnes de table_privileges

Type de la colonne Description
grantor sql_identifier Nom du rôle qui a accordé le privilège
grantee sql_identifier Nom du rôle bénéficiaire
table_catalog sql_identifier Nom de la base de données qui contient la table (toujours la base de données courante)
table_schema sql_identifier Nom du schéma qui contient la table
table_name sql_identifier Nom de la table
privilege_type character_data Type de privilège : SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES ou TRIGGER
is_grantable yes_or_no YES si le droit peut être transmis, NO sinon
with_hierarchy yes_or_no Dans le standard SQL, WITH HIERARCHY OPTION est un (sous-)droit séparé autorisant certaines opérations sur la hiérarchie de l'héritage des tables. Dans PostgreSQL, ceci est inclus dans le droit SELECT, donc cette colonne affiche YES si le droit est SELECT, et NO sinon.

37.54. tables

La vue tables contient toutes les tables et vues définies dans la base de données courantes. Seules sont affichées les tables et vues auxquelles l'utilisateur courant a accès (parce qu'il en est le propriétaire ou qu'il possède certains privilèges).

Tableau 37.52. Colonnes de tables

Type de la colonne Description
table_catalog sql_identifier Nom de la base de données qui contient la table (toujours la base de données courante)
table_schema sql_identifier Nom du schéma qui contient la table
table_name sql_identifier

Type de la colonne Description
Nom de la table
`table_type` `character_data` Type de table : `BASE TABLE` pour une table de base persistante (le type de table normal), `VIEW` pour une vue, `FOREIGN` pour une table distante ou `LOCAL TEMPORARY` pour une table temporaire
`self_referencing_column_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`reference_generation` `character_data` S'applique à une fonctionnalité non disponible dans PostgreSQL.
`user_defined_type_catalog` `sql_identifier` Si la table est une table typée, le nom de la base de données qui contient le type de données sous-jacent (toujours la base de données actuel), sinon NULL.
`user_defined_type_schema` `sql_identifier` Si la table est une table typée, le nom du schéma qui contient le type de données sous-jacent, sinon NULL.
`user_defined_type_name` `sql_identifier` Si la table est une table typée, le nom du type de données sous-jacent, sinon NULL.
`is_insertable_into` `yes_or_no` `YES` s'il est possible d'insérer des données dans la table, `NO` dans le cas contraire. (Il est toujours possible d'insérer des données dans une table de base, pas forcément dans les vues.)
`is_typed` `yes_or_no` `YES` si la table est une table typée, `NO` dans le cas contraire
`commit_action` `character_data` Pas encore implémenté

37.55. `transforms`

La vue `transforms` contient des informations sur les transformations définies dans la base de données courante. Plus précisément, il contient une ligne pour chaque fonction contenue dans une transformation (la fonction « from SQL » ou « to SQL »).

Tableau 37.53. Colonnes de `transforms`

Type de la colonne Description
`udt_catalog` `sql_identifier` Nom de la base de données contenant le type ciblé par la transformation (toujours la base de données courante)
`udt_schema` `sql_identifier` Nom du schéma contenant le type ciblé par la transformation
`udt_name` `sql_identifier` Nom du type ciblé par la transformation
`specific_catalog` `sql_identifier` Nom de la base de données contenant la fonction (toujours la base de données courante)
`specific_schema` `sql_identifier` Nom du schéma contenant la fonction
`specific_name` `sql_identifier` Le « nom spécifique » de la fonction. Voir Section 37.45 pour plus d'informations.

Type de la colonne Description
`group_name` `sql_identifier` Le standard SQL autorise la définition de transformations en « groupes », et la sélection d'un groupe à l'exécution. PostgreSQL ne supporte pas cela. À la place, les transformations sont spécifiques à un langage. Comme compromis, ce champ contient le langage concernant cette transformation.
`transform_type` `character_data` `FROM SQL` ou `TO SQL`

37.56. `triggered_update_columns`

Pour les triggers de la base de données actuelle qui spécifient une liste de colonnes (comme `UPDATE OF colonne1, colonne2`), la vue `triggered_update_columns` identifie ces colonnes. Les triggers qui ne spécifient pas une liste de colonnes ne sont pas inclus dans cette vue. Seules sont affichées les colonnes que l'utilisateur actuel possède ou poru lesquelles l'utilisateur a des droits autre que `SELECT`.

Tableau 37.54. Colonnes de `triggered_update_columns`

Type de la colonne Description
`trigger_catalog` `sql_identifier` Nom de la base de données qui contient le trigger (toujours la base de données courante)
`trigger_schema` `sql_identifier` Nom du schéma qui contient le trigger
`trigger_name` `sql_identifier` Nom du trigger
`event_object_catalog` `sql_identifier` Nom de la base de données qui contient la table sur laquelle est défini le trigger (toujours la base de données courante)
`event_object_schema` `sql_identifier` Nom du schéma qui contient la table sur laquelle est défini le trigger
`event_object_table` `sql_identifier` Nom de la table sur laquelle est défini le trigger
`event_object_column` `sql_identifier` Nom de la colonne sur laquelle est défini le trigger

37.57. `triggers`

La vue `triggers` contient tous les triggers définis dans la base de données actuelles sur les tables et vues que l'utilisateur actuel possède ou sur lesquels il a d'autres droits que le `SELECT`.

Tableau 37.55. Colonnes de `triggers`

Type de la colonne Description
`trigger_catalog` `sql_identifier` Nom de la base contenant le trigger (toujours la base de données actuelle)
`trigger_schema` `sql_identifier` Nom du schéma contenant le trigger

Type de la colonne **Description**
`trigger_name` `sql_identifier` Nom du trigger
`event_manipulation` `character_data` Événement qui déclenche le trigger (`INSERT`, `UPDATE` ou `DELETE`)
`event_object_catalog` `sql_identifier` Nom de la base contenant la table où le trigger est défini (toujours la base de données actuelle)
`event_object_schema` `sql_identifier` Nom du schéma qui contient la table où le trigger est défini
`event_object_table` `sql_identifier` Nom de la table où le trigger est défini
`action_order` `cardinal_number` Déclencher parmi les triggers sur la même table qui ont les mêmes `event_manipulation`, `action_timing` et `action_orientation`. Dans PostgreSQL, les triggers sont déclenchés dans l'ordre des noms, et cette colonne reflète cela.
`action_condition` `character_data` La condition `WHEN` du trigger, NULL si aucun (NULL aussi si la table n'appartient pas à un rôle actuellement activé)
`action_statement` `character_data` Instruction exécutée par le trigger (actuellement toujours `EXECUTE FUNCTION` *`fonction`*`(...)`)
`action_orientation` `character_data` Indique si le trigger est exécuté une fois par ligne traitée ou une fois par instruction (`ROW` ou `STATEMENT`)
`action_timing` `character_data` Moment où le trigger se déclenche (`BEFORE`, `AFTER` ou `INSTEAD OF`)
`action_reference_old_table` `sql_identifier` Nom de la table de transition « old », ou NULL sinon
`action_reference_new_table` `sql_identifier` Nom de la table de transition « new », ou NULL sinon
`action_reference_old_row` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`action_reference_new_row` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`created` `time_stamp` S'applique à une fonctionnalité non disponible dans PostgreSQL

Les triggers dans PostgreSQL ont deux incompatibilités avec le standard SQL qui affectent leur représentation dans le schéma d'information.

Premièrement, les noms des triggers sont locaux à chaque table sous PostgreSQL, et ne sont pas des objets du schéma indépendants. De ce fait, il peut exister des triggers de même noms au sein d'un schéma, pour peu qu'ils s'occupent de tables différentes. (`trigger_catalog` et `trigger_schema` sont les champs qui décrivent effectivement la table sur laquelle est défini le trigger.)

Deuxièmement, les triggers peuvent être définis pour s'exécuter sur plusieurs événements sous PostgreSQL (c'est-à-dire `ON INSERT OR UPDATE`) alors que le standard SQL n'en autorise qu'un. Si un trigger est défini pour s'exécuter sur plusieurs événements, il est représenté sur plusieurs lignes dans le schéma d'information, une pour chaque type d'événement.

En conséquence, la clé primaire de la vue `triggers` est en fait `(trigger_catalog, trigger_schema, event_object_table, trigger_name, event_manipulation)` et non `(trigger_catalog, trigger_schema, trigger_name)` comme le spécifie le standard SQL. Néanmoins, si les triggers sont définis de manière conforme au standard SQL (des noms de triggers uniques dans le schéma et un seul type d'événement par trigger), il n'y a pas lieu de se préoccuper de ces deux incompatibilités.

> **Note**
>
> Avant PostgreSQL 9.1, les colonnes `action_timing`, `action_reference_old_table`, `action_reference_new_table`, `action_reference_old_row` et `action_reference_new_row` de cette vue étaient nommées respectivement `condition_timing`, `condition_reference_old_table`, `condition_reference_new_table`, `condition_reference_old_row` et `condition_reference_new_row`. Cela reflétaient leur nommage dans le standard SQL:1999. Le nouveau nommage est conforme à SQL:2003 et les versions ultérieures.

37.58. `udt_privileges`

La vue view `udt_privileges` identifie les droits `USAGE` donnés pour des types définis par l'utilisateur pour lesquels celui qui donne le droit et celui qui le reçoit sont des rôles actuellement activés. Il existe une ligne par chaque combinaison de colonne, rôle récupérant le droit, rôle donnant le droit. Cette vue affiche seulement les types composites (pour comprendre pourquoi, voir Section 37.60 ; voir Section 37.59 pour les droits sur les domaines.

Tableau 37.56. Colonnes de `udt_privileges`

Type de la colonne **Description**
`grantor` `sql_identifier` Nom du rôle donnant le droit
`grantee` `sql_identifier` Nom du rôle recevant le droit
`udt_catalog` `sql_identifier` Nom de la base contenant le type (actuellement toujours la base de données courante)
`udt_schema` `sql_identifier` Nom du schéma contenant le type
`udt_name` `sql_identifier` Nom du type
`privilege_type` `character_data` Toujours `TYPE USAGE`
`is_grantable` `yes_or_no` `YES` s'il est possible de donner le droit, `NO` sinon

37.59. `usage_privileges`

La vue `usage_privileges` identifie les privilèges d'`USAGE` accordés sur différents objets à un rôle actif ou par un rôle actif. Sous PostgreSQL, cela s'applique aux domaines. Puisqu'il n'y a pas de réels privilèges sur les domaines sous PostgreSQL, cette vue est affiche les privilèges `USAGE` implicitement octroyés à `PUBLIC` pour tous les collationnements, domaines, wrappers de données distantes, serveurs distants et séquences. Il y a une ligne pour chaque combinaison d'objet, de donneur et de receveur.

Comme les collationnements n'ont pas de vrais droits dans PostgreSQL, cette vue affiche des droits USAGE implicites, non donnables à d'autres, et donnés par le propriétaire à PUBLIC pour tous les collationnements. Les autres types d'objets affichent néanmoins de vrais droits.

Dans PostgreSQL, les séquences supportent aussi les droits SELECT et UPDATE en plus du droit USAGE. Ils ne sont pas dans le standard et du coup ils ne sont pas visibles dans le schéma d'information.

Tableau 37.57. Colonnes de `usage_privileges`

Type de la colonne Description
`grantor` `sql_identifier` Nom du rôle qui a donné ce droit
`grantee` `sql_identifier` Nom du rôle auquel ce droit a été donné
`object_catalog` `sql_identifier` Nom de la base de données qui contient l'objet (toujours la base de données courante)
`object_schema` `sql_identifier` Nom du schéma qui contient l'objet, si applicable, sinon une chaîne vide
`object_name` `sql_identifier` Nom de l'objet
`object_type` `character_data` COLLATION, DOMAIN, FOREIGN DATA WRAPPER FOREIGN SERVER ou SEQUENCE
`privilege_type` `character_data` Toujours USAGE
`is_grantable` `yes_or_no` YES si le droit peut être donné, NO dans le cas contraire

37.60. `user_defined_types`

La vue `user_defined_types` contient actuellement tous les types composites définis dans la base de données courante. Seuls sont montrés les types auxquels l'utilisateur courant a accès (parce qu'il en est le propriétaire soit parce qu'il dispose de certains droits).

SQL connaît deux genres de types définis par les utilisateurs : les types structurés (aussi connu sous le nom de types composites dans PostgreSQL) et les types distincts (non implémentés dans PostgreSQL). Pour être prêt, utilisez la colonne `user_defined_type_category` pour les différencier. Les autres types définis par l'utilisateur comme les types de base et les énumérations, qui sont des extensions PostgreSQL, ne sont pas affichés ici. Pour les domaines, voir Section 37.23.

Tableau 37.58. Colonnes de `user_defined_types`

Type de la colonne Description
`user_defined_type_catalog` `sql_identifier` Nom de la base de données qui contient ce type (toujours la base de données courante)
`user_defined_type_schema` `sql_identifier` Nom du schéma contenant ce type
`user_defined_type_name` `sql_identifier` Nom du type
`user_defined_type_category` `character_data` Actuellement, toujours STRUCTURED
`is_instantiable` `yes_or_no`

Type de la colonne **Description**
S'applique à une fonctionnalité non disponible dans PostgreSQL
`is_final` `yes_or_no` S'applique à une fonctionnalité non disponible dans PostgreSQL
`ordering_form` `character_data` S'applique à une fonctionnalité non disponible dans PostgreSQL
`ordering_category` `character_data` S'applique à une fonctionnalité non disponible dans PostgreSQL
`ordering_routine_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`ordering_routine_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`ordering_routine_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`reference_type` `character_data` S'applique à une fonctionnalité non disponible dans PostgreSQL
`data_type` `character_data` S'applique à une fonctionnalité non disponible dans PostgreSQL
`character_maximum_length` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`character_octet_length` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`character_set_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`character_set_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`character_set_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`collation_catalog` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`collation_schema` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`collation_name` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL
`numeric_precision` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`numeric_precision_radix` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`numeric_scale` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`datetime_precision` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`interval_type` `character_data` S'applique à une fonctionnalité non disponible dans PostgreSQL
`interval_precision` `cardinal_number` S'applique à une fonctionnalité non disponible dans PostgreSQL
`source_dtd_identifier` `sql_identifier`

Type de la colonne Description
S'applique à une fonctionnalité non disponible dans PostgreSQL
`ref_dtd_identifier` `sql_identifier` S'applique à une fonctionnalité non disponible dans PostgreSQL

37.61. `user_mapping_options`

La vue `user_mapping_options` contient toutes les options définies pour les correspondances d'utilisateur définies dans la base de données en cours. Seules sont affichées les correspondances pour lesquelles le serveur distant correspondant peut être accédé par l'utilisateur connecté (qu'il en soit le propriétaire ou qu'il ait quelques droits dessus).

Tableau 37.59. Colonnes de `user_mapping_options`

Type de la colonne Description
`authorization_identifier` `sql_identifier` Nom de l'utilisateur, ou `PUBLIC` si la correspondance est publique
`foreign_server_catalog` `sql_identifier` Nom de la base de données dans laquelle est défini le serveur distant correspondant (toujours la base de données en cours)
`foreign_server_name` `sql_identifier` Nom du serveur distant utilisé par cette correspondance
`option_name` `sql_identifier` Nom d'une option
`option_value` `character_data` Valeur de l'option. Cette colonne s'affichera comme NULL sauf si l'utilisateur connecté est l'utilisateur en cours de correspondance ou si la correspondance est pour `PUBLIC` et que l'utilisateur connecté est le propriétaire de la base de données ou un superutilisateur. Le but est de protéger les informations de mot de passe stockées comme option.

37.62. `user_mappings`

La vue `user_mappings` contient toutes les correspondances utilisateurs définies dans la base de données en cours. Seules sont affichées les correspondances pour lesquelles le serveur distant correspondant peut être accédé par l'utilisateur connecté (qu'il en soit le propriétaire ou qu'il ait quelques droits dessus).

Tableau 37.60. Colonnes de `user_mappings`

Type de la colonne Description
`authorization_identifier` `sql_identifier` Nom de l'utilisateur en cours de correspondance ou `PUBLIC` si la correspondance est publique
`foreign_server_catalog` `sql_identifier` Nom de la base de données dans laquelle est défini le serveur distant correspondant (toujours la base de données en cours)
`foreign_server_name` `sql_identifier` Nom du serveur distant utilisé par cette correspondance

37.63. `view_column_usage`

La vue `view_column_usage` identifie toutes les colonnes utilisées dans l'expression de la requête d'une vue (l'instruction `SELECT` définissant la vue). Une colonne n'est incluse que si la table contenant la colonne appartient à un rôle actif.

> **Note**
>
> Les colonnes des tables système ne sont pas incluses. Cela sera probablement corrigé un jour.

Tableau 37.61. Colonnes de `view_column_usage`

Type de la colonne **Description**
`view_catalog` `sql_identifier` Nom de la base de données qui contient la vue (toujours la base de données courante)
`view_schema` `sql_identifier` Nom du schéma qui contient la vue
`view_name` `sql_identifier` Nom de la vue
`table_catalog` `sql_identifier` Nom de la base de données qui contient la table qui contient la colonne utilisée par la vue (toujours la base de données courante)
`table_schema` `sql_identifier` Nom du schéma qui contient la table qui contient la colonne utilisée par la vue
`table_name` `sql_identifier` Nom de la table qui contient la colonne utilisée par la vue
`column_name` `sql_identifier` Nom de la colonne utilisée par la vue

37.64. `view_routine_usage`

La vue `view_routine_usage` identifie toutes les routines (fonctions et procédures) utilisées dans la requête d'une vue (l'instruction `SELECT` qui définit la vue). Une routine n'est incluse que si la routine appartient à un rôle actif.

Tableau 37.62. Colonnes de `view_routine_usage`

Type de la colonne **Description**
`table_catalog` `sql_identifier` Nom de la base qui contient la vue (toujours la base en cours)
`table_schema` `sql_identifier` Nom du schéma qui contient la vue
`table_name` `sql_identifier` Nom de la vue
`specific_catalog` `sql_identifier` Nom de la base qui contient la fonction (toujours la base en cours)
`specific_schema` `sql_identifier` Nom du schéma qui contient la fonction

Type de la colonne Description
`specific_name` `sql_identifier` Le « nom spécifique » de la fonction. Voir Section 37.45 pour plus d'informations.

37.65. `view_table_usage`

La vue `view_table_usage` identifie toutes les tables utilisées dans l'expression de la requête d'une vue (l'instruction `SELECT` définissant la vue). Une table n'est incluse que son propriétaire est un rôle actif.

> **Note**
>
> Les tables système ne sont pas incluses. Cela sera probablement corrigé un jour.

Tableau 37.63. Colonnes de `view_table_usage`

Type de la colonne Description
`view_catalog` `sql_identifier` Nom de la base de données qui contient la vue (toujours la base de données courante)
`view_schema` `sql_identifier` Nom du schéma qui contient la vue
`view_name` `sql_identifier` Nom de la vue
`table_catalog` `sql_identifier` Nom de la base de données qui contient la table utilisée par la vue (toujours la base de données courante)
`table_schema` `sql_identifier` Nom du schéma qui contient la table utilisée par la vue
`table_name` `sql_identifier` Nom de la table utilisée par la vue

37.66. `views`

La vue `views` contient toutes les vues définies dans la base de données courantes. Seules sont affichées les vues auxquelles l'utilisateur a accès (parce qu'il en est le propriétaire ou qu'il dispose de privilèges).

Tableau 37.64. Colonnes de `views`

Type de la colonne Description
`table_catalog` `sql_identifier` Nom de la base de données qui contient la vue (toujours la base de données courante)
`table_schema` `sql_identifier` Nom du schéma qui contient la vue
`table_name` `sql_identifier` Nom de la vue
`view_definition` `character_data`

Type de la colonne **Description**
Expression de la requête définissant la vue (NULL si la vue n'appartient pas à un rôle actif)
`check_option` `character_data` `CASCADED` ou `LOCAL` si la vue est définie avec l'option `CHECK OPTION`, `NONE` dans le cas contraire
`is_updatable` `yes_or_no` `YES` si la vue est actualisable (autorise `UPDATE` et `DELETE`), `NO` dans le cas contraire
`is_insertable_into` `yes_or_no` `YES` s'il est possible d'insérer des données dans la vue (autorise `INSERT`), `NO` dans le cas contraire
`is_trigger_updatable` `yes_or_no` `YES` si la vue dispose d'un trigger `INSTEAD OF` pour l'opération `UPDATE`, `NO` dans le cas contraire
`is_trigger_deletable` `yes_or_no` `YES` si la vue dispose d'un trigger `INSTEAD OF` pour l'opération `DELETE`, `NO` dans le cas
`is_trigger_insertable_into` `yes_or_no` `YES` si la vue dispose d'un trigger `INSTEAD OF` pour l'opération `INSERT`, `NO` dans le cas

Partie V. Programmation serveur

Cette partie traite des possibilités d'extension des fonctionnalités du serveur par l'ajout de fonctions utilisateur, de types de données, de triggers, etc. Il est préférable de n'aborder ces sujets avancés qu'après avoir compris tout le reste de la documentation. Les derniers chapitres décrivent les langages de programmation serveur disponibles avec PostgreSQL ainsi que les problèmes généraux des langages de programmation côté serveur. Il est essentiel de lire au minimum les premières sections du Chapitre 38 (sur les fonctions) avant de se plonger dans les langages de programmation serveur.

Table des matières

38. Étendre SQL 1224
38.1. L'extensibilité 1224
38.2. Le système des types de PostgreSQL 1224
38.2.1. Les types de base 1224
38.2.2. Les types conteneurs 1224
38.2.3. Les domaines 1225
38.2.4. Pseudo-types 1225
38.2.5. Types et fonctions polymorphes 1225
38.3. Fonctions utilisateur 1228
38.4. Procédures utilisateur 1228
38.5. Fonctions en langage de requêtes (SQL) 1229
38.5.1. Arguments pour les fonctions SQL 1230
38.5.2. Fonctions SQL sur les types de base 1231
38.5.3. Fonctions SQL sur les types composites 1233
38.5.4. Fonctions SQL avec des paramètres en sortie 1235
38.5.5. SQL Procedures with Output Parameters 1236
38.5.6. Fonctions SQL avec un nombre variables d'arguments 1237
38.5.7. Fonctions SQL avec des valeurs par défaut pour les arguments 1238
38.5.8. Fonctions SQL comme sources de table 1239
38.5.9. Fonctions SQL renvoyant un ensemble 1239
38.5.10. Fonctions SQL renvoyant TABLE 1243
38.5.11. Fonctions SQL polymorphes 1244
38.5.12. Fonctions SQL et collationnement 1246
38.6. Surcharge des fonctions 1247
38.7. Catégories de volatilité des fonctions 1248
38.8. Fonctions en langage de procédures 1249
38.9. Fonctions internes 1250
38.10. Fonctions en langage C 1250
38.10.1. Chargement dynamique 1250
38.10.2. Types de base dans les fonctions en langage C 1251
38.10.3. Conventions d'appel de la version 1 1254
38.10.4. Écriture du code 1258
38.10.5. Compiler et lier des fonctions chargées dynamiquement 1258
38.10.6. Arguments de type composite 1260
38.10.7. Renvoi de lignes (types composites) 1261
38.10.8. Renvoi d'ensembles 1263
38.10.9. Arguments polymorphes et types renvoyés 1269
38.10.10. Mémoire partagée et LWLocks 1271
38.10.11. Coder des extensions en C++ 1271
38.11. Informations d'optimisation de fonction 1272
38.12. Agrégats utilisateur 1273
38.12.1. Mode d'agrégat en déplacement 1275
38.12.2. Agrégats polymorphiques et variadiques 1276
38.12.3. Agrégats d'ensemble trié 1278
38.12.4. Agrégation partielle 1279
38.12.5. Fonctions de support pour les agrégats 1280
38.13. Types utilisateur 1281
38.13.1. Considérations sur les TOAST 1284
38.14. Opérateurs définis par l'utilisateur 1285
38.15. Informations sur l'optimisation d'un opérateur 1286
38.15.1. COMMUTATOR 1286
38.15.2. NEGATOR 1287
38.15.3. RESTRICT 1287
38.15.4. JOIN 1288
38.15.5. HASHES 1288

38.15.6. MERGES 1290
38.16. Interfacer des extensions d'index 1290
38.16.1. Méthodes d'indexation et classes d'opérateurs 1290
38.16.2. Stratégies des méthode d'indexation 1291
38.16.3. Routines d'appui des méthodes d'indexation 1293
38.16.4. Exemple 1296
38.16.5. Classes et familles d'opérateur 1298
38.16.6. Dépendances du système pour les classes d'opérateurs 1301
38.16.7. Opérateurs de tri 1303
38.16.8. Caractéristiques spéciales des classes d'opérateurs 1303
38.17. Empaqueter des objets dans une extension 1304
38.17.1. Fichiers des extensions 1305
38.17.2. Possibilités concernant le déplacement des extensions 1308
38.17.3. Tables de configuration des extensions 1309
38.17.4. Mise à jour d'extension 1310
38.17.5. Installer des extensions en utilisant des scripts de mise à jour 1311
38.17.6. Considérations de sécurité pour les extensions 1311
38.17.7. Exemples d'extensions 1312
38.18. Outils de construction d'extension 1314
39. Triggers (triggers) 1318
39.1. Aperçu du comportement des triggers 1318
39.2. Visibilité des modifications des données 1321
39.3. Écrire des fonctions triggers en C 1322
39.4. Un exemple complet de trigger 1325
40. Triggers sur événement 1329
40.1. Aperçu du fonctionnement des triggers sur événement 1329
40.2. Matrice de déclenchement des triggers sur événement 1330
40.3. Écrire des fonctions trigger sur événement en C 1334
40.4. Un exemple complet de trigger sur événement 1335
40.5. Un exemple de trigger sur événement de table modifiée 1336
41. Système de règles 1338
41.1. Arbre de requêtes 1338
41.2. Vues et système de règles 1340
41.2.1. Fonctionnement des règles select 1340
41.2.2. Règles de vue dans des instructions autres que select 1345
41.2.3. Puissance des vues dans PostgreSQL 1346
41.2.4. Mise à jour d'une vue 1346
41.3. Vues matérialisées 1347
41.4. Règles sur insert, update et delete 1350
41.4.1. Fonctionnement des règles de mise à jour 1351
41.4.2. Coopération avec les vues 1355
41.5. Règles et droits 1362
41.6. Règles et statut de commande 1364
41.7. Règles contre triggers 1365
42. Langages de procédures 1368
42.1. Installation des langages de procédures 1368
43. PL/pgSQL -- Langage de procédures SQL 1371
43.1. Aperçu 1371
43.1.1. Avantages de l'utilisation de PL/pgSQL 1371
43.1.2. Arguments supportés et types de données résultats 1372
43.2. Structure de PL/pgSQL 1372
43.3. Déclarations 1374
43.3.1. Déclarer des paramètres de fonctions 1375
43.3.2. ALIAS 1378
43.3.3. Copie de types 1378
43.3.4. Types ligne 1379
43.3.5. Types record 1379
43.3.6. Collationnement des variables PL/pgSQL 1380

43.4. Expressions 1381
43.5. Instructions de base 1382
43.5.1. Affectation 1382
43.5.2. Exécuter des commandes SQL 1382
43.5.3. Exécuter une commande avec une seule ligne de résultats 1384
43.5.4. Exécuter des commandes dynamiques 1385
43.5.5. Obtention du statut du résultat 1389
43.5.6. Ne rien faire du tout 1390
43.6. Structures de contrôle 1391
43.6.1. Retour d'une fonction 1391
43.6.2. Retour d'une procédure 1393
43.6.3. Appeler une procédure 1393
43.6.4. Contrôles conditionnels 1394
43.6.5. Boucles simples 1397
43.6.6. Boucler dans les résultats de requêtes 1400
43.6.7. Boucler dans des tableaux 1401
43.6.8. Récupérer les erreurs 1402
43.6.9. Obtenir des informations sur l'emplacement d'exécution 1405
43.7. Curseurs 1406
43.7.1. Déclaration de variables curseur 1406
43.7.2. Ouverture de curseurs 1407
43.7.3. Utilisation des curseurs 1409
43.7.4. Boucler dans les résultats d'un curseur 1412
43.8. Gestion des transactions 1412
43.9. Erreurs et messages 1414
43.9.1. Rapporter des erreurs et messages 1414
43.9.2. Vérification d'assertions 1416
43.10. Fonctions trigger 1416
43.10.1. Triggers sur les modifications de données 1416
43.10.2. Triggers sur des événements 1424
43.11. Les dessous de PL/pgSQL 1425
43.11.1. Substitution de variables 1425
43.11.2. Mise en cache du plan 1427
43.12. Astuces pour développer en PL/pgSQL 1429
43.12.1. Utilisation des guillemets simples (quotes) 1429
43.12.2. Vérifications supplémentaires à la compilation et à l'exécution 1431
43.13. Portage d'Oracle PL/SQL 1433
43.13.1. Exemples de portages 1434
43.13.2. Autres choses à surveiller 1439
43.13.3. Annexe 1440
44. PL/Tcl -- Langage de procédures Tcl 1443
44.1. Aperçu 1443
44.2. Fonctions et arguments PL/Tcl 1443
44.3. Valeurs des données avec PL/Tcl 1445
44.4. Données globales avec PL/Tcl 1446
44.5. Accès à la base de données depuis PL/Tcl 1446
44.6. Fonctions triggers en PL/Tcl 1449
44.7. Fonctions trigger sur événement en PL/Tcl 1451
44.8. Gestion des erreurs avec PL/Tcl 1451
44.9. Sous-transactions explicites dans PL/Tcl 1452
44.10. Gestion des transactions 1453
44.11. Configuration PL/Tcl 1454
44.12. Noms de procédure Tcl 1454
45. PL/Perl -- Langage de procédures Perl 1455
45.1. Fonctions et arguments PL/Perl 1455
45.2. Valeurs en PL/Perl 1460
45.3. Fonctions incluses 1460
45.3.1. Accès à la base de données depuis PL/Perl 1460

45.3.2. Fonctions utiles en PL/Perl 1464
45.4. Valeurs globales dans PL/Perl 1465
45.5. Niveaux de confiance de PL/Perl 1466
45.6. Triggers PL/Perl 1468
45.7. Triggers sur événements avec PL/Perl 1469
45.8. PL/Perl sous le capot 1470
45.8.1. Configuration 1470
45.8.2. Limitations et fonctionnalités absentes 1471
46. PL/Python -- Langage de procédures Python 1472
46.1. Fonctions PL/Python 1472
46.2. Valeur des données avec PL/Python 1473
46.2.1. Type de données 1474
46.2.2. Null, None 1474
46.2.3. Tableaux, Listes 1475
46.2.4. Types composites 1476
46.2.5. Fonctions renvoyant des ensembles 1478
46.3. Partage de données 1479
46.4. Blocs de code anonymes 1479
46.5. Fonctions de triggers 1479
46.6. Accès à la base de données 1480
46.6.1. Fonctions d'accès à la base de données 1480
46.6.2. Récupérer les erreurs 1483
46.7. Sous-transactions explicites 1484
46.7.1. Gestionnaires de contexte de sous-transaction 1484
46.8. Gestion des transactions 1485
46.9. Fonctions outils 1486
46.10. Python 2 versus Python 3 1487
46.11. Variables d'environnement 1487
47. Interface de programmation serveur 1489
47.1. Fonctions d'interface 1489
47.2. Fonctions de support d'interface 1531
47.3. Gestion de la mémoire 1540
47.4. Gestion des transactions 1550
47.5. Visibilité des modifications de données 1553
47.6. Exemples 1553
48. Processus en tâche de fond (background worker) 1557
49. Décodage logique (Logical Decoding) 1561
49.1. Exemples de décodage logique 1561
49.2. Concepts de décodage logique 1565
49.2.1. Décodage logique 1565
49.2.2. Slots de réplication 1565
49.2.3. Plugins de sortie 1566
49.2.4. Instantanés exportés 1566
49.3. Interface du protocole de réplication par flux 1566
49.4. Interface SQL de décodage logique 1566
49.5. Catalogues systèmes liés au décodage logique 1567
49.6. Plugins de sortie de décodage logique 1567
49.6.1. Fonction d'initialisation 1567
49.6.2. Capacités 1568
49.6.3. Modes de sortie 1568
49.6.4. Callbacks de plugin de sortie 1568
49.6.5. Fonction pour produire une sortie 1574
49.7. Écrivains de sortie de décodage logique 1575
49.8. Support de la réplication synchrone pour le décodage logique 1575
49.8.1. Overview 1575
49.8.2. Mises en garde 1575
49.9. Envoi en flux de grosses transactions pour le décodage logique 1576
49.10. Support du Two-phase commit pour le décodage logique 1577

50. Tracer la progression de la réplication 1579
51. Modules d'archivage 1580
51.1. Fonctions d'initialisation 1580
51.2. Fonctions de support d'un module d'archivage 1580
51.2.1. Fonction de vérification 1580
51.2.2. Fonction d'archivage 1581
51.2.3. Fonction d'arrêt 1581

Chapitre 38. Étendre SQL

Les sections qui suivent présentent les possibilités d'étendre le langage SQL de requêtage de PostgreSQL par l'ajout :

- de fonctions (Section 38.3) ;
- d'agrégats (Section 38.12) ;
- de types de données (Section 38.13) ;
- d'opérateurs (Section 38.14) ;
- de classes d'opérateurs pour les index (Section 38.16).
- d'extensions permettant de créer un paquetage d'objets qui disposent d'un point commun (voir Section 38.17)

38.1. L'extensibilité

PostgreSQL est extensible parce qu'il opère grâce à un système de catalogues. Quiconque est familier des systèmes de bases de données relationnelles standard sait que les informations concernant les bases, les tables, les colonnes, etc. y sont stockées dans ce qu'on nomme communément des catalogues systèmes (certains systèmes appellent cela le dictionnaire de données). Pour l'utilisateur, les catalogues ressemblent à des tables ordinaires, mais le SGBD y enregistre ses registres internes. À la différence des autres systèmes, PostgreSQL enregistre beaucoup d'informations dans ses catalogues : non seulement l'information concernant les tables et les colonnes, mais aussi l'information concernant les types de données, les fonctions, les méthodes d'accès, etc.

Ces tables peuvent être modifiées par l'utilisateur. Qui plus est, puisque PostgreSQL fonde ses opérations sur ces tables, il peut être étendu par les utilisateurs. En comparaison, les systèmes de bases de données conventionnels ne peuvent être étendus qu'en modifiant les procédures dans le code source ou en installant des modules spécifiquement écrits par le vendeur du SGBD.

De plus, le serveur PostgreSQL peut incorporer du code utilisateur par chargement dynamique. C'est-à-dire que l'utilisateur peut indiquer un fichier de code objet (par exemple une bibliothèque partagée) qui code un nouveau type ou une nouvelle fonction et PostgreSQL le charge au besoin. Il est encore plus facile d'ajouter au serveur du code écrit en SQL. La possibilité de modifier son fonctionnement « à la volée » fait de PostgreSQL un outil unique pour le prototypage rapide de nouvelles applications et de structures de stockage.

38.2. Le système des types de PostgreSQL

Les types de données de PostgreSQL sont répartis en types de base, types conteneurs, domaines et pseudo-types.

38.2.1. Les types de base

Les types de base sont ceux qui, comme `integer`, sont implantés sous le niveau du langage SQL (typiquement dans un langage de bas niveau comme le C). Ils correspondent généralement à ce que l'on appelle les types de données abstraits. PostgreSQL ne peut opérer sur de tels types qu'au moyen de fonctions utilisateur et n'en comprend le fonctionnement que dans la limite de la description qu'en a fait l'utilisateur. Les types de base internes sont décrits dans Chapitre 8.

Les types énumérés (enum) peuvent être considérés comme une sous-catégorie des types de base. La différence principale est qu'ils peuvent être créés en utilisant juste les commandes SQL, sans programmation de bas niveau. Référez-vous à Section 8.7 pour plus d'informations.

38.2.2. Les types conteneurs

PostgreSQL a trois genres de types « conteneurs », qui sont des types contenant plusieurs valeurs d'autres types. Ce sont des tableaux, des valeurs composites et des intervalles.

Les tableaux peuvent contenir plusieurs valeurs qui sont toutes du même type. Un type tableau est automatiquement créé pour chaque type de base, type composite, type intervalle et type domaine. Par contre, il n'y a pas de tableaux de tableaux. Pour ce qui concerne le système de typage, les tableaux multi-dimensionnels sont identiques aux tableaux uni-dimensionnels. Référez-vous à Section 8.15 pour plus d'informations.

Les types composites, ou types lignes, sont créés chaque fois qu'un utilisateur crée une table. Il est également possible de définir un type composite autonome sans table associée. Un type composite n'est qu'une simple liste de types de base avec des noms de champs associés. Une valeur de type composite est une ligne ou un enregistrement de valeurs de champ. La Section 8.16 fournit de plus amples informations sur ces types.

Un type intervalle (*range*) peut contenir deux valeurs de même type, qui sont les bornes inférieure et supérieure de l'intervalle. Les types intervalle sont créés par les utilisateurs, bien que quelques-uns soient intégrés. Référez-vous à Section 8.17 pour plus d'informations.

38.2.3. Les domaines

Un domaine est basé sur un type sous-jacent donné particulier, et est interchangeable avec ce type dans beaucoup d'utilisations. Cependant, un domaine peut avoir des contraintes restreignant ses valeurs à un sous-ensemble de ce que permet le type sous-jacent. Les domaines sont créés avec la fonction SQL CREATE DOMAIN. Référez-vous à Section 8.18 pour plus d'informations.

38.2.4. Pseudo-types

Il existe quelques « pseudo-types » pour des besoins particuliers. Les pseudo-types ne peuvent pas apparaître comme champs de table ou comme composants de types conteneurs, mais ils peuvent être utilisés pour déclarer les types des arguments et des résultats de fonctions. Dans le système de typage, ils fournissent un mécanisme d'identification des classes spéciales de fonctions. La Tableau 8.27 donne la liste des pseudo-types qui existent.

38.2.5. Types et fonctions polymorphes

Les pseudo-types ayant un intérêt particulier sont les *types polymorphiques*, utilisés pour déclarer des *fonctions polymorphiques*. Cette puissante fonctionnalité permet à une définition unique de fonction d'opérer sur différents types de données, le type de données étant déterminé par le type passé lors d'un appel donné. Les types polymorphiques sont indiqués dans Tableau 38.1. Quelques exemples de leurs usages apparaissent dans Section 38.5.11.

Tableau 38.1. Types polymorphiques

Nom	Famille	Description
`anyelement`	Simple	Indique qu'une fonction accepte n'importe quel type de données
`anyarray`	Simple	Indique qu'une fonction accepte n'importe quel type de données tableau.
`anynonarray`	Simple	Indique qu'une fonction accepte n'importe quel type de données qui n'est pas un tableau
`anyenum`	Simple	Indique que cette fonction accepte n'importe quel type de données enum (voir Section 8.7)
`anyrange`	Simple	Indique que cette fonction accepte n'importe quel type interval (voir Section 8.17)

Nom	Famille	Description
anymultirange	Simple	Indique qu'une fonction accepte tout type de données multirange (voir Section 8.17)
anycompatible	Common	Indique que la fonction accepte n'importe quel type de données, avec promotion automatique d'arguments multiples en un type de données commun.
anycompatiblearray	Common	Indique que la fonction accepte n'importe quel type de données tableau, avec promotion automatique d'arguments multiples en un type de données commun.
anycompatiblenonarray	Common	Indique que la fonction accepte n'importe quel type de données qui n'est pas un tableau, avec promotion automatique d'arguments multiples en un type de données commun.
anycompatiblerange	Common	Indique que la fonction accepte n'importe quel type de données interval, avec promotion automatique d'arguments multiples en un type de données commun.
anycompatiblemultirange	Common	Indique qu'une fonction accepte tout type de données multirange, avec une promotion automatique de plusieurs arguments vers un type de données commun

Les arguments et résultats polymorphes sont liés entre eux et sont résolus dans un type de données spécifique quand une requête faisant appel à une fonction polymorphe est analysée. Quand il y a plus d'un argument polymorphique, les types de données courants doivent correspondre comme décrit ci-dessous. Si le type de résultat de la fonction est polymorphique, or que ses arguments de sortie sont de types polymorphiques, alors les types de ces résultats sont déduits par les types polymorphiques en entrée présents comme décrit ci-dessous.

Pour les types polymorphiques de famille « simple », la correspondance et les règles de déduction fonctionnent ainsi :

Chaque occurrence (argument ou valeur de retour) déclarée comme `anyelement` peut prendre n'importe quel type réel de données mais, lors d'un appel de fonction donné, elles doivent toutes avoir le *même* type réel. Chaque occurrence déclarée comme `anyarray` peut prendre n'importe quel type de données tableau. De façon similaire, les occurrences déclarées en tant que `anyrange` doivent toutes être du même type. `anymultirange`.

De la même façon, si des occurrences sont déclarées comme `anyarray` et d'autres comme `anyelement` ou `anyarray`, le type réel de tableau des occurrences `anyarray` doit être un tableau dont les éléments sont du même type que ceux apparaissant dans les occurrences de type `anyelement`. `anynonarray` est traité de la même façon que `anyelement` mais ajoute une contrainte supplémentaire. Le type réel ne doit pas être un tableau. `anyenum` est traité de la même façon que `anyelement` mais ajoute une contrainte supplémentaire. Le type doit être un type enuméré.

De façon similaire, S'il existe des positions déclarées `anyrange` et d'autres déclarées `anyelement` ou `anyarray`, le type intervalle réel dans les positions `anyrange` doit être un intervalle dont le

sous-type est le même type apparaissant dans les positions `anyelement` et le même que le type élément des positions `anyarray`. S'il existe des positions déclarées `anymultirange`, leur type multirange réel doit contenir des intervalles correspondant aux paramètres déclarés `anyrange` et des éléments de base correspondant aux paramètres déclarés `anyelement` et `anyarray`.

Ainsi, quand plusieurs occurrences d'argument sont déclarées avec un type polymorphe, seules certaines combinaisons de types réels d'argument sont autorisées. Par exemple, une fonction déclarée comme `foo(anyelement, anyelement)` peut prendre comme arguments n'importe quelles valeurs à condition qu'elles soient du même type de données.

Quand la valeur renvoyée par une fonction est déclarée de type polymorphe, il doit exister au moins une occurrence d'argument également polymorphe, et le type réel de donnée passé comme argument détermine le(s) type(s) réel(s) du résultat renvoyé lors de cet appel à la fonction. Par exemple, s'il n'existe pas déjà un mécanisme d'indexation d'éléments de tableau, on peut définir une fonction qui code ce mécanisme : `indice(anyarray, integer) returns anyelement`. La déclaration de fonction contraint le premier argument réel à être de type tableau et permet à l'analyseur d'inférer le type correct de résultat à partir du type réel du premier argument. Une fonction déclarée de cette façon `f(anyarray) returns anyenum` n'accepte que des tableaux contenant des valeurs de type enum.

Dans la plupart des cas, l'analyseur peut inférer que le type de données réel pour un type résultat polymorphique pour des arguments qui sont d'un type polymorphique différent dans la même famille par exemple `anyarray` peut être déduit à partir de `anyelement` et vice versa. Une exception est qu'un résultat polymorphique de type `anyrange` nécessite un argument de type `anyrange` ; il ne peut pas être déduit d'arguments `anyarray` ou `anyelement`. Ceci est dû au fait qu'il pourrait y avoir plusieurs types d'intervalles avec le même sous-type.

`anynonarray` et `anyenum` ne représentent pas des variables de type séparé ; elles sont du même type que `anyelement`, mais avec une contrainte supplémentaire. Par exemple, déclarer une fonction `f(anyelement, anyenum)` est équivalent à la déclarer `f(anyenum, anyenum)` : les deux arguments réels doivent être du même type enum.

Pour la famille de types polymorphiques « common », la correspondance et les règles de déduction fonctionnent presque pareilles que pour la famille « simple », avec une différence majeure : les types courants des arguments n'ont pas besoin d'être identiques, aussi longtemps qu'ils peuvent être implicitement convertis en un type commun simple. Le type commun est sélectionné en suivant la même règle que pour `UNION` et les constructions liées (voir Section 10.5). La sélection du type commun considère les types présents `anycompatible` et `anycompatiblenonarray` en entrée, les éléments de type tableau `anycompatiblearray` en entrée, et les sous-types `anycompatiblerange` et `anycompatiblemultirange` en entrée. Si `anycompatiblenonarray` est présent, alors le type commun doit être un type qui n'est pas un tableau. Une fois le type commun identifié, les arguments en position `anycompatible` et `anycompatiblenonarray` sont automatiquement convertis en ce type, et les arguments en position `anycompatiblearray` sont automatiquement convertis en type tableau pour ce type.

Du fait qu'il n'y a pas de moyen de sélectionner le type interval à partir uniquement de son sous-type, l'usage de `anycompatiblerange` et/ou `anycompatiblemultirange` requiert que tous les arguments déclarés avec ce type aient le même type range et/ou multirange, et que le type du sous-type est en accord avec le type commun sélectionné, pour qu'il n'y ait pas de conversion nécessaire de la valeur de l'intervalle. Comme avec `anyrange` et `anymultirange`, l'usage de `anycompatiblerange` et `anymultirange` en tant que type de résultat de fonction demande qu'il y ait un argument `anycompatiblerange` ou `anycompatiblemultirange`.

Notez qu'il n'y a aucun type `anycompatibleenum`. Un tel type ne serait pas très utile, du fait qu'il n'y a normalement aucune conversion implicite de types enum, signifiant ainsi qu'il n'y aura aucune façon de résoudre un type commun pour des entrées enum différentes.

Les familles polymorphiques « simple » et « common » représentent deux ensembles de types de variable indépendants. Considérez par exemple :

```
CREATE FUNCTION myfunc(a anyelement, b anyelement,
                       c anycompatible, d anycompatible)
RETURNS anycompatible AS ...
```

Dans un appel de cette fonction, les deux premières entrées doivent avoir le même type exact. Les deux dernières entrées doivent être convertibles en un type commun, mais ce type nécessite d'être complétement différent du type des deux premières entrées. Le résultat aura le type commun des deux dernières entrées.

Une fonction variadic (c'est-à-dire une fonction acceptant un nombre variable d'arguments, comme dans Section 38.5.6) peut être polymorphique : cela se fait en déclarant son dernier paramètre `VARIADIC anyarray` ou `VARIADIC anycompatiblearray`. Pour s'assurer de la correspondance des arguments et déterminer le type de la valeur en retour, ce type de fonction se comporte de la même façon que si vous aviez écrit le nombre approprié de paramètres `anynonarray` ou `anycompatiblenonarray`.

38.3. Fonctions utilisateur

PostgreSQL propose quatre types de fonctions :

- fonctions en langage de requête (fonctions écrites en SQL, Section 38.5)
- fonctions en langage procédural (fonctions écrites, par exemple, en PL/pgSQL ou PL/Tcl, Section 38.8)
- fonctions internes (Section 38.9)
- fonctions en langage C (Section 38.10)

Chaque type de fonction peut accepter comme arguments (paramètres) des types de base, des types composites ou une combinaison de ceux-ci. De plus, chaque sorte de fonction peut renvoyer un type de base ou un type composite. Les fonctions pourraient aussi être définies pour renvoyer des ensembles de valeurs de base ou de valeurs composites.

De nombreuses sortes de fonctions peuvent accepter ou renvoyer certains pseudo-types (comme les types polymorphes) mais avec des fonctionnalités variées. Consultez la description de chaque type de fonction pour plus de détails.

Il est plus facile de définir des fonctions SQL aussi allons-nous commencer par celles-ci. La plupart des concepts présentés pour les fonctions SQL seront aussi gérés par les autres types de fonctions.

Lors de la lecture de ce chapitre, il peut être utile de consulter la page de référence de la commande `CREATE FUNCTION` pour mieux comprendre les exemples. Quelques exemples extraits de ce chapitre peuvent être trouvés dans les fichiers `funcs.sql` et `funcs.c` du répertoire du tutoriel de la distribution source de PostgreSQL.

38.4. Procédures utilisateur

Les différences clés sont :

- Les procédures sont définies avec la commande `CREATE PROCEDURE`, et non pas `CREATE FUNCTION`.
- Les procédures ne renvoient pas une valeur ; de ce fait, `CREATE PROCEDURE` n'a pas de clause `RETURNS`. Néanmoins, des procédures peuvent renvoyer à la place des données à leurs appelants via des paramètres en sortie.

- Alors qu'une fonction est appelée au sein d'une requête ou d'une commande DML, une procédure est appelée en isolation en utilisant la commande `CALL`.

- Une procédure peut valider ou annuler des transactions lors de son exécution (puis commencer automatiquement une nouvelle transaction), à partir du moment où la commande `CALL` qui l'a appelé ne fait pas partie d'un bloc de transaction explicite. Une fonction ne peut pas faire ça.

- Certains attributs de fonction, tels que la volatilité, ne s'appliquent pas aux procédures. Ces attributs contrôlent l'utilisation de la fonction dans une requête, et ne sont pas pertinents dans le cas de procédures.

Les explications présentes dans les sections suivantes concernant comment définir des fonctions utilisateurs s'appliquent également aux procédures, à la différence des points ci-dessus.

Collectivement, les fonctions et les procédures sont également appelées des *routines*. Il y a des commandes telles que `ALTER ROUTINE` et `DROP ROUTINE` qui peuvent s'appliquer sur des fonctions ou des procédures sans avoir besoin de savoir de quel type il s'agit. Veuillez noter, toutefois, qu'il n'y a pas de commande `CREATE ROUTINE`.

38.5. Fonctions en langage de requêtes (SQL)

Les fonctions SQL exécutent une liste arbitraire d'instructions SQL et renvoient le résultat de la dernière requête de cette liste. Dans le cas d'un résultat simple (pas d'ensemble), la première ligne du résultat de la dernière requête sera renvoyée (gardez à l'esprit que « la première ligne » d'un résultat multiligne n'est pas bien définie à moins d'utiliser `ORDER BY`). Si la dernière requête de la liste ne renvoie aucune ligne, la valeur NULL est renvoyée.

Une fonction SQL peut être déclarée de façon à renvoyer un ensemble (« set », ce qui signifie un ensemble de lignes) en spécifiant le type renvoyé par la fonction comme `SETOF` *`un_type`*, ou de façon équivalente en la déclarant comme `RETURNS TABLE(`*`colonnes`*`)`. Dans ce cas, toutes les lignes de la dernière requête sont renvoyées. Des détails supplémentaires sont donnés plus loin dans ce chapitre.

Le corps d'une fonction SQL doit être constitué d'une liste d'une ou de plusieurs instructions SQL séparées par des points-virgule. Un point-virgule après la dernière instruction est optionnel. Sauf si la fonction déclare renvoyer `void`, la dernière instruction doit être un `SELECT` ou un `INSERT`, `UPDATE` ou un `DELETE` qui a une clause `RETURNING`.

Toute collection de commandes dans le langage SQL peut être assemblée et définie comme une fonction. En plus des requêtes `SELECT`, les commandes peuvent inclure des requêtes de modification des données (`INSERT`, `UPDATE`, `DELETE` et `MERGE`) ainsi que d'autres commandes SQL (sans toutefois pouvoir utiliser les commandes de contrôle de transaction, telles que `COMMIT`, `SAVEPOINT`, et certaines commandes utilitaires, comme `VACUUM`, dans les fonctions SQL). Néanmoins, la commande finale doit être un `SELECT` ou doit avoir une clause `RETURNING` qui renvoie ce qui a été spécifié comme type de retour de la fonction. Autrement, si vous voulez définir une fonction SQL qui réalise des actions mais n'a pas de valeur utile à renvoyer, vous pouvez la définir comme renvoyant `void`. Par exemple, cette fonction supprime les lignes avec des salaires négatifs depuis la table `emp` :

```
CREATE FUNCTION nettoie_emp() RETURNS void AS '
    DELETE FROM emp WHERE salaire < 0;
' LANGUAGE SQL;

SELECT nettoie_emp();

  nettoie_emp
  -----------

  (1 row)
```

Vous pouvez aussi l'écrire comme une procédure, ce qui évite d'avoir à renseigner un paramètre de sortie. Par exemple:

```
CREATE PROCEDURE clean_emp() AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;

CALL clean_emp();
```

Dans les cas simples comme celui-ci, la différence entre une fonction retournant un `void` et une procédure est purement esthétique. Cependant, les procédures offrent des fonctionnalités supplémentaires comme le contrôle des transactions qui n'est pas disponible dans une fonction. De plus, les procédures sont conformes à la norme SQL alors que renvoyer `void` est une fonctionnalité spécifique à PostgreSQL.

> **Note**
>
> Le corps entier d'une fonction SQL est analysé avant d'être exécuté. Bien qu'une fonction SQL puisse contenir des commandes qui modifient les catalogues systèmes (par exemple `CREATE TABLE`), les effets de telles commandes ne seront pas visibles lors de l'analyse des commandes suivantes dans la fonction. De ce fait, par exemple, `CREATE TABLE foo (...); INSERT INTO foo VALUES(...);` ne fonctionnera pas si c'est intégré dans une seule fonction SQL car `foo` n'existera pas encore quand la commande SQL `INSERT` sera analysée. Il est recommandé d'utiliser PL/PgSQL à la place de SQL dans ce genre de situations.

La syntaxe de la commande `CREATE FUNCTION` requiert que le corps de la fonction soit écrit comme une constante de type chaîne. Il est habituellement plus agréable d'utiliser les guillemets dollar (voir la Section 4.1.2.4) pour cette constante. Si vous choisissez d'utiliser la syntaxe habituelle avec des guillemets simples, vous devez doubler les marques de guillemet simple (`'`) et les antislashs (\), en supposant que vous utilisiez la syntaxe d'échappement de chaînes, utilisés dans le corps de la fonction (voir la Section 4.1.2.1).

38.5.1. Arguments pour les fonctions SQL

Les arguments d'une fonction SQL peuvent être référencés dans le corps de la fonction en utilisant soit les noms soit les numéros. Des exemples de chaque méthode se trouvent ci-dessous.

Pour utiliser un nom, déclarez l'argument de la fonction comme ayant un nom, puis écrivez le nom dans le corps de la fonction. Si le nom de l'argument est le même que celui d'une colonne dans la commande SQL en cours, le nom de la colonne est prioritaire. Pour contourner ce comportement, qualifiez le nom de l'argument avec le nom de la fonction, autrement dit *`nom_fonction.nom_argument`*. (Si cela entre en conflit avec un nom de colonne qualifié, cette fois encore, la colonne l'emporte. Vous pouvez éviter toute ambiguïté en choisissant un alias différent pour la table à l'intérieur de la commande SQL.)

Dans l'ancienne approche numérique, les arguments sont référencés en utilisant la syntaxe `$`*`n`* : `$1` fait référence au premier argument, `$2` au second, et ainsi de suite. Ceci fonctionnera que l'argument ait été déclaré avec un nom ou pas.

Si un argument est de type composite, la notation à point, *`nom_argument.nom_champ`* ou `$1.`*`nom_champ`* peut être utilisé pour accéder aux attributs de l'argument. Encore une fois, vous pourriez avoir besoin de qualifier le nom de l'argument avec le nom de la fonction pour qu'il n'y ait pas d'ambiguité.

Les arguments de fonctions SQL peuvent seulement être utilisés comme valeurs de données, et non pas comme identifiants. Du coup, par exemple, ceci est accepté :

```
INSERT INTO mytable VALUES ($1);
```

mais ceci ne fonctionnera pas :

```
INSERT INTO $1 VALUES (42);
```

> **Note**
>
> La possibilité d'utiliser des noms pour référencer les arguments d'une fonction SQL a été ajoutée à PostgreSQL 9.2. Les fonctions devant être utilisées sur des versions antérieures doivent utiliser la notation $*n*.

38.5.2. Fonctions SQL sur les types de base

La fonction SQL la plus simple possible n'a pas d'argument et retourne un type de base tel que `integer` :

```
CREATE FUNCTION un() RETURNS integer AS $$
    SELECT 1 AS resultat;
$$ LANGUAGE SQL;

-- Autre syntaxe pour les chaînes littérales :
CREATE FUNCTION un() RETURNS integer AS '
    SELECT 1 AS resultat;
' LANGUAGE SQL;

SELECT un();

 un
----
  1
```

Notez que nous avons défini un alias de colonne avec le nom `resultat` dans le corps de la fonction pour se référer au résultat de la fonction mais cet alias n'est pas visible hors de la fonction. En effet, le résultat est nommé `un` au lieu de `resultat`.

Il est presque aussi facile de définir des fonctions SQL acceptant des types de base comme arguments :

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$
    SELECT x + y;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

Autrement, nous pourrions nous passer des noms pour les arguments et utiliser à la place des numéros :

```
CREATE FUNCTION ajoute(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT ajoute(1, 2) AS reponse;

 reponse
---------
       3
```

Voici une fonction plus utile, qui pourrait être utilisée pour débiter un compte bancaire :

```
CREATE FUNCTION tf1 (no_compte integer, debit numeric) RETURNS
 numeric AS $$
     UPDATE bank
        SET balance = balance - debit
        WHERE no_compte = tf1.no_compte;
    SELECT 1;
$$ LANGUAGE SQL;
```

Un utilisateur pourrait exécuter cette fonction pour débiter le compte 17 de 100 000 euros ainsi :

```
SELECT tf1(17, 100.000);
```

Dans cet exemple, nous choisissons le nom `no_compte` comme premier argument mais ce nom est identique au nom d'une colonne dans la table `banque`. Dans la commande `UPDATE`, `no_compte` fait référence à la colonne `banque.no_compte`, donc `tf1.no_compte` doit être utilisé pour faire référence à l'argument. Nous pouvons bien sûr éviter cela en utilisant un nom différent pour l'argument.

Dans la pratique, on préférera vraisemblablement un résultat plus utile que la constante 1. Une définition plus probable est :

```
CREATE FUNCTION tf1 (no_compte integer, debit numeric) RETURNS
 numeric AS $$
     UPDATE bank
        SET balance = balance - debit
        WHERE no_compte = tf1.no_compte;
    SELECT balance FROM banque WHERE no_compte = tf1.no_compte;
$$ LANGUAGE SQL;
```

qui ajuste le solde et renvoie sa nouvelle valeur. La même chose peut se faire en une commande en utilisant la clause `RETURNING` :

```
CREATE FUNCTION tf1 (no_compte integer, debit numeric) RETURNS
 numeric AS $$
     UPDATE bank
        SET balance = balance - debit
        WHERE no_compte = tf1.no_compte
     RETURNING balance;
$$ LANGUAGE SQL;
```

Si le `SELECT` final ou la clause `RETURNING` dans une fonction SQL ne renvoie pas exactement le type de résultat déclaré pour la fonction, PostgreSQL convertira automatiquement la valeur dans le type requis, si c'est possible avec une conversion implicite ou d'affectation. Sinon vous devrez écrire une

conversion explicite. Par exemple, supposons que nous voulions que la précédente fonction `add_em` retourne un type `float8` à la place. Il suffit d'écrire :

```
CREATE FUNCTION add_em(integer, integer) RETURNS float8 AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;
```

car la somme de type `integer` peut être convertie implicitement en `float8` (voir Chapitre 10 or CREATE CAST pour plus d'informations sur les conversions).

38.5.3. Fonctions SQL sur les types composites

Quand nous écrivons une fonction avec des arguments de type composite, nous devons non seulement spécifier l'argument utilisé, mais aussi spécifier l'attribut désiré de cet argument (champ). Par exemple, supposons que `emp` soit le nom d'une table contenant des données sur les employés et donc également le nom du type composite correspondant à chaque ligne de la table. Voici une fonction `double_salaire` qui calcule ce que serait le salaire de quelqu'un s'il était doublé :

```
CREATE TABLE emp (
    nom         text,
    salaire     numeric,
    age         integer,
    cubicle     point
);

INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salaire(emp) RETURNS numeric AS $$
    SELECT $1.salaire * 2 AS salaire;
$$ LANGUAGE SQL;

SELECT nom, double_salaire(emp.*) AS reve
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';

 name | reve
------+-------
 Bill |  8400
```

Notez l'utilisation de la syntaxe `$1.salaire` pour sélectionner un champ dans la valeur de la ligne argument. Notez également comment la commande `SELECT` utilise *`nom_table`*`.*` pour sélectionner la ligne courante entière de la table comme une valeur composite (`emp`). La ligne de la table peut aussi être référencée en utilisant seulement le nom de la table ainsi :

```
SELECT nom, double_salaire(emp) AS reve
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

mais cette utilisation est obsolète car elle est facilement obscure. (Voir Section 8.16.5 pour des détails sur ces deux syntaxes pour la valeur composite d'une ligne de table.)

Quelque fois, il est pratique de construire une valeur d'argument composite en direct. Ceci peut se faire avec la construction `ROW`. Par exemple, nous pouvons ajuster les données passées à la fonction :

```
SELECT nom, double_salaire(ROW(nom, salaire*1.1, age, cubicle)) AS
 reve
```

```
  FROM emp;
```

Il est aussi possible de construire une fonction qui renvoie un type composite. Voici un exemple de fonction renvoyant une seule ligne de type `emp` :

```
CREATE FUNCTION nouvel_emp() RETURNS emp AS $$
    SELECT text 'Aucun' AS nom,
        1000.0 AS salaire,
        25 AS age,
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

Dans cet exemple, nous avons spécifié chacun des attributs avec une valeur constante, mais un quelconque calcul aurait pu être substitué à ces valeurs.

Notez deux aspects importants à propos de la définition de fonction :

- L'ordre de la liste du SELECT doit être exactement le même que celui dans lequel les colonnes apparaissent dans le type composite (donner des noms aux colonnes dans le corps de la fonction, comme nous l'avons fait dans l'exemple, n'a aucune interaction avec le système).

- Il faut s'assurer que le type de chaque expression peut être converti dans le type de la colonne correspondante du type composite. Sinon, une erreur telle que :

  ```
  +ERROR:  return type mismatch in function declared to return emp
  DETAIL:  Final statement returns text instead of point at column
   4.
  ```

  sera renvoyée. Comme c'est le cas pour les types de base, le système ne fera pas de conversion explicite automatiquement, mais seulement des conversions implicites ou d'affectation.

Une autre façon de définir la même fonction est :

```
CREATE FUNCTION nouveau_emp() RETURNS emp AS $$
    SELECT ROW('Aucun', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

Ici, nous écrivons un `SELECT` qui renvoie seulement une colonne du bon type composite. Ceci n'est pas vraiment meilleur dans cette situation mais c'est une alternative pratique dans certains cas -- par exemple, si nous avons besoin de calculer le résultat en appelant une autre fonction qui renvoie la valeur composite désirée. Un autre exemple est que si nous essayons d'écrire une fonction qui retourne un domaine sur un type composite, plutôt qu'un simple type composite, il est toujours nécessaire de l'écrire comme retournant une seule colonne, puisqu'il n'y a aucune autre manière de produire une valeur qui soit exactement du type de ce domaine.

Nous pouvons appeler cette fonction soit en l'utilisant dans une expression de valeur :

```
SELECT nouveau_emp();

        nouveau_emp
-------------------------
 (None,1000.0,25,"(2,2)")
```

soit en l'utilisant comme une fonction table :

```
SELECT * FROM nouveau_emp();

  nom  | salaire | age | cubicle
-------+---------+-----+---------
 Aucun |  1000.0 |  25 | (2,2)
```

La deuxième façon est décrite plus complètement dans la Section 38.5.8.

Quand vous utilisez une fonction qui renvoie un type composite, vous pourriez vouloir seulement un champ (attribut) depuis ce résultat. Vous pouvez le faire avec cette syntaxe :

```
SELECT (nouveau_emp()).nom;

 nom
------
 None
```

Les parenthèses supplémentaires sont nécessaires pour éviter une erreur de l'analyseur. Si vous essayez de le faire sans, vous obtiendrez quelque chose comme ceci :

```
SELECT nouveau_emp().nom;
ERROR:  syntax error at or near "."
LINE 1: SELECT nouveau_emp().nom;
                            ^
```

Une autre option est d'utiliser la notation fonctionnelle pour extraire un attribut :

```
SELECT nom(nouveau_emp());

 name
------
 None
```

Comme expliqué dans Section 8.16.5, la notation avec des champs et la notation fonctionnelle sont équivalentes.

38.5.4. Fonctions SQL avec des paramètres en sortie

Une autre façon de décrire les résultats d'une fonction est de la définir avec des *paramètres en sortie* comme dans cet exemple :

```
CREATE FUNCTION ajoute (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
LANGUAGE SQL;

SELECT ajoute(3,7);
 ajoute
--------
     10
(1 row)
```

Ceci n'est pas vraiment différent de la version d'`ajoute` montrée dans la Section 38.5.2. La vraie valeur des paramètres en sortie est qu'ils fournissent une façon agréable de définir des fonctions qui renvoient plusieurs colonnes. Par exemple,

```
CREATE FUNCTION ajoute_n_produit (x int, y int, OUT sum int, OUT
 product int)
AS 'SELECT x + y, x * y'
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

Ce qui est arrivé ici est que nous avons créé un type composite anonyme pour le résultat de la fonction. L'exemple ci-dessus a le même résultat final que

```
CREATE TYPE produit_ajoute AS (somme int, produit int);

CREATE FUNCTION ajoute_n_produit (int, int) RETURNS produit_ajoute
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

mais ne pas avoir à s'embêter avec la définition séparée du type composite est souvent agréable. Notez que les noms attachés aux paramètres de sortie ne sont pas juste décoratif, mais déterminent le nom des colonnes du type composite anonyme. (Si vous omettez un nom pour un paramètre en sortie, le système choisira un nom lui-même.)

Notez que les paramètres en sortie ne sont pas inclus dans la liste d'arguments lors de l'appel d'une fonction de ce type en SQL. Ceci parce que PostgreSQL considère seulement les paramètres en entrée pour définir la signature d'appel de la fonction. Cela signifie aussi que seuls les paramètres en entrée sont importants lors de références de la fonction pour des buts comme sa suppression. Nous pouvons supprimer la fonction ci-dessus avec l'un des deux appels ci-dessous :

```
DROP FUNCTION ajoute_n_produit (x int, y int, OUT somme int, OUT
 produit int);
DROP FUNCTION ajoute_n_produit (int, int);
```

Les paramètres peuvent être marqués comme `IN` (par défaut), `OUT` ou `INOUT` ou `VARIADIC`. Un paramètre `INOUT` sert à la fois de paramètre en entrée (il fait partie de la liste d'arguments en appel) et comme paramètre de sortie (il fait partie du type d'enregistrement résultat). Les paramètres `VARIADIC` sont des paramètres en entrées, mais sont traités spécifiquement comme indiqué ci-dessous.

38.5.5. SQL Procedures with Output Parameters

Les paramètres de sortie sont aussi utilisables dans les procédures mais leur fonctionnement est légèrement différent par rapport aux paramètres de sortie des fonctions. Dans les commandes `CALL`, les paramètres en sortie doivent être incluses dans la liste des arguments. Par exemple, la procédure précédente permettant de débiter un compte bancaire pourrait être écrite ainsi :

```
CREATE PROCEDURE tp1 (accountno integer, debit numeric, OUT
 new_balance numeric) AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tp1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```

Pour appeler cette procédure, an argument matching the `OUT` parameter must be included. Il est habituel d'écrire `NULL`.

```
CALL tp1(17, 100.0, NULL);
```

Si vous écrivez autre chose, cela doit être une expression qui est convertible implicitement vers le type déclaré du paramètre, comme pour les paramètres en entrée. Notez néanmoins qu'une telle expression ne sera pas évaluée.

Lors de l'appel d'une procédure à partir de PL/pgSQL, au lieu d'écrire `NULL`, vous devez écrire une variable qui recevra la sortie de la procédure. Voir Section 43.6.3 pour plus d'informations.

38.5.6. Fonctions SQL avec un nombre variables d'arguments

Les fonctions SQL peuvent accepter un nombre variable d'arguments à condition que tous les arguments « optionnels » sont du même type. Les arguments optionnels seront passés à la fonction sous forme d'un tableau. La fonction est déclarée en marquant le dernier paramètre comme `VARIADIC` ; ce paramètre doit être déclaré de type tableau. Par exemple :

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $
$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
 mleast
--------
     -1
(1 row)
```

En fait, tous les arguments à la position ou après la position de l'argument `VARIADIC` sont emballés dans un tableau à une dimension, comme si vous aviez écrit

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);    -- doesn't work
```

Vous ne pouvez pas vraiment écrire cela, ou tout du moins cela ne correspondra pas à la définition de la fonction. Un paramètre marqué `VARIADIC` correspond à une ou plusieurs occurrences de son type d'élément, et non pas de son propre type.

Quelque fois, il est utile de pouvoir passer un tableau déjà construit à une fonction variadic ; ceci est particulièrement intéressant quand une fonction variadic veut passer son paramètre tableau à une autre fonction. En outre, il s'agit de la seule méthode sûre pour appeler une fonction `VARIADIC` trouvée dans un schéma qui autorise les utilisateurs qui ne sont pas de confiance à créer des objets ; voir Section 10.3. Vous pouvez faire cela en spécifiant `VARIADIC` dans l'appel :

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

Ceci empêche l'expansion du paramètre variadic de la fonction dans le type des éléments, ce qui permet à la valeur tableau de correspondre. `VARIADIC` peut seulement être attaché au dernier argument d'un appel de fonction.

Spécifier `VARIADIC` dans l'appel est aussi la seule façon de passer un tableau vide à une fonction variadique. Par exemple :

```
SELECT mleast(VARIADIC ARRAY[]::numeric[]);
```

Écrire simplement `SELECT mleast()` ne fonctionne pas car un paramètre variadique doit correspondre à au moins un argument réel. (Vous pouvez définir une deuxième fonction aussi nommée `mleast`, sans paramètres, si vous voulez permettre ce type d'appels.)

Les paramètres de l'élément tableau générés à partir d'un paramètre variadic sont traités comme n'ayant pas de noms propres. Cela signifie qu'il n'est pas possible d'appeler une fonction variadic en utilisant des arguments nommés (Section 4.3), sauf quand vous spécifiez `VARIADIC`. Par exemple, ceci fonctionnera :

```
SELECT mleast(VARIADIC arr => ARRAY[10, -1, 5, 4.4]);
```

mais pas cela :

```
SELECT mleast(arr => 10);
SELECT mleast(arr => ARRAY[10, -1, 5, 4.4]);
```

38.5.7. Fonctions SQL avec des valeurs par défaut pour les arguments

Les fonctions peuvent être déclarées avec des valeurs par défaut pour certains des paramètres en entrée ou pour tous. Les valeurs par défaut sont insérées quand la fonction est appelée avec moins d'arguments qu'à priori nécessaires. Comme les arguments peuvent seulement être omis à partir de la fin de la liste des arguments, tous les paramètres après un paramètre disposant d'une valeur par défaut disposeront eux-aussi d'une valeur par défaut. (Bien que l'utilisation de la notation avec des arguments nommés pourrait autoriser une relâche de cette restriction, elle est toujours forcée pour que la notation des arguments de position fonctionne correctement.) Que vous l'utilisez ou non, cette possibilité implique la nécessité de prendre des précautions lors de l'appel de fonctions dans les bases de données où certains utilisateurs ne font pas confiance à d'autres utilisateurs ; voir Section 10.3.

Par exemple :

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;

SELECT foo(10, 20, 30);
 foo
-----
  60
(1 row)

SELECT foo(10, 20);
 foo
```

```
-----
  33
(1 row)

SELECT foo(10);
 foo
-----
  15
(1 row)

SELECT foo();  -- échec car il n'y a pas de valeur par défaut pour
 le premier argument
ERROR:  function foo() does not exist
```

Le signe = peut aussi être utilisé à la place du mot clé DEFAULT,

38.5.8. Fonctions SQL comme sources de table

Toutes les fonctions SQL peuvent être utilisées dans la clause FROM d'une requête mais ceci est particulièrement utile pour les fonctions renvoyant des types composites. Si la fonction est définie pour renvoyer un type de base, la fonction table produit une table d'une seule colonne. Si la fonction est définie pour renvoyer un type composite, la fonction table produit une colonne pour chaque attribut du type composite.

Voici un exemple :

```
CREATE TABLE foo (fooid int, foosousid int, foonom text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION recupfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(foonom) FROM recupfoo(1) AS t1;

 fooid | foosubid | foonom | upper
-------+----------+--------+-------
     1 |        1 | Joe    | JOE
(1 row)
```

Comme le montre cet exemple, nous pouvons travailler avec les colonnes du résultat de la fonction comme s'il s'agissait des colonnes d'une table normale.

Notez que nous n'obtenons qu'une ligne comme résultat de la fonction. Ceci parce que nous n'avons pas utilisé l'instruction SETOF. Cette instruction est décrite dans la prochaine section.

38.5.9. Fonctions SQL renvoyant un ensemble

Quand une fonction SQL est déclarée renvoyer un SETOF *un_type*, la requête finale de la fonction est complètement exécutée et chaque ligne extraite est renvoyée en tant qu'élément de l'ensemble résultat.

Cette caractéristique est normalement utilisée lors de l'appel d'une fonction dans une clause FROM. Dans ce cas, chaque ligne renvoyée par la fonction devient une ligne de la table vue par la requête. Par exemple, supposons que la table foo ait le même contenu que précédemment et écrivons :

```
CREATE FUNCTION recupfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM recupfoo(1) AS t1;
```

Alors nous obtenons :

```
 fooid | foosousid | foonom
-------+-----------+--------
     1 |         1 | Joe
     1 |         2 | Ed
(2 rows)
```

Il est aussi possible de renvoyer plusieurs lignes avec les colonnes définies par des paramètres en sortie, comme ceci :

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT
 product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

SELECT * FROM sum_n_product_with_tab(10);
 sum | product
-----+---------
  11 |      10
  13 |      30
  15 |      50
  17 |      70
(4 rows)
```

Le point clé ici est que vous devez écrire `RETURNS SETOF record` pour indiquer que la fonction renvoie plusieurs lignes et non pas une seule. S'il n'y a qu'un paramètre en sortie, indiquez le type de paramètre plutôt que `record`.

Il est souvent utile de construire le résultat d'une requête par l'appel répété d'une fonction retournant un ensemble, dont les paramètres varient à chaque appel avec les valeurs des lignes d'une table ou d'une sous-requête. La manière idéale de le réaliser est d'utiliser le mot clé `LATERAL`, qui est décrit dans Section 7.2.1.5. Voici un exemple de fonction retournant un ensemble permettant d'énumérer les éléments d'une structure en arbre :

```
SELECT * FROM noeuds;
   nom        | parent
--------------+--------
 Haut         |
 Enfant1      | Haut
 Enfant2      | Haut
 Enfant3      | Haut
 Sous-Enfant1 | Enfant1
 Sous-Enfant2 | Enfant1
```

```
(6 rows)

CREATE FUNCTION listeenfant(text) RETURNS SETOF text AS $$
    SELECT nom FROM noeuds WHERE parent = $1
$$ LANGUAGE SQL STABLE;

SELECT * FROM listeenfant('Haut');
 listeenfant
--------------
 Enfant1
 Enfant2
 Enfant3
(3 rows)

SELECT nom, enfant FROM noeuds, LATERAL listeenfant(nom) AS enfant;
  name   |    child
---------+-------------
 Haut    | Enfant1
 Haut    | Enfant2
 Haut    | Enfant3
 Enfant1 | Sous-Enfant1
 Enfant1 | Sous-Enfant2
(5 rows)
```

Cet exemple ne fait rien de plus que ce qui aurait été possible avec une simple jointure mais, dans des cas plus complexes, l'alternative consistant à reporter du travail dans une fonction peut se révéler assez pratique.

Les fonctions retournant des ensembles peuvent aussi être appelées dans la clause select d'une requête. Pour chaque ligne que cette requête génère elle-même, la fonction retournant un ensemble est appelée, et une ligne résultat est générée pour chaque élément de l'ensemble retourné par la fonction. L'exemple précédent peut aussi être implémenté avec des requêtes telles que :

```
SELECT listeenfant('Haut');
 listeenfant
--------------
 Enfant1
 Enfant2
 Enfant3
(3 rows)

SELECT nom, listeenfant(nom) FROM noeuds;
  nom    | listeenfant
---------+--------------
 Haut    | Enfant1
 Haut    | Enfant2
 Haut    | Enfant3
 Enfant1 | Sous-Enfant1
 Enfant1 | Sous-Enfant2
(5 rows)
```

Notez, dans le dernier `SELECT`, qu'aucune ligne n'est renvoyée pour `Enfant2`, `Enfant3`, etc. Ceci est dû au fait que la fonction `listeenfant` renvoie un ensemble vide pour ces arguments et ainsi aucune ligne n'est générée. Ce comportement est identique à celui attendu par une requête de jointure interne join avec le résultat de la fonction utilisant la syntaxe `LATERAL`.

Le comportement de PostgreSQL pour une fonction renvoyant des lignes (SETOF) dans la liste SELECT d'une requête est pratiquement identique à celui d'une fonction SETOF écrite dans une clause `LATERAL FROM`. Par exemple :

```
SELECT x, generate_series(1,5) AS g FROM tab;
```

est pratiquement équivalente à :

```
SELECT x, g FROM tab, LATERAL generate_series(1,5) AS g;
```

Ce serait exactement la même chose, sauf que dans cet exemple spécifique, le planificateur pourrait choisir de placer `g` à l'extérieur de la jointure de boucle imbriquée puisque `g` n'a pas de réelle dépendance latérale sur `tab`. Cela résulterait en un ordre différent des lignes en sortie. Les fonctions SETOF dans la liste SELECT sont toujours évaluées comme si elles étaient à l'intérieur d'une jointure de boucle imbriquée avec le reste de la clause `FROM`, pour que les fonctions soient exécutées complètement avant de considérer la prochaine ligne provenant de la clause `FROM`.

S'il y a plus d'une fonction SETOF dans la liste du SELECT de la requête, le comportement est similaire à ce que vous obtiendriez en plaçant les fonctions dans une seule clause `FROM` de `LATERAL ROWS FROM( ... )`. Pour chaque ligne de la requête sous-jacente, il existe une ligne en sortie utilisant le premier résultat de chaque fonction, ensuite une ligne en sortie utilisant le deuxième résultat, et ainsi de suite. Si certaines des fonctions SETOF produisent moins de résultats que les autres, des valeurs NULL sont ajoutées pour les données manquantes, pour que le nombre total de lignes émises pour une ligne sous-jacente soit la même que pour la fonction SETOF qui a produit le plus de lignes. De ce fait, les fonctions SETOF s'exécutent complètement jusqu'à ce qu'elles aient terminé, puis l'exécution continue avec la prochaine ligne sous-jacente.

Les fonctions SETOF peuvent être imbriquées dans une liste SELECT, bien que cela ne soit pas autorisées dans les éléments d'une clause `FROM`. Dans de tels cas, chaque niveau d'imbrication est traité séparément, comme s'il s'agissait d'un élément `LATERAL ROWS FROM( ... )` séparé. Par exemple, dans

```
SELECT srf1(srf2(x), srf3(y)), srf4(srf5(z)) FROM tab;
```

les fonctions SETOF `srf2`, `srf3`, et `srf5` seront exécutées ligne par ligne pour chaque ligne de `tab`, puis `srf1` et `srf4` seront appliquées ligne par ligne pour chaque ligne produite par les fonctions inférieures.

Les fonctions SETOF ne peuvent pas être utilisées à l'intérieur de constructions d'évaluations conditionnelles, telles que `CASE` ou `COALESCE`. Ce comportement signifie aussi que des fonctions SETOF seront évaluées même quand il pourra apparaître qu'elles devraient être ignorées grâce à une construction d'évaluation conditionnelle, telle que `CASE` ou `COALESCE`. Par exemple, considérez :

```
SELECT x, CASE WHEN x > 0 THEN generate_series(1, 5) ELSE 0 END
 FROM tab;
```

Il pourrait sembler que cela produit cinq répétitions des lignes en entrée qui ont `x > 0`, et une seule répétition des autres parce que `generate_series(1, 5)` serait exécuté dans un élément `LATERAL FROM` implicite, l'expression `CASE` est toujours évaluée, elle produirait cinq répétitions de chaque ligne en entrée. Pour diminuer la confusion, ce genre de cas renvoie une erreur au moment de l'analyse.

Note

Si la dernière commande d'une fonction est `INSERT`, `UPDATE` ou `DELETE` avec une clause `RETURNING`, cette commande sera toujours exécutée jusqu'à sa fin, même si la fonction n'est pas déclarée avec `SETOF` ou que la requête appelante ne renvoie pas toutes les lignes résultats. Toutes les lignes supplémentaires produites par la clause `RETURNING` sont silencieusement abandonnées mais les modifications de table sont pris en compte (et sont toutes terminées avant que la fonction ne se termine).

Note

Avant PostgreSQL 10, placer plus d'une fonction renvoyant des lignes dans la même clause SELECT n'avait pas un comportement très simple, sauf si elles produisaient le même nombre de lignes. Dans le cas contraire, on obtenait un nombre de lignes en sortie égale au plus petit multiple commun du nombre de lignes produit par les différentes fonctions. De plus, les fonctions SETOF imbriquées ne fonctionnaient comme ce qui est décrit ci-dessus. À la place, une fonction EOF pouvait avoir tout au plus un argument SETOF, et chaque imbrication de fonctions SETOF était exécutée séparément. De plus, une exécution conditionnelle (fonctions SETOF à l'intérieur d'un `CASE`, etc) était auparavant autorisée, ce qui compliquait encore plus les choses. L'utilisation de la syntaxe `LATERAL` est recommandée lors de l'écriture de requêtes devant fonctionner avec les versions plus anciennes de PostgreSQL pour produire des résultats cohérents sur différentes versions. Si vous avez une requête qui se base sur une exécution conditionnelle d'une fonction SETOF, vous pourriez la corriger en déplaçant le test conditionnel dans une fonction SETOF personnalisée. Par exemple :

```
SELECT x, CASE WHEN y > 0 THEN generate_series(1, z) ELSE 5
 END FROM tab;
```

pourrait devenir

```
CREATE FUNCTION case_generate_series(cond bool, start int, fin
 int, els int)
  RETURNS SETOF int AS $$
BEGIN
  IF cond THEN
    RETURN QUERY SELECT generate_series(start, fin);
  ELSE
    RETURN QUERY SELECT els;
  END IF;
END$$ LANGUAGE plpgsql;

SELECT x, case_generate_series(y > 0, 1, z, 5) FROM tab;
```

Cette formulation fonctionnera de la même façon sur toutes les versions de PostgreSQL.

38.5.10. Fonctions SQL renvoyant TABLE

Il existe une autre façon de déclarer une fonction comme renvoyant un ensemble de données. Cela passe par la syntaxe `RETURNS TABLE(`*`colonnes`*`)`. C'est équivalent à utiliser un ou plusieurs

paramètres OUT et à marquer la fonction comme renvoyant un SETOF record (ou SETOF d'un type simple en sortie, comme approprié). Cette notation est indiquée dans les versions récentes du standard SQL et, du coup, devrait être plus portable que SETOF.

L'exemple précédent, sum-and-product, peut se faire aussi de la façon suivante :

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

Il n'est pas autorisé d'utiliser explicitement des paramètres OUT ou INOUT avec la notation RETURNS TABLE -- vous devez indiquer toutes les colonnes en sortie dans la liste TABLE.

38.5.11. Fonctions SQL polymorphes

Les fonctions SQL peuvent être déclarées pour accepter et renvoyer les types décrits dans Section 38.2.5. Voici une fonction polymorphe cree_tableau qui construit un tableau à partir de deux éléments de type arbitraire :

```
CREATE FUNCTION cree_tableau(anyelement, anyelement) RETURNS
 anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;
SELECT cree_tableau(1, 2) AS tableau_entier,
 cree_tableau('a'::text, 'b') AS
tableau_texte;

 tableau_entier | tableau_texte
----------------+---------------
 {1,2}          | {a,b}
(1 row)
```

Notez l'utilisation du transtypage 'a'::text pour spécifier le type text de l'argument. Ceci est nécessaire si l'argument est une chaîne de caractères car, autrement, il serait traité comme un type unknown, et un tableau de type unknown n'est pas un type valide. Sans le transtypage, vous obtiendrez ce genre d'erreur :

```
ERROR:  could not determine polymorphic type because input has type
 unknown
```

Avec make_array déclaré comme ci-dessus, vous devez fournir deux arguments qui sont exactement du même type de données ; le système ne tentera pas de résoudre les différences de type. De ce fait, par exemple, ceci ne fonctionnera pas :

```
SELECT make_array(1, 2.5) AS numericarray;
ERROR:  function make_array(integer, numeric) does not exist
```

Une approche alternative est d'utiliser la famille « commune » de types polymorphiques, qui autorise le système à identifier le type commun convenable :

```
CREATE FUNCTION make_array2(anycompatible, anycompatible)
RETURNS anycompatiblearray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array2(1, 2.5) AS numericarray;
 numericarray
--------------
 {1,2.5}
(1 row)
```

Comme les règles pour la résolution de type commun reviennent par défaut à choisir le type `text` quand toutes les entrées sont de type inconnu, ceci fonctionne aussi :

```
SELECT make_array2('a', 'b') AS textarray;
 textarray
-----------
 {a,b}
(1 row)
```

Il est permis d'avoir des arguments polymorphes avec un type de renvoi fixe, mais non l'inverse. Par exemple :

```
CREATE FUNCTION est_plus_grand(anyelement, anyelement) RETURNS bool
 AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT est_plus_grand(1, 2);
 est_plus_grand
----------------
 f
(1 row)

CREATE FUNCTION fonction_invalide() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result datatype
DETAIL:  A result of type anyelement requires at least one input of
 type anyelement, anyarray, anynonarray, anyenum, or anyrange.
```

Le polymorphisme peut être utilisé avec les fonctions qui ont des arguments en sortie. Par exemple :

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3
 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |   f3
----+---------
 22 | {22,22}
(1 row)
```

Le polymorphisme peut aussi être utilisé avec des fonctions variadic. Par exemple :

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS
 $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);
 anyleast
----------
       -1
(1 row)

SELECT anyleast('abc'::text, 'def');
 anyleast
----------
 abc
(1 row)

CREATE FUNCTION concat_values(text, VARIADIC anyarray) RETURNS text
 AS $$
    SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat_values('|', 1, 4, 2);
 concat_values
---------------
 1|4|2
(1 row)
```

38.5.12. Fonctions SQL et collationnement

Lorsqu'une fonction SQL dispose d'un ou plusieurs paramètres d'un type de données collationnable, le collationnement applicable est déterminé pour chacun des appels à la fonction afin de correspondre au collationnement assigné aux arguments, tel que décrit à la section Section 24.2. Si un collationnement peut être correctement identifié (c'est-à-dire qu'il ne subsiste aucun conflit entre les collationnements implicites des arguments), alors l'ensemble des paramètres collationnables sera traité en fonction de ce collationnement. Ce comportement peut donc avoir une incidence sur les opérations sensibles aux collationnements se trouvant dans le corps de la fonction. Par exemple, en utilisant la fonction `anyleast` décrite ci-dessus, le résultat de

```
SELECT anyleast('abc'::text, 'ABC');
```

dépendra du collationnement par défaut de l'instance. Ainsi, pour la locale `C`, le résultat sera `ABC`, alors que pour de nombreuses autres locales, la fonction retournera `abc`. L'utilisation d'un collationnement particulier peut être forcé lors de l'appel de la fonction en spécifiant la clause `COLLATE` pour chacun des arguments, par exemple

```
SELECT anyleast('abc'::text, 'ABC' COLLATE "C");
```

Par ailleurs, si vous souhaitez qu'une fonction opère avec un collationnement particulier, sans tenir compte du collationnement des paramètres qui lui seront fournis, il faudra alors spécifier la clause `COLLATE` souhaitée lors de la définition de la fonction. Cette version de la fonction `anyleast` utilisera systématiquement la locale `fr_FR` pour la comparaison des chaines de caractères :

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS
 $$
    SELECT min($1[i] COLLATE "fr_FR") FROM generate_subscripts($1,
 1) g(i);
$$ LANGUAGE SQL;
```

Mais il convient de bien noter que cette modification risque d'entraîner une erreur si des données d'un type non sensible au collationnement lui sont fournies.

Si aucun collationnement commun ne peut être déterminé entre les arguments fournis, la fonction SQL appliquera aux paramètres le collationnement par défaut de leur type de donnée (qui correspond généralement au collationnement par défaut de l'instance, mais qui peut différer entre des domaines différents).

Le comportement des paramètres collationnables peut donc être assimilé à une forme limitée de polymorphisme, uniquement applicable aux types de données textuels.

38.6. Surcharge des fonctions

Plusieurs fonctions peuvent être définies avec le même nom SQL à condition que les arguments soient différents. En d'autres termes, les noms de fonction peuvent être *surchargés*. Que vous l'utilisiez ou non, cette possibilité implique des précautions au niveau de la sécurité lors de l'appel de fonctions dans les bases de données où certains utilisateurs ne font pas confiance aux autres utilisateurs ; voir Section 10.3. Quand une requête est exécutée, le serveur déterminera la fonction à appeler à partir des types de données des arguments et du nombre d'arguments. La surcharge peut aussi être utilisée pour simuler des fonctions avec un nombre variable d'arguments jusqu'à un nombre maximum fini.

Lors de la création d'une famille de fonctions surchargées, vous devriez être attentif à ne pas créer d'ambiguïtés. Par exemple, avec les fonctions :

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

Savoir quelle fonction sera appelée avec une entrée triviale comme `test(1, 1.5)` n'est pas immédiatement clair. Les règles de résolution actuellement implémentées sont décrites dans le Chapitre 10 mais il est déconseillé de concevoir un système qui serait basé subtilement sur ce comportement.

Une fonction qui prend un seul argument d'un type composite devrait généralement ne pas avoir le même nom que tout attribut (champ) de ce type. Rappelez-vous que *`attribut`*`(`*`table`*`)` est considéré comme équivalent à *`table`*`.`*`attribut`*. Dans le cas où il existe une ambiguïté entre une fonction sur un type composite et sur un attribut d'un type composite, l'attribut sera toujours utilisé. Il est possible de contourner ce choix en qualifiant le nom de la fonction avec celui du schéma (c'est-à-dire *`schema`*`.`*`fonction`*`(`*`table`*`)`) mais il est préférable d'éviter le problème en ne choisissant aucun nom conflictuel.

Un autre conflit possible se trouve entre les fonctions variadic et les autres. En fait, il est possible de créer à la fois `foo(numeric)` et `foo(VARIADIC numeric[])`. Dans ce cas, il n'est pas simple de savoir lequel sera sélectionné lors d'un appel avec un seul argument numérique, par exemple `foo(10.1)`. La règle est que la fonction apparaissant plus tôt dans le chemin des schémas est utilisé. De même, si les deux fonctions sont dans le même schéma, la non variadic est préféré.

Lors de la surcharge de fonctions en langage C, il existe une contrainte supplémentaire : le nom C de chaque fonction dans la famille des fonctions surchargées doit être différent des noms C de toutes les autres fonctions, soit internes soit chargées dynamiquement Si cette règle est violée, le comportement n'est pas portable. Vous pourriez obtenir une erreur de l'éditeur de lien ou une des

fonctions sera appelée (habituellement l'interne). L'autre forme de clause AS pour la commande SQL CREATE FUNCTION découple le nom de la fonction SQL à partir du nom de la fonction dans le code source C. Par exemple :

```
CREATE FUNCTION test(int) RETURNS int
    AS 'filename', 'test_1arg'
LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'filename', 'test_2arg'
LANGUAGE C;
```

Les noms des fonctions C reflètent ici une des nombreuses conventions possibles.

38.7. Catégories de volatilité des fonctions

Chaque fonction a une classification de volatilité (*volatility*) comprenant VOLATILE, STABLE ou IMMUTABLE. VOLATILE est la valeur par défaut si la commande CREATE FUNCTION ne spécifie pas de catégorie. La catégorie de volatilité est une promesse à l'optimiseur sur le comportement de la fonction :

- Une fonction VOLATILE peut tout faire, y compris modifier la base de données. Elle peut renvoyer différents résultats sur des appels successifs avec les mêmes arguments. L'optimiseur ne fait aucune supposition sur le comportement de telles fonctions. Une requête utilisant une fonction volatile réévaluera la fonction à chaque ligne où sa valeur est nécessaire.

- Une fonction STABLE ne peut pas modifier la base de données et est garantie de renvoyer les mêmes résultats si elle est appelée avec les mêmes arguments pour toutes les lignes à l'intérieur d'une même instruction. Cette catégorie permet à l'optimiseur d'optimiser plusieurs appels de la fonction dans une seule requête. En particulier, vous pouvez utiliser en toute sécurité une expression contenant une telle fonction dans une condition de parcours d'index (car un parcours d'index évaluera la valeur de la comparaison une seule fois, pas une fois pour chaque ligne, utiliser une fonction VOLATILE dans une condition de parcours d'index n'est pas valide).

- Une fonction IMMUTABLE ne peut pas modifier la base de données et est garantie de toujours renvoyer les mêmes résultats si elle est appelée avec les mêmes arguments. Cette catégorie permet à l'optimiseur de pré-évaluer la fonction quand une requête l'appelle avec des arguments constants. Par exemple, une requête comme SELECT ... WHERE x = 2 + 2 peut être simplifiée pour obtenir SELECT ... WHERE x = 4 car la fonction sous-jacente de l'opérateur d'addition est indiquée IMMUTABLE.

Pour une meilleure optimisation des résultats, vous devez mettre un label sur les fonctions avec la catégorie la plus volatile valide pour elles.

Toute fonction avec des effets de bord *doit* être indiquée comme VOLATILE, de façon à ce que les appels ne puissent pas être optimisés. Même une fonction sans effets de bord doit être indiquée comme VOLATILE si sa valeur peut changer à l'intérieur d'une seule requête ; quelques exemples sont random(), currval(), timeofday().

Un autre exemple important est que la famille de fonctions current_timestamp est qualifiée comme STABLE car leurs valeurs ne changent pas à l'intérieur d'une transaction.

Il y a relativement peu de différences entre les catégories STABLE et IMMUTABLE en considérant les requêtes interactives qui sont planifiées et immédiatement exécutées : il importe peu que la fonction soit exécutée une fois lors de la planification ou une fois au lancement de l'exécution de la requête mais cela fait une grosse différence si le plan est sauvegardé et utilisé plus tard. Placer un label IMMUTABLE sur une fonction quand elle ne l'est pas vraiment pourrait avoir comme conséquence de la considérer prématurément comme une constante lors de la planification et résulterait en une

valeur erronée lors d'une utilisation ultérieure de ce plan d'exécution. C'est un danger qui arrive lors de l'utilisation d'instructions préparées ou avec l'utilisation de langages de fonctions mettant les plans d'exécutions en cache (comme PL/pgSQL).

Pour les fonctions écrites en SQL ou dans tout autre langage de procédure standard, la catégorie de volatibilité détermine une deuxième propriété importante, à savoir la visibilité de toute modification de données effectuées par la commande SQL qui a appelé la fonction. Une fonction `VOLATILE` verra les changements, une fonction `STABLE` ou `IMMUTABLE` ne les verra pas. Ce comportement est implémenté en utilisant le comportement par images de MVCC (voir Chapitre 13) : les fonctions `STABLE` et `IMMUTABLE` utilisent une image établie au lancement de la requête appelante alors que les fonctions `VOLATILE` obtiennent une image fraiche au début de chaque requête qu'elles exécutent.

> **Note**
>
> Les fonctions écrites en C peuvent gérer les images de la façon qu'elles le souhaitent, mais il est préférable de coder les fonctions C de la même façon.

À cause du comportement à base d'images, une fonction contenant seulement des commandes `SELECT` peut être indiquée `STABLE` en toute sécurité même s'il sélectionne des données à partir de tables qui pourraient avoir subi des modifications entre temps par des requêtes concurrentes. PostgreSQL exécutera toutes les commandes d'une fonction `STABLE` en utilisant l'image établie par la requête appelante et n'aura qu'une vision figée de la base de données au cours de la requête.

Ce même comportement d'images est utilisé pour les commandes `SELECT` à l'intérieur de fonctions `IMMUTABLE`. Il est généralement déconseillé de sélectionner des tables de la base de données à l'intérieur de fonctions `IMMUTABLE` car l'immutabilité sera rompue si le contenu de la table change. Néanmoins, PostgreSQL ne vous force pas à ne pas le faire.

Une erreur commune est de placer un label sur une fonction `IMMUTABLE` quand son résultat dépend d'un paramètre de configuration. Par exemple, une fonction qui manipule des types date/heure pourrait bien avoir des résultats dépendant du paramètre TimeZone. Pour être sécurisées, de telles fonctions devraient avoir le label `STABLE` à la place.

> **Note**
>
> PostgreSQL requiert que les fonctions `STABLE` et `IMMUTABLE` ne contiennent aucune commande SQL autre que `SELECT` pour éviter les modifications de données (ceci n'a pas été complètement testé car de telles fonctions pourraient toujours appeler des fonctions `VOLATILE` qui modifient la base de données. Si vous le faites, vous trouverez que la fonction `STABLE` ou `IMMUTABLE` n'est pas au courant des modifications effectuées sur la base de données par la fonction appelée, car elles sont cachées depuis son image).

38.8. Fonctions en langage de procédures

PostgreSQL autorise l'écriture de fonctions définies par l'utilisateur dans d'autres langages que SQL et C. Ces autres langages sont appelés des *langages de procédure* (PL). Les langages de procédures ne sont pas compilés dans le serveur PostgreSQL ; ils sont fournis comme des modules chargeables. Voir le Chapitre 42 et les chapitres suivants pour plus d'informations.

Il y a actuellement quatre langages de procédures disponibles dans la distribution PostgreSQL standard : PL/pgSQL, PL/Tcl, PL/Perl et PL/Python. Référez-vous au Chapitre 42 pour plus d'informations. D'autres langages peuvent être définis par les utilisateurs. Les bases du développement d'un nouveau langage de procédures sont traitées dans le Chapitre 58.

38.9. Fonctions internes

Les fonctions internes sont des fonctions écrites en C qui ont été liées de façon statique dans le serveur PostgreSQL. Le « corps » de la définition de la fonction spécifie le nom en langage C de la fonction, qui n'est pas obligatoirement le même que le nom déclaré pour l'utilisation en SQL (pour des raisons de rétro compatibilité, un corps vide est accepté pour signifier que le nom de la fonction en langage C est le même que le nom SQL).

Normalement, toutes les fonctions internes présentes dans le serveur sont déclarées pendant l'initialisation du groupe de base de données (voir Section 19.2) mais un utilisateur peut utiliser la commande `CREATE FUNCTION` pour créer des noms d'alias supplémentaires pour une fonction interne. Les fonctions internes sont déclarées dans la commande `CREATE FUNCTION` avec le nom de langage `internal`. Par exemple, pour créer un alias de la fonction `sqrt` :

```
CREATE FUNCTION racine_carree(double precision) RETURNS double
 precision    AS
'dsqrt'
LANGUAGE internal    STRICT;
```

(la plupart des fonctions internes doivent être déclarées « STRICT »)

> **Note**
>
> Toutes les fonctions « prédéfinies » ne sont pas internes (au sens explicité ci-dessus). Quelques fonctions prédéfinies sont écrites en SQL.

38.10. Fonctions en langage C

Les fonctions définies par l'utilisateur peuvent être écrites en C (ou dans un langage pouvant être rendu compatible avec C, comme le C++). Ces fonctions sont compilées en objets dynamiques chargeables (encore appelés bibliothèques partagées) et sont chargées par le serveur à la demande. Cette caractéristique de chargement dynamique est ce qui distingue les fonctions en « langage C » des fonctions « internes » -- les véritables conventions de codage sont essentiellement les mêmes pour les deux (c'est pourquoi la bibliothèque standard de fonctions internes est une source abondante d'exemples de code pour les fonctions C définies par l'utilisateur).

Actuellement, seule une convention d'appel est utilisée pour les fonctions C (« version 1 »). Le support pour cette convention d'appel est indiqué en ajoutant un appel à la macro `PG_FUNCTION_INFO_V1()` pour la fonction, comme illustré ci-dessous.

38.10.1. Chargement dynamique

La première fois qu'une fonction définie par l'utilisateur dans un fichier objet particulier chargeable est appelée dans une session, le chargeur dynamique charge ce fichier objet en mémoire de telle sorte que la fonction peut être appelée. La commande `CREATE FUNCTION` pour une fonction en C définie par l'utilisateur doit par conséquent spécifier deux éléments d'information pour la fonction : le nom du fichier objet chargeable et le nom en C (lien symbolique) de la fonction spécifique à appeler à l'intérieur de ce fichier objet. Si le nom en C n'est pas explicitement spécifié, il est supposé être le même que le nom de la fonction SQL.

L'algorithme suivant, basé sur le nom donné dans la commande `CREATE FUNCTION`, est utilisé pour localiser le fichier objet partagé :

1. Si le nom est un chemin absolu, le fichier est chargé.

2. Si le nom commence par la chaîne `$libdir`, cette chaîne est remplacée par le nom du répertoire de la bibliothèque du paquetage PostgreSQL, qui est déterminé au moment de la compilation.

3. Si le nom ne contient pas de partie répertoire, le fichier est recherché par le chemin spécifié dans la variable de configuration dynamic_library_path.

4. Dans les autres cas, (nom de fichier non trouvé dans le chemin ou ne contenant pas de partie répertoire non absolu), le chargeur dynamique essaiera d'utiliser le nom donné, ce qui échouera très vraisemblablement (dépendre du répertoire de travail en cours n'est pas fiable).

Si cette séquence ne fonctionne pas, l'extension pour les noms de fichier des bibliothèques partagées spécifiques à la plateforme (souvent `.so`) est ajoutée au nom attribué et la séquence est à nouveau tentée. En cas de nouvel échec, le chargement échoue.

Il est recommandé de localiser les bibliothèques partagées soit relativement à `$libdir` ou via le chemin dynamique des bibliothèques. Ceci simplifie les mises à jour de versions si la nouvelle installation est à un emplacement différent. Le répertoire actuel représenté par `$libdir` est trouvable avec la commande `pg_config --pkglibdir`.

L'identifiant utilisateur sous lequel fonctionne le serveur PostgreSQL doit pouvoir suivre le chemin jusqu'au fichier que vous essayez de charger. Une erreur fréquente revient à définir le fichier ou un répertoire supérieur comme non lisible et/ou non exécutable par l'utilisateur postgres.

Dans tous les cas, le nom de fichier donné dans la commande `CREATE FUNCTION` est enregistré littéralement dans les catalogues systèmes, de sorte que, si le fichier doit être à nouveau chargé, la même procédure sera appliquée.

> **Note**
>
> PostgreSQL ne compilera pas une fonction C automatiquement. Le fichier objet doit être compilé avant d'être référencé dans une commande `CREATE FUNCTION`. Voir la Section 38.10.5 pour des informations complémentaires.

Pour s'assurer qu'un fichier objet chargeable dynamiquement n'est pas chargé dans un serveur incompatible, PostgreSQL vérifie que le fichier contient un « bloc magique » avec un contenu approprié. Ceci permet au serveur de détecter les incompatibilités évidentes comme du code compilé pour une version majeure différente de PostgreSQL. Pour inclure un bloc magique, écrivez ceci dans un (et seulement un) des fichiers source du module, après avoir inclus l'en-tête `fmgr.h` :

```
PG_MODULE_MAGIC;
```

Après avoir été utilisé pour la première fois, un fichier objet chargé dynamiquement est conservé en mémoire. Les futurs appels de fonction(s) dans ce fichier pendant la même session provoqueront seulement une légère surcharge due à la consultation d'une table de symboles. Si vous devez forcer le chargement d'un fichier objet, par exemple après une recompilation, commencez une nouvelle session.

De façon optionnelle, un fichier chargé dynamiquement peut contenir une fonction d'initialisation. Si le fichier inclut une fonction nommée `_PG_init`, cette fonction sera appelée immédiatement après le chargement du fichier. La fonction ne reçoit aucun paramètre et doit renvoyer void. Il n'existe actuellement aucun moyen de décharger un fichier chargé dynamiquement.

38.10.2. Types de base dans les fonctions en langage C

Pour savoir comment écrire des fonctions en langage C, vous devez savoir comment PostgreSQL représente en interne les types de données de base et comment elles peuvent être passés vers et depuis les fonctions. En interne, PostgreSQL considère un type de base comme un « blob de mémoire ». Les fonctions que vous définissez sur un type définissent à leur tour la façon que PostgreSQL opère sur lui. C'est-à-dire que PostgreSQL ne fera que conserver et retrouver les données sur le disque et utilisera votre fonction pour entrer, traiter et restituer les données.

Les types de base peuvent avoir un des trois formats internes suivants :

- passage par valeur, longueur fixe ;
- passage par référence, longueur fixe ;
- passage par référence, longueur variable.

Les types par valeur peuvent seulement avoir une longueur de 1, 2 ou 4 octets (également 8 octets si `sizeof(Datum)` est de huit octets sur votre machine). Vous devriez être attentif lors de la définition de vos types de sorte à qu'ils aient la même taille sur toutes les architectures. Par exemple, le type `long` est dangereux car il a une taille de quatre octets sur certaines machines et huit octets sur d'autres, alors que le type `int` est de quatre octets sur la plupart des machines Unix. Une implémentation raisonnable du type `int4` sur une machine Unix pourrait être

```
/* entier sur quatre octets, passé par valeur */
typedef int int4;
```

(le code C de PostgreSQL appelle ce type `int32` car il existe une convention en C disant que `int`*XX* signifie *XX bits*. Il est à noter toutefois que le type C `int8` a une taille d'un octet. Le type SQL `int8` est appelé `int64` en C. Voir aussi Tableau 38.2.)

D'autre part, les types à longueur fixe d'une taille quelconque peuvent être passés par référence. Par exemple, voici l'implémentation d'un type PostgreSQL :

```
/* structure de 16 octets, passée par référence */
typedef struct
{
    double  x, y;
} Point;
```

Seuls des pointeurs vers de tels types peuvent être utilisés en les passant dans et hors des fonctions PostgreSQL. Pour renvoyer une valeur d'un tel type, allouez la quantité appropriée de mémoire avec `palloc`, remplissez la mémoire allouée et renvoyez un pointeur vers elle (de plus, si vous souhaitez seulement renvoyer la même valeur qu'un de vos arguments en entrée qui se trouve du même type, vous pouvez passer le `palloc` supplémentaire et simplement renvoyer le pointeur vers la valeur en entrée).

Enfin, tous les types à longueur variable doivent aussi être passés par référence. Tous les types à longueur variable doivent commencer avec un champ d'une longueur d'exactement quatre octets, qui sera initialisé à `SET_VARSIZE` ; ne jamais configurer ce champ directement ! Toutes les données devant être stockées dans ce type doivent être localisées dans la mémoire à la suite immédiate de ce champ longueur. Le champ longueur contient la longueur totale de la structure, c'est-à-dire incluant la longueur du champ longueur lui-même.

Un autre point important est d'éviter de laisser des bits non initialisés dans les structures de types de données ;; par exemple, prenez bien soin de remplir avec des zéros tous les octets de remplissage qui sont présents dans les structures de données à des fins d'alignement. À défaut, des constantes logiquement équivalentes de vos types de données pourraient être considérées comme inégales par l'optimiseur, impliquant une planification inefficace (bien que les résultats puissent malgré tout être corrects).

> **Avertissement**
>
> Ne *jamais* modifier le contenu d'une valeur en entrée passée par référence. Si vous le faites, il y a de forts risques pour que vous réussissiez à corrompre les données sur disque car le pointeur que vous avez reçu pourrait bien pointer directement vers un tampon disque. La seule exception à cette règle est expliquée dans la Section 38.12.

Comme exemple, nous pouvons définir le type `text` comme ceci :

```
typedef struct {
    int32 length;
    char data[FLEXIBLE_ARRAY_MEMBER];
} text;
```

La notation `[FLEXIBLE_ARRAY_MEMBER]` signifie que la longueur actuelle de la donnée n'est pas indiquée par cette déclaration.

En manipulant les types à longueur variable, nous devons être attentifs à allouer la quantité correcte de mémoire et à fixer correctement le champ longueur. Par exemple, si nous voulons stocker 40 octets dans une structure `text`, nous devrions utiliser un fragment de code comme celui-ci :

```
#include "postgres.h"
...
char buffer[40]; /* notre donnée source */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

`VARHDRSZ` est équivalent à `sizeof(int32)` mais est considéré comme une meilleure tournure de référence à la taille de l'overhead pour un type à longueur variable. De plus, le champ de longueur *doit* être configuré en utilisant la macro `SET_VARSIZE`, pas une simple affectation.

Le Tableau 38.2 spécifie la correspondance entre les types C et certains des types internes SQL de PostgreSQL. La colonne « Défini dans » donne le fichier d'en-tête devant être inclus pour accéder à la définition du type (la définition effective peut se trouver dans un fichier différent inclus dans le fichier indiqué). Il est recommandé que les utilisateurs s'en tiennent à l'interface définie). Notez que vous devriez toujours inclure `postgres.h` en premier dans tout fichier source du code serveur car il déclare un grand nombre d'éléments dont vous aurez besoin de toute façon et parce qu'inclure d'autres en-têtes en premier pourrait causer des problèmes de portabilité.

Tableau 38.2. Équivalence des types C et des types SQL intégrés

Type SQL	Type C	Défini dans
`boolean`	`bool`	`postgres.h` (peut-être interne compilateur)
`box`	`BOX*`	`utils/geo_decls.h`
`bytea`	`bytea*`	`postgres.h`
`"char"`	`char`	(interne compilateur)
`character`	`BpChar*`	`postgres.h`
`cid`	`CommandId`	`postgres.h`
`date`	`DateADT`	`utils/date.h`
`float4` (`real`)	`float4`	`postgres.h`

Type SQL	Type C	Défini dans
float8 (double precision)	float8	postgres.h
int2 (smallint)	int16	postgres.h
int4 (integer)	int32	postgres.h
int8 (bigint)	int64	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
numeric	Numeric	utils/numeric.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	RegProcedure	postgres.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp	datatype/timestamp.h
timestamp with time zone	TimestampTz	datatype/timestamp.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

Maintenant que nous avons passé en revue toutes les structures possibles pour les types de base, nous pouvons donner quelques exemples de vraies fonctions.

38.10.3. Conventions d'appel de la version 1

La convention d'appel version-1 repose sur des macros pour supprimer la plus grande partie de la complexité du passage d'arguments et de résultats. La déclaration C d'une fonction en version-1 est toujours :

```
Datum nom_fonction(PG_FUNCTION_ARGS)
```

De plus, la macro d'appel :

```
PG_FUNCTION_INFO_V1(nom_fonction);
```

doit apparaître dans le même fichier source (par convention, elle est écrite juste avant la fonction elle-même). Cette macro n'est pas nécessaire pour les fonctions `internal` puisque PostgreSQL assume que toutes les fonctions internes utilisent la convention version-1. Elle est toutefois requise pour les fonctions chargées dynamiquement.

Dans une fonction version-1, chaque argument existant est traité par une macro `PG_GETARG_`*xxx*`()` correspondant au type de donnée de l'argument. (Dans les fonctions non strictes, il est nécessaire d'avoir une vérification précédente sur la possibilité que l'argument soit NULL en utilisant

PG_ARGISNULL() ; voir ci-dessous.) Le résultat est renvoyé par une macro PG_RETURN_*xxx*() correspondant au type renvoyé. PG_GETARG_*xxx*() prend comme argument le nombre d'arguments de la fonction à parcourir, le compteur commençant à 0. PG_RETURN_*xxx*() prend comme argument la valeur effective à renvoyer.

Voici quelques exemples utilisant la convention d'appel version-1 :

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

PG_MODULE_MAGIC;

/* par valeur */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* par référence, longueur fixe */

PG_FUNCTION_INFO_V1(add_one_float8);

Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* La macro pour FLOAT8 cache sa nature de passage par
 référence. */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Ici, la nature de passage par référence de Point n'est pas
 cachée. */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* par référence, longueur variable */
```

```
PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_PP(0);
    /*
     * VARSIZE_ANY_EXHDR is the size of the struct in bytes, minus
 the
     * VARHDRSZ or VARHDRSZ_SHORT of its header.  Construct the
 copy with a
     * full-length header.
     */
    text     *new_t = (text *) palloc(VARSIZE_ANY_EXHDR(t) +
 VARHDRSZ);
    SET_VARSIZE(new_t, VARSIZE_ANY_EXHDR(t) + VARHDRSZ);

    /*
     * VARDATA is a pointer to the data region of the new struct.
 The source
     * could be a short datum, so retrieve its data through
 VARDATA_ANY.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA_ANY(t), /* source */
           VARSIZE_ANY_EXHDR(t));   /* how many bytes */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_PP(0);
    text  *arg2 = PG_GETARG_TEXT_PP(1);
    int32 arg1_size = VARSIZE_ANY_EXHDR(arg1);
    int32 arg2_size = VARSIZE_ANY_EXHDR(arg2);
    int32 new_text_size = arg1_size + arg2_size + VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA_ANY(arg1), arg1_size);
    memcpy(VARDATA(new_text) + arg1_size, VARDATA_ANY(arg2),
 arg2_size);
    PG_RETURN_TEXT_P(new_text);
}
```

En supposant que le code ci-dessus a été enregistré dans le fichier `funcs.c` et compilé en un objet partagé, nous pouvons définir les fonctions dans PostgreSQL avec les commandes suivantes :

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS 'DIRECTORY/funcs', 'add_one'
     LANGUAGE C STRICT;

-- note overloading of SQL function name "add_one"
```

```
CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS 'DIRECTORY/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'DIRECTORY/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS 'DIRECTORY/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'DIRECTORY/funcs', 'concat_text'
     LANGUAGE C STRICT;
```

Ici, *`DIRECTORY`* indique le répertoire de la bibliothèque partagée (par exemple le répertoire du tutoriel PostgreSQL qui contient le code des exemples utilisés dans cette section). (Il serait préférable d'utiliser seulement `'funcs'` dans la clause AS, après avoir ajouté *`DIRECTORY`* au chemin de recherches. Dans tous les cas, nous pouvons omettre l'extension spécifique du système pour une bibliothèque partagée, généralement `.so`.)

Notez que nous avons spécifié les fonctions comme « strict », ceci signifiant que le système pourrait automatiquement supposer un résultat NULL si une des valeurs en entrée était NULL. En le faisant, nous évitons la vérification des entrées NULL dans le code de la fonction. Sans cela, nous devrions vérifier les valeurs NULL explicitement en utilisant la macro `PG_ARGISNULL()`.

La macro `PG_ARGISNULL(`*`n`*`)` permet à une fonction de tester si chaque entrée est NULL (évidemment, ceci n'est nécessaire que pour les fonctions déclarées non « STRICT »). Comme avec les macros `PG_GETARG_`*`xxx`*`()`, les arguments en entrée sont comptés à partir de zéro. Notez qu'on doit se garder d'exécuter `PG_GETARG_`*`xxx`*`()` jusqu'à ce qu'on ait vérifié que l'argument n'est pas NULL. Pour renvoyer un résultat NULL, exécutez la fonction `PG_RETURN_NULL()` ; ceci convient aussi bien dans les fonctions STRICT que non STRICT.

Au premier coup d'œud, les conventions de codage version-1 pourraient ressembler à de l'obscurantisme sans raison. Il pourrait sembler préférable d'utiliser les conventions d'appel `C`. Néanmoins, elles permettent de gérer des valeurs NULL pour les arguments et la valeur de retour calling, ainsi que des valeurs TOAST (compressées ou hors-ligne).

Les autres options proposées dans l'interface version 1 sont deux variantes des macros `PG_GETARG_`*`xxx`*`()`. La première d'entre elles, `PG_GETARG_`*`xxx`*`_COPY()`, garantit le renvoi d'une copie de l'argument spécifié où nous pouvons écrire en toute sécurité (les macros normales peuvent parfois renvoyer un pointeur vers une valeur physiquement mise en mémoire dans une table qui ne doit pas être modifiée). En utilisant les macros `PG_GETARG_`*`xxx`*`_COPY()`, on garantit l'écriture du résultat). La seconde variante se compose des macros `PG_GETARG_`*`xxx`*`_SLICE()` qui prennent trois arguments. Le premier est le nombre d'arguments de la fonction (comme ci-dessus). Le second et le troisième sont le décalage et la longueur du segment qui doit être renvoyé. Les décalages sont comptés à partir de zéro et une longueur négative demande le renvoi du reste de la valeur. Ces macros procurent un accès plus efficace à des parties de valeurs à grande dimension dans le cas où elles ont un type de stockage en mémoire « external » (le type de stockage d'une colonne peut être spécifié en utilisant `ALTER TABLE` *`nom_table`* `ALTER COLUMN` *`nom_colonne`* `SET STORAGE` *`typestockage`*. *`typestockage`* est un type parmi `plain`, `external`, `extended` ou `main`).

Enfin, les conventions d'appels de la version-1 rendent possible le renvoi de résultats d'ensemble (Section 38.10.8), l'implémentation de fonctions triggers (Chapitre 39) et d'opérateurs d'appel de langage procédural (Chapitre 58). Pour plus de détails, voir `src/backend/utils/fmgr/README` dans les fichiers sources de la distribution.

38.10.4. Écriture du code

Avant de nous intéresser à des sujets plus avancés, nous devons discuter de quelques règles de codage des fonctions en langage C de PostgreSQL. Bien qu'il soit possible de charger des fonctions écrites dans des langages autres que le C dans PostgreSQL, c'est habituellement difficile (quand c'est possible) parce que les autres langages comme C++, FORTRAN ou Pascal ne suivent pas fréquemment les mêmes conventions de nommage que le C. C'est-à-dire que les autres langages ne passent pas les arguments et ne renvoient pas les valeurs entre fonctions de la même manière. Pour cette raison, nous supposerons que nos fonctions en langage C sont réellement écrites en C.

Les règles de base pour l'écriture de fonctions C sont les suivantes :

- Utilisez `pg_config --includedir-server` pour découvrir où sont installés les fichiers d'en-tête du serveur PostgreSQL sur votre système (ou sur le système de vos utilisateurs).

- Compilez et liez votre code de façon à ce qu'il soit chargé dynamiquement dans PostgreSQL, ce qui requiert des informations spéciales. Voir Section 38.10.5 pour une explication détaillée sur la façon de le faire pour votre système d'exploitation spécifique.

- Rappelez-vous de définir un « bloc magique » pour votre bibliothèque partagée, comme décrit dans Section 38.10.1.

- Quand vous allouez de la mémoire, utilisez les fonctions PostgreSQL `palloc` et `pfree` au lieu des fonctions correspondantes `malloc` et `free` de la bibliothèque C. La mémoire allouée par `palloc` sera libérée automatiquement à la fin de chaque transaction, empêchant des débordements de mémoire.

- Remettez toujours à zéro les octets de vos structures en utilisant `memset` (ou allouez-les avec la fonction `palloc0`). Même si vous assignez chacun des champs de votre structure, il pourrait rester des espaces de remplissage (trous dans la structure) afin de respecter l'alignement des données qui contiennent des valeurs parasites. Sans cela, il sera difficile de calculer des hachages pour les index ou les jointures, dans la mesure où vous devrez uniquement tenir compte des octets significatifs de vos structures de données pour calculer ces hachages. Le planificateur se base également sur des comparaisons de constantes via des égalités de bits, aussi vous pouvez obtenir des planifications incorrectes si des valeurs logiquement équivalentes ne sont pas identiques bit à bit.

- La plupart des types internes PostgreSQL sont déclarés dans `postgres.h` alors que les interfaces de gestion des fonctions (`PG_FUNCTION_ARGS`, etc.) sont dans `fmgr.h`. Du coup, vous aurez besoin d'inclure au moins ces deux fichiers. Pour des raisons de portabilité, il vaut mieux inclure `postgres.h` *en premier* avant tout autre fichier d'en-tête système ou utilisateur. En incluant `postgres.h`, il incluera également `elog.h` et `palloc.h` pour vous.

- Les noms de symboles définis dans les objets ne doivent pas entrer en conflit entre eux ou avec les symboles définis dans les exécutables du serveur PostgreSQL. Vous aurez à renommer vos fonctions ou variables si vous recevez un message d'erreur à cet effet.

38.10.5. Compiler et lier des fonctions chargées dynamiquement

Avant de pouvoir être utilisées dans PostgreSQL, les fonctions d'extension écrites en C doivent être compilées et liées d'une certaine façon, ceci afin de produire un fichier dynamiquement chargeable par le serveur. Pour être plus précis, une *bibliothèque partagée* doit être créée.

Pour obtenir plus d'informations que celles contenues dans cette section, il faut se référer à la documentation du système d'exploitation, en particulier les pages traitant du compilateur C, de `cc` et de l'éditeur de lien, `ld`. Par ailleurs, le code source de PostgreSQL contient de nombreux exemples fonctionnels dans le répertoire `contrib`. Néanmoins, ces exemples entraînent la création de modules qui dépendent de la disponibilité du code source de PostgreSQL.

La création de bibliothèques partagées est un processus analogue à celui utilisé pour lier des exécutables : les fichiers sources sont d'abord compilés en fichiers objets puis sont liés ensemble. Les fichiers objets doivent être compilés sous la forme de *code indépendant de sa position* (PIC, acronyme de *position-independent code*) . Conceptuellement, cela signifie qu'ils peuvent être placés dans une position arbitraire de la mémoire lorsqu'ils sont chargés par l'exécutable. (Les fichiers objets destinés aux exécutables ne sont généralement pas compilés de cette manière.) La commande qui permet de lier des bibliothèques partagées nécessite des options spéciales qui la distinguent de celle permettant de lier un exécutable. En théorie, tout du moins. La réalité est, sur certains systèmes, beaucoup plus complexe.

Les exemples suivants considèrent que le code source est un fichier `foo.c` et qu'une bibliothèque partagée `foo.so` doit être créée. Sans précision, le fichier objet intermédiaire est appelé `foo.o`. Une bibliothèque partagée peut contenir plusieurs fichiers objet. Cela dit, un seul est utilisé ici.

FreeBSD

L'option du compilateur pour créer des PIC est `-fPIC`. L'option de l'éditeur de liens pour créer des bibliothèques partagées est `-shared`.

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

Ceci est applicable à partir de la version 3.0 de FreeBSD.

HP-UX

L'option du compilateur du système pour créer des PIC est `+z`. Avec GCC, l'option est `-fPIC`. Le commutateur de l'éditeur de liens pour les bibliothèques partagées est `-b`. Ainsi :

```
cc +z -c foo.c
```

ou :

```
gcc -fPIC -c foo.c
```

puis :

```
ld -b -o foo.sl foo.o
```

HP-UX utilise l'extension `.sl` pour les bibliothèques partagées, à la différence de la plupart des autres systèmes.

Linux

L'option du compilateur pour créer des PIC est `-fPIC`. L'option de compilation pour créer des bibliothèques partagées est `-shared`. Un exemple complet ressemble à :

```
cc -fPIC -c foo.c
cc -shared -o foo.so foo.o
```

macOS

L'exemple suivant suppose que les outils de développement sont installés.

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

L'option du compilateur pour créer des PIC est `-fPIC`. Pour les systèmes ELF, l'option de compilation pour lier les bibliothèques partagées est `-shared`. Sur les systèmes plus anciens et non-ELF, on utilise `ld -Bshareable`.

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

L'option du compilateur pour créer des PIC est `-fPIC`. Les bibliothèques partagées peuvent être créées avec `ld -Bshareable`.

```
gcc -fPIC -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

L'option du compilateur pour créer des PIC est `-KPIC` avec le compilateur de Sun et `-fPIC` avec GCC. Pour lier les bibliothèques partagées, l'option de compilation est respectivement `-G` ou `-shared`.

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

ou

```
gcc -fPIC -c foo.c
gcc -G -o foo.so foo.o
```

> **Astuce**
>
> Si cela s'avère trop compliqué, GNU Libtool[1] peut être utilisé. Cet outil permet de s'affranchir des différences entre les nombreux systèmes au travers d'une interface uniformisée.

La bibliothèque partagée résultante peut être chargée dans PostgreSQL. Lorsque l'on précise le nom du fichier dans la commande `CREATE FUNCTION`, il faut indiquer le nom de la bibliothèque partagée et non celui du fichier objet intermédiaire. L'extension standard pour les bibliothèques partagées (en général `.so` ou `.sl`) peut être omise dans la commande `CREATE FUNCTION`, et doit l'être pour une meilleure portabilité.

La Section 38.10.1 indique l'endroit où le serveur s'attend à trouver les fichiers de bibliothèques partagées.

38.10.6. Arguments de type composite

Les types composites n'ont pas une organisation fixe comme les structures en C. Des instances d'un type composite peuvent contenir des champs NULL. De plus, les types composites faisant partie d'une hiérarchie d'héritage peuvent avoir des champs différents des autres membres de la même hiérarchie. En conséquence, PostgreSQL propose une interface de fonction pour accéder depuis le C aux champs des types composites.

Supposons que nous voulions écrire une fonction pour répondre à la requête :

[1] https://www.gnu.org/software/libtool/

```
SELECT nom, c_surpaye(emp, 1500) AS surpaye
    FROM emp
    WHERE nom = 'Bill' OR nom = 'Sam';
```

En utilisant les conventions d'appel de la version 1, nous pouvons définir `c_surpaye` comme :

```
#include "postgres.h"
#include "executor/executor.h"  /* pour GetAttributeByName() */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(c_surpaye);

Datum
c_surpaye(PG_FUNCTION_ARGS)
{
    HeapTupleHeader  *t = (HeapTupleHeader *)
 PG_GETARG_HEAPTUPLEHEADER(0);
    int32            limite = PG_GETARG_INT32(1);
    bool isNULL;
    Datum salaire;

    salaire = GetAttributeByName(t, "salaire", &isNULL);
    if (isNULL)
        PG_RETURN_BOOL(false);
    /* Autrement, nous pourrions préférer de lancer
 PG_RETURN_NULL() pour un
       salaire NULL.
    */

    PG_RETURN_BOOL(DatumGetInt32(salaire) > limite);
}
```

`GetAttributeByName` est la fonction système PostgreSQL qui renvoie les attributs depuis une colonne spécifiée. Elle a trois arguments : l'argument de type `HeapTupleHeader` passé à la fonction, le nom de l'attribut recherché et un paramètre de retour qui indique si l'attribut est NULL. `GetAttributeByName` renvoie une valeur de type `Datum` que vous pouvez convertir dans un type voulu en utilisant la macro appropriée `DatumGet`*`XXX`*`()`. Notez que la valeur de retour est insignifiante si le commutateur NULL est positionné ; il faut toujours vérifier le commutateur NULL avant de commencer à faire quelque chose avec le résultat.

Il y a aussi `GetAttributeByNum`, qui sélectionne l'attribut ciblé par le numéro de colonne au lieu de son nom.

La commande suivante déclare la fonction `c_surpaye` en SQL :

```
CREATE FUNCTION c_surpaye(emp, integer) RETURNS boolean
    AS 'DIRECTORY/funcs', 'c_surpaye'
    LANGUAGE C STRICT;
```

Notez que nous avons utilisé `STRICT` pour que nous n'ayons pas à vérifier si les arguments en entrée sont NULL.

38.10.7. Renvoi de lignes (types composites)

Pour renvoyer une ligne ou une valeur de type composite à partir d'une fonction en langage C, vous pouvez utiliser une API spéciale qui fournit les macros et les fonctions dissimulant en grande partie

la complexité liée à la construction de types de données composites. Pour utiliser cette API, le fichier source doit inclure :

```
#include "funcapi.h"
```

Il existe deux façons de construire une valeur de données composites (autrement dit un « tuple ») : vous pouvez le construire à partir d'un tableau de valeurs Datum ou à partir d'un tableau de chaînes C qui peuvent passer dans les fonctions de conversion des types de données du tuple. Quel que soit le cas, vous avez d'abord besoin d'obtenir et de construire un descripteur `TupleDesc` pour la structure du tuple. En travaillant avec des Datums, vous passez le `TupleDesc` à `BlessTupleDesc`, puis vous appelez `heap_form_tuple` pour chaque ligne. En travaillant avec des chaînes C, vous passez `TupleDesc` à `TupleDescGetAttInMetadata`, puis vous appelez `BuildTupleFromCStrings` pour chaque ligne. Dans le cas d'une fonction renvoyant un ensemble de tuple, les étapes de configuration peuvent toutes être entreprises une fois lors du premier appel à la fonction.

Plusieurs fonctions d'aide sont disponibles pour configurer le `TupleDesc` requis. La façon recommandée de le faire dans la plupart des fonctions renvoyant des valeurs composites est d'appeler :

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

en passant la même structure `fcinfo` que celle passée à la fonction appelante (ceci requiert bien sûr que vous utilisez les conventions d'appel version-1). `resultTypeId` peut être spécifié comme `NULL` ou comme l'adresse d'une variable locale pour recevoir l'OID du type de résultat de la fonction. `resultTupleDesc` devrait être l'adresse d'une variable `TupleDesc` locale. Vérifiez que le résultat est `TYPEFUNC_COMPOSITE` ; dans ce cas, `resultTupleDesc` a été rempli avec le `TupleDesc` requis (si ce n'est pas le cas, vous pouvez rapporter une erreur pour une « fonction renvoyant un enregistrement appelé dans un contexte qui ne peut pas accepter ce type enregistrement »).

> **Astuce**
>
> `get_call_result_type` peut résoudre le vrai type du résultat d'une fonction polymorphique ; donc, il est utile pour les fonctions qui renvoient des résultats scalaires polymorphiques, pas seulement les fonctions qui renvoient des types composites. Le résultat `resultTypeId` est principalement utile pour les fonctions renvoyant des scalaires polymorphiques.

> **Note**
>
> `get_call_result_type` a une fonction cousine `get_expr_result_type`, qui peut être utilisée pour résoudre le tupe attendu en sortie en un appel de fonction représenté par un arbre d'expressions. Ceci peut être utilisé pour tenter de déterminer le type de résultat sans entrer dans la fonction elle-même. Il existe aussi `get_func_result_type`, qui peut seulement être utilisée quand l'OID de la fonction est disponible. Néanmoins, ces fonctions ne sont pas capables de gérer les fonctions déclarées renvoyer des enregistrements (`record`). `get_func_result_type` ne peut pas résoudre les types polymorphiques, donc vous devriez utiliser de préférence `get_call_result_type`.

Les fonctions anciennes, et maintenant obsolètes, qui permettent d'obtenir des `TupleDesc` sont :

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

pour obtenir un `TupleDesc` pour le type de ligne d'une relation nommée ou :

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

pour obtenir une `TupleDesc` basée sur l'OID d'un type. Ceci peut être utilisé pour obtenir un `TupleDesc` soit pour un type de base, soit pour un type composite. Néanmoins, cela ne fonctionnera pas pour une fonction qui renvoie `record` et cela ne résoudra pas les types polymorphiques.

Une fois que vous avez un `TupleDesc`, appelez :

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

si vous pensez travailler avec des Datums ou :

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

si vous pensez travailler avec des chaînes C. Si vous écrivez une fonction renvoyant un ensemble, vous pouvez sauvegarder les résultats de ces fonctions dans la structure dans le `FuncCallContext` -- utilisez le champ `tuple_desc` ou `attinmeta` respectivement.

Lorsque vous fonctionnez avec des Datums, utilisez :

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool
 *isnull)
```

pour construire une donnée utilisateur `HeapTuple` indiquée dans le format Datum.

Lorsque vous travaillez avec des chaînes C, utilisez :

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char
 **values)
```

pour construire une donnée utilisateur `HeapTuple` indiquée dans le format des chaînes C. *`values`* est un tableau de chaîne C, une pour chaque attribut de la ligne renvoyée. Chaque chaîne C doit être de la forme attendue par la fonction d'entrée du type de donnée de l'attribut. Afin de renvoyer une valeur NULL pour un des attributs, le pointeur correspondant dans le tableau de valeurs (*`values`*) doit être fixé à `NULL`. Cette fonction demandera à être appelée pour chaque ligne que vous renvoyez.

Une fois que vous avez construit un tuple devant être renvoyé par votre fonction, vous devez le convertir en type `Datum`. Utilisez :

```
HeapTupleGetDatum(HeapTuple tuple)
```

pour convertir un type `HeapTuple` en un Datum valide. Ce `Datum` peut être renvoyé directement si vous envisagez de renvoyer juste une simple ligne ou bien il peut être utilisé pour renvoyer la valeur courante dans une fonction renvoyant un ensemble.

Un exemple figure dans la section suivante.

38.10.8. Renvoi d'ensembles

Les fonctions en langage C ont deux options pour renvoyer des ensembles (plusieurs lignes). Dans la première méthode, appelée mode *ValuePerCall*, une fonction renvoyant un ensemble de lignes est appelée de façon répétée (en passant les mêmes arguments à chaque fois) et elle renvoie une nouvelle

ligne pour chaque appel, jusqu'à ce qu'il n'y ait plus de lignes à renvoyer et qu'elle le signale en renvoyant NULL. La fonction SRF doit de ce fait sauvegarder l'état entre appels pour se rappeler ce qu'elle faisait et renvoyer le bon prochain élément à chaque appel. Dans l'autre méthode, appelée mode *Materialize*, une SRF remplit et renvoie un objet *tuplestore* contenant le résultat entier. Un seul appel survient pour le résultat complet, et il n'est pas nécessaire de conserver l'état entre appels.

Lors de l'utilisation du mode ValuePerCall, il est important de se rappeler que la requête n'est pas garantie de se terminer ; c'est-à-dire, avec des options telles que `LIMIT`, l'exécuteur pourrait stopper les appels à la fonction SRF avant que toutes les lignes ne soient récupérées. Ceci signifie qu'il ne faut pas utiliser le dernier appel pour nettoyer l'activité réalisée, car ce dernier appel pourrait bien ne jamais survenir. Il est recommandé d'utiliser le mode Materialize pour les fonctions ayant besoin d'accéder aux ressources externes, tels que des descripteurs de fichiers.

Le reste de cette section documente un ensemble de macros d'aide qui sont communément utilisées (bien que non requis pour être utilisés) pour les SRF utilisant le mode ValuePerCall. Des détails supplémentaires sur le mode Materialize peuvent être trouvés dans `src/backend/utils/fmgr/README`. De plus, les modules `contrib` dans la distribution des sources de the PostgreSQL contiennent de nombreux exemples de SRF utilisant à la fois les modes ValuePerCall et Materialize.

Pour utiliser les macros de support ValuePerCall décrites ici, inclure `funcapi.h`. Ces macros fonctionnent avec une structure `FuncCallContext` qui contient l'état devant être sauvegardé au travers des appels. À l'intérieur de la SRF appelante, `fcinfo->flinfo->fn_extra` est utilisé pour détenir un pointeur vers `FuncCallContext` entre les appels. Les macros remplissent automatiquement ce champ à la première utilisation et s'attendent à y trouver le même pointeur lors des prochains appels.

```
typedef struct FuncCallContext
{
    /*
     * Number of times we've been called before
     *
     * call_cntr is initialized to 0 for you by
 SRF_FIRSTCALL_INIT(), and
     * incremented for you every time SRF_RETURN_NEXT() is called.
     */
    uint64 call_cntr;

    /*
     * OPTIONAL maximum number of calls
     *
     * max_calls is here for convenience only and setting it is
 optional.
     * If not set, you must provide alternative means to know when
 the
     * function is done.
     */
    uint64 max_calls;

    /*
     * OPTIONAL pointer to miscellaneous user-provided context
 information
     *
     * user_fctx is for use as a pointer to your own data to retain
     * arbitrary context information between calls of your
 function.
     */
    void *user_fctx;

    /*
```

```
     * OPTIONAL pointer to struct containing attribute type input
 metadata
     *
     * attinmeta is for use when returning tuples (i.e., composite
 data types)
     * and is not used when returning base data types. It is only
 needed
     * if you intend to use BuildTupleFromCStrings() to create the
 return
     * tuple.
     */
    AttInMetadata *attinmeta;

    /*
     * memory context used for structures that must live for
 multiple calls
     *
     * multi_call_memory_ctx is set by SRF_FIRSTCALL_INIT() for
 you, and used
     * by SRF_RETURN_DONE() for cleanup. It is the most appropriate
 memory
     * context for any memory that is to be reused across multiple
 calls
     * of the SRF.
     */
    MemoryContext multi_call_memory_ctx;

    /*
    * OPTIONAL pointer to struct containing tuple description
    *
    * tuple_desc is for use when returning tuples (i.e. composite
 data types)
    * and is only needed if you are going to build the tuples with
    * heap_form_tuple() rather than with BuildTupleFromCStrings().
 Note that
    * the TupleDesc pointer stored here should usually have been
 run through
    * BlessTupleDesc() first.
    */
    TupleDesc tuple_desc;

} FuncCallContext;
```

Les macros à utiliser par une SRF utilisant cette infrastructure sont :

```
SRF_IS_FIRSTCALL()
```

Utilisez ceci pour déterminer si votre fonction est appelée pour la première fois ou la prochaine fois. Au premier appel (seulement), utilisez :

```
SRF_FIRSTCALL_INIT()
```

pour initialiser la structure `FuncCallContext`. À chaque appel de fonction, y compris le premier, utilisez :

```
SRF_PERCALL_SETUP()
```

pour configurer l'utilisation de `FuncCallContext`.

Si votre fonction a des données à renvoyer dans l'appel courant, utilisez :

```
SRF_RETURN_NEXT(funcctx, result)
```

pour les renvoyer à l'appelant. (`result` doit être de type `Datum`, soit une valeur simple, soit un tuple préparé comme décrit ci-dessus.) Enfin, quand votre fonction a fini de renvoyer des données, utilisez :

```
SRF_RETURN_DONE(funcctx)
```

pour nettoyer et terminer la SRF.

Lors de l'appel de la SRF, le contexte mémoire courant est un contexte transitoire qui est effacé entre les appels. Cela signifie que vous n'avez pas besoin d'appeler `pfree` sur tout ce que vous avez alloué en utilisant `palloc` ; ce sera supprimé de toute façon. Toutefois, si vous voulez allouer des structures de données devant persister tout au long des appels, vous avez besoin de les conserver quelque part. Le contexte mémoire référencé par `multi_call_memory_ctx` est un endroit approprié pour toute donnée devant survivre jusqu'à l'achèvement de la fonction SRF. Dans la plupart des cas, cela signifie que vous devrez basculer vers `multi_call_memory_ctx` au moment de la préparation du premier appel. Utilisez `funcctx->user_fctx` pour récupérer un pointeur vers de telles structures de données inter-appels (les données que vous allouez dans `multi_call_memory_ctx` partiront automatiquement à la fin de la requête, donc il n'est pas nécessaire de libérer cette donnée manuellement).

> **Avertissement**
>
> Quand les arguments réels de la fonction restent inchangés entre les appels, si vous lisez la valeur des arguments (ce qui se fait de façon transparente par la macro `PG_GETARG_`*`xxx`*) dans le contexte, alors les copies seront libérées sur chaque cycle. De la même façon, si vous conservez des références vers de telles valeurs dans votre `user_fctx`, vous devez soit les copier dans `multi_call_memory_ctx`, soit vous assurer que vous procédez vous-même au traitement des valeurs dans ce contexte.

Voici un exemple complet de pseudo-code :

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;
    further declarations as needed

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx-
>multi_call_memory_ctx);
        /* One-time setup code appears here: */
        user code
        if returning composite
            build TupleDesc, and perhaps AttInMetadata
```

```
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }

    /* Each-time setup code appears here: */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code

    /* this is just one way we might test whether we are done: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* Here we want to return another item: */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* Here we are done returning items, so just report that
 fact. */
        /* (Resist the temptation to put cleanup code here.) */
        SRF_RETURN_DONE(funcctx);
    }
}
```

Et voici un exemple complet d'une simple SRF retournant un type composite :

```
PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    AttInMetadata       *attinmeta;

     /* stuff done only on the first call of the function */
     if (SRF_IS_FIRSTCALL())
     {
        MemoryContext  oldcontext;

        /* create a function context for cross-call persistence */
        funcctx = SRF_FIRSTCALL_INIT();

        /* switch to memory context appropriate for multiple
 function calls */
        oldcontext = MemoryContextSwitchTo(funcctx-
>multi_call_memory_ctx);

        /* total number of tuples to be returned */
        funcctx->max_calls = PG_GETARG_INT32(0);

        /* Build a tuple descriptor for our result type */
```

```
       if (get_call_result_type(fcinfo, NULL, &tupdesc) !=
TYPEFUNC_COMPOSITE)
           ereport(ERROR,
                   (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                    errmsg("function returning record called in
context "
                           "that cannot accept type record")));

       /*
        * generate attribute metadata needed later to produce
tuples from raw
        * C strings
        */
       attinmeta = TupleDescGetAttInMetadata(tupdesc);
       funcctx->attinmeta = attinmeta;

       MemoryContextSwitchTo(oldcontext);
   }

   /* stuff done on every call of the function */
   funcctx = SRF_PERCALL_SETUP();

   call_cntr = funcctx->call_cntr;
   max_calls = funcctx->max_calls;
   attinmeta = funcctx->attinmeta;

   if (call_cntr < max_calls)    /* do when there is more left to
send */
   {
       char       **values;
       HeapTuple    tuple;
       Datum        result;

       /*
 * Prepare a values array for building the returned tuple.
        * This should be an array of C strings which will
        * be processed later by the type input functions.
        */
       values = (char **) palloc(3 * sizeof(char *));
       values[0] = (char *) palloc(16 * sizeof(char));
       values[1] = (char *) palloc(16 * sizeof(char));
       values[2] = (char *) palloc(16 * sizeof(char));

       snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
       snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
       snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

       /* build a tuple */
       tuple = BuildTupleFromCStrings(attinmeta, values);

       /* make the tuple into a datum */
       result = HeapTupleGetDatum(tuple);

       /* clean up (this is not really necessary) */
       pfree(values[0]);
       pfree(values[1]);
       pfree(values[2]);
       pfree(values);
```

```
        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* do when there is no more left */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

Voici une façon de déclarer cette fonction en SQL :

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

Une façon différente de le faire est d'utiliser des paramètres OUT :

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

Notez que dans cette méthode le type en sortie de la fonction est du type `record` anonyme.

38.10.9. Arguments polymorphes et types renvoyés

Les fonctions en langage C peuvent être déclarées pour accepter et renvoyer les types « polymorphes » décrits dans Section 38.2.5. Si les types des arguments ou du renvoi de la fonction sont définis comme polymorphes, l'auteur de la fonction ne peut pas savoir à l'avance quel type de données sera appelé ou bien quel type doit être renvoyé. Il y a deux routines offertes par `fmgr.h` qui permettent à une fonction en version-1 de découvrir les types de données effectifs de ses arguments et le type qu'elle doit renvoyer. Ces routines s'appellent `get_fn_expr_rettype(FmgrInfo *flinfo)` et `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)`. Elles renvoient l'OID du type du résultat ou de l'argument ou InvalidOID si l'information n'est pas disponible. L'accès à la structure `flinfo` se fait normalement avec `fcinfo->flinfo`. Le paramètre `argnum` est basé à partir de zéro. `get_call_result_type` peut aussi être utilisé comme alternative à `get_fn_expr_rettype`. Il existe aussi `get_fn_expr_variadic`, qui peut être utilisé pour trouver les arguments variables en nombre qui ont été assemblés en un tableau. C'est principalement utile dans le cadre des fonctions `VARIADIC "any"` car de tels assemblages surviendront toujours pour les fonctions variadiques prenant des types de tableaux ordinaires.

Par exemple, supposons que nous voulions écrire une fonction qui accepte un argument de n'importe quel type et qui renvoie un tableau uni-dimensionnel de ce type :

```
PG_FUNCTION_INFO_V1(make_array);
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo,
 0);
    Datum       element;
    bool        isnull;
```

```
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* get the provided element, being careful in case it's NULL */
    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);

    /* we have one dimension */
    ndims = 1;
    /* and one element */
    dims[0] = 1;
    /* and lower bound is 1 */
    lbs[0] = 1;

    /* get required info about the element type */
    get_typlenbyvalalign(element_type, &typlen, &typbyval,
&typalign);

    /* now build the array */
    result = construct_md_array(&element, &isnull, ndims, dims,
 lbs,
                                element_type, typlen, typbyval,
 typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

La commande suivante déclare la fonction `make_array` en SQL :

```
CREATE FUNCTION make_array(anyelement)
    RETURNS anyarray
    AS 'DIRECTORY/funcs', 'make_array'
    LANGUAGE 'C' IMMUTABLE;
```

Notez l'utilisation de STRICT ; ceci est primordial car le code ne se préoccupe pas de tester une entrée NULL.

Il existe une variante du polymorphisme qui est seulement disponible pour les fonctions en langage C : elles peuvent être déclarées prendre des paramètres de type `"any"`. (Notez que ce nom de type doit être placé entre des guillemets doubles car il s'agit d'un mot SQL réservé.) Ceci fonctionne comme `anyelement` sauf qu'il ne contraint pas les différents arguments `"any"` à être du même type, pas plus qu'ils n'aident à déterminer le type de résultat de la fonction. Une fonction en langage C peut aussi déclarer son paramètre final ainsi : `VARIADIC "any"`. Cela correspondra à un ou plusieurs arguments réels de tout type (pas nécessairement le même type). Ces arguments ne seront *pas* placés dans un tableau comme c'est le cas pour les fonctions variadic normales ; ils seront passés séparément à la fonction. La macro `PG_NARGS()` et les méthodes décrites ci-dessus doivent être utilisées pour déterminer le nombre d'arguments réels et leur type lors de l'utilisation de cette fonctionnalité. Ainsi,

les utilisateurs d'une telle fonction voudront probablement utilisé le mot-clé `VARIADIC` dans leur appel de fonction, de manière à ce que la fonction traite les éléments du tableau comme des arguments séparés. La fonction elle-même doit implémenter ce comportement si nécessaire, après avoir utilisé `get_fn_expr_variadic` pour savoir si les arguments actuels ont été marqués avec `VARIADIC`.

38.10.10. Mémoire partagée et LWLocks

Les modules peuvent réserver des LWLocks et allouer de la mémoire partagée au lancement du serveur. La bibliothèque partagée du module doit être préchargée en l'ajoutant shared_preload_libraries. La bibliothèque partagée doit enregistrer un `shmem_request_hook` dans sa fonction `_PG_init`. Ce `shmem_request_hook` peut réserver des LWLocks ou de la mémoire partagée. La mémoire partagée est réservée en appelant :

```
void
      RequestAddinShmemSpace(int size)
```

à partir de votre fonction `shmem_request_hook`.

Les LWLocks sont réservés en appelant :

```
void RequestNamedLWLockTranche(const char *tranche_name, int
 num_lwlocks)
```

à partir de `shmem_request_hook`. Ceci assurera qu'un tableau de `num_lwlocks` LWLocks est disponible sous le nom de `tranche_name`. Utilisez `GetNamedLWLockTranche` pour obtenir un pointeur vers ce tableau.

Un exemple de `shmem_request_hook` est disponible dans le fichier `contrib/pg_stat_statements/pg_stat_statements.c` des sources de PostgreSQL.

Pour éviter des cas rares possibles, chaque moteur devrait utiliser la fonction `AddinShmemInitLock` lors de la connexion et de l'initialisation de la mémoire partagée, comme indiquée ci-dessous :

```
        static mystruct *ptr = NULL;

        if (!ptr)
        {
                bool    found;

                LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
                ptr = ShmemInitStruct("my struct name", size,
 &found);
                if (!found)
                {
                        initialize contents of shmem area;
                        acquire any requested LWLocks using:
                        ptr->locks = GetNamedLWLockTranche("my
 tranche name");
                }
                LWLockRelease(AddinShmemInitLock);
        }
```

38.10.11. Coder des extensions en C++

Bien que le moteur PostgreSQL soit écrit en C, il est possible de coder des extensions en C++ si les lignes de conduite suivantes sont respectées :

- Toutes les fonctions accessibles par le serveur doivent présenter une interface en C ; seules ces fonctions C pourront alors appeler du code C++. Ainsi, l'édition de liens `extern C` est nécessaire pour les fonctions appelées par le serveur. Ceci est également obligatoire pour toutes les fonctions passées comme pointeur entre le serveur et du code C++.

- Libérez la mémoire en utilisant la méthode de désallocation appropriée. Par exemple, la majeure partie de la mémoire allouée par le serveur l'est par appel de la fonction `palloc()`, aussi, il convient de libérer ces zones mémoire en utilisant la fonction `pfree()`. L'utilisation de la fonction C++ `delete` échouerait pour ces blocs de mémoire.

- Évitez la propagation d'exceptions dans le code C (utilisez un bloc catch-all au niveau le plus haut de toute fonction `extern C`. Ceci est nécessaire, même si le code C++ n'émet explicitement aucune exception, dans la mesure où la survenue d'événements tels qu'un manque de mémoire peut toujours lancer une exception. Toutes les exceptions devront être gérées et les erreurs correspondantes transmises via l'interface du code C. Si possible, compilez le code C++ avec l'option `-fno-exceptions` afin d'éliminer entièrement la venue d'exceptions ; dans ce cas, vous devrez effectuer vous-même les vérifications correspondantes dans votre code C++, par exemple, vérifier les éventuels paramètres NULL retournés par la fonction `new()`.

- Si vous appelez des fonctions du serveur depuis du code C++, assurez-vous que la pile d'appels ne contienne que des structures C (POD). Ceci est nécessaire dans la mesure où les erreurs au niveau du serveur génèrent un saut via l'instruction `longjmp()` qui ne peut dépiler proprement une pile d'appels C++ comportant des objets non-POD.

Pour résumer, le code C++ doit donc être placé derrière un rempart de fonctions `extern C` qui fourniront l'interface avec le serveur, et devront éviter toute fuite de mécanismes propres au C++ (exceptions, allocation/libération de mémoire et objets non-POD dans la pile).

38.11. Informations d'optimisation de fonction

Par défaut, une fonction est juste une « boîte noire » dont la base de données connait très peu le comportement. De ce fait, il peut arriver que les requêtes utilisant la fonction se trouvent exécutées beaucoup moins efficacement qu'elles ne pourraient. Il est possible de fournir une connaissance supplémentaire qui aide le planificateur à optimiser les appels de fonctions.

Quelques indications basiques peuvent être données via les annotations déclaratives présentes dans la commande `CREATE FUNCTION`. La plus importante de celles-ci est la catégorie de volatilité (`IMMUTABLE`, `STABLE`, ou `VOLATILE`) ; on doit toujours être très soucieux de la spécifier correctement lorsqu'on définit une fonction. La propriété de sûreté face au parallélisme (`PARALLEL UNSAFE`, `PARALLEL RESTRICTED`, ou `PARALLEL SAFE`) doit également être spécifiée pour espérer utiliser la fonction dans des requêtes parallélisées. Il peut aussi être utile de spécifier le coût estimé de l'exécution de la fonction, et le nombre de lignes qu'une fonction renvoyant un ensemble d'enregistrements est estimée renvoyer. Toutefois, la manière déclarative de spécifier ces deux informations ne permet seulement que de donner une valeur constante, ce qui est souvent inadéquat.

Il est aussi possible de rattacher une *fonction de support de planification* à une fonction appelable en SQL (appelée *fonction cible*), et ainsi de fournir sur la fonction cible une connaissance qui serait trop complexe à être représentée déclarativement. Les fonctions de support de planification doivent être écrites en C (alors que leurs fonctions cibles peuvent ne pas l'être), ce qui en fait une fonctionnalité avancée que relativement peu de personnes utiliseront.

Une fonction de support de planification doit avoir la signature SQL

```
supportfn(internal) returns internal
```

Elle est rattachée à sa fonction cible en spécifiant la clause `SUPPORT` dans la création de la fonction cible.

Les détails de l'API des fonctions de support de planification se trouvent dans le fichier `src/include/nodes/supportnodes.h` dans le code source de PostgreSQL. Ici, on ne fournit qu'une vue d'ensemble de ce que les fonctions de support de planification peuvent faire. L'ensemble des demandes possibles adressables à une fonction de support est extensible, si bien que d'autres choses seront possibles dans des versions ultérieures.

Certains appels de fonctions peuvent être simplifiés lors de la planification en se basant sur les propriétés spécifiques de la fonction. Par exemple, `int4mul(n, 1)` pourrait être simplifié par `n`. Ce type de transformation peut être exécuté par une fonction de support de planification, en lui faisant implémenter le type de demande `SupportRequestSimplify`. La fonction de support va être appelée pour chaque instance de la fonction cible trouvée dans l'arbre d'analyse de la requête. Si elle trouve qu'un appel particulier peut être simplifié en une autre forme, elle est capable de construire et renvoyer un arbre d'analyse représentant cette expression. Cela fonctionnera automatiquement pour les opérateurs basés sur cette fonction, également -- dans l'exemple juste ci-dessus, `n * 1` serait aussi simplifié en `n`. (Mais notez que c'est juste un exemple ; cette optimisation particulière n'est pas réellement mise en œuvre par PostgreSQL.) Nous ne garantissons pas que PostgreSQL n'appelera jamais la fonction cible dans les cas que la fonction de support pourrait simplifier. Assurez-vous d'une équivalence rigoureuse entre l'expression simplifiée et l'exécution réelle de la fonction cible.

Pour les fonctions cible qui renvoient un booléen, il est souvent utile d'estimer la fraction des lignes qui vont être sélectionnées par une clause `WHERE` utilisant cette fonction. Ceci est réalisable avec une fonction de support qui implémente le type de demande `SupportRequestSelectivity`.

Si le temps d'exécution d'une fonction cible est très dépendant de ses entrées, il peut être utile de fournir un coût d'exécution non constant pour celle-ci. Ceci est réalisable avec une fonction de support implémentant le type de demande `SupportRequestCost`.

Pour les fonctions cibles qui renvoient des ensembles de lignes, il est souvent utile de fournir une estimation non constante du nombre de lignes renvoyées. Ceci est réalisable avec une fonction de support implémentant le type de demande `SupportRequestRows`.

Pour les fonctions cibles qui renvoient un booléen, il est envisageable de convertir un appel de fonction au niveau d'un `WHERE` vers une ou plusieurs clauses d'un opérateur indexable. Ces clauses peuvent être exactement équivalentes à la condition de la fonction, ou bien elles peuvent être plus faibles (c'est-à-dire qu'elles peuvent accepter certaines valeurs que la condition via la fonction n'accepte pas). Dans ce dernier cas, la condition d'index est dite *avec perte*; elle peut toujours être utilisée pour un parcours d'index, mais la fonction devra être appelée pour chaque ligne renvoyée par l'index pour vérifier qu'elle satisfait la condition `WHERE`. Pour créer de telles conditions, la fonction de support doit implémenter le type de demande `SupportRequestIndexCondition`.

38.12. Agrégats utilisateur

Dans PostgreSQL, les fonctions d'agrégat sont exprimées comme des *valeurs d'état* et des *fonctions de transition d'état*. C'est-à-dire qu'un agrégat opère en utilisant une valeur d'état qui est mis à jour à chaque ligne traitée. Pour définir une nouvelle fonction d'agrégat, on choisit un type de donnée pour la valeur d'état, une valeur initiale pour l'état et une fonction de transition d'état. La fonction de transition d'état prend la valeur d'état précédente et les valeurs en entrée de l'agrégat pour la ligne courante, et renvoie une nouvelle valeur d'état. Une *fonction finale* peut également être spécifiée pour le cas où le résultat désiré comme agrégat est différent des données conservées comme valeur d'état courant. La fonction finale prend la dernière valeur de l'état, et renvoie ce qui est voulu comme résultat de l'agrégat. En principe, les fonctions de transition et finale sont des fonctions ordinaires qui pourraient aussi être utilisées en dehors du contexte de l'agrégat. (En pratique, il est souvent utile pour des raisons

de performance de créer des fonctions de transition spécialisées qui ne peuvent fonctionner que quand elles sont appelées via l'agrégat.)

Ainsi, en plus des types de données d'argument et de résultat vus par l'utilisateur, il existe un type de données pour la valeur d'état interne qui peut être différent des deux autres.

Un agrégat qui n'utilise pas de fonction finale est un agrégat qui utilise pour chaque ligne une fonction dépendante des valeurs de colonnes. `sum` en est un exemple. `sum` débute à zéro et ajoute la valeur de la ligne courante à son total en cours. Par exemple, pour obtenir un agrégat `sum` qui opère sur un type de données de nombres complexes, il suffira de décrire la fonction d'addition pour ce type de donnée. La définition de l'agrégat sera :

```
CREATE AGGREGATE somme (complex)
(
    sfunc = ajout_complexe,
    stype = complexe,
    initcond = '(0,0)'
);
```

que nous pourrions utiliser ainsi :

```
SELECT somme(a) FROM test_complexe;

   somme
-----------
 (34,53.9)
```

(Notez que nous nous reposons sur une surcharge de fonction : il existe plus d'un agrégat nommé `sum` mais PostgreSQL trouve le type de somme s'appliquant à une colonne de type `complex`.)

La définition précédente de `sum` retournera zéro (la condition d'état initial) s'il n'y a que des valeurs d'entrée NULL. Dans ce cas, on peut souhaiter qu'elle retourne NULL -- le standard SQL prévoit que la fonction `sum` se comporte ainsi. Cela peut être obtenu par l'omission de l'instruction `initcond`, de sorte que la condition d'état initial soit NULL. Dans ce cas, `sfunc` vérifie l'entrée d'une condition d'état NULL mais, pour `sum` et quelques autres agrégats simples comme `max` et `min`, il suffit d'insérer la première valeur d'entrée non NULL dans la variable d'état et d'appliquer la fonction de transition d'état à partir de la seconde valeur non NULL. PostgreSQL fait cela automatiquement si la condition initiale est NULL et si la fonction de transition est marquée « strict » (elle n'est pas appelée pour les entrées NULL).

Par défaut également, pour les fonctions de transition « strict », la valeur d'état précédente reste inchangée pour une entrée NULL. Les valeurs NULL sont ainsi ignorées. Pour obtenir un autre comportement, il suffit de ne pas déclarer la fonction de transition « strict ». À la place, codez-la de façon à ce qu'elle vérifie et traite les entrées NULL.

`avg` (average = moyenne) est un exemple plus complexe d'agrégat. Il demande deux états courants : la somme des entrées et le nombre d'entrées. Le résultat final est obtenu en divisant ces quantités. La moyenne est typiquement implémentée en utilisant comme valeur d'état un tableau. Par exemple, l'implémentation intégrée de `avg(float8)` ressemble à :

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

> **Note**
>
> `float8_accum` nécessite un tableau à trois éléments, et non pas seulement deux, car il accumule la somme des carrés, ainsi que la somme et le nombre des entrées. Cela permet son utilisation pour d'autres agrégats que `avg`.

Les appels de fonctions d'agrégat en SQL autorisent les options `DISTINCT` et `ORDER BY` qui contrôlent les lignes envoyées à la fonction de transition de l'agrégat et leur ordre. Ces options sont implémentées en arrière-plan et ne concernent pas les fonctions de support de l'agrégat.

Pour plus de détails, voir la commande CREATE AGGREGATE.

38.12.1. Mode d'agrégat en déplacement

Les fonctions d'agrégat peuvent accepter en option un *mode d'agrégat en déplacement*, qui autorise une exécution bien plus rapide des fonctions d'agrégats pour les fenêtres dont le point de démarrage se déplace. (Voir Section 3.5 et Section 4.2.8 pour des informations sur l'utilisation des fonctions d'agrégats en tant que fonctions de fenêtrage.) L'idée de base est qu'en plus d'une fonction de transition « en avant », l'agrégat doit fournir une *fonction de transition inverse*, qui permet aux lignes d'être supprimées de la valeur d'état de l'agrégat quand elles quittent l'étendue de la fenêtre. Par exemple, un agrégat `sum` qui utilise l'addition comme fonction de transition en avant pourrait utiliser la soustraction comme fonction de transition inverse. Sans fonction de transition inverse, le mécanisme de fonction de fenêtrage doit recalculer l'agrégat à partir du début à chaque fois que le point de départ de la fenêtre est déplacé, ce qui a pour effet d'augmenter la durée d'exécution proportionnellement au nombre de lignes en entrée multiplié par la longueur moyenne de la fenêtre. Avec une fonction de transition inverse, la durée d'exécution est uniquement proportionnelle au nombre de lignes en entrée.

La fonction de transition inverse se voit fournie la valeur de l'état courant et les valeurs en entrée de l'agrégat pour la première ligne inclus dans l'état courant. Il doit reconstruire la valeur d'état telle qu'elle aurait été si la ligne en entrée n'avait pas été agrégé, mais seulement les lignes suivantes. Ceci demande parfois que la fonction de transition en avant conserve plus d'informations sur l'état que ce qui était nécessaire auparavant. De ce fait, le mode d'agrégat en déplacement utilise une implémentation complètement séparée du mode standard : il a son propre type de données d'état, sa propre fonction de transition en avant, et sa propre fonction finale si nécessaire. Ils peuvent être les mêmes que le type de données et les fonctions du mode standard si rien de particulier n'est nécessaire.

Comme exemple, nous pouvons étendre l'agrégat `sum` donné ci-dessus pour supporter le mode d'agrégat en déplacement, comme ceci :

```
CREATE AGGREGATE somme (complex)
(
    sfunc = ajout_complexe,
    stype = complexe,
    initcond = '(0,0)',
    msfunc = ajout_complexe,
    minvfunc = retire_complexe,
    mstype = complexe,
    minitcond = '(0,0)'
);
```

Les paramètres dont les noms commencent par un `m` définissent l'implémentation des agrégats en mouvement. En dehors de la fonction de transition inverse `minvfunc`, ils correspondent aux paramètres des agrégats standards sans `m`.

La fonction de transition en avant pour le mode d'agrégat en déplacement n'est pas autorisée à renvoyer NULL comme nouvelle valeur d'état. Si la fonction de transition inverse renvoie NULL, c'est pris comme indication que la fonction ne peut pas inverser le calcul de l'état sur ce point particulier, et donc le calcul d'agrégat sera refait à partir de rien pour la position de début actuelle. Cette convention permet au mode d'agrégat par déplacement à être utilisé dans des situations où il existe quelques cas rares où réaliser l'inverse de la fonction de transition n'est pas possible. La fonction de transition inverse peut ne pas fonctionner sur ces cas, et être toujours utilisée pour la plupart des cas où elle est fonctionnelle. Comme exemple, un agrégat travaillant avec des nombres à virgules flottantes pourrait choisir de ne pas fonctionner quand une entrée `NaN` doit être supprimée de la valeur d'état en cours.

Lors de l'écriture des fonctions de support d'un agrégat en déplacement, il est important d'être certain que la fonction de transition inverse peut reconstruire exactement la valeur d'état correct. Sinon, il pourrait y avoir des différences visibles pour l'utilisateur dans les résultats, suivant que le mode d'agrégat en déplacement est utilisé ou pas. Un exemple d'agrégat pour lequel ajouter une fonction de transition inverse semble facile au premier coup d'œil, mais où les prérequis ne peuvent pas être assurés, est la fonction is `sum` sur des entrées de type `float4` ou `float8`. Une déclaration naïve de `sum(float8)` pourrait être :

```
CREATE AGGREGATE unsafe_sum (float8)
(
    stype = float8,
    sfunc = float8pl,
    mstype = float8,
    msfunc = float8pl,
    minvfunc = float8mi
);
```

Cependant, cet agrégat peut renvoyer des résultats très différents qu'il ne l'aurait fait sans fonction de transition inverse. Considérez par exemple :

```
SELECT
  unsafe_sum(x) OVER (ORDER BY n ROWS BETWEEN CURRENT ROW AND 1
 FOLLOWING)
FROM (VALUES (1, 1.0e20::float8),
             (2, 1.0::float8)) AS v (n,x);
```

Cette requête renvoie `0` en deuxième résultat, plutôt que la réponse attendue, `1`. La raison vient de la précision limitée des valeurs en virgule flottante : ajouter `1` à `1e20` renvoie de nouveau `1e20`, alors qu'y soustraire `1e20` renvoie `0`, et non pas `1`. Notez que c'est une limitation générale des opérations de calcul sur des nombres en virgule flottante, pas une limitation spécifique de PostgreSQL.

38.12.2. Agrégats polymorphiques et variadiques

Les fonctions d'agrégat peuvent utiliser des fonctions d'état transitionnelles ou des fonctions finales polymorphes. De cette façon, les mêmes fonctions peuvent être utilisées pour de multiples agrégats. Voir la Section 38.2.5 pour une explication des fonctions polymorphes. La fonction d'agrégat elle-même peut être spécifiée avec un type de base et des types d'état polymorphes, ce qui permet ainsi à une unique définition de fonction de servir pour de multiples types de données en entrée. Voici un exemple d'agrégat polymorphe :

```
CREATE AGGREGATE array_accum (anycompatible)
(
    sfunc = array_append,
    stype = anycompatiblearray,
    initcond = '{}'
```

```
);
```

Dans ce cas, le type d'état effectif pour tout appel d'agrégat est le type tableau avec comme éléments le type effectif d'entrée. Le comportement de l'agrégat est de concaténer toutes les entrées dans un tableau de ce type. (Note : l'agrégat `array_agg` fournit une fonctionnalité similaire, avec de meilleures performances que ne pourrait avoir cette définition.)

Voici le résultat pour deux types de données différents en arguments :

```
SELECT attrelid::regclass, array_accum(attname)
FROM pg_attribute WHERE attnum > 0
AND attrelid = 'pg_tablespace'::regclass GROUP BY attrelid;
   attrelid    |              array_accum
---------------+---------------------------------------
 pg_tablespace | {spcname,spcowner,spcacl,spcoptions}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid::regtype)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |        array_accum
---------------+---------------------------
 pg_tablespace | {name,oid,aclitem[],text[]}
(1 row)
```

D'habitude, une fonction d'agrégat avec un type de résultat polymorphique a un type d'état polymorphique, comme dans l'exemple ci-dessus. C'est nécessaire, sinon la fonction finale ne peut pas être déclarée correctement. Elle devrait avoir un type de résultat polymorphique mais pas d'argument polymorphique, ce que `CREATE FUNCTION` rejetera sur la base que le type en résultat ne peut pas être déduit de cet appel. Parfois, il est inconfortable d'utiliser un type d'état polymorphique. Le cas le plus fréquent arrive quand les fonctions de support de l'agrégat sont à écrire en C et que le type d'état doit être déclaré comme `internal` parce qu'il n'existe pas d'équivalent SQL pour lui. Dans ce cas, il est possible de déclarer la fonction finale comme prenant des arguments « inutiles » qui correspondent aux arguments en entrée de l'agrégat. Ce type d'argument est toujours passé avec une valeur NULL car aucune valeur spécifique n'est disponible quand la fonction finale est appelée. Leur seule utilisée est de permettre à un type de résultat d'une fonction finale polymorphique d'être connecté au type de données en entrée de l'agrégat. Par exemple, la définition de l'agrégat interne `array_agg` est équivalent à :

```
CREATE FUNCTION array_agg_transfn(internal, anynonarray)
  RETURNS internal ...;
CREATE FUNCTION array_agg_finalfn(internal, anynonarray)
  RETURNS anyarray ...;

CREATE AGGREGATE array_agg (anynonarray)
(
    sfunc = array_agg_transfn,
    stype = internal,
    finalfunc = array_agg_finalfn,
    finalfunc_extra
);
```

Dans cet exemple, l'option `finalfunc_extra` spécifie que la fonction finale reçoit, en plus de la valeur d'état, tout argument supplémentaire correspondant aux arguments en entrée de l'agrégat.

L'argument supplémentaire `anynonarray` permet que la déclaration de `array_agg_finalfn` soit valide.

Il est possible de créer une fonction d'agrégat qui accepte un nombre variable d'arguments en déclarant ses derniers arguments dans un tableau `VARIADIC`, un peu de la même façon que les fonctions standards ; voir Section 38.5.6. La fonction de transition de l'agrégat doit avoir le même type tableau que leur dernier argument. Les fonctions de transition seront typiquement marquées comme `VARIADIC`, mais cela n'est pas requis.

> **Note**
>
> Les agrégats variadiques sont facilement mal utilisés avec l'option `ORDER BY` (voir Section 4.2.7), car l'analyseur ne peut pas dire si le nombre d'arguments réels donnés était bon ou pas. Gardez à l'esprit que toutes les expressions à droite de `ORDER BY` sont la clé de tri, pas un argument de l'agrégat. Par exemple, dans :
>
> ```
> SELECT mon_agregat(a ORDER BY a, b, c) FROM ...
> ```
>
> l'analyseur verra cela comme un seul argument pour la fonction d'agrégat, et trois clés de tri. Alors que l'utilisateur pouvait vouloir dire :
>
> ```
> SELECT myaggregate(a, b, c ORDER BY a) FROM ...
> ```
>
> Si `mon_agregat` est variadique, ces deux appels peuvent être parfaitement valides.
>
> Pour la même raison, il est conseillé d'y réfléchir à deux fois avant de créer des fonctions d'agrégat avec les mêmes noms et différents nombres d'arguments standards.

38.12.3. Agrégats d'ensemble trié

Les agrégats que nous avons décrit jusqu'à maintenant sont des agrégats « normaux ». PostgreSQL accepte aussi les *agrégats d'ensemble trié*, qui diffèrent des agrégats normaux de deux façons. Tout d'abord, en plus des arguments standards d'agrégats qui sont évalués une fois par ligne en entrée, un agrégat d'ensemble trié peut avoir des arguments « directs » qui sont évalués seulement une fois par opération d'agrégation. Ensuite, la syntaxe pour les arguments standards agrégés indique un ordre de tri explicitement pour eux. Un agrégat d'ensemble de tri est habituellement utilisé pour ajouter un calcul dépendant d'un ordre spécifique des lignes, par exemple le rang ou le centile. Par exemple, la définition interne de `percentile_disc` est équivalent à :

```
CREATE FUNCTION ordered_set_transition(internal, anyelement)
  RETURNS internal ...;
CREATE FUNCTION percentile_disc_final(internal, float8, anyelement)
  RETURNS anyelement ...;

CREATE AGGREGATE percentile_disc (float8 ORDER BY anyelement)
(
    sfunc = ordered_set_transition,
    stype = internal,
    finalfunc = percentile_disc_final,
    finalfunc_extra
);
```

Cet agrégat prend un argument direct `float8` (la fraction du percentile) et une entrée agrégée qui peut être de toute type de données triées. Il pourrait être utilisé pour obtenir le revenu médian des ménages comme ceci :

```
SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY revenu) FROM
 menages;
 percentile_disc
-----------------
           50489
```

Ici, `0.5` est un argument direct ; cela n'aurait pas de sens que la fraction de centile soit une valeur variant suivant les lignes.

Contrairement aux agrégats normaux, le tri des lignes en entrée pour un agrégat d'ensemble trié n'est *pas* fait de façon caché mais est la responsabilité des fonctions de support de l'agrégat. L'approche typique de l'implémentation est de conserver une référence à l'objet « tuplesort » dans la valeur d'état de l'agrégat, d'alimenter cet objet par les lignes en entrée, et de terminer le tri et de lire les données dans la fonction finale. Ce design permet à la fonction finale de réaliser des opérations spéciales comme l'injection de lignes supplémentaires « hypothétiques » dans les données à trier. Alors que les agrégats normaux peuvent souvent être implémentés avec les fonctions de support écrites en PL/pgSQL ou dans un autre langage PL, les agrégats d'ensemble trié doivent généralement être écrit en C car leurs valeurs d'état ne sont pas définissables sous la forme de type de données SQL. (Dans l'exemple ci-dessus, notez que la valeur d'état est déclarée en tant que `internal` -- c'est typique.) De plus, comme la fonction finale réalise le tri, il n'est pas possible de continuer à ajouter des lignes en entrée en exécutant de nouveau la fonction de transition. Ceci signifie que la fonction finale n'est pas `READ_ONLY` ; elle doit être exécutée dans `CREATE AGGREGATE` en `READ_WRITE` ou en `SHAREABLE` s'il est possible que des appels supplémentaires à la fonction finale utilisent l'état déjà triée.

La fonction de transition d'état pour un agrégat d'ensemble trié reçoit la valeur d'état courante ainsi que les valeurs agrégées en entrée pour chaque ligne. Elle renvoie la valeur d'état mise à jour. Il s'agit de la même définition que pour les agrégats normaux mais notez que les arguments directs (si présents) ne sont pas fournis. La fonction finale reçoit la valeur du dernier état, les valeurs des arguments directs si présents et (si `finalfunc_extra` est indiqué) des valeurs NULL correspondant aux entrées agrégées. Comme avec les agrégats normaux, `finalfunc_extra` est seulement réellement utile si l'agrégat est polymorphique ; alors les arguments inutiles supplémentaires sont nécessaires pour connecter le type de résultat de la fonction finale au type de l'entrée de l'agrégat.

Actuellement, les agrégats d'ensemble trié ne peuvent pas être utilisé comme fonctions de fenêtrage, et du coup, il n'est pas nécessaire qu'ils supportent le mode d'agrégat en déplacement.

38.12.4. Agrégation partielle

En option, une fonction d'agrégat peut supporter une *agrégation partielle*. L'idée d'agrégation partielle est d'exécuter la fonction de transition d'état de l'agrégat sur différents sous-ensembles des données en entrée de façon indépendante, puis de combiner les valeurs d'état provenant de ces sous-ensembles pour produire la même valeur d'état que ce qui aurait résultat du parcours de toutes les entrées en une seule opération. Ce mode peut être utilisé pour l'agrégation parallèle en ayant différents processus parallèles parcourant des portions différentes d'une table. Chaque processus produit une valeur d'état partiel et, à la fin, ces valeurs d'état sont combinées pour produire une valeur d'état finale. (Dans le futur, ce mode pourrait aussi être utilisé dans d'autres cas comme l'agrégation combinée sur des tables locales et externes ; mais ce n'est pas encore implémenté.)

Pour supporter une agrégation partielle, la définition de l'agrégat doit fournir une *fonction de combinaison*, qui prend deux valeurs du type de l'état d'agrégat (représentant les résultats de l'agrégat sur deux sous-ensembles de lignes en entrée) et produit une nouvelle valeur du type de l'état, représentant l'état qu'on aurait eu en réalisant l'agrégat sur la combinaison de ces deux ensembles de données. L'ordre relatif des lignes entrées n'est pas spécifié pour les deux ensembles de données.

Ceci signifie qu'il est habituellement impossible de définir une fonction de combinaison utile pour les agrégats sensibles à l'ordre des lignes en entrée.

Comme exemples simples, les agrégats `MAX` et `MIN` peuvent supporter l'agrégation partielle en indiquant la fonction de combinaison comme étant la même fonction plus- grand-que ou plus-petit-que que celle utilisée comme fonction de transition function. L'agrégat `SUM` a besoin d'une fonction supplémentaire comme fonction de combinaison. (Encore une fois, c'est la même que leur fonction de transition, sauf si la valeur d'état est plus grand que le type de données en entrée.)

La fonction de combinaison est traitée un peu comme une fonction de transition qui prend une valeur du type d'état, pas de celle du type d'entrée sous-jacent, comme deuxième argument. En particulier, les règles pour gérer les valeurs nulles et les fonctions strictes sont similaires. De plus, si la définition de l'agrégat indique un `initcond` non nul, gardez en tête que ce sera utilisé non seulement comme état initial pour chaque exécution de l'agrégat partiel, mais aussi comme état initiale de la fonction de combinaison, qui sera appelée pour combiner chaque résultat partiel dans cet état.

Si le type d'état de l'agrégat est déclaré comme `internal`, il est de la responsabilité de la fonction de combinaison que son résultat soit alloué dans le contexte mémoire correct pour les valeurs d'état de l'agrégat. Ceci signifie en particulier que, quand la première entrée est `NULL`, il est invalide de renvoyer simplement la deuxième entrée car cette valeur sera dans le mauvais contexte et n'aura pas une durée de vie suffisante.

Quand le type d'état de l'agrégat est déclaré comme `internal`, il est aussi habituellement approprié que la définition de l'agrégat fournisse une *fonction de sérialisation* et une *fonction de désérialisation*, qui permet qu'une telle valeur d'état soit copiée d'un processus à un autre. Sans ces fonctions, l'agrégation parallèle ne peut pas être réalisée, et les applications futures telles que l'agrégation locale/distante ne fonctionnera probablement pas non plus.

Une fonction de sérialisation doit prendre un seul argument de type `internal` et renvoyer un résultat de type `bytea`, qui représente la valeur d'état packagé en un paquet plat d'octets. De la même façon, une fonction de désérialisation inverse cette conversion. Elle doit prendre deux arguments de type `bytea` et `internal`, et renvoyer un résultat de type `internal`. (Le deuxième argument n'est pas utilisé et vaut toujours zéro, mais il est requis pour des raisons de sécurité du type.) Le résultat de la fonction de désérialisation doit simplement être alloué dans le contexte mémoire courant car, contrairement au résultat de la fonction de combinaison, il ne vit pas longtemps.

Il est bon de noter aussi que, pour qu'un agrégat soit exécuté en parallèle, l'agrégat lui-même doit être marqué `PARALLEL SAFE`. Les marques de parallélisation sur les fonctions de support ne sont pas consultées.

38.12.5. Fonctions de support pour les agrégats

Une fonction écrite en C peut détecter si elle est appelée en tant que fonction de support d'un agrégat en appelant `AggCheckCallContext`, par exemple :

```
if (AggCheckCallContext(fcinfo, NULL))
```

Une raison de surveiller ceci est que, si le retour de cette fonction vaut true, la première valeur doit être une valeur de transition temporaire et peut du coup être modifiée en toute sûreté sans avoir à allouer une nouvelle copie. Voir `int8inc()` pour un exemple. (Alors que les fonctions de transition des agrégats sont toujours autorisées à modifier en ligne la valeur de transition, les fonctions finales des agrégats ne sont généralement pas encouragées à le faire ; si elles le font, le comportement doit être déclaré lors de la création de l'agrégat. Voir CREATE AGGREGATE pour plus de détails.)

Le deuxième argument de `AggCheckCallContext` peut être utilisé pour récupérer le contexte mémoire dans lequel les valeurs d'état de l'agrégat sont conservées. Ceci est utile pour que les fonctions de transition qui souhaitent utiliser les objets « étendus » (voir Section 38.13.1) comme leurs valeurs d'état. Au premier appel, la fonction de transition doit renvoyer un objet étendu dont le contexte

mémoire est un enfant du contexte d'état de l'agrégat. Puis, pour les appels suivants, il doit renvoyer le même objet étendu. Voir `array_append()` pour un exemple. (`array_append()` n'est pas la fonction de transition d'un agrégat interne mais il est écrit pour se comporter efficacement lorsqu'elle est utilisée comme fonction de transition d'un agrégat personnalisé.)

Une autre routine de support disponible pour les fonctions d'agrégat écrites en langage C est `AggGetAggref`, qui renvoie le nœud d'analyse `Aggref` qui définit l'appel d'agrégat. Ceci est particulièrement utile pour les agrégats d'ensemble trié, qui peuvent inspecter la sous-structure du nœud `Aggref` pour trouver l'ordre de tri qu'elles sont supposées implémenter. Des exemples sont disponibles dans le fichier `orderedsetaggs.c` du code source de PostgreSQL.

38.13. Types utilisateur

Comme cela est décrit dans la Section 38.2, PostgreSQL peut être étendu pour supporter de nouveaux types de données. Cette section décrit la définition de nouveaux types basiques. Ces types de données sont définis en-dessous du SQL. Créer un nouveau type requiert d'implanter des fonctions dans un langage de bas niveau, généralement le C.

Les exemples de cette section sont disponibles dans `complex.sql` et `complex.c` du répertoire `src/tutorial` de la distribution. Voir le fichier `README` de ce répertoire pour les instructions d'exécution des exemples.

Un type utilisateur doit toujours posséder des fonctions d'entrée et de sortie. Ces fonctions déterminent la présentation du type en chaînes de caractères (pour la saisie par l'utilisateur et le renvoi à l'utilisateur) et son organisation en mémoire. La fonction d'entrée prend comme argument une chaîne de caractères terminée par NULL et retourne la représentation interne (en mémoire) du type. La fonction de sortie prend en argument la représentation interne du type et retourne une chaîne de caractères terminée par NULL.

Il est possible de faire plus que stocker un type, mais il faut pour cela implanter des fonctions supplémentaires gérant les opérations souhaitées.

Soit le cas d'un type `complex` représentant les nombres complexes. Une façon naturelle de représenter un nombre complexe en mémoire passe par la structure C suivante :

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

Ce type ne pouvant tenir sur une simple valeur `Datum`, il sera passé par référence.

La représentation externe du type se fera sous la forme de la chaîne `(x,y)`.

En général, les fonctions d'entrée et de sortie ne sont pas compliquées à écrire, particulièrement la fonction de sortie. Mais lors de la définition de la représentation externe du type par une chaîne de caractères, il faudra peut-être écrire un analyseur complet et robuste, comme fonction d'entrée, pour cette représentation. Par exemple :

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;
```

```
    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for type: \"%s\"",
                        str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

La fonction de sortie peut s'écrire simplement :

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = psprintf("(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

Il est particulièrement important de veiller à ce que les fonctions d'entrée et de sortie soient bien inversées l'une par rapport à l'autre. Dans le cas contraire, de grosses difficultés pourraient apparaître lors de la sauvegarde de la base dans un fichier en vue d'une future relecture de ce fichier. Ceci est un problème particulièrement fréquent lorsque des nombres à virgule flottante entrent en jeu.

De manière optionnelle, un type utilisateur peut fournir des routines d'entrée et de sortie binaires. Les entrées/sorties binaires sont normalement plus rapides mais moins portables que les entrées/sorties textuelles. Comme avec les entrées/sorties textuelles, c'est l'utilisateur qui définit précisément la représentation binaire externe. La plupart des types de données intégrés tentent de fournir une représentation binaire indépendante de la machine. Dans le cas du type `complex`, des convertisseurs d'entrées/sorties binaires pour le type `float8` sont utilisés :

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;
```

```
    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

Lorsque les fonctions d'entrée/sortie sont écrites et compilées en une bibliothèque partagée, le type `complex` peut être défini en SQL. Tout d'abord, il est déclaré comme un type shell :

```
CREATE TYPE complex;
```

Ceci sert de paramètre qui permet de mettre en référence le type pendant la définition de ses fonctions E/S. Les fonctions E/S peuvent alors être définies :

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
   RETURNS bytea
   AS 'filename'
   LANGUAGE C IMMUTABLE STRICT;
```

La définition du type de données peut ensuite être fournie complètement :

```
CREATE TYPE complex (
   internallength = 16,
   input = complex_in,
   output = complex_out,
   receive = complex_recv,
   send = complex_send,
   alignment = double
);
```

Quand un nouveau type de base est défini, PostgreSQL fournit automatiquement le support pour des tableaux de ce type. Le type tableau a habituellement le nom du type de base préfixé par un caractère souligné (_).

Lorsque le type de données existe, il est possible de déclarer les fonctions supplémentaires de définition des opérations utiles pour ce type. Les opérateurs peuvent alors être définis par dessus ces fonctions et, si nécessaire, des classes d'opérateurs peuvent être créées pour le support de l'indexage du type de données. Ces couches supplémentaires sont discutées dans les sections suivantes.

Si la représentation interne du type de données est de longueur variable, la représentation interne doit poursuivre l'organisation standard pour une donnée de longueur variable : les quatre premiers octets

doivent être un champ `char[4]` qui n'est jamais accédé directement (nommé `vl_len_`). Vous devez utiliser la macro `SET_VARSIZE()` pour enregistrer la taille totale de la donnée (ceci incluant le champ de longueur lui-même) dans ce champ et `VARSIZE()` pour la récupérer. (Ces macros existent parce que le champ de longueur pourrait être encodé suivant la plateforme.)

Pour plus de détails, voir la description de la commande CREATE TYPE.

38.13.1. Considérations sur les TOAST

Si les valeurs du type de données varient en taille (sous la forme interne), il est généralement préférable que le type de données soit marqué comme TOAST-able (voir Section 73.2). Vous devez le faire même si les données sont trop petites pour être compressées ou stockées en externe car TOAST peut aussi gagner de la place sur des petites données en réduisant la surcharge de l'en-tête.

Pour supporter un stockage TOAST, les fonctions C opérant sur le type de données doivent toujours faire très attention à déballer les valeurs dans le TOAST qui leur sont données par `PG_DETOAST_DATUM`. (Ce détail est généralement caché en définissant les macros `GETARG_DATATYPE_P` spécifiques au type.) Puis, lors de l'exécution de la commande `CREATE TYPE`, indiquez la longueur interne comme `variable` et sélectionnez certaines options de stockage spécifiques autres que `plain`.

Si l'alignement n'est pas important (soit seulement pour une fonction spécifique soit parce que le type de données spécifie un alignement par octet), alors il est possible d'éviter `PG_DETOAST_DATUM`. Vous pouvez utiliser `PG_DETOAST_DATUM_PACKED` à la place (habituellement caché par une macro `GETARG_DATATYPE_PP`) et utiliser les macros `VARSIZE_ANY_EXHDR` et `VARDATA_ANY` pour accéder à un datum potentiellement packagé. Encore une fois, les données renvoyées par ces macros ne sont pas alignées même si la définition du type de données indique un alignement. Si l'alignement est important pour vous, vous devez passer par l'interface habituelle, `PG_DETOAST_DATUM`.

> **Note**
>
> Un ancien code déclare fréquemment `vl_len_` comme un champ de type `int32` au lieu de `char[4]`. C'est correct tant que la définition de la structure a d'autres champs qui ont au moins un alignement `int32`. Mais il est dangereux d'utiliser une telle définition de structure en travaillant avec un datum potentiellement mal aligné ; le compilateur peut le prendre comme une indication pour supposer que le datum est en fait aligné, ceci amenant des « core dump » sur des architectures qui sont strictes sur l'alignement.

Une autre fonctionnalité, activée par le support des TOAST est la possibilité d'avoir une représentation des données *étendue* en mémoire qui est plus agréable à utiliser que le format enregistré sur disque. Le format de stockage varlena standard ou plat (« flat ») est en fait juste un ensemble d'octets ; par exemple, il ne peut pas contenir de pointeurs car il pourrait être copié à d'autres emplacements en mémoire. Pour les types de données complexes, le format plat pourrait être assez coûteux à utiliser, donc PostgreSQL fournit une façon d'« étendre » le format plat en une représentation qui est plus confortable à utiliser, puis passe ce format en mémoire entre les fonctions du type de données.

Pour utiliser le stockage étendu, un type de données doit fournir un format étendu qui suit les règles données dans `src/include/utils/expandeddatum.h`, et fournir des fonctions pour « étendre » une valeur varlena plate en un format étendu et « aplatir » un format étendu en une représentation varlena standard. Puis s'assurer que toutes les fonctions C pour le type de données puissent accepter chaque représentation, si possible en convertissant l'une en l'autre immédiatement à réception. Ceci ne nécessite pas de corriger les fonctions existantes pour le type de données car la macro standard `PG_DETOAST_DATUM` est définie pour convertir les entrées étendues dans le format plat standard. De ce fait, les fonctions existantes qui fonctionnent avec le format varlena plat continueront de fonctionner, bien que moins efficacement, avec des entrées étendues ; elles n'ont pas besoin d'être converties jusqu'à ou à moins que d'avoir de meilleures performances soit important.

Les fonctions C qui savent comment fonctionner avec une représentation étendue tombent typiquement dans deux catégories : celles qui savent seulement gérer le format étendu et celles qui peuvent gérer les deux formats. Les premières sont plus simples à écrire mais peuvent être moins performantes car la conversion d'une entrée à plat vers sa forme étendue par une seule fonction pourrait coûter plus que ce qui est gagné par le format étendu. Lorsque seul le format étendu est géré, la conversion des entrées à plat vers le format étendu peut être cachée à l'intérieur d'une macro de récupération des arguments, pour que la fonction n'apparaisse pas plus complexe qu'une fonction travaillant avec le format varlena standard. Pour gérer les deux types d'entrée, écrire une fonction de récupération des arguments qui peut enelver du toast les entrées varlena externes, à court en-tête et compressées, mais qui n'étend pas les entrées. Une telle fonction peut être définie comme renvoyant un pointeur vers une union du fichier varlena à plat et du format étendu. Ils peuvent utiliser la macro `VARATT_IS_EXPANDED_HEADER()` pour déterminer le format reçu.

L'infrastructure TOAST permet non seulement de distinguer les valeurs varlena standard des valeurs étendues, mais aussi de distinguer les pointeurs « read-write » et « read-only » vers les valeurs étendues. Les fonctions C qui ont seulement besoin d'examiner une valeur étendue ou qui vont seulement la changer d'une façon sûre et non visible sémantiquement, doivent ne pas faire attention au type de pointeur qu'elles ont reçus. Les fonctions C qui produisent une version modifiée d'une valeur en entrée sont autorisées à modifier une valeur étendue en entrée directement si elles reçoivent un pointeur read-only ; dans ce cas, elles doivent tout d'abord copier la valeur pour produire la nouvelle valeur à modifier. Une fonction C qui a construit une nouvelle valeur étendue devrait toujours renvoyer un pointeur read-write vers ce dernier. De plus, une fonction C qui modifie une valeur étendue en read-write devrait faire attention à laisser la valeur dans un état propre s'il échoue en chemin.

Pour des exemples de code sur des valeurs étendues, voir l'infrastructure sur les tableaux standards, tout particulièrement `src/backend/utils/adt/array_expanded.c`.

38.14. Opérateurs définis par l'utilisateur

chaque opérateur est un « sucre syntaxique » pour l'appel d'une fonction sous-jacente qui effectue le véritable travail ; aussi devez-vous en premier lieu créer cette fonction avant de pouvoir créer l'opérateur. Toutefois, un opérateur n'est pas *simplement* un « sucre syntaxique » car il apporte des informations supplémentaires qui aident le planificateur de requête à optimiser les requêtes utilisées par l'opérateur. La prochaine section est consacrée à l'explication de ces informations additionnelles.

postgresql accepte les opérateurs préfixes et infixes. Les opérateurs peuvent être surchargés ; c'est-à-dire que le même nom d'opérateur peut être utilisé pour différents opérateurs à condition qu'ils aient des nombres et des types différents d'opérandes. Quand une requête est exécutée, le système détermine l'opérateur à appeler en fonction du nombre et des types d'opérandes fournis.

Voici un exemple de création d'opérateur pour l'addition de deux nombres complexes. Nous supposons avoir déjà créé la définition du type `complex` (voir la Section 38.13). premièrement, nous avons besoin d'une fonction qui fasse le travail, ensuite nous pouvons définir l'opérateur :

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C;

 CREATE OPERATOR + (
     leftarg = complex,
     rightarg = complex,
     function = complex_add,
     commutator = +
 );
```

Maintenant nous pouvons exécuter la requête comme ceci :

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

Nous avons montré comment créer un opérateur binaire. Pour créer des opérateurs préfixes, il suffit d'omettre `leftarg`. La clause `function` et les clauses argument sont les seuls éléments requis dans la commande `create operator`. la clause `commutator` montrée dans l'exemple est une indication optionnelle pour l'optimiseur de requête. Des détails supplémentaires sur la clause `commutator` et d'autres compléments d'optimisation sont donnés dans la prochaine section.

38.15. Informations sur l'optimisation d'un opérateur

Une définition d'opérateur PostgreSQL peut inclure plusieurs clauses optionnelles qui donnent au système des informations utiles sur le comportement de l'opérateur. Ces clauses devraient être fournies chaque fois que c'est utile car elles peuvent considérablement accélérer l'exécution des requêtes utilisant cet opérateur. Mais si vous le faites, vous devez être sûr de leur justesse ! L'usage incorrect d'une clause d'optimisation peut être la cause de requêtes lentes, des sorties subtilement fausses ou d'autres effets pervers. Vous pouvez toujours abandonner une clause d'optimisation si vous n'êtes pas sûr d'elle ; la seule conséquence est un possible ralentissement des requêtes.

Des clauses additionnelles d'optimisation pourront être ajoutées dans les futures versions de postgresql. celles décrites ici sont toutes celles que cette version comprend.

Il est aussi possible d'attacher une fonction de support pour l'optimiseur à la fonction sous-jacente d'un opérateur, fournissant une autre façon de dire au système la façon dont se comporte l'opérateur. Voir Section 38.11 pour plus d'informations.

38.15.1. `COMMUTATOR`

Si elle est fournie, la clause `commutator` désigne un opérateur qui est le commutateur de l'opérateur en cours de définition. Nous disons qu'un opérateur A est le commutateur de l'opérateur B si (x A y) est égal à (y B x) pour toute valeur possible de x, y. Notez que B est aussi le commutateur de A. Par exemple, les opérateurs < et > pour un type particulier de données sont habituellement des commutateurs l'un pour l'autre, et l'opérateur + est habituellement commutatif avec lui-même. Mais l'opérateur - n'est habituellement commutatif avec rien.

Le type de l'opérande gauche d'un opérateur commuté est le même que l'opérande droit de son commutateur, et vice versa. Aussi postgresql n'a besoin que du nom de l'opérateur commutateur pour consulter le commutateur, et c'est tout ce qui doit être fourni à la clause `commutator` .

Vous avez juste à définir un opérateur auto-commutateur. Mais les choses sont un peu plus compliquées quand vous définissez une paire de commutateurs : comment peut-on définir la référence du premier au second alors que ce dernier n'est pas encore défini ? Il y a deux solutions à ce problème :

- Une façon d'opérer est d'omettre la clause `commutator` dans le premier opérateur que vous définissez et ensuite d'en insérer une dans la définition du second opérateur. Puisque postgresql sait que les opérateurs commutatifs vont par paire, quand il voit la seconde définition, il retourne instantanément remplir la clause `commutator` manquante dans la première définition.

- L'autre façon, plus directe, est de simplement inclure les clauses `commutator` dans les deux définitions. quand postgresql traite la première définition et réalise que la clause `commutator` se réfère à un opérateur inexistant, le système va créer une entrée provisoire pour cet opérateur dans le catalogue système. Cette entrée sera pourvue seulement de données valides pour le nom

de l'opérateur, les types d'opérande droit et gauche et le type du résultat, puisque c'est tout ce que postgresql peut déduire à ce point. la première entrée du catalogue pour l'opérateur sera liée à cette entrée provisoire. Plus tard, quand vous définirez le second opérateur, le système mettra à jour l'entrée provisoire avec les informations additionnelles fournies par la seconde définition. Si vous essayez d'utiliser l'opérateur provisoire avant qu'il ne soit complété, vous aurez juste un message d'erreur.

38.15.2. NEGATOR

La clause `negator` dénomme un opérateur qui est l'opérateur de négation de l'opérateur en cours de définition. Nous disons qu'un opérateur A est l'opérateur de négation de l'opérateur B si tous les deux renvoient des résultats booléens et si (x A y) est égal à NOT (x B y) pour toutes les entrées possible x, y. Notez que B est aussi l'opérateur de négation de A. Par exemple, < et >= forment une paire d'opérateurs de négation pour la plupart des types de données. Un opérateur ne peut jamais être validé comme son propre opérateur de négation .

Au contraire des commutateurs, une paire d'opérateurs unaires peut être validée comme une paire d'opérateurs de négation réciproques ; ce qui signifie que (A x) est égal à NOT (B x) pour tout x.

L'opérateur de négation d'un opérateur doit avoir les mêmes types d'opérandes gauche et/ou droit que l'opérateur à définir comme avec `commutator`. seul le nom de l'opérateur doit être donné dans la clause `negator`.

Définir un opérateur de négation est très utile pour l'optimiseur de requêtes car il permet de simplifier des expressions telles que `not (x = y)` en `x <> y`. ceci arrive souvent parce que les opérations `not` peuvent être insérées à la suite d'autres réarrangements.

Des paires d'opérateurs de négation peuvent être définies en utilisant la même méthode que pour les commutateurs.

38.15.3. RESTRICT

La clause `restrict`, si elle est invoquée, nomme une fonction d'estimation de sélectivité de restriction pour cet opérateur (notez que c'est un nom de fonction, et non pas un nom d'opérateur). Les clauses `restrict` n'ont de sens que pour les opérateurs binaires qui renvoient un type `boolean`. un estimateur de sélectivité de restriction repose sur l'idée de prévoir quelle fraction des lignes dans une table satisfera une condition de clause `where` de la forme :

```
colonne OP constante
```

pour l'opérateur courant et une valeur constante particulière. Ceci aide l'optimiseur en lui donnant une idée du nombre de lignes qui sera éliminé par les clauses `where` qui ont cette forme (vous pouvez vous demander, qu'arrivera-t-il si la constante est à gauche ? hé bien, c'est une des choses à laquelle sert le `commutator`...).

L'écriture de nouvelles fonctions d'estimation de restriction de sélectivité est éloignée des objectifs de ce chapitre mais, heureusement, vous pouvez habituellement utiliser un des estimateurs standards du système pour beaucoup de vos propres opérateurs. Voici les estimateurs standards de restriction :

`eqsel` pour =
`neqsel` pour <>
`scalarltsel` pour <
`scalarlesel` pour <=
`scalargtsel` pour >
`scalargesel` pour >=

Vous pouvez fréquemment vous en sortir à bon compte en utilisant soit `eqsel` ou `neqsel` pour des opérateurs qui ont une très grande ou une très faible sélectivité, même s'ils ne sont pas réellement

égalité ou inégalité. Par exemple, les opérateurs géométriques d'égalité approchée utilisent `eqsel` en supposant habituellement qu'ils ne correspondent qu'à une petite fraction des entrées dans une table.

Vous pouvez utiliser `scalarltsel`, `scalarlesel`, `scalargtsel` et `scalargesel` pour des comparaisons de types de données qui possèdent un moyen de conversion en scalaires numériques pour les comparaisons de rang. Si possible, ajoutez le type de données à ceux acceptés par la fonction `convert_to_scalar()` dans `src/backend/utils/adt/selfuncs.c` (finalement, cette fonction devrait être remplacée par des fonctions pour chaque type de données identifié grâce à une colonne du catalogue système `pg_type` ; mais cela n'a pas encore été fait). si vous ne faites pas ceci, les choses fonctionneront mais les estimations de l'optimiseur ne seront pas aussi bonnes qu'elles pourraient l'être.

Une autre fonction native d'estimation de sélectivité est `matchingsel`, qui fonctionnera pour pratiquement tous les opérateurs binaires, si les statistiques MCV ou histogramme sont calculées pour ce type de données en entrée. Son estimation par défaut est configurée à deux fois l'estimation par défaut utilisé dans `eqsel`, le rendant convenable pour les opérateurs de comparaison qui sont un peu moins strictque l'égalité. (Vous pouvez appeler la fonction sous-jacente `generic_restriction_selectivity` en fournissant une estimation par défaut différente.)

D'autres fonctions d'estimation de sélectivité conçues pour les opérateurs géométriques sont placées dans `src/backend/utils/adt/geo_selfuncs.c` : `areasel`, `positionsel` et `contsel`. lors de cette rédaction, ce sont seulement des fragments mais vous pouvez vouloir les utiliser (ou mieux les améliorer).

38.15.4. `JOIN`

La clause `join`, si elle est invoquée, nomme une fonction d'estimation de sélectivité de jointure pour l'opérateur (notez que c'est un nom de fonction, et non pas un nom d'opérateur). Les clauses `join` n'ont de sens que pour les opérateurs binaires qui renvoient un type `boolean`. un estimateur de sélectivité de jointure repose sur l'idée de prévoir quelle fraction des lignes dans une paire de tables satisfera une condition de clause `where` de la forme :

```
table1.colonne1 OP table2.colonne2
```

pour l'opérateur courant. Comme pour la clause `restrict`, ceci aide considérablement l'optimiseur en lui indiquant parmi plusieurs séquences de jointure possibles laquelle prendra vraisemblablement le moins de travail.

Comme précédemment, ce chapitre n'essaiera pas d'expliquer comment écrire une fonction d'estimation de sélectivité de jointure mais suggérera simplement d'utiliser un des estimateurs standard s'il est applicable :

`eqjoinsel` pour =
`neqjoinsel` pour <>
`scalarltjoinsel` pour <
`scalarlejoinsel` pour <=
`scalargtjoinsel` pour >
`scalargejoinsel` pour >=
`matchingjoinsel` for generic matching operators
`areajoinsel` pour des comparaisons basées sur une aire 2D
`positionjoinsel` pour des comparaisons basées sur des positions 2D
`contjoinsel` pour des comparaisons basées sur un appartenance 2D

38.15.5. `HASHES`

La clause `hashes` indique au système qu'il est permis d'utiliser la méthode de jointure-découpage pour une jointure basée sur cet opérateur. `hashes` n'a de sens que pour un opérateur binaire qui

renvoie un `boolean` et en pratique l'opérateur égalité doit représenter l'égalité pour certains types de données ou paire de type de données.

La jointure-découpage repose sur l'hypothèse que l'opérateur de jointure peut seulement renvoyer la valeur vrai pour des paires de valeurs droite et gauche qui correspondent au même code de découpage. Si deux valeurs sont placées dans deux différents paquets (« buckets »), la jointure ne pourra jamais les comparer avec la supposition implicite que le résultat de l'opérateur de jointure doit être faux. Ainsi, il n'y a aucun sens à spécifier `hashes` pour des opérateurs qui ne représentent pas une certaine forme d'égalité. Dans la plupart des cas, il est seulement pratique de supporter le hachage pour les opérateurs qui prennent le même type de données sur chaque côté. Néanmoins, quelque fois, il est possible de concevoir des fonctions de hachage compatibles pour deux type de données, voire plus ; c'est-à-dire pour les fonctions qui généreront les mêmes codes de hachage pour des valeurs égales même si elles ont des représentations différentes. Par exemple, il est assez simple d'arranger cette propriété lors du hachage d'entiers de largeurs différentes.

Pour être marqué `hashes`, l'opérateur de jointure doit apparaître dans une famille d'opérateurs d'index de découpage. Ceci n'est pas rendu obligatoire quand vous créez l'opérateur, puisque évidemment la classe référençant l'opérateur peut ne pas encore exister. Mais les tentatives d'utilisation de l'opérateur dans les jointure-découpage échoueront à l'exécution si une telle famille d'opérateur n'existe pas. Le système a besoin de la famille d'opérateur pour définir la fonction de découpage spécifique au type de données d'entrée de l'opérateur. Bien sûr, vous devez également créer des fonctions de découpage appropriées avant de pouvoir créer la famille d'opérateur.

On doit apporter une grande attention à la préparation des fonctions de découpage parce qu'il y a des processus dépendants de la machine qui peuvent ne pas faire les choses correctement. Par exemple, si votre type de données est une structure dans laquelle peuvent se trouver des bits de remplissage sans intérêt, vous ne pouvez pas simplement passer la structure complète à la fonction `hash_any` (à moins d'écrire vos autres opérateurs et fonctions de façon à s'assurer que les bits inutilisés sont toujours zéro, ce qui est la stratégie recommandée). Un autre exemple est fourni sur les machines qui respectent le standard de virgule-flottante ieee, le zéro négatif et le zéro positif sont des valeurs différentes (les motifs de bit sont différents) mais ils sont définis pour être égaux. Si une valeur flottante peut contenir un zéro négatif, alors une étape supplémentaire est nécessaire pour s'assurer qu'elle génère la même valeur de découpage qu'un zéro positif.

Un opérateur joignable par hachage doit avoir un commutateur (lui-même si les types de données des deux opérandes sont identiques, ou un opérateur d'égalité relatif dans le cas contraire) qui apparaît dans la même famille d'opérateur. Si ce n'est pas le cas, des erreurs du planificateur pourraient apparaître quand l'opérateur est utilisé. De plus, une bonne idée (mais pas obligatoire) est qu'une famille d'opérateur de hachage supporte les tupes de données multiples pour fournir des opérateurs d'égalité pour chaque combinaison des types de données ; cela permet une meilleure optimisation.

> **Note**
>
> La fonction sous-jacente à un opérateur de jointure-découpage doit être marquée immuable ou stable. Si elle est volatile, le système n'essaiera jamais d'utiliser l'opérateur pour une jointure hachage.

> **Note**
>
> Si un opérateur de jointure-hachage a une fonction sous-jacente marquée stricte, la fonction doit également être complète : cela signifie qu'elle doit renvoyer TRUE ou FALSE, jamais NULL, pour n'importe quelle double entrée non NULL. Si cette règle n'est pas respectée, l'optimisation de découpage des opérations `in` peut générer des résultats faux (spécifiquement, `in` devrait renvoyer false quand la réponse correcte devrait être NULL ; ou bien il devrait renvoyer une erreur indiquant qu'il ne s'attendait pas à un résultat NULL).

38.15.6. MERGES

La clause `merges`, si elle est présente, indique au système qu'il est permis d'utiliser la méthode de jointure-union pour une jointure basée sur cet opérateur. `merges` n'a de sens que pour un opérateur binaire qui renvoie un `boolean` et, en pratique, cet opérateur doit représenter l'égalité pour des types de données ou des paires de types de données.

La jointure-union est fondée sur le principe d'ordonner les tables gauche et droite et ensuite de les comparer en parallèle. Ainsi, les deux types de données doivent être capable d'être pleinement ordonnées, et l'opérateur de jointure doit pouvoir réussir seulement pour des paires de valeurs tombant à la « même place » dans l'ordre de tri. En pratique, cela signifie que l'opérateur de jointure doit se comporter comme l'opérateur égalité. Mais il est possible de faire une jointure-union sur deux types de données distincts tant qu'ils sont logiquement compatibles. Par exemple, l'opérateur d'égalité `smallint`-contre-`integer` est susceptible d'opérer une jointure-union. Nous avons seulement besoin d'opérateurs de tri qui organisent les deux types de données en séquences logiquement comparables.

Pour être marqué `MERGES`, l'opérateur de jointure doit apparaître en tant que membre d'égalité d'une famille opérateur d'index `btree`. Ceci n'est pas forcé quand vous créez l'opérateur puisque, bien sûr, la famille d'opérateur référente n'existe pas encore. Mais l'opérateur ne sera pas utilisé pour les jointures de fusion sauf si une famille d'opérateur correspondante est trouvée. L'option `MERGES` agit en fait comme une aide pour le planificateur lui indiquant qu'il est intéressant de chercher une famille d'opérateur correspondant.

Un opérateur joignable par fusion doit avoir un commutateur (lui-même si les types de données des deux opérateurs sont identiques, ou un opérateur d'égalité en relation dans le cas contraire) qui apparaît dans la même famille d'opérateur. Si ce n'est pas le cas, des erreurs du planificateur pourraient apparaître quand l'opérateur est utilisé. De plus, une bonne idée (mais pas obligatoire) est qu'une famille d'opérateur de hachage supporte les tupes de données multiples pour fournir des opérateurs d'égalité pour chaque combinaison des types de données ; cela permet une meilleure optimisation.

> **Note**
>
> La fonction sous-jacente à un opérateur de jointure-union doit être marquée immuable ou stable. Si elle est volatile, le système n'essaiera jamais d'utiliser l'opérateur pour une jointure union.

38.16. Interfacer des extensions d'index

Les procédures décrites jusqu'à maintenant permettent de définir de nouveaux types, de nouvelles fonctions et de nouveaux opérateurs. Néanmoins, nous ne pouvons pas encore définir un index sur une colonne d'un nouveau type de données. Pour cela, nous devons définir une *classe d'opérateur* pour le nouveau type de données. Plus loin dans cette section, nous illustrerons ce concept avec un exemple : une nouvelle classe d'opérateur pour la méthode d'indexation B-tree qui enregistre et trie des nombres complexes dans l'ordre ascendant des valeurs absolues.

Les classes d'opérateur peuvent être groupées en *familles d'opérateur* pour afficher les relations entre classes compatibles sémantiquement. Quand un seul type de données est impliqué, une classe d'opérateur est suffisant, donc nous allons nous fixer sur ce cas en premier puis retourner aux familles d'opérateur.

38.16.1. Méthodes d'indexation et classes d'opérateurs

La table `pg_am` contient une ligne pour chaque méthode d'indexation (connue en interne comme méthode d'accès). Le support pour l'accès normal aux tables est implémenté dans PostgreSQL mais

toutes les méthodes d'index sont décrites dans pg_am. Il est possible d'ajouter une nouvelle méthode d'accès aux index en écrivant le code nécessaire et en ajoutant ensuite une ligne dans la table pg_am -- mais ceci est au-delà du sujet de ce chapitre (voir le Chapitre 64).

Les routines pour une méthode d'indexation n'ont pas à connaître directement les types de données sur lesquels opère la méthode d'indexation. Au lieu de cela, une *classe d'opérateur* identifie l'ensemble d'opérations que la méthode d'indexation doit utiliser pour fonctionner avec un type particulier de données. Les classes d'opérateurs sont ainsi dénommées parce qu'une de leur tâche est de spécifier l'ensemble des opérateurs de la clause WHERE utilisables avec un index (c'est-à-dire, qui peuvent être requalifiés en balayage d'index). Une classe d'opérateur peut également spécifier des *fonctions de support*, nécessaires pour les opérations internes de la méthode d'indexation mais sans correspondance directe avec un quelconque opérateur de clause WHERE pouvant être utilisé avec l'index.

Il est possible de définir plusieurs classes d'opérateurs pour le même type de données et la même méthode d'indexation. Ainsi, de multiples ensembles de sémantiques d'indexation peuvent être définis pour un seul type de données. Par exemple, un index B-tree exige qu'un tri ordonné soit défini pour chaque type de données auquel il peut s'appliquer. Il peut être utile pour un type de donnée de nombre complexe de disposer d'une classe d'opérateur B-tree qui trie les données selon la valeur absolue complexe, une autre selon la partie réelle, etc. Typiquement, une des classes d'opérateur sera considérée comme plus utile et sera marquée comme l'opérateur par défaut pour ce type de données et cette méthode d'indexation.

Le même nom de classe d'opérateur peut être utilisé pour plusieurs méthodes d'indexation différentes (par exemple, les méthodes d'index B-tree et hash ont toutes les deux des classes d'opérateur nommées int4_ops) mais chacune de ces classes est une entité indépendante et doit être définie séparément.

38.16.2. Stratégies des méthode d'indexation

Les opérateurs associés à une classe d'opérateur sont identifiés par des « numéros de stratégie », servant à identifier la sémantique de chaque opérateur dans le contexte de sa classe d'opérateur. Par exemple, les B-trees imposent un classement strict selon les clés, du plus petit au plus grand. Ainsi, des opérateurs comme « plus petit que » et « plus grand que » sont intéressants pour un B-tree. Comme PostgreSQL permet à l'utilisateur de définir des opérateurs, PostgreSQL ne peut pas rechercher le nom d'un opérateur (par exemple, < ou >=) et rapporter de quelle comparaison il s'agit. Au lieu de cela, la méthode d'indexation définit un ensemble de « stratégies », qui peuvent être comprises comme des opérateurs généralisés. Chaque classe d'opérateur spécifie l'opérateur effectif correspondant à chaque stratégie pour un type de donnée particulier et pour une interprétation de la sémantique d'index.

La méthode d'indexation B-tree définit cinq stratégies, qui sont exposées dans le Tableau 38.3.

Tableau 38.3. Stratégies B-tree

Opération	Numéro de stratégie
plus petit que	1
plus petit ou égal	2
égal	3
plus grand ou égal	4
plus grand que	5

Les index de découpage permettent seulement des comparaisons d'égalité et utilisent ainsi une seule stratégie exposée dans le Tableau 38.4.

Tableau 38.4. Stratégies de découpage

Opération	Numéro de stratégie
égal à	1

Les index GiST sont plus flexibles : ils n'ont pas du tout un ensemble fixe de stratégies. À la place, la routine de support de « cohérence » de chaque classe d'opérateurs GiST interprète les numéros de stratégie comme elle l'entend. Comme exemple, plusieurs des classes d'opérateurs GiST indexent les objets géométriques à deux dimensions fournissant les stratégies « R-tree » affichées dans Tableau 38.5. Quatre d'entre elles sont des vrais tests à deux dimensions (surcharge, identique, contient, contenu par) ; quatre autres considèrent seulement la direction X ; et les quatre dernières fournissent les mêmes tests dans la direction Y.

Tableau 38.5. Stratégies « R-tree » pour GiST à deux dimensions

Opération	Numéro de stratégie
strictement à gauche de	1
ne s'étend pas à droite de	2
surcharge	3
ne s'étend pas à gauche de	4
strictement à droite de	5
identique	6
contient	7
contenu par	8
ne s'étend pas au dessus	9
strictement en dessous	10
strictement au dessus	11
ne s'étend pas en dessous	12

Les index SP-GiST sont similaires aux index GiST en flexibilité : ils n'ont pas un ensemble fixe de stratégie. À la place, les routines de support de chaque classe d'opérateurs interprètent les numéros de stratégie suivant la définition de la classe d'opérateurs. Comme exemple, les numéros des stratégies utilisés par les classes d'opérateurs sur des points sont affichés dans Tableau 38.6.

Tableau 38.6. Stratégies point SP-GiST

Opération	Numéro de stratégie
strictement à gauche	1
strictement à droite	5
identique	6
contenu par	8
strictement en dessous	10
strictement au dessus	11

Les index GIN sont similaires aux index GiST et SP-GiST, dans le fait qu'ils n'ont pas d'ensemble fixé de stratégies. À la place, les routines support de chaque opérateur de classe interprètent les numéros de stratégie suivant la définition de la classe d'opérateurs. Comme exemple, les numéros de stratégie utilisés par la classe d'opérateurs internes pour les tableaux sont affichés dans Tableau 38.7.

Tableau 38.7. Stratégies des tableaux GIN

Opération	Numéro de stratégie
surcharge	1
contient	2
est contenu par	3

Opération	Numéro de stratégie
identique	4

Les index BRIN sont similaires aux index GiST, SP-GiST et GIN dans le fait qu'ils n'ont pas un ensemble fixe de stratégies. À la place, les routines de support de chaque classe d'opérateurs interprètent les numéros de stratégie suivant la définition de la classe d'opérateurs. Par exemple, les numéros de stratégie utilisés par les classes d'opérateurs `Minmax` sont indiqués dans Tableau 38.8.

Tableau 38.8. Stratégies MinMax pour BRIN

Opération	Numéro de stratégie
inférieur	1
inférieur ou égal	2
égal	3
supérieur ou égal	4
supérieur	5

Notez que tous les opérateurs ci-dessus renvoient des valeurs de type booléen. Dans la pratique, tous les opérateurs définis comme index method search operators doivent renvoyer un type `boolean` puisqu'ils doivent apparaître au plus haut niveau d'une clause `WHERE` pour être utilisés avec un index. (Certaines méthodes d'accès aux index prennent également en charge les *opérateurs de tri*, qui ne renvoient généralement pas de valeurs booléennes ; cette fonctionnalité est discutée dans Section 38.16.7.)

38.16.3. Routines d'appui des méthodes d'indexation

Généralement, les stratégies n'apportent pas assez d'informations au système pour indiquer comment utiliser un index. Dans la pratique, les méthodes d'indexation demandent des routines d'appui additionnelles pour fonctionner. Par exemple, les méthodes d'index B-tree doivent être capables de comparer deux clés et de déterminer laquelle est supérieure, égale ou inférieure à l'autre. De la même façon, la méthode d'indexation hash doit être capable de calculer les codes de hachage pour les valeurs de clés. Ces opérations ne correspondent pas à des opérateurs utilisés dans les commandes SQL ; ce sont des routines administratives utilisées en interne par des méthodes d'index.

Comme pour les stratégies, la classe d'opérateurs énumère les fonctions spécifiques et le rôle qu'elles doivent jouer pour un type de donnée donné et une interprétation sémantique donnée. La méthode d'indexation définit l'ensemble des fonctions dont elle a besoin et la classe d'opérateurs identifie les fonctions exactes à utiliser en les assignant aux « numéros de fonction d'appui » spécifiés par la méthode d'indexage.

De plus, certaines classes d'opérateurs autorisent les utilisateurs à indiquer des paramètres pour contrôler leur comportement. Chaque méthode native d'accès aux index a une fonction de support optionnelle appelée `options`, qui définit un ensemble de paramètres spécifiques à la classe d'opérateurs.

Les B-trees requièrent une fonction support de comparaison et permettent quatre fonctions support supplémentaires à fournir comme option de la classe d'opérateurs, comme indiqué dans Tableau 38.9. Les prérequis pour ces fonctions support sont expliqués en détails dans Section 67.3.

Tableau 38.9. Fonctions d'appui de B-tree

Fonction	Numéro d'appui
Comparer deux clés et renvoyer un entier inférieur à zéro, zéro ou supérieure à zéro indiquant si la première clé est inférieure, égale ou supérieure à la deuxième.	1

Fonction	Numéro d'appui
Renvoyer les adresses des fonctions de support de tri, appelables en C (optionnel)	2
Comparer une valeur test à une valeur de base plus/moins un décalage, et renvoyer true ou false suivant le résultat de la comparaison (optionnel)	3
Détermine s'il est sûr que les index utilisant la classe d'opérateurs s'appliquent à l'optimisation de déduplication des Btree (optionnel)	4
Définit des options spécifiques à cette classe d'opérateurs (optionnel)	5

Les index hash requièrent une fonction d'appui, et permettent deux fonctions supplémentaires à fournir à la classe d'opérateurs, comme indiqué dans Tableau 38.10.

Tableau 38.10. Fonctions d'appui pour découpage

Fonction	Numéro d'appui
Calculer la valeur de hachage 32 bits pour une clé	1
Calcule la valeur de hachage sur 64 bits d'une clé pour un sel de 64 bits donné ; si le sel vaut 0, les 32 bits de poids faible du résultat doivent correspondre à la valeur qui aurait été calculée par la fonction 1 (facultative)	2
Définit des options spécifiques à cette classe d'opérateurs (optionnel)	3

Les index GiST ont onze fonctions d'appui, dont six facultatives, exposées dans le Tableau 38.11. (Pour plus d'informations, voir Chapitre 68.)

Tableau 38.11. Fonctions d'appui pour GiST

Fonction	Description	Numéro d'appui
`consistent`	détermine si la clé satisfait le qualifiant de la requête (variante Booléenne) (facultatif si la fonction d'appui 6 est présente)	1
`union`	calcule l'union d'un ensemble de clés	2
`compress`	calcule une représentation compressée d'une clé ou d'une valeur à indexer (optionnelle)	3
`decompress`	calcule une représentation décompressée d'une clé compressée (optionnelle)	4
`penalty`	calcule la pénalité pour l'insertion d'une nouvelle clé dans un sous-arbre avec la clé du sous-arbre indiqué	5
`picksplit`	détermine les entrées d'une page qui sont à déplacer vers la nouvelle page et calcule les clés d'union pour les pages résultantes	6
`same`	compare deux clés et renvoie true si elles sont identiques	7
`distance`	détermine la distance de la clé à la valeur de la requête (optionnel)	8
`fetch`	calcule la représentation originale d'une clé compressée pour les parcours d'index seul (optionnel)	9
`options`	Définit des options spécifiques à cette classe d'opérateurs (optionnel)	10

Fonction	Description	Numéro d'appui
sortsupport	fournit un comparateur utilisé pour la construction rapide d'index (optionnelle)	11

Les index SP-GiST ont six fonctions d'appui, dont une facultative, comme indiquées dans Tableau 38.12. (Pour plus d'informations, voir Chapitre 69.)

Tableau 38.12. Fonctions de support SP-GiST

Fonction	Description	Numéro de support
config	fournit des informations basiques sur la classe d'opérateurs	1
choose	détermine comment insérer une nouvelle valeur dans une ligne interne	2
picksplit	détermine comment partitionner un ensemble de valeurs	3
inner_consistent	détermine la sous-partition à rechercher pour une requête	4
leaf_consistent	détermine si la clé satisfait le qualificateur de la requête	5
triConsistent	détermine si la valeur satisfait le qualificateur de la requête (variante ternaire) (facultatif si la fonction de support 4 est présente)	6

Les index GIN ont sept fonctions d'appui, dont quatre optionnelles, exposées dans le Tableau 38.13. (Pour plus d'informations, voir Chapitre 70.)

Tableau 38.13. Fonctions d'appui GIN

Fonction	Description	Numéro d'appui
compare	Compare deux clés et renvoie un entier plus petit que zéro, zéro ou plus grand que zéro, indiquant si la première clé est plus petit, égal à ou plus grand que la seconde.	1
extractValue	Extrait les clés à partir d'une condition de requête	2
extractQuery	Extrait les clés à partir d'une condition de requête	3
consistent	Détermine la valeur correspondant à la condition de requête	4
comparePartial	compare la clé partielle de la requête et la clé de l'index, et renvoie un entier négatif, nul ou positif, indiquant si GIN doit ignorer cette entrée d'index, traiter l'entrée comme une correspondance ou arrêter le parcours d'index (optionnelle)	5
triConsistent	détermine si la valeur correspond à la condition de la requête (variante ternaire) (optionnelle si la fonction d'appui 4 est présente)	6

Fonction	Description	Numéro d'appui
`options`	Définit des options spécifiques à cette classe d'opérateurs (optionnel)	7

Les index BRIN ont cinq fonctions d'appui basiques, dont une optionnelle, comme indiqué dans Tableau 38.14 ; ces versions des fonctions basiques nécessitent des fonctions d'appui supplémentaires. (Pour plus d'informations, voir Section 71.3.)

Tableau 38.14. Fonctions de support BRIN

Fonction	Description	Numéro de support
`opcInfo`	renvoie des informations internes décrivant les données de résumé des colonnes indexées	1
`add_value`	ajoute une nouvelle valeur à un enregistrement d'index existant	2
`consistent`	détermine si la valeur correspond à une condition de la requête	3
`union`	calcule l'union de deux enregistrements résumés	4
`options`	Définit des options spécifiques à cette classe d'opérateurs (optionnel)	5

Contrairement aux opérateurs de recherche, les fonctions d'appui renvoient le type de donnée, quel qu'il soit, que la méthode d'indexation particulière attend, par exemple, dans le cas de la fonction de comparaison des B-trees, un entier signé. Le nombre et le type des arguments pour chaque fonction de support peuvent dépendre de la méthode d'indexage. Pour les index B-tree et de hachage, les fonctions de support pour la comparaison et le hachage prennent les mêmes types de données en entrée que les opérateurs inclus dans la classe d'opérateurs, mais ce n'est pas le cas pour la plupart des fonctions de support GiST, SP-GiST, GIN et BRIN.

38.16.4. Exemple

Maintenant que nous avons vu les idées, voici l'exemple promis de création d'une nouvelle classe d'opérateurs. Cette classe d'opérateurs encapsule les opérateurs qui trient les nombres complexes selon l'ordre de la valeur absolue, aussi avons-nous choisi le nom de `complex_abs_ops`. En premier lieu, nous avons besoin d'un ensemble d'opérateurs. La procédure pour définir des opérateurs a été discutée dans la Section 38.14. Pour une classe d'opérateurs sur les B-trees, nous avons besoin des opérateurs :

- valeur absolue inférieure à (stratégie 1) ;
- valeur absolue inférieure ou égale à (stratégie 2) ;
- valeur absolue égale à (stratégie 3) ;
- valeur absolue supérieure-ou-égale à (stratégie 4) ;
- valeur absolue supérieure à (stratégie 5) ;

Le plus simple moyen de définir un ensemble d'opérateurs de comparaison est d'écrire en premier la fonction de comparaison B-tree, puis d'écrire les autres fonctions en tant que wrapper de la fonction de support. Ceci réduit les risques de résultats incohérents pour les cas spécifiques. En suivant cette approche, nous devons tout d'abord écrire :

```
#define Mag(c)  ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
```

```
    double      amag = Mag(a),
                bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

Maintenant, la fonction plus-petit-que ressemble à ceci :

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

Les quatre autres fonctions diffèrent seulement sur la façon dont elles comparent le résultat de la fonction interne au zéro.

Maintenant, déclarons en SQL les fonctions et les opérateurs basés sur ces fonctions :

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'nom_fichier', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = complex, rightarg = complex, procedure =
 complex_abs_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

Il est important de spécifier les fonctions de sélectivité de restriction et de jointure, sinon l'optimiseur sera incapable de faire un usage effectif de l'index.

Voici d'autres choses importantes à noter :

- Il ne peut y avoir qu'un seul opérateur nommé, disons, = et acceptant un type `complex` pour ses deux opérandes. Dans le cas présent, nous n'avons aucun autre opérateur = pour `complex` mais, si nous construisons un type de donnée fonctionnel, nous aurions certainement désiré que = soit l'opération ordinaire d'égalité pour les nombres complexes (et non pour l'égalité de leurs valeurs absolues). Dans ce cas, nous aurions eu besoin d'utiliser un autre nom d'opérateur pour notre fonction `complex_abs_eq`.
- Bien que PostgreSQL puisse se débrouiller avec des fonctions ayant le même nom SQL, tant qu'elles ont en argument des types de données différents, en C il ne peut exister qu'une fonction globale pour un nom donné. Aussi ne devons-nous pas donner un nom simple comme `abs_eq`. Habituellement, inclure le nom du type de données dans le nom de la fonction C est une bonne habitude pour ne pas provoquer de conflit avec des fonctions pour d'autres types de donnée.
- Nous aurions pu faire de `abs_eq` le nom SQL de la fonction, en laissant à PostgreSQL le soin de la distinguer de toute autre fonction SQL de même nom par les types de données en argument.

Pour la simplicité de l'exemple, nous donnerons à la fonction le même nom au niveau de C et au niveau de SQL.

La prochaine étape est l'enregistrement de la routine d'appui nécessaire pour les B-trees. Le code exemple C qui implémente ceci est dans le même fichier qui contient les fonctions d'opérateur. Voici comment déclarer la fonction :

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C;
```

Maintenant que nous avons les opérateurs requis et la routine d'appui, nous pouvons enfin créer la classe d'opérateurs.

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

Et c'est fait ! Il devrait être possible maintenant de créer et d'utiliser les index B-tree sur les colonnes `complex`.

Nous aurions pu écrire les entrées de l'opérateur de façon plus explicite comme dans :

```
        OPERATOR        1       < (complex, complex) ,
```

mais il n'y a pas besoin de faire ainsi quand les opérateurs prennent le même type de donnée que celui pour lequel la classe d'opérateurs a été définie.

Les exemples ci-dessus supposent que vous voulez que cette nouvelle classe d'opérateur soit la classe d'opérateurs B-tree par défaut pour le type de donnée `complex`. Si vous ne voulez pas, supprimez simplement le mot `DEFAULT`.

38.16.5. Classes et familles d'opérateur

Jusqu'à maintenant, nous avons supposé implicitement qu'une classe d'opérateurs s'occupe d'un seul type de données. Bien qu'il ne puisse y avoir qu'un seul type de données dans une colonne d'index particulière, il est souvent utile d'indexer les opérations qui comparent une colonne indexée à une valeur d'un type de données différent. De plus, s'il est intéressant d'utiliser un opérateur inter-type en connexion avec une classe d'opérateurs, fréquemment cet autre type de donnée à sa propre classe d'opérateurs. Rendre explicite les connexions entre classes en relation est d'une grande aide pour que le planificateur optimise les requêtes SQL (tout particulièrement pour les classes d'opérateurs B-tree car le planificateur sait bien comme les utiliser).

Pour gérer ces besoins, PostgreSQL utilise le concept d'une *famille d'opérateur* . Une famille d'opérateur contient une ou plusieurs classes d'opérateurs et peut aussi contenir des opérateurs indexables et les fonctions de support correspondantes appartenant à la famille entière mais pas à une classe particulière de la famille. Nous disons que ces opérateurs et fonctions sont « lâches » à l'intérieur de la famille, en opposition à être lié à une classe spécifique. Typiquement, chaque classe d'opérateurs contient des opérateurs de types de données simples alors que les opérateurs inter-type sont lâches dans la famille.

Tous les opérateurs et fonctions d'une famille d'opérateurs doivent avoir une sémantique compatible où les pré-requis de la compatibilité sont dictés par la méthode indexage. Du coup, vous pouvez vous demander la raison pour s'embarrasser de distinguer les sous-ensembles de la famille en tant que classes d'opérateurs. En fait, dans beaucoup de cas, les divisions en classe sont inutiles et la famille est le seul groupe intéressant. La raison de la définition de classes d'opérateurs est qu'ils spécifient à quel point la famille est nécessaire pour supporter un index particulier. S'il existe un index utilisant une classe d'opérateurs, alors cette classe d'opérateur ne peut pas être supprimée sans supprimer l'index -- mais les autres parties de la famille d'opérateurs, donc les autres classes et les opérateurs lâches, peuvent être supprimées. Du coup, une classe d'opérateur doit être indiquée pour contenir l'ensemble minimum d'opérateurs et de fonctions qui sont raisonnablement nécessaire pour travailler avec un index sur un type de données spécifique, et ensuite les opérateurs en relation mais peuvent être ajoutés en tant que membres lâches de la famille d'opérateur.

Comme exemple, PostgreSQL a une famille d'opérateur B-tree interne `integer_ops`, qui inclut les classes d'opérateurs `int8_ops`, `int4_ops` et `int2_ops` pour les index sur les colonnes `bigint` (`int8`), `integer` (`int4`) et `smallint` (`int2`) respectivement. La famille contient aussi des opérateurs de comparaison inter-type permettant la comparaison de deux de ces types, pour qu'un index parmi ces types puisse être parcouru en utilisant une valeur de comparaison d'un autre type. La famille peut être dupliquée par ces définitions :

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
  -- standard int8 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ,
  FUNCTION 2 btint8sortsupport(internal) ,
  FUNCTION 3 in_range(int8, int8, int8, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
  -- standard int4 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint4cmp(int4, int4) ,
  FUNCTION 2 btint4sortsupport(internal) ,
  FUNCTION 3 in_range(int4, int4, int4, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ;

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
  -- standard int2 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint2cmp(int2, int2) ,
```

```
  FUNCTION 2 btint2sortsupport(internal) ,
  FUNCTION 3 in_range(int2, int2, int2, boolean, boolean) ,
  FUNCTION 4 btequalimage(oid) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
  -- cross-type comparisons int8 vs int2
  OPERATOR 1 < (int8, int2) ,
  OPERATOR 2 <= (int8, int2) ,
  OPERATOR 3 = (int8, int2) ,
  OPERATOR 4 >= (int8, int2) ,
  OPERATOR 5 > (int8, int2) ,
  FUNCTION 1 btint82cmp(int8, int2) ,

  -- cross-type comparisons int8 vs int4
  OPERATOR 1 < (int8, int4) ,
  OPERATOR 2 <= (int8, int4) ,
  OPERATOR 3 = (int8, int4) ,
  OPERATOR 4 >= (int8, int4) ,
  OPERATOR 5 > (int8, int4) ,
  FUNCTION 1 btint84cmp(int8, int4) ,

  -- cross-type comparisons int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- cross-type comparisons int4 vs int8
  OPERATOR 1 < (int4, int8) ,
  OPERATOR 2 <= (int4, int8) ,
  OPERATOR 3 = (int4, int8) ,
  OPERATOR 4 >= (int4, int8) ,
  OPERATOR 5 > (int4, int8) ,
  FUNCTION 1 btint48cmp(int4, int8) ,

  -- cross-type comparisons int2 vs int8
  OPERATOR 1 < (int2, int8) ,
  OPERATOR 2 <= (int2, int8) ,
  OPERATOR 3 = (int2, int8) ,
  OPERATOR 4 >= (int2, int8) ,
  OPERATOR 5 > (int2, int8) ,
  FUNCTION 1 btint28cmp(int2, int8) ,

  -- cross-type comparisons int2 vs int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ,

  -- cross-type in_range functions
  FUNCTION 3 in_range(int4, int4, int8, boolean, boolean) ,
  FUNCTION 3 in_range(int4, int4, int2, boolean, boolean) ,
  FUNCTION 3 in_range(int2, int2, int8, boolean, boolean) ,
  FUNCTION 3 in_range(int2, int2, int4, boolean, boolean) ;
```

Notez que cette définition « surcharge » la stratégie de l'opérateur et les numéros de fonction support : chaque numéro survient plusieurs fois dans la famille. Ceci est autorisé aussi longtemps que chaque instance d'un numéro particulier a des types de données distincts en entrée. Les instances qui ont les deux types en entrée égalent au type en entrée de la classe d'opérateurs sont les opérateurs primaires et les fonctions de support pour cette classe d'opérateurs et, dans la plupart des cas, doivent être déclarées comme membre de la classe d'opérateurs plutôt qu'en tant que membres lâches de la famille.

Dans une famille d'opérateur B-tree, tous les opérateurs de la famille doivent trier de façon compatible, comme c'est spécifié en détail dans Section 67.2. Pour chaque opérateur de la famille, il doit y avoir une fonction de support pour les deux mêmes types de données en entrée que celui de l'opérateur. Il est recommandé qu'une famille soit complète, c'est-à-dire que pour chaque combinaison de types de données, tous les opérateurs sont inclus. Chaque classe d'opérateurs doit juste inclure les opérateurs non inter-types et les fonctions de support pour ce type de données.

Pour construire une famille d'opérateurs de hachage pour plusieurs types de données, des fonctions de support de hachage compatibles doivent être créées pour chaque type de données supporté par la famille. Ici, compatibilité signifie que les fonctions sont garanties de renvoyer le même code de hachage pour toutes les paires de valeurs qui sont considérées égales par les opérateurs d'égalité de la famille, même quand les valeurs sont de type différent. Ceci est habituellement difficile à accomplir quand les types ont différentes représentations physiques, mais cela peut se faire dans la plupart des cas. De plus, convertir une valeur à partir d'un type de données représenté dans la famille d'opérateur vers un autre type de données aussi représenté dans la famille d'opérateur via une coercion implicite ou binaire ne doit pas changer la valeur calculée du hachage. Notez qu'il y a seulement une fonction de support par type de données, pas une par opérateur d'égalité. Il est recommandé qu'une famille soit terminée, c'est-à-dire fournit un opérateur d'égalité pour chaque combinaison de types de données. Chaque classe d'opérateurs doit inclure l'opérateur d'égalité non-inter-type et la fonction de support pour ce type de données.

Les index GIN, SP-GiST et GiST n'ont pas de notion explicite d'opérations inter-types. L'ensemble des opérateurs supportés est simplement ce que les fonctions de support primaire peuvent supporter pour un opérateur donné.

Dans BRIN, les pré-requis dépendent de l'ensemble de travail fourni par les classes d'opérateurs. Pour les classes basées sur `minmax`, le comportement requis est le même que pour les familles d'opérateur B-tree : tous les opérateurs d'une famille doivent avoir un tri compatible, et les conversions ne doivent pas changer l'ordre de tri associé.

> **Note**
>
> Avant PostgreSQL 8.3, le concept des familles d'opérateurs n'existait pas. Donc, tous les opérateurs inter-type dont le but était d'être utilisés avec un index étaient liés directement à la classe d'opérateurs de l'index. Bien que cette approche fonctionne toujours, elle est obsolète car elle rend trop importantes les dépendances de l'index et parce que le planificateur peut gérer des comparaisons inter-type avec plus d'efficacité que quand les typdes de données ont des opérateurs dans la même famille d'opérateur.

38.16.6. Dépendances du système pour les classes d'opérateurs

PostgreSQL utilise les classes d'opérateurs pour inférer les propriétés des opérateurs de plusieurs autres façons que le seul usage avec les index. Donc, vous pouvez créer des classes d'opérateur même si vous n'avez pas l'intention d'indexer une quelconque colonne de votre type de donnée.

En particulier, il existe des caractéristiques de SQL telles que `ORDER BY` et `DISTINCT` qui requièrent la comparaison et le tri des valeurs. Pour implémenter ces caractéristiques sur un type de donnée défini par l'utilisateur, PostgreSQL recherche la classe d'opérateurs B-tree par défaut pour le type de donnée.

Le membre « equals » de cette classe d'opérateurs définit pour le système la notion d'égalité des valeurs pour `GROUP BY` et `DISTINCT`, et le tri ordonné imposé par la classe d'opérateurs définit le `ORDER BY` par défaut.

S'il n'y a pas de classe d'opérateurs B-tree par défaut pour le type de donnée, le système cherchera une classe d'opérateurs de découpage. Mais puisque cette classe d'opérateurs ne fournit que l'égalité, il est seulement capable de supporter le regroupement mais pas le tri.

Quand il n'y a pas de classe d'opérateurs par défaut pour un type de donnée, vous obtenez des erreurs telles que « could not identify an ordering operator » si vous essayez d'utiliser ces caractéristiques SQL avec le type de donnée.

> **Note**
>
> Dans les versions de PostgreSQL antérieures à la 7.4, les opérations de tri et de groupement utilisaient implicitement les opérateurs nommés `=`, `<` et `>`. Le nouveau comportement qui repose sur les classes d'opérateurs par défaut évite d'avoir à faire une quelconque supposition sur le comportement des opérateurs avec des noms particuliers.

Trier par une classe d'opérateurs B-tree qui n'est pas celle par défaut est possible en précisant l'opérateur inférieur-à de la classe dans une option `USING`, par exemple

```
SELECT * FROM mytable ORDER BY somecol USING ~<~;
```

Sinon, préciser l'opérateur supérieur-à de la classe dans un `USING` sélectionne un tri par ordre décroissant.

La comparaison de tableaux d'un type de données utilisateur repose également sur la sémantique définie par la classe d'opérateurs B-tree par défaut du type. S'il n'y a pas de classe d'opérateurs B-tree par défaut, mais qu'il y a une classe d'opérateurs de type hash, alors l'égalité de tableau est supportée, mais pas la comparaison pour les tris.

Une autre fonctionnalité SQL qui nécessite une connaissance encore plus spécifique du type de données est l'option `RANGE` *`offset`* `PRECEDING`/`FOLLOWING` de fenêtre pour les fonctions de fenêtrage (voir Section 4.2.8). Pour une requête telle que

```
SELECT sum(x) OVER (ORDER BY x RANGE BETWEEN 5 PRECEDING AND 10
 FOLLOWING)
  FROM mytable;
```

il n'est pas suffisant de savoir comment trier par `x` ; la base de données doit également comprendre comment « soustraire 5 » ou « additionner 10 » à la valeur de `x` de la ligne courante pour identifier les limites de la fenêtre courante. Comparer les limites résultantes aux valeurs de `x` des autres lignes est possible en utilisant les opérateurs de comparaison fournis par la classe d'opérateurs B-tree qui définit le tri de l'`ORDER BY` -- mais les opérateurs d'addition et de soustraction ne font pas partie de la classe d'opérateurs, alors lesquels devraient être utilisés ? Inscrire en dur ce choix ne serait pas désirable, car différents ordres de tris (différentes classes d'opérateurs B-tree) pourraient nécessiter des comportements différents. Ainsi, une classe d'opérateurs B-tree peut préciser une fonction de support *in_range* qui encapsule les comportements d'addition et de soustraction faisant sens pour son ordre de tri. Elle peut même fournir plus d'une fonction de support in_range, s'il fait sens d'utiliser plus d'un type de données comme offset dans la clause `RANGE`. Si la classe d'opérateurs B-tree associée à la clause `ORDER BY` de la fenêtre n'a pas de fonction de support in_range correspondante, l'option `RANGE` *`offset`* `PRECEDING`/`FOLLOWING` n'est pas supportée.

Un autre point important est qu'un opérateur apparaissant dans une famille d'opérateur de hachage est un candidat pour les jointures de hachage, les agrégations de hachage et les optimisations relatives. La famille d'opérateur de hachage est essentiel ici car elle identifie le(s) fonction(s) de hachage à utiliser.

38.16.7. Opérateurs de tri

Certaines méthodes d'accès aux index (actuellement seulement GiST et SP-GiST) supportent le concept d'*opérateurs de tri*. Nous avons discuté jusqu'à maintenant d'*opérateurs de recherche*. Un opérateur de recherche est utilisable pour rechercher dans un index toutes les lignes satisfaisant le prédicat `WHERE` *`colonne_indexée operateur constante`*. Notez que rien n'est promis sur l'ordre dans lequel les lignes correspondantes seront renvoyées. Au contraire, un opérateur de tri ne restreint pas l'ensemble de lignes qu'il peut renvoyer mais, à la place, détermine leur ordre. Un opérateur de tri est utilisé pour que l'index puisse être parcouru pour renvoyer les lignes dans l'ordre représenté par `ORDER BY` *`colonne_indexée opérateur constante`*. Le but de définir des opérateurs de tri de cette façon est de supporter les recherches du type plus-proche-voisin si l'opérateur sait mesurer les distances. Par exemple, une requête comme

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT
 10;
```

trouve les dix emplacements les plus proches d'un point cible donné. Un index GiST sur la colonne location peut faire cela de façon efficace parce que `<->` est un opérateur de tri.

Bien que les opérateurs de recherche doivent renvoyer des résultats booléens, les opérateurs de tri renvoient habituellement d'autres types, tel que des float ou numeric pour les distances. Ce type n'est habituellement pas le même que le type de données indexé. Pour éviter les suppositions en dur sur le comportement des différents types de données, la définition d'un opérateur de tro doit nommer une famille d'opérateur B-tree qui spécifie l'ordre de tri du type de données résultant. Comme indiqué dans la section précédente, les familles d'opérateur B-tree définissent la notion de tri de PostgreSQL, donc c'est une représentation naturelle. Comme l'opérateur `<->` renvoie `float8`, il peut être indiqué dans la commande de création d'une classe d'opérateurs :

```
OPERATOR 15    <-> (point, point) FOR ORDER BY float_ops
```

où `float_ops` est la famille d'opérateur interne qui inclut les opérations sur `float8`. Cette déclaration indique que l'index est capable de renvoyer des lignes dans l'ordre de valeurs de plus en plus hautes de l'opérateur `<->`.

38.16.8. Caractéristiques spéciales des classes d'opérateurs

Il y a deux caractéristiques spéciales des classes d'opérateurs dont nous n'avons pas encore parlé, essentiellement parce qu'elles ne sont pas utiles avec les méthodes d'index les plus communément utilisées.

Normalement, déclarer un opérateur comme membre d'une classe ou d'une famille d'opérateur signifie que la méthode d'indexation peut retrouver exactement l'ensemble de lignes qui satisfait la condition `WHERE` utilisant cet opérateur. Par exemple :

```
SELECT * FROM table WHERE colonne_entier < 4;
```

peut être accompli exactement par un index B-tree sur la colonne entière. Mais il y a des cas où un index est utile comme un guide inexact vers la colonne correspondante. Par exemple, si un index GiST enregistre seulement les rectangles limite des objets géométriques, alors il ne peut pas exactement satisfaire une condition `WHERE` qui teste le chevauchement entre des objets non rectangulaires comme des polygones. Cependant, nous pourrions utiliser l'index pour trouver des objets dont les rectangles limites chevauchent les limites de l'objet cible. Dans ce cas, l'index est dit être à perte pour l'opérateur. Les recherches par index à perte sont implémentées en ayant une méthode d'indexage qui renvoie un drapeau *recheck* quand une ligne pourrait ou non satisfaire la condition de la requête. Le système principal testera ensuite la condition originale de la requête sur la ligne récupérée pour s'assurer que la correspondance est réelle. Cette approche fonctionne si l'index garantit de renvoyer toutes les lignes requises, ainsi que quelques lignes supplémentaires qui pourront être éliminées par la vérification. Les méthodes d'indexage qui supportent les recherches à perte (actuellement GiST, SP-GiST et GIN) permettent aux fonctions de support des classes individuelles d'opérateurs de lever le drapeau recheck, et donc c'est essentiellement une fonctionnalité pour les classes d'opérateurs.

Considérons à nouveau la situation où nous gardons seulement dans l'index le rectangle délimitant un objet complexe comme un polygone. Dans ce cas, il n'est pas très intéressant de conserver le polygone entier dans l'index - nous pouvons aussi bien conserver uniquement un objet simple du type `box`. Cette situation est exprimée par l'option `STORAGE` dans la commande `CREATE OPERATOR CLASS` : nous aurons à écrire quelque chose comme :

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

Actuellement, seules les méthodes d'indexation GiST, SP-GiST, GIN et BRIN supportent un type `STORAGE` qui soit différent du type de donnée de la colonne. Les routines d'appui de GiST pour la compression (`compress`) et la décompression (`decompress`) doivent s'occuper de la conversion du type de donnée quand `STORAGE` est utilisé. De la même manière, SP-GiST nécessite une fonction d'appui `compress` pour convertir le type de stockage quand il est différent ; si une classe d'opérateurs SP-GiST supporte aussi la récupération des données, la conversion inverse doit être gérée par la fonction `consistent`. Avec GIN, le type `STORAGE` identifie le type des valeurs « key », qui est normalement différent du type de la colonne indexée -- par exemple, une classe d'opérateurs pour des colonnes de tableaux d'entiers pourrait avoir des clés qui sont seulement des entiers. Les routines de support GIN `extractValue` et `extractQuery` sont responsables de l'extraction des clés à partir des valeurs indexées. BRIN est similaire à GIN : le type `STORAGE` identifie le type de valeurs résumées stockées, et les procédures de support des classes d'opérateurs sont responsables de l'interprétation correcte des valeurs résumées.

38.17. Empaqueter des objets dans une extension

Une extension à PostgreSQL utile contient généralement plusieurs objets SQL. Par exemple, un nouveau type de données va nécessiter de nouvelles fonctions, de nouveaux opérateurs et probablement de nouvelles méthodes d'indexation. Il peut être utile de les grouper en un unique paquetage pour simplifier la gestion des bases de données. Avec PostgreSQL, ces paquetages sont appelés *extension*. Pour créer une extension, vous avez besoin au minimum d'un *fichier de script* qui contient les commandes SQL permettant de créer ses objets, et un *fichier de contrôle* qui rapporte quelques propriétés de base de cette extension. Si cette extension inclut du code C, elle sera aussi généralement accompagnée d'une bibliothèque dans lequel le code C aura été compilé. Une fois ces fichiers en votre possession, un simple appel à la commande `CREATE EXTENSION` vous permettra de charger ses objets dans la base de données.

Le principal avantage des extensions n'est toutefois pas de pouvoir de charger une grande quantité d'objets dans votre base de donnée. Les extensions permettent en effet surtout à PostgreSQL de

comprendre que ces objets sont liés par cette extension. Vous pouvez par exemple supprimer tous ces objets avec une simple commande `DROP EXTENSION`. Il n'est ainsi pas nécessaire de maintenir un script de « désinstallation ». Plus utile encore, l'outil pg_dump saura reconnaître les objets appartenant à une extension et, plutôt que de les extraire individuellement, ajoutera simplement une commande `CREATE EXTENSION` à la sauvegarde. Ce mécanisme simplifie aussi la migration à une nouvelle version de l'extension qui peut contenir de nouveaux objets ou des objets différents de la version d'origine. Notez bien toutefois qu'il est nécessaire de disposer des fichiers de contrôles, de script, et autres pour permettre la restauration d'une telle sauvegarde dans une nouvelle base de donnée.

PostgreSQL ne vous laissera pas supprimer de manière individuelle les objets d'une extension sans supprimer l'extension tout entière. Aussi, bien que vous ayez la possibilité de modifier la définition d'un objet inclus dans une extension (par exemple via la commande `CREATE OR REPLACE FUNCTION` dans le cas d'une fonction), il faut garder en tête que cette modification ne sera pas sauvegardée par l'outil pg_dump. Une telle modification n'est en pratique raisonnable que si vous modifiez parallèlement le fichier de script de l'extension. Il existe toutefois des cas particuliers comme celui des tables qui contiennent des données de configuration (voir Section 38.17.3.) Dans les situations de production, il est généralement préférable de créer un script de mise à jour de l'extension pour réaliser les modifications sur les objets membres de l'extension.

Le script de l'extension peut mettre en place des droits sur les objets qui font partie de l'extension via les instructions `GRANT` et `REVOKE`. La configuration finale des droits pour chaque objet (si des droits sont à configurer) sera enregistrée dans le catalogue système `pg_init_privs`. Quand pg_dump est utilisé, la commande `CREATE EXTENSION` sera inclue dans la sauvegarde, suivi de la mise en place des instructions `GRANT` et `REVOKE` pour configurer les droits sur les objets, tels qu'ils étaient au moment où la sauvegarde a été faite.

PostgreSQL ne supporte pas l'exécution d'instructions `CREATE POLICY` et `SECURITY LABEL` par le script. Elles doivent être exécutées après la création de l'extension. Toutes les politiques RLS et les labels de sécurité placés sur les objets d'une extension seront inclus dans les sauvegardes créées par pg_dump.

Il existe aussi un mécanisme permettant de créer des scripts de mise à jour de la définition des objets SQL contenus dans une extension. Par exemple, si la version 1.1 d'une extension ajoute une fonction et change le corps d'une autre vis-à-vis de la version 1.0 d'origine, l'auteur de l'extension peut fournir un *script de mise à jour* qui effectue uniquement ces deux modifications. La commande `ALTER EXTENSION UPDATE` peut alors être utilisée pour appliquer ces changements et vérifier quelle version de l'extension est actuellement installée sur une base de donnée spécifiée.

Les catégories d'objets SQL qui peuvent être inclus dans une extension sont spécifiées dans la description de la commande `ALTER EXTENSION`. D'une manière générale, les objets qui sont communs à l'ensemble de la base ou du cluster, comme les bases de données, les rôles, les tablespaces ne peuvent être inclus dans une extension car une extension n'est référencée qu'à l'intérieur d'une base de donnée. À noter que rien n'empêche la création de fichier de script qui crée de tels objets, mais qu'ils ne seront alors pas considérés après leur création comme faisant partie de l'extension. À savoir en outre que bien que les tables puissent être incluses dans une extension, les objets annexes tels que les index ne sont pas automatiquement inclus dans l'extension et devront être explicitement mentionnés dans les fichiers de script.

Si un script d'extension créé n'importe quel objet temporaire (comme des tables temporaires), ces objets seront traités comme des membres de l'extension pour le reste de la session courante, mais seront automatiquement supprimés à la fin de la session, tout comme n'importe quel objet temporaire le serait également. C'est une exception à la règle que les objets membres d'une extension ne peuvent pas être supprimés sans supprimer l'intégralité de l'extension.

38.17.1. Fichiers des extensions

La commande `CREATE EXTENSION` repose sur un fichier de contrôle associé à chaque extension. Ce fichier doit avoir le même nom que l'extension suivi du suffixe `.control`, et doit être placé dans

le sous-répertoire SHAREDIR/extension du répertoire d'installation. Il doit être accompagné d'au moins un fichier de script SQL dont le nom doit répondre à la syntaxe *extension--version*.sql (par exemple, foo--1.0.sql pour la version 1.0 de l'extension foo). Par défaut, les fichiers de script sont eux-aussi situés dans le répertoire SHAREDIR/extension. Le fichier de contrôle peut toutefois spécifier un répertoire différent pour chaque fichier de script.

Le format du fichier de contrôle d'une extension est le même que pour le fichier postgresql.conf, à savoir une liste d'affectation *nom_paramètre* = *valeur* avec un maximum d'une affectation par ligne. Les lignes vides et les commentaires introduits par # sont eux-aussi autorisés. Prenez garde à placer entre guillemets les valeurs qui ne sont ni des nombres ni des mots isolés.

Un fichier de contrôle peut définir les paramètres suivants :

directory (string)

Le répertoire qui inclut les scripts SQL de l'extension. Si un chemin relatif est spécifié, le sous-répertoire SHAREDIR du répertoire d'installation sera choisi comme base. Le comportement par défaut de ce paramètre revient à le définir tel que directory = 'extension'.

default_version (string)

La version par défaut de l'extension, qui sera installée si aucune version n'est spécifiée avec la commande CREATE EXTENSION. Ainsi, bien que ce paramètre puisse ne pas être précisé, il reste recommandé de le définir pour éviter que la commande CREATE EXTENSION ne provoque une erreur en l'absence de l'option VERSION.

comment (string)

Un commentaire de type chaîne de caractère au sujet de l'extension. Le commentaire est appliqué à la création de l'extension, mais pas pendant les mises à jour de cette extension (car cela pourrait écraser des commentaires ajoutés par l'utilisateur). Une alternative consiste à utiliser la commande COMMENT dans le script de l'extension.

encoding (string)

L'encodage des caractères utilisé par les fichiers de script. Ce paramètre doit être spécifié si les fichiers de script contiennent des caractères non ASCII. Le comportement par défaut en l'absence de ce paramètre consiste à utiliser l'encodage de la base de donnée.

module_pathname (string)

La valeur de ce paramètre sera utilisée pour toute référence à MODULE_PATHNAME dans les fichiers de script. Si ce paramètre n'est pas défini, la substitution ne sera pas effectuée. La valeur $libdir/*nom_de_bibliothèque* lui est usuellement attribuée et dans ce cas, MODULE_PATHNAME est utilisé dans la commande CREATE FUNCTION concernant les fonctions en langage C, de manière à ne pas mentionner « en dur » le nom de la bibliothèque partagée.

requires (string)

Une liste de noms d'extension dont dépend cette extension, comme par exemple requires = 'foo, bar'. Ces extensions doivent être installées avant que l'extension puisse être installée.

superuser (boolean)

Si ce paramètre est à true (il s'agit de la valeur par défaut), seuls les superutilisateurs pourront créer cet extension ou la mettre à jour. Si ce paramètre est à false, seuls les droits nécessaires seront requis pour installer ou mettre à jour l'extension. Ce paramètre devrait normalement être mis à true si au moins une commandes du script nécessite les privilèges superuser. (Ces commandes échoueront en tous cas, mais il est plus convivial de fournir une erreur en amont.)

`trusted` (`boolean`)

Ce paramètre, si mis à `true` (qui n'est pas le défaut), permet à des utilisateurs standards, sans attribut `SUPERUSER`, d'installer une extension qui a `superuser` affecté à `true`. Plus précisemment, l'installation sera permise pour toute personne qui a le droit `CREATE` sur la base courante. Si l'utilisateur qui exécute `CREATE EXTENSION` n'est pas superutilisateur mais est autorisé à installer de part ce paramètre, alors le script d'installation ou de mise à jour est exécuté comme bootstrap superutilisateur, et non pas comme l'utilisateur appelant. Ce paramètre n'as pas d'effet si `superuser` est `false`. En général, le paramètre ne doit pas être mis à `true` pour les extensions qui peuvent autoriser des accès aux capacités réservées aux superutilisateurs, telles que les accès au système de fichiers.

`relocatable` (`boolean`)

Une extension est dite « déplaçable » (*relocatable*) s'il est possible de déplacer les objets qu'elle contient dans un schéma différent de celui attribué initialement par l'extension. La valeur par défaut est à `false`, ce qui signifie que l'extension n'est pas déplaçable. Voir Section 38.17.2 pour des informations complémentaires.

`schema` (`string`)

Ce paramètre ne peut être spécifié que pour les extensions non déplaçables. Il permet de forcer l'extension à charger ses objets dans le schéma spécifié et aucun autre. Le paramètre `schema` est uniquement consulté lors de la création initiale de l'extension, pas pendant ses mises à jour. Voir Section 38.17.2 pour plus d'informations.

En complément au fichier de contrôle *`extension`*`.control`, une extension peut disposer de fichiers de contrôle secondaires pour chaque version dont le nommage correspond à *`extension`*`--`*`version`*`.control`. Ces fichiers doivent se trouver dans le répertoire des fichiers de script de l'extension. Les fichiers de contrôle secondaires suivent le même format que le fichier de contrôle principal. Tout paramètre spécifié dans un fichier de contrôle secondaire surcharge la valeur spécifiée dans le fichier de contrôle principal concernant les installations ou mises à jour à la version considérée. Cependant, il n'est pas possible de spécifier les paramètres `directory` et `default_version` dans un fichier de contrôle secondaire.

Un fichier de script SQL d'une extension peut contenir toute commande SQL, à l'exception des commandes de contrôle de transaction (`BEGIN`, `COMMIT`, etc), et des commandes qui ne peuvent être exécutées au sein d'un bloc transactionnel (comme la commande `VACUUM`). Cette contrainte est liée au fait que les fichiers de script sont implicitement exécutés dans une transaction.

Les scripts SQL d'une extension peuvent aussi contenir des lignes commençant par `\echo`, qui seront ignorées (traitées comme des commentaires) par le mécanisme d'extension. Ceci est souvent utilisé pour renvoyer une erreur si le script est passé à psql plutôt qu'exécuter par `CREATE EXTENSION` (voir un script d'exemple dans Section 38.17.7). Sans cela, les utilisateurs pourraient charger accidentellement le contenu de l'extension sous la forme d'objets « autonomes » plutôt que faisant partie d'une extension, ce qui est assez pénible à corriger.

Si le script d'extension contient la chaine `@extowner@`, celle-ci est remplacé avec le nom (encadré de double apostrophe si nécessaire) de l'utilisateur exécutant `CREATE EXTENSION` ou `ALTER EXTENSION`. Habituellement, cette fonctionnalité est employée par les extensions notées de confiance pour affecter l'appropriation des objets sélectionnés par l'utilisateur exécutant plutôt que par le bootstrap superuser. (Il faut cependant être prudent avec cela. Par exemple, affecter à une fonction en langage C un propriétaire non superutilisateur ouvre une voie pour une escalade de privilèges à cet utilisateur.)

Bien que les fichiers de script puissent contenir n'importe quel caractère autorisé par l'encodage spécifié, les fichiers de contrôle ne peuvent contenir que des caractères ASCII non formatés. En effet, PostgreSQL ne peut pas déterminer l'encodage utilisé par les fichiers de contrôle. Dans la pratique, cela ne pose problème que dans le cas où vous voudriez utiliser des caractères non ASCII dans le commentaire de l'extension. Dans ce cas de figure, il est recommandé de ne pas utiliser le paramètre

comment du fichier de contrôle pour définir ce commentaire, mais plutôt la commande COMMENT ON EXTENSION dans un fichier de script.

38.17.2. Possibilités concernant le déplacement des extensions

Les utilisateurs souhaitent souvent charger les objets d'une extension dans un schéma différent de celui imposé par l'auteur. Trois niveaux de déplacement sont supportés :

- Une extension supportant complétement le déplacement peut être déplacé dans un autre schéma à tout moment, y compris après son chargement dans une base de donnée. Initialement, tous les objets de l'extension installée appartiennent à un premier schéma (excepté les objets qui n'appartiennent à aucun schéma comme les langages procéduraux). L'opération de déplacement peut alors être réalisée avec la commande ALTER EXTENSION SET SCHEMA, qui renomme automatiquement tous les objets de l'extension pour être intégrés dans le nouveau schéma. Le déplacement ne sera toutefois fonctionnel que si l'extension ne contient aucune référence de l'appartenance d'un de ses objets à un schéma. Dans ce cadre, il est alors possible de spécifier qu'une extension supporte complétement le déplacement en initialisant relocatable = true dans son fichier de contrôle.

- Une extension peut être déplaçable durant l'installation et ne plus l'être par la suite. Un exemple courant est celui du fichier de script de l'extension qui doit référencer un schéma cible de manière explicite pour des fonctions SQL, par exemple en définissant la propriété search_path. Pour de telles extensions, il faut définir relocatable = false dans son fichier de contrôle, et utiliser @extschema@ pour référencer le schéma cible dans le fichier de script. Toutes les occurences de cette chaîne dans le fichier de script seront remplacées par le nom du schéma choisi avant son exécution. Le nom du schéma choisi peut être fixé par l'option SCHEMA de la commande CREATE EXTENSION.

- Si l'extension ne permet pas du tout le déplacement, il faut définir relocatable = false dans le fichier de contrôle, mais aussi définir schema comme étant le nom du schéma cible. Cette précaution permettra d'empêcher l'usage de l'option SCHEMA de la commande CREATE EXTENSION, à moins que cette option ne référence la même valeur que celle spécifiée dans le fichier de contrôle. Ce choix est à priori nécessaire si l'extension contient des références à des noms de schéma qui ne peuvent être remplacés par @extschema@. À noter que même si son usage reste relativement limité dans ce cas de figure puisque le nom du schéma est alors fixé dans le fichier de contrôle, le mécanisme de substitution de @extschema@ reste toujours opérationnel.

Dans tous les cas, le fichier de script sera exécuté avec comme valeur de search_path le schéma cible. Cela signifie que la commande CREATE EXTENSION réalisera l'équivalent de la commande suivante :

```
SET LOCAL search_path TO @extschema@, pg_temp;
```

Cela permettra aux objets du fichier de script d'être créés dans le schéma cible. Le fichier de script peut toutefois modifier la valeur de search_path si nécessaire, mais cela n'est généralement pas le comportement souhaité. La variable search_path retrouvera sa valeur initiale à la fin de l'exécution de la commande CREATE EXTENSION.

Le schéma cible est déterminé par le paramètre schema dans le fichier de contrôle s'il est précisé, sinon par l'option SCHEMA de la commande CREATE EXTENSION si elle est spécifiée, sinon par le schéma de création par défaut actuel (le premier rencontré en suivant le chemin de recherche search_path de l'appelant). Quand le paramètre schema du fichier de contrôle est utilisé, le schéma cible sera créé s'il n'existe pas encore. Dans les autres cas, il devra exister au préalable.

Si des extensions requises sont définies par requires dans le fichier de contrôle, leur schéma cible est ajouté à la valeur initiale de search_path, d'après le schéma cible de la nouvelle extension. Cela permet à leurs objets d'être visibles dans le fichier de script de l'extension installée.

Pour des raisons de sécurité, `pg_temp` est ajouté automatiquement à la fin de `search_path` dans tous les cas.

Une extension peut contenir des objets répartis dans plusieurs schémas. Il est alors conseillé de regrouper dans un unique schéma l'ensemble des objets destinés à un usage externe à l'extension, qui sera alors le schéma cible de l'extension. Une telle organisation est compatible avec la définition par défaut de `search_path` pour la création d'extensions qui en seront dépendantes.

38.17.3. Tables de configuration des extensions

Certaines extensions incluent des tables de configuration, contenant des données qui peuvent être ajoutées ou changées par l'utilisateur après l'installation de l'extension. Normalement, si la table fait partie de l'extension, ni la définition de la table, ni son contenu ne sera sauvegardé par pg_dump. Mais ce comportement n'est pas celui attendu pour une table de configuration. Les données modifiées par un utilisateur nécessitent d'être sauvegardées, ou l'extension aura un comportement différent après rechargement.

Pour résoudre ce problème, un fichier de script d'extension peut marquer une table ou une séquence comme étant une relation de configuration, ce qui indiquera à pg_dump d'inclure le contenu de la table ou de la séquence (et non sa définition) dans la sauvegarde. Pour cela, il s'agit d'appeler la fonction `pg_extension_config_dump(regclass, text)` après avoir créé la table ou la séquence, par exemple

```
CREATE TABLE my_config (key text, value text);
CREATE SEQUENCE my_config_seq;

SELECT pg_catalog.pg_extension_config_dump('my_config', '');
SELECT pg_catalog.pg_extension_config_dump('my_config_seq', '');
```

Cette fonction permet de marquer autant de tables ou de séquences que nécessaire. Les séquences associées avec des colonnes de type `serial` ou `bigserial` peuvent être marquées ainsi.

Si le second argument de `pg_extension_config_dump` est une chaîne vide, le contenu entier de la table sera sauvegardé par l'application pg_dump. Cela n'est correct que si la table était initialement vide après l'installation du script. Si un mélange de données initiales et de données ajoutées par l'utilisateur est présent dans la table, le second argument de `pg_extension_config_dump` permet de spécifier une condition `WHERE` qui sélectionne les données à sauvegarder. Par exemple, vous pourriez faire

```
CREATE TABLE my_config (key text, value text, standard_entry
 boolean);

SELECT pg_catalog.pg_extension_config_dump('my_config', 'WHERE NOT
 standard_entry');
```

et vous assurer que la valeur de `standard_entry` soit true uniquement lorsque les lignes ont été créées par le script de l'extension.

Pour les séquences, le deuxième argument de `pg_extension_config_dump` n'a pas d'effet.

Des situations plus compliquées, comme des données initiales qui peuvent être modifiées par l'utilisateur, peuvent être prises en charge en créant des triggers sur la table de configuration pour s'assurer que les lignes ont été marquées correctement.

Vous pouvez modifier la condition du filtre associé avec une table de configuration en appelant de nouveau `pg_extension_config_dump`. (Ceci serait typiquement utile dans un script de mise à jour d'extension.) La seule façon de marquer une table est de la dissocier de l'extension avec la commande `ALTER EXTENSION ... DROP TABLE`.

Notez que les relations de clés étrangères entre ces tables dicteront l'ordre dans lequel les tables seront sauvegardées par pg_dump. Plus spécifiquement, pg_dump tentera de sauvegarder en premier la table référencé, puis la table référante. Comme les relations de clés étrangères sont configurées lors du CREATE EXTENSION (avant que les données ne soient chargées dans les tables), les dépendances circulaires ne sont pas gérées. Quand des dépendances circulaires existent, les données seront toujours sauvegardées mais ne seront pas restaurables directement. Une intervention de l'utilisateur sera nécessaire.

Les séquences associées avec des colonnes de type `serial` ou `bigserial` doivent être directement marquées pour sauvegarder leur état. Marquer la relation parent n'est pas suffisant pour ça.

38.17.4. Mise à jour d'extension

Un des avantages du mécanisme d'extension est de proposer un moyen simple de gérer la mise à jour des commandes SQL qui définissent les objets de l'extension. Cela est rendu possible par l'association d'un nom ou d'un numéro de version à chaque nouvelle version du script d'installation de l'extension. En complément, si vous voulez qu'un utilisateur soit capable de mettre à jour sa base de données dynamiquement d'une version à une autre, vous pouvez fournir *des scripts de mise à jour* qui feront les modifications nécessaires. Les scripts de mise à jour ont un nom qui correspond au format *`extension--ancienne_version--nouvelle_version`*`.sql` (par exemple, `foo--1.0--1.1.sql` contient les commandes pour modifier la version `1.0` de l'extension `foo` en la version `1.1`).

En admettant qu'un tel script de mise à jour soit disponible, la commande `ALTER EXTENSION UPDATE` mettra à jour une extension installée vers la nouvelle version spécifiée. Le script de mise à jour est exécuté dans le même environnement que celui que la commande `CREATE EXTENSION` fournit pour l'installation de scripts : en particulier, la variable `search_path` est définie de la même façon et tout nouvel objet créé par le script est automatiquement ajouté à l'extension. De plus, si le script choisit de supprimer des objets membres de l'extension, ils sont automatiquement dissociés de l'extension.

Si une extension a un fichier de contrôle secondaire, les paramètres de contrôle qui sont utilisés par un script de mise à jour sont ceux définis par le script de la version cible.

La commande `ALTER EXTENSION` peut exécuter des mises à jour en séquence pour réussir une mise à jour. Par exemple, si seuls les fichiers `foo--1.0--1.1.sql` et `foo--1.1--2.0.sql` sont disponibles, la commande `ALTER EXTENSION` les exécutera séquentiellement si une mise à jour vers la version `2.0` est demandée alors que la version `1.0` est installée.

PostgreSQL ne suppose rien au sujet des noms de version. Par exemple, il ne sait pas si `1.1` suit `1.0`. Il effectue juste une correspondance entre les noms de version et suit un chemin qui nécessite d'appliquer le moins de fichier de script possible. Un nom de version peut en réalité être toute chaîne qui ne contiendrait pas `--` ou qui ne commencerait ou ne finirait pas par `-`.

Il peut parfois être utile de fournir des scripts de retour en arrière, comme par exemple `foo--1.1--1.0.sql` pour autoriser d'inverser les modifications effectuées par la mise à jour en version `1.1`. Si vous procédez ainsi, ayez conscience de la possibilité laissée à PostgreSQL d'exécuter un tel script de retour en arrière s'il permet d'atteindre la version cible d'une mise à jour en un nombre réduit d'étapes. La cause du risque se trouve dans les scripts de mise à jour optimisés permettant de passer plusieurs versions en un seul script. La longueur du chemin commençant par un retour en arrière suivi d'un script optimisé pourrait être inférieure à la longueur du chemin qui monterait de version une par une. Si le script de retour en arrière supprime un objet irremplaçable, les conséquences pourraient en être facheuses.

Pour vérifier que vous ne serez pas confronté à des chemins de mise à jour inattendus, utilisez cette commande :

```
SELECT * FROM pg_extension_update_paths('extension_name');
```

Cette commande permet d'afficher chaque paire de noms de version connues pour l'extension spécifiée, ainsi que le chemin de mise à jour qui serait suivi depuis la version de départ jusque la version cible, ou la valeur `NULL` si aucun chemin valable n'est disponible. Le chemin est affiché sous une forme textuelle avec des séparateurs `--`. Vous pouvez utiliser `regexp_split_to_array(path,'--')` si vous préférez le format tableau.

38.17.5. Installer des extensions en utilisant des scripts de mise à jour

Une extension qui a existé un certain temps existera probabement dans plusieurs version, pour lesquelles l'auteur devra écrire des scripts de mise à jour. Par exemple, si vous avez sorti une extension `foo` dans les versions `1.0`, `1.1`, et `1.2`, il devrait exister les scripts de mise à jour `foo--1.0--1.1.sql` et `foo--1.1--1.2.sql`. Avant PostgreSQL 10, il était nécessaire de créer également de nouveaux fichiers de scripts `foo--1.1.sql` et `foo--1.2.sql` qui construisent directement les nouvelles versions de l'extension, ou sinon les nouvelles version ne pourraient pas être installées directement, mais uniquement en installant `1.0` puis en effectuant les mises à jour. C'était fastidieux et source de doublons, mais c'est maintenant inutile car `CREATE EXTENSION` peut suivre les chaînes de mise à jour automatiquement. Par exemple, si seuls les fichiers de script `foo--1.0.sql`, `foo--1.0--1.1.sql`, et `foo--1.1--1.2.sql` sont disponibles, alors une demande d'installation de la version `1.2` pourra être effectuée en lançant ces trois scripts les uns à la suite des autres. Le traitement est le même que si vous aviez d'abord installé `1.0` puis mis à jour vers `1.2`. (Comme pour `ALTER EXTENSION UPDATE`, si de multiples chemins sont disponibles alors le plus court sera choisi.) Arranger les fichiers de script d'une extension de cette façon peut réduire la quantité nécessaire d'effort de maintenance à fournir pour produires de petites mises à jour.

Si vous utilisez des fichiers de contrôle secondaires (spécifique à la version) avec une extension maintenant de cette façon, gardez à l'esprit que chaque version nécessite un fichier de contrôle même s'il n'y a pas de script d'installation spécifique pour cette version, car ce fichier de contrôle déterminera comment une mise à jour implicite vers cette version est effectuée. Par exemple, si `foo--1.0.control` spécifie `requires = 'bar'` mais que l'autre fichier de contrôle de `foo` ne le spécifie pas, la dépendance sur l'extension `bar` sera supprimée lors de la mise à jour de `1.0` vers une autre version.

38.17.6. Considérations de sécurité pour les extensions

Les extensions largement distribuées devraient assumer peu sur la base qu'elles occupent. De ce fait, il est adéquat d'écrire des fonctions fournies par une extension dans un style sécurisé qui ne peut pas être compromis par des attaques basées sur le search_path.

Une extension qui dispose de la propriété `superuser` configurée à true doit aussi considérer les risques de sécurité pour les actions effectuées par ses scripts d'installation et de mise à jour. Il n'est pas particulièrement compliqué pour un utilisateur mal intentionné de créer des objets chevaux de Troie qui compromettront une exécution ultérieure d'un script d'extension mal écrit, permettant à son utilisateur de gagner les droits d'un superutilisateur.

Des conseils sur l'écriture de fonctions sécurisées sont donnés dans Section 38.17.6.1 ci-dessous, et d'autres conseils, sur l'écriture de scripts d'installation sécurisés, sont donnés dans Section 38.17.6.2.

38.17.6.1. Considérations de sécurité pour les fonctions d'extensions

Les fonctions en langage SQL et PL fournies par les extensions peuvent être l'objet d'attaques basées sur le search_path quand elles sont exécutées car l'analyse de ces fonctions survient lors de leur exécution et non pas lors de leur création.

La page de référence de `CREATE FUNCTION` contient des conseils sur la bonne écriture de fonctions `SECURITY DEFINER`. Il est conseillé d'appliquer ces techniques pour toute fonction fournie par une extension car la fonction pourrait être appelée par un utilisateur avec des droits importants.

Si vous ne pouvez pas configurer le `search_path` pour contenir seulement les schémas sécurisés, supposez que chaque nom non qualifié pourrait désigner un objet défini par un utilisateur mal intentionné. Faites attention aux requêtes qui pourraient dépendre implicitement d'un `search_path` ; par exemple, `IN` et `CASE` *`expression`* `WHEN` sélectionnent toujours un opérateur utilisant le chemin de recherche. À la place, utilisez `OPERATOR(`*`schema`*`.=) ANY` et `CASE WHEN` *`expression`*.

Une extension standard ne devrait généralement pas supposer qu'elle a été installée dans un schéma sécurisé, ce qui signifie que même les références à ses propres objets en qualifiant leur nom de celui du schéma ne sont pas entièrement sans risque. Par exemple, si l'extension a défini une fonction `monschema.mafonction(bigint)`, alors un appel tel que `monschema.mafonction(42)` pourrait être capturée par une fonction hostile `monschema.mafonction(integer)`. Faites attention que les types de données de la fonction et les paramètres de l'opérateur correspondent exactement aux types d'argument déclarés, en utilisant des conversions explicites si nécessaire.

38.17.6.2. Considérations de sécurité pour les scripts d'extension

Un script d'installation ou de mise à jour d'extension devrait être écrit pour se garder contre les attaques se basant sur le schéma, survenant lors de l'exécution du script. Si la référence d'un objet dans le script peut se faire en résolvant un autre objet que celui voulu par l'auteur de script, une compromission peut survenir immédiatement ou ultérieurement quand l'object mal défini est utilisé.

Les commandes DDL telles que `CREATE FUNCTION` et `CREATE OPERATOR CLASS` sont généralement sécurisées, mais il convient de faire attention à toute commande ayant une expression standard comme composant. Par exemple, `CREATE VIEW` nécessite d'être validé, comme une expression `DEFAULT` dans `CREATE FUNCTION`.

Quelque fois, un script d'extension peut avoir besoin d'exécuter un SQL, par exemple pour réaliser des ajustements sur le catalogue qui ne seraient pas possibles via une DDL. Faites bien attention d'exécuter de telles commandes avec un `search_path` sécurisé ; ne faites *pas* confiance au chemin fourni par `CREATE/ALTER EXTENSION` comme étant sécurisé. Une meilleure approche est de configurer temporairement `search_path` à `pg_catalog, pg_temp` et d'insérer explicitement des références au schéma d'installation de l'expression si nécessaire. (Cette pratique peut être utile pour créer des vues.) Des exemples sont disponibles dans les modules `contrib` de la distribution des sources de PostgreSQL.

Les références entre extensions sont très difficiles à sécuriser complètement, en partie à cause de l'incertitude sur le schéma d'installation de l'autre extension. Ce risque est réduit si les deux extensions sont installées dans le même schéma parce que, dans ce cas, un objet hostile ne peut pas être placé avant l'extension référencée dans le `search_path` d'installation. Néanmoins, aucun mécanisme n'existe actuellement pour forcer cela.

38.17.7. Exemples d'extensions

Ci-après, un exemple complet d'une extension écrite uniquement en SQL, un type composite de deux éléments qui peut stocker n'importe quelle valeur dans chaque emplacement, qui sont nommés « k » et « v ». Les valeurs non textuelles sont automatiquement changées en texte avant stockage.

Le fichier de script `pair--1.0.sql` ressemble à ceci:

```
-- se plaint si le script est exécuté directement dans psql, plutôt
 que via CREATE EXTENSION
\echo Use "CREATE EXTENSION pair" to load this file. \quit

CREATE TYPE pair AS ( k text, v text );

CREATE pair(text, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1,
 $2)::@extschema@.pair;';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, FUNCTION =
 pair);

-- "SET search_path" is easy to get right, but qualified names
 perform better.
CREATE FUNCTION lower(pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW(lower($1.k), lower($1.v))::@extschema@.pair;'
SET search_path = pg_temp;

CREATE FUNCTION pair_concat(pair, pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW($1.k OPERATOR(pg_catalog.||) $2.k,
               $1.v OPERATOR(pg_catalog.||)
 $2.v)::@extschema@.pair;';
```

Le fichier de contrôle `pair.control` ressemble à ceci:

```
# extension pair
comment = 'Un type de donnees representant un couple clef/valeur'
default_version = '1.0'
# n'est pas déplaçable à cause de l'utilisation de @extschema@
relocatable = false
```

Si vous avez besoin d'un fichier d'installation pour installer ces deux fichiers dans le bon répertoire, vous pouvez utiliser le fichier `Makefile` qui suit :

```
EXTENSION = pair
DATA = pair--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Ce fichier d'installation s'appuye sur PGXS, qui est décrit dans Section 38.18. La commande `make install` va installer les fichiers de contrôle et de script dans le répertoire adéquat tel qu'indiqué par pg_config.

Une fois les fichiers installés, utilisez la commande `CREATE EXTENSION` pour charger les objets dans une base de donnée.

38.18. Outils de construction d'extension

Si vous comptez distribuer vos propres modules d'extension PostgreSQL, la mise en œuvre d'un système de construction multiplateforme sera réellement difficile. Cependant, PostgreSQL met à disposition des outils pour construire des extensions, appelés PGXS, permettant à de simples extensions d'être construites sur un serveur déjà installé. PGXS est principalement destiné aux extensions qui incluent du code C, bien qu'il puisse être utilisé aussi pour des extensions composées exclusivement de code SQL. PGXS n'a pas toutefois été conçu pour être un framework de construction universel qui pourrait construire tout logiciel s'interfaçant avec PostgreSQL. Il automatise simplement des règles de construction communes pour des extensions simples. Pour des paquetages plus complexes, vous aurez toujours besoin d'écrire vos propres systèmes de construction.

Pour utiliser le système PGXS pour votre extension, vous devez écrire un simple makefile. Dans ce makefile, vous devez définir plusieurs variables et inclure le makefile de PGXS. Voici un exemple qui construit une extension nommée `isbn_issn`, qui consiste en une bibliothèque qui contient du code C, un fichier de contrôle d'extension, un script SQL, un fichier d'en-tête (seulement nécessaire si les autres modules pourraient avoir besoin d'accéder aux fonctions de l'extension sans passer par le SQL) et une documentation texte :

```
MODULES = isbn_issn
EXTENSION = isbn_issn
DATA = isbn_issn--1.0.sql
DOCS = README.isbn_issn
HEADERS_isbn_issn = isbn_issn.h

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

Les trois dernières lignes devraient toujours être les mêmes. En début de fichier, vous pouvez assigner des variables ou ajouter des règles make personnalisées.

Définissez une de ces trois variables pour spécifier ce qui est construit :

`MODULES`

liste des bibliothèques à construire depuis les fichiers sources communs (ne pas inclure les suffixes de bibliothèques dans la liste)

`MODULE_big`

Une bibliothèque à construire depuis plusieurs fichiers source (listez les fichiers objets dans la variable `OBJS`).

`PROGRAM`

Un programme exécutable à construire (listez les fichiers objet dans la variable `OBJS`).

Les variables suivantes peuvent aussi être définies :

`EXTENSION`

Nom(s) de l'extension ; pour chaque nom, vous devez fournir un fichier *`extension`*`.control`, qui sera installé dans le répertoire *`prefix`*`/share/extension`

`MODULEDIR`

Sous-répertoire de *`prefix`*`/share` dans lequel les fichiers DATA et DOCS seront installés (s'il n'est pas défini, la valeur par défaut est `extension` si `EXTENSION` est défini et `contrib` dans le cas contraire)

`DATA`

Fichiers divers à installer dans *`prefix`*`/share/$MODULEDIR`

`DATA_built`

Fichiers divers à installer dans *`prefix`*`/share/$MODULEDIR`, qui nécessitent d'être construit au préalable

`DATA_TSEARCH`

Fichiers divers à installer dans *`prefix`*`/share/tsearch_data`

`DOCS`

Fichiers divers à installer dans *`prefix`*`/doc/$MODULEDIR`

`HEADERS_built`

Fichiers pour (en option construire et) installer sous *`prefix`*`/include/server/$MODULEDIR/$MODULE_big`.

Contrairement à `DATA_built`, les fichiers dans `HEADERS_built` ne sont pas supprimés par la cible `clean` ; si vous voulez les supprimer, ajoutez les aussi à `EXTRA_CLEAN` ou ajoutez vos propres règles pour le faire.

`HEADERS_built_$MODULE`

fichiers à installer (après la construction si indiqué) sous *`prefix`*`/include/server/$MODULEDIR/$MODULE`, où `$MODULE` doit être un nom de module utilisé dans `MODULES` ou `MODULE_big`.

Contrairement à `DATA_built`, les fichiers dans `HEADERS_built_$MODULE` ne sont pas supprimées par la cible `clean` ; si vous voulez les supprimer, ajoutez les aussi à `EXTRA_CLEAN` ou ajoutez vos propres règles pour le faire.

Il est autorisé d'utiliser les deux variables pour le même module ou toute combinaison, sauf si vous avez deux noms de module dans la liste `MODULES` qui diffèrent seulement par la présence d'un préfixe `built_`, qui causerait une ambiguité. Dans ce cas (peu) probable, vous devez utiliser seulement les variables `HEADERS_built_$MODULE`.

`SCRIPTS`

Fichiers de scripts (non binaires) à installer dans *`prefix`*`/bin`

`SCRIPTS_built`

Fichiers de script (non binaires) à installer dans *`prefix`*`/bin`, qui nécessitent d'être construit au préalable.

`REGRESS`

Liste de tests de regression (sans suffixe), voir plus bas

`REGRESS_OPTS`

Options supplémentaires à passer à pg_regress

`ISOLATION`

Liste de cas de tests d'isolation, voir ci-dessous pour plus de détails

`ISOLATION_OPTS`

Options supplémentaires pour réussir pg_isolation_regress

`TAP_TESTS`

Option définissant si les tests TAP doivent être exécutées, voir ci-dessous.

`NO_INSTALL`

Ne pas définir de cible `install`, utile pour les modules de test qui n'ont pas besoin que le produit soit installé

`NO_INSTALLCHECK`

Ne pas définir de cible `installcheck`, utile par exemple si les tests nécessitent une configuration spéciale, ou n'utilisent pas pg_regress

`EXTRA_CLEAN`

Fichiers supplémentaire à supprimer par la commande `make clean`

`PG_CPPFLAGS`

Sera ajouté au début de `CPPFLAGS`

`PG_CFLAGS`

Sera ajouté à `CFLAGS`

`PG_CXXFLAGS`

Sera ajouté à `CXXFLAGS`

`PG_LDFLAGS`

Sera ajouté au début de `LDFLAGS`

`PG_LIBS`

Sera ajouté à la ligne d'édition de lien de `PROGRAM`

`SHLIB_LINK`

Sera ajouté à la ligne d'édition de lien de `MODULE_big`

`PG_CONFIG`

Chemin vers le programme pg_config de l'installation de PostgreSQL pour laquelle construire la bibliothèque ou le binaire (l'utilisation de `pg_config` seul permet d'utiliser le premier accessible par votre `PATH`)

Placez ce fichier de construction comme `Makefile` dans le répertoire qui contient votre extension. Puis vous pouvez exécuter la commande `make` pour compiler, et ensuite `make install` pour déployer le module. Par défaut, l'extension est compilée et installée pour l'installation de PostgreSQL qui correspond au premier programme `pg_config` trouvé dans votre `PATH`. Vous pouvez utiliser une installation différente en définissant `PG_CONFIG` pour pointer sur le programme `pg_config` de votre choix, soit dans le fichier makefile, soit à partir de la ligne de commande de la commande `make`.

Vous pouvez aussi exécuter `make` dans un répertoire en dehors de l'arborescence des sources de votre extension, notamment si vous voulez séparer le répertoire de construction. Cette procédure est aussi appelée une construction *VPATH*. Voici comment :

```
mkdir build_dir
cd build_dir
make -f /path/to/extension/source/tree/Makefile
make -f /path/to/extension/source/tree/Makefile install
```

Autrement, vous pouvez configurer un répertoire pour une construction VPATH d'une façon similaire à ce qui est fait pour le code du moteur. Une façon de le faire revient à utiliser le script `config/prep_buildtree`. Une fois que cela est fait, vous pouvez lancer la construction en configurant la variable `VPATH` de `make` ainsi

```
make VPATH=/path/to/extension/source/tree
make VPATH=/path/to/extension/source/tree install
```

Cette procédure peut fonctionner avec une grande variété de disposition de répertoires.

Les scripts listés dans la variable `REGRESS` sont utilisés pour des tests de regression de votre module, qui peut être invoqué par `make installcheck` après avoir effectué `make install`. Pour que cela fonctionne, vous devez lancer le serveur PostgreSQL préalablement. Les fichiers de script listés dans la variable `REGRESS` doivent apparaître dans le sous-répertoire appelé `sql/` du répertoire de votre extension. Ces fichiers doivent avoir l'extension `.sql`, qui ne doit pas être inclus dans la liste `REGRESS` du makefile. Pour chaque test, il doit aussi y avoir un fichier qui contient les résultats attendus dans un sous-répertoire nommé `expected`, avec le même nom mais l'extension `.out`. La commande `make installcheck` exécute chaque script de test avec psql, et compare la sortie résultante au fichier de résultat correspondant. Toute différence sera écrite dans le fichier `regression.diffs` au format `diff -c`. Notez que l'exécution d'un test qui ne dispose pas des fichiers nécessaires sera rapportée comme une erreur dans le test, donc assurez-vous que tous les fichiers nécessaires soient présents.

Les scripts listés dans la variable `ISOLATION` sont utilisés pour des tests sur le comportement en cas de stress dû à des sessions concurrentes avec votre module, tests qui peuvent être invoqués par `make installcheck` après avoir exécuté `make install`. Pour que ceci fonctionne, vous devez avoir un serveur PostgreSQL fonctionnel. Les fichiers scripts listés dans `ISOLATION` doivent apparaître dans un sous-répertoire nommé `specs/` du répertoire de votre extension. Ces fichiers doivent avoir une extension `.spec`, qui ne doit pas être incluse dans la liste `ISOLATION` du makefile. Pour chaque test, il doit aussi y avoir un fichier contenant la sortie attendue dans un sous-répertoire nommé `expected/`, avec le même nom et une extension `.out`. `make installcheck` exécute chaque script de test et compare la sortie résultante au fichier correspondant attendu. Toute différence sera écrite dans le fichier `output_iso/regression.diffs` au format `diff -c`. Notez qu'essayer d'exécuter un test dont le fichier attendu manque sera rapporté comme un problème, donc assurez-vous que vous avez tous les fichiers attendus.

`TAP_TESTS` active l'utilisation des tests TAP. Les données de chaque exécution sont présentes dans un sous-répertoire nommé `tmp_check/`. Voir aussi Section 33.4 pour plus de détails.

> **Astuce**
>
> Le moyen le plus simple de créer les fichiers nécessaires est de créer des fichiers vides, puis d'effectuer un jeu d'essai (qui bien sûr retournera des anomalies). Étudiez les résultats trouvés dans le répertoire `results` (for tests in `REGRESS`), or `output_iso/results/` directory (for tests in `ISOLATION`), et copiez-les dans le répertoire `expected/` s'ils correspondent à ce que vous attendiez du test correspondant.

Chapitre 39. Triggers (triggers)

Ce chapitre fournit des informations générales sur l'écriture des fonctions pour trigger. Ces fonctions peuvent être écrites dans la plupart des langages de procédure disponibles incluant PL/pgSQL (Chapitre 43), PL/Tcl (Chapitre 44), PL/Perl (Chapitre 45) et PL/Python (Chapitre 46). Après avoir lu ce chapitre, vous devriez consulter le chapitre sur votre langage de procédure favori pour découvrir les spécificités de l'écriture de triggers dans ce langage.

Il est aussi possible d'écrire une fonction trigger en C, bien que la plupart des gens trouvent plus facile d'utiliser un des langages de procédure. Il est actuellement impossible d'écrire une fonction trigger en simple SQL.

39.1. Aperçu du comportement des triggers

Un trigger spécifie que la base de données doit exécuter automatiquement une fonction donnée chaque fois qu'un certain type d'opération est exécuté. Les fonctions trigger peuvent être attachées à une table (partitionnée ou non), une vue ou une table distante.

Sur des tables et tables distantes, les triggers peuvent être définies pour s'exécuter avant ou après une commande `INSERT`, `UPDATE` ou `DELETE`, soit une fois par ligne modifiée, soit une fois par expression SQL. Les triggers `UPDATE` peuvent en plus être configurées pour n'être déclenchés que si certaines colonnes sont mentionnées dans la clause `SET` de l'instruction `UPDATE`. Les triggers peuvent aussi se déclencher pour des instructions `TRUNCATE`. Si un événement d'un trigger intervient, la fonction du trigger est appelée au moment approprié pour gérer l'événement.

Des triggers peuvent être définies sur des vues pour exécuter des opérations à la place des commandes `INSERT`, `UPDATE` ou `DELETE`. Les triggers `INSTEAD OF` sont déclenchés une fois par ligne devant être modifiée dans la vue. C'est de la responsabilité de la fonction trigger de réaliser les modifications nécessaires pour que les tables de base sous-jacentes d'une vue et, si approprié, de renvoyer la ligne modifiée comme elle apparaîtra dans la vue. Les triggers sur les vues peuvent aussi être définis pour s'exécuter une fois par requête SQL statement, avant ou après des opérations `INSERT`, `UPDATE` ou `DELETE`. Néanmoins, de tels triggers sont déclenchés seulement s'il existe aussi un trigger `INSTEAD OF` sur la vue. Dans le cas contraire, toute requête ciblant la vue doit être réécrite en une requête affectant sa (ou ses) table(s) de base. Les triggers déclenchés seront ceux de(s) table(s) de base.

La fonction trigger doit être définie avant que le trigger lui-même puisse être créé. La fonction trigger doit être déclarée comme une fonction ne prenant aucun argument et retournant un type `trigger` (la fonction trigger reçoit ses entrées via une structure `TriggerData` passée spécifiquement, et non pas sous la forme d'arguments ordinaires de fonctions).

Une fois qu'une fonction trigger est créée, le trigger est créé avec CREATE TRIGGER. La même fonction trigger est utilisable par plusieurs triggers.

PostgreSQL offre des triggers *par ligne* et *par instruction*. Avec un trigger en mode ligne, la fonction du trigger est appelée une fois pour chaque ligne affectée par l'instruction qui a lancé le trigger. Au contraire, un trigger en mode instruction n'est appelé qu'une seule fois lorsqu'une instruction appropriée est exécutée, quel que soit le nombre de lignes affectées par cette instruction. En particulier, une instruction n'affectant aucune ligne résultera toujours en l'exécution de tout trigger en mode instruction applicable. Ces deux types sont parfois appelés respectivement des *triggers niveau ligne* et des *triggers niveau instruction*. Les triggers sur `TRUNCATE` peuvent seulement être définis au niveau instruction, et non pas au niveau ligne.

Les triggers sont aussi classifiées suivant qu'ils se déclenchent avant (*before*), après (*after*) ou à la place (*instead of*) de l'opération. Ils sont référencés respectivement comme des triggers `BEFORE`, `AFTER` et `INSTEAD OF`. Les triggers `BEFORE` au niveau requête se déclenchent avant que la requête ne commence quoi que ce soit alors que les triggers `AFTER` au niveau requête se déclenchent tout à la fin de la requête. Ces types de triggers peuvent être définis sur les tables, vues et tables externes. Les triggers `BEFORE` au niveau ligne se déclenchent immédiatement avant l'opération sur une ligne

particulière alors que les triggers AFTER au niveau ligne se déclenchent à la fin de la requête (mais avant les triggers AFTER au niveau requête). Ces types de triggers peuvent seulement être définis sur les tables et sur les tables distantes, et non pas sur les vues. Les triggers INSTEAD OF peuvent seulement être définis sur des vues, et seulement au niveau ligne. Ils se déclenchent immédiatement pour chaque ligne de la vue identifiée comme nécessitant une action.

L'exécution d'un trigger AFTER peut être reporté à la fin de la transaction, plutôt qu'à la fin de la requête, s'il a été défini comme un *trigger de contrainte*. Dans tous les cas, un trigger est exécuté comme faisant partie de la même transaction que la requête qui l'a exécuté, donc si soit la requête soit le trigger renvoie une erreur, l'effet sera une annulation par ROLLBACK.

Une instruction qui cible une table parent dans un héritage ou une hiérarchie de partitionnement ne cause pas le déclenchement des tiggers au niveau requête des tables filles affectées. Seuls les triggers au niveau requête de la table parent sont déclenchés. Néanmoins, les triggers niveau ligne de toute table fille affecté seront déclenchés.

Si une commande INSERT contient une clause ON CONFLICT DO UPDATE, il est possible que les effets des triggers niveau ligne BEFORE INSERT et BEFORE UPDATE puissent être tous les deux appliqués de telle sorte que leurs effets soient visibles dans la version finale de la ligne mise à jour, si une colonne EXCLUDED est référencée. Il n'est néanmoins pas nécessaire qu'il soit fait référence à une colonne EXCLUDED pour que les deux types de triggers BEFORE s'exécutent tout de même. La possibilité d'avoir des résultats surprenants devrait être prise en compte quand il existe des triggers niveau ligne BEFORE INSERT et BEFORE UPDATE qui tous les deux modifient la ligne sur le point d'être insérée ou mise à jour (ceci peut être problématique si les modifications sont plus ou moins équivalentes et si elles ne sont pas idempotente). Notez que les triggers UPDATE niveau instruction sont exécutés lorsque la clause ON CONFLICT DO UPDATE est spécifiée, quand bien même aucune ligne ne serait affectée par la commande UPDATE (et même si la commande UPDATE n'est pas exécutée). Une commande INSERT avec une clause ON CONFLICT DO UPDATE exécutera d'abord les triggers niveau instruction BEFORE INSERT, puis les triggers niveau instruction BEFORE UPDATE, suivis par les triggers niveau instruction AFTER UPDATE, puis finalement les triggers niveau instruction AFTER INSERT.

Si un UPDATE sur une table partitionnée implique le déplacement d'une ligne vers une autre partition, il sera réalisé comme un DELETE de la partition originale, suivi d'un INSERT dans la nouvelle partition. Dans ce cas, les triggers BEFORE UPDATE niveau ligne et tous les triggers BEFORE DELETE niveau ligne sont déclenchés sur la partition originale. Puis tous les triggers BEFORE INSERT niveau ligne sont déclenchés sur la partition destination. La possibilité de résultats surprenants doit être considéré quand tous les triggers affectent la ligne déplacée. En ce qui concerne les triggers AFTER ROW, les triggers AFTER DELETE et AFTER INSERT sont appliqués mais les triggers AFTER UPDATE ne le sont pas car UPDATE a été convertis en un DELETE et un INSERT. Quant aux triggers niveau instruction, aucun des triggers DELETE et INSERT ne sont déclenchés, y compris en cas de déplacement de lignes. Seuls les triggers UPDATE définis sur la table cible utilisée dans une instruction UPDATE seront déclenchés.

Aucun trigger spécifique n'est défini pour MERGE. À la place, des triggers niveau instruction ou niveau ligne sont déclenchés pour les instructions UPDATE, DELETE et INSERT suivant l'action indiquée dans la requête MERGE (pour les triggers niveau instruction) et suivant les actions réellement exécutées (pour les triggers niveau instruction).

Lors de l'exécution d'une commande MERGE, les triggers BEFORE and AFTER au niveau instruction sont déclenchées pour les événements spécifiés dans les actions de la commande MERGE, que l'action soit exécutée ou non au final. C'est identique à l'instruction UPDATE qui ne met à jour aucune ligne, mais pour laquelle, néanmoins, les triggers niveau instruction ont été exécutés. Les triggers niveau ligne sont déclenchés uniquement quand une ligne est réellement mise à jour, insérée ou supprimée. Donc il est parfaitement normal que, bien que les triggers niveau instruction soient déclenchés pour certains types d'action, les triggers niveau ligne ne le soient pas pour les mêmes actions.

Les fonctions triggers appelées par des triggers niveau instruction devraient toujours renvoyer NULL. Les fonctions triggers appelées par des triggers niveau ligne peuvent renvoyer une ligne de la table

(une valeur de type HeapTuple) vers l'exécuteur appelant, s'ils le veulent. Un trigger niveau ligne exécuté avant une opération a les choix suivants :

- Il peut retourner un pointeur NULL pour sauter l'opération pour la ligne courante. Ceci donne comme instruction à l'exécuteur de ne pas exécuter l'opération niveau ligne qui a lancé le trigger (l'insertion, la modification ou la suppression d'une ligne particulière de la table).
- Pour les triggers INSERT et UPDATE de niveau ligne uniquement, la valeur de retour devient la ligne qui sera insérée ou remplacera la ligne en cours de mise à jour. Ceci permet à la fonction trigger de modifier la ligne en cours d'insertion ou de mise à jour.

Un trigger BEFORE niveau ligne qui ne serait pas conçu pour avoir l'un de ces comportements doit prendre garde à retourner la même ligne que celle qui lui a été passée comme nouvelle ligne (c'est-à-dire : pour des triggers INSERT et UPDATE : la nouvelle (NEW) ligne, et pour les triggers DELETE) : l'ancienne (OLD) ligne .

Un trigger INSTEAD OF niveau ligne devrait renvoyer soit NULL pour indiquer qu'il n'a pas modifié de données des tables de base sous-jacentes de la vue, soit la ligne de la vue qui lui a été passé (la ligne NEW pour les opérations INSERT et UPDATE, ou la ligne OLD pour l'opération DELETE). Une valeur de retour différent de NULL est utilisée comme signal indiquant que le trigger a réalisé les modifications de données nécessaires dans la vue. Ceci causera l'incrémentation du nombre de lignes affectées par la commande. Pour les opérations INSERT et UPDATE seulement, le trigger peut modifier la ligne NEW avant de la renvoyer. Ceci modifiera les données renvoyées par INSERT RETURNING ou UPDATE RETURNING, et est utile quand la vue n'affichera pas exactement les données fournies.

La valeur de retour est ignorée pour les triggers niveau ligne lancés après une opération. Ils peuvent donc renvoyer la valeur NULL.

Certaines considérations s'appliquent pour les colonnes générées. Les colonnes générées sont calculées après les triggers BEFORE et avant les triggers AFTER. De ce fait, la valeur générée peut être inspectée dans les triggers AFTER. Dans les triggers BEFORE, la ligne OLD contient l'ancienne valeur générée, comme on pourrait s'y attendre, mais la ligne NEW ne contient pas encore la nouvelle valeur générée et ne doit pas être accédée. Dans l'interface en langage C, le contenu de la colonne est non défini à ce moment ; un langage de programmation de plus haut niveau doit empêcher l'accès à une colonne générée dans la ligne NEW pour un trigger BEFORE. Les modifications de la valeur d'une colonne générée dans un trigger BEFORE sont ignorées et seront écrasées.

Si plus d'un trigger est défini pour le même événement sur la même relation, les triggers seront lancés dans l'ordre alphabétique de leur nom. Dans le cas de triggers BEFORE et INSTEAD OF, la ligne renvoyée par chaque trigger, qui a éventuellement été modifiée, devient l'argument du prochain trigger. Si un des triggers BEFORE ou INSTEAD OF renvoie un pointeur NULL, l'opération est abandonnée pour cette ligne et les triggers suivants ne sont pas lancés (pour cette ligne).

Une définition de trigger peut aussi spécifier une condition booléenne WHEN qui sera testée pour savoir si le trigger doit bien être déclenché. Dans les triggers de niveau ligne, la condition WHEN peut examiner l'ancienne et la nouvelle valeur des colonnes de la ligne. (les triggers de niveau instruction peuvent aussi avoir des conditions WHEN mais cette fonctionnalité est moins intéressante pour elles). Dans un trigger *avant*, la condition WHEN est évaluée juste avant l'exécution de la fonction, donc l'utilisation de WHEN n'est pas réellement différente du test de la même condition au début de la fonction trigger. Néanmoins, dans un tigger AFTER, la condition WHEN est évaluée juste avant la mise à jour de la ligne et détermine si un événement va déclencher le trigger à la fin de l'instruction. Donc, quand la condition WHEN d'un trigger AFTER ne renvoie pas true, il n'est pas nécessaire de mettre en queue un événement ou de récupérer de nouveau la ligne à la fin de l'instruction. Ceci permet une amélioration conséquente des performances pour les instructions qui modifient un grand nombre de lignes si le trigger a seulement besoin d'être exécuté que sur quelques lignes. Les triggers INSTEAD OF n'acceptent pas les conditions WHEN.

Les triggers BEFORE en mode ligne sont typiquement utilisés pour vérifier ou modifier les données qui seront insérées ou mises à jour. Par exemple, un trigger BEFORE pourrait être utilisé pour insérer l'heure actuelle dans une colonne de type timestamp ou pour vérifier que deux éléments d'une ligne

sont cohérents. Les triggers AFTER en mode ligne sont pour la plupart utilisés pour propager des mises à jour vers d'autres tables ou pour réaliser des tests de cohérence avec d'autres tables. La raison de cette division du travail est qu'un trigger AFTER peut être certain qu'il voit la valeur finale de la ligne alors qu'un trigger BEFORE ne l'est pas ; il pourrait exister d'autres triggers BEFORE qui seront exécutés après lui. Si vous n'avez aucune raison spéciale pour le moment du déclenchement, le cas BEFORE est plus efficace car l'information sur l'opération n'a pas besoin d'être sauvegardée jusqu'à la fin du traitement.

Si une fonction trigger exécute des commandes SQL, alors ces commandes peuvent lancer à leur tour des triggers. On appelle ceci un trigger en cascade. Il n'y a pas de limitation directe du nombre de niveaux de cascade. Il est possible que les cascades causent un appel récursif du même trigger ; par exemple, un trigger INSERT pourrait exécuter une commande qui insère une ligne supplémentaire dans la même table, entraînant un nouveau lancement du trigger INSERT. Il est de la responsabilité du programmeur d'éviter les récursions infinies dans de tels scénarios.

Quand un trigger est défini, des arguments peuvent être spécifiés pour lui. L'objectif de l'inclusion d'arguments dans la définition du trigger est de permettre à différents triggers ayant des exigences similaires d'appeler la même fonction. Par exemple, il pourrait y avoir une fonction trigger généralisée qui prend comme arguments deux noms de colonnes et place l'utilisateur courant dans l'une et un horodatage dans l'autre. Correctement écrit, cette fonction trigger serait indépendante de la table particulière sur laquelle il se déclenche. Ainsi, la même fonction pourrait être utilisée pour des événements INSERT sur n'importe quelle table ayant des colonnes adéquates, pour automatiquement suivre les créations d'enregistrements dans une table de transactions par exemple. Elle pourrait aussi être utilisée pour suivre les dernières mises à jour si elle est définie comme un trigger UPDATE.

Chaque langage de programmation supportant les triggers à sa propre méthode pour rendre les données en entrée disponible à la fonction du trigger. Cette donnée en entrée inclut le type d'événement du trigger (c'est-à-dire INSERT ou UPDATE) ainsi que tous les arguments listés dans CREATE TRIGGER. Pour un trigger niveau ligne, la donnée en entrée inclut aussi la ligne NEW pour les triggers INSERT et UPDATE et/ou la ligne OLD pour les triggers UPDATE et DELETE.

Par défaut, les triggers niveau instruction n'ont aucun moyen d'examiner le ou les lignes individuelles modifiées par la requête. Mais un trigger AFTER STATEMENT peut demander que des *tables de transition* soient créées pour rendre disponible les ensembles de lignes affectées au trigger. AFTER ROW peut aussi demander les tables de transactions, pour accéder au changement global dans la table, ainsi qu'au changement de lignes individuelles pour lesquels ils ont été déclenchés. La méthode d'examen des tables de transition dépend là-aussi du langage de programmation utilisé mais l'approche typique est de transformer les tables de transition en tables temporaires en lecture seule pouvant être accédées par des commandes SQL lancées par la fonction trigger.

39.2. Visibilité des modifications des données

Si vous exécutez des commandes SQL dans votre fonction SQL et que ces commandes accèdent à la table pour laquelle vous créez ce trigger, alors vous avez besoin de connaître les règles de visibilité des données car elles déterminent si les commandes SQL voient les modifications de données pour lesquelles est exécuté le trigger. En bref :

- Les triggers niveau instruction suivent des règles de visibilité simples : aucune des modifications réalisées par une instruction n'est visible aux triggers niveau instruction appelés avant l'instruction alors que toutes les modifications sont visibles aux triggers AFTER niveau instruction.
- Les modifications de données (insertion, mise à jour ou suppression) lançant le trigger ne sont naturellement *pas* visibles aux commandes SQL exécutées dans un trigger BEFORE en mode ligne parce qu'elles ne sont pas encore survenues.
- Néanmoins, les commandes SQL exécutées par un trigger BEFORE en mode ligne *verront* les effets des modifications de données pour les lignes précédemment traitées dans la même commande

externe. Ceci requiert une grande attention car l'ordre des événements de modification n'est en général pas prévisible ; une commande SQL affectant plusieurs lignes pourrait visiter les lignes dans n'importe quel ordre.

- De façon similaire, un trigger niveau ligne de type `INSTEAD OF` verra les effets des modifications de données réalisées par l'exécution des autres triggers `INSTEAD OF` dans la même commande.
- Quand un trigger `AFTER` en mode ligne est exécuté, toutes les modifications de données réalisées par la commande externe sont déjà terminées et sont visibles par la fonction appelée par le trigger.

Si votre fonction trigger est écrite dans un des langages de procédures standard, alors les instructions ci-desus s'appliquent seulement si la fonction est déclarée `VOLATILE`. Les fonctions déclarées `STABLE` ou `IMMUTABLE` ne verront pas les modifications réalisées par la commande appelante dans tous les cas.

Il existe plus d'informations sur les règles de visibilité des données dans la Section 47.5. L'exemple dans la Section 39.4 contient une démonstration de ces règles.

39.3. Écrire des fonctions triggers en C

Cette section décrit les détails de bas niveau de l'interface d'une fonction trigger. Ces informations ne sont nécessaires que lors de l'écriture d'une fonction trigger en C. Si vous utilisez un langage de plus haut niveau, ces détails sont gérés pour vous. Dans la plupart des cas, vous devez considérer l'utilisation d'un langage de procédure avant d'écrire vos triggers en C. La documentation de chaque langage de procédures explique comment écrire un trigger dans ce langage.

Les fonctions triggers doivent utiliser la « version 1 » de l'interface du gestionnaire de fonctions.

Quand une fonction est appelée par le gestionnaire de trigger, elle ne reçoit aucun argument classique, mais un pointeur de « contexte » pointant sur une structure `TriggerData`. Les fonctions C peuvent vérifier si elles sont appelées par le gestionnaire de triggers ou pas en exécutant la macro :

```
CALLED_AS_TRIGGER(fcinfo)
```

qui se décompose en :

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

Si elle retourne la valeur vraie, alors il est bon de convertir `fcinfo->context` en type `TriggerData *` et de faire usage de la structure pointée `TriggerData`. La fonction *ne* doit *pas* modifier la structure `TriggerData` ou une donnée quelconque vers laquelle elle pointe.

`struct TriggerData` est définie dans `commands/trigger.h` :

```
typedef struct TriggerData
{
    NodeTag          type;
    TriggerEvent     tg_event;
    Relation         tg_relation;
    HeapTuple        tg_trigtuple;
    HeapTuple        tg_newtuple;
    Trigger         *tg_trigger;
    TupleTableSlot  *tg_trigslot;
    TupleTableSlot  *tg_newslot;
    Tuplestorestate *tg_oldtable;
    Tuplestorestate *tg_newtable;
    const Bitmapset *tg_updatedcols;
} TriggerData;
```

où les membres sont définis comme suit :

`type`

Toujours `T_TriggerData`.

`tg_event`

Décrit l'événement pour lequel la fonction est appelée. Vous pouvez utiliser les macros suivantes pour examiner `tg_event` :

`TRIGGER_FIRED_BEFORE(tg_event)`

Renvoie vrai si le trigger est lancé avant l'opération.

`TRIGGER_FIRED_AFTER(tg_event)`

Renvoie vrai si le trigger est lancé après l'opération.

`TRIGGER_FIRED_INSTEAD(tg_event)`

Renvoie vrai si le trigger a été lancé à la place de l'opération.

`TRIGGER_FIRED_FOR_ROW(tg_event)`

Renvoie vrai si le trigger est lancé pour un événement en mode ligne.

`TRIGGER_FIRED_FOR_STATEMENT(tg_event)`

Renvoie vrai si le trigger est lancé pour un événement en mode instruction.

`TRIGGER_FIRED_BY_INSERT(tg_event)`

Retourne vrai si le trigger est lancé par une commande `INSERT`.

`TRIGGER_FIRED_BY_UPDATE(tg_event)`

Retourne vrai si le trigger est lancé par une commande `UPDATE`.

`TRIGGER_FIRED_BY_DELETE(tg_event)`

Retourne vrai si le trigger est lancé par une commande `DELETE`.

`TRIGGER_FIRED_BY_TRUNCATE(tg_event)`

Renvoie true si le trigger a été déclenché par une commande `TRUNCATE`.

`tg_updatedcols`

Pour les triggers `UPDATE`, un champ bitmap indique les colonnes mises à jour par la commande qui a déclenché le trigger. Les fonctions des triggers génériques peuvent utiliser ce champ pour optimiser les actions en n'ayant pas à s'occuper des colonnes qui n'ont pas été modifiées.

Par exemple, pour déterminer si la colonne de numéro d'attribut `attnum` (commençant à 1) est un membre de ce champ, appelez `bms_is_member(attnum - FirstLowInvalidHeapAttributeNumber, trigdata->tg_updatedcols)`.

Pour les triggers autres que ceux en `UPDATE`, ce champ sera à la valeur `NULL`.

`tg_relation`

Un pointeur vers une structure décrivant la relation pour laquelle le trigger est lancé. Voir `utils/reltrigger.h` pour les détails de cette structure. Les choses les plus intéressantes

sont `tg_relation->rd_att` (descripteur de nuplets de la relation) et `tg_relation->rd_rel->relname` (nom de la relation ; le type n'est pas `char*` mais `NameData` ; utilisez `SPI_getrelname(tg_relation)` pour obtenir un `char*` si vous avez besoin d'une copie du nom).

`tg_trigtuple`

Un pointeur vers la ligne pour laquelle le trigger a été lancé. Il s'agit de la ligne étant insérée, mise à jour ou effacée. Si ce trigger a été lancé pour une commande `INSERT` ou `DELETE`, c'est cette valeur que la fonction doit retourner si vous ne voulez pas remplacer la ligne par une ligne différente (dans le cas d'un `INSERT`) ou sauter l'opération. Dans le cas de triggers sur tables distantes, les valeurs des colonnes systèmes ne sont pas spécifiées ici.

`tg_newtuple`

Un pointeur vers la nouvelle version de la ligne, si le trigger a été lancé pour un `UPDATE` et `NULL` si c'est pour un `INSERT` ou un `DELETE`. C'est ce que la fonction doit retourner si l'événement est un `UPDATE` et que vous ne voulez pas remplacer cette ligne par une ligne différente ou bien sauter l'opération. Dans le cas de triggers sur tables distantes, les valeurs des colonnes systèmes ne sont pas spécifiées ici.

`tg_trigger`

Un pointeur vers une structure de type `Trigger`, définie dans `utils/rel.h` :

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    char        tgenabled;
    bool        tgisinternal;
    bool        tgisclone;
    Oid         tgconstrrelid;
    Oid         tgconstrindid;
    Oid         tgconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16      *tgattr;
    char      **tgargs;
    char       *tgqual;
    char       *tgoldtable;
    char       *tgnewtable;
} Trigger;
```

où `tgname` est le nom du trigger, `tgnargs` est le nombre d'arguments dans `tgargs` et `tgargs` est un tableau de pointeurs vers les arguments spécifiés dans l'expression contenant la commande `CREATE TRIGGER`. Les autres membres ne sont destinés qu'à un usage interne.

`tg_trigslot`

Le slot contenant `tg_trigtuple`, ou un pointeur `NULL` si cette ligne n'existe pas.

`tg_newslot`

Le slot contenant `tg_newtuple`, ou un pointeur `NULL` si cette ligne n'existe pas.

`tg_oldtable`

Un pointeur vers une structure de type `Tuplestorestate` contenant zéro ou plusieurs lignes dans le format spécifié par `tg_relation`, ou un pointeur `NULL` s'il n'y a pas de relation de transition `OLD TABLE`.

`tg_newtable`

Un pointeur vers une structure de type `Tuplestorestate` contenant zéro ou plusieurs lignes dans le format spécifié par `tg_relation`, ou un pointeur `NULL` s'il n'y a pas de relation de transition `NEW TABLE`.

Pour permettre aux requêtes exécutées via SPI de référencer les tables de transition, voir SPI_register_trigger_data.

Une fonction trigger doit retourner soit un pointeur `HeapTuple` soit un pointeur `NULL` (*pas* une valeur SQL NULL, donc ne positionnez pas *`isNull`* à true). Faites attention de renvoyer soit un `tg_trigtuple` soit un `tg_newtuple`, comme approprié, si vous ne voulez pas changer la ligne en cours de modification.

39.4. Un exemple complet de trigger

Voici un exemple très simple de fonction trigger écrite en C (les exemples de triggers écrits avec différents langages de procédures se trouvent dans la documentation de ceux-ci).

La fonction `trigf` indique le nombre de lignes de la table `ttest` et saute l'opération si la commande tente d'insérer une valeur NULL dans la colonne `x` (ainsi le trigger agit comme une contrainte non NULL mais n'annule pas la transaction).

Tout d'abord, la définition des tables :

```
CREATE TABLE ttest (
    x integer
);
```

Voici le code source de la fonction trigger :

```
#include "postgres.h"
#include "fmgr.h"
#include "executor/spi.h"       /* nécessaire pour fonctionner avec
 SPI */
#include "commands/trigger.h"   /* ... les triggers */
#include "utils/rel.h"          /* ... et relations */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checkNULL = false;
    bool        isNULL;
    int         ret, i;
```

```
    /* on s'assure que la fonction est appelée en tant que trigger
 */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* nuplet à retourner à l'exécuteur */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* vérification des valeurs NULL */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checkNULL = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connexion au gestionnaire SPI */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d",
 when, ret);

    /* obtient le nombre de lignes dans la table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when,
 ret);

    /* count(*) renvoie int8, prenez garde à bien convertir */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isNULL));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest",
 when, i);

    SPI_finish();

    if (checkNULL)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isNULL);
        if (isNULL)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

Après avoir compilé le code source (voir Section 38.10.5), déclarez la fonction et les triggers :

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'nomfichier'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE FUNCTION trigf();
```

À présent, testez le fonctionnement du trigger :

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Insertion supprimée et trigger AFTER non exécuté

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                       ^^^^^^^^
                        souvenez-vous de ce que nous avons dit sur
 la visibilité.
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                       ^^^^^^
                        souvenez-vous de ce que nous avons dit sur
 la visibilité.
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
```

```
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                        ^^^^^^
                        souvenez-vous de ce que nous avons dit sur
 la visibilité.
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

Vous trouverez des exemples plus complexes dans `src/test/regress/regress.c` et dans spi.

Chapitre 40. Triggers sur événement

Afin d'améliorer le mécanisme des triggers expliqué dans Chapitre 39, PostgreSQL fournit également des triggers sur événement. À la différence des triggers normaux, qui sont attachés à une seule table et ne capturent que des événements DML, les triggers sur événements sont globaux sur une base en particulier et sont capables de capturer tous les événements DDL.

Comme les triggers normaux, les triggers sur événement peuvent être écrits dans n'importe quel langage procédural qui inclut le support des triggers sur événement, ou en C, mais pas en pur SQL.

40.1. Aperçu du fonctionnement des triggers sur événement

Un trigger sur événement se déclenche chaque fois que l'événement qui lui est associé se déclenche sur la base qui lui est définie. Pour le moment, les seuls événements supportés sont `ddl_command_start`, `ddl_command_end`, `table_rewrite` et `sql_drop`. Le support pour des événements additionnels pourrait être ajouté dans des versions ultérieures.

L'événement `ddl_command_start` se déclenche juste avant l'exécution d'une commande `CREATE`, `ALTER`, `DROP`, `SECURITY LABEL`, `COMMENT`, `GRANT` ou `REVOKE`. Aucune vérification n'est effectuée sur l'existence ou non de l'objet avant de déclencher le trigger sur événement. Attention, cet événement ne se déclenche pas pour les commandes DDL visant les objets partagés -- bases de données, rôles, et tablespaces -- ou pour les commandes visant les triggers sur événement eux-même. Le mécanisme de trigger sur événement ne supporte pas ces types d'objets. `ddl_command_start` se déclenche également juste avant l'exécution d'une commande `SELECT INTO`, celle-ci étant l'équivalent de `CREATE TABLE AS`.

L'événement `ddl_command_end` se déclenche juste après l'exécution de ces même ensembles de commandes. Pour obtenir plus de détails sur les opérations DDL qui interviennent, utilisez la fonction renvoyant un ensemble de lignes `pg_event_trigger_ddl_commands()` à partir du code du trigger répondant à l'événement `ddl_command_end` (voir Section 9.29). Notez que le trigger est exécuté après les actions qui sont intervenues (mais avant les validations de transactions), aussi les catalogues systèmes qui peuvent être lus ont déjà été modifiés.

L'événement `sql_drop` se déclenche juste avant le trigger sur événement `ddl_command_end` pour toute opération qui supprime des objets de la base. Pour lister les objets qui ont été supprimés, utilisez la fonction retournant des ensembles d'objets `pg_event_trigger_dropped_objects()` depuis le code du trigger sur événement `sql_drop` (voir Section 9.29). Notez que le trigger est exécuté après que les objets aient été supprimés du catalogue système, il n'est donc plus possible de les examiner.

L'événement `table_rewrite` se déclenche juste avant qu'une table soit modifiée par certaines actions des commandes `ALTER TABLE` et `ALTER TYPE`. Il existe d'autres commandes qui permettent de modifier une table, tel que `CLUSTER` et `VACUUM`, mais l'événement `table_rewrite` n'est pas déclenché pour eux. Pour trouver l'OID de la table qui a été réécrite, utilisez la fonction `pg_event_trigger_table_rewrite_oid()` (voir Section 9.29). Pour trouver la raison de la réécriture, utilisez la fonction `pg_event_trigger_table_rewrite_reason()`.

Les triggers sur événement (comme les autres fonctions) ne peuvent être exécutés dans une transaction annulée. Ainsi, si une commande DDL échoue avec une erreur, tout trigger `ddl_command_end` associé ne sera pas exécuté. Inversement, si un trigger `ddl_command_start` échoue avec une erreur, aucun autre trigger sur événement ne se déclenchera, et aucune tentative ne sera faite pour exécuter la commande elle-même. De la même façon, si une commande `ddl_command_end` échoue avec une erreur, les effets de la commande DDL seront annulés, comme elles l'auraient été dans n'importe quel autre cas où la transaction qui la contient est annulée.

Pour une liste complète des commandes supportées par le mécanisme des triggers sur événement, voir Section 40.2.

Les triggers sur événement sont créés en utilisant la commande CREATE EVENT TRIGGER. Afin de créer un trigger sur événement, vous devez d'abord créer une fonction avec le type de retour spécial `event_trigger`. Cette fonction n'a pas besoin (et ne devrait pas) retourner de valeur ; le type de retour sert uniquement comme signal pour que la fonction soit appelée comme un trigger sur événement.

Si plus d'un trigger sur événement est défini pour un événement particulier, ils seront déclenchés par ordre alphabétique de leur nom.

Une définition de trigger peut également spécifier une condition `WHEN` pour que, par exemple, un trigger `ddl_command_start` ne soit déclenché que pour des commandes particulières que l'utilisateur souhaite intercepter. Une utilisation typique de tels triggers serait de restreindre la portée des opérations DDL que les utilisateurs peuvent exécuter.

40.2. Matrice de déclenchement des triggers sur événement

Tableau 40.1 liste toutes les commandes pour lesquelles les triggers sur événement sont supportés.

Tableau 40.1. Support des triggers sur événement par commande

Commande	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	Notes
ALTER AGGREGATE	X	X	-	-	
ALTER COLLATION	X	X	-	-	
ALTER CONVERSION	X	X	-	-	
ALTER DOMAIN	X	X	-	-	
ALTER DEFAULT PRIVILEGES	X	X	-	-	
ALTER EXTENSION	X	X	-	-	
ALTER FOREIGN DATA WRAPPER	X	X	-	-	
ALTER FOREIGN TABLE	X	X	X	-	
ALTER FUNCTION	X	X	-	-	
ALTER LANGUAGE	X	X	-	-	
ALTER LARGE OBJECT	X	X	-	-	
ALTER MATERIALIZED VIEW	X	X	-	X	
ALTER OPERATOR	X	X	-	-	
ALTER OPERATOR CLASS	X	X	-	-	
ALTER OPERATOR FAMILY	X	X	-	-	
ALTER POLICY	X	X	-	-	
ALTER PROCEDURE	X	X	-	-	
ALTER ROUTINE	X	X	-	-	

Commande	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	Notes
ALTER PUBLICATION	X	X	-	-	
ALTER SCHEMA	X	X	-	-	
ALTER SEQUENCE	X	X	-	-	
ALTER SERVER	X	X	-	-	
ALTER STATISTICS	X	X	-	-	
ALTER SUBSCRIPTION	X	X	-	-	
ALTER TABLE	X	X	X	X	
ALTER TEXT SEARCH CONFIGURATION	X	X	-	-	
ALTER TEXT SEARCH DICTIONARY	X	X	-	-	
ALTER TEXT SEARCH PARSER	X	X	-	-	
ALTER TEXT SEARCH TEMPLATE	X	X	-	-	
ALTER TRIGGER	X	X	-	-	
ALTER TYPE	X	X	-	X	
ALTER USER MAPPING	X	X	-	-	
ALTER VIEW	X	X	-	-	
COMMENT	X	X	-	-	Seulement pour les objets locaux
CREATE ACCESS METHOD	X	X	-	-	
CREATE AGGREGATE	X	X	-	-	
CREATE CAST	X	X	-	-	
CREATE COLLATION	X	X	-	-	
CREATE CONVERSION	X	X	-	-	
CREATE DOMAIN	X	X	-	-	
CREATE EXTENSION	X	X	-	-	
CREATE FOREIGN DATA WRAPPER	X	X	-	-	
CREATE FOREIGN TABLE	X	X	-	-	
CREATE FUNCTION	X	X	-	-	
CREATE INDEX	X	X	-	-	
CREATE LANGUAGE	X	X	-	-	
CREATE MATERIALIZED VIEW	X	X	-	-	
CREATE OPERATOR	X	X	-	-	
CREATE OPERATOR CLASS	X	X	-	-	

Commande	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	Notes
CREATE OPERATOR FAMILY	X	X	-	-	
CREATE POLICY	X	X	-	-	
CREATE PROCEDURE	X	X	-	-	
CREATE PUBLICATION	X	X	-	-	
CREATE RULE	X	X	-	-	
CREATE SCHEMA	X	X	-	-	
CREATE SEQUENCE	X	X	-	-	
CREATE SERVER	X	X	-	-	
CREATE STATISTICS	X	X	-	-	
CREATE SUBSCRIPTION	X	X	-	-	
CREATE TABLE	X	X	-	-	
CREATE TABLE AS	X	X	-	-	
CREATE TEXT SEARCH CONFIGURATION	X	X	-	-	
CREATE TEXT SEARCH DICTIONARY	X	X	-	-	
CREATE TEXT SEARCH PARSER	X	X	-	-	
CREATE TEXT SEARCH TEMPLATE	X	X	-	-	
CREATE TRIGGER	X	X	-	-	
CREATE TYPE	X	X	-	-	
CREATE USER MAPPING	X	X	-	-	
CREATE VIEW	X	X	-	-	
DROP ACCESS METHOD	X	X	X	-	
DROP AGGREGATE	X	X	X	-	
DROP CAST	X	X	X	-	
DROP COLLATION	X	X	X	-	
DROP CONVERSION	X	X	X	-	
DROP DOMAIN	X	X	X	-	
DROP EXTENSION	X	X	X	-	
DROP FOREIGN DATA WRAPPER	X	X	X	-	
DROP FOREIGN TABLE	X	X	X	-	
DROP FUNCTION	X	X	X	-	
DROP INDEX	X	X	X	-	
DROP LANGUAGE	X	X	X	-	
DROP MATERIALIZED VIEW	X	X	X	-	

Commande	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	Notes
DROP OPERATOR	X	X	X	-	
DROP OPERATOR CLASS	X	X	X	-	
DROP OPERATOR FAMILY	X	X	X	-	
DROP OWNED	X	X	X	-	
DROP POLICY	X	X	X	-	
DROP PROCEDURE	X	X	X	-	
DROP ROUTINE	X	X	X	-	
DROP PUBLICATION	X	X	X	-	
DROP RULE	X	X	X	-	
DROP SCHEMA	X	X	X	-	
DROP SEQUENCE	X	X	X	-	
DROP SERVER	X	X	X	-	
DROP STATISTICS	X	X	X	-	
DROP SUBSCRIPTION	X	X	X	-	
DROP TABLE	X	X	X	-	
DROP TEXT SEARCH CONFIGURATION	X	X	X	-	
DROP TEXT SEARCH DICTIONARY	X	X	X	-	
DROP TEXT SEARCH PARSER	X	X	X	-	
DROP TEXT SEARCH TEMPLATE	X	X	X	-	
DROP TRIGGER	X	X	X	-	
DROP TYPE	X	X	X	-	
DROP USER MAPPING	X	X	X	-	
DROP VIEW	X	X	X	-	
GRANT	X	X	-	-	Seulement pour les objets locaux
IMPORT FOREIGN SCHEMA	X	X	-	-	
REFRESH MATERIALIZED VIEW	X	X	-	-	
REVOKE	X	X	-	-	Seulement pour les objets locaux
SECURITY LABEL	X	X	-	-	Seulement pour les objets locaux
SELECT INTO	X	X	-	-	

40.3. Écrire des fonctions trigger sur événement en C

Cette section décrit les détails bas niveau de l'interface pour une fonction trigger sur événement bas niveau. Ces informations sont seulement nécessaires si vous écrivez des fonctions triggers sur événement en C. Si vous utilisez un langage de plus haut niveau, ces détails sont gérés pour vous. Dans la plupart des cas, vous devriez songer sérieusement à utiliser un langage procédural avant d'écrire vos triggers sur événement en C. La documentation de chaque langage procédurale explique comment écrire un trigger sur événement dans ce langage.

Les fonctions de trigger sur événement doivent utiliser l'interface du gestionnaire de fonctions « version 1 ».

Quand une fonction est appelée par le gestionnaire de triggers sur événement, elle ne reçoit aucun argument normal mais un pointeur « context » lui est fourni. Il pointe vers une structure de type `EventTriggerData`. Les fonctions C peuvent vérifier si elles ont été appelées par le gestionnaire de triggers sur événement en exécutant la macro :

```
CALLED_AS_EVENT_TRIGGER(fcinfo)
```

qui vaut en fait :

```
((fcinfo)->context != NULL && IsA((fcinfo)->context,
 EventTriggerData))
```

Si cela renvoie la valeur true, alors il est possible de convertir `fcinfo->context` vers le type `EventTriggerData *` et d'utiliser la structure pointée `EventTriggerData`. La fonction ne doit *pas* modifier la structure `EventTriggerData` ou toute donnée qu'elle fournit.

`struct EventTriggerData` est défini dans `commands/event_trigger.h` :

```
typedef struct EventTriggerData
{
    NodeTag     type;
    const char *event;      /* event name */
    Node       *parsetree;  /* parse tree */
    CommandTag  tag;        /* command tag */
} EventTriggerData;
```

dont les membres sont définis ainsi :

`type`

Always `T_EventTriggerData`.

`event`

Décrit l'événement pour lequel la fonction a été appelée. Ce sera soit `"ddl_command_start"`, soit `"ddl_command_end"`, soit `"sql_drop"`, soit `"table_rewrite"`. Voir Section 40.1 pour la signification de ces événements.

`parsetree`

Un pointeur vers l'arbre d'analyse de la commande. Vérifiez le code source de PostgreSQL pour les détails. La structure de l'arbre d'analyse est sujet à modification sans notification.

`tag`

La balise de la commande associée avec l'événement pour lequel le trigger sur événement est exécuté, par exemple `"CREATE FUNCTION"`.

Une fonction trigger sur événement doit renvoyer un pointeur `NULL` (et *pas* une valeur SQL NULL, autrement dit ne pas configurer *`isNull`* à true).

40.4. Un exemple complet de trigger sur événement

Voici un exemple très simple d'une fonction trigger sur événement écrite en C. (Les exemples de triggers écrits en langage procédural peuvent être trouvés dans la documentation de ces langages procédurals.)

La fonction `noddl` lève une exception à chaque fois qu'elle est appelée. La définition du trigger événement associe la fonction à l'événement `ddl_command_start`. L'effet est qu'aucune commande DDL (à l'exception de celles mentionnées dans Section 40.1) ne peut être exécutée.

Voici le code source de la fonction trigger :

```
#include "postgres.h"
#include "commands/event_trigger.h"


PG_MODULE_MAGIC;

Datum noddl(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(noddl);

Datum
noddl(PG_FUNCTION_ARGS)
{
    EventTriggerData *trigdata;

    if (!CALLED_AS_EVENT_TRIGGER(fcinfo))  /* internal error */
        elog(ERROR, "not fired by event trigger manager");

    trigdata = (EventTriggerData *) fcinfo->context;

    (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
     errmsg("command \"%s\" denied",
            GetCommandTagName(trigdata->tag))));

    PG_RETURN_NULL();
}
```

Après avoir compilé le code source (voir Section 38.10.5), déclarez la fonction et les triggers :

```
CREATE FUNCTION noddl() RETURNS event_trigger
    AS 'noddl' LANGUAGE C;

CREATE EVENT TRIGGER noddl ON ddl_command_start
    EXECUTE FUNCTION noddl();
```

Maintenant vous pouvez tester le bon fonctionnement du trigger :

```
=# \dy
                     List of event triggers
 Name  |       Event       | Owner | Enabled | Function | Tags
-------+-------------------+-------+---------+----------+------
 noddl | ddl_command_start | dim   | enabled | noddl    |
(1 row)

=# CREATE TABLE foo(id serial);
ERROR:  command "CREATE TABLE" denied
```

Dans ce cas, pour pouvoir activier l'exécution de certaines commandes DDL, vous pouvez soit supprimer le trigger sur événement soit le désactiver. Il est généralement plus simple de désactiver le trigger pendant la durée de la transaction :

```
BEGIN;
ALTER EVENT TRIGGER noddl DISABLE;
CREATE TABLE foo (id serial);
ALTER EVENT TRIGGER noddl ENABLE;
COMMIT;
```

(Pour rappel, les commandes DDL sur les triggers sur événement ne sont pas affectées par les triggers sur événement.)

40.5. Un exemple de trigger sur événement de table modifiée

Grâce à l'événement `table_rewrite`, il est possible d'écrire une fonction qui autorise les modifications d'une table seulement pendant les heures de maintenance.

Ci-dessous un exemple d'implémentation d'une telle règle.

```
CREATE OR REPLACE FUNCTION pas_de_modification()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
---
--- Implémentation d'une règle de modification de table:
---  pas de modifications de public.foo, les
---  autres tables peuvent l'être entre 01:00 et 06:00 du matin
---  sauf si elles ont plus de 100 blocs
---
DECLARE
  table_oid oid := pg_event_trigger_table_rewrite_oid();
  heure_courante integer := extract('hour' from current_time);
  pages integer;
  max_pages integer := 100;
BEGIN
  IF pg_event_trigger_table_rewrite_oid() = 'public.foo'::regclass
  THEN
```

```
        RAISE EXCEPTION 'Vous n''êtes pas autorisé à modifier la
 table %',
                        table_oid::regclass;
  END IF;

  SELECT INTO pages relpages FROM pg_class WHERE oid = table_oid;
  IF pages > max_pages
  THEN
        RAISE EXCEPTION 'les modifications ne sont seulement
 permises que pour les tables ayant un nombre de blocs inférieur à
 %',
                        max_pages;
  END IF;

  IF heure_courante NOT BETWEEN 1 AND 6
  THEN
        RAISE EXCEPTION 'les modifications sont seulement
 autorisées entre 01:00 et 06:00 du matin';
  END IF;
END;
$$;

CREATE EVENT TRIGGER pas_de_modifications_permises
                  ON table_rewrite
   EXECUTE FUNCTION pas_de_modification();
```

Chapitre 41. Système de règles

Ce chapitre discute du système de règles dans PostgreSQL. Les systèmes de règles de production sont simples conceptuellement mais il existe de nombreux points subtils impliqués dans leur utilisation.

Certains autres systèmes de bases de données définissent des règles actives pour la base de données, conservées habituellement en tant que procédures stockées et triggers. Avec PostgreSQL, elles peuvent aussi être implémentées en utilisant des fonctions et des triggers.

Le système de règles (plus précisément, le système de règles de réécriture de requêtes) est totalement différent des procédures stockées et des triggers. Il modifie les requêtes pour prendre en considération les règles puis passe la requête modifiée au planificateur de requêtes pour planification et exécution. Il est très puissant et peut être utilisé pour beaucoup de choses comme des procédures en langage de requêtes, des vues et des versions. Les fondations théoriques et la puissance de ce système de règles sont aussi discutées dans [ston90b] et [ong90].

41.1. Arbre de requêtes

Pour comprendre comment fonctionne le système de règles, il est nécessaire de comprendre quand il est appelé et quelles sont ses entrées et sorties.

Le système de règles est situé entre l'analyseur et le planificateur. Il prend la sortie de l'analyseur, un arbre de requête et les règles de réécriture définies par l'utilisateur qui sont aussi des arbres de requêtes avec quelques informations supplémentaires, et crée zéro ou plusieurs arbres de requêtes comme résultat. Donc, son entrée et sortie sont toujours des éléments que l'analyseur lui-même pourrait avoir produit et, du coup, tout ce qu'il voit est représentable basiquement comme une instruction SQL.

Maintenant, qu'est-ce qu'un arbre de requêtes ? C'est une représentation interne d'une instruction SQL où les parties qui le forment sont stockées séparément. Ces arbres de requêtes sont affichables dans le journal de traces du serveur si vous avez configuré les paramètres `debug_print_parse`, `debug_print_rewritten`, ou `debug_print_plan`. Les actions de règles sont aussi enregistrées comme arbres de requêtes dans le catalogue système `pg_rewrite`. Elles ne sont pas formatées comme la sortie de traces mais elles contiennent exactement la même information.

Lire un arbre de requête brut requiert un peu d'expérience. Mais comme les représentations SQL des arbres de requêtes sont suffisantes pour comprendre le système de règles, ce chapitre ne vous apprendra pas à les lire.

Lors de la lecture des représentations SQL des arbres de requêtes dans ce chapitre, il est nécessaire d'être capable d'identifier les morceaux cassés de l'instruction lorsqu'ils sont dans la structure de l'arbre de requête. Les parties d'un arbre de requêtes sont

le type de commande

C'est une simple valeur indiquant quelle commande (`select`, `insert`, `update`, `delete`) l'arbre de requêtes produira.

la table d'échelles

La table d'échelles est une liste des relations utilisées dans la requête. Dans une instruction `select`, ce sont les relations données après le mot clé `from`.

Chaque entrée de la table d'échelles identifie une table ou une vue et indique par quel nom elle est désignée dans les autres parties de la requête. Dans l'arbre de requêtes, les entrées de la table d'échelles sont référencées par des numéros plutôt que par des noms. Il importe donc peu, ici, de savoir s'il y a des noms dupliqués comme cela peut être le cas avec une instruction SQL. Cela peut arriver après l'assemblage des tables d'échelle des règles. Les exemples de ce chapitre ne sont pas confrontés à cette situation.

la relation résultat

C'est un index dans la table d'échelles qui identifie la relation où iront les résultats de la requête.

Les requêtes `select` n'ont pas de relation résultat. Le cas spécial d'un `select into` est pratiquement identique à un `create table` suivi par un `insert ... select` et n'est pas discuté séparément ici.

Pour les commandes `insert`, `update` et `delete`, la relation de résultat est la table (ou vue !) où les changements prennent effet.

la liste cible

La liste cible est une liste d'expressions définissant le résultat d'une requête. Dans le cas d'un `select`, ces expressions sont celles qui construisent la sortie finale de la requête. Ils correspondent aux expressions entre les mots clés `select` et `from` (`*` est seulement une abréviation pour tous les noms de colonnes d'une relation. Il est étendu par l'analyseur en colonnes individuelles, pour que le système de règles ne le voie jamais).

Les commandes `delete` n'ont pas besoin d'une liste normale de colonnes car elles ne produisent aucun résultat. En fait, l'optimiseur ajoutera une entrée spéciale ctid pour aller jusqu'à la liste de cibles vide pour permettre à l'exécuteur de trouver la ligne à supprimer. (CTID est ajouté quand la relation résultante est une table ordinaire. S'il s'agit d'une vue, une variable de type ligne est ajoutée à la place, par le système de règles, comme décrit dans Section 41.2.4.)

Pour les commandes `insert`, la liste cible décrite les nouvelles lignes devant aller dans la relation résultat. Elle consiste en des expressions de la clause `values` ou en celles de la clause `select` dans `insert ... SELECT`. La première étape du processus de réécriture ajoute les entrées de la liste cible pour les colonnes n'ont affecté par la commande originale mais ayant des valeurs par défaut. Toute colonne restante (avec soit une valeur donnée soit une valeur par défaut) sera remplie par le planificateur avec une expression NULL constante.

Pour les commandes `update`, la liste cible décrite les nouvelles lignes remplaçant les anciennes. Dans le système des règles, elle contient seulement les expressions de la partie `set colonne = expression` de la commande. Le planificateur gèrera les colonnes manquantes en insérant des expressions qui copient les valeurs provenant de l'ancienne ligne dans la nouvelle. Comme pour `DELETE`, un CTID ou une variable de type ligne est ajouté pour que l'exécuteur puisse identifier l'ancienne ligne à mettre à jour.

Chaque entrée de la liste cible contient une expression qui peut être une valeur constante, une variable pointant vers une colonne d'une des relations de la table d'échelles, un paramètre ou un arbre d'expressions réalisé à partir d'appels de fonctions, de constantes, de variables, d'opérateurs, etc.

la qualification

La qualification de la requête est une expression ressemblant à une de celles contenues dans les entrées de la liste cible. La valeur résultant de cette expression est un booléen indiquant si l'opération (`insert`, `update`, `delete` ou `select`) pour la ligne de résultat final devrait être exécutée ou non. Elle correspond à la clause `where` d'une instruction SQL.

l'arbre de jointure

L'arbre de jointure de la requête affiche la structure de la clause `from`. Pour une simple requête comme `select ... from a, b, c`, l'arbre de jointure est une simple liste d'éléments de `from` parce que nous sommes autorisés à les joindre dans tout ordre. Mais quand des expressions `join`, et plus particulièrement les jointures externes, sont utilisées, nous devons les joindre dans l'ordre affiché par les jointures. Dans ce cas, l'arbre de jointure affiche la structure des expressions `join`. Les restrictions associées avec ces clauses `join` particulières (à partir d'expressions `on` ou `using`) sont enregistrées comme des expressions de qualification attachées aux nœuds de l'arbre de jointure. Il s'avère agréable d'enregistrer l'expression de haut niveau `where` comme une

qualification attachée à l'élément de l'arbre de jointure de haut niveau. Donc, réellement, l'arbre de jointure représente à la fois les clauses `from` et `where` d'un `select`.

le reste

Les autres parties de l'arbre de requête comme la clause `order BY` n'ont pas d'intérêt ici. Le système de règles substitue quelques entrées lors de l'application des règles mais ceci n'a pas grand-chose à voir avec les fondamentaux du système de règles.

41.2. Vues et système de règles

Avec PostgreSQL, les vues sont implémentées en utilisant le système de règles. En fait, il n'y a essentiellement pas de différences entre

```
CREATE VIEW ma_vue AS SELECT * FROM ma_table;
```

et ces deux commandes :

```
CREATE TABLE ma_vue (liste de colonnes identique à celle de
 ma_table);
CREATE RULE "_RETURN" AS ON SELECT TO ma_vue DO INSTEAD
    SELECT * FROM ma_table;
```

parce que c'est exactement ce que fait la commande `create VIEW` en interne. Cela présente quelques effets de bord. L'un d'entre eux est que l'information sur une vue dans les catalogues système PostgreSQL est exactement la même que celle d'une table. Donc, pour l'analyseur, il n'y a aucune différence entre une table et une vue. Elles représentent la même chose : des relations.

41.2.1. Fonctionnement des règles `select`

Les règles `on select` sont appliquées à toutes les requêtes comme la dernière étape, même si la commande donnée est un `insert`, `update` ou `delete`. Et ils ont une sémantique différente à partir des règles sur les autres types de commandes dans le fait qu'elles modifient l'arbre de requêtes en place au lieu d'en créer un nouveau. Donc, les règles `select` sont décrites avant.

Actuellement, il n'existe qu'une action dans une règle `on SELECT` et elle doit être une action `select` inconditionnelle qui est `instead`. Cette restriction était requise pour rendre les règles assez sûres pour les ouvrir aux utilisateurs ordinaires et cela restreint les règles `on select` à agir comme des vues.

Pour ce chapitre, les exemples sont deux vues jointes réalisant quelques calculs et quelques vues supplémentaires les utilisant à leur tour. Une des deux premières vues est personnalisée plus tard en ajoutant des règles pour des opérations `insert`, `update` et `delete` de façon à ce que le résultat final sera une vue qui se comporte comme une vraie table avec quelques fonctionnalités magiques. Il n'existe pas un tel exemple pour commencer et ceci rend les choses plus difficiles à obtenir. Mais il est mieux d'avoir un exemple couvrant tous les points discutés étape par étape plutôt que plusieurs exemples, rendant la compréhension plus difficile.

Les tables réelles dont nous avons besoin dans les deux premières descriptions du système de règles sont les suivantes :

```
CREATE TABLE donnees_chaussure (
    nom_chaussure         text,     -- clé primaire
    dispo_chaussure       integer,  -- nombre de pairs disponibles
    couleur_chaussure     text,     -- couleur de lacet préférée
    long_min_chaussure    real,     -- longueur minimum du lacet
    long_max_chaussure    real,     -- longueur maximum du lacet
    unite_long_chaussure  text      -- unité de longueur
```

```
);

CREATE TABLE donnees_lacet (
    nom_lacet             text,     -- clé primaire
    dispo_lacet           integer,  -- nombre de pairs disponibles
    couleur_lacet         text,     -- couleur du lacet
    longueur_lacet        real,     -- longueur du lacet
    unite_lacet           text      -- unité de longueur
);

CREATE TABLE unite (
    nom_unite             text,     -- clé primaire
    facteur_unite         real      -- facteur pour le transformer
 en cm
);
```

Comme vous pouvez le constater, elles représentent les données d'un magasin de chaussures.

Les vues sont créées avec :

```
CREATE VIEW chaussure AS
    SELECT sh.nom_chaussure,
           sh.dispo_chaussure,
           sh.couleur_chaussure,
           sh.long_min_chaussure,
           sh.long_min_chaussure * un.facteur_unite AS
 long_min_chaussure_cm,
           sh.long_max_chaussure,
           sh.long_max_chaussure * un.facteur_unite AS
 long_max_chaussure_cm,
           sh.unite_long_chaussure
      FROM donnees_chaussure sh, unite un
     WHERE sh.unite_long_chaussure = un.nom_unite;

CREATE VIEW lacet AS
    SELECT s.nom_lacet,
           s.dispo_lacet,
           s.couleur_lacet,
           s.longueur_lacet,
           s.unite_lacet,
           s.longueur_lacet * u.facteur_unite AS longueur_lacet_cm
      FROM donnees_lacet s, unite u
     WHERE s.unite_lacet = u.nom_unite;

CREATE VIEW chaussure_prete AS
    SELECT rsh.nom_chaussure,
           rsh.dispo_chaussure,
           rsl.nom_lacet,
           rsl.dispo_lacet,
           least(rsh.dispo, rsl.dispo_lacet) AS total_avail
      FROM chaussure rsh, lacet rsl
     WHERE rsl.couleur_lacet = rsh.couleur
       AND rsl.longueur_lacet_cm >= rsh.long_min_chaussure_cm
       AND rsl.longueur_lacet_cm <= rsh.long_max_chaussure_cm;
```

La commande `create view` pour la vue `lacet` (qui est la plus simple que nous avons) écrira une relation `lacet` et une entrée dans `pg_rewrite` indiquant la présence d'une règle de réécriture

devant être appliquée à chaque fois que la relation `lacet` est référencée dans une table de la requête. La règle n'a aucune qualification de règle (discuté plus tard, avec les règles autres que `select` car les règles `select` ne le sont pas encore) et qu'il s'agit de `instead`. Notez que les qualifications de règles ne sont pas identiques aux qualifications de requêtes. L'action de notre règle a une qualification de requête. L'action de la règle a un arbre de requête qui est une copie de l'instruction `select` dans la commande de création de la vue.

> **Note**
>
> Les deux entrées supplémentaires de la table d'échelles pour `new` et `old` que vous pouvez voir dans l'entrée de `pg_rewrite` ne sont d'aucun intérêt pour les règles `select`.

Maintenant, nous remplissons `unit`, `donnees_chaussure` et `donnees_lacet`, puis nous lançons une requête simple sur une vue :

```
INSERT INTO unite VALUES ('cm', 1.0);
INSERT INTO unite VALUES ('m', 100.0);
INSERT INTO unite VALUES ('inch', 2.54);

INSERT INTO donnees_chaussure VALUES ('sh1', 2, 'black', 70.0,
 90.0, 'cm');
INSERT INTO donnees_chaussure VALUES ('sh2', 0, 'black', 30.0,
 40.0, 'inch');
INSERT INTO donnees_chaussure VALUES ('sh3', 4, 'brown', 50.0,
 65.0, 'cm');
INSERT INTO donnees_chaussure VALUES ('sh4', 3, 'brown', 40.0,
 50.0, 'inch');

INSERT INTO donnees_lacet VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO donnees_lacet VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO donnees_lacet VALUES ('sl3', 0, 'black', 35.0 ,
 'inch');
INSERT INTO donnees_lacet VALUES ('sl4', 8, 'black', 40.0 ,
 'inch');
INSERT INTO donnees_lacet VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO donnees_lacet VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO donnees_lacet VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO donnees_lacet VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM lacet;

 nom_lacet   | dispo_lacet | couleur_lacet | longueur_lacet |
 unite_lacet | longueur_lacet_cm
-------------+-------------+---------------+----------------
+-------------+------------------
 sl1         |           5 | black         |             80 | cm
       |              80
 sl2         |           6 | black         |            100 | cm
       |             100
 sl7         |           7 | brown         |             60 | cm
       |              60
 sl3         |           0 | black         |             35 | inch
       |            88.9
 sl4         |           8 | black         |             40 | inch
       |           101.6
 sl8         |           1 | brown         |             40 | inch
       |           101.6
```

```
 sl5         |           4 | brown         |              1 | m
       |             100
 sl6         |           0 | brown         |            0.9 | m
       |              90
(8 rows)
```

C'est la requête `select` la plus simple que vous pouvez lancer sur nos vues, donc nous prenons cette opportunité d'expliquer les bases des règles de vues. `select * from lacet` a été interprété par l'analyseur et a produit l'arbre de requête :

```
SELECT lacet.nom_lacet, lacet.dispo_lacet,
       lacet.couleur_lacet, lacet.longueur_lacet,
       lacet.unite_lacet, lacet.longueur_lacet_cm
  FROM lacet lacet;
```

et ceci est transmis au système de règles. Ce système traverse la table d'échelle et vérifie s'il existe des règles pour chaque relation. Lors du traitement d'une entrée de la table d'échelles pour `lacet` (la seule jusqu'à maintenant), il trouve la règle `_return` avec l'arbre de requête :

```
SELECT s.nom_lacet, s.dispo_lacet,
       s.couleur_lacet, s.longueur_lacet, s.unite_lacet,
       s.longueur_lacet * u.facteur_unite AS longueur_lacet_cm
  FROM lacet old, lacet new,
       donnees_lacet s, unit u
 WHERE s.unite_lacet = u.nom_unite;
```

Pour étendre la vue, la réécriture crée simplement une entrée de la table d'échelle de sous-requête contenant l'arbre de requête de l'action de la règle et substitue cette entrée avec l'original référencé dans la vue. L'arbre d'échelle résultant de la réécriture est pratiquement identique à celui que vous avez saisi :

```
SELECT lacet.nom_lacet, lacet.dispo_lacet,
       lacet.couleur_lacet, lacet.longueur_lacet,
       lacet.unite_lacet, lacet.longueur_lacet_cm
  FROM (SELECT s.nom_lacet,
               s.dispo_lacet,
               s.couleur_lacet,
               s.longueur_lacet,
               s.unite_lacet,
               s.longueur_lacet * u.facteur_unite AS
 longueur_lacet_cm
          FROM donnees_lacet s, unit u
         WHERE s.unite_lacet = u.nom_unite) lacet;
```

Néanmoins, il y a une différence : la table d'échelles de la sous-requête a deux entrées supplémentaires, `lacet old` et `lacet new`. Ces entrées ne participent pas directement dans la requête car elles ne sont pas référencées par l'arbre de jointure de la sous-requête ou par la liste cible. La réécriture les utilise pour enregistrer l'information de vérification des droits d'accès qui étaient présents à l'origine dans l'entrée de table d'échelles référencée par la vue. De cette façon, l'exécution vérifiera toujours que l'utilisateur a les bons droits pour accéder à la vue même s'il n'y a pas d'utilisation directe de la vue dans la requête réécrite.

C'était la première règle appliquée. Le système de règles continuera de vérifier les entrées restantes de la table d'échelles dans la requête principale (dans cet exemple, il n'en existe pas plus), et il vérifiera récursivement les entrées de la table d'échelles dans la sous-requête ajoutée pour voir si une d'elle

référence les vues. (Mais il n'étendra ni `old` ni `new` -- sinon nous aurions une récursion infinie !) Dans cet exemple, il n'existe pas de règles de réécriture pour `donnees_lacet` ou `unit`, donc la réécriture est terminée et ce qui est ci-dessus est le résultat final donné au planificateur.

Maintenant, nous voulons écrire une requête qui trouve les chaussures en magasin dont nous avons les lacets correspondants (couleur et longueur) et pour lesquels le nombre total de pairs correspondants exactement est supérieur ou égal à deux.

```
SELECT * FROM chaussure_prete WHERE total_avail >= 2;

 nom_chaussure | dispo | nom_lacet | dispo_lacet | total_avail
---------------+-------+-----------+-------------+-------------
 sh1           |     2 | sl1       |           5 |           2
 sh3           |     4 | sl7       |           7 |           4
(2 rows)
```

Cette fois, la sortie de l'analyseur est l'arbre de requête :

```
SELECT chaussure_prete.nom_chaussure, chaussure_prete.dispo,
       chaussure_prete.nom_lacet, chaussure_prete.dispo_lacet,
       chaussure_prete.total_avail
  FROM chaussure_prete chaussure_prete
 WHERE chaussure_prete.total_avail >= 2;
```

La première règle appliquée sera celle de la vue `chaussure_prete` et cela résultera en cet arbre de requête :

```
SELECT chaussure_prete.nom_chaussure, chaussure_prete.dispo,
       chaussure_prete.nom_lacet, chaussure_prete.dispo_lacet,
       chaussure_prete.total_avail
  FROM (SELECT rsh.nom_chaussure,
               rsh.dispo,
               rsl.nom_lacet,
               rsl.dispo_lacet,
               least(rsh.dispo, rsl.dispo_lacet) AS total_avail
          FROM chaussure rsh, lacet rsl
         WHERE rsl.couleur_lacet = rsh.couleur
          AND rsl.longueur_lacet_cm >= rsh.long_min_chaussure_cm
          AND rsl.longueur_lacet_cm <= rsh.long_max_chaussure_cm)
 chaussure_prete
 WHERE chaussure_prete.total_avail >= 2;
```

De façon similaire, les règles pour `chaussure` et `lacet` sont substituées dans la table d'échelles de la sous-requête, amenant à l'arbre de requête final à trois niveaux :

```
SELECT chaussure_prete.nom_chaussure, chaussure_prete.dispo,
       chaussure_prete.nom_lacet, chaussure_prete.dispo_lacet,
       chaussure_prete.total_avail
  FROM (SELECT rsh.nom_chaussure,
               rsh.dispo,
               rsl.nom_lacet,
               rsl.dispo_lacet,
               least(rsh.dispo, rsl.dispo_lacet) AS total_avail
          FROM (SELECT sh.nom_chaussure,
                       sh.dispo,
                       sh.couleur,
```

```
                      sh.long_min_chaussure,
                      sh.long_min_chaussure * un.facteur_unite AS
long_min_chaussure_cm,
                      sh.long_max_chaussure,
                      sh.long_max_chaussure * un.facteur_unite AS
long_max_chaussure_cm,
                      sh.unite_long_chaussure
                 FROM donnees_chaussure sh, unit un
                WHERE sh.unite_long_chaussure = un.nom_unite) rsh,
              (SELECT s.nom_lacet,
                      s.dispo_lacet,
                      s.couleur_lacet,
                      s.longueur_lacet,
                      s.unite_lacet,
                      s.longueur_lacet * u.facteur_unite AS
longueur_lacet_cm
                 FROM donnees_lacet s, unit u
                WHERE s.unite_lacet = u.nom_unite) rsl
        WHERE rsl.couleur_lacet = rsh.couleur
          AND rsl.longueur_lacet_cm >= rsh.long_min_chaussure_cm
          AND rsl.longueur_lacet_cm <= rsh.long_max_chaussure_cm)
chaussure_prete
WHERE chaussure_prete.total_avail >= 2;
```

Ceci pourrait sembler inefficace mais le planificateur rassemblera ceci en un arbre de requête à un seul niveau en « remontant » les sous-requêtes, puis il planifiera les jointures comme si nous les avions écrites manuellement. Donc remonter l'arbre de requête est une optimisation dont le système de réécriture n'a pas à se soucier lui-même.

41.2.2. Règles de vue dans des instructions autres que `select`

Deux détails de l'arbre de requête n'ont pas été abordés dans la description des règles de vue ci-dessus. Ce sont le type de commande et la relation résultante. En fait, le type de commande n'est pas nécessaire pour les règles de la vue mais la relation résultante pourrait affecter la façon dont la requête sera réécrite car une attention particulière doit être prise si la relation résultante est une vue.

Il existe seulement quelques différences entre un arbre de requête pour un `select` et un pour une autre commande. de façon évidente, ils ont un type de commande différent et pour une commande autre qu' un `select`, la relation résultante pointe vers l'entrée de table d'échelles où le résultat devrait arriver. Tout le reste est absolument identique. Donc, avec deux tables `t1` et `t2` avec les colonnes `a` et `b`, les arbres de requêtes pour les deux commandes :

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

sont pratiquement identiques. En particulier :

- Les tables d'échelle contiennent des entrées pour les tables `t1` et `t2`.
- Les listes cibles contiennent une variable pointant vers la colonne `b` de l'entrée de la table d'échelles pour la table `t2`.
- Les expressions de qualification comparent les colonnes `a` des deux entrées de table d'échelles pour une égalité.

- Les arbres de jointure affichent une jointure simple entre `t1` et `t2`.

La conséquence est que les deux arbres de requête résultent en des plans d'exécution similaires : ce sont tous les deux des jointures sur les deux tables. Pour l'`update`, les colonnes manquantes de `t1` sont ajoutées à la liste cible par le planificateur et l'arbre de requête final sera lu de cette façon :

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

et, du coup, l'exécuteur lancé sur la jointure produira exactement le même résultat qu'un :

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

Mais il existe un petit problème dans `UPDATE` : la partie du plan d'exécution qui fait la jointure ne prête pas attention à l'intérêt des résultats de la jointure. Il produit un ensemble de lignes. Le fait qu'il y a une commande `SELECT` et une commande `UPDATE` est géré plus haut dans l'exécuteur où cette partie sait qu'il s'agit d'une commande `UPDATE`, et elle sait que ce résultat va aller dans la table `t1`. Mais lesquels de ces lignes vont être remplacées par la nouvelle ligne ?

Pour résoudre ce problème, une autre entrée est ajoutée dans la liste cible de l'`update` (et aussi dans les instructions `delete`) : l'identifiant actuel du tuple (ctid, acronyme de *current tuple ID*). cette colonne système contient le numéro de bloc du fichier et la position dans le bloc pour cette ligne. Connaissant la table, le ctid peut être utilisé pour récupérer la ligne originale de `t1` à mettre à jour. Après avoir ajouté le ctid dans la liste cible, la requête ressemble à ceci :

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Maintenant, un autre détail de PostgreSQL entre en jeu. Les anciennes lignes de la table ne sont pas surchargées et cela explique pourquoi `rollback` est rapide. Avec un `update`, la nouvelle ligne résultat est insérée dans la table (après avoir enlevé le ctid) et, dans le nouvel en-tête de ligne de l'ancienne ligne, vers où pointe le ctid, les entrées `cmax` et `xmax` sont configurées par le compteur de commande actuel et par l'identifiant de transaction actuel. Du coup, l'ancienne ligne est cachée et, après validation de la transaction, le nettoyeur (vacuum) peut éventuellement la supprimer.

Connaissant tout ceci, nous pouvons simplement appliquer les règles de vues de la même façon que toute autre commande. Il n'y a pas de différence.

41.2.3. Puissance des vues dans PostgreSQL

L'exemple ci-dessus démontre l'incorporation des définitions de vues par le système de règles dans l'arbre de requête original. Dans le deuxième exemple, un simple `select` d'une vue a créé un arbre de requête final qui est une jointure de quatre tables (`unit` a été utilisé deux fois avec des noms différents).

Le bénéfice de l'implémentation des vues avec le système de règles est que le planificateur a toute l'information sur les tables à parcourir et sur les relations entre ces tables et les qualifications restrictives à partir des vues et les qualifications à partir de la requête originale dans un seul arbre de requête. Et c'est toujours la situation quand la requête originale est déjà une jointure sur des vues. Le planificateur doit décider du meilleur chemin pour exécuter la requête et plus le planificateur a d'informations, meilleure sera la décision. Le système de règles implémenté dans PostgreSQL s'en assure, c'est toute l'information disponible sur la requête à ce moment.

41.2.4. Mise à jour d'une vue

Qu'arrive-t'il si une vue est nommée comme la relation cible d'un `insert`, `update` ou `delete` ? Faire simplement les substitutions décrites ci-dessus donnerait un arbre de requêtes dont le résultat pointerait vers une entrée de la table en sous-requête. Cela ne fonctionnera pas. Néanmoins, il existe

différents moyens permettant à PostgreSQL de supporter la mise à jour d'une vue. Dans l'ordre de complexité pour l'expérience utilisateur : substituer automatiquement la table sous-jacente par la vue, exécuter un trigger utilisateur, ou réécrire la requête par une règle utilisateur. Ces options sont discutées ci-dessous.

Si la sous-requête fait une sélection à partir d'une relation simple et qu'elle est suffisamment simple, le processus de réécriture peut automatiquement remplacé la sous-requête avec la relation sous-jacente pour que l'`INSERT`, l'`UPDATE` ou le `DELETE` soit appliqué correctement sur la relation de base. Les vues qui sont « suffisamment simples » pour cela sont appelées des vues *automatiquement modifiables*. Pour des informations détaillées sur ce type de vue, voir CREATE VIEW.

Sinon, l'opération peut être gérée par un trigger `INSTEAD OF`, créé par l'utilisateur, sur la vue (see CREATE TRIGGER). La réécriture fonctionne légèrement différemment dans ce cas. Pour `INSERT`, la réécriture ne fait rien du tout avec la vue, la laissant comme relation résultante de la requête. Pour `UPDATE` et `DELETE`, il est toujours nécessaire d'étendre la requête de la vue pour récupérer les « anciennes » lignes que la commande va essayer de mettre à jour ou supprimer. Donc la vue est étendue comme d'habitude mais une autre entrée de table non étendue est ajoutée à la requête pour représenter la vue en tant que relation résultante.

Le problème qui survient maintenant est d'identifier les lignes à mettre à jour dans la vue. Rappelez-vous que, quand la relation résultante est une table, une entrée CTID spéciale est ajoutée à la liste cible pour identifier les emplacements physiques des lignes à mettre à jour. Ceci ne fonctionne pas si la relation résultante est une vue car une vue n'a pas de CTID, car ses lignes n'ont pas d'emplacements physiques réels. À la place, pour une opération `UPDATE` ou `DELETE`, une entrée `wholerow` (ligne complète) spéciale est ajoutée à la liste cible, qui s'étend pour inclure toutes les colonnes d'une vue. L'exécuteur utilise cette valeur pour fournir l'« ancienne » ligne au trigger `INSTEAD OF`. C'est au trigger de savoir ce que la mise à jour est supposée faire sur les valeurs des anciennes et nouvelles lignes.

Une autre possibilité est que l'utilisateur définisse des vues `INSTEAD` qui indiquent les actions à substituer pour les commandes `INSERT`, `UPDATE` et `DELETE` sur une vue. Ces règles vont réécrire la commande, typiquement en une commande qui met à jour une ou plusieurs tables, plutôt que des vues. C'est le thème de Section 41.4.

Notez que les règles sont évaluées en premier, réécrivant la requête originale avant qu'elle ne soit optimisée et exécutée. Du coup, si une vue a des triggers `INSTEAD OF` en plus de règles sur `INSERT`, `UPDATE` ou `DELETE`, alors les règles seront évaluées en premier et, suivant le résultat, les triggers pourraient être utilisés.

La réécriture automatique d'une requête `INSERT`, `UPDATE` ou `DELETE` sur une vue simple est toujours essayée en dernier. Du coup, si une vue a des règles ou des triggers, ces derniers surchargeront le comportement par défaut des vues automatiquement modifiables.

S'il n'y a pas de règles `INSTEAD` ou de triggers `INSTEAD OF` sur la vue et que le processus de réécriture ne peut pas réécrire automatiquement la requête sous la forme d'une mise à jour de la relation sous-jacente, une erreur sera renvoyée car l'exécuteur ne peut pas modifier une vue.

41.3. Vues matérialisées

Les vues matérialisées dans PostgreSQL utilisent le système des règles, tout comme les vues, mais les résultats persistent sous la forme d'une table. Les principales différences entre :

```
CREATE MATERIALIZED VIEW ma_vue_mat AS SELECT * FROM ma_table;
```

et :

```
CREATE TABLE ma_vue_mat AS SELECT * FROM ma_table;
```

sont que la vue matérialisée ne peut pas être directement mise à jour et que la requête utilisée pour créer la vue matérialisée est enregistrée exactement de la même façon qu'une requête d'une vue standard. Des données fraiches peuvent être générées pour la vue matérialisée avec cette commande :

```
REFRESH MATERIALIZED VIEW ma_vue_mat;
```

L'information sur une vue matérialisée est stockée dans les catalogues systèmes de PostgreSQL exactement de la même façon que pour les tables et les vues. Quand une vue matérialisée est référencée dans une requête, les données sont renvoyées directement à partir de la vue matérialisée, tout comme une table ; la règle est seulement utilisée pour peupler la vue matérialisée.

Bien que l'accès aux données d'une vue matérialisée est souvent bien plus rapide qu'accèder aux tables sous-jacentes directement ou par l'intermédiaire d'une vue, les données ne sont pas toujours fraiches. Cependant, quelques fois, des données plus fraiches ne sont pas nécessaires. Considérez une table qui enregistre les ventes :

```
CREATE TABLE facture (
    no_facture    integer        PRIMARY KEY,
    no_vendeur    integer,       -- identifiant du vendeur
    date_facture  date,          -- date de la vente
    mtt_facture   numeric(13,2)  -- montant de la vente
);
```

Si des personnes souhaitent grapher rapidement les données de vente, elles peuvent vouloir résumer l'information et ne pas avoir besoin des données incomplètes du jour :

```
CREATE MATERIALIZED VIEW resume_ventes AS
  SELECT
      no_vendeur,
      date_facture,
      sum(mtt_facture)::numeric(13,2) as mtt_ventes
    FROM facture
    WHERE date_facture < CURRENT_DATE
    GROUP BY
      no_vendeur,
      date_facture;

CREATE UNIQUE INDEX ventes_resume_vendeur
  ON resume_ventes (no_vendeur, date_facture);
```

Cette vue matérialisée peut être utile pour afficher un graphe dans l'affichage créé pour les vendeurs. Une tâche de fond pourrait être planifiée pour mettre à jour les statistiques chaque nuit en utilisant cette requête SQL :

```
REFRESH MATERIALIZED VIEW resume_ventes;
```

Une autre utilisation des vues matérialisées est de permettre un accès rapide aux données provenant d'un système distant, au travers d'un wrapper de données distantes. Un exemple utilisant `file_fdw`

est donné ci-dessous, avec des chronométrages mais comme cela utilise le cache du système local, les performances comparées à l'accès à un système distant seront supérieures à celles montrées ici. Notez que nous exploitons aussi la capacité à placer un index sur la vue matérialisée alors que `file_fdw` n'autorise pas les index ; cet avantage pourrait ne pas s'appliquer pour d'autres types d'accès à des données distantes.

Configuration ::

```
CREATE EXTENSION file_fdw;
CREATE SERVER fichier_local FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE mots (mot text NOT NULL)
  SERVER fichier_local
  OPTIONS (filename '/usr/share/dict/words');
CREATE MATERIALIZED VIEW vmots AS SELECT * FROM mots;
CREATE UNIQUE INDEX idx_vmots ON vmots (mot);
CREATE EXTENSION pg_trgm;
CREATE INDEX vmots_trgm ON vmots USING gist (mot gist_trgm_ops);
VACUUM ANALYZE vmots;
```

Maintenant, vérifions un mot. En utilisant `file_fdw` directement :

```
SELECT count(*) FROM mots WHERE mot = 'caterpiler';

 count
-------
     0
(1 row)
```

Avec `EXPLAIN ANALYZE`, nous voyons :

```
 Aggregate  (cost=21763.99..21764.00 rows=1 width=0) (actual
 time=188.180..188.181 rows=1 loops=1)
   ->  Foreign Scan on words  (cost=0.00..21761.41 rows=1032
 width=0) (actual time=188.177..188.177 rows=0 loops=1)
         Filter: (word = 'caterpiler'::text)
         Rows Removed by Filter: 479829
         Foreign File: /usr/share/dict/words
         Foreign File Size: 4953699
 Planning time: 0.118 ms
 Execution time: 188.273 ms
```

Si la vue matérialisée est utilisée à la place, la requête est bien plus rapide :

```
 Aggregate  (cost=4.44..4.45 rows=1 width=0) (actual
 time=0.042..0.042 rows=1 loops=1)
   ->  Index Only Scan using wrd_word on wrd  (cost=0.42..4.44
 rows=1 width=0) (actual time=0.039..0.039 rows=0 loops=1)
         Index Cond: (word = 'caterpiler'::text)
         Heap Fetches: 0
 Planning time: 0.164 ms
 Execution time: 0.117 ms
```

Dans les deux cas, le mot est mal orthographié. Donc cherchons le bon mot. Toujours en utilisant `file_fdw` et `pg_trgm` :

```
SELECT mot FROM mots ORDER BY mot <-> 'caterpiler' LIMIT 10;

     mot
---------------
 cater
 caterpillar
 Caterpillar
 caterpillars
 caterpillar's
 Caterpillar's
 caterer
 caterer's
 caters
 catered
(10 rows)
```

```
 Limit  (cost=11583.61..11583.64 rows=10 width=32) (actual
 time=1431.591..1431.594 rows=10 loops=1)
   ->  Sort  (cost=11583.61..11804.76 rows=88459 width=32) (actual
 time=1431.589..1431.591 rows=10 loops=1)
         Sort Key: ((word <-> 'caterpiler'::text))
          Sort Method: top-N heapsort  Memory: 25kB
         ->  Foreign Scan on words  (cost=0.00..9672.05 rows=88459
 width=32) (actual time=0.057..1286.455 rows=479829 loops=1)
               Foreign File: /usr/share/dict/words
               Foreign File Size: 4953699
 Planning time: 0.128 ms
 Execution time: 1431.679 ms
```

Et en utilisant la vue matérialisée :

```
 Limit  (cost=0.29..1.06 rows=10 width=10) (actual
 time=187.222..188.257 rows=10 loops=1)
   ->  Index Scan using wrd_trgm on wrd  (cost=0.29..37020.87
 rows=479829 width=10) (actual time=187.219..188.252 rows=10
 loops=1)
         Order By: (word <-> 'caterpiler'::text)
 Planning time: 0.196 ms
 Execution time: 198.640 ms
```

Si vous pouvez tolérer des mises à jour périodiques sur les données distantes pour votre base locale, les bénéfices en performance seront importants.

41.4. Règles sur `insert`, `update` et `delete`

Les règles définies sur `insert`, `update` et `delete` sont significativement différentes des règles de vue décrites dans la section précédente. Tout d'abord, leur commande `create rule` permet plus de choses :

- Elles peuvent n'avoir aucune action.

- Elles peuvent avoir plusieurs actions.
- Elles peuvent être de type `instead` ou `also` (valeur par défaut).
- Les pseudo relations `new` et `old` deviennent utiles.
- Elles peuvent avoir des qualifications de règles.

Ensuite, elles ne modifient pas l'arbre de requête en place. À la place, elles créent de nouveaux arbres de requêtes et peuvent abandonner l'original.

> **Attention**
>
> Dans de nombreux cas, les tâches réalisables par des règles sur des `INSERT`/`UPDATE`/`DELETE` sont mieux réalisés avec des triggers. Les triggers ont une notation un peu plus complexe mais leur sémantique est plus simple à comprendre. Les règles peuvent avoir des résultats surprenants quand la requête originale contient des fonctions volatiles : les fonctions volatiles pourraient être exécutées plus de fois qu'escompté lors du traitement de la règle.
>
> De plus, il existe aussi certains cas non supportés par ces types de règles, ceci incluant notamment les clauses `WITH` dans la requête originale et les sous-requêtes (sous `SELECT`) dans la liste `SET` de requêtes `UPDATE`. Ceci est dû au fait que la copie de ces constructions dans la requête d'une règle pourrait résulter en des évaluations multiples de la sous-requête, contrairement à l'intention réelle de l'auteur de la requête.

41.4.1. Fonctionnement des règles de mise à jour

Gardez en tête la syntaxe :

```
CREATE [ OR REPLACE ] RULE nom as on evenement
    TO table [ where condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | commande | ( commande
 ; commande ... ) }
```

Dans la suite, *règles de mise à jour* signifie les règles qui sont définies sur `insert`, `update` ou `delete`.

Les règles de mise à jour sont appliquées par le système de règles lorsque la relation résultante et le type de commande d'un arbre de requête sont égaux pour l'objet et l'événement donné dans la commande `create RULE`. Pour les règles de mise à jour, le système de règles crée une liste d'arbres de requêtes. Initialement, la liste d'arbres de requêtes est vide. Il peut y avoir aucune (mot clé `nothing`), une ou plusieurs actions. Pour simplifier, nous verrons une règle avec une action. Cette règle peut avoir une qualification et peut être de type `instead` ou `also` (valeur par défaut).

Qu'est-ce qu'une qualification de règle ? C'est une restriction indiquant le moment où doivent être réalisés les actions de la règle. Cette qualification peut seulement référencer les pseudo relations `new` et/ou `old`, qui représentent basiquement la relation qui a été donné comme objet (mais avec une signification spéciale).

Donc, nous avons trois cas qui produisent les arbres de requêtes suivants pour une règle à une seule action.

sans qualification avec soit `ALSO` soit `INSTEAD`

> l'arbre de requête à partir de l'action de la règle avec l'ajout de la qualification de l'arbre de requête original

qualification donnée et `also`

l'arbre de requête à partir de l'action de la règle avec l'ajout de la qualification de la règle et de la qualification de l'arbre de requête original

qualification donnée avec `instead`

l'arbre de requête à partir de l'action de la règle avec la qualification de la requête et la qualification de l'arbre de requête original ; et l'ajout de l'arbre de requête original avec la qualification inverse de la règle

Enfin, si la règle est `also`, l'arbre de requête original est ajouté à la liste. Comme seules les règles qualifiées `instead` ont déjà ajouté l'arbre de requête original, nous finissons avec un ou deux arbres de requête en sortie pour une règle avec une action.

Pour les règles `on insert`, la requête originale (si elle n'est pas supprimée par `instead`) est réalisée avant toute action ajoutée par les règles. Ceci permet aux actions de voir les lignes insérées. Mais pour les règles `on update` et `on delete`, la requête originale est réalisée après les actions ajoutées par les règles. Ceci nous assure que les actions pourront voir les lignes à mettre à jour ou à supprimer ; sinon, les actions pourraient ne rien faire parce qu'elles ne trouvent aucune ligne correspondant à leurs qualifications.

Les arbres de requêtes générés à partir des actions de règles sont envoyés de nouveau dans le système de réécriture et peut-être que d'autres règles seront appliquées résultant en plus ou moins d'arbres de requêtes. Donc, les actions d'une règle doivent avoir soit un type de commande différent soit une relation résultante différente de celle où la règle elle-même est active, sinon ce processus récursif se terminera dans une boucle infinie. (L'expansion récursive d'une règle sera détectée et rapportée comme une erreur.)

Les arbres de requête trouvés dans les actions du catalogue système `pg_rewrite` sont seulement des modèles. Comme ils peuvent référencer les entrées de la table d'échelles pour `new` et `old`, quelques substitutions ont dû être faites avant qu'elles ne puissent être utilisées. Pour toute référence de `new`, une entrée correspondante est recherchée dans la liste cible de la requête originale. Si elle est trouvée, cette expression de l'entrée remplace la référence. Sinon, `new` signifie la même chose que `old` (pour un `update`) où est remplacé par une valeur null (pour un `insert`). Toute référence à `old` est remplacée par une référence à l'entrée de la table d'échelles qui est la relation résultante.

Après que le système a terminé d'appliquer des règles de mise à jour, il applique les règles de vues pour le(s) arbre(s) de requête produit(s). Les vues ne peuvent pas insérer de nouvelles actions de mise à jour, donc il n'est pas nécessaire d'appliquer les règles de mise à jour à la sortie d'une réécriture de vue.

41.4.1.1. Une première requête étape par étape

Disons que nous voulons tracer les modifications dans la colonne `dispo_lacet` de la relation `donnees_lacet`. donc, nous allons configurer une table de traces et une règle qui va écrire une entrée lorsqu'un `update` est lancé sur `donnees_lacet`.

```
CREATE TABLE lacet_log (
    nom_lacet    text,          -- modification de lacet
    dispo_lacet   integer,       -- nouvelle valeur disponible
    log_who    text,          -- qui l'a modifié
    log_when   timestamp      -- quand
);

CREATE RULE log_lacet AS ON UPDATE TO donnees_lacet
    WHERE NEW.dispo_lacet <> OLD.dispo_lacet
    DO INSERT INTO lacet_log VALUES (
                                    NEW.nom_lacet,
                                    NEW.dispo_lacet,
                                    current_user,
```

```
                                    current_timestamp
                                );
```

Maintenant, quelqu'un exécute :

```
UPDATE donnees_lacet SET dispo_lacet = 6 WHERE nom_lacet = 'sl7';
```

et voici le contenu de la table des traces :

```
SELECT * FROM lacet_log;

 nom_lacet | dispo_lacet | log_who | log_when
-----------+-------------+---------
+----------------------------------
 sl7       |           6 | Al      | Tue Oct 20 16:14:45 1998 MET
 DST
(1 row)
```

C'est ce à quoi nous nous attendions. Voici ce qui s'est passé en tâche de fond. L'analyseur a créé l'arbre de requête :

```
UPDATE donnees_lacet SET dispo_lacet = 6
  FROM donnees_lacet donnees_lacet
 WHERE donnees_lacet.nom_lacet = 'sl7';
```

Il existe une règle `log_lacet` qui est `on UPDATE` avec l'expression de qualification de la règle :

```
NEW.dispo_lacet <> OLD.dispo_lacet
```

et l'action :

```
INSERT INTO lacet_log VALUES (
       new.nom_lacet, new.dispo_lacet,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old;
```

(ceci semble un peu étrange car, normalement, vous ne pouvez pas écrire `insert … values … from`. Ici, la clause `from` indique seulement qu'il existe des entrées de la table d'échelles dans l'arbre de requête pour `new` et `old`. Elles sont nécessaires pour qu'elles puissent être référencées par des variables dans l'arbre de requête de la commande `insert`).

La règle est une règle qualifiée `also` de façon à ce que le système de règles doit renvoyer deux arbres de requêtes : l'action de la règle modifiée et l'arbre de requête original. Dans la première étape, la table d'échelles de la requête originale est incorporée dans l'arbre de requête d'action de la règle. Ceci a pour résultat :

```
INSERT INTO lacet_log VALUES (
       new.nom_lacet, new.dispo_lacet,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old,
       donnees_lacet donnees_lacet;
```

Pour la deuxième étape, la qualification de la règle lui est ajoutée, donc l'ensemble de résultat est restreint aux lignes où `dispo_lacet` a changé :

```
INSERT INTO lacet_log VALUES (
       new.nom_lacet, new.dispo_lacet,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old,
       donnees_lacet donnees_lacet
 where new.dispo_lacet <> old.dispo_lacet;
```

(Ceci semble encore plus étrange car `insert ... values` n'a pas non plus une clause `where` mais le planificateur et l'exécuteur n'auront pas de difficultés avec ça. Ils ont besoin de supporter cette même fonctionnalité pour `insert ... select`.)

À l'étape 3, la qualification de l'arbre de requête original est ajoutée, restreignant encore plus l'ensemble de résultats pour les seules lignes qui auront été modifiées par la requête originale :

```
INSERT INTO lacet_log VALUES (
       new.nom_lacet, new.dispo_lacet,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old,
       donnees_lacet donnees_lacet
 WHERE new.dispo_lacet <> old.dispo_lacet
   and donnees_lacet.nom_lacet = 'sl7';
```

La quatrième étape remplace les références à `new` par les entrées de la liste cible à partir de l'arbre de requête original ou par les références de la variable correspondante à partir de la relation résultat :

```
INSERT INTO lacet_log VALUES (
       donnees_lacet.nom_lacet, 6,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old,
       donnees_lacet donnees_lacet
 WHERE 6 <> old.dispo_lacet
   AND donnees_lacet.nom_lacet = 'sl7';
```

L'étape 5 modifie les références `old` en référence de la relation résultat :

```
INSERT INTO lacet_log VALUES (
       donnees_lacet.nom_lacet, 6,
       current_user, current_timestamp )
  FROM donnees_lacet new, donnees_lacet old,
       donnees_lacet donnees_lacet
 WHERE 6 <> donnees_lacet.dispo_lacet
   AND donnees_lacet.nom_lacet = 'sl7';
```

C'est tout. Comme la règle est de type `also`, nous affichons aussi l'arbre de requêtes original. En bref, l'affichage à partir du système de règles est une liste de deux arbres de requêtes est une liste de deux arbres de requêtes correspondant à ces instructions :

```
INSERT INTO lacet_log VALUES (
       donnees_lacet.nom_lacet, 6,
       current_user, current_timestamp )
  FROM donnees_lacet
 WHERE 6 <> donnees_lacet.dispo_lacet
   AND donnees_lacet.nom_lacet = 'sl7';

UPDATE donnees_lacet SET dispo_lacet = 6
 WHERE nom_lacet = 'sl7';
```

Elles sont exécutées dans cet ordre et c'est exactement le but de la règle.

Les substitutions et les qualifications ajoutées nous assurent que, si la requête originale était :

```
UPDATE donnees_lacet SET couleur_lacet = 'green'
 WHERE nom_lacet = 'sl7';
```

aucune trace ne serait écrite. Dans ce cas, l'arbre de requête original ne contient pas une entrée dans la liste cible pour `dispo_lacet`, donc `new.dispo_lacet` sera remplacé par `donnees_lacet.dispo_lacet`. Du coup, la commande supplémentaire générée par la règle est :

```
INSERT INTO lacet_log VALUES (
       donnees_lacet.nom_lacet, donnees_lacet.dispo_lacet,
       current_user, current_timestamp )
  FROM donnees_lacet
 WHERE donnees_lacet.dispo_lacet <> donnees_lacet.dispo_lacet
   AND donnees_lacet.nom_lacet = 'sl7';
```

et la qualification ne sera jamais vraie.

Si la requête originale modifie plusieurs lignes, cela fonctionne aussi. Donc, si quelqu'un a lancé la commande :

```
UPDATE donnees_lacet SET dispo_lacet = 0
 WHERE couleur_lacet = 'black';
```

en fait, quatre lignes sont modifiées (`sl1`, `sl2`, `sl3` et `sl4`). Mais `sl3` a déjà `dispo_lacet = 0`. dans ce cas, la qualification des arbres de requêtes originaux sont différents et cela produit un arbre de requête supplémentaire :

```
INSERT INTO lacet_log
SELECT donnees_lacet.nom_lacet, 0,
       current_user, current_timestamp
  FROM donnees_lacet
 WHERE 0 <> donnees_lacet.dispo_lacet
   AND donnees_lacet.couleur_lacet = 'black';
```

à générer par la règle. Cet arbre de requête aura sûrement inséré trois nouvelles lignes de traces. Et c'est tout à fait correct.

Ici, nous avons vu pourquoi il est important que l'arbre de requête original soit exécuté en premier. Si l'`update` a été exécuté avant, toutes les lignes pourraient aussi être initialisées à zéro, donc le `insert` tracé ne trouvera aucune ligne à `0 <> donnees_lacet.dispo_lacet`.

41.4.2. Coopération avec les vues

Une façon simple de protéger les vues d'une exécution d'`insert`, d'`update` ou de `delete` sur elles est de laisser s'abandonner ces arbres de requête. Donc, nous pourrions créer les règles :

```
CREATE RULE chaussure_ins_protect AS ON INSERT TO chaussure
    DO INSTEAD NOTHING;
CREATE RULE chaussure_upd_protect AS ON UPDATE TO chaussure
    DO INSTEAD NOTHING;
```

```
CREATE RULE chaussure_del_protect AS ON DELETE TO chaussure
    DO INSTEAD NOTHING;
```

Maintenant, si quelqu'un essaie de faire une de ces opérations sur la vue `chaussure`, le système de règles appliquera ces règles. Comme les règles n'ont pas d'action et sont de type `instead`, la liste résultante des arbres de requêtes sera vide et la requête entière deviendra vide car il ne reste rien à optimiser ou exécuter après que le système de règles en ait terminé avec elle.

Une façon plus sophistiquée d'utiliser le système de règles est de créer les règles qui réécrivent l'arbre de requête en un arbre faisant la bonne opération sur les vraies tables. Pour réaliser cela sur la vue `lacet`, nous créons les règles suivantes :

```
CREATE RULE lacet_ins AS ON INSERT TO lacet
    DO INSTEAD
    INSERT INTO donnees_lacet VALUES (
           NEW.nom_lacet,
           NEW.dispo_lacet,
           NEW.couleur_lacet,
           NEW.longueur_lacet,
           NEW.unite_lacet
    );

CREATE RULE lacet_upd AS ON UPDATE TO lacet
    DO INSTEAD
    UPDATE donnees_lacet
       SET nom_lacet = NEW.nom_lacet,
           dispo_lacet = NEW.dispo_lacet,
           couleur_lacet = NEW.couleur_lacet,
           longueur_lacet = NEW.longueur_lacet,
           unite_lacet = NEW.unite_lacet
     WHERE nom_lacet = OLD.nom_lacet;

CREATE RULE lacet_del AS ON DELETE TO lacet
    DO INSTEAD
    DELETE FROM donnees_lacet
     WHERE nom_lacet = OLD.nom_lacet;
```

Si vous voulez supporter les requêtes `RETURNING` sur la vue, vous devrez faire en sorte que les règles incluent les clauses `RETURNING` qui calcule les lignes de la vue. Ceci est assez simple pour des vues sur une seule table mais cela devient rapidement complexe pour des vues de jointure comme `lacet`. Voici un exemple pour le cas d'un INSERT :

```
CREATE RULE lacet_ins AS ON INSERT TO lacet
    DO INSTEAD
    INSERT INTO donnees_lacet VALUES (
           NEW.nom_lacet,
    NEW.dispo_lacet,
    NEW.couleur_lacet,
    NEW.longueur_lacet,
    NEW.unite_lacet
    )
    RETURNING
           donnees_lacet.*,
    (SELECT donnees_lacet.longueur_lacet * u.facteur_unite
            FROM unite u WHERE donnees_lacet.unite_lacet =
 u.nom_unite);
```

Notez que cette seule règle supporte à la fois les `INSERT` et les `INSERT RETURNING` sur la vue -- la clause `RETURNING` est tout simplement ignoré pour un `INSERT`.

Maintenant, supposons que, quelque fois, un paquet de lacets arrive au magasin avec une grosse liste. Mais vous ne voulez pas mettre à jour manuellement la vue `lacet` à chaque fois. à la place, nous configurons deux petites tables, une où vous pouvez insérer les éléments de la liste et une avec une astuce spéciale. Voici les commandes de création :

```
CREATE TABLE lacet_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE lacet_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE lacet_ok_ins AS ON INSERT TO lacet_ok
    DO INSTEAD
    UPDATE lacet
       SET dispo_lacet = dispo_lacet + NEW.ok_quant
     WHERE nom_lacet = NEW.ok_name;
```

Maintenant, vous pouvez remplir la table `lacet_arrive` avec les données de la liste :

```
SELECT * FROM lacet_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

Jetez un œil rapidement aux données actuelles :

```
SELECT * FROM lacet;

 nom_lacet  | dispo_lacet | couleur_lacet | longueur_lacet |
 unite_lacet | longueur_lacet_cm
------------+-------------+---------------+----------------
+-------------+-------------------
 sl1        |           5 | black         |             80 | cm
      |                80
 sl2        |           6 | black         |            100 | cm
      |               100
 sl7        |           6 | brown         |             60 | cm
      |                60
 sl3        |           0 | black         |             35 | inch
      |              88.9
 sl4        |           8 | black         |             40 | inch
      |             101.6
 sl8        |           1 | brown         |             40 | inch
      |             101.6
 sl5        |           4 | brown         |              1 | m
      |               100
```

```
 sl6        |           0 | brown         |            0.9 | m
      |                90
(8 rows)
```

Maintenant, déplacez les lacets arrivés dans :

```
INSERT INTO lacet_ok SELECT * FROM lacet_arrive;
```

et vérifiez le résultat :

```
SELECT * FROM lacet ORDER BY nom_lacet;

 nom_lacet  | dispo_lacet | couleur_lacet | longueur_lacet |
 unite_lacet | longueur_lacet_cm
------------+-------------+---------------+----------------
+-------------+-------------------
 sl1        |           5 | black         |             80 | cm
      |                80
 sl2        |           6 | black         |            100 | cm
      |               100
 sl7        |           6 | brown         |             60 | cm
      |                60
 sl4        |           8 | black         |             40 | inch
      |             101.6
 sl3        |          10 | black         |             35 | inch
      |              88.9
 sl8        |          21 | brown         |             40 | inch
      |             101.6
 sl5        |           4 | brown         |              1 | m
      |               100
 sl6        |          20 | brown         |            0.9 | m
      |                90
(8 rows)

SELECT * FROM lacet_log;

 nom_lacet | dispo_lacet | log_who| log_when
-----------+-------------+--------
+---------------------------------
 sl7       |           6 | Al     | Tue Oct 20 19:14:45 1998 MET
 DST
 sl3       |          10 | Al     | Tue Oct 20 19:25:16 1998 MET
 DST
 sl6       |          20 | Al     | Tue Oct 20 19:25:16 1998 MET
 DST
 sl8       |          21 | Al     | Tue Oct 20 19:25:16 1998 MET
 DST
(4 rows)
```

C'est un long chemin du `insert ... select` à ces résultats. Et la description de la transformation de l'arbre de requêtes sera la dernière dans ce chapitre. Tout d'abord, voici la sortie de l'analyseur :

```
INSERT INTO lacet_ok
SELECT lacet_arrive.arr_name, lacet_arrive.arr_quant
  FROM lacet_arrive lacet_arrive, lacet_ok lacet_ok;
```

Maintenant, la première règle `lacet_ok_ins` est appliquée et transforme ceci en :

```
UPDATE lacet
   SET dispo_lacet = lacet.dispo_lacet + lacet_arrive.arr_quant
  FROM lacet_arrive lacet_arrive, lacet_ok lacet_ok,
       lacet_ok old, lacet_ok new,
       lacet lacet
 WHERE lacet.nom_lacet = lacet_arrive.arr_name;
```

et jette l'`insert` actuel sur `lacet_ok`. La requête réécrite est passée de nouveau au système de règles et la seconde règle appliquée `lacet_upd` produit :

```
UPDATE donnees_lacet
   SET nom_lacet = lacet.nom_lacet,
       dispo_lacet = lacet.dispo_lacet + lacet_arrive.arr_quant,
       couleur_lacet = lacet.couleur_lacet,
       longueur_lacet = lacet.longueur_lacet,
       unite_lacet = lacet.unite_lacet
  FROM lacet_arrive lacet_arrive, lacet_ok lacet_ok,
       lacet_ok old, lacet_ok new,
       lacet lacet, lacet old,
       lacet new, donnees_lacet donnees_lacet
 WHERE lacet.nom_lacet = lacet_arrive.arr_name
   AND donnees_lacet.nom_lacet = lacet.nom_lacet;
```

De nouveau, il s'agit d'une règle `instead` et l'arbre de requête précédent est jeté. Notez que cette requête utilise toujours la vue `lacet`. Mais le système de règles n'a pas fini cette étape, donc il continue et lui applique la règle `_return`. Nous obtenons :

```
UPDATE donnees_lacet
   SET nom_lacet = s.nom_lacet,
       dispo_lacet = s.dispo_lacet + lacet_arrive.arr_quant,
       couleur_lacet = s.couleur_lacet,
       longueur_lacet = s.longueur_lacet,
       unite_lacet = s.unite_lacet
  FROM lacet_arrive lacet_arrive, lacet_ok lacet_ok,
       lacet_ok old, lacet_ok new,
       lacet lacet, lacet old,
       lacet new, donnees_lacet donnees_lacet,
       lacet old, lacet new,
       donnees_lacet s, unit u
 WHERE s.nom_lacet = lacet_arrive.arr_name
   AND donnees_lacet.nom_lacet = s.nom_lacet;
```

Enfin, la règle `log_lacet` est appliquée, produisant l'arbre de requête supplémentaire :

```
INSERT INTO lacet_log
SELECT s.nom_lacet,
       s.dispo_lacet + lacet_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM lacet_arrive lacet_arrive, lacet_ok lacet_ok,
       lacet_ok old, lacet_ok new,
       lacet lacet, lacet old,
       lacet new, donnees_lacet donnees_lacet,
       lacet old, lacet new,
```

```
       donnees_lacet s, unit u,
       donnees_lacet old, donnees_lacet new
       lacet_log lacet_log
 WHERE s.nom_lacet = lacet_arrive.arr_name
   AND donnees_lacet.nom_lacet = s.nom_lacet
   AND (s.dispo_lacet + lacet_arrive.arr_quant) <> s.dispo_lacet;
```

une fois que le système de règles tombe en panne de règles et renvoie les arbres de requêtes générés.

Donc, nous finissons avec deux arbres de requêtes finaux qui sont équivalents aux instructions SQL :

```
INSERT INTO lacet_log
SELECT s.nom_lacet,
       s.dispo_lacet + lacet_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM lacet_arrive lacet_arrive, donnees_lacet donnees_lacet,
       donnees_lacet s
 WHERE s.nom_lacet = lacet_arrive.arr_name
   AND donnees_lacet.nom_lacet = s.nom_lacet
   AND s.dispo_lacet + lacet_arrive.arr_quant <> s.dispo_lacet;

UPDATE donnees_lacet
   SET dispo_lacet = donnees_lacet.dispo_lacet +
 lacet_arrive.arr_quant
  FROM lacet_arrive lacet_arrive,
       donnees_lacet donnees_lacet,
       donnees_lacet s
 WHERE s.nom_lacet = lacet_arrive.nom_lacet
   AND donnees_lacet.nom_lacet = s.nom_lacet;
```

Le résultat est que la donnée provenant d'une relation insérée dans une autre, modifiée en mise à jour dans une troisième, modifiée en mise à jour dans une quatrième, cette dernière étant tracée dans une cinquième, se voit réduite à deux requêtes.

Il y a un petit détail assez horrible. En regardant les deux requêtes, nous nous apercevons que la relation `donnees_lacet` apparaît deux fois dans la table d'échelles où cela pourrait être réduit à une seule occurrence. Le planificateur ne gère pas ceci et, du coup, le plan d'exécution de la sortie du système de règles pour `insert` sera :

```
Nested Loop
     ->  Merge Join
     ->  Seq Scan
     ->  Sort
     ->  Seq Scan on s
     ->  Seq Scan
     ->  Sort
     ->  Seq Scan on lacet_arrive
     ->  Seq Scan on donnees_lacet
```

alors qu'omettre la table d'échelles supplémentaire résulterait en un :

```
Merge Join
     ->  Seq Scan
     ->  Sort
     ->  Seq Scan on s
```

```
     ->  Seq Scan
     ->  Sort
     ->  Seq Scan on lacet_arrive
```

qui produit exactement les mêmes entrées dans la table des traces. Du coup, le système de règles a causé un parcours supplémentaire dans la table `donnees_lacet` qui n'est absolument pas nécessaire. Et le même parcours redondant est fait une fois de plus dans l'`update`. Mais ce fut réellement un travail difficile de rendre tout ceci possible.

Maintenant, nous faisons une démonstration finale du système de règles de PostgreSQL et de sa puissance. disons que nous ajoutons quelques lacets avec des couleurs extraordinaires à votre base de données :

```
INSERT INTO lacet VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO lacet VALUES ('sl10', 1000, 'magenta', 40.0, 'inch',
 0.0);
```

Nous voulons créer une vue vérifiant les entrées `lacet` qui ne correspondent à aucune chaussure pour la couleur. Voici la vue :

```
CREATE VIEW lacet_mismatch AS
    SELECT * FROM lacet WHERE NOT EXISTS
        (SELECT nom_chaussure FROM chaussure WHERE couleur =
 couleur_lacet);
```

Sa sortie est :

```
SELECT * FROM lacet_mismatch;

 nom_lacet | dispo_lacet | couleur_lacet | longueur_lacet |
 unite_lacet | longueur_lacet_cm
-----------+-------------+---------------+----------------
+-------------+-------------------
 sl9       |           0 | pink          |             35 | inch
     |              88.9
 sl10      |        1000 | magenta       |             40 | inch
     |             101.6
```

Maintenant, nous voulons la configurer pour que les lacets qui ne correspondent pas et qui ne sont pas en stock soient supprimés de la base de données. Pour rendre la chose plus difficile à PostgreSQL, nous ne les supprimons pas directement. À la place, nous créons une vue supplémentaire :

```
CREATE VIEW lacet_can_delete AS
    SELECT * FROM lacet_mismatch WHERE dispo_lacet = 0;
```

et le faisons de cette façon :

```
DELETE FROM lacet WHERE EXISTS
    (SELECT * FROM lacet_can_delete
             WHERE nom_lacet = lacet.nom_lacet);
```

Les résultats sont :

```
SELECT * FROM lacet;
```

```
 nom_lacet | dispo_lacet | couleur_lacet | longueur_lacet |
 unite_lacet | longueur_lacet_cm
-----------+-------------+---------------+----------------
+-------------+------------------
 sl1       |           5 | black         |             80 | cm
     |                80
 sl2       |           6 | black         |            100 | cm
     |               100
 sl7       |           6 | brown         |             60 | cm
     |                60
 sl4       |           8 | black         |             40 | inch
     |             101.6
 sl3       |          10 | black         |             35 | inch
     |              88.9
 sl8       |          21 | brown         |             40 | inch
     |             101.6
 sl10      |        1000 | magenta       |             40 | inch
     |             101.6
 sl5       |           4 | brown         |              1 | m
     |               100
 sl6       |          20 | brown         |            0.9 | m
     |                90
(9 rows)
```

Un `delete` sur une vue, avec une qualification de sous-requête qui utilise au total quatre vues imbriquées/jointes, où l'une d'entre elles a une qualification de sous-requête contenant une vue et où les colonnes des vues calculées sont utilisées, est réécrite en un seul arbre de requête qui supprime les données demandées sur la vraie table.

Il existe probablement seulement quelques situations dans le vrai monde où une telle construction est nécessaire. Mais, vous vous sentez mieux quand cela fonctionne.

41.5. Règles et droits

À cause de la réécriture des requêtes par le système de règles de PostgreSQL, d'autres tables/vues que celles utilisées dans la requête originale pourraient être accédées. Lorsque des règles de mise à jour sont utilisées, ceci peut inclure des droits d'écriture sur les tables.

Les règles de réécriture n'ont pas de propriétaire séparé. Le propriétaire d'une relation (table ou vue) est automatiquement le propriétaire des règles de réécriture qui lui sont définies. Le système de règles de PostgreSQL modifie le comportement du système de contrôle d'accès par défaut. Avec l'exception des règles `SELECT` associées aux vues en mode *security invoker* (voir `CREATE VIEW`), toutes les relations utilisées pour la vérification des règles sont vérifiées de nouveau avec les droits du propriétaire de la règle, et non pas avec ceux de l'utilisateur ayant appelé la règle. Ceci signifie que, sauf pour les vues en mode *security invoker*, les utilisateurs ont seulement besoin des droits pour les tables/vues qui sont explicitement nommées dans leur requêtes.

Par exemple : un utilisateur a une liste de numéros de téléphone dont certains sont privés, les autres étant d'intérêt pour l'assistant du bureau. Il peut construire de cette façon :

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO assistant;
```

Personne en dehors de cet utilisateur (et les superutilisateurs de la base de données) ne peut accéder à la table phone_data. Mais, à cause du grant, l'assistant peut lancer un select sur la vue phone_number. Le système de règles réécrira le select sur phone_number en un select sur phone_data. Comme l'utilisateur est le propriétaire de phone_number et du coup le propriétaire de la règle, le droit de lecture de phone_data est maintenant vérifié avec ses propres privilèges et la requête est autorisée. La vérification de l'accès à phone_number est aussi réalisée mais ceci est fait avec l'utilisateur appelant, donc personne sauf l'utilisateur et l'assistant ne peut l'utiliser.

Les droits sont vérifiés règle par règle. Donc, l'assistant est actuellement le seul à pouvoir voir les numéros de téléphone publiques. Mais l'assistant peut configurer une autre vue et autoriser l'accès au public. Du coup, tout le monde peut voir les données de phone_number via la vue de l'assistant. Ce que l'assistant ne peut pas faire est de créer une vue qui accède directement à phone_data (en fait, il le peut mais cela ne fonctionnera pas car tous les accès seront refusés lors de la vérification des droits). Dès que l'utilisateur s'en rendra compte, du fait que l'assistant a ouvert la vue phone_number à tout le monde, il peut révoquer son accès. Immédiatement, tous les accès de la vue de l'assistant échoueront.

Il pourrait être dit que cette vérification règle par règle est une brèche de sécurité mais ce n'est pas le cas. Si cela ne fonctionne pas de cette façon, l'assistant pourrait copier une table avec les mêmes colonnes que phone_number et y copier les données une fois par jour. Du coup, ce sont ces propres données et il peut accorder l'accès à tout le monde si il le souhaite. Une commande grant signifie « j'ai confiance en vous ». Si quelqu'un en qui vous avez confiance se comporte ainsi, il est temps d'y réfléchir et d'utiliser revoke.

Notez que, bien que les vues puissent être utilisées pour cacher le contenu de certaines colonnes en utilisant la technique montrée ci-dessus, elles ne peuvent pas être utilisées de manière fiable pour cacher des données dans des lignes invisibles sauf si le drapeau security_barrier a été initialisé. Par exemple, la vue suivante n'est pas sécurisée :

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE
 '412%';
```

Cette vue peut sembler sécurisée car le système de règles va réécrire tout SELECT à partir de phone_number dans un SELECT à partir de phone_data et ajouter la qualification permettant de filter les enregistrements dont la colonne phone ne commence pas par 412. Mais si l'utilisateur peut créer ses propres fonctions, il n'est pas difficile de convaincre le planificateur d'exécuter la fonction définie par l'utilisateur avant l'expression NOT LIKE.

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END
$$ LANGUAGE plpgsql COST 0.0000000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

Chaque personne et chaque numéro de téléphone de la table phone_data sera affiché dans un NOTICE car le planificateur choisira d'exécuter la procédure tricky avant le NOT LIKE car elle est moins coûteuse. Même si l'utilisateur ne peut pas définir des nouvelles fonctions, les fonctions internes peuvent être utilisées pour des attaques similaires. (Par exemple, la plupart des fonctions de conversions affichent les valeurs en entrée dans le message d'erreur qu'elles fournissent.)

Des considérations similaires s'appliquent aussi aux règles de mise à jour. Dans les exemples de la section précédente, le propriétaire des tables de la base de données d'exemple pourrait accorder les droits select, insert, update et delete sur la vue lacet à quelqu'un d'autre mais seulement

`select` sur `lacet_log`. L'action de la règle pourrait écrire des entrées de trace qui seraient toujours exécutées avec succès et que l'autre utilisateur pourrait voir. Mais il ne peut pas créer de fausses entrées, pas plus qu'il ne peut manipuler ou supprimer celles qui existent. Dans ce cas, il n'existe pas de possibilité de subvertir les règles en convainquant le planificateur de modifier l'ordre des opérations car la seule règle qui fait référence à `shoelace_log` est un `INSERT` non qualifié. Ceci pourrait ne plus être vrai dans les scénarios complexes.

Lorsqu'il est nécessaire qu'une vue fournisse une sécurité au niveau des lignes, l'attribut `security_barrier` doit être appliqué à la vue. Ceci empêche des fonctions et des opérateurs choisis spécialement de voir des valeurs de lignes jusqu'à ce que la vue ait fait son travail. Par exemple, si la vue montrée ci-dessus a été créée ainsi, elle serait sécurisée :

```
CREATE VIEW phone_number WITH (security_barrier) AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE
 '412%';
```

Les vues créées avec l'attribut `security_barrier` peuvent avoir de bien pires performances que les vues créées sans cette option. En général, il n'y a pas de moyen de l'éviter : le plan le plus rapide doit être éviter si cela compromet la sécurité. Pour cette raison, cette option n'est pas activée par défaut.

Le planificateur de requêtes a plus de flexibilité lorsqu'il s'occupe de fonctions qui n'ont pas d'effets de bord. Ces fonctions sont qualifiées de `LEAKPROOF` et incluent de nombreux opérateurs simples fréquemment utilisés, comme les opérateurs d'égalité. Le planificateur de requêtes peut en tout sécurité permettre à de telles fonctions d'être évaluées à tout moment dans l'exécution de la requête car les appeler sur des lignes invisibles à l'utilisateur ne pourra pas faire transpirer ces informations sur les lignes invisibles. De plus, les fonctions qui ne prennent pas d'arguments ou à qui aucun argument n'est passé à partir de la vue disposant de l'option security_barrier n'ont pas besoin d'être marquées `LEAKPROOF` pour être exécutées avant car elles ne reçoivent jamais des données de la vue. En contraste complet, une fonction qui peut envoyer des erreurs dépendant des valeurs reçues en argument (comme les fonctions qui renvoient une erreur dans le cas d'un dépassement de capacité ou de division par zéro) ne sont pas `LEAKPROOF`, et risquent de fournir des informations sur les lignes invisibles si elles sont exécutées avant que la vue ne les filtre.

Il est important de comprendre que, même une vue créée avec l'option `security_barrier` est supposée être sécurisée dans le sens où le contenu de lignes invisibles ne sera pas passé à des fonctions supposées non sécurisées. L'utilisateur pourrait bien avoir d'autres moyens pour accéder aux données non vues ; par exemple, ils peuvent voir le plan d'exécution en utilisant `EXPLAIN` ou mesurer la durée d'exécution de requêtes sur la vue. Un attaquant pourrait être capable de deviner certaines informations comme la quantité de données invisibles, voire obtenir des informations sur la distribution des données ou les valeurs les plus communes (ces informations affectent la durée d'exécution de la requête ; ou même, comme elles font partie des statistiques de l'optimiseur, du choix du plan). Si ces types d'attaques vous posent problème, il est alors déconseillé de donner l'accès aux données.

41.6. Règles et statut de commande

Le serveur PostgreSQL renvoie une chaîne de statut de commande, comme `insert 149592 1`, pour chaque commande qu'il reçoit. C'est assez simple lorsqu'il n'y a pas de règles impliquées. Mais qu'arrive-t'il lorsque la requête est réécrite par des règles ?

Les règles affectent le statut de la commande de cette façon :

- S'il n'y a pas de règle `instead` inconditionnelle pour la requête, alors la requête donnée originellement sera exécutée et son statut de commande sera renvoyé comme d'habitude. (Mais notez que s'il y avait des règles `instead` conditionnelles, la négation de leur qualifications sera ajouté à la requête initiale. Ceci pourrait réduire le nombre de lignes qu'il traite et, si c'est le cas, le statut rapporté en sera affecté.)

- S'il y a des règles `instead` inconditionnelles pour la requête, alors la requête originale ne sera pas exécutée du tout. Dans ce cas, le serveur renverra le statut de la commande pour la dernière requête qui a été insérée par une règle `instead` (conditionnelle ou non) et est du même type de commande (`insert`, `update` ou `delete`) que la requête originale. Si aucune requête ne rencontrant ces pré-requis n'est ajoutée à une règle, alors le statut de commande renvoyé affiche le type de requête original et annule le compteur de ligne et le champ OID.

Le programmeur peut s'assurer que toute règle `instead` désirée est celle qui initialise le statut de commande dans le deuxième cas en lui donnant un nom de règle étant le dernier en ordre alphabétique parmi les règles actives pour qu'elle soit appliquée en dernier.

41.7. Règles contre triggers

Beaucoup de choses pouvant se faire avec des triggers peuvent aussi être implémentées en utilisant le système de règles de PostgreSQL. Un des points qui ne pourra pas être implémenté par les règles en certains types de contraintes, notamment les clés étrangères. Il est possible de placer une règle qualifiée qui réécrit une commande en `nothing` si la valeur d'une colonne n'apparaît pas dans l'autre table. Mais alors les données sont jetées et ce n'est pas une bonne idée. Si des vérifications de valeurs valides sont requises et dans le cas où il y a une erreur invalide, un message d'erreur devrait être généré et cela devra se faire avec un trigger.

Dans ce chapitre, nous avons ciblé l'utilisation des règles pour mettre à jour des vues. Tous les exemples de règles de mise à jour de ce chapitre peuvent aussi être implémentés en utilisant les triggers `INSTEAD OF` sur les vues. Écrire ce type de triggers est souvent plus facile qu'écrire des règles, tout particulièrement si une logique complexe est requise pour réaliser la mise à jour.

Pour les éléments qui peuvent être implémentés par les deux, ce qui sera le mieux dépend de l'utilisation de la base de données. Un trigger est exécuté une fois pour chaque ligne affectée. Une règle modifie la requête ou en génère une autre. Donc, si un grand nombre de lignes sont affectées pour une instruction, une règle lançant une commande supplémentaire sera certainement plus rapide qu'un trigger appelé pour chaque ligne et qui devra exécuter ces opérations autant de fois. Néanmoins, l'approche du trigger est conceptuellement plus simple que l'approche de la règle et est plus facile à utiliser pour les novices.

Ici, nous montrons un exemple où le choix d'une règle ou d'un trigger joue sur une situation. Voici les deux tables :

```
CREATE TABLE ordinateur (
    nom_hote        text,    -- indexé
    constructeur    text     -- indexé
);

CREATE TABLE logiciel (
    logiciel        text,    -- indexé
    nom_hote        text     -- indexé
);
```

Les deux tables ont plusieurs milliers de lignes et les index sur `nom_hote` sont uniques. La règle ou le trigger devrait implémenter une contrainte qui supprime les lignes de `logiciel` référençant un ordinateur supprimé. Le trigger utiliserait cette commande :

```
DELETE FROM logiciel WHERE nom_hote = $1;
```

Comme le trigger est appelé pour chaque ligne individuelle supprimée à partir de `ordinateur`, il peut préparer et sauvegarder le plan pour cette commande et passer la valeur `nom_hote` dans le paramètre. La règle devra être réécrite ainsi :

```
CREATE RULE ordinateur_del AS ON DELETE TO ordinateur
```

```
    DO DELETE FROM logiciel WHERE nom_hote = OLD.nom_hote;
```

Maintenant, nous apercevons différents types de suppressions. Dans le cas d'un :

```
DELETE FROM ordinateur WHERE nom_hote = 'mypc.local.net';
```

la table `ordinateur` est parcourue par l'index (rapide), et la commande lancée par le trigger pourrait aussi utiliser un parcours d'index (aussi rapide). La commande supplémentaire provenant de la règle serait :

```
DELETE FROM logiciel WHERE ordinateur.nom_hote = 'mypc.local.net'
                       AND logiciel.nom_hote = ordinateur.nom_hote;
```

Comme il y a une configuration appropriée des index, le planificateur créera un plan :

```
Nestloop
    ->  Index Scan using comp_hostidx on ordinateur
    ->  Index Scan using soft_hostidx on logiciel
```

Donc, il n'y aurait pas trop de différence de performance entre le trigger et l'implémentation de la règle.

Avec la prochaine suppression, nous voulons nous débarrasser des 2000 ordinateurs où `nom_hote` commence avec `old`. Il existe deux commandes possibles pour ce faire. Voici l'une d'elle :

```
DELETE FROM ordinateur WHERE nom_hote >= 'old'
                       AND nom_hote <  'ole'
```

La commande ajoutée par la règle sera :

```
DELETE FROM logiciel WHERE ordinateur.nom_hote >= 'old'
                       AND ordinateur.nom_hote < 'ole'
                       AND logiciel.nom_hote = ordinateur.nom_hote;
```

avec le plan :

```
Hash Join
    ->  Seq Scan on logiciel
    ->  Hash
    ->  Index Scan using comp_hostidx on ordinateur
```

L'autre commande possible est :

```
DELETE FROM ordinateur WHERE nom_hote ~ '^old';
```

ce qui finira dans le plan d'exécution suivant pour la commande ajoutée par la règle :

```
Nestloop
    ->  Index Scan using comp_hostidx on ordinateur
    ->  Index Scan using soft_hostidx on logiciel
```

Ceci monte que le planificateur ne réalise pas que la qualification pour `nom_hote` dans `ordinateur` pourrait aussi être utilisée pour un parcours d'index sur `logiciel` quand il existe

plusieurs expressions de qualifications combinées avec `and`, ce qui correspond à ce qu'il fait dans la version expression rationnelle de la commande. Le trigger sera appelé une fois pour chacun des 2000 anciens ordinateurs qui doivent être supprimées, et ceci résultera en un parcours d'index sur `ordinateur` et 2000 parcours d'index sur `logiciel`. L'implémentation de la règle le fera en deux commandes qui utilisent les index. Et cela dépend de la taille globale de la table `logiciel`, si la règle sera toujours aussi rapide dans la situation du parcours séquentiel. 2000 exécutions de commandes à partir du trigger sur le gestionnaire SPI prend un peu de temps, même si tous les blocs d'index seront rapidement dans le cache.

La dernière commande que nous regardons est :

```
DELETE FROM ordinateur WHERE constructeur = 'bim';
```

De nouveau, ceci pourrait résulter en de nombreuses lignes à supprimer dans `ordinateur`. Donc, le trigger lancera de nouveau de nombreuses commandes via l'exécuteur. La commande générée par la règle sera :

```
DELETE FROM logiciel WHERE ordinateur.constructeur = 'bim'
                       AND logiciel.nom_hote = ordinateur.nom_hote;
```

Le plan pour cette commande sera encore la boucle imbriquée sur les deux parcours d'index, en utilisant seulement un index différent sur `ordinateur` :

```
Nestloop
  ->  Index Scan using comp_manufidx on ordinateur
  ->  Index Scan using soft_hostidx on logiciel
```

Dans chacun de ces cas, les commandes supplémentaires provenant du système de règles seront plus ou moins indépendantes du nombre de lignes affectées en une commande.

Voici le résumé, les règles seront seulement significativement plus lentes que les triggers si leur actions résultent en des jointures larges et mal qualifiées, une situation où le planificateur échoue.

Chapitre 42. Langages de procédures

PostgreSQL permet l'écriture de fonctions et de procédures dans des langages différents du SQL et du C. Ces autres langages sont appelés génériquement des *langages de procédures* (LP, PL en anglais). Le serveur ne possède pas d'interpréteur interne des fonctions écrites dans un langage de procédures. La tâche est donc dévolue à un gestionnaire particulier qui, lui, connait les détails du langage. Le gestionnaire peut prendre en charge le travail de découpage, d'analyse syntaxique, d'exécution, etc., ou simplement servir de « colle » entre PostgreSQL et une implémentation existante d'un langage de programmation. Le gestionnaire est lui-même une fonction en langage C compilée dans une bibliothèque partagée et chargée à la demande, comme toute autre fonction C.

Il existe à ce jour quatre langages de procédures dans la distribution standard de PostgreSQL : PL/pgSQL (Chapitre 43), PL/Tcl (Chapitre 44), PL/Perl (Chapitre 45) et PL/Python (Chapitre 46).

Il existe d'autres langages de procédures qui ne sont pas inclus dans la distribution principale. L'Annexe H propose des pistes pour les trouver. De plus, d'autres langages peuvent être définis par les utilisateurs. Les bases de développement d'un nouveau langage de procédures sont couvertes dans le Chapitre 58.

42.1. Installation des langages de procédures

Un langage de procédures doit être « installé » dans toute base de données amenée à l'utiliser. Les langages de procédures installés dans la base de données `template1` sont automatiquement disponibles dans toutes les bases de données créées par la suite. `CREATE DATABASE` recopie en effet toutes les informations disponibles dans la base `template1`. Il est ainsi possible pour l'administrateur de définir, par base, les langages disponibles et d'en rendre certains disponibles par défaut.

Pour les langages fournis avec la distribution standard, l'installation dans la base courante se fait simplement par l'exécution de la commande `CREATE EXTENSION` *`langage`*. La procédure manuelle décrite ci-dessous n'est recommandée que pour installer des langages qui ne sont pas disponibles sous la forme d'extensions.

Installation manuelle de langages de procédures

Un langage de procédures s'installe en cinq étapes effectuées obligatoirement par le superutilisateur des bases de données. Dans la plupart des cas, les commandes SQL nécessaires doivent être placées dans un script d'installation d'une « extension », pour que la commande `CREATE EXTENSION` puisse être utilisé pour installer le langage.

1. La bibliothèque partagée du gestionnaire de langage doit être compilée et installée dans le répertoire de bibliothèques approprié. Cela se déroule comme la construction et l'installation de modules de classiques fonctions C utilisateur ; voir la Section 38.10.5. Il arrive souvent que le gestionnaire du langage dépende d'une bibliothèque externe fournissant le moteur de langage ; dans ce cas, elle doit aussi être installée.

2. Le gestionnaire doit être déclaré par la commande

   ```
   CREATE FUNCTION nom_fonction_gestionnaire()
       RETURNS gestionnaire_langage
       AS 'chemin-vers-objet-partagé'
       LANGUAGE C STRICT;
   ```

   Le type de retour spécial `gestionnaire_langage` indique au système que cette fonction ne renvoie pas un type de données SQL et n'est, de ce fait, pas utilisable directement dans des expressions SQL.

3. (Optional) En option, le gestionnaire de langages peut fournir une fonction de gestion « en ligne » qui permet l'exécution de blocs de code anonyme (commandes `DO`) écrits dans ce langage. Si une fonction de gestion en ligne est fourni par le langage, déclarez-le avec une commande comme

```
CREATE FUNCTION nom_fonction_en_ligne(internal)
    RETURNS void
    AS 'chemin-vers-objet-partagé'
    LANGUAGE C;
```

4. (Optional) En option, le gestionnaire de langages peut fournir une fonction de « validation » qui vérifie la définition d'une fonction sans réellement l'exécuter. La fonction de validation, si elle existe, est appelée par `CREATE FUNCTION`. Si une telle fonction est fournie par le langage, elle sera déclarée avec une commande de la forme

```
CREATE FUNCTION nom_fonction_validation(oid)
    RETURNS void
    AS 'chemin-vers-objet-partagé'
    LANGUAGE C;
```

5. Le LP doit être déclaré par la commande

```
CREATE [TRUSTED] LANGUAGE nom_langage
    HANDLER nom_fonction_gestionnaire
    [INLINE nom_fonction_en_ligne]
    [VALIDATOR nom_fonction_valideur] ;
```

   Le mot clé optionnel `TRUSTED` (autrement dit, digne de confiance) indique que le langage n'autorise pas l'accès à des données normalement inaccessible à cet utilisateur. Les langages de confiance sont conçus pour les utilisateurs standards de la base de données, c'est-à-dire ceux qui ne sont pas superutilisateurs, et les autorisent à créer en tout sécurité des fonctions et des procédures. Les fonctions en langage de procédures étant exécutées au sein du serveur, le paramètre `TRUSTED` ne devrait être positionné que pour les langages n'accédant pas aux organes internes du serveur ou au système de fichiers. Les langages PL/pgSQL, PL/Tcl, et PL/Perl sont considérés comme dignes de confiance ; les langages PL/TclU, PL/PerlU, et PL/PythonU sont conçus pour fournir des fonctionnalités illimitées et *ne* devraient *pas* être marqués dignes de confiance.

L'Exemple 42.1 présente le fonctionnement de la procédure d'installation manuelle du langage PL/Perl.

Exemple 42.1. Installation manuelle de PL/Perl

La commande suivante indique au serveur l'emplacement de la bibliothèque partagée pour la fonction de gestion des appels du langage PL/Perl.

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
    '$libdir/plperl' LANGUAGE C;
```

PL/Perl a une fonction de gestion en ligne et une fonction de validation, donc nous déclarons aussi celles-ci :

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
    '$libdir/plperl' LANGUAGE C STRICT;

CREATE FUNCTION plperl_validator(oid) RETURNS void AS
```

```
    '$libdir/plperl' LANGUAGE C STRICT;
```

La commande :

```
CREATE TRUSTED LANGUAGE plperl
    HANDLER plperl_call_handler
    INLINE plperl_inline_handler
    VALIDATOR plperl_validator;
```

indique l'évocation des fonctions précédentes pour les fonctions et procédures lorsque l'attribut de langage est `plperl`.

Lors de l'installation par défaut de PostgreSQL, le gestionnaire du langage PL/pgSQL est compilé et installé dans le répertoire des bibliothèques (« lib ») ; de plus, le langage PL/pgSQL est installé dans toutes les bases de données. Si le support de Tcl est configuré, les gestionnaires pour PL/Tcl et PL/TclU sont construits et installés dans le répertoire des bibliothèques mais le langage lui-même n'est pas installé par défaut dans les bases de données. De la même façon, les gestionnaires pour PL/Perl et PL/PerlU sont construits et installés si le support de Perl est configuré et le gestionnaire pour PL/PythonU est installé si le support de Python est configuré mais ces langages ne sont pas installés par défaut.

Chapitre 43. PL/pgSQL -- Langage de procédures SQL

43.1. Aperçu

PL/pgSQL est un langage de procédures chargeable pour le système de bases de données PostgreSQL. Les objectifs de la conception de PL/pgSQL ont été de créer un langage de procédures chargeable qui

- est utilisé pour créer des fonctions, des procédures et triggers,
- ajoute des structures de contrôle au langage SQL,
- permet d'effectuer des traitements complexes,
- hérite de tous les types, fonctions, procédures et opérateurs définis par les utilisateurs,
- est défini comme digne de confiance par le serveur,
- est facile à utiliser.

Les fonctions PL/pgSQL acceptent un nombre variable d'arguments en utilisant le marqueur `VARIADIC`. Cela fonctionne exactement de la même façon pour les fonctions SQL, comme indiqué dans Section 38.5.6.

Les fonctions écrites en PL/pgSQL peuvent être utilisées partout où une fonction intégrée peut l'être. Par exemple, il est possible de créer des fonctions complexes de traitement conditionnel et, par la suite, de les utiliser pour définir des opérateurs ou de les utiliser dans des expressions d'index.

À partir de la version 9.0 de PostgreSQL, PL/pgSQL est installé par défaut. Il reste toutefois un module chargeable et les administrateurs craignant pour la sécurité de leur instance pourront le retirer.

43.1.1. Avantages de l'utilisation de PL/pgSQL

SQL est le langage que PostgreSQL et la plupart des autres bases de données relationnelles utilisent comme langage de requête. Il est portable et facile à apprendre, mais chaque expression SQL doit être exécutée individuellement par le serveur de bases de données.

Cela signifie que votre application client doit envoyer chaque requête au serveur de bases de données, attendre que celui-ci la traite, recevoir et traiter les résultats, faire quelques calculs, et enfin envoyer d'autres requêtes au serveur. Tout ceci induit des communications interprocessus et induit aussi une surcharge du réseau si votre client est sur une machine différente du serveur de bases de données.

Grâce à PL/pgSQL vous pouvez grouper un bloc de traitement et une série de requêtes *au sein* du serveur de bases de données, et bénéficier ainsi de la puissance d'un langage de procédures, mais avec de gros gains en termes de communication client/serveur.

- Les allers/retours entre le client et le serveur sont éliminés
- Il n'est pas nécessaire de traiter ou transférer entre le client et le serveur les résultats intermédiaires dont le client n'a pas besoin
- Les va-et-vient des analyses de requêtes peuvent être évités

Ceci a pour conséquence une augmentation considérable des performances en comparaison à une application qui n'utilise pas les procédures stockées.

Ainsi, avec PL/pgSQL vous pouvez utiliser tous les types de données, opérateurs et fonctions du SQL.

43.1.2. Arguments supportés et types de données résultats

Les fonctions écrites en PL/pgSQL peuvent accepter en argument n'importe quel type de données supporté par le serveur, et peuvent renvoyer un résultat de n'importe lequel de ces types. Elles peuvent aussi accepter ou renvoyer n'importe quel type composite (type ligne) spécifié par nom. Il est aussi possible de déclarer une fonction PL/pgSQL renvoyant un type `record`, ce qui signifie que n'importe quel type composé conviendra, ou comme renvoyant `record` signifiant que le résultat est un type ligne dont les colonnes sont déterminées par spécification dans la requête appelante (voir la Section 7.2.1.4).

Les fonctions PL/pgSQL acceptent en entrée et en sortie les types polymorphes décrit dans Section 38.2.5, permettant ainsi que les types de données réels gérés par la fonction varient d'appel en appel. Des exemples sont disponibles sur Section 43.3.1.

Les fonctions PL/pgSQL peuvent aussi renvoyer un ensemble de lignes (ou une table) de n'importe lequel des types de données dont les fonctions peuvent renvoyer une instance unique. Ces fonctions génèrent leur sortie en exécutant `RETURN NEXT` pour chaque élément désiré de l'ensemble résultat ou en utilisant `RETURN QUERY` pour afficher le résultat de l'évaluation d'une requête.

Enfin, une fonction PL/pgSQL peut être déclarée comme renvoyant `void` si elle n'a pas de valeur de retour utile. (Il est possible de l'écrire comme une procédure dans ce cas.)

Les fonctions PL/pgSQL peuvent aussi être déclarées avec des paramètres en sortie à la place de la spécification explicite du code de retour. Ceci n'ajoute pas de fonctionnalité fondamentale au langage mais c'est un moyen agréable principalement pour renvoyer plusieurs valeurs. La notation `RETURNS TABLE` peut aussi être utilisé à la place de `RETURNS SETOF`.

Des exemples spécifiques apparaissent dans la Section 43.3.1 et la Section 43.6.1.

43.2. Structure de PL/pgSQL

Les fonctions écrites en PL/pgSQL sont définies auprès du serveur en exécutant les commandes CREATE FUNCTION. Une telle commande pourrait ressembler à ceci :

```
CREATE FUNCTION une_fonction(integer, text) RETURNS integer
AS 'texte du corps de la fonction'
LANGUAGE plpgsql;
```

Le corps de la fonction est une simple chaîne littérale pour ce qui concerne `CREATE FUNCTION`. Il est souvent utile d'utiliser les guillemets dollars (voir Section 4.1.2.4) pour écrire le corps de la fonction, plutôt que la syntaxe normale à base de guillemets simples. Sans les guillemets dollar, tout guillemet simple et antislashs dans le corps de la fonction doit être échappé en les doublant. Pratiquement tous les exemples de ce chapitre utilisent les littéraux en guillemets dollars dans les corps des fonctions.

PL/pgSQL est un langage structuré en blocs. Le texte complet du corps d'une fonction doit être un *bloc*. Un bloc est défini comme :

```
[ <<label>> ]
[ DECLARE
    déclarations ]
BEGIN
    instructions
```

```
END [ label ];
```

Chaque déclaration et chaque expression au sein du bloc est terminé par un point-virgule. Un bloc qui apparaît à l'intérieur d'un autre bloc doit avoir un point-virgule après `END` (voir l'exemple ci-dessus) ; néanmoins, le `END` final qui conclut le corps d'une fonction n'a pas besoin de point-virgule.

> **Astuce**
>
> Une erreur habituelle est d'écrire un point-virgule immédiatement après `BEGIN`. C'est incorrect et a comme résultat une erreur de syntaxe.

Un *`label`* est seulement nécessaire si vous voulez identifier le bloc à utiliser dans une instruction `EXIT` ou pour qualifier les noms de variables déclarées dans le bloc. Si un label est écrit après `END`, il doit correspondre au label donné au début du bloc.

Tous les mots clés sont insensibles à la casse. Les identifiants sont convertis implicitement en minuscule sauf dans le cas de l'utilisation de guillemets doubles. Le comportement est donc identique à celui des commandes SQL habituelles.

Les commentaires fonctionnent de la même manière tant dans du PL/pgSQL que dans le code SQL. Un double tiret (`--`) commence un commentaire et celui-ci continue jusqu'à la fin de la ligne. Un `/*` commence un bloc de commentaire qui continue jusqu'au `*/` correspondant. Les blocs de commentaires peuvent imbriquer les uns dans les autres.

Chaque expression de la section expression d'un bloc peut être un *sous-bloc*. Les sous-blocs peuvent être utilisés pour des groupements logiques ou pour situer des variables locales dans un petit groupe d'instructions. Les variables déclarées dans un sous-bloc masquent toute variable nommée de façon similaire dans les blocs externes pendant toute la durée du sous-bloc. Cependant, vous pouvez accéder aux variables externes si vous qualifiez leur nom du label de leur bloc. Par exemple :

```
CREATE FUNCTION une_fonction() RETURNS integer AS $$
<< blocexterne >>
DECLARE
    quantite integer := 30;
BEGIN
    RAISE NOTICE 'quantité vaut ici %', quantite;  -- affiche 30
    quantite := 50;
    --
    -- Crée un sous-bloc
    --
    DECLARE
        quantite integer := 80;
    BEGIN
        RAISE NOTICE 'quantite vaut ici %', quantite;  -- affiche
 80
 RAISE NOTICE 'la quantité externe vaut ici %',
 blocexterne.quantite;  -- affiche 50
    END;

    RAISE NOTICE 'quantité vaut ici %', quantite;  -- affiche 50

    RETURN quantite;
END;
$$ LANGUAGE plpgsql;
```

> **Note**
>
> Il existe un bloc externe caché entourant le corps de toute fonction PL/pgSQL. Ce bloc fournit la déclaration des paramètres de la fonction ainsi que quelques variables spéciales comme `FOUND` (voir la Section 43.5.5). Le bloc externe a pour label le nom de la fonction. Cela a pour conséquence que les paramètres et les variables spéciales peuvent être qualifiés du nom de la fonction.

Il est important de ne pas confondre l'utilisation de `BEGIN`/`END` pour grouper les instructions dans PL/pgSQL avec les commandes pour le contrôle des transactions. Les `BEGIN`/`END` de PL/pgSQL ne servent qu'au groupement ; ils ne débutent ni ne terminent une transaction. Voir Section 43.8 pour plus d'informations sur la gestion des transactions dans PL/pgSQL. De plus, un bloc contenant une clause `EXCEPTION` forme réellement une sous-transaction qui peut être annulée sans affecter la transaction externe. Pour plus d'informations sur ce point, voir la Section 43.6.8.

43.3. Déclarations

Toutes les variables utilisées dans un bloc doivent être déclarées dans la section déclaration du bloc. Les seules exceptions sont que la variable de boucle d'une boucle `FOR` effectuant une itération sur des valeurs entières est automatiquement déclarée comme variable entière (type integer), et de la même façon une variable de boucle `FOR` effectuant une itération sur le résultat d'un curseur est automatiquement déclarée comme variable de type record.

Les variables PL/pgSQL peuvent être de n'importe quel type de données tels que `integer`, `varchar` et `char`.

Quelques exemples de déclaration de variables :

```
id_utilisateur integer;
quantité numeric(5);
url varchar;
ma_ligne nom_table%ROWTYPE;
mon_champ nom_table.nom_colonne%TYPE;
une_ligne RECORD;
```

La syntaxe générale d'une déclaration de variable est :

```
nom [ CONSTANT ] type [ COLLATE nom_collationnement ] [ NOT NULL ]
 [ { DEFAULT | := | = } expression ];
```

La clause `DEFAULT`, si indiquée, spécifie la valeur initiale affectée à la variable quand on entre dans le bloc. Si la clause `DEFAULT` n'est pas indiquée, la variable est initialisée à la valeur SQL NULL. L'option `CONSTANT` empêche la modification de la variable après initialisation, de sorte que sa valeur reste constante pour la durée du bloc. L'option `COLLATE` indique le collationnement à utiliser pour la variable (voir Section 43.3.6). Si `NOT NULL` est spécifié, l'affectation d'une valeur NULL aboutira à une erreur d'exécution. Les valeurs par défaut de toutes les variables déclarées `NOT NULL` doivent être précisées, donc non NULL. Le signe d'égalité (`=`) peut être utilisé à la place de `:=`, qui lui est conforme au PL/SQL.

La valeur par défaut d'une variable est évaluée et affectée à la variable à chaque entrée du bloc (pas seulement une fois lors de l'appel de la fonction). Ainsi, par exemple, l'affectation de `now()` à une variable de type `timestamp` donnera à la variable l'heure de l'appel de la fonction courante, et non l'heure au moment où la fonction a été précompilée.

Exemples :

```
quantité integer DEFAULT 32;
url varchar := 'http://mysite.com';
transaction_time CONSTANT timestamp with time zone := now();
```

Une fois déclarée, la valeur d'une valeur peut être utilisée dans les expressions d'initialisation qui suivent, dans le même bloc. Par exemple :

```
DECLARE
  x integer := 1;
  y integer := x + 1;
```

43.3.1. Déclarer des paramètres de fonctions

Les paramètres passés aux fonctions sont nommés par les identifiants `$1`, `$2`, etc. Éventuellement, des alias peuvent être déclarés pour les noms de paramètres de type `$`*n* afin d'améliorer la lisibilité. L'alias ou l'identifiant numérique peuvent être utilisés indifféremment pour se référer à la valeur du paramètre.

Il existe deux façons de créer un alias. La façon préférée est de donner un nom au paramètre dans la commande `CREATE FUNCTION`, par exemple :

```
CREATE FUNCTION taxe_ventes(sous_total real) RETURNS real AS $$
BEGIN
    RETURN sous_total * 0.06;
END;
$$ LANGUAGE plpgsql;
```

L'autre façon est de déclarer explicitement un alias en utilisant la syntaxe de déclaration :

```
nom ALIAS FOR $n;
```

Le même exemple dans ce style ressemble à ceci :

```
CREATE FUNCTION taxe_ventes(real) RETURNS real AS $$
DECLARE
    sous_total ALIAS FOR $1;
BEGIN
    RETURN sous_total * 0.06;
END;
$$ LANGUAGE plpgsql;
```

> **Note**
>
> Ces deux exemples ne sont pas complètement identiques. Dans le premier cas, `sous_total` peut être référencé comme `taxe_ventes.sous_total`, alors que ce n'est pas possible dans le second cas. (Si nous avions attaché un label au bloc interne, `sous_total` aurait pu utiliser ce label à la place.)

Quelques exemples de plus :

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
```

```
    -- quelques traitements utilisant ici v_string et index
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_champs_selectionnes(in_t un_nom_de_table)
 RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

Quand une fonction PL/pgSQL est déclarée avec des paramètres en sortie, ces derniers se voient attribués les noms $*n* et des alias optionnels de la même façon que les paramètres en entrée. Un paramètre en sortie est une variable qui commence avec la valeur NULL ; il devrait se voir attribuer une valeur lors de l'exécution de la fonction. La valeur finale du paramètre est ce qui est renvoyée. Par exemple, l'exemple taxe_ventes peut s'écrire de cette façon :

```
CREATE FUNCTION taxe_ventes(sous_total real, OUT taxe real) AS $$
BEGIN
    taxe := sous_total * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Notez que nous avons omis `RETURNS real`. Nous aurions pu l'inclure mais cela aurait été redondant.

Pour appeler une fonction avec des paramètres `OUT`, omettez les paramètres en sortie dans l'appel de la fonction :

```
SELECT sales_tax(100.00);
```

Les paramètres en sortie sont encore plus utiles lors du retour de plusieurs valeurs. Un exemple trivial est :

```
CREATE FUNCTION somme_n_produits(x int, y int, OUT somme int, OUT
 produit int) AS $$
BEGIN
    somme := x + y;
    produit := x * y;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM somme_n_produits(2, 4);
 somme | produit
-------+---------
     6 |       8
```

D'après ce qui a été vu dans la Section 38.5.4, ceci crée réellement un type d'enregistrement anonyme pour les résultats de la fonction. Si une clause `RETURNS` est donnée, elle doit spécifier `RETURNS record`.

Ceci fonctionne aussi avec les procédures, par exemple :

```
CREATE PROCEDURE sum_n_product(x int, y int, OUT sum int, OUT prod
 int) AS $$
BEGIN
```

```
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

Dans un appel à une procédure, tous les paramètres doivent être spécifiés. Pour les paramètres en sortie, `NULL` pourrait être indiqué lors de l'appel de la procédure en SQL :

```
CALL sum_n_product(2, 4, NULL, NULL);

 sum | prod
-----+------
   6 |    8
```

Néanmoins, lors de l'appel d'une procédure à partir de PL/pgSQL, vous devez écrire à la place une variable pour chaque paramètre en sortie ; la variable recevra le résultat de l'appel. Voir Section 43.6.3 pour les détails.

Voici une autre façon de déclarer une fonction PL/pgSQL, cette fois avec `RETURNS TABLE` :

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales
 AS s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

C'est exactement équivalent à déclarer un ou plusieurs paramètres `OUT` et à spécifier `RETURNS SETOF` *`un_type`*.

Lorsque le type de retour d'une fonction PL/pgSQL est déclaré comme type polymorphe (see Section 38.2.5), un paramètre spécial `$0` est créé. Son type de donnée est le type effectif de retour de la fonction, déduit d'après les types en entrée (voir la Section 38.2.5). Ceci permet à la fonction d'accéder à son type de retour réel comme on le voit ici avec la Section 43.3.3. `$0` est initialisé à NULL et peut être modifié par la fonction, de sorte qu'il peut être utilisé pour contenir la variable de retour si besoin est, bien que cela ne soit pas requis. On peut aussi donner un alias à `$0`. Par exemple, cette fonction s'exécute comme un opérateur `+` pour n'importe quel type de données :

```
CREATE FUNCTION ajoute_trois_valeurs(v1 anyelement, v2 anyelement,
 v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    resultat ALIAS FOR $0;
BEGIN
    resultat := v1 + v2 + v3;
    RETURN resultat;
END;
$$ LANGUAGE plpgsql;
```

Le même effet peut être obtenu en déclarant un ou plusieurs paramètres polymorphes en sortie de types. Dans ce cas, le paramètre spécial `$0` n'est pas utilisé ; les paramètres en sortie servent ce même but. Par exemple :

```
CREATE FUNCTION ajoute_trois_valeurs(v1 anyelement, v2 anyelement,
 v3 anyelement,
```

```
                                    OUT somme anyelement)
AS $$
BEGIN
    somme := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

En pratique, il pourrait être plus utile de déclarer une fonction polymorphique en utilisant la famille de types `anycompatible`, pour que survienne la promotion automatique des arguments en entrée vers un type commune. Par exemple :

```
CREATE FUNCTION ajoute_trois_valeurs(v1 anycompatible, v2
 anycompatible, v3 anycompatible)
RETURNS anycompatible AS $$
BEGIN
    RETURN v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

Avec cet exemple, un appel tel que

```
SELECT ajoute_trois_valeurs(1, 2, 4.7);
```

fonctionnera, promouvant automatiquement les arguments entiers en numériques. La fonction utilisant `anyelement` nécessiterait que vous convertissiez manuellement les trois arguments sur le même type de données.

43.3.2. ALIAS

nouveaunom ALIAS FOR *anciennom*;

La syntaxe `ALIAS` est plus générale que la section précédente pourrait faire croire : vous pouvez déclarer un alias pour n'importe quelle variable et pas seulement des paramètres de fonction. L'utilisation principale de cette instruction est l'attribution d'un autre nom aux variables aux noms prédéterminés, telles que `NEW` ou `OLD` au sein d'une fonction trigger.

Exemples:

```
DECLARE
  anterieur ALIAS FOR old;
  misajour ALIAS FOR new;
```

`ALIAS` créant deux manières différentes de nommer le même objet, son utilisation à outrance peut prêter à confusion. Il vaut mieux ne l'utiliser uniquement pour se passer des noms prédéterminés.

43.3.3. Copie de types

variable%TYPE

`%TYPE` fournit le type de données d'une variable ou d'une colonne de table. Vous pouvez l'utiliser pour déclarer des variables qui contiendront des valeurs de base de données. Par exemple, disons que vous avez une colonne nommée `id_utilisateur` dans votre table `utilisateurs`. Pour déclarer une variable du même type de données que `utilisateurs.id_utilisateur`, vous pouvez écrire :

```
id_utilisateur utilisateurs.id_utilisateur%TYPE;
```

En utilisant `%TYPE` vous n'avez pas besoin de connaître le type de données de la structure à laquelle vous faites référence et, plus important, si le type de données de l'objet référencé change dans le futur (par exemple : vous changez le type de `id_utilisateur` de `integer` à `real`), vous pouvez ne pas avoir besoin de changer votre définition de fonction.

`%TYPE` est particulièrement utile dans le cas de fonctions polymorphes puisque les types de données nécessaires aux variables internes peuvent changer d'un appel à l'autre. Des variables appropriées peuvent être créées en appliquant `%TYPE` aux arguments de la fonction ou à la variable fictive de résultat.

43.3.4. Types ligne

```
nom nom_table%ROWTYPE;
nom nom_type_composite;
```

Une variable de type composite est appelée variable *ligne* (ou variable *row-type*). Une telle variable peut contenir une ligne entière de résultat de requête `SELECT` ou `FOR`, du moment que l'ensemble de colonnes de la requête correspond au type déclaré de la variable. Les champs individuels de la valeur row sont accessibles en utilisant la notation pointée, par exemple `varligne.champ`.

Une variable ligne peut être déclarée de façon à avoir le même type que les lignes d'une table ou d'une vue existante, en utilisant la notation *`nom_table`*`%ROWTYPE`. Elle peut aussi être déclarée en donnant un nom de type composite. Chaque table ayant un type de données associé du même nom, il importe peu dans PostgreSQL que vous écriviez `%ROWTYPE` ou pas. Cependant, la forme utilisant `%ROWTYPE` est plus portable.

Les paramètres d'une fonction peuvent être des types composites (lignes complètes de tables). Dans ce cas, l'identifiant correspondant $*n* sera une variable ligne à partir de laquelle les champs peuvent être sélectionnés avec la notation pointée, par exemple `$1.id_utilisateur`.

Voici un exemple d'utilisation des types composites. `table1` et `table2` sont des tables ayant au moins les champs mentionnés :

```
CREATE FUNCTION assemble_champs(t_ligne table1) RETURNS text AS $$
DECLARE
    t2_ligne table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_ligne FROM table2 WHERE ... ;
    RETURN t_ligne.f1 || t2_ligne.f3 || t_ligne.f5 || t2_ligne.f7;
END;
$$ LANGUAGE plpgsql;

SELECT assemble_champs(t.*) FROM table1 t WHERE ... ;
```

43.3.5. Types record

```
nom RECORD;
```

Les variables record sont similaires aux variables de type ligne mais n'ont pas de structure prédéfinie. Elles empruntent la structure effective de type ligne de la ligne à laquelle elles sont affectées durant une commande `SELECT` ou `FOR`. La sous-structure d'une variable record peut changer à chaque fois qu'on l'affecte. Une conséquence de cela est qu'elle n'a pas de sous-structure jusqu'à ce qu'elle ait été affectée, et toutes les tentatives pour accéder à un de ses champs entraînent une erreur d'exécution.

Notez que `RECORD` n'est pas un vrai type de données mais seulement un paramètre fictif (placeholder). Il faut aussi réaliser que lorsqu'une fonction PL/pgSQL est déclarée renvoyer un type `record`, il ne s'agit pas tout à fait du même concept qu'une variable record, même si une telle fonction peut aussi utiliser une variable record pour contenir son résultat. Dans les deux cas, la structure réelle de la ligne n'est pas connue quand la fonction est écrite mais, dans le cas d'une fonction renvoyant un type `record`, la structure réelle est déterminée quand la requête appelante est analysée, alors qu'une variable record peut changer sa structure de ligne à la volée.

43.3.6. Collationnement des variables PL/pgSQL

Quand une fonction PL/pgSQL a un ou plusieurs paramètres dont le type de données est collationnable, un collationnement est identifié pour chaque appel de fonction dépendant des collationnements affectés aux arguments réels, comme décrit dans Section 24.2. Si un collationnement est identifié avec succès (autrement dit, qu'il n'y a pas de conflit de collationnements implicites parmi les arguments), alors tous les paramètres collationnables sont traités comme ayant un collationnement implicite. Ceci affectera le comportement des opérations sensibles au collationnement dans la fonction. Par exemple, avec cette fonction

```
CREATE FUNCTION plus_petit_que(a text, b text) RETURNS boolean AS $
$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT plus_petit_que(champ_text_1, champ_text_2) FROM table1;
SELECT plus_petit_que(champ_text_1, champ_text_2 COLLATE "C") FROM
 table1;
```

La première utilisation de `less_than` utilisera le collationnement par défaut de `champ_text_1` et de `champ_text_2` pour la comparaison alors que la seconde utilisation prendra le collationnement `C`.

De plus, le collationnement identifié est aussi considéré comme le collationnement de toute variable locale de type collationnable. Du coup, cette procédure stockée ne fonctionnera pas différemment de celle-ci :

```
CREATE FUNCTION plus_petit_que(a text, b text) RETURNS boolean AS $
$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
END;
$$ LANGUAGE plpgsql;
```

S'il n'y a pas de paramètres pour les types de données collationnables ou qu'aucun collationnement commun ne peut être identifié pour eux, alors les paramètres et les variables locales utilisent le

collationnement par défaut de leur type de données (qui est habituellement le collationnement par défaut de la base de données mais qui pourrait être différent pour les variables des types domaines).

Une variable locale d'un type de données collationnable peut avoir un collationnement différent qui lui est associé en incluant l'option COLLATE dans sa déclaration, par exemple

```
DECLARE
    local_a text COLLATE "en_US";
```

Cette option surcharge le collationnement qui serait normalement donné à la variable d'après les règles ci-dessus.

De plus, les clauses COLLATE explicites peuvent être écrites à l'intérieur d'une fonction si forcer l'utilisation d'un collationnement particulier est souhaité pour une opération particulière. Par exemple,

```
CREATE FUNCTION plus_petit_que_c(a text, b text) RETURNS boolean AS
 $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

Ceci surcharge les collationnements associés avec les colonnes de la table, les paramètres ou la variable locales utilisées dans l'expression, comme cela arriverait dans une commande SQL simple.

43.4. Expressions

Toutes les expressions utilisées dans les instructions PL/pgSQL sont traitées par l'exécuteur SQL classique du serveur. En effet, une requête comme

```
SELECT expression
```

est traité par le moteur SQL principal. Bien qu'utilisant la commande SELECT, tout nom de variable PL/pgSQL est remplacé par les paramètres de la requête (ceci est expliqué en détail dans la Section 43.11.1). Cela permet au plan de requête du SELECT d'être préparé une seule fois, puis d'être réutilisé pour les évaluations suivantes avec différentes valeurs des variables. Du coup, ce qui arrive réellement à la première utilisation d'une expression est simplement une commande PREPARE. Par exemple, si nous déclarons deux variables de type integer, x et y, et que nous écrivons :

```
IF x < y THEN ...
```

ce qui se passe en arrière plan est équivalent à :

```
PREPARE nom_instruction(integer, integer) AS SELECT $1 < $2;
```

puis cette instruction préparée est exécutée (via EXECUTE) pour chaque exécution de l'instruction IF, avec les valeurs actuelles des variables PL/pgSQL fournies en tant que valeurs des paramètres. Généralement, ces détails ne sont pas importants pour un utilisateur de PL/pgSQL, mais ils sont utiles à connaître pour diagnostiquer un problème. Vous trouverez plus d'informations dans Section 43.11.2.

Comme une *expression* est convertie en une commande SELECT, elle peut contenir les mêmes clauses qu'un SELECT ordinaire, sauf qu'elle ne peut pas inclure une clause UNION, INTERSECT ou EXCEPT dans la requête principale. Donc, par exemple, vous pourriez tester si une table est remplie avec

```
IF count(*) > 0 FROM my_table THEN ...
```

parce que l'*expression* entre IF et THEN est analysé comme si c'était SELECT count(*) > 0 FROM my_table. Le SELECT doit produire une seule colonne, et pas plus d'une.(S'il ne produit aucune ligne, le résultat est considéré comme NULL.)

43.5. Instructions de base

Dans cette section ainsi que les suivantes, nous décrirons tous les types d'instructions explicitement compris par PL/pgSQL. Tout ce qui n'est pas reconnu comme l'un de ces types d'instruction est présumé être une commande SQL et est envoyé au moteur principal de bases de données pour être exécutée comme décrit dans la Section 43.5.2.

43.5.1. Affectation

L'affectation d'une valeur à une variable PL/pgSQL s'écrit ainsi :

```
variable { := | = } expression;
```

Comme expliqué précédemment, l'expression dans cette instruction est évaluée au moyen de la commande SQL SELECT envoyée au moteur principal de bases de données. L'expression ne doit manier qu'une seule valeur (éventuellement une valeur de rangée, si cette variable est une variable de rangée ou d'enregistrement). La variable cible peut être une simple variable (éventuellement qualifiée avec un nom de bloc), un champ d'une rangée ou variable d'enregistrement ou un élément ou partie de tableau cible. Le signe d'égalité (=) peut être utilisé à la place de :=, qui lui est conforme au PL/SQL.

Si le type de données du résultat de l'expression ne correspond pas au type de donnée de la variable, la valeur sera convertie via une conversion d'affectation (cf Section 10.4. Si aucune conversion d'affectation n'est connue pour les deux types de données concernées, l'interpréteur PL/pgSQL tentera de convertir le résultat textuellement, c'est-à-dire en appliquant successivement la fonction de sortie du type résultat puis la fonction d'entrée du type de la variable. Notez que la fonction d'entrée peut générer des erreurs à l'exécution si la chaîne passée en paramètre n'est pas acceptable pour le type de la variable.

Exemples :

```
taxe := sous_total * 0.06;
mon_enregistrement.id_utilisateur := 20;
my_array[j] := 20;
my_array[1:3] := array[1,2,3];
complex_array[n].realpart = 12.3;
```

43.5.2. Exécuter des commandes SQL

En général, toute commande SQL qui ne renvoie pas de lignes peut être exécutée à l'intérieur d'une fonction PL/pgSQL en écrivant simplement la commande. Par exemple, vous pouvez créer et remplir une table en écrivant :

```
CREATE TABLE mytable (id int primary key, data text);
INSERT INTO mytable VALUES (1,'one'), (2,'two');
```

Si la commande renvoie des lignes (par exemple SELECT, ou INSERT/UPDATE/DELETE avec RETURNING), il existe deux façons de procéder. Quand la commande renverra au plus une ligne ou quand vous faites seulement attention à la première ligne en sortie, écrivez la commande comme d'habitude mais ajoutez une clause INTO pour capturer la sortie, comme décrit dans Section 43.5.3. Pour traiter toutes les lignes en sortie, écrivez la commande comme la source de données pour une boucle FOR, comme décrit dans Section 43.6.6.

Habituellement, il n'est pas suffisant de simplement exécuter des commandes SQL statiques. Typiquement, vous voudrez une commande à utiliser des valeurs de données variables ou même pour varier de façon plus fondamentale en utilisant par exemple des noms de tables différents à différents moments. Encore une fois, ilexiste deux façons de procéder suivant la situation.

Les valeurs des variables PL/pgSQL peuvent être insérées automatiquement dans des commandes SQL optimisables, comme SELECT, INSERT, UPDATE, DELETE, MERGE et certaines commandes utilitaires qui les incorporent, telles que EXPLAIN et CREATE TABLE ... AS SELECT. Dans ces commandes, tout nom de variable PL/pgSQL apparaissant dans le texte de la commande est remplacé par un paramètre de la requête, puis la valeur actuelle de la variable est fournie comme valeur du paramètre à l'exécution. C'est le traitement exact décrit précédemment pour les expressions. Pour les détails, voir la Section 43.11.1.

Lors de l'exécution d'une commande SQL optimisable de cette façon, PL/pgSQL peut placer le plan en cache et le réutiliser plus tard, comme indiqué dans Section 43.11.2.

Les commandes SQL non optimisables (aussi appelées commandes utilitaires) ne sont pas capables d'accepter des paramètres aux requêtes. Donc la substitution automatique des variables PL/pgSQL ne fonctionne pas sur ces commandes. Pour inclure un texte non constant dans une commande utilitaire exécutée à partir de PL/pgSQL, vous devez construire la commande utilitaire avec une chaîne de caractères, puis l'exécuter avec EXECUTE, comme indiqué dans discussed in Section 43.5.4.

EXECUTE doit aussi être utilisé si vous voulez modifier la commande de toute autre façon que de fournir une valeur de donnée, par exemple en changeant le nom d'une table.

Parfois, il est utile d'évaluer une expression ou une requête SELECT mais sans récupérer le résultat, par exemple lors de l'appel d'une fonction qui a des effets de bord mais dont la valeur du résultat n'est pas utile. Pour faire cela en PL/pgSQL, utilisez l'instruction PERFORM :

```
PERFORM requête;
```

Ceci exécute la *requête* et ne tient pas compte du résultat. Écrivez la *requête* de la même façon que vous écririez une commande SELECT mais remplacez le mot clé initial SELECT avec PERFORM. Pour les requêtes WITH, utilisez PERFORM puis placez la requête entre parenthèses. (De cette façon, la requête peut seulement renvoyer une ligne.) Les variables PL/pgSQL seront substituées dans la requête comme décrit ci-dessus. Le plan est mis en cache de la même façon. La variable spéciale FOUND est configurée à true si la requête a produit au moins une ligne, false dans le cas contraire (voir la Section 43.5.5).

> **Note**
>
> Vous pourriez vous attendre à ce que l'utilisation directe de SELECT aboutisse au même résultat mais, actuellement, la seule façon acceptée de le faire est d'utiliser PERFORM. Une commande SQL qui peut renvoyer des lignes comme SELECT sera rejetée comme une erreur si elle n'a pas de clause INTO, ce qui est discuté dans la section suivante.

Un exemple :

```
PERFORM creer_vuemat('cs_session_page_requests_mv', ma_requete);
```

43.5.3. Exécuter une commande avec une seule ligne de résultats

Le résultat d'une commande SQL ne retourne qu'une seule ligne (mais avec une ou plusieurs colonnes) peut être affecté à une variable de type record, row ou à une liste de variables scalaires. Ceci se fait en écrivant la commande SQL de base et en ajoutant une clause `INTO`. Par exemple,

```
SELECT expressions_select INTO [STRICT] cible FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] cible;
UPDATE ... RETURNING expressions INTO [STRICT] cible;
DELETE ... RETURNING expressions INTO [STRICT] cible;
```

où *`cible`* peut être une variable de type record, row ou une liste de variables ou de champs record/row séparées par des virgules. Les variables PL/pgSQL seront substituées dans le reste de la commande (c'est-à-dire, tout sauf la clause `INTO` décrite ci-dessus), et le plan est mis en cache de la même façon. Ceci fonctionne pour `SELECT`, `INSERT`/`UPDATE`/`DELETE` avec `RETURNING`, et certaines commandes utilitaires qui renvoient des résultats de type rowset(comme `EXPLAIN`). Sauf pour la clause `INTO`, la commande SQL est identique à celle qui aurait été écrite en dehors de PL/pgSQL.

> **Astuce**
>
> Notez que cette interprétation de `SELECT` avec `INTO` est assez différente de la commande habituelle `SELECT INTO` où la cible `INTO` est une table nouvellement créée. Si vous voulez créer une table à partir du résultat d'un `SELECT` à l'intérieur d'une fonction PL/pgSQL, utilisez la syntaxe `CREATE TABLE ... AS SELECT`.

Si une variable de ligne ou une liste de variables est utilisée comme cible, les colonnes du résultat de la commande doivent correspondre exactement à la structure de la cible (nombre de champs et types de données). Dans le cas contraire, une erreur sera rapportée à l'exécution. Quand une variable record est la cible, elle se configure automatiquement avec le type row des colonnes du résultat de la commande.

La clause `INTO` peut apparaître pratiquement partout dans la commande SQL. Elle est écrite soit juste avant soit juste après la liste d'*`expressions_select`* dans une commande `SELECT`, ou à la fin de la commande pour d'autres types de commande. Il est recommandé de suivre cette convention au cas où l'analyseur PL/pgSQL devient plus strict dans les versions futures.

Si `STRICT` n'est pas spécifié dans la clause `INTO`, alors *`cible`* sera configuré avec la première ligne renvoyée par la commande ou à NULL si la commande n'a renvoyé aucune ligne. (Notez que « la première ligne » n'est bien définie que si vous avez utilisé `ORDER BY`.) Toute ligne résultat après la première ligne est annulée. Vous pouvez vérifier la valeur de la variable spéciale `FOUND` (voir la Section 43.5.5) pour déterminer si une ligne a été renvoyée :

```
SELECT * INTO monrec FROM emp WHERE nom = mon_nom;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employé % introuvable', mon_nom;
END IF;
```

Si l'option `STRICT` est indiquée, la commande doit renvoyer exactement une ligne. Dans le cas contraire, une erreur sera rapportée à l'exécution, soit `NO_DATA_FOUND` (aucune ligne) soit `TOO_MANY_ROWS` (plus d'une ligne). Vous pouvez utiliser un bloc d'exception si vous souhaitez récupérer l'erreur, par exemple :

```
BEGIN
    SELECT * INTO STRICT monrec FROM emp WHERE nom = mon_nom;
    EXCEPTION
```

```
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employé % introuvable', mon_nom;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employé % non unique', mon_nom;
END;
```

Une exécution réussie de la commande avec `STRICT` renvoie toujours true pour `FOUND`.

Pour les commandes `INSERT` / `UPDATE` / `DELETE` utilisées avec la clause `RETURNING`, PL/pgSQL renvoie une erreur si plus d'une ligne est renvoyée, même si la clause `STRICT` n'est pas indiquée. Ceci est dû au fait qu'il n'existe pas d'option `ORDER BY` qui permettrait de déterminer la ligne affectée à renvoyer.

Si `print_strict_params` est activé pour cette fonction, alors, quand une erreur est renvoyée parce que les conditions de `STRICT` ne sont pas rencontrées, la partie `DETAIL` du message d'erreur incluera les informations sur les paramètres passés à la commande. Vous pouvez modifier la configuration de `print_strict_params` pour toutes les fonctions en configurant `plpgsql.print_strict_params`, bien que seules les compilations suivantes des fonctions seront affectées. Vous pouvez aussi l'activer fonction par fonction en utilisant une option du compilateur, par exemple :

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
#print_strict_params on
DECLARE
userid int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END;
$$ LANGUAGE plpgsql;
```

En cas d'échec, cette fonction pourrait renvoyer un message d'erreur tel que :

```
ERROR:  query returned no rows
DETAIL:  parameters: username = 'nosuchuser'
CONTEXT:  PL/pgSQL function get_userid(text) line 6 at SQL
 statement
```

> **Note**
>
> L'option `STRICT` correspond au comportement du `SELECT INTO` d'Oracle PL/SQL et des instructions relatives.

43.5.4. Exécuter des commandes dynamiques

Créer dynamique des requêtes SQL est un besoin habituel dans les fonctions PL/pgSQL, par exemple des requêtes qui impliquent différentes tables ou différents types de données à chaque fois qu'elles sont exécutées. Les tentatives normales de PL/pgSQL pour garder en cache les planifications des commandes (voir la Section 43.11.2) ne fonctionneront pas dans de tels scénarios. Pour gérer ce type de problème, l'instruction `EXECUTE` est proposée :

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression
 [, ...] ];
```

où *chaîne-commande* est une expression manipulant une chaîne (de type `text`) contenant la commande à exécuter. La *cible* optionnelle est une variable record ou ligne ou même une liste de variables simples ou de champs de lignes/enregistrements séparées par des virgules, dans lesquels les résultats de la commande seront enregistrés. Les expressions `USING` optionnelles fournissent des valeurs à insérer dans la commande.

Aucune substitution des variables PL/pgSQL ne se fait dans la chaîne de commande calculée. Toutes les valeurs des variables requises doivent être insérées dans la chaîne de commande au moment de sa construction ; ou vous pouvez utiliser des paramètres comme décrits ci-dessous.

De plus, il n'y a pas mise en cache des commandes exécutées via `EXECUTE`. À la place, la commande est planifiée à chaque fois que l'instruction est lancée. La chaîne commande peut être créée dynamiquement à l'intérieur de la fonction pour agir sur des tables ou colonnes différentes.

La clause `INTO` spécifie où devraient être affectés les résultats d'une commande SQL renvoyant des lignes. Si une variable de ligne ou une liste de variable est fournie, elle doit correspondre exactement à la structure des résultats de la commande. Si une variable de type record est utilisée, elle sera automatiquement typée pour correspondre à la structure du résultat. Si plusieurs lignes sont renvoyées, alors seule la première sera assignée à la (ou les) variables `INTO`. Si aucune ligne n'est renvoyée, NULL est affectée à la variable `INTO`. Si aucune clause `INTO` n'est spécifiée, les résultats de la commande sont ignorés.

Si l'option `STRICT` est indiquée, une erreur est rapportée sauf si la commande produit exactement une ligne.

La chaîne de commande peut utiliser des valeurs de paramètres, référencées dans la commande avec `$1`, `$2`, etc. Ces symboles font référence aux valeurs fournies dans la clause `USING`. Cette méthode est souvent préférable à l'insertion des valeurs en texte dans une chaîne de commande : cela évite la surcharge à l'exécution pour la conversion des valeurs en texte et vice-versa. C'est aussi moins sensible aux attaques par injection SQL car il n'est pas nécessaire de mettre entre guillemets ou d'échapper les valeurs. Voici un exemple :

```
EXECUTE 'SELECT count(*) FROM matable WHERE insere_par = $1 AND
 insere <= $2'
   INTO c
   USING utilisateur_verifie, date_verifiee;
```

Notez que les symboles de paramètres peuvent seulement être utilisés pour des valeurs de données -- si vous voulez utiliser des noms de tables et/ou colonnes déterminés dynamiquement, vous devez les insérer dans la chaîne de commande en texte. Par exemple, si la requête précédente devait se faire avec une table sélectionnée dynamiquement, vous devriez faire ceci :

```
EXECUTE 'SELECT count(*) FROM '
    || quote_ident(tabname)
    || ' WHERE insere_par = $1 AND insere <= $2'
   INTO c
   USING utilisateur_verifie, date_verifiee;
```

Une meilleure solution est d'utiliser la spécification de formatage `%I` de la fonction `format()` pour insérer les noms de table ou de colonne avec des guillemets automatiques :

```
EXECUTE format('SELECT count(*) FROM %I '
   'WHERE insere_par = $1 AND insere <= $2', matable)
   INTO c
   USING utilisateur_verifie, date_verifiee;
```

(Cet exemple se base sur la règle SQL qui dit que les constantes de chaînes séparées par un retour à la ligne sont implicitement concaténées.)

Une autre restriction sur les symboles de paramètres est qu'ils ne fonctionnent que dans les commandes SQL optimisables (`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE` et certaines commandes contenant l'une d'entre elles). Dans les autres types d'instructions(appelés de manière générique commandes utilitaires), vous devez insérer les valeurs sous forme de texte même si ce ne sont que des données.

Un `EXECUTE` avec une chaîne de commande constante et des paramètres `USING`, comme dans le premier exemple ci-dessus, est équivalent fonctionnellement à l'écriture simple d'une commande directement dans PL/pgSQL et permet le remplacement automatique des variables PL/pgSQL. La différence importante est que `EXECUTE` va planifier de nouveau la commande pour chaque exécution, générant un plan qui est spécifique aux valeurs actuelles des paramètres ; alors que PL/pgSQL pourrait sinon créer un plan générique et le stocke pour le réutiliser. Dans des situations où le meilleur plan dépend fortement des valeurs des paramètres, cela peut être utile d'utiliser `EXECUTE` pour s'assurer qu'un plan générique n'est pas sélectionné.

`SELECT INTO` n'est actuellement pas supporté à l'intérieur de `EXECUTE` ; à la place, exécutez une commande `SELECT` et spécifiez `INTO` comme faisant partie lui-même d'`EXECUTE`.

> **Note**
>
> L'instruction `EXECUTE` de PL/pgSQL n'a pas de relation avec l'instruction SQL `EXECUTE` supportée par le serveur PostgreSQL. L'instruction `EXECUTE` du serveur ne peut pas être utilisée directement dans les fonctions PL/pgSQL. En fait, elle n'est pas nécessaire.

Exemple 43.1. Mettre entre guillemets des valeurs dans des requêtes dynamiques

En travaillant avec des commandes dynamiques, vous aurez souvent à gérer des échappements de guillemets simples. La méthode recommandée pour mettre entre guillemets un texte fixe dans le corps de votre fonction est d'utiliser les guillemets dollar (si votre code n'utilise pas les guillemets dollar, référez-vous à l'aperçu dans la Section 43.12.1, ce qui peut vous faire gagner des efforts lors du passage de ce code à un schéma plus raisonnable).

Les valeurs dynamiques à insérer dans la requête construite requièrent une attention spéciale car elles pourraient elles-même contenir des guillemets. Voici un exemple utilisant la fonction `format()` (cet exemple suppose que vous utilisiez les guillemets dollar pour la fonction dans sa globalité pour que les guillemets n'aient pas besoin d'être doublés) :

```
EXECUTE format('UPDATE table SET %I = $1 '
   'WHERE clef = $2', nom_colonne) USING nouvelle_valeur,
 valeur_clef;
```

Il est également possible d'appeler explicitement les fonctions d'échappement:

```
EXECUTE 'UPDATE tbl SET '
    || quote_ident(nom_colonne)
    || ' = '
    || quote_literal(nouvelle_valeur)
```

```
    || ' WHERE cle = '
    || quote_literal(valeur_cle);
```

Cet exemple démontre l'utilisation des fonctions `quote_ident` et `quote_literal` (voir Section 9.4). Pour plus de sûreté, les expressions contenant les identifiants des colonnes et des tables doivent être passées à la fonction `quote_ident` avant l'insertion dans une requête dynamique. Les expressions contenant des valeurs de type chaîne de caractères doivent être passées à `quote_literal`. Ce sont les étapes appropriées pour renvoyer le texte en entrée entouré par des guillemets doubles ou simples respectivement, en échappant tout caractère spécial.

Comme `quote_literal` est labelisé `STRICT`, elle renverra toujours NULL lorsqu'elle est appelée avec un argument NULL. Dans l'exemple ci-dessus, si `nouvelle_valeur` ou `valeur_clé` étaient NULL, la requête dynamique entière deviendrait NULL, amenant une erreur à partir du `EXECUTE`. Vous pouvez éviter ce problème en utilisant la fonction `quote_nullable` qui fonctionne de façon identique à `quote_literal` sauf si elle est appelée avec un argument NULL, elle renvoie la chaîne `NULL`. Par exemple,

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(nom_colonne)
        || ' = '
        || quote_nullable(nouvelle_valeur)
        || ' WHERE key = '
        || quote_nullable(valeur_clé);
```

Si vous travaillez avez des valeurs qui peuvent être NULL, vous devez utiliser `quote_nullable` à la place de `quote_literal`.

Comme toujours, il faut s'assurer que les valeurs NULL d'une requête ne ramènent pas des valeurs inattendues. Par exemple, la clause `WHERE`

```
'WHERE key = ' || quote_nullable(valeur_clé)
```

ne sera jamais vrai si `valeur_clé` est NULL car le résultat de l'opérateur d'égalité, `=`, avec au moins un des opérandes NULL est toujours NULL. Si vous souhaitez que NULL fonctionne comme toute autre valeur de clé ordinaire, vous devez ré-écrire la clause ci-dessus de cette façon :

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

(Actuellement, `IS NOT DISTINCT FROM` est géré moins efficacement que `=`, donc ne l'utilisez pas sauf en cas d'extrême nécessité. Voir Section 9.2 pour plus d'informations sur les NULL et `IS DISTINCT`.)

Notez que les guillemets dollar sont souvent utiles pour placer un texte fixe entre guillemets. Ce serait une très mauvaise idée d'écrire l'exemple ci-dessus de cette façon :

```
EXECUTE 'UPDATE tbl SET '
|| quote_ident(nom_colonne)
|| ' = $$'
|| nouvelle_valeur
|| '$$ WHERE cle = '
|| quote_literal(valeur_cle);
```

car cela casserait si le contenu de `nouvelle_valeur` pouvait contenir `$$`. La même objection s'applique à tout délimiteur dollar que vous pourriez choisir. Donc, pour mettre un texte inconnu

entre guillemets de façon sûr, vous *devez* utiliser `quote_literal`, `quote_nullable` ou `quote_ident`, comme approprié.

Les requêtes SQL dynamiques peuvent aussi être construites en toute sécurité en utilisant la fonction `format` (voir Section 9.4.1). Par exemple :

```
EXECUTE format('UPDATE matable SET %I = %L '
   'WHERE clef = %L', nom_colonne, nouvelle_valeur, valeur_clef);
```

`%I` est équivalent à `quote_ident`, et `%L` est équivalent à `quote_nullable`. La fonction `format` peut être utilisée avec la clause `USING` :

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE cle = $2',
 nom_colonne)
   USING nouvellevaleur, clevaleur;
```

Cette forme est meilleure car les variables sont traitées dans le format natif à leur type plutôt que de les convertir inconditionnellement en texte et de les échapper via le spécifieur de format `%L`. C'est également plus performant.

Un exemple bien plus important d'une commande dynamique et d'`EXECUTE` est disponible dans l'Exemple 43.10, qui construit et exécute une commande `CREATE FUNCTION` pour définir une nouvelle fonction.

43.5.5. Obtention du statut du résultat

Il y a plusieurs moyens pour déterminer l'effet d'une commande. La première méthode est d'utiliser `GET DIAGNOSTICS` :

```
GET [ CURRENT ] DIAGNOSTICS variable { = | := } élément [ , ... ];
```

Cette commande récupère les indicateurs de statut du système. `CURRENT` est un mot optionnel (mais voir aussi `GET STACKED DIAGNOSTICS` dans Section 43.6.8.1). Chaque *`élément`* est un mot clé identifiant une valeur de statut à affecter à la *`variable`* indiquée (qui doit être du bon type de données pour la recevoir). Les éléments de statut actuellement disponibles sont affichés dans Tableau 43.1. L'opérateur deux-points- égal (`:=`) peut être utilisé à la place de l'opérateur `=` qui lui est compatible avec le standard SQL. Exemple :

```
GET DIAGNOSTICS var_entier = ROW_COUNT;
```

Tableau 43.1. Éléments de diagnostics disponibles

Nom	Type	Description
`ROW_COUNT`	`bigint`	le nombre de lignes traitées par la commande SQL la plus récente
`PG_CONTEXT`	`text`	ligne(s) de texte décrivant la pile d'appels actuelle (voir Section 43.6.9)

La seconde méthode permettant de déterminer les effets d'une commande est la variable spéciale nommée `FOUND` de type `boolean`. La variable `FOUND` est initialisée à false au début de chaque fonction PL/pgSQL. Elle est positionnée par chacun des types d'instructions suivants :

- Une instruction SELECT INTO positionne FOUND à true si une ligne est affectée, false si aucune ligne n'est renvoyée.
- Une instruction PERFORM positionne FOUND à true si elle renvoie une ou plusieurs lignes, false si aucune ligne n'est produite.
- Les instructions UPDATE, INSERT, DELETE et MERGE positionnent FOUND à true si au moins une ligne est affectée, false si aucune ligne n'est affectée.
- Une instruction FETCH positionne FOUND à true si elle renvoie une ligne, false si aucune ligne n'est renvoyée.
- Une instruction MOVE initialise FOUND à true si elle repositionne le curseur avec succès. Dans le cas contraire, elle le positionne à false.
- Une instruction FOR ou FOREACH initialise FOUND à la valeur true s'il itère une ou plusieurs fois, et à false dans les autres cas. FOUND est initialisé de cette façon quand la boucle se termine : pendant l'exécution de la boucle, FOUND n'est pas modifié par la boucle, bien qu'il pourrait être modifié par l'exécution d'autres requêtes dans le corps de la boucle.
- Les instructions RETURN QUERY et RETURN QUERY EXECUTE mettent à jour la variable FOUND à true si la requête renvoie au moins une ligne, et false si aucune ligne n'est renvoyée.

Les autres instructions PL/pgSQL ne changent pas l'état de FOUND. Notez que la commande EXECUTE modifie la sortie de GET DIAGNOSTICS mais ne change pas FOUND.

FOUND est une variable locale à l'intérieur de chaque fonction PL/pgSQL ; chaque changement qui y est fait n'affecte que la fonction en cours.

43.5.6. Ne rien faire du tout

Quelque fois, une instruction qui ne fait rien est utile. Par exemple, elle indique qu'une partie de la chaîne IF/THEN/ELSE est délibérément vide. Pour cela, utilisez l'instruction :

```
NULL;
```

Par exemple, les deux fragments de code suivants sont équivalents :

```
BEGIN
  y := x / 0;
  EXCEPTION
  WHEN division_by_zero THEN
    NULL;  -- ignore l'erreur
  END;

BEGIN
  y := x / 0;
  EXCEPTION
  WHEN division_by_zero THEN  -- ignore l'erreur
  END;
```

Ce qui est préférable est une question de goût.

> **Note**
>
> Dans le PL/SQL d'Oracle, les listes d'instructions vides ne sont pas autorisées et, du coup, les instructions NULL sont *requises* dans les situations telles que celles-ci. PL/pgSQL vous permet d'écrire simplement rien.

43.6. Structures de contrôle

Les structures de contrôle sont probablement la partie la plus utile (et importante) de PL/pgSQL. Grâce aux structures de contrôle de PL/pgSQL, vous pouvez manipuler les données PostgreSQL de façon très flexible et puissante.

43.6.1. Retour d'une fonction

Il y a deux commandes disponibles qui vous permettent de renvoyer des données d'une fonction : `RETURN` et `RETURN NEXT`.

43.6.1.1. RETURN

```
RETURN expression;
```

`RETURN` accompagné d'une expression termine la fonction et renvoie le valeur de l'*`expression`* à l'appelant. Cette forme doit être utilisée avec des fonctions PL/pgSQL qui ne renvoient pas d'ensemble de valeurs.

Dans une fonction qui renvoie un type scalaire, le résultat de l'expression sera automatiquement convertie dans le type que la fonction renvoie. Mais pour renvoyer une valeur composite (ligne), vous devez écrire une expression renvoyant exactement l'ensemble de colonnes souhaité. Ceci peut demander l'utilisation de conversion explicite.

Si vous déclarez la fonction avec des paramètres en sortie, écrivez seulement `RETURN` sans expression. Les valeurs courantes des paramètres en sortie seront renvoyées.

Si vous déclarez que la fonction renvoie `void`, une instruction `RETURN` peut être utilisée pour quitter rapidement la fonction ; mais n'écrivez pas d'expression après `RETURN`.

La valeur de retour d'une fonction ne peut pas être laissée indéfinie. Si le contrôle atteint la fin du bloc de haut niveau de la fonction, sans parvenir à une instruction `RETURN`, une erreur d'exécution survient. Néanmoins, cette restriction ne s'applique pas aux fonctions sans paramètre de sortie et aux fonctions renvoyant `void`. Dans ces cas, une instruction `RETURN` est automatiquement exécutée si le bloc de haut niveau est terminé.

Quelques exemples :

```
-- fonctions renvoyant un type scalaire
RETURN 1 + 2;
RETURN scalar_var;

-- fonctions renvoyant un type composite
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- must cast columns to correct
 types
```

43.6.1.2. RETURN NEXT et RETURN QUERY

```
RETURN NEXT expression;
RETURN QUERY requete;
RETURN QUERY EXECUTE command-string [ USING expression [, ...] ];
```

Quand une fonction PL/pgSQL déclare renvoyer SETOF *un_certain_type*, la procédure à suivre est un peu différente. Dans ce cas, les éléments individuels à renvoyer sont spécifiés par une séquence de commandes RETURN NEXT ou RETURN QUERY, suivies de la commande finale RETURN sans argument qui est utilisée pour indiquer la fin de l'exécution de la fonction. RETURN NEXT peut être utilisé avec des types de données scalaires comme composites ; avec un type de résultat composite, une « table » entière de résultats sera renvoyée. RETURN QUERY ajoute les résultats de l'exécution d'une requête à l'ensemble des résultats de la fonction. RETURN NEXT et RETURN QUERY peuvent être utilisés dans la même fonction, auquel cas leurs résultats seront concaténées.

RETURN NEXT et RETURN QUERY ne quittent pas réellement la fonction -- elles ajoutent simplement zéro ou plusieurs lignes à l'ensemble de résultats de la fonction. L'exécution continue ensuite avec l'instruction suivante de la fonction PL/pgSQL. Quand plusieurs commandes RETURN NEXT et/ou RETURN QUERY successives sont exécutées, l'ensemble de résultats augmente. Un RETURN, sans argument, permet de quitter la fonction mais vous pouvez aussi continuer jusqu'à la fin de la fonction.

RETURN QUERY dispose d'une variante RETURN QUERY EXECUTE, qui spécifie la requête à exécuter dynamiquement. Les expressions de paramètres peuvent être insérées dans la chaîne calculée via USING, de la même façon que le fait la commande EXECUTE.

Si vous déclarez la fonction avec des paramètres en sortie, écrivez RETURN NEXT sans expression. À chaque exécution, les valeurs actuelles des variables paramètres en sortie seront sauvegardées pour un renvoi éventuel en tant que résultat en sortie. Notez que vous devez déclarer la fonction en tant que SETOF record quand il y a plusieurs paramètres en sortie, ou SETOF *un_certain_type* quand il y a un seul paramètre en sortie, et de type *un_certain_type*, pour créer une fonction SRF avec des paramètres en sortie.

Voici un exemple d'une fonction utilisant RETURN NEXT :

```
CREATE TABLE truc (id_truc INT, sousid_truc INT, nom_truc TEXT);
INSERT INTO truc VALUES (1, 2, 'trois');
INSERT INTO truc VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION obtenir_tous_les_trucs() RETURNS SETOF
 foo AS
$BODY$
DECLARE
    r truc%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM truc WHERE id_truc > 0
    LOOP
        -- quelques traitements
        RETURN NEXT r; -- renvoie la ligne courante du SELECT
    END LOOP;
    RETURN;
END;
$BODY$
LANGUAGE plpgsql;

SELECT * FROM obtenir_tous_les_trucs();
```

Voici un exemple de fonction utilisant RETURN QUERY :

```
CREATE FUNCTION obtient_idvol_disponibles(date) RETURNS SETOF
 integer AS
$BODY$
```

```
BEGIN
    RETURN QUERY SELECT idvol
                   FROM vol
                  WHERE datevol >= $1
                    AND datevol < ($1 + 1);

    -- Comme l'exécution n'est pas terminée, nous vérifions si les
 lignes
    -- ont été renvoyées et levons une exception dans le cas
 contraire.
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Aucun vol à %.', $1;
    END IF;

    RETURN;
 END
$BODY$
LANGUAGE plpgsql;

-- Renvoie les vols disponibles ou lève une exception si aucun vol
-- n'est disponible.
SELECT * FROM obtient_idvol_disponibles(CURRENT_DATE);
```

Note

L'implémentation actuelle de `RETURN NEXT` et de `RETURN QUERY` pour PL/pgSQL récupère la totalité de l'ensemble des résultats avant d'effectuer le retour de la fonction, comme vu plus haut. Cela signifie que si une fonction PL/pgSQL produit une structure résultat très grande, les performances peuvent être faibles : les données seront écrites sur le disque pour éviter un épuisement de la mémoire mais la fonction en elle-même ne renverra rien jusqu'à ce que l'ensemble complet des résultats soit généré. Une version future de PL/pgSQL permettra aux utilisateurs de définir des fonctions renvoyant des ensembles qui n'auront pas cette limitation. Actuellement, le point auquel les données commencent à être écrites sur le disque est contrôlé par la variable de configuration work_mem. Les administrateurs ayant une mémoire suffisante pour enregistrer des ensembles de résultats plus importants en mémoire doivent envisager l'augmentation de ce paramètre.

43.6.2. Retour d'une procédure

Une procédure n'a pas de valeur de retour. De ce fait, une procédure peut se terminer sans instruction `RETURN`. Si vous souhaitez utiliser l'instruction `RETURN` pour quitter le code en avance, écrivez juste `RETURN` sans expression.

Si une procédure a des paramètres en sortie, les valeurs finales des paramètres en sortie seront renvoyées à l'appelant.

43.6.3. Appeler une procédure

Une fonction, une procédure et un bloc `DO` en PL/pgSQL peut appeler une procédure en appelant `CALL`. Les paramètres en sortie sont gérées différemment de la façon dont `CALL` fonctionne en SQL. Chaque paramètre `OUT` ou `INOUT` de la procédure doit correspondre à une variable dans l'instruction `CALL` et le retour de la procédure est affecté à cette variable au retour. Par exemple :

```
CREATE PROCEDURE triple(INOUT x int)
LANGUAGE plpgsql
AS $$
BEGIN
    x := x * 3;
END;
$$;

DO $$
DECLARE myvar int := 5;
BEGIN
  CALL triple(myvar);
  RAISE NOTICE 'myvar = %', myvar;  -- prints 15
END
$$;
```

La variable correspondant à un paramètre en sortie peut être une variable ou un champ simple d'une variable de type composite. Actuellement, cela ne peut pas être un élément d'un tableau.

43.6.4. Contrôles conditionnels

Les instructions `IF` et `CASE` vous permettent d'exécuter des commandes basées sur certaines conditions. PL/pgSQL a trois formes de `IF` :

- `IF ... THEN ... END IF`
- `IF ... THEN ... ELSE ... END IF`
- `IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF`

et deux formes de `CASE` :

- `CASE ... WHEN ... THEN ... ELSE ... END CASE`
- `CASE WHEN ... THEN ... ELSE ... END CASE`

43.6.4.1. `IF-THEN`

```
IF expression-booleenne THEN
    instructions
END IF;
```

Les instructions `IF-THEN` sont la forme la plus simple de `IF`. Les instructions entre `THEN` et `END IF` seront exécutées si la condition est vraie. Autrement, elles seront ignorées.

Exemple :

```
IF v_id_utilisateur <> 0 THEN
    UPDATE utilisateurs SET email = v_email WHERE id_utilisateur =
 v_id_utilisateur;
END IF;
```

43.6.4.2. `IF-THEN-ELSE`

```
IF expression-booleenne THEN
```

```
    instructions
ELSE
    instructions
END IF;
```

Les instructions `IF-THEN-ELSE` s'ajoutent au `IF-THEN` en vous permettant de spécifier un autre ensemble d'instructions à exécuter si la condition n'est pas vraie (notez que ceci inclut le cas où la condition s'évalue à NULL.).

Exemples :

```
IF id_parent IS NULL OR id_parent = ''
THEN
    RETURN nom_complet;
ELSE
    RETURN hp_true_filename(id_parent) || '/' || nom_complet;
END IF;
```

```
IF v_nombre > 0 THEN
    INSERT INTO nombre_utilisateurs (nombre) VALUES (v_nombre);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

43.6.4.3. `IF-THEN-ELSIF`

```
IF expression-booleenne THEN
    instructions
[ ELSIF expression-booleenne THEN
    instructions
[ ELSIF expression-booleenne THEN
    instructions
    ...
]
]
[ ELSE
    instructions ]
END IF;
```

Quelques fois, il existe plus de deux alternatives. `IF-THEN-ELSIF` fournit une méthode agréable pour vérifier différentes alternatives. Les conditions `IF` sont testées successivement jusqu'à trouver la bonne. Alors les instructions associées sont exécutées, puis le contrôle est passé à la prochaine instruction après `END IF`. (Toute autre condition `IF` n'est *pas* testée.) Si aucune des conditions `IF` n'est vraie, alors le bloc `ELSE` (s'il y en a un) est exécuté.

Voici un exemple :

```
IF nombre = 0 THEN
    resultat := 'zero';
ELSIF nombre > 0 THEN
    resultat := 'positif';
ELSIF nombre < 0 THEN
    resultat := 'negatif';
ELSE
```

```
    -- hmm, la seule possibilité est que le nombre soit NULL
    resultat := 'NULL';
END IF;
```

Le mot clé `ELSIF` peut aussi s'écrire `ELSEIF`.

Une façon alternative d'accomplir la même tâche est d'intégrer les instructions `IF-THEN-ELSE`, comme dans l'exemple suivant :

```
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

Néanmoins, cette méthode requiert d'écrire un `END IF` pour chaque `IF`, donc c'est un peu plus compliqué que d'utiliser `ELSIF` quand il y a beaucoup d'autres alternatives.

43.6.4.4. CASE simple

```
CASE expression_recherche
    WHEN expression [, expression [ ... ]] THEN
      instructions
  [ WHEN expression [, expression [ ... ]] THEN
      instructions
    ... ]
  [ ELSE
      instructions ]
END CASE;
```

La forme simple de `CASE` fournit une exécution conditionnelle basée sur l'égalité des opérandes. L'*`expression-recherche`* est évaluée (une fois) puis comparée successivement à chaque *`expression`* dans les clauses `WHEN`. Si une correspondance est trouvée, alors les *`instructions`* correspondantes sont exécutées, puis le contrôle est passé à la prochaine instruction après `END CASE`. (Les autres expressions `WHEN` ne sont pas testées.) Si aucune correspondance n'est trouvée, les *`instructions`* du bloc `ELSE` sont exécutées ; s'il n'y a pas de bloc `ELSE`, une exception `CASE_NOT_FOUND` est levée.

Voici un exemple simple :

```
CASE x
    WHEN 1, 2 THEN
        msg := 'un ou deux';
    ELSE
        msg := 'autre valeur que un ou deux';
END CASE;
```

43.6.4.5. CASE recherché

```
CASE
    WHEN expression_booléenne THEN
      instructions
  [ WHEN expression_booléenne THEN
      instructions
    ... ]
  [ ELSE
      instructions ]
END CASE;
```

La forme recherchée de CASE fournit une exécution conditionnelle basée sur la vérification d'expressions booléennes. Chaque *expression-booléenne* de la clause WHEN est évaluée à son tour jusqu'à en trouver une qui est validée (true). Les *instructions* correspondantes sont exécutées, puis le contrôle est passé à la prochaine instruction après END CASE. (Les expressions WHEN suivantes ne sont pas testées.) Si aucun résultat vrai n'est trouvé, les *instructions* du bloc ELSE sont exécutées. Si aucun bloc ELSE n'est présent, une exception CASE_NOT_FOUND est levée.

Voici un exemple :

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'valeur entre zéro et dix';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'valeur entre onze et vingt';
END CASE;
```

Cette forme de CASE est entièrement équivalente à IF-THEN-ELSIF, sauf pour la règle qui dit qu'atteindre une clause ELSE omise résulte dans une erreur plutôt que ne rien faire.

43.6.5. Boucles simples

Grâce aux instructions LOOP, EXIT, CONTINUE, WHILE FOR et FOREACH, vous pouvez faire en sorte que vos fonctions PL/pgSQL répètent une série de commandes.

43.6.5.1. LOOP

```
[<<label>>]
LOOP
    instructions
END LOOP [ label ];
```

LOOP définit une boucle inconditionnelle répétée indéfiniment jusqu'à ce qu'elle soit terminée par une instruction EXIT ou RETURN. Le *label* optionnel peut être utilisé par les instructions EXIT et CONTINUE dans le cas de boucles imbriquées pour définir la boucle impliquée.

43.6.5.2. EXIT

```
EXIT [ label ] [ WHEN expression-booléenne ];
```

Si aucun *label* n'est donné, la boucle la plus imbriquée se termine et l'instruction suivant END LOOP est exécutée. Si un *label* est donné, ce doit être le label de la boucle, du bloc courant ou d'un niveau moins imbriqué. La boucle ou le bloc nommé se termine alors et le contrôle continue avec l'instruction située après le END de la boucle ou du bloc correspondant.

Si WHEN est spécifié, la sortie de boucle ne s'effectue que si *expression-booléenne* est vraie. Sinon, le contrôle passe à l'instruction suivant le EXIT.

EXIT peut être utilisé pour tous les types de boucles ; il n'est pas limité aux boucles non conditionnelles.

Lorsqu'il est utilisé avec un bloc BEGIN, EXIT passe le contrôle à la prochaine instruction après la fin du bloc. Notez qu'un label doit être utilisé pour cela ; un EXIT sans label n'est jamais pris en compte pour correspondre à un bloc BEGIN. (Ceci est un changement de la version 8.4 de PostgreSQL. Auparavant, il était permis de faire correspondre un EXIT sans label avec un bloc BEGIN.)

Exemples :

```
LOOP
    -- quelques traitements
    IF nombre > 0 THEN
        EXIT;  -- sortie de boucle
    END IF;
END LOOP;

LOOP
    -- quelques traitements
    EXIT WHEN nombre > 0;
END LOOP;

<<un_bloc>>
BEGIN
    -- quelques traitements
    IF stocks > 100000 THEN
        EXIT un_bloc;  -- cause la sortie (EXIT) du bloc BEGIN
    END IF;
    -- les traitements ici seront ignorés quand stocks > 100000
END;
```

43.6.5.3. CONTINUE

```
CONTINUE [ label ] [ WHEN expression-booléenne ];
```

Si aucun *label* n'est donné, la prochaine itération de la boucle interne est commencée. C'est-à-dire que toutes les instructions restantes dans le corps de la boucle sont ignorées et le contrôle revient à l'expression de contrôle de la boucle pour déterminer si une autre itération de boucle est nécessaire. Si le *label* est présent, il spécifie le label de la boucle dont l'exécution va être continuée.

Si WHEN est spécifié, la prochaine itération de la boucle est commencée seulement si l'*expression-booléenne* est vraie. Sinon, le contrôle est passé à l'instruction après CONTINUE.

CONTINUE peut être utilisé avec tous les types de boucles ; il n'est pas limité à l'utilisation des boucles inconditionnelles.

Exemples :

```
LOOP
    -- quelques traitements
    EXIT WHEN nombre > 100;
    CONTINUE WHEN nombre < 50;
    -- quelques traitements pour nombre IN [50 .. 100]
END LOOP;
```

43.6.5.4. `WHILE`

```
[<<label>>]
WHILE expression-booléenne LOOP
    instructions
END LOOP [ label ];
```

L'instruction `WHILE` répète une séquence d'instructions aussi longtemps que *`expression-booléenne`* est évaluée à vrai. L'expression est vérifiée juste avant chaque entrée dans le corps de la boucle.

Par exemple :

```
WHILE montant_possede > 0 AND balance_cadeau > 0 LOOP
    -- quelques traitements ici
END LOOP;

WHILE NOT termine LOOP
    -- quelques traitements ici
END LOOP;
```

43.6.5.5. `FOR` (variante avec entier)

```
[<<label>>]
FOR nom IN [ REVERSE ] expression .. expression [ BY expression ]
 LOOP
    instruction
END LOOP [ label ];
```

Cette forme de `FOR` crée une boucle qui effectue une itération sur une plage de valeurs entières. La variable *`nom`* est automatiquement définie comme un type `integer` et n'existe que dans la boucle (toute définition de la variable est ignorée à l'intérieur de la boucle). Les deux expressions donnant les limites inférieures et supérieures de la plage sont évaluées une fois en entrant dans la boucle. Si la clause `BY` n'est pas spécifiée, l'étape d'itération est de 1, sinon elle est de la valeur spécifiée dans la clause `BY`, qui est évaluée encore une fois à l'entrée de la boucle. Si `REVERSE` est indiquée, alors la valeur de l'étape est soustraite, plutôt qu'ajoutée, après chaque itération.

Quelques exemples de boucles `FOR` avec entiers :

```
FOR i IN 1..10 LOOP
    -- prend les valeurs 1,2,3,4,5,6,7,8,9,10 dans la boucle
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- prend les valeurs 10,9,8,7,6,5,4,3,2,1 dans la boucle
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- prend les valeurs 10,8,6,4,2 dans la boucle
END LOOP;
```

Si la limite basse est plus grande que la limite haute (ou moins grande dans le cas du `REVERSE`), le corps de la boucle n'est pas exécuté du tout. Aucune erreur n'est renvoyée.

Si un *label* est attaché à la boucle FOR, alors la variable entière de boucle peut être référencée avec un nom qualifié en utilisant ce *label*.

43.6.6. Boucler dans les résultats de requêtes

En utilisant un type de FOR différent, vous pouvez itérer au travers des résultats d'une requête et par là-même manipuler ces données. La syntaxe est la suivante :

```
[<<label>>]
FOR cible IN requête LOOP
    instructions
END LOOP [ label ];
```

La *cible* est une variable de type record, row ou une liste de variables scalaires séparées par une virgule. La *cible* est affectée successivement à chaque ligne résultant de la *requête* et le corps de la boucle est exécuté pour chaque ligne. Voici un exemple :

```
CREATE FUNCTION rafraichir_vuemat() RETURNS integer AS $$
DECLARE
    vues_mat RECORD;
BEGIN
    RAISE NOTICE 'Rafraichissement de toutes les vues
 matérialisées...';

    FOR mviews IN
       SELECT n.nspname AS mv_schema,
              c.relname AS mv_name,
              pg_catalog.pg_get_userbyid(c.relowner) AS owner
         FROM pg_catalog.pg_class c
    LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.relnamespace)
        WHERE c.relkind = 'm'
     ORDER BY 1
    LOOP

        -- Maintenant "mviews" contient un enregistrement avec les
 informations sur la vue matérialisée

        RAISE NOTICE 'Rafraichissement de la vue matérialisée %.%
 (propriétaire : %)...',
                     quote_ident(mviews.mv_schema),
                     quote_ident(mviews.mv_name),
                     quote_ident(mviews.owner);
        EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I',
 mviews.mv_schema, mviews.mv_name);
    END LOOP;

    RAISE NOTICE 'Fin du rafraichissement des vues matérialisées.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Si la boucle est terminée par une instruction EXIT, la dernière valeur ligne affectée est toujours accessible après la boucle.

La *requête* utilisée dans ce type d'instruction FOR peut être toute commande SQL qui renvoie des lignes à l'appelant : SELECT est le cas le plus commun mais vous pouvez aussi utiliser INSERT,

UPDATE ou DELETE avec une clause RETURNING. Certaines commandes comme EXPLAIN fonctionnent aussi.

Les variables PL/pgSQL sont remplacées par les paramètres de la requête, et le plan de requête est mis en cache pour une réutilisation possible. C'est couvert en détail dans la Section 43.11.1 et dans la Section 43.11.2.

L'instruction FOR-IN-EXECUTE est un moyen d'itérer sur des lignes :

```
[<<label>>]
FOR target IN EXECUTE text_expression [ USING expression [, ...] ]
 LOOP
    instructions
END LOOP [ label ];
```

Ceci est identique à la forme précédente, à ceci près que l'expression de la requête source est spécifiée comme une expression chaîne, évaluée et replanifiée à chaque entrée dans la boucle FOR. Ceci permet au développeur de choisir entre la vitesse d'une requête préplanifiée et la flexibilité d'une requête dynamique, uniquement avec l'instruction EXECUTE. Comme avec EXECUTE, les valeurs de paramètres peuvent être insérées dans la commande dynamique via USING.

Une autre façon de spécifier la requête dont les résultats devront être itérés est de la déclarer comme un curseur. Ceci est décrit dans Section 43.7.4.

43.6.7. Boucler dans des tableaux

La boucle FOREACH ressemble beaucoup à une boucle FOR mais, au lieu d'itérer sur les lignes renvoyées par une requêtes SQL, elle itère sur les éléments d'une valeur de type tableau. (En général, FOREACH est fait pour boucler sur les composants d'une expression composite ; les variantes pour boucler sur des composites en plus des tableaux pourraient être ajoutées dans le futur.) L'instruction FOREACH pour boucler sur un tableau est :

```
[ <<label>> ]
FOREACH target [ SLICE nombre ] IN ARRAY expression LOOP
    instructions
END LOOP [ label ];
```

Sans SLICE ou si SLICE 0 est indiqué, la boucle itère au niveau des éléments individuels du tableau produit par l'évaluation de l'*expression*. La variable *cible* se voit affectée chaque valeur d'élément en séquence, et le corps de la boucle est exécuté pour chaque élément. Voici un exemple de boucle sur les éléments d'un tableau d'entiers :

```
CREATE FUNCTION somme(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Les éléments sont parcourus dans l'ordre de leur stockage, quelque soit le nombre de dimensions du tableau. Bien que la *cible* est habituellement une simple variable, elle peut être une liste de variables lors d'une boucle dans un tableau de valeurs composites (des enregistrements). Dans ce cas, pour chaque élément du tableau, les variables se voient affectées les colonnes de la valeur composite.

Avec une valeur SLICE positive, FOREACH itère au travers des morceaux du tableau plutôt que des éléments seuls. La valeur de SLICE doit être un entier constant, moins large que le nombre de dimensions du tableau. La variable *cible* doit être un tableau et elle reçoit les morceaux successifs de la valeur du tableau, où chaque morceau est le nombre de dimensions indiquées par SLICE. Voici un exemple d'itération sur des morceaux à une dimension :

```
CREATE FUNCTION parcourt_lignes(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'ligne = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT parcourt_lignes(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  ligne = {1,2,3}
NOTICE:  ligne = {4,5,6}
NOTICE:  ligne = {7,8,9}
NOTICE:  ligne = {10,11,12}
```

43.6.8. Récupérer les erreurs

Par défaut, toute erreur survenant dans une fonction PL/pgSQL annule l'exécution de la fonction mais aussi de la transaction qui l'entoure. Vous pouvez récupérer les erreurs en utilisant un bloc BEGIN avec une clause EXCEPTION. La syntaxe est une extension de la syntaxe habituelle pour un bloc BEGIN :

```
[ <<label>> ]
[ DECLARE
  declarations ]
BEGIN
instructions
EXCEPTION
WHEN condition [ OR condition ... ] THEN
instructions_gestion_erreurs
[ WHEN condition [ OR condition ... ] THEN
  instructions_gestion_erreurs
  ... ]
END;
```

Si aucune erreur ne survient, cette forme de bloc exécute simplement toutes les *instructions* puis passe le contrôle à l'instruction suivant END. Mais si une erreur survient à l'intérieur des *instructions*, le traitement en cours des *instructions* est abandonné et le contrôle est passé à la liste d'EXCEPTION. Une recherche est effectuée sur la liste pour la première *condition* correspondant à l'erreur survenue. Si une correspondance est trouvée, les *instructions_gestion_erreurs* correspondantes sont exécutées puis le contrôle est passé à l'instruction suivant le END. Si aucune correspondance n'est trouvée, l'erreur se propage comme si la

clause EXCEPTION n'existait pas du tout : l'erreur peut être récupérée par un bloc l'enfermant avec EXCEPTION ou, s'il n'existe pas, elle annule le traitement de la fonction.

Les noms des *condition* sont indiquées dans l'Annexe A. Un nom de catégorie correspond à toute erreur contenue dans cette catégorie. Le nom de condition spéciale OTHERS correspond à tout type d'erreur sauf QUERY_CANCELED et ASSERT_FAILURE (il est possible, mais pas recommandé, de récupérer ces deux types d'erreur par leur nom). Les noms des conditions ne sont pas sensibles à la casse. De plus, une condition d'erreur peut être indiquée par un code SQLSTATE ; par exemple, ces deux cas sont équivalents :

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

Si une nouvelle erreur survient à l'intérieur des *instructions_gestion_erreurs* sélectionnées, elle ne peut pas être récupérée par cette clause EXCEPTION mais est propagée en dehors. Une clause EXCEPTION l'englobant pourrait la récupérer.

Quand une erreur est récupérée par une clause EXCEPTION, les variables locales de la fonction PL/pgSQL restent dans le même état qu'au moment où l'erreur est survenue mais toutes les modifications à l'état persistant de la base de données à l'intérieur du bloc sont annulées. Comme exemple, considérez ce fragment :

```
INSERT INTO mon_tableau(prenom, nom) VALUES('Tom', 'Jones');
BEGIN
  UPDATE mon_tableau SET prenom = 'Joe' WHERE nom = 'Jones';
  x := x + 1;
  y := x / 0;
  EXCEPTION
    WHEN division_by_zero THEN
      RAISE NOTICE 'récupération de l''erreur division_by_zero';
RETURN x;
END;
```

Quand le contrôle parvient à l'affectation de y, il échouera avec une erreur division_by_zero. Elle sera récupérée par la clause EXCEPTION. La valeur renvoyée par l'instruction RETURN sera la valeur incrémentée de x mais les effets de la commande UPDATE auront été annulés. La commande INSERT précédant le bloc ne sera pas annulée, du coup le résultat final est que la base de données contient Tom Jones et non pas Joe Jones.

> **Astuce**
>
> Un bloc contenant une clause EXCEPTION est significativement plus coûteuse en entrée et en sortie qu'un bloc sans. Du coup, n'utilisez pas EXCEPTION sans besoin.

Exemple 43.2. Exceptions avec UPDATE/INSERT

Cet exemple utilise un gestionnaire d'exceptions pour réaliser soit un UPDATE soit un INSERT, comme approprié. Il est recommandé d'utiliser la commande INSERT avec la clause ON CONFLICT DO UPDATE plutôt que cette logique. Cet exemple ne sert qu'à illustrer l'usage des structures de contrôle de PL/pgSQL :

```
CREATE TABLE base (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION fusionne_base(cle INT, donnee TEXT) RETURNS VOID AS
$$
```

```
BEGIN
    LOOP
        -- commençons par tenter la mise à jour de la clé
        UPDATE base SET b = donnee WHERE a = cle;
        IF found THEN
            RETURN;
        END IF;

        -- si elle n'est pas dispo, tentons l'insertion de la clé
        -- si quelqu'un essaie d'insérer la même clé en même temps,
        -- il y aura une erreur pour violation de clé unique
        BEGIN
            INSERT INTO base(a,b) VALUES (cle, donnee);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- ne rien faire, et tente de nouveau la mise à jour
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT fusionne_base(1, 'david');
SELECT fusionne_base(1, 'dennis');
```

Ce code suppose que l'erreur `unique_violation` est causée par la commande `INSERT`, et pas par un `INSERT` dans une fonction trigger sur la table. Cela pourrait avoir un mauvais comportement s'il y a plus d'un index unique sur la table car il ré-essaiera l'opération quelque soit l'index qui a causé l'erreur. On pourrait avoir plus de sécurité en utilisant la fonctionnalité discuté ci-après pour vérifier que l'erreur récupérée était celle attendue.

43.6.8.1. Obtenir des informations sur une erreur

Les gestionnaires d'exception ont fréquemment besoin d'identifier l'erreur spécifique qui est survenue. Il existe deux façons d'obtenir l'information sur l'exception en cours dans PL/pgSQL : des variables spéciales et la commande `GET STACKED DIAGNOSTICS`.

Avec un gestionnaire d'exceptions, la variable spéciale `SQLSTATE` contient le code d'erreur qui correspond à l'exception qui a été levée (voir Tableau A.1 pour la liste de codes d'erreur possibles). La variable spéciale `SQLERRM` contient le message d'erreur associé à l'exception. Ces variables ne sont pas définies en dehors des gestionnaires d'exception.

Dans le gestionnaire d'exceptions, il est possible de récupérer des informations sur l'exception en cours en utilisant la commande `GET STACKED DIAGNOSTICS` qui a la forme :

```
GET STACKED DIAGNOSTICS variable { = | := } élément [ , ... ];
```

Chaque *`élément`* est un mot clé identifiant une valeur de statut à assigner à la *`variable`* spécifiée (qui doit être du bon type de données). Les éléments de statut actuellement disponibles sont indiqués dans Tableau 43.2.

Tableau 43.2. Diagnostics et erreurs

Nom	Type	Description
`RETURNED_SQLSTATE`	`text`	le code d'erreur SQLSTATE de l'exception

Nom	Type	Description
COLUMN_NAME	text	le nom de la colonne en relation avec l'exception
CONSTRAINT_NAME	text	le nom de la contrainte en relation avec l'exception
PG_DATATYPE_NAME	text	le nom du type de données en relation avec l'exception
MESSAGE_TEXT	text	le texte du message principal de l'exception
TABLE_NAME	text	le nom de la table en relation avec l'exception
SCHEMA_NAME	text	le nom du schéma en relation avec l'exception
PG_EXCEPTION_DETAIL	text	le texte du message détaillée de l'exception, si disponible
PG_EXCEPTION_HINT	text	le texte du message d'astuce de l'exception, si disponible
PG_EXCEPTION_CONTEXT	text	ligne(s) de texte décrivant la pile d'appel au moment de l'exception (voir Section 43.6.9)

Si l'exception n'a pas configuré une valeur pour un élément, une chaîne vide sera renvoyée.

Voici un exemple :

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;
BEGIN
  -- un traitement qui cause une exception
  ...
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = PG_EXCEPTION_DETAIL,
                          text_var3 = PG_EXCEPTION_HINT;
END;
```

43.6.9. Obtenir des informations sur l'emplacement d'exécution

La commande `GET DIAGNOSTICS`, précédemment décrite dans Section 43.5.5, récupère des informations sur l'état d'exécution courant (alors que la commande `GET STACKED DIAGNOSTICS` discutée ci-dessus rapporte des informations sur l'état d'exécution de l'erreur précédente). Son élément de statut `PG_CONTEXT` est utile pour vérifier l'emplacement d'exécution courant. `PG_CONTEXT` renvoie une chaîne de texte dont les lignes correspondent à la pile d'appels. La première ligne fait référence à la fonction en cours et qui exécute `GET DIAGNOSTICS`. La seconde ligne et toutes les lignes suivantes font référence aux fonctions appelantes dans la pile d'appel. Par exemple :

```
CREATE OR REPLACE FUNCTION fonction_externe() RETURNS integer AS $
$BEGIN
```

```
  RETURN fonction_interne();
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION fonction_interne() RETURNS integer AS $$
DECLARE
  stack text;
BEGIN
  GET DIAGNOSTICS stack = PG_CONTEXT;
  RAISE NOTICE E'--- Pile d''appel ---\n%', stack;
  RETURN 1;
END;
$$ LANGUAGE plpgsql;

SELECT fonction_externe();

NOTICE:  --- Call Stack ---
PL/pgSQL function fonction_interne() line 5 at GET DIAGNOSTICS
PL/pgSQL function fonction_externe() line 3 at RETURN
CONTEXT:  PL/pgSQL function fonction_externe() line 3 at RETURN
 fonction_externe
 -----------------
                1
(1 row)
```

`GET STACKED DIAGNOSTICS ... PG_EXCEPTION_CONTEXT` renvoie le même type de pile d'appels, mais en décrivant l'emplacement où l'erreur a été détectée, plutôt que l'emplacement actuel.

La clause `SCROLL` ne peut pas être utilisée quand la requête du curseur utilise `FOR UPDATE/SHARE`. De plus, il est préférable d'utiliser `NO SCROLL` avec une requête qui implique des fonctions volatiles. L'implémentation de `SCROLL` suppose que relire la sortie de la requête donnera des résultats cohérents, ce qu'une fonction volatile pourrait ne pas faire.

43.7. Curseurs

Plutôt que d'exécuter la totalité d'une requête à la fois, il est possible de créer un *curseur* qui encapsule la requête, puis en lit le résultat quelques lignes à la fois. Une des raisons pour faire de la sorte est d'éviter les surcharges de mémoire quand le résultat contient un grand nombre de lignes (cependant, les utilisateurs PL/pgSQL n'ont généralement pas besoin de se préoccuper de cela puisque les boucles `FOR` utilisent automatiquement un curseur en interne pour éviter les problèmes de mémoire). Un usage plus intéressant est de renvoyer une référence à un curseur qu'une fonction a créé, permettant à l'appelant de lire les lignes. C'est un moyen efficace de renvoyer de grands ensembles de lignes à partir des fonctions.

43.7.1. Déclaration de variables curseur

Tous les accès aux curseurs dans PL/pgSQL se font par les variables curseur, qui sont toujours du type de données spécial `refcursor`. Un des moyens de créer une variable curseur est de simplement la déclarer comme une variable de type `refcursor`. Un autre moyen est d'utiliser la syntaxe de déclaration de curseur qui est en général :

nom [[NO] SCROLL] CURSOR [(*arguments*)] FOR *requête*;

(`FOR` peut être remplacé par `IS` pour la compatibilité avec Oracle). Si `SCROLL` est spécifié, le curseur sera capable d'aller en sens inverse ; si `NO SCROLL` est indiqué, les récupérations en sens inverses

seront rejetées ; si rien n'est indiqué, cela dépend de la requête. *`arguments`* est une liste de paires de *`nom type-de-donnée`* qui définit les noms devant être remplacés par les valeurs des paramètres dans la requête donnée. La valeur effective à substituer pour ces noms sera indiquée plus tard lors de l'ouverture du curseur.

Quelques exemples :

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (cle integer) FOR SELECT * FROM tenk1 WHERE
 unique1 = cle;
```

Ces variables sont toutes trois du type de données `refcursor` mais la première peut être utilisée avec n'importe quelle requête alors que la seconde a une requête complètement spécifiée qui lui est déjà *liée*, et la dernière est liée à une requête paramétrée (`cle` sera remplacée par un paramètre de valeur entière lors de l'ouverture du curseur). La variable `curs1` est dite *non liée* puisqu'elle n'est pas liée à une requête particulière.

L'option `SCROLL` ne peut pas être utilisée quand la requête du curseur utilise `FOR UPDATE/SHARE`. De plus, il est préférable d'utiliser `NO SCROLL` avec une requête qui implique des fonctions volatiles. L'implémentation de `SCROLL` suppose que relire la sortie de la requête donnera des résultats cohérents, ce qu'une fonction volatile ne peut pas faire.

43.7.2. Ouverture de curseurs

Avant qu'un curseur puisse être utilisé pour rapatrier des lignes, il doit être *ouvert* (c'est l'action équivalente de la commande SQL `DECLARE CURSOR`). PL/pgSQL dispose de trois formes pour l'instruction `OPEN`, dont deux utilisent des variables curseur non liées et la dernière une variable curseur liée.

> **Note**
>
> Les variables des curseurs liés peuvent aussi être utilisés sans les ouvrir explicitement, via l'instruction `FOR` décrite dans Section 43.7.4.

43.7.2.1. `OPEN FOR` *`requête`*

```
OPEN var_curseur_nonlie [ [ NO ] SCROLL ] FOR requete;
```

La variable curseur est ouverte et reçoit la requête spécifiée à exécuter. Le curseur ne peut pas être déjà ouvert, et il doit avoir été déclaré comme une variable de curseur non lié (c'est-à-dire comme une simple variable `refcursor`). La requête doit être un `SELECT` ou quelque chose d'autre qui renvoie des lignes (comme `EXPLAIN`). La requête est traitée de la même façon que les autres commandes SQL dans PL/pgSQL : les noms de variables PL/pgSQL sont substitués et le plan de requête est mis en cache pour une possible ré-utilisation. Quand une variable PL/pgSQL est substituée dans une requête de type curseur, la valeur qui est substituée est celle qu'elle avait au moment du `OPEN` ; les modifications ultérieures n'auront pas affectées le comportement du curseur. Les options `SCROLL` et `NO SCROLL` ont la même signification que pour un curseur lié.

Exemple :

```
OPEN curs1 FOR SELECT * FROM foo WHERE cle = ma_cle;
```

43.7.2.2. OPEN FOR EXECUTE

```
OPEN var_curseur_nonlie [ [ NO ] SCROLL ] FOR EXECUTE requete
                                        [ USING expression [, ... ] ];
```

La variable curseur est ouverte et reçoit la requête spécifiée à exécuter. Le curseur ne peut pas être déjà ouvert et il doit avoir été déclaré comme une variable de curseur non lié (c'est-à-dire comme une simple variable `refcursor`). La requête est spécifiée comme une expression chaîne de la même façon que dans une commande `EXECUTE`. Comme d'habitude, ceci donne assez de flexibilité pour que le plan de la requête puisse changer d'une exécution à l'autre (voir la Section 43.11.2), et cela signifie aussi que la substitution de variable n'est pas faite sur la chaîne de commande. Comme avec la commande `EXECUTE`, les valeurs de paramètre peuvent être insérées dans la commande dynamique avec `format()` ou `USING`. Les options `SCROLL` et `NO SCROLL` ont la même signification que pour un curseur lié.

Exemple :

```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE nom_colonne =
 $1', ma_table) USING valeur_clef;
```

Dans cet exemple, le nom de la table est inséré dans la requête via la fonction `format()`. La valeur de la colonne `nom_colonne` utilisée pour la comparaison est insérée via le paramètre `USING`, c'est la raison pour laquelle elle n'a pas besoin d'être échappée.

43.7.2.3. Ouverture d'un curseur lié

```
OPEN var_curseur_lié [ ( [ nom_argument := ] valeur_argument
 [, ...] ) ];
```

Cette forme d'`OPEN` est utilisée pour ouvrir une variable curseur à laquelle la requête est liée au moment de la déclaration. Le curseur ne peut pas être déjà ouvert. Une liste des expressions arguments doit apparaître si et seulement si le curseur a été déclaré comme acceptant des arguments. Ces valeurs seront remplacées dans la requête.

Le plan de requête pour un curseur lié est toujours considéré comme pouvant être mis en cache ; il n'y a pas d'équivalent de la commande `EXECUTE` dans ce cas. Notez que `SCROLL` et `NO SCROLL` ne peuvent pas être indiqués dans `OPEN` car le comportement du curseur était déjà déterminé.

Les valeurs des arguments peuvent être passées en utilisant soit la notation *en position* soit la notation *nommée*. Dans la première, tous les arguments sont indiqués dans l'ordre. Dans la seconde, chaque nom d'argument est indiqué en utilisant `:=` pour la séparer de l'expression de l'argument. De façon similaire à l'appel de fonctions, décrit dans Section 4.3, il est aussi autorisé de mixer notation en position et notation nommée.

Voici quelques exemples (ils utilisent les exemples de déclaration de curseur ci-dessus) :

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

Comme la substitution de variable est faite sur la requête d'un curseur lié, il existe en fait deux façons de passer les valeurs au curseur : soit avec un argument explicite pour `OPEN` soit en référençant implicitement une variable PL/pgSQL dans la requête. Néanmoins, seules les variables déclarées avant que le curseur lié ne soit déclaré lui seront substituées. Dans tous les cas, la valeur passée est déterminée

au moment de l'exécution de la commande OPEN. Par exemple, une autre façon d'obtenir le même effet que l'exemple curs3 ci-dessus est la suivante :

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

43.7.3. Utilisation des curseurs

Une fois qu'un curseur a été ouvert, il peut être manipulé grâce aux instructions décrites ci-dessous.

Ces manipulations n'ont pas besoin de se dérouler dans la même fonction que celle qui a ouvert le curseur. Vous pouvez renvoyer une valeur refcursor à partir d'une fonction et laisser l'appelant opérer sur le curseur (d'un point de vue interne, une valeur refcursor est simplement la chaîne de caractères du nom d'un portail contenant la requête active pour le curseur. Ce nom peut être passé à d'autres, affecté à d'autres variables refcursor et ainsi de suite, sans déranger le portail).

Tous les portails sont implicitement fermés à la fin de la transaction. C'est pourquoi une valeur refcursor est utilisable pour référencer un curseur ouvert seulement jusqu'à la fin de la transaction.

43.7.3.1. FETCH

```
FETCH [ direction { FROM | IN } ] curseur INTO cible;
```

FETCH récupère la prochaine ligne (dans la direction indiquée) à partir d'un curseur et la place dans une cible, qui peut être une variable ligne, une variable record ou une liste de variables simples séparées par des virgules, comme dans un SELECT INTO. S'il n'y a pas de ligne convenable, la cible est mise à NULL. Comme avec SELECT INTO, la variable spéciale FOUND peut être lue pour voir si une ligne a été récupérée. Si aucune ligne n'est obtenue, le curseur est positionné après la dernière ligne ou avant la première ligne, suivant la direction du mouvement.

La clause *direction* peut être une des variantes suivantes autorisées pour la commande SQL FETCH sauf celles qui peuvent récupérer plus d'une ligne ; nommément, cela peut être NEXT, PRIOR, FIRST, LAST, ABSOLUTE *nombre*, RELATIVE *nombre*, FORWARD ou BACKWARD. Omettre *direction* est identique à spécifier NEXT. Quand la syntaxe utilise un *count*, le *count* peut être une expression de type integer (contrairement à la commande SQL FETCH, qui autorise seulement une constante de type integer). Les valeurs *direction* qui nécessitent d'aller en sens inverse risquent d'échouer sauf si le curseur a été déclaré ou ouvert avec l'option SCROLL.

curseur doit être le nom d'une variable refcursor qui référence un portail de curseur ouvert.

Exemples :

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

43.7.3.2. MOVE

```
MOVE [ direction { FROM | IN } ] curseur;
```

`MOVE` repositionne un curseur sans récupérer de données. `MOVE` fonctionne comme la commande `FETCH` sauf qu'elle ne fait que repositionner le curseur et ne renvoie donc pas les lignes du déplacement. La clause *`direction`* peut valoir toute variante autorisée dans la commande SQL FETCH, incluant celles qui récupèrent plus d'une ligne ; le curseur est positionné sur la dernière ligne. (Néanmoins, le cas où la clause *`direction`* est simplement une expression *`count`* sans mot clé est maintenant obsolète dans PL/pgSQL. La syntaxe est ambigu dans le cas où la clause *`direction`* est omise et donc cela pourrait échouer si *`count`* n'est pas une constante.) Comme avec `SELECT INTO`, la variable spéciale `FOUND` peut être lue pour vérifier s'il y avait bien les lignes correspondant au déplacement. Si aucune ligne n'est obtenue, le curseur est positionné après la dernière ligne ou avant la première ligne, suivant la direction du mouvement.

Exemples :

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

43.7.3.3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF curseur;
DELETE FROM table WHERE CURRENT OF curseur;
```

Quand un curseur est positionné sur une ligne d'une table, cette ligne peut être mise à jour ou supprimée en utilisant le curseur qui identifie la ligne. Il existe des restrictions sur ce que peut être la requête du curseur (en particulier, pas de regroupement) et il est mieux d'utiliser `FOR UPDATE` dans le curseur. Pour des informations supplémentaires, voir la page de référence DECLARE.

Un exemple :

```
UPDATE foo SET valdonnee = mavaleur WHERE CURRENT OF curs1;
```

43.7.3.4. CLOSE

```
CLOSE curseur;
```

`CLOSE` ferme le portail sous-tendant un curseur ouvert. Ceci peut être utilisé pour libérer des ressources avant la fin de la transaction ou pour libérer la variable curseur pour pouvoir la rouvrir.

Exemple :

```
CLOSE curs1;
```

43.7.3.5. Renvoi de curseurs

Les fonctions PL/pgSQL peuvent renvoyer des curseurs à l'appelant. Ceci est utile pour renvoyer plusieurs lignes ou colonnes, spécialement avec des ensembles de résultats très grands. Pour cela, la

fonction ouvre le curseur et renvoie le nom du curseur à l'appelant (ou simplement ouvre le curseur en utilisant un nom de portail spécifié par ou autrement connu par l'appelant). L'appelant peut alors récupérer les lignes à partir du curseur. Le curseur peut être fermé par l'appelant ou il sera fermé automatiquement à la fin de la transaction.

Le nom du portail utilisé pour un curseur peut être spécifié par le développeur ou peut être généré automatiquement. Pour spécifier un nom de portail, affectez simplement une chaîne à la variable `refcursor` avant de l'ouvrir. La valeur de la variable `refcursor` sera utilisée par `OPEN` comme nom du portail sous-jacent. Néanmoins, si la variable `refcursor` est NULL, `OPEN` génère automatiquement un nom qui n'entre pas en conflit avec tout portail existant et l'affecte à la variable `refcursor`.

> **Note**
>
> Une variable curseur avec limites est initialisée avec la valeur de la chaîne représentant son nom, de façon à ce que le nom du portail soit identique au nom de la variable curseur, sauf si le développeur le surcharge par affectation avant d'ouvrir le curseur. Mais, une variable curseur sans limite aura par défaut la valeur NULL, dont il reçoit un nom unique généré automatiquement sauf s'il est surchargé.

L'exemple suivant montre une façon de fournir un nom de curseur par l'appelant :

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION fonction_reference(refcursor) RETURNS refcursor AS
 $$
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
$$ LANGUAGE plpgsql;

BEGIN;
SELECT fonction_reference('curseur_fonction');
FETCH ALL IN curseur_fonction;
COMMIT;
```

L'exemple suivant utilise la génération automatique du nom du curseur :

```
CREATE FUNCTION fonction_reference2() RETURNS refcursor AS $$
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
$$ LANGUAGE plpgsql;

-- Il faut être dans une transaction pour utiliser les curseurs.
BEGIN;
SELECT fonction_reference2();

   fonction_reference2
--------------------------
 <unnamed cursor 1>
```

```
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

L'exemple suivant montre une façon de renvoyer plusieurs curseurs à une seule fonction :

```
CREATE FUNCTION ma_fonction(refcursor, refcursor) RETURNS SETOF
 refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- doit être dans une transaction pour utiliser les curseurs.
BEGIN;

SELECT * FROM ma_fonction('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

43.7.4. Boucler dans les résultats d'un curseur

C'est une variante de l'instruction `FOR` qui permet l'itération sur les lignes renvoyées par un curseur. La syntaxe est :

```
[ <<label>> ]
FOR var_record IN var_curseur_lié [ ( [ nom_argument
 := ] valeur_argument [, ...] ) ] LOOP
    instructions
END LOOP [ label ];
```

La variable curseur doit avoir été liée à une requête lors de sa déclaration et il *ne peut pas* être déjà ouvert. L'instruction `FOR` ouvre automatiquement le curseur, et il ferme le curseur en sortie de la boucle. Une liste des expressions de valeurs des arguments doit apparaître si et seulement si le curseur a été déclaré prendre des arguments. Ces valeurs seront substituées dans la requête, de la même façon que lors d'un `OPEN` (voir Section 43.7.2.3).

La variable *`var_record`* est définie automatiquement avec le type `record` et existe seulement dans la boucle (toute définition existante d'un nom de variable est ignorée dans la boucle). Chaque ligne renvoyée par le curseur est successivement affectée à la variable d'enregistrement et le corps de la boucle est exécuté.

43.8. Gestion des transactions

Une nouvelle transaction démarre avec les caractéristiques de transaction avec leurs valeurs par défaut, comme le niveau d'isolation. Dans le cas où les transactions sont validées à l'intérieur d'une boucle, on peut désirer qu'une transaction démarre automatiquement avec les mêmes caractéristiques que la précédente. Les commandes `COMMIT AND CHAIN` et `ROLLBACK AND CHAIN` font cela.

Dans les procédures appelées par la commande CALL ainsi que dans les blocs de code anonymes (commande DO), il est possible de terminer les transactions en utilisant les commandes COMMIT et ROLLBACK. Une nouvelle transaction est démarrée automatiquement après qu'une transaction ait été terminée en utilisant ces commandes, donc il n'existe pas de commande START TRANSACTION. (Notez que BEGIN et END ont une signification différente dans PL/pgSQL.)

Voici un exemple simple :

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plpgsql
AS $$
BEGIN
    FOR i IN 0..9 LOOP
        INSERT INTO test1 (a) VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
END
$$;

CALL transaction_test1();
```

Le contrôle des transactions est seulement possible dans les appels à CALL et DO à partir du plus haut niveau ou dans les appels imbriqués à CALL ou DO sans autre commande. Par exemple, si la pile d'appel est CALL proc1() → CALL proc2() → CALL proc3(), alors la deuxième et la troisième procédures peuvent exécuter les actions de contrôle de transaction. Mais si la pile d'appel est CALL proc1() → SELECT func2() → CALL proc3(), alors la dernière procédure ne peut pas faire de contrôle de transactions à cause du SELECT.

Des considérations spéciales s'appliquent aux boucles de curseur. Considérez cet exemple :

```
CREATE PROCEDURE transaction_test2()
LANGUAGE plpgsql
AS $$
DECLARE
    r RECORD;
BEGIN
    FOR r IN SELECT * FROM test2 ORDER BY x LOOP
        INSERT INTO test1 (a) VALUES (r.x);
        COMMIT;
    END LOOP;
END;
$$;

CALL transaction_test2();
```

Habituellement, les curseurs sont automatiquement fermés au moment de la validation de la transaction. Néanmoins, un curseur créé dans une boucle comme celle-ci est automatiquement converti en un curseur maintenable par le premier COMMIT ou ROLLBACK. Ceci signifie que le curseur est complètement évalué au premier COMMIT ou ROLLBACK plutôt que ligne par ligne. Le curseur est toujours automatiquement supprimé après la boucle, donc c'est pratiquement invisible pour l'utilisateur.

Les commandes de transaction ne sont pas autorisées dans les boucles de curseur exécutés par des commandes qui ne sont pas en lecture seule (par exemple `UPDATE ... RETURNING`).

Une transaction ne peut pas être terminée dans un bloc contenant un gestionnaire d'exceptions.

43.9. Erreurs et messages

43.9.1. Rapporter des erreurs et messages

Utilisez l'instruction `RAISE` pour rapporter des messages et lever des erreurs.

```
RAISE [ niveau ] 'format' [, expression [, ...]] [ USING option
 = expression [, ... ] ];
RAISE [ niveau ] nom_condition [ USING option = expression
 [, ... ] ];
RAISE [ niveau ] SQLSTATE 'état_sql' [ USING option = expression
 [, ... ] ];
RAISE [ niveau ] USING option = expression [, ... ];
RAISE ;
```

L'option *`niveau`* indique la sévérité de l'erreur. Les niveaux autorisés sont `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING` et `EXCEPTION`, ce dernier étant la valeur par défaut. `EXCEPTION` lève une erreur (ce qui annule habituellement la transaction en cours). Les autres niveaux ne font que générer des messages aux différents niveaux de priorité. Les variables de configuration log_min_messages et client_min_messages contrôlent l'envoi de messages dans les traces, au client ou aux deux. Voir le Chapitre 20 pour plus d'informations.

Après *`niveau`*, vous pouvez écrire un *`format`* (qui doit être une chaîne litérale, pas une expression). La chaîne format indique le texte du message d'erreur à rapporter. Elle peut être suivie par des expressions optionnelles à insérer dans le message. Dans la chaîne, % est remplacé par la représentation de la valeur du prochain argument. Écrivez %% pour saisir un % litéral. Le nombre des arguments doit correspondre au nombre de % dans la chaîne format, sinon une erreur est levée durant la compilation de la fonction.

Dans cet exemple, la valeur de `v_job_id` remplace le % dans la chaîne.

```
RAISE NOTICE 'Appel de cs_creer_job(%)', v_job_id;
```

Vous pouvez attacher des informations supplémentaires au rapport d'erreur en écrivant `USING` suivi par des éléments *`option`* = *`expression`*. Chaque *`expression`* peut valoir n'importe quel expression sous forme de chaîne. Les mots clés autorisés *`option`* sont :

`MESSAGE`

Configure le texte du message d'erreur. Cette option ne peut pas être utilisée dans la forme d'un `RAISE` qui inclut une chaîne de format avec `USING`.

`DETAIL`

Fournit un message de détail sur l'erreur.

`HINT`

Fournit un message de conseil sur l'erreur.

`ERRCODE`

Spécifie le code d'erreur (SQLSTATE) à rapporter, soit par son nom de condition comme indiqué dans Annexe A, soit directement sous la forme d'un code SQLSTATE sur cinq caractères.

```
COLUMN
CONSTRAINT
DATATYPE
TABLE
SCHEMA
```

Fournit le nom de l'objet.

Cet exemple annulera la transaction avec le message d'erreur et l'astuce donnés :

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
   USING HINT = 'Please check your user id';
```

Ces deux exemples affichent des façons équivalentes pour initialiser SQLSTATE :

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE =
 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

Il existe une deuxième syntaxe `RAISE` pour laquelle l'argument principale est le nom de la condition ou le SQLSTATE à rapporter, par exemple :

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

Dans cette syntaxe, `USING` peut être utilisé pour fournir un message d'erreur, un détail ou une astuce personnalisée. Voici une autre façon de faire l'exemple précédent :

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' ||
 user_id;
```

Une autre variante est d'écrire `RAISE USING` ou `RAISE` *`niveau`* `USING` et de placer tout le reste dans la liste `USING`.

La dernière variante de `RAISE` n'a aucun paramètre. Cette forme peut seulement être utilisée dans un bloc `BEGIN` d'une clause `EXCEPTION` ; cela fait que l'erreur est renvoyée.

> **Note**
>
> Avant PostgreSQL 9.1, `RAISE` sans paramètres était interprété comme un renvoi de l'erreur à partir du bloc contenant le gestionnaire actif d'exceptions. Du coup, une clause `EXCEPTION` imbriquée dans ce gestionnaire ne la récupérerait pas, même si le `RAISE` était intégrée dans le bloc de la clause `EXCEPTION`. C'était très surprenant et incompatible avec PL/SQL d'Oracle.

Si aucun nom de condition ou SQLSTATE n'est indiqué dans une commande `RAISE EXCEPTION`, la valeur par défaut est d'utiliser `raise_exception` (`P0001`). Si aucun message texte n'est indiqué, la valeur par défaut est d'utiliser le nom de la condition ou le SQLSTATE comme texte de message.

> **Note**
>
> Lors de la spécification du code d'erreur par un code SQLSTATE, vous n'êtes pas limité aux codes d'erreur prédéfinis, mais pouvez sélectionner tout code d'erreur consistant en cinq

chiffres et/ou des lettres ASCII majuscules, autre que `00000`. Il est recommandé d'éviter d'envoyer des codes d'erreur qui se terminent avec trois zéros car il y a des codes de catégorie, et peuvent seulement être récupérés en filtrant la catégorie complète.

43.9.2. Vérification d'assertions

L'instruction `ASSERT` est un moyen pratique d'insérer dans les fonctions PL/pgSQL des vérifications d'assertions.

```
ASSERT condition [ , message ];
```

La *`condition`* est une expression booléenne qui est censée être toujours vraie. Si c'est le cas, l'instruction `ASSERT` ne fait rien. Si le résultat est faux ou NULL, alors une exception `ASSERT_FAILURE` est levée (si une erreur survient lors de l'évaluation de la *`condition`*, elle est rapportée normalement).

Si le *`message`* optionnel est fourni, cela doit être une expression dont le résultat (si non NULL) remplacera le message d'erreur (par défaut « assertion failed ») si la *`condition`* est fausse. L'expression *`message`* n'est pas évaluée dans le cas normal où l'assertion est vraie.

La vérification des assertions peut être activée ou désactivée via le paramètre de configuration `plpgsql.check_asserts` qui prend une valeur booléenne, par défaut à `on`. Si ce paramètre est à `off` alors l'instruction `ASSERT` ne fait rien.

Notez que l'instruction `ASSERT` sert à détecter des erreurs de programmation, pas à rapporter des erreurs ordinaires. Pour cela, veuillez utiliser l'instruction `RAISE` décrite ci-dessus.

43.10. Fonctions trigger

PL/pgSQL peut être utilisé pour définir des fonctions trigger sur les modifications de données ou sur les événements en base. Une fonction trigger est créée avec la commande, en la déclarant comme une fonction sans argument et avec un type en retour `trigger` (pour les triggers sur les modifications de données) ou `event_trigger` (pour les triggers sur les événements en base). Des variables locales spéciales, nommées `TG_`*`quelquechose`* sont automatiquement définies pour décrire la condition qui a déclenché l'appel.

43.10.1. Triggers sur les modifications de données

Un trigger sur modification de données est déclaré comme une fonction sans arguments et renvoyant le type `trigger`. Notez que la fonction doit être déclarée sans arguments même si elle s'attend à recevoir des arguments spécifiés dans `CREATE TRIGGER` -- ce type d'argument est passé via `TG_ARGV`, comme indiqué ci-dessous.

Quand une fonction PL/pgSQL est appelée en tant que trigger, plusieurs variables spéciales sont créées automatiquement dans le bloc de plus haut niveau. Ce sont :

`NEW`

Type de données `RECORD` ; variable contenant la nouvelle ligne de base de données pour les opérations `INSERT` / `UPDATE` dans les triggers de niveau ligne. Cette variable est NULL dans un trigger de niveau instruction et pour les opérations `DELETE`.

`OLD`

Type de données `RECORD` ; variable contenant l'ancienne ligne de base de données pour les opérations `UPDATE`/`DELETE` dans les triggers de niveau ligne. Cette variable est NULL dans les triggers de niveau instruction et pour les opérations `INSERT`.

`TG_NAME`

Type de données `name` ; variable qui contient le nom du trigger réellement lancé.

`TG_WHEN`

Type de données `text` ; une chaîne, soit `BEFORE` soit `AFTER`, soit `INSTEAD OF` selon la définition du trigger.

`TG_LEVEL`

Type de données `text` ; une chaîne, soit `ROW` soit `STATEMENT`, selon la définition du trigger.

`TG_OP`

Type de données `text` ; une chaîne, `INSERT`, `UPDATE`, `DELETE` ou `TRUNCATE` indiquant pour quelle opération le trigger a été lancé.

`TG_RELID`

Type de données `oid` ; l'ID de l'objet de la table qui a causé le déclenchement du trigger.

`TG_RELNAME`

Type de données `name` ; le nom de la table qui a causé le déclenchement. C'est obsolète et pourrait disparaître dans une prochaine version. À la place, utilisez `TG_TABLE_NAME`.

`TG_TABLE_NAME`

Type de données `name` ; le nom de la table qui a déclenché le trigger.

`TG_TABLE_SCHEMA`

Type de données `name` ; le nom du schéma de la table qui a appelé le trigger.

`TG_NARGS`

Type de données `integer` ; le nombre d'arguments donnés à la fonction trigger dans l'instruction `CREATE TRIGGER`.

`TG_ARGV[]`

Type de donnée `text` ; les arguments de l'instruction `CREATE TRIGGER`. L'index débute à 0. Les indices invalides (inférieurs à 0 ou supérieurs ou égaux à `tg_nargs`) auront une valeur NULL.

Une fonction trigger doit renvoyer soit `NULL` soit une valeur record ayant exactement la structure de la table pour laquelle le trigger a été lancé.

Les triggers de niveau ligne lancés `BEFORE` peuvent renvoyer NULL pour indiquer au gestionnaire de trigger de sauter le reste de l'opération pour cette ligne (les triggers suivants ne sont pas lancés, et les `INSERT`/`UPDATE`/`DELETE` ne se font pas pour cette ligne). Si une valeur non NULL est renvoyée alors l'opération se déroule avec cette valeur ligne. Renvoyer une valeur ligne différente de la valeur originale de `NEW` modifie la ligne qui sera insérée ou mise à jour. De ce fait, si la fonction de trigger veut que l'action réussisse sans modifier la valeur de rangée, `NEW` (ou une valeur égale) doit être renvoyée. Pour modifier la rangée à être stockée, il est possible de remplacer les valeurs directement dans `NEW` et renvoyer le `NEW` modifié ou de générer un nouvel enregistrement à renvoyer. Dans le cas d'un before-trigger sur une commande `DELETE`, la valeur renvoyée n'a aucun effet direct mais doit être non-nulle pour permettre à l'action trigger de continuer. Notez que `NEW` est nul dans le cadre des triggers `DELETE` et que renvoyer ceci n'est pas recommandé dans les cas courants. Une pratique utile dans des triggers `DELETE` serait de renvoyer `OLD`.

Les triggers INSTEAD OF (qui sont toujours des triggers au niveau ligne et peuvent seulement être utilisés sur des vues) peuvent renvoyer NULL pour signaler qu'ils n'ont fait aucune modification et que le reste de l'opération pour cette ligne doit être ignoré (autrement dit, les triggers suivants ne sont pas déclenchés et la ligne n'est pas comptée dans le statut des lignes affectées pour la requête INSERT/UPDATE/DELETE). Une valeur différente de NULL doit être renvoyée pour indiquer que le trigger a traité l'opération demandée. Pour les opérations INSERT et UPDATE, la valeur de retour doit être NEW, que la fonction trigger peut modifier pour supporter une clause RETURNING d'une requête INSERT ou UPDATE (ceci affectera aussi la valeur de ligne passée aux triggers suivants ou passée à l'alias spécial EXCLUDED dans une instruction INSERT dotée d'une clause ON CONFLICT DO UPDATE). Pour les opérations DELETE, la valeur de retour doit être OLD.

La valeur de retour d'un trigger de niveau rangée déclenché AFTER ou un trigger de niveau instruction déclenché BEFORE ou AFTER est toujours ignoré ; il pourrait aussi bien être NULL. Néanmoins, tous les types de triggers peuvent toujours annuler l'opération complète en envoyant une erreur.

L'Exemple 43.3 montre un exemple d'une fonction trigger dans PL/pgSQL.

Exemple 43.3. Une fonction trigger PL/pgSQL

Cet exemple de trigger assure qu'à chaque moment où une ligne est insérée ou mise à jour dans la table, le nom de l'utilisateur courant et l'heure sont estampillés dans la ligne. Et cela vous assure qu'un nom d'employé est donné et que le salaire est une valeur positive.

```
CREATE TABLE emp (
    nom_employe text,
    salaire integer,
    date_dermodif timestamp,
    utilisateur_dermodif text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Verifie que nom_employe et salary sont donnés
        IF NEW.nom_employe IS NULL THEN
            RAISE EXCEPTION 'nom_employe ne peut pas être NULL';
        END IF;
        IF NEW.salaire IS NULL THEN
            RAISE EXCEPTION '% ne peut pas avoir un salaire',
 NEW.nom_employe;
        END IF;

        -- Qui travaille pour nous si la personne doit payer pour
 cela ?
        IF NEW.salaire < 0 THEN
            RAISE EXCEPTION '% ne peut pas avoir un salaire
 négatif', NEW.nom_employe;
        END IF;

        -- Rappelons-nous qui a changé le salaire et quand
        NEW.date_dermodif := current_timestamp;
        NEW.utilisateur_dermodif := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE FUNCTION emp_stamp();
```

Une autre façon de tracer les modifications sur une table implique la création d'une nouvelle table qui contient une ligne pour chaque insertion, mise à jour ou suppression qui survient. Cette approche peut être vue comme un audit des modifications sur une table. L'Exemple 43.4 montre un exemple d'une fonction d'audit par trigger en PL/pgSQL.

Exemple 43.4. Une fonction d'audit par trigger en PL/pgSQL

Cet exemple de trigger nous assure que toute insertion, modification ou suppression d'une ligne dans la table `emp` est enregistrée dans la table `emp_audit`. L'heure et le nom de l'utilisateur sont conservées dans la ligne avec le type d'opération réalisé.

```
CREATE TABLE emp (
    nom_employe       text NOT NULL,
    salaire           integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    tampon            timestamp NOT NULL,
    id_utilisateur    text      NOT NULL,
    nom_employe       text      NOT NULL,
    salaire           integer
);

CREATE OR REPLACE FUNCTION audit_employe() RETURNS TRIGGER AS
 $emp_audit$
BEGIN
    --
    -- Ajoute une ligne dans emp_audit pour refléter l'opération
 réalisée
    -- sur emp,
    -- utilise la variable spéciale TG_OP pour cette opération.
    --
    IF (TG_OP = 'DELETE') THEN
        INSERT INTO emp_audit SELECT 'D', now(), current_user,
 OLD.*;
    ELSIF (TG_OP = 'UPDATE') THEN
        INSERT INTO emp_audit SELECT 'U', now(), current_user,
 NEW.*;
    ELSIF (TG_OP = 'INSERT') THEN
        INSERT INTO emp_audit SELECT 'I', now(), current_user,
 NEW.*;
    END IF;
    RETURN NULL; -- le résultat est ignoré car il s'agit d'un
 trigger AFTER
END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE FUNCTION audit_employe();
```

Une variation de l'exemple précédent utilise une vue joignant la table principale et la table d'audit pour montrer les derniers enregistrements modifiés. Cette approche enregistre toujours toutes les modifications sur la table mais présente aussi une vue simple de l'audit, n'affichant que le date et heure de la dernière modification pour chaque enregistrement. Exemple 43.5 montre un exemple d'un trigger d'audit sur une vue avec PL/pgSQL.

Exemple 43.5. Une fonction trigger en PL/pgSQL sur une vue pour un audit

Cet exemple utilise un trigger sur une vue pour la rendre modifiable, et s'assure que toute insertion, mise à jour ou suppression d'une ligne dans la vue est enregistrée (pour l'audit) dans la table `emp_audit`. La date et l'heure courante ainsi que le nom de l'utilisateur sont enregistrés, avec le type d'opération réalisé pour que la vue affiche la date et l'heure de la dernière modification de chaque ligne.

```
CREATE TABLE emp (
    nom_employe       text PRIMARY KEY,
    salaire           integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    id_utilisateur    text      NOT NULL,
    nom_employe       text      NOT NULL,
    salaire           integer,
    dmodif            timestamp NOT NULL
);

CREATE VIEW emp_vue AS
    SELECT e.nom_employe,
           e.salaire,
           max(ea.dmodif) AS derniere_modification
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.nom_employe = e.nom_employe
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION miseajour_emp_vue() RETURNS TRIGGER AS $
$
    BEGIN
        --
        -- Perform the required operation on emp, and create a row
 in emp_audit
        -- to reflect the change made to emp.
        --
        IF (TG_OP = 'DELETE') THEN
            DELETE FROM emp WHERE nom_employe = OLD.nom_employe;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.derniere_modification = now();
            INSERT INTO emp_audit VALUES('D', current_user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE nom_employe =
 OLD.nom_employe;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.derniere_modification = now();
            INSERT INTO emp_audit VALUES('U', current_user, NEW.*);
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.nom_employe, NEW.salaire);

            NEW.derniere_modification = now();
            INSERT INTO emp_audit VALUES('I', current_user, NEW.*);
            RETURN NEW;
        END IF;
```

```
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_vue
    FOR EACH ROW EXECUTE FUNCTION miseajour_emp_vue();
```

Une utilisation des triggers est le maintien d'une table résumée d'une autre table. Le résumé résultant peut être utilisé à la place de la table originale pour certaines requêtes -- souvent avec des temps d'exécution bien réduits. Cette technique est souvent utilisée pour les statistiques de données où les tables de données mesurées ou observées (appelées des tables de faits) peuvent être extrêmement grandes. L'Exemple 43.6 montre un exemple d'une fonction trigger en PL/pgSQL maintenant une table résumée pour une table de faits dans un système de données (data warehouse).

Exemple 43.6. Une fonction trigger PL/pgSQL pour maintenir une table résumée

Le schéma détaillé ici est partiellement basé sur l'exemple du *Grocery Store* provenant de *The Data Warehouse Toolkit* par Ralph Kimball.

```
--
-- Tables principales - dimension du temps de ventes.
--
CREATE TABLE time_dimension (
  time_key                    integer NOT NULL,
  day_of_week                 integer NOT NULL,
  day_of_month                integer NOT NULL,
  month                       integer NOT NULL,
  quarter                     integer NOT NULL,
  year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
  time_key                    integer NOT NULL,
  product_key                 integer NOT NULL,
  store_key                   integer NOT NULL,
  amount_sold                 numeric(12,2) NOT NULL,
  units_sold                  integer NOT NULL,
  amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Table résumé - ventes sur le temps.
--
CREATE TABLE sales_summary_bytime (
  time_key                    integer NOT NULL,
  amount_sold                 numeric(15,2) NOT NULL,
  units_sold                  numeric(12) NOT NULL,
  amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON
 sales_summary_bytime(time_key);

--
-- Fonction et trigger pour amender les colonnes résumées
```

```
-- pour un UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS
 TRIGGER
AS $maint_sales_summary_bytime$
DECLARE
  delta_time_key          integer;
  delta_amount_sold       numeric(15,2);
  delta_units_sold        numeric(12);
  delta_amount_cost       numeric(15,2);
BEGIN

  -- Travaille sur l'ajout/la suppression de montant(s).
  IF (TG_OP = 'DELETE') THEN

    delta_time_key = OLD.time_key;
    delta_amount_sold = -1 * OLD.amount_sold;
    delta_units_sold = -1 * OLD.units_sold;
    delta_amount_cost = -1 * OLD.amount_cost;

  ELSIF (TG_OP = 'UPDATE') THEN

    -- interdit les mises à jour qui modifient time_key -
    -- (probablement pas trop cher, car DELETE + INSERT est la
 façon la plus
    -- probable de réaliser les modifications).
    IF ( OLD.time_key != NEW.time_key) THEN
      RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                      OLD.time_key, NEW.time_key;
    END IF;

    delta_time_key = OLD.time_key;
    delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
    delta_units_sold = NEW.units_sold - OLD.units_sold;
    delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

  ELSIF (TG_OP = 'INSERT') THEN

    delta_time_key = NEW.time_key;
    delta_amount_sold = NEW.amount_sold;
    delta_units_sold = NEW.units_sold;
    delta_amount_cost = NEW.amount_cost;

  END IF;


  -- Insertion ou mise à jour de la ligne de résumé avec les
 nouvelles valeurs.
  <<insert_update>>
  LOOP
  UPDATE sales_summary_bytime
  SET amount_sold = amount_sold + delta_amount_sold,
    units_sold = units_sold + delta_units_sold,
    amount_cost = amount_cost + delta_amount_cost
    WHERE time_key = delta_time_key;

    EXIT insert_update WHEN found;
```

```
    BEGIN
      INSERT INTO sales_summary_bytime (
        time_key,
        amount_sold,
        units_sold,
        amount_cost)
        VALUES (
        delta_time_key,
        delta_amount_sold,
        delta_units_sold,
        delta_amount_cost
        );
      EXIT insert_update;

      EXCEPTION
      WHEN UNIQUE_VIOLATION THEN
      -- do nothing
      END;
      END LOOP insert_update;

  RETURN NULL;

END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
  AFTER INSERT OR UPDATE OR DELETE ON sales_fact
  FOR EACH ROW EXECUTE FUNCTION maint_sales_summary_bytime();

INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

Les triggers `AFTER` peuvent aussi utiliser les *tables de transition* pour inspecter l'ensemble complet de lignes modifiées par l'instruction trigger. La commande `CREATE TRIGGER` donne des noms à la ou aux tables de transition. La fonction peut ensuite se référer à ces noms comme s'il s'agissait de tables temporaires en lecture seule. Exemple 43.7 montre un exemple.

Exemple 43.7. Auditer avec les tables de transition

Cet exemple produit les mêmes résultats que Exemple 43.4 mais au lieu d'utiliser un trigger qui se déclenche pour chaque ligne, il utilise un trigger qui se déclenche une fois par instruction, après avoir récupérer les informations intéressantes dans une table de transition. Cela peut être significativement plus rapide que l'approche du trigger par ligne lorsque l'instruction trigger a modifié beaucoup de lignes. Notez qu'il est nécessaire de faire une déclaration séparée du trigger pour chaque type d'événement car les clauses `REFERENCING` doivent être différentes dans chaque cas. Mais ceci ne nous empêche pas d'utiliser une seule fonction trigger si nous le souhaitons. (En pratique, il pourrait être préférable d'utiliser trois fonctions séparées et d'éviter les tests à l'exécution sur la variable `TG_OP`.)

```
CREATE TABLE emp (
```

```
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS
 $emp_audit$
    BEGIN
        --
        -- Create rows in emp_audit to reflect the operations
 performed on emp,
        -- making use of the special variable TG_OP to work out the
 operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit
                SELECT 'D', now(), current_user, o.* FROM old_table
 o;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit
                SELECT 'U', now(), current_user, n.* FROM new_table
 n;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit
                SELECT 'I', now(), current_user, n.* FROM new_table
 n;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER
 trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit_ins
    AFTER INSERT ON emp
    REFERENCING NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_upd
    AFTER UPDATE ON emp
    REFERENCING OLD TABLE AS old_table NEW TABLE AS new_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
CREATE TRIGGER emp_audit_del
    AFTER DELETE ON emp
    REFERENCING OLD TABLE AS old_table
    FOR EACH STATEMENT EXECUTE FUNCTION process_emp_audit();
```

43.10.2. Triggers sur des événements

PL/pgSQL peut être utilisé pour définir des triggers sur des événements. PostgreSQL requiert qu'une fonction qui doit appelée en tant que trigger d'événement soit déclarée sans argument et avec un type `event_trigger` en retour.

Quand une fonction PL/pgSQL est appelée en tant que trigger d'événement, plusieurs variables spéciales sont créées automatiquement dans son bloc de niveau haut. Les voici :

`TG_EVENT`

Type de données `text` ; une chaîne représentant l'événement pour lequel le trigger est déclenché.

`TG_TAG`

Type de données `text` ; variable contenant la balise commande pour laquelle le trigger a été déclenché.

Exemple 43.8 montre un exemple d'une fonction pour un trigger d'événement écrit en PL/pgSQL.

Exemple 43.8. Une fonction PL/pgSQL pour un trigger d'événement

Cet exemple de trigger lève simplement un message `NOTICE` à chaque fois qu'une commande supportée est exécutée.

```
CREATE OR REPLACE FUNCTION rapporte() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'rapporte: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER rapporte ON ddl_command_start EXECUTE FUNCTION
 rapporte();
```

43.11. Les dessous de PL/pgSQL

Cette section discute des détails d'implémentation les plus importants à connaître pour les utilisateurs de PL/pgSQL.

43.11.1. Substitution de variables

Les instructions et expressions SQL au sein d'une fonction PL/pgSQL peuvent faire appel aux variables et paramètres d'une fonction. En coulisses, PL/pgSQL remplace les paramètres de requêtes par des références. Les paramètres de la requête ne seront remplacés qu'aux endroits où un paramètre ou une référence de colonne sont autorisés par la syntaxe. Dans un cas extrême, considérez cet exemple de mauvaise programmation :

```
INSERT INTO foo (foo) VALUES (foo
  (foo));
```

La première occurrence de `foo` doit être un nom de table, d'après la syntaxe et ne sera donc pas remplacée, même si la fonction a une variable nommée `foo`. La deuxième occurrence doit être le nom d'une colonne de cette table et ne sera donc pas remplacée non plus. De la même façon, la troisième occurrence doit être un nom de fonction, donc il ne sera pas non plus substitué. Seule la dernière occurrence peut être une référence à une variable de la fonction PL/pgSQL.

Une autre façon de comprendre cela est que la substitution de variable peut seulement insérer des valeurs de type données dans une commande SQL ; celane peut pas modifier dynamiquement les objets de la base référencés par la commande. (Si ous voulez le faire, vous devez construire dynamiquement une chaîne de caractère contenant les commandes, comme expliqué dans Section 43.5.4.)

Les noms de variables n'étant pas différents des noms de colonnes, d'après la syntaxe, il peut y avoir ambigüité dans les instructions qui font référence aux deux : un nom donné fait-il référence à un nom de colonne ou à une variable ? Modifions l'exemple précédent.

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

Ici, `dest` et `src` doivent être des noms de table et `col` doit être une colonne de `dest` mais `foo` et `bar` peuvent être aussi bien des variables de la fonction que des colonnes de `src`.

Par défaut, PL/pgSQL signalera une erreur si un nom dans une requête SQL peut faire référence à la fois à une variable et à une colonne. Vous pouvez corriger ce problème en renommant la variable ou colonne, en qualifiant la référence ambigüe ou en précisant à PL/pgSQL quelle est l'interprétation à privilégier.

Le choix le plus simple est de renommer la variable ou colonne. Une règle de codage récurrente est d'utiliser une convention de nommage différente pour les variables de PL/pgSQL que pour les noms de colonne. Par exemple, si vous utilisez toujours des variables de fonctions en `v_`*`quelquechose`* tout en vous assurant qu'aucun nom de colonne ne commence par `v_`, aucun conflit ne sera possible.

Autrement, vous pouvez qualifier les références ambigües pour les rendre plus claires. Dans l'exemple ci-dessus, `src.foo` serait une référence sans ambigüité à une colonne de table. Pour créer une référence sans ambigüité à une variable, déclarez-la dans un bloc nommé et utilisez le nom du bloc (voir Section 43.2). Par exemple,

```
<<bloc>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT bloc.foo + bar FROM src;
```

Ici, `bloc.foo` désigne la variable même s'il existe une colonne `foo` dans la base `src`. Les paramètres de fonction, ainsi que les variables spéciales tel que `FOUND`, peuvent être qualifiés par le nom de la fonction, parce qu'ils sont implicitement déclarés dans un bloc extérieur portant le nom de la fonction.

Quelque fois, il n'est pas envisageable de lever toutes les ambigüités dans une grande quantité de code PL/pgSQL. Dans ces cas-ci, vous pouvez spécifier à PL/pgSQL qu'il doit traiter les références ambigües comme étant une variable (ce qui est compatible avec le comportement de PL/pgSQL avant PostgreSQL 9.0) ou comme étant la colonne d'une table (ce qui est compatible avec d'autres systèmes tels que Oracle).

Pour modifier ce comportement dans toute l'instance, mettez le paramètre de configuration `plpgsql.variable_conflict` à l'un de `error`, `use_variable` ou `use_column` (où `error` est la valeur par défaut). Ce paramètre agit sur les compilations posterieures d'instructions dans les fonctions PL/pgSQL mais pas les instructions déjà compilées dans la session en cours. Cette modification pouvant affecter de manière inattendue le comportement des fonctions PL/pgSQL, elle ne peut être faite que par un administrateur.

Vous pouvez modifier ce comportement fonction par fonction, en insérant l'une de ces commandes spéciales au début de la fonction :

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

Ces commandes n'agissent que sur les fonctions qui les contient et surchargent la valeur de `plpgsql.variable_conflict`. Un exemple est

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
          WHERE users.id = id;
    END;
$$ LANGUAGE plpgsql;
```

Dans la commande `UPDATE`, `curtime`, `comment`, et `id` font référence aux variables et paramètres de la fonction, que la table `users` ait ou non des colonnes portant ces noms. Notez qu'il a fallu qualifier la référence à `users.id` dans la clause `WHERE` pour qu'elle fasse référence à la colonne. Mais nous ne qualifions pas la référence à `comment` comme cible dans la liste `UPDATE` car, d'après la syntaxe, elle doit être une colonne de `users`. Nous pourrions écrire la même fonction sans dépendre de la valeur de `variable_conflict` de cette manière :

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment =
 stamp_user.comment
          WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

La substitution de variable n'arrive pas dans la chaîne de commande donnée à `EXECUTE` ou une de ces variantes. Si vous avez besoin d'insérer une valeur dans une telle commande, faites-le lors de la construction d'une valeur de chaîne, illustrée dans la Section 43.5.4, ou utilisez `USING`.

La substitution de variable fonctionne seulement dans les commandes `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE` et les commandes contenant l'une de ces commandes (telles que `EXPLAIN` et `CREATE TABLE ... AS SELECT`) parce que le moteur SQL principal autorise les paramètres de la requête seulement dans ces commandes. Pour utiliser un nom variable ou une valeur dans les autres types d'instructions (généralement appelées des instructions utilitaires), vous devez construire l'instruction en question comme une chaîne et l'exécuter via `EXECUTE`.

43.11.2. Mise en cache du plan

L'interpréteur PL/pgSQL analyse le source d'une fonction et produit un arbre binaire interne d'instructions la première fois que la fonction est appelée (à l'intérieur de chaque session). L'arbre des instructions se traduit complètement par la structure d'instructions PL/pgSQL mais les expressions et les commandes SQL individuelles utilisées dans la fonction ne sont pas traduites immédiatement.

Au moment où chaque expression et commande SQL est exécutée en premier lieu dans la fonction, l'interpréteur PL/pgSQL lit et analyse la commande pour créer une instruction préparée en utilisant la fonction `SPI_prepare` du gestionnaire SPI. Les appels suivants à cette expression ou commande réutilisent le plan préparé. Donc, une fonction avec des chemins de code conditionnel peu fréquemment exécutés n'auront jamais la surcharge de l'analyse de ces commandes qui ne sont jamais exécutées à l'intérieur de la session en cours. Un inconvénient est que les erreurs dans une expression ou commande spécifique ne peuvent pas être détectées avant que la fonction a atteint son exécution. (Les erreurs de syntaxe triviales seront détectées à la première passe d'analyse mais quelque chose de plus complexe ne sera pas détecté avant son exécution.)

PL/pgSQL (ou plus exactement le gestionnaire SPI) peut tenter de mettre en cache le plan d'exécution associé à toute requête préparée. Si un plan en cache n'est pas utilisé, alors un nouveau plan d'exécution est généré pour chaque appel de la requête, et les valeurs actuelles du paramètre (autrement dit les valeurs de la variable PL/pgSQL) peuvent être utilisées pour optimiser le plan sélectionné. Si la requête n'a pas de paramètres ou est exécuté plusieurs fois, le gestionnaire SPI considérera la création d'un plan *générique* qui n'est pas dépendant des valeurs du paramètre et placera ce plan en cache pour le réutiliser. Habituellement, ceci survient seulement si le plan d'exécution n'est pas très sensible aux valeurs des variables PL/pgSQL référencées. Si ce n'est pas le cas, générer un nouveau plan à chaque fois est un gain net. Voir PREPARE pour plus d'informations sur le comportement des requêtes préparées.

Comme PL/pgSQL sauvegarde des instructions préparées et quelques fois des plans d'exécution de cette façon, les commandes SQL qui apparaissent directement dans une fonction PL/pgSQL doivent faire référence aux même tables et aux mêmes colonnes à chaque exécution ; c'est-à-dire que vous ne pouvez pas utiliser un paramètre comme le nom d'une table ou d'une colonne dans une commande SQL. Pour contourner cette restriction, vous pouvez construire des commandes dynamiques en utilisant l'instruction `EXECUTE` de PL/pgSQL -- au prix d'une nouvelle analyse du plan et de la construction d'un nouveau plan d'exécution sur chaque exécution.

La nature muable des variables de type record présente un autre problème dans cette connexion. Quand les champs d'une variable record sont utilisés dans les expressions ou instructions, les types de données des champs ne doivent pas modifier d'un appel de la fonction à un autre car chaque expression sera analysée en utilisant le type de données qui est présent quand l'expression est atteinte en premier. `EXECUTE` peut être utilisé pour contourner ce problème si nécessaire.

Si la même fonction est utilisée comme trigger pour plus d'une table, PL/pgSQL prépare et met en cache les instructions indépendamment pour chacune de ses tables -- c'est-à-dire qu'il y a un cache pour chaque combinaison fonction trigger/table, pas uniquement pour chaque fonction. Ceci diminue certains des problèmes avec les types de données variables ; par exemple, une fonction trigger pourra fonctionner correctement avec une colonne nommée `cle` même si cette colonne a différents types dans différentes tables.

De la même façon, les fonctions ayant des types polymorphiques pour les arguments ont un cache séparé des instructions pour chaque combinaison des types d'argument réels avec lesquels elles ont été appelées, donc les différences de type de données ne causent pas d'échecs inattendus.

La mise en cache des instructions peut parfois avoir des effets surprenants sur l'interprétation des valeurs sensibles à l'heure. Par exemple, il y a une différence entre ce que font ces deux fonctions :

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
    END;
$$ LANGUAGE plpgsql;
```

et :

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

Dans le cas de `logfunc1`, l'analyseur principal de PostgreSQL sait lors de l'analyser du `INSERT` que la chaîne `'now'` devrait être interprétée comme un `timestamp` car la colonne cible de `logtable` est de ce type. Du coup, `'now'` sera converti en une constante `timestamp` quand `INSERT` est analysé, puis utilisé dans tous les appels de `logfunc1` tout au long de la vie de la session. Il est inutile de dire que ce n'est pas ce que voulait le développeur. Une meilleure idée reviendrait à utiliser la fonction `now()` ou `current_timestamp`.

Dans le cas de `logfunc2`, l'analyseur principal de PostgreSQL ne connaît pas le type que deviendra `'now'` et, du coup, il renvoie une valeur de type `text` contenant la chaîne `now`. Lors de l'affectation à la variable `curtime` locale, l'interpréteur PL/pgSQL convertie cette chaîne dans le type `timestamp` en appelant les fonctions `textout` et `timestamp_in` pour la conversion. Du coup, l'heure calculée est mise à jour à chaque exécution comme le suppose le développeur. Même s'il arrive que ça fonctionne ainsi, ce n'est pas très efficace, donc l'utilisation de la fonction `now()` sera encore une fois une meilleur idée.

43.12. Astuces pour développer en PL/pgSQL

Un bon moyen de développer en PL/pgSQL est d'utiliser l'éditeur de texte de votre choix pour créer vos fonctions, et d'utiliser psql dans une autre fenêtre pour charger et tester ces fonctions. Si vous procédez ainsi, une bonne idée est d'écrire la fonction en utilisant `CREATE OR REPLACE FUNCTION`. De cette façon vous pouvez simplement recharger le fichier pour mettre à jour la définition de la fonction. Par exemple :

```
CREATE OR REPLACE FUNCTION fonction_test(integer) RETURNS integer
 AS $$
   ....
$$ LANGUAGE plpgsql;
```

Pendant que psql s'exécute, vous pouvez charger ou recharger des définitions de fonction avec :

```
\i nom_fichier.sql
```

puis immédiatement soumettre des commandes SQL pour tester la fonction.

Un autre bon moyen de développer en PL/pgSQL est d'utiliser un outil d'accès à la base de données muni d'une interface graphique qui facilite le développement dans un langage de procédures. Un exemple d'un tel outil est pgAdmin, bien que d'autres existent. Ces outils fournissent souvent des fonctionnalités pratiques telles que la détection des guillemets ouverts et facilitent la re-création et le débogage des fonctions.

43.12.1. Utilisation des guillemets simples (quotes)

Le code d'une fonction PL/pgSQL est spécifié dans la commande `CREATE FUNCTION` comme une chaîne de caractères. Si vous écrivez la chaîne littérale de la façon ordinaire en l'entourant de guillemets simples, alors tout guillemet simple dans le corps de la fonction doit être doublé ; de la même façon, les antislashs doivent être doublés (en supposant que la syntaxe d'échappement de chaînes est utilisée). Doubler les guillemets devient rapidement difficile et, dans la plupart des cas compliqués, le code peut devenir rapidement incompréhensible parce que vous pouvez facilement vous trouver avec une douzaine, voire plus, de guillemets adjacents. À la place, il est recommandé d'écrire le corps de la fonction en tant qu'une chaîne littérale « avec guillemets dollar » (voir la Section 4.1.2.4). Dans cette approche, vous ne doublez jamais les marques de guillemets mais vous devez faire attention à choisir un délimiteur dollar différent pour chaque niveau d'imbrication dont vous avez besoin. Par exemple, vous pouvez écrire la commande `CREATE FUNCTION` en tant que :

```
CREATE OR REPLACE FUNCTION fonction_test(integer) RETURNS integer
 AS $PROC$
   ....
```

```
$PROC$ LANGUAGE plpgsql;
```

À l'intérieur de ceci, vous pouvez utiliser des guillemets pour les chaînes littérales simples dans les commandes SQL et $$ pour délimiter les fragments de commandes SQL que vous assemblez comme des chaînes. Si vous avez besoin de mettre entre guillemets du texte qui inclut $$, vous pouvez utiliser Q, et ainsi de suite.

Le graphe suivant montre ce que vous devez faire lors de l'écriture de guillemets simples sans guillemets dollar. Cela pourrait être utile lors de la traduction de code avec guillemets simples en quelque chose de plus compréhensible.

1 guillemet simple

Pour commencer et terminer le corps de la fonction, par exemple :

```
CREATE FUNCTION foo() RETURNS integer AS '
 .....
' LANGUAGE plpgsql;
```

Partout au sein du corps de la fonction entouré de guillemets simples, les guillemets simples *doivent* aller par paires.

2 guillemets simples

Pour les chaînes de caractères à l'intérieur du corps de la fonction, par exemple :

```
une_sortie := ''Blah'';
SELECT * FROM utilisateurs WHERE f_nom=''foobar'';
```

Dans l'approche du guillemet dollar, vous devriez juste écrire :

```
une_sortie := 'Blah';
SELECT * FROM utilisateurs WHERE f_nom='foobar';
```

ce qui serait exactement ce que l'analyseur PL/pgSQL verrait dans les deux cas.

4 guillemets simples

Quand vous avez besoin d'un guillemet simple dans une chaîne constante à l'intérieur du corps de la fonction, par exemple :

```
une_sortie := une_sortie || '' AND nom LIKE ''''foobar'''' AND
 xyz''
```

La valeur effectivement concaténée à `une_sortie` est : `AND nom LIKE 'foobar' AND xyz`.

Dans l'approche du guillemet dollar, vous auriez écrit :

```
une_sortie := une_sortie || $$ AND nom LIKE 'foobar' AND xyz$$
```

Faites attention que chaque délimiteur en guillemet dollar ne soient pas simplement $$.

6 guillemets simples

Quand un simple guillemet dans une chaîne à l'intérieur du corps d'une fonction est adjacent à la fin de cette chaîne constante, par exemple :

```
une_sortie := une_sortie || '' AND nom LIKE ''''foobar''''''
```

La valeur effectivement concaténée à `une_sortie` est alors : `AND nom LIKE 'foobar'`.

Dans l'approche guillemet dollar, ceci devient :

```
une_sortie := une_sortie || $$ AND nom LIKE 'foobar'$$
```

10 guillemets simples

Lorsque vous voulez deux guillemets simples dans une chaîne constante (qui compte pour huit guillemets simples) et qu'elle est adjacente à la fin de cette chaîne constante (deux de plus). Vous n'aurez probablement besoin de ceci que si vous écrivez une fonction qui génère d'autres fonctions comme dans l'Exemple 43.10. Par exemple :

```
une_sortie := une_sortie || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

La valeur de `une_sortie` sera alors :

```
if v_... like ''...'' then return ''...''; end if;
```

Dans l'approche du guillemet dollar, ceci devient :

```
une_sortie := une_sortie || $$ if v_$$ || referrer_keys.kind ||
 $$ like '$$
|| referrer_keys.key_string || $$'
then return '$$  || referrer_keys.referrer_type
|| $$'; end if;$$;
```

où nous supposons que nous avons seulement besoin de placer des marques de guillemets simples dans `une_sortie` parce que les guillemets seront recalculés avant utilisation.

43.12.2. Vérifications supplémentaires à la compilation et à l'exécution

Pour aider l'utilisateur à trouver les problèmes simples mais fréquents avant qu'ils ne posent de vrais problèmes, PL/PgSQL fournit des *`vérifications`* supplémentaires. Une fois activées, suivant la configuration, elles peuvent être utilisées pour émettre soit un `WARNING` soit un `ERROR` pendant la compilation d'une fonction. Une fonction qui a reçu un `WARNING` peut être exécutée sans produire d'autres messages, mais vous êtes averti de la tester dans un environnement de développement séparé.

Positionner `plpgsql.extra_warnings` ou `plpgsql.extra_errors`, selon les cas, à `"all"` est encouragé dans les environnements de développement et de test.

Ces vérifications supplémentaires sont activées via les variables de configuration `plpgsql.extra_warnings` pour les messages d'avertissement et `plpgsql.extra_errors` pour les erreurs. Les deux peuvent être configurés soit avec une liste de vérifications séparées par des virgules, soit pour aucune (`"none"`), soit pour toutes (`"all"`). La valeur par défaut est `"none"`. Actuellement la liste des vérifications possibles comprend :

`shadowed_variables`

Vérifie si une déclaration cache une variable définie précédemment.

`strict_multi_assignment`

Certaines commandes PL/PgSQL permettent d'affecter des valeurs à plus d'une variable à la fois, comme `SELECT INTO`. Typiquement, le nombre de variables cibles et le nombre de variables sources devraient correspondre, même si PL/PgSQL utilise `NULL` pour les valeurs manquantes et que les variables en excès sont ignorées. Le fait d'autoriser ce test fera que PL/PgSQL provoquera un avertissement (`WARNING`) ou une erreur (`ERROR`) si jamais les nombres de variables cibles et sources sont différents.

`too_many_rows`

L'activation de ce test fait que PL/PgSQL vérifie si une requête utilisée avec un clause `INTO` renvoie plus d'une ligne. Comme une instruction avec `INTO` ne prendra en compte qu'une seule ligne, le fait d'avoir une requête qui en retourne plusieurs est généralement inefficace et/ou non-déterministe et par conséquent est probablement une erreur.

L'exemple suivant montre l'effet de `plpgsql.extra_warnings` configuré à `shadowed_variables` :

```
SET plpgsql.extra_warnings TO 'shadowed_variables';

CREATE FUNCTION foo(f1 int) RETURNS int AS $$
DECLARE
f1 int;
BEGIN
RETURN f1;
END;
$$ LANGUAGE plpgsql;
WARNING:  variable "f1" shadows a previously defined variable
LINE 3: f1 int;
        ^
CREATE FUNCTION
```

L'exemple suivant montre l'effet de `plpgsql.extra_warnings` configuré à `strict_multi_assignment` :

```
SET plpgsql.extra_warnings TO 'strict_multi_assignment';

CREATE OR REPLACE FUNCTION public.foo()
 RETURNS void
 LANGUAGE plpgsql
AS $$
DECLARE
  x int;
  y int;
BEGIN
  SELECT 1 INTO x, y;
  SELECT 1, 2 INTO x, y;
  SELECT 1, 2, 3 INTO x, y;
END;
$$;

SELECT foo();
```

```
WARNING:  number of source and target fields in assignment does not
 match
DETAIL:  strict_multi_assignment check of extra_warnings is active.
HINT:  Make sure the query returns the exact list of columns.
WARNING:  number of source and target fields in assignment does not
 match
DETAIL:  strict_multi_assignment check of extra_warnings is active.
HINT:  Make sure the query returns the exact list of columns.

 foo
-----

(1 row)
```

43.13. Portage d'Oracle PL/SQL

Cette section explicite les différences entre le PL/pgSQL de PostgreSQL et le langage PL/SQL d'Oracle, afin d'aider les développeurs qui portent des applications d'Oracle® vers PostgreSQL.

PL/pgSQL est similaire à PL/SQL sur de nombreux aspects. C'est un langage itératif structuré en blocs et toutes les variables doivent être déclarées. Les affectations, boucles, conditionnelles sont similaires. Les principales différences que vous devez garder à l'esprit quand vous portez de PL/SQL vers PL/pgSQL sont:

- Si un nom utilisé dans une commande SQL peut être soit un nom de colonne d'une table utilisé dans la commande soit une référence à une variable de la fonction, PL/SQL le traite comme un nom de colonne. Par défaut, PL/pgSQL renverra une erreur se plaignant que le nom est ambigu. Vous pouvez indiquer `plpgsql.variable_conflict = use_column`, pour changer ce comportement et correspondre à PL/SQL, comme expliqué dans Section 43.11.1. Il est préféreable d'éviter de tels ambigüités des le début mais si vous devez migrer une grande quantité de code qui dépend de ce comportement, paramétrer `variable_conflict` peut s'avérer être la meilleure solution.

- Dans PostgreSQL, le corps de la fonction doit être écrit comme une chaîne litérale. Du coup, vous avez besoin d'utiliser les guillemets dollar ou l'échappement des simples guillemets dans le corps de la fonction. Voir la Section 43.12.1.

- Les noms de type de données ont besoin d'une conversion. Par exemple, les valeurs de type chaîne de caractères sont souvent déclarées de type `varchar2`, qui n'est pas un type standard. Avec PostgreSQL, utilisez à la place le type `varchar` ou `text`. De la même façon, remplacez le type `number` avec `numeric`, ou utilisez un autre type de données numériques s'il en existe un plus approprié.

- À la place des packages, utilisez des schémas pour organiser vos fonctions en groupes.

- Comme il n'y a pas de paquetages, il n'y a pas non plus de variables au niveau paquetage. Ceci est un peu ennuyant. Vous pourriez être capable de conserver un état par session dans les tables temporaires à la place.

- Les boucles `FOR` d'entiers en ordre inverse (`REVERSE`) fonctionnent différemment ; PL/SQL compte du second numéro jusqu'au premier alors que PL/pgSQL compte du premier jusqu'au second, ceci réclamant que les limites de la boucle soient échangées lors du portage. Cette incompatibilité est malheureuse mais a peu de chance d'être changée. (Voir Section 43.6.5.5.)

- Les boucles `FOR` sur des requêtes (autres que des curseurs) fonctionnent aussi différemment : la variable cible doit avoir été déclarée alors que PL/SQL les déclare toujours implicitement. Un avantage de ceci est que les valeurs des variables sont toujours accessibles à la sortie de la boucle.

- Il existe plusieurs différences de notation pour l'utilisation des variables curseurs.

43.13.1. Exemples de portages

L'Exemple 43.9 montre comment porter une simple fonction de PL/SQL vers PL/pgSQL.

Exemple 43.9. Portage d'une fonction simple de PL/SQL vers PL/pgSQL

Voici une fonction en PL/SQL Oracle :

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar2,
 v_version varchar2)
RETURN varchar2 IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

Parcourons cette fonction et voyons les différences avec PL/pgSQL :

- Le nom du type `varchar2` a dû être changé en `varchar` ou `text`. Dans les exemples de cette section, nous utiliserons `varchar` mais `text` est souvent un meilleur choix si nous n'avons pas besoin de limite spécifique de taille.
- Le mot clé `RETURN` dans le prototype de la fonction (pas dans le corps de la fonction) devient `RETURNS` dans PostgreSQL. De plus, `IS` devient `AS` et vous avez besoin d'ajouter une clause `LANGUAGE` parce que PL/pgSQL n'est pas le seul langage de procédures disponible.
- Dans PostgreSQL, le corps de la fonction est considéré comme une chaîne littérale, donc vous avez besoin d'utiliser les guillemets simples ou les guillemets dollar tout autour. Ceci se substitue au `/` de fin dans l'approche d'Oracle.
- La commande `show errors` n'existe pas dans PostgreSQL et n'est pas nécessaire car les erreurs sont rapportées automatiquement.

Voici de quoi aurait l'air cette fonction portée sous PostgreSQL :

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
 v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        return v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

L'Exemple 43.10 montre comment porter une fonction qui crée une autre fonction et comment gérer les problèmes de guillemets résultants.

Exemple 43.10. Portage d'une fonction qui crée une autre fonction de PL/SQL vers PL/pgSQL

La procédure suivante récupère des lignes d'une instruction `SELECT` et construit une grande fonction dont les résultats sont dans une instruction `IF` pour favoriser l'efficacité.

Voici la version Oracle :

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
 ORDER BY try_order;

    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION
 cs_find_referrer_type(v_host IN VARCHAR2,
                 v_domain IN VARCHAR2, v_url IN VARCHAR2) RETURN
 VARCHAR2 IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
        ' IF v_' || referrer_key.kind
 || ' LIKE ''' || referrer_key.key_string
 || ''' THEN RETURN ''' || referrer_key.referrer_type
 || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

Voici comment la fonction serait dans PostgreSQL :

```
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS
 void AS $func$
DECLARE
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN' ;

    FOR referrer_key IN SELECT * FROM cs_referrer_keys ORDER BY
 try_order LOOP
 func_body := func_body ||
 ' IF v_' || referrer_key.kind
 || ' LIKE ' || quote_literal(referrer_key.key_string)
 || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
 || '; END IF;' ;
    END LOOP;

    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
 v_domain varchar,
 v_url varchar)
 RETURNS varchar AS '
```

```
 || quote_literal(func_body)
 || ' LANGUAGE plpgsql;' ;

 EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

Notez comment le corps de la fonction est construit séparément et est passé au travers de `quote_literal` pour doubler tout symbole guillemet qu'il peut contenir. Cette technique est nécessaire parce que nous ne pouvons pas utiliser à coup sûr les guillemets dollar pour définir la nouvelle fonction : nous ne sommes pas sûr de savoir quelle chaîne sera interpolée à partir du champ `referrer_key.key_string` (nous supposons ici que ce `referrer_key.kind` vaut à coup sûr `host`, `domain` ou `url` mais `referrer_key.key_string` pourrait valoir autre chose, il pourrait contenir en particulier des signes dollar). Cette fonction est en fait une amélioration de l'original Oracle parce qu'il ne génèrera pas de code cassé quand `referrer_key.key_string` ou `referrer_key.referrer_type` contient des guillemets.

L'Exemple 43.11 montre comment porter une fonction ayant des paramètres `OUT` et effectuant des manipulations de chaînes. PostgreSQL n'a pas de fonction `instr` intégrée mais vous pouvez en créer une en utilisant une combinaison d'autres fonctions. Dans la Section 43.13.3, il y a une implémentation PL/pgSQL d'`instr` que vous pouvez utiliser pour faciliter votre portage.

Exemple 43.11. Portage d'une procédure avec manipulation de chaînes et paramètres `OUT` de PL/SQL vers PL/pgSQL

La procédure Oracle suivante est utilisée pour analyser une URL et renvoyer plusieurs éléments (hôte, chemin et requête). Les fonctions PL/pgSQL ne peuvent renvoyer qu'une seule valeur.

Voici la version Oracle :

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR2,
    v_host OUT VARCHAR2,  -- Celle-ci sera passée en retour
    v_path OUT VARCHAR2,  -- Celle-là aussi
    v_query OUT VARCHAR2) -- Et celle-là
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);
```

```
    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

Voici une traduction possible en PL/pgSQL :

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- This will be passed back
    v_path OUT VARCHAR,  -- This one too
    v_query OUT VARCHAR) -- And this one
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

Cette fonction pourrait être utilisée ainsi :

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

L'Exemple 43.12 montre comment porter une procédure qui utilise de nombreuses fonctionnalités spécifiques à Oracle.

Exemple 43.12. Portage d'une procédure de PL/SQL vers PL/pgSQL

La version Oracle :

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE
 end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock
        raise_application_error(-20000, 'Unable to create a new
 job: a job is currently running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id,
 sysdate);
        EXCEPTION
   WHEN dup_val_on_index THEN NULL; -- ne vous inquietez pas si
 cela existe déjà
    END;
    COMMIT;
END;
/
show errors
```

Voici comment nous pourrions porter cette procédure vers PL/pgSQL :

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id integer) AS $$
DECLARE
  a_running_job_count integer;
BEGIN
  LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

  SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE
 end_stamp IS NULL;

  IF a_running_job_count > 0 THEN
    COMMIT; -- free lock
    RAISE EXCEPTION 'Unable to create a new job: a job is currently
 running'; -- 1
  END IF;

  DELETE FROM cs_active_job;
  INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

  BEGIN
    INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id,
 now());
  EXCEPTION
    WHEN unique_violation THEN -- 2
```

```
    -- ne vous inquietez pas si cela existe déjà
  END;
  COMMIT;
END;
$$ LANGUAGE plpgsql;
```

1. La syntaxe de `RAISE` est considérablement différente de l'instruction Oracle similaire, bien que le cas basique du `RAISE` *`nom_exception`* fonctionne de façon similaire.
2. Les noms d'exceptions supportées par PL/pgSQL sont différents de ceux d'Oracle. L'ensemble de noms d'exceptions intégré est plus important (voir l'Annexe A). Il n'existe actuellement pas de façon de déclarer des noms d'exceptions définis par l'utilisateur, bien que vous puissiez aussi ignorer les valeurs SQLSTATE choisies par l'utilisateur.

43.13.2. Autres choses à surveiller

Cette section explique quelques autres choses à surveiller quand on effectue un portage de fonctions PL/SQL Oracle vers PostgreSQL.

43.13.2.1. Annulation implicite après une exception

Dans PL/pgSQL, quand une exception est récupérée par une clause `EXCEPTION`, toutes les modifications de la base de données depuis le bloc `BEGIN` sont automatiquement annulées. C'est-à-dire que le comportement est identique à celui obtenu à partir d'Oracle avec :

```
BEGIN
SAVEPOINT s1;
... code ici ...
EXCEPTION
WHEN ... THEN
ROLLBACK TO s1;
... code ici ...
WHEN ... THEN
ROLLBACK TO s1;
... code ici ...
END;
```

Si vous traduisez une procédure d'Oracle qui utilise `SAVEPOINT` et `ROLLBACK TO` dans ce style, votre tâche est facile : omettez `SAVEPOINT` et `ROLLBACK TO`. Si vous avez une procédure qui utilise `SAVEPOINT` et `ROLLBACK TO` d'une façon différente, alors un peu de réflexion supplémentaire sera nécessaire.

43.13.2.2. EXECUTE

La version PL/pgSQL d'`EXECUTE` fonctionne de façon similaire à la version PL/SQL mais vous devez vous rappeler d'utiliser `quote_literal` et `quote_ident` comme décrit dans la Section 43.5.4. Les constructions de type `EXECUTE 'SELECT * FROM $1';` ne fonctionneront pas de façon fiable à moins d'utiliser ces fonctions.

43.13.2.3. Optimisation des fonctions PL/pgSQL

PostgreSQL vous donne deux modificateurs de création de fonctions pour optimiser l'exécution : la « volatilité » (la fonction renvoie toujours le même résultat quand on lui donne les mêmes arguments) et la « rigueur » (une fonction renvoie NULL si tous ses arguments sont NULL). Consultez la page de référence de CREATE FUNCTION pour les détails.

Pour faire usage de ces attributs d'optimisation, votre instruction `CREATE FUNCTION` devrait ressembler à ceci :

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
```

```
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

43.13.3. Annexe

Cette section contient le code d'un ensemble de fonctions `instr` compatible Oracle que vous pouvez utiliser pour simplifier vos efforts de portage.

```
--
-- fonctions instr qui reproduisent l'équivalent Oracle
-- Syntaxe : instr(string1, string2 [, n [, m]])
-- où [] signifie paramètre optionnel.
--
-- Cherche string1 en commençant par le  n-ième caractère pour la
 m-ième occurrence
-- de string2.  Si n est négatif, cherche en sens inverse, en
 commençant au caractère
-- en position abs(n) à partir de la fin de string1.
-- If n n'est pas fourni, suppose 1 (la recherche commence au
 premier caractère).
-- Si m n'est pas fourni, suppose 1 (la recherche commence au
 premier caractère).
-- Renvoie l'index de début de string2 dans string1, ou 0 si
 string2 n'est pas trouvé.
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
BEGIN
    RETURN instr($1, $2, 1);
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search_for IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
```

```
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF occur_index <= 0 THEN
        RAISE 'argument ''%'' is out of range', occur_index
          USING ERRCODE = '22003';
    END IF;

    IF beg_index > 0 THEN
        beg := beg_index - 1;
        FOR i IN 1..occur_index LOOP
            temp_str := substring(string FROM beg + 1);
            pos := position(string_to_search_for IN temp_str);
            IF pos = 0 THEN
                RETURN 0;
            END IF;
            beg := beg + pos;
        END LOOP;

        RETURN beg;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                occur_number := occur_number + 1;
                IF occur_number = occur_index THEN
```

```
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

Chapitre 44. PL/Tcl -- Langage de procédures Tcl

PL/Tcl est un langage de procédures chargeable pour le système de bases de données PostgreSQL, activant l'utilisation du langage Tcl[1] pour l'écriture de fonctions et de procédures PostgreSQL.

44.1. Aperçu

PL/Tcl offre un grand nombre de fonctionnalités qu'un codeur de fonctions dispose avec le langage C, avec quelques restrictions et couplé à de puissantes bibliothèques de traitement de chaînes de caractères disponibles pour Tcl.

Une *bonne* restriction est que tout est exécuté dans le contexte de l'interpréteur Tcl. En plus de l'ensemble sûr de commandes limitées de Tcl, seules quelques commandes sont disponibles pour accéder à la base via SPI et pour envoyer des messages via `elog()`. PL/Tcl ne fournit aucun moyen pour accéder aux internes du serveur de bases ou pour gagner un accès au niveau système d'exploitation avec les droits du processus serveur PostgreSQL comme le fait une fonction C. Du coup, les utilisateurs de la base, sans droits, peuvent utiliser ce langage en toute confiance ; il ne leur donne pas une autorité illimitée.

L'autre restriction d'implémentation est que les fonctions Tcl ne peuvent pas être utilisées pour créer des fonctions d'entrées/sorties pour les nouveaux types de données.

Quelques fois, il est préférable d'écrire des fonctions Tcl non restreintes par le Tcl sûr. Par exemple, vous pourriez vouloir une fonction Tcl pour envoyer un courrier électronique. Pour gérer ces cas, il existe une variante de PL/Tcl appelée `PL/TclU` (Tcl non accrédité). C'est exactement le même langage sauf qu'un interpréteur Tcl complet est utilisé. *Si PL/TclU est utilisé, il doit être installé comme langage de procédures non accrédité* de façon à ce que seuls les superutilisateurs de la base de données puissent créer des fonctions avec lui. Le codeur d'une fonction PL/TclU doit faire attention au fait que la fonction ne pourra pas être utilisé pour faire autre chose que son but initial, car il sera possible de faire tout ce qu'un administrateur de la base de données peut faire.

Le code de l'objet partagé pour les gestionnaires d'appel PL/Tcl et PL/TclU est automatiquement construit et installé dans le répertoire des bibliothèques de PostgreSQL si le support de Tcl est spécifié dans l'étape de configuration de la procédure d'installation. Pour installer PL/Tcl et/ou PL/TclU dans une base de données particulière, utilisez la commande `CREATE EXTENSION`, par exemple `CREATE EXTENSION pltcl` ou `CREATE EXTENSION pltclu`.

44.2. Fonctions et arguments PL/Tcl

Pour créer une fonction dans le langage PL/Tcl, utilisez la syntaxe standard de CREATE FUNCTION :

```
CREATE FUNCTION nom_fonction (types_arguments) RETURNS
type_en_retour AS $$
    # corps de la fonction PL/Tcl
$$ LANGUAGE pltcl;
```

PL/TclU est identique sauf que le langage doit être `pltclu`.

Le corps de la fonction est simplement un bout de script Tcl. Quand la fonction est appelée, les valeurs d'argument sont passées au script Tcl comme des variables nommées `1`... *n*. Le résultat est retournée depuis le code Tcl de la manière habituelle, avec un ordre `return`. Pour une procédure, la valeur de retour du code Tcl est ignorée.

[1] https://www.tcl.tk/

Par exemple, une fonction renvoyant le plus grand de deux valeurs entières pourrait être définie ainsi :

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

Notez la clause `STRICT` qui nous permet d'éviter de penser aux valeurs NULL en entrées : si une valeur NULL est passée, la fonction ne sera pas appelée du tout mais renverra automatiquement un résultat nul.

Dans une fonction non stricte, si la valeur réelle d'un argument est NULL, la variable `$`*n* correspondante sera initialisée avec une chaîne vide. Pour détecter si un argument particulier est NULL, utilisez la fonction `argisnull`. Par exemple, supposez que nous voulons `tcl_max` avec un argument NULL et un non NULL pour renvoyer l'argument non NULL plutôt que NULL :

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

Comme indiqué ci-dessus, pour renvoyer une valeur NULL à partir d'une fonction PL/Tcl, exécutez `return_null`. Ceci peut être fait que la fonction soit stricte ou non.

Les arguments de type composé sont passés à la fonction comme des tableaux Tcl. Les noms des éléments du tableau sont les noms d'attribut du type composite. Si un attribut dans la ligne passée a la valeur NULL, il n'apparaîtra pas dans le tableau. Voici un exemple :

```
CREATE TABLE employe (
    nom text,
    salaire integer,
    age integer
);

CREATE FUNCTION surpaye(employe) RETURNS boolean AS $$
    if {200000.0 < $1(salaire)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salaire)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

Les fonctions PL/Tcl peuvent également retourner des résultats de type composite. Pour cela, le code Tcl doit retourner une liste de paires nom de colonnes / valeurs correspondant au type de résultat attendu. Tout nom de colonne omis de la liste sera retourné comme NULL, et une erreur est levée s'il y a un nom de colonne inattendu. Voici un exemple :

```
CREATE FUNCTION square_cube(in int, out squared int, out cubed int)
 AS $$
```

```
    return [list squared [expr {$1 * $1}] cubed [expr {$1 * $1 *
 $1}]]
$$ LANGUAGE pltcl;
```

Les arguments en sortie des procédures sont renvoyées de la même façon. Par exemple :

```
CREATE PROCEDURE tcl_triple(INOUT a integer, INOUT b integer) AS $$
    return [list a [expr {$1 * 3}] b [expr {$2 * 3}]]
$$ LANGUAGE pltcl;

CALL tcl_triple(5, 10);
```

> **Astuce**
>
> La liste résultat peut être faite à partir d'une représentation de tableau du tuple désiré avec la commande Tcl `array get`. Par exemple:
>
> ```
> CREATE FUNCTION raise_pay(employee, delta int) RETURNS
>  employee AS $$
>     set 1(salary) [expr {$1(salary) + $2}]
>     return [array get 1]
> $$ LANGUAGE pltcl;
> ```

Les fonctions PL/Tcl peuvent retourner des ensembles. Pour cela, le code Tcl devrait appeler `return_next` une fois par ligne à être retournée, passant soit la valeur appropriée quand un type scalaire est retourné, soit une liste de paires de nom de colonne / valeur quand un type composite est retourné. Voici un exemple retournant un type scalaire :

```
CREATE FUNCTION sequence(int, int) RETURNS SETOF int AS $$
    for {set i $1} {$i < $2} {incr i} {
        return_next $i
    }
$$ LANGUAGE pltcl;
```

et voici un exemple retournant un type composite :

```
CREATE FUNCTION table_of_squares(int, int) RETURNS TABLE (x int, x2
 int) AS $$
    for {set i $1} {$i < $2} {incr i} {
        return_next [list x $i x2 [expr {$i * $i}]]
    }
$$ LANGUAGE pltcl;
```

44.3. Valeurs des données avec PL/Tcl

Les valeurs des arguments fournies au code d'une fonction PL/Tcl sont simplement les arguments en entrée convertis au format texte (comme s'ils avaient été affichés par une instruction `SELECT`). Les

commandes `return` et `return_next` accepteront toute chaîne qui est un format d'entrée acceptable pour le type de résultat déclaré pour la fonction, ou pour la colonne spécifiée d'un type de retour composite.

44.4. Données globales avec PL/Tcl

Quelque fois, il est utile d'avoir des données globales qui sont conservées entre deux appels à une fonction ou qui sont partagées entre plusieurs fonctions. Ceci peut être facilement obtenu car toutes les fonctions PL/Tcl exécutées dans une session partagent le même interpréteur Tcl sûr. Donc, toute variable globale Tcl est accessible aux appels de fonctions PL/Tcl et persisteront pour la durée de la session SQL (notez que les fonctions PL/TclU partagent de la même façon les données globales mais elles sont dans un interpréteur Tcl différent et ne peuvent pas communiquer avec les fonctions PL/Tcl). C'est facile à faire en PL/Tcl mais il existe quelques restrictions qui doivent être comprises.

Pour des raisons de sécurité, PL/Tcl exécute les fonctions appelées par tout rôle SQL dans un interpréteur Tcl séparé pour ce rôle. Ceci empêche une interférence accidentelle ou malicieuse d'un utilisateur avec le comportement des fonctions PL/Tcl d'un autre utilisateur. Chaque interpréteur aura ses propres valeurs pour toutes les variables globales Tcl. Du coup, deux fonctions PL/Tcl partageront les mêmes variables globales si et seulement si elles sont exécutées par le même rôle SQL. Dans une application où une seule session exécute du code sous plusieurs rôles SQL(via des fonctions `SECURITY DEFINER`, l'utilisation de `SET ROLE`, etc), vous pouvez avoir besoin de mettre des étapes explicites pour vous assurer que les fonctions PL/Tcl peuvent partager des données. Pour cela, assurez-vous que les fonctions qui doivent communiques ont pour propriétaire le même utilisateur et marquez-les avec l'option `SECURITY DEFINER`. Bien sûr, vous devez faire attention à ce que de telles fonctions ne puissent pas être utilisées pour faire des choses non souhaitées.

Toutes les fonctions PL/TclU utilisées dans une session s'exécutent avec le même interpréteur Tcl, qui est bien sûr différent des interpréteurs utilisées pour les fonctions PL/Tcl. Donc les données globales sont automatiquement partagées entre des fonctions PL/TclU. Ceci n'est pas considéré comme un risque de sécurité parce que toutes les fonctions PL/TclU s'exécutent dans le même niveau de confiance, celui d'un superutilisateur.

Pour aider à la protection des fonctions PL/Tcl sur les interférences non intentionnelles, un tableau global est rendu disponible pour chaque fonction via la commande `upvar`. Le nom global de cette variable est le nom interne de la fonction alors que le nom local est `GD`. Il est recommandé d'utiliser `GD` pour les données privées persistantes d'une fonction. Utilisez les variables globales Tcl uniquement pour les valeurs que vous avez l'intention de partager avec les autres fonctions. (Notez que les tableaux `GD` sont seulement globaux à l'intérieur d'un interpréteur particulier, pour qu'ils ne franchissent pas les restrictions de sécurité mentionnées ci-dessus.)

Un exemple de l'utilisation de `GD` apparaît dans l'exemple `spi_execp` ci-dessous.

44.5. Accès à la base de données depuis PL/Tcl

Dans cette section, nous suivons la convention Tcl habituelle pour l'utilisation des points d'interrogation, plutôt que les crochets, pour indiquer un élément optionnel dans un synopsis de syntaxe. Les commandes suivantes sont disponibles pour accéder à la base de données depuis le corps d'une fonction PL/Tcl :

`spi_exec ?-count` *`n`*`? ?-array` *`name`*`? `*`command`*` ?`*`loop-body`*`?`

> Exécute une commande SQL donnée en tant que chaîne. Une erreur dans la commande lève une erreur. Sinon, la valeur de retour de `spi_exec` est le nombre de lignes intéressées dans le processus (sélection, insertion, mise à jour ou suppression) par la commande ou zéro si la commande est une instruction utilitaire. De plus, si la commande est une instruction `SELECT`, les valeurs des données sélectionnées sont placées dans des variables Tcl décrites ci-dessous.

La valeur optionnelle `-count` indique à `spi_exec` d'arrêter une fois que *`n`* lignes ont été récupérées, tout comme si la requête incluait une clause `LIMIT`. Si *`n`* vaut zéro, la requête est exécutée jusqu'à sa fin, tout comme si `-count` était omis.

Si la commande est une instruction `SELECT`, les valeurs des colonnes de résultat sont placées dans les variables Tcl nommées d'après les colonnes. Si l'option `-array` est donnée, les valeurs de colonnes sont stockées à la place dans les éléments d'un tableau associatif nommé, les noms des colonnes étant utilisés comme index du tableau. De plus, le numéro de ligne courant dans le résultat (en commençant par zéro) est enregistré dans l'élément de tableau nommé « `.tupno` », sauf si ce nom est utilisé comme nom de colonne dans le résultat.

Si la commande est une instruction `SELECT` et qu'aucun script *`loop-body`* n'est donné, alors seule la première ligne de résultats est stockée dans des variables Tcl ou des éléments de tableau ; les lignes suivantes sont ignorées. Aucun stockage n'intervient si la requête ne renvoie pas de ligne (ce cas est détectable avec le résultat de la fonction `spi_exec`). Par exemple :

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

initialisera la variable Tcl `$cnt` avec le nombre de lignes dans le catalogue système `pg_proc`.

Si l'argument *`loop-body`* optionnel est donné, il existe un morceau de script Tcl qui est exécuté une fois pour chaque ligne du résultat de la requête (*`loop-body`* est ignoré si la commande donnée n'est pas un `SELECT`). Les valeurs des colonnes de la ligne actuelle sont stockées dans des variables Tcl avant chaque itération. Par exemple :

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

affichera un message de trace pour chaque ligne de `pg_class`. Cette fonctionnalité travaille de façon similaire aux autres constructions de boucles de Tcl ; en particulier, `continue` et `break` fonctionnent de la même façon à l'intérieur de *`loop-body`*.

Si une colonne d'un résultat de la requête est NULL, la variable cible est « dés-initialisée » plutôt qu'initialisée.

`spi_prepare` *`query typelist`*

Prépare et sauvegarde un plan de requête pour une exécution future. Le plan sauvegardé sera conservé pour la durée de la session actuelle.

La requête peut utiliser des paramètres, c'est-à-dire des emplacements pour des valeurs à fournir lorsque le plan sera réellement exécuté. Dans la chaîne de requête, faites référence aux paramètres avec les symboles `$1` ... `$`*`n`*. Si la requête utilise les paramètres, les noms des types de paramètre doivent être donnés dans une liste Tcl (écrivez une liste vide pour *`typelist`* si aucun paramètre n'est utilisé).

La valeur de retour de `spi_prepare` est l'identifiant de la requête à utiliser dans les appels suivants à `spi_execp`. Voir `spi_execp` pour un exemple.

`spi_execp ?-count` *`n`*`? ?-array` *`name`*`? ?-nulls` *`string`*`? queryid ?value-list? ?loop-body?`

Exécute une requête préparée précédemment avec `spi_prepare`. *`queryid`* est l'identifiant renvoyé par `spi_prepare`. Si la requête fait référence à des paramètres, une liste de valeurs (*`value-list`*) doit être fournie. C'est une liste Tcl des valeurs réelles des paramètres. La liste doit être de la même longueur que la liste de types de paramètres donnée précédemment lors de l'appel à `spi_prepare`. Oubliez-la si la requête n'a pas de paramètres.

La valeur optionnelle pour `-nulls` est une chaîne d'espaces et de caractères `'n'` indiquant à `spi_execp` les paramètres nuls. Si indiqué, elle doit avoir exactement la même longueur que *`value-list`*. Si elle est omise, toutes les valeurs de paramètres sont non NULL.

Sauf si la requête et ses paramètres sont spécifiés, `spi_execp` fonctionne de la même façon que `spi_exec`. Les options `-count`, `-array` et *`loop-body`* sont identiques. Du coup, la valeur du résultat l'est aussi.

Voici un exemple d'une fonction PL/Tcl utilisant un plan préparé :

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {
        # prépare le plan sauvegardé au premier appel
        set GD(plan) [ spi_prepare \
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1
 AND num <= \$2" \
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
$$ LANGUAGE pltcl;
```

Nous avons besoin des antislashs à l'intérieur de la chaîne de la requête passée à `spi_prepare` pour s'assurer que les marqueurs $*n* sont passés au travers de `spi_prepare` sans transformation et ne sont pas remplacés avec la substitution de variables de Tcl.

`subtransaction` *`command`*

Le script Tcl contenu dans *`command`* est exécuté à l'intérieur d'une sous-transaction SQL. Si le script renvoie une erreur, la sous-transaction entière est annulée avant de renvoyer une erreur au code Tcl appelant. Voir Section 44.9 pour plus de détails et un exemple.

`quote` *`string`*

Double toutes les occurrences de guillemet simple et d'antislash dans la chaîne donnée. Ceci peut être utilisé pour mettre entre guillemets des chaînes de façon sûr et pour qu'elles puissent être insérées dans des commandes SQL passées à `spi_exec` ou `spi_prepare`. Par exemple, pensez à une chaîne de commande SQL comme :

```
"SELECT '$val' AS ret"
```

où la variable Tcl `val` contient actuellement le mot `doesn't`. Ceci finirait avec la chaîne de commande :

```
SELECT 'doesn't' AS ret
```

qui va causer une erreur d'analyse lors de `spi_exec` ou de `spi_prepare`. Pour fonctionner correctement, la commande soumise devrait contenir :

```
SELECT 'doesn''t' AS ret
```

qui peut-être créé avec PL/Tcl en utilisant :

```
"SELECT '[ quote $val ]' AS ret"
```

Un avantage de `spi_execp` est que vous n'avez pas à mettre entre guillemets des valeurs de paramètres comme ceux-ci car les paramètres ne sont jamais analysés comme faisant partie de la chaîne de la commande SQL.

`elog` *`level msg`*

Émet une trace ou un message d'erreur. Les niveaux possibles sont `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `ERROR` et `FATAL`. `ERROR` élève une condition d'erreur ; si elle n'est pas récupérée par le code Tcl, l'erreur est propagée à la requête appelante, causant l'annulation de la transaction ou sous-transaction en cours. Ceci est en fait identique à la commande `error`. `FATAL` annule la transaction et fait que la session courante s'arrête (il n'existe probablement aucune raison d'utiliser ce niveau d'erreur dans les fonctions PL/Tcl mais il est fourni pour que tous les messages soient tout de même disponibles). Les autres niveaux génèrent seulement des messages de niveaux de priorité différent. Le fait que les messages d'un niveau de priorité particulier sont reportés au client, écrit dans les journaux du serveur ou les deux à la fois, est contrôlé par les variables de configuration log_min_messages et client_min_messages. Voir le Chapitre 20 et Section 44.8 pour plus d'informations.

44.6. Fonctions triggers en PL/Tcl

Les fonctions trigger peuvent être écrites en PL/Tcl. PostgreSQL requiert qu'une fonction appelée en tant que trigger doit être déclarée comme une fonction sans arguments et retourner une valeur de type `trigger`.

L'information du gestionnaire du trigger est passée au corps de la fonction avec les variables suivantes :

`$TG_name`

Nom du trigger provenant de l'instruction `CREATE TRIGGER`.

`$TG_relid`

L'identifiant objet de la table qui est à la cause du lancement du trigger.

`$TG_table_name`

Le nom de la table qui est à la cause du lancement du trigger.

`$TG_table_schema`

Le schéma de la table qui est à la cause du lancement du trigger.

`$TG_relatts`

Une liste Tcl des noms des colonnes de la table, préfixée avec un élément de liste vide. Donc, rechercher un nom de colonne dans la liste avec la commande `lsearch` de Tcl renvoie le numéro de l'élément, en commençant à 1 pour la première colonne, de la même façon que les colonnes sont numérotées personnellement avec PostgreSQL.

`$TG_when`

La chaîne `BEFORE`, `AFTER` ou `INSTEAD OF` suivant le type de l'événement du trigger.

`$TG_level`

La chaîne `ROW` ou `STATEMENT` suivant le type de l'événement du trigger.

`$TG_op`

La chaîne `INSERT`, `UPDATE`, `DELETE` ou `TRUNCATE` suivant le type de l'événement du trigger.

`$NEW`

Un tableau associatif contenant les valeurs de la nouvelle ligne de la table pour les actions `INSERT` ou `UPDATE` ou vide pour `DELETE`. Le tableau est indexé par nom de colonne. Les colonnes NULL n'apparaissent pas dans le tableau. Ce paramètre n'est pas initialisé pour les triggers de niveau instruction.

`$OLD`

Un tableau associatif contenant les valeurs de l'ancienne ligne de la table pour les actions `UPDATE` or `DELETE` ou vide pour `INSERT`. Le tableau est indexé par nom de colonne. Les colonnes NULL n'apparaissent pas dans le tableau. Ce paramètre n'est pas initialisé pour les triggers de niveau instruction.

`$args`

Une liste Tcl des arguments de la fonction ainsi que l'instruction `CREATE TRIGGER`. Ces arguments sont aussi accessibles par `$1` ... `$`*n* dans le corps de la fonction.

Le code de retour d'une fonction trigger peut être faite avec une des chaînes `OK` ou `SKIP` ou une liste de paires nom de colonne/valeur renvoyée par la commande Tcl `array get`. Si la valeur de retour est `OK`, l'opération (`INSERT`/`UP DATE`/`DELETE`) qui a lancé le trigger continuera normalement. `SKIP` indique au gestionnaire de triggers de supprimer silencieusement l'opération pour cette ligne. Si une liste est renvoyée, elle indique à PL/Tcl de renvoyer la ligne modifiée au gestionnaire de triggers ; le contenu de la ligne modifiée est spécifié par les noms de colonne et par les valeurs dans la liste. Toute colonne non mentionné dans la liste est configuré à NULL. Renvoyer une ligne modifiée n'a un intérêt que pour les triggers niveau ligne pour `BEFORE`, `INSERT` ou `UPDATE` pour laquelle la ligne modifiée sera insérée au lieu de celle donnée dans `$NEW` ; ou pour les triggers niveau ligne `INSTEAD OF`, `INSERT` ou `UPDATE` où la ligne renvoyée est utilisée comme données sources pour les clauses `INSERT RETURNING` ou `UPDATE RETURNING`. Dans les triggers `BEFORE DELETE` ou `INSTEAD OF DELETE` de niveau ligne, renvoyer une ligne modifiée a le même effet que renvoyer `OK`, autrement dit l'opération continue. La valeur de retour est ignorée pour les autres types de triggers.

> **Astuce**
>
> La liste de résultats peut être réalisée à partir une représentation en tableau de la ligne modifiée avec la commande Tcl `array get`.

Voici un petit exemple de fonction trigger qui force une valeur entière dans une table pour garder trace du nombre de mises à jour réalisées sur la ligne. Pour les nouvelles lignes insérées, la valeur est initialisée à 0 puis incrémentée à chaque opération de mise à jour.

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
$$ LANGUAGE pltcl;
```

```
CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE FUNCTION trigfunc_modcount('modcnt');
```

Notez que la fonction trigger elle-même ne connaît pas le nom de la colonne ; c'est fourni avec les arguments du trigger. Ceci permet à la fonction trigger d'être ré-utilisée avec différentes tables.

44.7. Fonctions trigger sur événement en PL/Tcl

Les fonctions triggers sur événement peuvent être écrites en PL/Tcl. PostgreSQL requiert qu'une fonction qui doit être appelée comme trigger sur événement doit être déclarée comme une fonction sans arguments et comme renvoyant le type `event_trigger`.

L'information provenant du gestionnaire des triggers est passée au corps de la fonction avec les variables suivantes :

`$TG_event`

Le nom de l'événement pour lequel le trigger a été déclenché.

`$TG_tag`

La balise de la commande pour laquelle le trigger a été déclenché.

La valeur de retour de la fonction trigger est ignorée.

Voici un petit exemple de fonction trigger sur événement qui lève un message `NOTICE` à chaque fois qu'une commande supportée est exécutée :

```
CREATE OR REPLACE FUNCTION tclsnitch() RETURNS event_trigger AS $$
  elog NOTICE "tclsnitch: $TG_event $TG_tag"
$$ LANGUAGE pltcl;

CREATE EVENT TRIGGER tcl_a_snitch ON ddl_command_start EXECUTE
 FUNCTION tclsnitch();
```

44.8. Gestion des erreurs avec PL/Tcl

Le code Tcl contenu ou appelé à partir d'une fonction PL/Tcl peut lever une erreur, soit en exécutant des opérations invalides ou en générant une erreur en utilisant la commande Tcl `error` ou la commande PL/Tcl `elog`. Si une erreur n'est pas rattrapée mais est autorisée à être propagée en dehors du niveau racine de l'exécution de la fonction PL/Tcl, elle est rapportée comme une erreur SQL dans la requête appelant la fonction.

Les erreurs SQL survenant dans les commandes PL/Tcl `spi_exec`, `spi_prepare` et `spi_execp` sont rapportées comme des erreurs Tcl, donc elles sont récupérables par la commande Tcl `catch`. (Chacune des ces commandes PL/Tcl exécutent leurs opérations SQL dans une sous transaction, qui est annulée en cas d'erreur, si bien que n'importe quelle opération partiellement terminée sera automatiquement nettoyée.) De la même façon, si une erreur se propage en dehors du niveau racine sans avoir été rattrapée, elle sera rapportée en erreur SQL.

Tcl fournit une variable `errorCode` pouvant représenter des informations supplémentaires sur une erreur dans un format qui est simple à interpréter pour les programmes Tcl. Le contenu est dans le format liste Tcl, et le premier mot identifie le sous-système ou la bibliothèque rapportant l'erreur ; au-delà, le contenu est laissé au sous-système individuel ou à la bibliothèque. Pour les erreurs au niveau base rapportées par les commandes PL/Tcl commands, le premier mot est `POSTGRES`, le second est le numéro de version du serveur, et les mots supplémentaires sont les paires nom/valeur des champs fournissant des informations détaillées sur l'erreur. Les champs `SQLSTATE`, `condition` et `message` sont toujours fournies (les deux premiers représentent le code d'erreur et le nom de la condition comme indiqués dans Annexe A). Les champs potentiellement présents incluent `detail`, `hint`, `context`, `schema`, `table`, `column`, `datatype`, `constraint`, `statement`, `cursor_position`, `filename`, `lineno` et `funcname`.

Une façon agréable de travailler avec l'information `errorCode` de PL/Tcl est de la charger dans un tableau pour que les noms du champ deviennent des indices du tableau. Un code relatif ressemblerait à ceci :

```
if {[catch { spi_exec $sql_command }]} {
    if {[lindex $::errorCode 0] == "POSTGRES"} {
        array set errorArray $::errorCode
        if {$errorArray(condition) == "undefined_table"} {
            # gestion de la table manquante
        } else {
            # gestion des autres types d'erreur SQL
        }
    }
}
```

(Les symboles deux-points spécifient explicitement que `errorCode` est une variable globale.)

44.9. Sous-transactions explicites dans PL/Tcl

Récupérer des erreurs causées par des à la base de données comme décris dans Section 44.8 peut mener à une situation indésirable où certaines opérations réussissent avant que l'une d'entre elles échoue, et après avoir récupéré cette erreur la donnée est laissé dans un état incohérent. PL/Tcl offre une solution à ce problème sous la forme de sous-transactions explicites :

Étudions une fonction qui implémente un transfert entre deux compte :

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
    if [catch {
        spi_exec "UPDATE accounts SET balance = balance - 100 WHERE
 account_name = 'joe'"
        spi_exec "UPDATE accounts SET balance = balance + 100 WHERE
 account_name = 'mary'"
    } errormsg] {
        set result [format "error transferring funds: %s"
 $errormsg]
    } else {
        set result "funds transferred successfully"
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote
 $result]')"
$$ LANGUAGE pltcl;
```

Si le deuxième ordre UPDATE échoue en levant une exception, cette fonction tracera l'échec, mais le résultat du premier UPDATE sera néanmoins validé. Concrètement, le montant sera débité du compte de Joe, mais ne sera pas transféré sur le compte de Mary. C'est le cas car chaque appel à spi_exec est une sous-transaction séparé, et seule l'une de ces sous-transactions est annulée.

Pour gérer un cas comme ça, vous pouvez entourer vos opération sur la base d'une sous-transaction explicite, qui sera annulée ou validée comme un tour. PL/Tcl fournit une commande subtransaction pour gérer ça. Nous pouvons réécrire notre fonction ainsi :

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
    if [catch {
        subtransaction {
            spi_exec "UPDATE accounts SET balance = balance - 100
 WHERE account_name = 'joe'"
            spi_exec "UPDATE accounts SET balance = balance + 100
 WHERE account_name = 'mary'"
        }
    } errormsg] {
        set result [format "error transferring funds: %s"
 $errormsg]
    } else {
        set result "funds transferred successfully"
    }
    spi_exec "INSERT INTO operations (result) VALUES ('[quote
 $result]')"
$$ LANGUAGE pltcl;
```

Veuillez noter que l'utilisation de catch est toujours nécessaire pour gérer ce cas. Autrement l'erreur se propagerait jusqu'au niveau racine de la fonction, ce qui empêcherait l'insertion voulue dans la table operations. La commande subtransaction ne récupère pas les erreurs, elle s'assure seulement que tous les ordres sur la base exécutés dans sa portée seront annulés ensemble si une erreur survient.

L'annulation d'une sous-transaction explicite arrive lors de n'importe quelle erreur rapportée par le code Tcl contenu, pas uniquement pour celle en provenance de la base de données. Ainsi une exception standard Tcl levée dans une commande subtransaction aura également pour effet d'annuler la sous-transaction. Toutefois, les sortie du code Tcl contenu sans erreur (par exemple, du fait d'un return) ne déclencheront pas d'annulation.

44.10. Gestion des transactions

Dans une procédure appelée au haut niveau ou dans un bloc de code anonyme (commande DO) appelé au haut niveau, il est possible de contrôler les transactions. Pour valider la transaction en cours, appelez la commande commit command. Pour annuler la transaction en cours, appelez la commande rollback. (Notez qu'il n'est pas possible d'exécuter les commandes SQL COMMIT ou ROLLBACK via spi_exec ou similaire. Cela doit se faire en utilisant ces fonctions.) À la fin d'une transaction, une nouvelle transaction est automatiquement démarrée, donc il n'existe pas de commande séparée pour ça.

Voici un exemple :

```
CREATE PROCEDURE transaction_test1()
LANGUAGE pltcl
AS $$
```

```
for {set i 0} {$i < 10} {incr i} {
    spi_exec "INSERT INTO test1 (a) VALUES ($i)"
    if {$i % 2 == 0} {
        commit
    } else {
        rollback
    }
}
$$;

CALL transaction_test1();
```

Les transactions ne peuvent pas être terminées quand une sous-transaction explicite est active.

44.11. Configuration PL/Tcl

Cette section liste les paramètre de configuration qui affectent PL/Tcl.

`pltcl.start_proc` (`string`)

Ce paramètre, s'il est positionné à autre chose qu'une chaîne vide, spécifie le nom (potentiellement qualifié du schéma) d'ue fonction PL/Tcl sans paramètre qui doit être exécutée chaque fois qu'un nouvel interpréteur Tcl est créé pour Pl/Tcl. Une telle fonction peut effectuer une initialisation pour la session, comme charger du code Tcl additionnel. Un nouvel interpréteur Tcl est créé wuen une fonction PL/Tcl est exécutée pour la première fois dans une session, ou quand un autre interpréteur doit être créé du fait de l'appel à une fonction PL/Tcl par un nouveau rôle SQL.

La fonction référencée doit être écrite dans le langage `pltcl`, et ne doit pas être marquée comme `SECURITY DEFINER`. (Ces restrictions s'assurent que la fonction est lancée dans l'interpréteur qu'elle est censée initialisée.) L'utilisateur courant doit avoir l'autorisation d'appeler cette fonction également.

Si la fonction échoue avec une erreur cela annulera l'appel à la fonction qui a causée la création du nouvel interpréteur et l'erreur sera propagée jusqu'à la requête appelante, annulant de fait la transaction ou sous transaction courante. Toute action déjà effectuée au sein de Tcl ne sera pas annulée; toutefois, cette interpréteur ne sera jamais réutilisé. Si le langage est utilisé à nouveau l'initialisation sera de nouveau tentée avec une nouvel interpréteur Tcl.

Seuls les superutilisateurs peuvent modifier ce paramètre. Bien que ce paramètre puisse être changé au sein d'une session, un tel changement n'affectera pas les interpréteurs Tcl qui ont déjà été créés.

`pltclu.start_proc` (`string`)

Ce paramètre est exactement comme `pltcl.start_proc`, sauf qu'il s'applique à PL/TclU. La fonction référencée doit être écrite dans le langage `pltclu`.

44.12. Noms de procédure Tcl

Avec PostgreSQL, le même nom de fonction peut être utilisé par plusieurs fonctions tant que le nombre d'arguments ou leurs types diffèrent. Néanmoins, Tcl requiert que les noms de procédure soient distincts. PL/Tcl gère ceci en faisant en sorte que les noms de procédures Tcl internes contiennent l'identifiant de l'objet de la fonction depuis la table système `pg_proc`. Du coup, les fonctions PostgreSQL avec un nom identique et des types d'arguments différents seront aussi des procédures Tcl différentes. Ceci ne concerne normalement pas le développeur PL/Tcl mais cela pourrait apparaître dans une session de débogage.

Chapitre 45. PL/Perl -- Langage de procédures Perl

PL/Perl est un langage de procédures chargeable qui vous permet d'écrire des fonctions et procédures PostgreSQL dans le langage de programmation Perl[1].

Le principal avantage habituellement cité quant à l'utilisation de Perl est que cela permet l'utilisation des nombreux opérateurs et fonctions de « gestion de chaînes » disponibles grâce à Perl dans des fonctions et procédures stockées. L'analyse de chaînes complexes se trouve facilitée par l'utilisation de Perl et des fonctions et structures de contrôle fournies dans PL/pgSQL.

Pour installer PL/Perl dans une base de données spécifique, utilisez `CREATE EXTENSION plperl`.

> **Astuce**
>
> Si un langage est installé dans `template1`, toutes les bases de données créées ultérieurement disposeront automatiquement de ce langage.

> **Note**
>
> Les utilisateurs des paquetages sources doivent explicitement autoriser la construction de PL/Perl pendant le processus d'installation (se référer à la Chapitre 17 pour plus d'informations). Les utilisateurs des paquetages binaires peuvent trouver PL/Perl dans un sous-paquetage séparé.

45.1. Fonctions et arguments PL/Perl

Pour créer une fonction dans le langage PL/Perl, utilisez la syntaxe standard CREATE FUNCTION :

```
CREATE FUNCTION nom_fonction (types-arguments) RETURNS
type-retour
-- attributs de fonction
AS $$
    # Corps de la fonction PL/Perl
$$ LANGUAGE plperl;
```

Le corps de la fonction est du code Perl normal. En fait, le code supplémentaire PL/Perl l'emballe dans une sous-routine Perl. Une fonction PL/Perl est appelée dans un contexte scalaire, il ne peut donc pas retourner une liste. Vous pouvez retourner des valeurs non scalaire par référence comme indiqué ci-dessous.

Dans une procédure PL/Perl, toute valeur de retour du code Perl est ignorée.

PL/Perl peut aussi être utilisé au sein de blocs de procédures anonymes avec l'ordre DO :

```
DO $$
    # PL/Perl code
$$ LANGUAGE plperl;
```

[1] https://www.perl.org

Un bloc de procédure anonyme ne prend pas d'arguments et toute valeur retournée est ignorée. Ceci, mis à part, il se comporte comme une fonction classique.

> **Note**
>
> L'utilisation de sous-routines nommées est dangereux en Perl, spécialement si elles font références à des variables lexicales dans la partie englobante. Comme une fonction PL/Perl est englobée dans une sous-routine, toute sous-routine nommée que vous y créez sera englobée. En général, il est bien plus sûr de créer des sous-routines anonymes que vous appellerez via un coderef. Pour de plus amples détails, voir les entrées `Variable "%s" will not stay shared` et `Variable "%s" is not available` dans le manuel perldiag, ou recherchez « perl nested named subroutine » sur internet.

La syntaxe de la commande `CREATE FUNCTION` requiert que le corps de la fonction soit écrit comme une constante de type chaîne. Il est habituellement plus agréable d'utiliser les guillemets dollar (voir la Section 4.1.2.4) pour cette constante. Si vous choisissez d'utiliser la syntaxe d'échappement des chaînes `E''`, vous devez doubler les marques de guillemets simples (`'`) et les antislashs (\) utilisés dans le corps de la fonction (voir la Section 4.1.2.1).

Les arguments et les résultats sont manipulés comme dans n'importe quelle routine Perl : les arguments sont passés au tableau `@_` et une valeur de retour est indiquée par `return` ou par la dernière expression évaluée dans la fonction.

Par exemple, une fonction retournant le plus grand de deux entiers peut être définie comme suit :

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

> **Note**
>
> Les arguments seront convertis de l'encodage de la base de données en UTF-8 pour être utilisé par PL/perl, puis converti de l'UTF-8 vers l'encodage de la base.

Si une valeur NULL en SQL est passée à une fonction, cet argument apparaîtra comme « undefined » en Perl. La fonction définie ci-dessus ne se comportera pas correctement avec des arguments NULL (en fait, tout se passera comme s'ils avaient été des zéros). Nous aurions pu ajouter `STRICT` à la définition de la fonction pour forcer PostgreSQL à faire quelque chose de plus raisonnable : si une valeur NULL est passée en argument, la fonction ne sera pas du tout appelée mais retournera automatiquement un résultat NULL. D'une autre façon, nous aurions pu vérifier dans le corps de la fonction la présence d'arguments NULL. Par exemple, supposons que nous voulions que `perl_max` avec un argument NULL et un autre non NULL retourne une valeur non NULL plutôt qu'une valeur NULL, on aurait écrit :

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($x, $y) = @_;
    if (not defined $x) {
        return undef if not defined $y;
        return $y;
    }
    return $x if not defined $y;
    return $x if $x > $y;
```

```
    return $y;
$$ LANGUAGE plperl;
```

Comme le montre l'exemple ci-dessus, passer une valeur NULL en SQL à une fonction en PL/Perl retourne une valeur non définie. Et ceci, que la fonction soit déclarée stricte ou non.

Dans un argument de fonction, tout ce qui n'est pas une référence est une chaîne qui est dans la représentation texte externe standard de PostgreSQL pour ce type de données. Dans le cas de types numériques ou texte, Perl fera ce qu'il faut et le programmeur n'aura pas à s'en soucier. Néanmoins, dans d'autres cas, l'argument aura besoin d'être converti dans une forme qui est plus utilisable que Perl. Par exemple, la fonction `decode_bytea` peut-être utilisée pour convertir un argument de type `bytea` en données binaires non échappées.

De façon similaire, les valeurs renvoyées à PostgreSQL doivent être dans le format textuel. Par exemple, la fonction `encode_bytea` peut être utilisée pour échapper des données binaires en retournant une valeur de type `bytea`.

Un cas particulièrement important concerne les valeurs booléennes. Comme indiqué à l'instant, le comportement par défaut des valeurs de type `bool` est qu'elles sont passées en tant que text à l'interpréteur Perl, donc soit `'t'` soit `'f'`. Ceci est problématique parce que Perl ne traite pas `'f'` comme un false ! Il est possible d'améliorer les choses en utilisant une « transformation » (voir CREATE TRANSFORM). Des transformations intéressantes sont fournies par l'extension `bool_plperl`. Pour l'utiliser, installez l'extension :

```
CREATE EXTENSION bool_plperl;  -- or bool_plperlu for PL/PerlU
```

Puis utilisez l'attribut de fonction `TRANSFORM` pour une fonction PL/Perl qui prend ou renvoie une donnée de type `bool`, par exemple :

```
CREATE FUNCTION perl_and(bool, bool) RETURNS bool
TRANSFORM FOR TYPE bool
AS $$
  my ($a, $b) = @_;
  return $a && $b;
$$ LANGUAGE plperl;
```

Quand cette transformation est appliquée, les arguments `bool` seront vues par Perl comme valant `1` ou vide, soit des valeurs true ou false propres. Si le résultat de la fonction renvoie le type `bool`, il sera true ou false suivant comment Perl évaluera la valeur retournée. Des transformations similaires sont aussi réalisées pour les arguments de requêtes de type booléen et les résultats des requêtes SPI réalisées à l'intérieur de la fonction (Section 45.3.1).

Perl peut renvoyer des tableaux PostgreSQL comme référence à des tableaux Perl. Voici un exemple :

```
CREATE OR REPLACE function renvoit_tableau()
RETURNS text[][] AS $$
    return [['a"b','c,d'],['e\\f','g']];
$$ LANGUAGE plperl;

select renvoit_tableau();
```

Perl utilise les tableaux PostgreSQL comme des objets `PostgreSQL::InServer::ARRAY`. Cet objet sera traité comme une référence de tableau ou comme une chaîne, permettant une compatibilité ascendante avec le code Perl écrit pour les versions de PostgreSQL antérieures à la 9.1. Par exemple :

```
CREATE OR REPLACE FUNCTION concat_array_elements(text[]) RETURNS
 TEXT AS $$
    my $arg = shift;
    my $result = "";
    return undef if (!defined $arg);

    # en tant que référence de tableau
    for (@$arg) {
        $result .= $_;
    }

    # en tant que chaîne
    $result .= $arg;

    return $result;
$$ LANGUAGE plperl;

SELECT concat_array_elements(ARRAY['PL','/','Perl']);
```

> **Note**
>
> Les tableaux multi-dimensionnels sont représentés comme des références à des tableaux de reference et de moindre dimension, d'une façon connue de chaque développeur Perl.

Les arguments de type composite sont passés à la fonction en tant que références d'un tableau de découpage, les clés du tableau de découpage étant les noms des attributs du type composé. Voici un exemple :

```
CREATE TABLE employe (
    nom text,
    basesalaire integer,
    bonus integer
);

CREATE FUNCTION empcomp(employe) RETURNS integer AS $$
    my ($emp) = @_;
    return $emp->{basesalaire} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT nom, empcomp(employe.*) FROM employe;
```

Une fonction PL/Perl peut renvoyer un résultat de type composite en utilisant la même approche : renvoyer une référence à un hachage qui a les attributs requis. Par exemple

```
      CREATE TYPE testligneperl AS (f1 integer, f2 text, f3 text);

      CREATE OR REPLACE FUNCTION perl_ligne() RETURNS
 test_ligne_perl AS $$
      return {f2 => 'hello', f1 => 1, f3 => 'world'};
      $$ LANGUAGE plperl;

      SELECT * FROM perl_row();
```

Toute colonne dans le type de données déclaré du résultat qui n'est pas présente dans le hachage sera renvoyée NULL.

De façon similaire, les arguments en sortie des procédures peuvent être renvoyés sous la format d'une référence hash :

```
CREATE PROCEDURE perl_triple(INOUT a integer, INOUT b integer) AS $
$
    my ($a, $b) = @_;
    return {a => $a * 3, b => $b * 3};
$$ LANGUAGE plperl;

CALL perl_triple(5, 10);
```

Les fonctions PL/Perl peuvent aussi renvoyer des ensembles de types scalaires ou composites. Habituellement, vous voulez renvoyer une ligne à la fois, à la fois pour améliorer le temps de démarrage et pour éviter d'allonger la queue de l'ensemble des résultats en mémoire. Vous pouvez faire ceci avec `return_next` comme indiqué ci-dessous. Notez qu'après le dernier `return_next`, vous devez placer soit `return` soit (encore mieux) `return undef`.

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF test_ligne_perl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
$$ LANGUAGE plperl;
```

Pour les petits ensembles de résultats, vous pouvez renvoyer une référence à un tableau contenant soit des scalaires, soit des références à des tableaux soit des références à des hachages de types simples, de types tableaux ou de types composites. Voici quelques exemples simples pour renvoyer l'ensemble complet du résultant en tant que référence de tableau :

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER
 AS $$
      return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testligneperl
 AS $$
return [
      { f1 => 1, f2 => 'Bonjour', f3 => 'Monde' },
      { f1 => 2, f2 => 'Bonjour', f3 => 'PostgreSQL' },
      { f1 => 3, f2 => 'Bonjour', f3 => 'PL/Perl' }
```

```
      ];
$$  LANGUAGE plperl;

SELECT * FROM perl_set();
```

Si vous souhaitez utiliser le pragma `strict` dans votre code, vous avez plusieurs options. Pour une utilisation temporaire globale vous pouvez positionner (`SET`) `plperl.use_strict` à « true ». Ce paramètre affectera les compilations suivantes de fonctions PL/Perl, mais pas les fonctions déjà compilées dans la session en cours. Pour une utilisation globale permanente, vous pouvez positionner `plperl.use_strict` à « true » dans le fichier `postgresql.conf`.

Pour une utilisation permanente dans des fonctions spécifiques, vous pouvez simplement placer:

```
use strict;
```

en haut du corps de la fonction.

Le pragma `feature` est aussi disponible avec `use` si votre version de Perl est 5.10.0 ou supérieur.

45.2. Valeurs en PL/Perl

Les valeurs des arguments fournis au code d'une fonction PL/Perl sont simplement les arguments d'entrée convertis en tant que texte (comme s'ils avaient été affichés par une commande `SELECT`). Inversement, les commandes `return` et `return_next` accepteront toute chaîne qui a un format d'entrée acceptable pour le type de retour déclaré de la fonction.

Si ce comportement n'est pas convenable pour une utilisation particulière, il peut être amélioré en ajoutant une transformation comme cela a déjà été illustré pour les valeurs de type `bool`. Plusieurs exemples de modules de transformation sont inclus dans la distribution PostgreSQL.

45.3. Fonctions incluses

45.3.1. Accès à la base de données depuis PL/Perl

L'accès à la base de données à l'intérieur de vos fonctions écrites en Perl peut se faire à partir des fonctions suivantes :

`spi_exec_query(`*`query`* `[, `*`limit`*`])`

`spi_exec_query` exécute une commande SQL et renvoie l'ensemble complet de la ligne comme une référence à un table de références hachées. Si *`limit`* est indiqué et est supérieur à zéro, alors `spi_exec_query` récupère au plus *`limit`* lignes, tout comme si la requête avait une clause `LIMIT`. Omettre *`limit`* ou indiquer 0 fait disparaître la limite de lignes.

Vous ne devez utiliser cette commande que lorsque vous savez que l'ensemble de résultat sera relativement petit. Voici un exemple d'une requête (commande `SELECT`) avec le nombre optionnel maximum de lignes :

```
$rv = spi_exec_query('SELECT * FROM ma_table', 5);
```

Ceci entrevoit cinq lignes au maximum de la table `ma_table`. Si `ma_table` a une colonne `ma_colonne`, vous obtenez la valeur de la ligne `$i` du résultat de cette façon :

```
$foo = $rv->{rows}[$i]->{ma_colonne};
```

Le nombre total des lignes renvoyées d'une requête `SELECT` peut être accédé de cette façon :

```
$nrows = $rv->{processed}
```

Voici un exemple en utilisant un type de commande différent :

```
$query = "INSERT INTO ma_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

Ensuite, vous pouvez accéder au statut de la commande (c'est-à-dire, `SPI_OK_INSERT`) de cette façon :

```
$res = $rv->{status};
```

Pour obtenir le nombre de lignes affectées, exécutez :

```
$nrows = $rv->{processed};
```

Voici un exemple complet :

```
CREATE TABLE test (
    i int,
    v varchar
    );

INSERT INTO test (i, v) VALUES (1, 'première ligne');
INSERT INTO test (i, v) VALUES (2, 'deuxième ligne');
INSERT INTO test (i, v) VALUES (3, 'troisième ligne');
INSERT INTO test (i, v) VALUES (4, 'immortel');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};
    foreach my $rn (0 .. $nrows - 1) {
        my $row = $rv->{rows}[$rn];
        $row->{i} += 200 if defined($row->{i});
        $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
        return_next($row);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

`spi_query(`*`command`*`)`
`spi_fetchrow(`*`cursor`*`)`
`spi_cursor_close(`*`cursor`*`)`

`spi_query` et `spi_fetchrow` fonctionnent ensemble comme une paire d'ensembles de lignes pouvant être assez importants ou pour les cas où vous souhaitez renvoyer les lignes dès qu'elles arrivent. `spi_fetchrow` fonctionne *seulement* avec `spi_query`. L'exemple suivant illustre comment vous les utilisez ensemble :

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF
 foo_type AS $$
    use Digest::MD5 qw(md5_hex);
    my $file = '/usr/share/dict/words';
    my $t = localtime;
    elog(NOTICE, "opening file $file at $t" );
    open my $fh, '<', $file # ooh, it's a file access!
```

```
            or elog(ERROR, "cannot open $file for reading: $!");
        my @words = <$fh>;
        close $fh;
        $t = localtime;
        elog(NOTICE, "closed file $file at $t");
        chomp(@words);
        my $row;
        my $sth = spi_query("SELECT * FROM generate_series(1,$_[0])
     AS b(a)");
        while (defined ($row = spi_fetchrow($sth))) {
            return_next({
                the_num => $row->{a},
                the_text => md5_hex($words[rand @words])
            });
        }
        return;
    $$ LANGUAGE plperlu;

    SELECT * from lotsa_md5(500);
```

Habituellement, `spi_fetchrow` devra être répété jusqu'à ce qu'il renvoie `undef`, indiquant qu'il n'y a plus de lignes à lire. Le curseur renvoyé par `spi_query` est automatiquement libéré quand `spi_fetchrow` renvoie `undef`. Si vous ne souhaitez pas lire toutes les lignes, appelez à la place `spi_cursor_close` pour libérer le curseur. Un échec ici résultera en des pertes mémoire.

```
spi_prepare(command, argument types)
spi_query_prepared(plan, arguments)
spi_exec_prepared(plan [, attributes], arguments)
spi_freeplan(plan)
```

`spi_prepare`, `spi_query_prepared`, `spi_exec_prepared` et `spi_freeplan` implémentent la même fonctionnalité, mais pour des requêtes préparées. `spi_prepare` accepte une chaîne pour la requête avec des arguments numérotés ($1, $2, etc) et une liste de chaînes indiquant le type des arguments :

```
$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name =
 $2', 'INTEGER', 'TEXT');
```

Une fois qu'un plan est préparé suite à un appel à `spi_prepare`, le plan peut être utilisé à la place de la requête, soit dans `spi_exec_prepared`, où le résultat est identique à celui renvoyé par `spi_exec_query`, soit dans `spi_query_prepared` qui renvoie un curseur exactement comme le fait `spi_query`, qui peut ensuite être passé à `spi_fetchrow`. Le deuxième paramètre, optionnel, de `spi_exec_prepared` est une référence hachée des attributs ; le seul attribut actuellement supporté est `limit`, qui configure le nombre maximum de lignes renvoyées par une requête. Omettre `limit` ou le configurer à zéro fait qu'il n'y a pas de limite de lignes.

L'avantage des requêtes préparées est que cela rend possible l'utilisation d'un plan préparé par plusieurs exécutions de la requête. Une fois que le plan n'est plus utile, il peut être libéré avec `spi_freeplan` :

```
CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
        $_SHARED{my_plan} = spi_prepare( 'SELECT (now() +
 $1)::date AS now', 'INTERVAL');
$$ LANGUAGE plperl;
```

```
CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS
 $$
        return spi_exec_prepared(
                $_SHARED{my_plan},
                $_[0]
        )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
        spi_freeplan( $_SHARED{my_plan});
        undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3
 days');
SELECT done();

  add_time  |  add_time  |  add_time
------------+------------+------------
 2005-12-10 | 2005-12-11 | 2005-12-12
```

Notez que l'indice du paramètre dans `spi_prepare` est défini via $1, $2, $3, etc, donc évitez de déclarer des chaînes de requêtes qui pourraient aisément amener des bogues difficiles à trouver et corriger.

Cet autre exemple illustre l'utilisation d'un paramètre optionnel avec `spi_exec_prepared` :

```
CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS
 address FROM generate_series(1,3) AS id;

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
        $_SHARED{plan} = spi_prepare('SELECT * FROM hosts WHERE
 address << $1', 'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts
 AS $$
        return spi_exec_prepared(
                $_SHARED{plan},
                {limit => 2},
                $_[0]
        )->{rows};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS
 $$
        spi_freeplan($_SHARED{plan});
        undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();

    query_hosts
```

```
-----------------
 (1,192.168.1.1)
 (2,192.168.1.2)
(2 rows)
```

`spi_commit()`
`spi_rollback()`

Valide ou annule la transaction en cours. Ceci peut seulement être appelé dans une procédure ou un code de bloc anonyme (commande `DO`) appelé au plus haut niveau. (Notez qu'il n'est pas possible d'exécuter les commandes SQL `COMMIT` ou `ROLLBACK` via `spi_exec_query` ou similaire. Cela doit se faire en utilisant ces fonctions.) À la fin d'une transaction, une nouvelle transaction est automatiquement démarrée, donc il n'y a pas de fonction séparée pour cela.

En voici un exemple :

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plperl
AS $$
foreach my $i (0..9) {
    spi_exec_query("INSERT INTO test1 (a) VALUES ($i)");
    if ($i % 2 == 0) {
        spi_commit();
    } else {
        spi_rollback();
    }
}
$$;

CALL transaction_test1();
```

45.3.2. Fonctions utiles en PL/Perl

`elog(`*`level`*`, `*`msg`*`)`

Produit un message de trace ou d'erreur. Les niveaux possibles sont `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING` et `ERROR`. `ERROR` lève une condition d'erreur ; si elle n'est pas récupérée par le code Perl l'entourant, l'erreur se propage à l'extérieur de la requête appelante, causant l'annulation de la transaction ou sous-transaction en cours. Ceci est en fait identique à la commande `die` de Perl. Les autres niveaux génèrent seulement des messages de niveaux de priorité différents. Le fait que les messages d'un niveau de priorité particulier soient rapportés au client, écrit dans les journaux du serveur, voire les deux, est contrôlé par les variables de configuration log_min_messages et client_min_messages. Voir le Chapitre 20 pour plus d'informations.

`quote_literal(`*`string`*`)`

Retourne la chaîne donnée convenablement placé entre simple guillemets pour être utilisée comme une chaîne littérale au sein d'une chaîne représentant un ordre SQL. Les simples guillemets et antislashs de la chaîne sont correctement doublés Notez que `quote_literal` retourne undef avec une entrée undef ; si l'argument peut être undef, `quote_nullable` est souvent plus approprié.

`quote_nullable(`*`string`*`)`

Retourne la chaîne donnée convenablement placé entre simple guillemets pour être utilisée comme une chaîne littérale au sein d'une chaîne représentant un ordre SQL. Si l'argument d'entrée

est undef, retourne la chaîne "NULL" sans simple guillemet. Les simples guillemets et antislashs de la chaîne sont correctement doublés

`quote_ident(`*`string`*`)`

Retourne la chaîne donnée convenablement placé entre guillemets pour être utilisée comme un identifiant au sein d'une chaîne représentant un ordre SQL. Les guillemets sont ajoutées seulement si cela est nécessaire (i.e. si la chaîne contient des caractères non-identifiant ou est en majuscule). Les guillemets de la chaîne seront convenablement doublés.

`decode_bytea(`*`string`*`)`

Retourne les données binaires non échappées représentées par le contenu de la chaîne donnée, qui doit être encodé au format `bytea`.

`encode_bytea(`*`string`*`)`

Retourne sous la forme d'un `bytea` le contenu binaire dans la chaîne passé en argument.

`encode_array_literal(`*`array`*`)`
`encode_array_literal(`*`array, delimiter`*`)`

Retourne le contenu de tableau passé par référence sous forme d'une chaîne littérale. (voir Section 8.15.2). Retourne la valeur de l'argument non altérée si ce n'est pas une référence à un tableau. Le délimiteur utilisé entre les éléments du tableau sous forme littérale sera par défaut "`, `" si aucun délimiteur n'est spécifié ou s'il est undef.

`encode_typed_literal(`*`value, typename`*`)`

Convertit une variable Perl en une valeur du type de données passé en second argument et renvoie une représentation de type chaîne pour cette valeur. Gère correctement les tableaux imbriqués et les valeurs de types composites.

`encode_array_constructor(`*`array`*`)`

Retourne le contenu de tableau passé par référence sous forme d'une chaîne permettant de construire un tableau en SQL. (voir Section 4.2.12). Chaque élément est entouré de simple guillemets par `quote_nullable`. Retourne la valeur de l'argument, entouré de simple guillemets par `quote_nullable`, si ce n'est pas une référence à un tableau.

`looks_like_number(`*`string`*`)`

Retourne une valeur vraie si le contenu de la chaîne passée ressemble à un nombre, selon l'interprétation de Perl, et faux dans le cas contraire. Retourne undef si undef est passé en argument. Tout espace en début et fin de chaîne sont ignorés. `Inf` et `Infinity` sont vu comme des nombres.

`is_array_ref(`*`argument`*`)`

Renvoie une valeur true si l'argument donné peut être traité comme une référence de tableau, c'est-à-dire si la référence de l'argument est ARRAY ou `PostgreSQL::InServer::ARRAY`. Renvoie false sinon.

45.4. Valeurs globales dans PL/Perl

Vous pouvez utiliser le hachage global %_SHARED pour stocker les données, incluant les références de code, entre les appels de fonction pour la durée de vie de la session en cours.

Voici un exemple simple pour des données partagées :

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS
 text AS $$
if ($_SHARED{$_[0]} = $_[1]) {
      return 'ok';
} else {
      return "Ne peux pas initialiser la variable partagée $_[0] à
 $_[1]";
}
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
      return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Bonjour, PL/Perl ! Comment va ?');
SELECT get_var('sample');
```

Voici un exemple légèrement plus compliqué utilisant une référence de code :

```
CREATE OR REPLACE FUNCTION ma_fonction() RETURNS void AS $$
$_SHARED{myquote} = sub {
      my $arg = shift;
      $arg =~ s/(['\\])/\\$1/g;
      return "'$arg'";
      };
$$ LANGUAGE plperl;

SELECT ma_fonction(); /* initialise la fonction */

/* Initialise une fonction qui utilise la fonction quote */

CREATE OR REPLACE FUNCTION utilise_quote(TEXT) RETURNS text AS $$
      my $text_to_quote = shift;
      my $qfunc = $_SHARED{myquote};
      return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

(Vous pouviez avoir remplacé le code ci-dessus avec la seule ligne `return $_SHARED{myquote}->($_[0]);` au prix d'une mauvaise lisibilité.)

Pour des raisons de sécurité, PL/Perl exécute des fonctions appelées par un rôle SQL dans un interpréteur Perl séparé pour ce rôle. Ceci empêche l'interférence accidentelle ou malicieuse d'un utilisateur avec le comportement des fonctions PL/Perl d'un autre utilisateur. Chaque interpréteur a sa propre valeur de la variable `%_SHARED` et des autres états globaux. Du coup, deux fonctions PL/Perl partageront la même valeur de `%_SHARED` si et seulement si elles sont exécutées par le même rôle SQL. Dans une application où une session seule exécute du code sous plusieurs rôles SQL (via des fonctions `SECURITY DEFINER`, l'utilisation de `SET ROLE`, etc), vous pouvez avoir besoin de mettre en place des étapes explicites pour vous assurer que les fonctions PL/Perl peuvent partager des données `%_SHARED`. Pour cela, assurez-vous que les fonctions qui doivent communiquer ont pour propriétaire le même utilisateur et marquez les comme `SECURITY DEFINER`. Bien sûr, vous devez faire attention à ce que ces fonctions ne puissent pas être utilisées pour faire des choses qu'elles ne sont pas sensées faire.

45.5. Niveaux de confiance de PL/Perl

Normalement, PL/Perl est installé en tant que langage de programmation de « confiance », de nom `plperl`. Durant cette installation, certaines commandes Perl sont désactivées pour préserver la sécurité. En général, les commandes qui interagissent avec l'environnement sont restreintes. Cela inclut

les commandes sur les descripteurs de fichiers, `require` et `use` (pour les modules externes). Il n'est pas possible d'accéder aux fonctions et variables internes du processus du serveur de base de données ou d'obtenir un accès au niveau du système d'exploitation avec les droits du processus serveur, tel qu'une fonction C peut le faire. Ainsi, n'importe quel utilisateur sans droits sur la base de données est autorisé à utiliser ce langage.

> **Avertissement**
>
> Le langage PL/Perl de confiance se base sur le module `Opcode` de Perl pour préserver la sécurité. Perl documente[2] que ce module n'est pas efficace pour le cas d'utilisation d'un PL/Perl de confiance. Si vos besoins en sécurité sont incompatibles avec l'incertitude de cet avertissement, considérez d'exécuter `REVOKE USAGE ON LANGUAGE plperl FROM PUBLIC`.

Voici l'exemple d'une fonction qui ne fonctionnera pas car les commandes système ne sont pas autorisées pour des raisons de sécurité :

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    my $tmpfile = "/tmp/badfile";
    open my $fh, '>', $tmpfile
        or elog(ERROR, qq{could not open the file "$tmpfile": $!});
    print $fh "Testing writing to a file\n";
    close $fh or elog(ERROR, qq{could not close the file
 "$tmpfile": $!});
    return 1;
$$ LANGUAGE plperl;
```

La création de cette fonction échouera car le validateur détectera l'utilisation par cette fonction d'une opération interdite.

Il est parfois souhaitable d'écrire des fonctions Perl qui ne sont pas restreintes. Par exemple, on peut souhaiter vouloir envoyer des courriers électroniques. Pour supporter ce cas de figure, PL/Perl peut aussi être installé comme un langage « douteux » (habituellement nommé PL/PerlU). Dans ce cas, la totalité du langage Perl est accessible. Lors de l'installation du langage, le nom du langage `plperlu` sélectionnera la version douteuse de PL/Perl.

Les auteurs des fonctions PL/PerlU doivent faire attention au fait que celles-ci ne puissent être utilisées pour faire quelque chose de non désiré car cela donnera la possibilité d'agir comme si l'on possédait les privilèges d'administrateur de la base de données. Il est à noter que le système de base de données ne permet qu'aux superutilisateurs de créer des fonctions dans un langage douteux.

Si la fonction ci-dessus a été créée par un superutilisateur en utilisant le langage `plperlu`, l'exécution de celle-ci réussira.

De la même façon, les blocs de procédure anonymes écris en perl peuvent utiliser les opérations restreintes si le langage est spécifié comme `plperlu` plutôt que `plperl`, mais l'appelant doit être un superutilisateur.

> **Note**
>
> Bien que les fonctions PL/Perl s'exécutent dans un interpréteur Perl séparé pour chaque rôle SQL, toutes les fonctions PL/PerlU exécutées dans la même session utilisent un seul interpréteur Perl (qui n'est pas un de ceux utilisés par les fonctions PL/Perl). Ceci permet aux

[2] https://perldoc.perl.org/Opcode#WARNING

fonctions PL/PerlU de partager librement des données, mais aucune communication ne peut survenir entre des fonctions PL/Perl et PL/PerlU.

Note

Perl ne peut pas supporter plusieurs interpréteurs à l'intérieur d'un seul processus sauf s'il a été construit avec les bonnes options, soit `usemultiplicity` soit `useithreads`. (`usemultiplicity` est préféré sauf si vous avez besoin d'utiliser des threads. Pour plus de détails, voir la page de manuel de perlembed.) Si PL/Perl est utilisé avec une copie de Perl qui n'a pas été construite de cette façon, alors seul un interpréteur Perl par session sera disponible, et donc une session ne pourra exécuter soit que des fonctions PL/PerlU, soit que des fonctions PL/Perl qui sont appelées par le même rôle SQL.

45.6. Triggers PL/Perl

PL/Perl peut être utilisé pour écrire des fonctions pour triggers. Dans une fonction trigger, la référence hachée `$_TD` contient des informations sur l'événement du trigger en cours. `$_TD` est une variable globale qui obtient une valeur locale séparée à chaque appel du trigger. Les champs de la référence de hachage `$_TD` sont :

`$_TD->{new}{foo}`

Valeur `NEW` de la colonne `foo`

`$_TD->{old}{foo}`

Valeur `OLD` de la colonne `foo`

`$_TD->{name}`

Nom du trigger appelé

`$_TD->{event}`

Événement du trigger : `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `INSTEAD OF` ou `UNKNOWN`

`$_TD->{when}`

Quand le trigger a été appelé : `BEFORE` (avant), `AFTER` (après) ou `UNKNOWN` (inconnu)

`$_TD->{level}`

Le niveau du trigger : `ROW` (ligne), `STATEMENT` (instruction) ou `UNKNOWN` (inconnu)

`$_TD->{relid}`

L'OID de la table sur lequel le trigger a été exécuté

`$_TD->{table_name}`

Nom de la table sur lequel le trigger a été exécuté

`$_TD->{relname}`

Nom de la table sur lequel le trigger a été exécuté. Elle est obsolète et pourrait être supprimée dans une prochaine version. Utilisez $_TD->{table_name} à la place.

`$_TD->{table_schema}`

Nom du schéma sur lequel le trigger a été exécuté.

`$_TD->{argc}`

Nombre d'arguments de la fonction trigger

`@{$_TD->{args}}`

Arguments de la fonction trigger. N'existe pas si `$_TD->{argc}` vaut 0.

Les triggers niveau ligne peuvent renvoyer un des éléments suivants :

`return;`

Exécute l'opération

`"SKIP"`

N'exécute pas l'opération

`"MODIFY"`

Indique que la ligne `NEW` a été modifiée par la fonction trigger

Voici un exemple d'une fonction trigger illustrant certains points ci-dessus :

```
CREATE TABLE test (
    i int,
    v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
        return "SKIP";    # passe la commande INSERT/UPDATE
    } elsif ($_TD->{new}{v} ne "immortal") {
        $_TD->{new}{v} .= "(modified by trigger)";
        return "MODIFY";  # modifie la ligne et exécute la commande
 INSERT/UPDATE
    } else {
        return;           # exécute la commande INSERT/UPDATE
    }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE FUNCTION valid_id();
```

45.7. Triggers sur événements avec PL/Perl

PL/Perl peut être utilisé pour écrire des fonctions trigger sur événement. Dans ce type de fonctions, la référence hachée `$_TD` contient des informations sur l'événement du trigger. `$_TD` est une variable globale, qui obtient une valeur locale séparée à chaque invocation du trigger. Les champs disponibles via `$_TD` sont :

`$_TD->{event}`

Le nom de l'événement pour lequel le trigger a été déclenché.

`$_TD->{tag}`

La balise de la commande pour laquelle le trigger a été déclenché.

Le code de retour de la fonction trigger est ignoré.

Voici un exemple de fonction trigger sur événement, illustrant certaines des informations ci-dessus :

```
CREATE OR REPLACE FUNCTION perlsnitch() RETURNS event_trigger AS $$
  elog(NOTICE, "perlsnitch: " . $_TD->{event} . " " . $_TD->{tag} .
 " ");
$$ LANGUAGE plperl;

CREATE EVENT TRIGGER perl_a_snitch
    ON ddl_command_start
    EXECUTE FUNCTION perlsnitch();
```

45.8. PL/Perl sous le capot

45.8.1. Configuration

Cette section liste les paramètres de configuration de PL/Perl.

`plperl.on_init` (`string`)

Spécifie un code perl à exécuter lorsque l'interpréteur Perl est initialisé pour la première fois et avant qu'il soit spécialisé pour être utilisé par `plperl` ou `plperlu`. Les fonction SPI ne sont pas disponible lorsque ce code est exécuté. Si le code lève une erreur, il interrompra l'initialisation de l'interpréteur et la propagera à la requête originale, provoquant ainsi l'annulation de la transaction ou sous-transaction courante.

Le code Perl est limité à une seule ligne. Un code plus long peut être placé dans un module et chargé par `on_init`. Exemples:

```
plperl.on_init = 'require "plperlinit.pl"'
plperl.on_init = 'use lib "/my/app"; use MyApp::PgInit;'
```

Tous les modules chargés par `plperl.on_init`, directement ou indirectement, seront disponibles depuis `plperl`. Cela entraîne un problème de sécurité potentiel. Pour consulter la liste des modules chargés, vous pouvez utiliser :

```
DO 'elog(WARNING, join ", ", sort keys %INC)' LANGUAGE plperl;
```

L'initialisation aura lieu au sein du postmaster si la librairie `plperl` est incluse dans le paramètre shared_preload_libraries), auquel cas une plus grande attention doit être portée au risque de déstabiliser ce dernier. La raison principale d'utilisation de cette fonctionnalité est que les modules Perl chargés par `plperl.on_init` doivent être chargés seulement au démarrage de postmaster, et seront instantanément disponible sans surcoût dans chaque session individuelle. Néanmoins, gardez en tête que la surcharge est seulement évitée pour le premier interpréteur Perl utilisé par une session de base de données -- soit PL/PerlU, soit PL/Perl pour le premier rôle SQL qui appelle une fonction PL/Perl. Tout interpréteur Perl supplémentaire créé dans une session de base aura à exécuter `plperl.on_init`. De plus, sur Windows, il n'y aura aucun gain avec

le préchargement car l'interpréteur Perl créé par le processus postmaster ne se propage pas aux processus fils.

Ce paramètre ne peut être positionné que dans le fichier `postgresql.conf` ou depuis la ligne de commande de démarrage du serveur.

`plperl.on_plperl_init` (`string`)
`plperl.on_plperlu_init` (`string`)

Ces paramètres spécifient le code Perl à exécuter quand un interpréteur Perl est spécialisé respectivement pour `plperl` ou `plperlu`. Ceci n'arrivera que quand une fonction PL/Perl ou PL/PerlU est exécutée la première fois dans une session de base de données, ou quand un interpréteur supplémentaire doit être créé parce que l'autre langage a été appelé ou parce qu'une fonction PL/Perl a été appelée par un nouveau rôle SQL. Ceci suit toute initialisation réalisée par `plperl.on_init`. Les fonctions SPI ne sont pas disponibles quand ce code est exécuté. Le code Perl dans `plperl.on_plperl_init` est exécuté après le « verrouillage » de l'interpréteur, et donc il peut seulement réaliser des opérations de confiance.

Si le code lève une erreur, il interrompra l'initialisation et la propagera à la requête originale, provoquant ainsi l'annulation de la transaction ou sous-transaction courante. Toute action déjà réalisée dans Perl ne sera pas défaite ; néanmoins, cet interpréteur ne sera plus utilisé de nouveau. Si le langage est utilisé de nouveau, l'initialisation sera tentée de nouveau avec un nouvel interpréteur Perl.

Seuls les superutilisateurs peuvent modifier ces paramètres. Bien que ces paramètres peuvent être modifiés dans une session, de tels changements n'affecteront pas les interpréteurs Perl qui ont déjà été utilisés pour exécuter des fonctions.

`plperl.use_strict` (`boolean`)

Lorsqu'il est positionné à « true », les compilations des fonction PL/Perl suivantes auront le pragma `strict` activé. Ce paramètre n'affecte pas les fonctions déjà compilées au sein de la session courante.

45.8.2. Limitations et fonctionnalités absentes

Les fonctionnalités suivantes ne sont actuellement pas implémentées dans PL/Perl, mais peuvent faire l'objet de contributions généreuses de votre part.

- Les fonctions PL/Perl ne peuvent pas s'appeler entre elles.
- SPI n'est pas complètement implémenté.
- Si vous récupérez des ensembles de données très importants en utilisant `spi_exec_query`, vous devez être conscient qu'ils iront tous en mémoire. Vous pouvez l'éviter en utilisant `spi_query`/`spi_fetchrow` comme montré précédemment.

  Un problème similaire survient si une fonction renvoyant un ensemble passe un gros ensemble de lignes à PostgreSQL via `return`. Vous pouvez l'éviter aussi en utilisant à la place `return_next` pour chaque ligne renvoyée, comme indiqué précédemment.
- Lorsqu'une session se termine normalement, et pas à cause d'une erreur fatale, tous les blocs `END` qui ont été définis sont exécutés. Actuellement, aucune autre action n'est réalisée. Spécifiquement, les descripteurs de fichiers ne sont pas vidés automatiquement et les objets ne sont pas détruits automatiquement.

Chapitre 46. PL/Python -- Langage de procédures Python

Le langage de procédures PL/Python permet l'écriture de fonctions et de procédures PostgreSQL avec le langage Python[1] (mais voir aussi Section 46.10).

Pour installer PL/Python dans une base de données particulière, utilisez `CREATE EXTENSION plpython3u`.

> **Astuce**
>
> Si un langage est installé dans `template1`, toutes les bases nouvellement créées se verront installées ce langage automatiquement.

PL/Python est seulement disponible en tant que langage « sans confiance », ceci signifiant qu'il n'offre aucun moyen de restreindre ce que les utilisateurs en font). Il a donc été renommé en `plpython3u`. La variante de confiance `plpython` pourrait être de nouveau disponible dans le futur, si un nouveau mécanisme sécurisé d'exécution est développé dans Python. Le codeur d'une fonction dans PL/Python sans confiance doit faire attention à ce que cette fonction ne puisse pas être utilisée pour réaliser quelque chose qui n'est pas prévue car il sera possible de faire tout ce que peut faire un utilisateur connecté en tant qu'administrateur de la base de données. Seuls les superutilisateurs peuvent créer des fonctions dans des langages sans confiance comme `plpython3u`.

> **Note**
>
> Les utilisateurs des paquets sources doivent activer spécifiquement la construction de PL/Python lors des étapes d'installation (référez-vous aux instructions d'installation pour plus d'informations). Les utilisateurs de paquets binaires pourront trouver PL/Python dans un paquet séparé.

46.1. Fonctions PL/Python

Les fonctions PL/Python sont déclarées via la syntaxe standard CREATE FUNCTION :

```
CREATE FUNCTION nom_fonction (liste-arguments)
  RETURNS return-type
AS $$
  # corps de la fonction PL/Python
$$ LANGUAGE plpython3u;
```

Le corps d'une fonction est tout simplement un script Python. Quand la fonction est appelée, ses arguments sont passés au script Python comme des éléments de la liste `args` ; les arguments nommés sont en plus passés comme des variables ordinaires. L'utilisation des arguments nommés est beaucoup plus lisible. Le résultat est renvoyé par le code Python de la façon habituelle, avec `return` ou `yield` (dans le cas d'une instruction avec un ensemble de résultats). Si vous ne fournissez pas une valeur de retour, Python renvoie la valeur par défaut `None`. PL/Python traduit la valeur `None` de Python comme une valeur NULL SQL. Dans une procédure, le résultat d'un code Python doit être `None` (typiquement réalisé en terminant la procédure sans instruction `return` ou en utilisant une instruction `return` sans argument) ; sinon une erreur sera levée.

[1] https://www.python.org

Par exemple, une fonction renvoyant le plus grand de deux entiers peut être définie ainsi :

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpython3u;
```

Le code Python donné comme corps de la définition de fonction est transformé en fonction Python. Par exemple, le code ci-dessus devient :

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

en supposant que 23456 est l'OID affecté à la fonction par PostgreSQL.

Les arguments sont définis comme des variables globales. Conséquence subtile des règles sur la portée de variables dans Python, il n'est pas possible de réaffecter une variable à l'intérieur d'une fonction en conservant son nom, sauf si elle est préalablement déclarée comme globale à l'intérieur du bloc. Ainsi, l'exemple suivant ne fonctionnera pas :

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip()  # error
  return x
$$ LANGUAGE plpython3u;
```

car affecter la variable `x` la transforme en variable locale pour ce bloc et que, par conséquent, la variable `x` de l'expression de droite fait référence à une variable locale `x` non encore définie, et non pas au paramètre de la fonction PL/Python. L'utilisation du mot-clé `global` permet de résoudre le problème :

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip()  # ok now
  return x
$$ LANGUAGE plpython3u;
```

Cependant, il vaut mieux ne pas trop s'appuyer sur ce détail d'implémentation de PL/Python. Il est préférable de traiter les paramètres de fonction comme étant en lecture seule.

46.2. Valeur des données avec PL/Python

De manière générale, le but de PL/Python est de fournir une relation « naturelle » entre PostgreSQL et le monde Python. Ces règles relationelles sont décrites ci-dessous.

46.2.1. Type de données

Quand une procédure stockée PL/python est appelée, les paramètres de la fonction sont convertis de leur type de données PostgreSQL vers un type correspondant en Python :

- Le type PostgreSQL `boolean` est converti vers le type Python `bool`.
- Les types PostgreSQL `smallint`, `int`, `bigint` et `oid` sont convertis vers le type Python `int`.
- Les types PostgreSQL `real` et `double` sont convertis vers le type Python `float`.
- Le type PostgreSQL `numeric` est converti vers le type Python `Decimal`. Ce type est importé à partir du paquet `cdecimal` s'il est disponible. Dans le cas contraire, `decimal.Decimal` est utilisé à partir de la bibliothèque standard. `cdecimal` est bien plus performant que `decimal`. Néanmoins, avec Python 3.3 et les versions ultérieures, `cdecimal` a été intégré dans la bibliothèque standard sous le nom de `decimal`, donc la différence n'est plus valide.
- Le type PostgreSQL `bytea` est converti vers le type Python `bytes`.
- Tous les autres types de données, incluant les types chaînes de caractères de PostgreSQL, sont convertis vers un type Python `str` (en Unicode comme toutes les chaînes Python).
- Pour les données non scalaires, voir ci-dessous.

Quand une fonction PL/python renvoie des données, la valeur de retour est convertie en type de données PostgreSQL comme suit:

- Quand le type de la valeur PostgreSQL renvoyée est `boolean`, la valeur de retour sera évaluée en fonction des règles *Python*. Ainsi, les 0 et les chaines vides sont fausses, mais la valeur `'f'` est vraie.
- Quand le type de retour PostgreSQL est `bytea`, la valeur de retour sera convertie vers le type Python `bytes` en utilisant les fonctions internes de Python, et le résultat sera converti de nouveau en `bytea`.
- Pour tous les autres types de données renvoyées, la valeur de retour est convertie en une chaîne de caractère en utilisant la fonction Python interne `str`, et le résultat est passé à la fonction d'entrée du type de données PostgreSQL. (si la valeur Python est un flottant, il est converti en utilisant la fonction interne `repr` au lieu de `str`, pour éviter la perte de précision.)

  Les chaînes sont automatiquement converties dans l'encodage du serveur PostgreSQL quand elles lui sont passées à PostgreSQL.
- Pour les données non-scalaires, voir ci-dessous.

Notez que les erreurs logiques entre le type de retour déclaré dans PostgreSQL et le type de l'objet Python renvoyé ne sont pas détectées. La valeur sera convertie dans tous les cas.

46.2.2. Null, None

Si une valeur SQL NULL est passée à une fonction, la valeur de l'argument apparaîtra comme `None` au niveau de Python. Par exemple, la définition de la fonction `pymax` indiquée dans Section 46.1 renverra la mauvaise réponse pour des entrées NULL. Nous pouvons jouer `STRICT` à la définition de la fonction pour faire en sorte que PostgreSQL fasse quelque-chose de plus raisonnable : si une valeur NULL est passée, la fonction ne sera pas appelée du tout mais renverra juste un résultat NULL automatiquement. Sinon, vous pouvez vérifier les entrées NULL dans le corps de la fonction :

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
```

```
  if a > b:
    return a
  return b
$$ LANGUAGE plpython3u;
```

Comme montré ci-dessus, pour renvoyer une valeur SQL NULL à partir d'une fonction PL/Python, renvoyez la valeur `None`. Ceci peut se faire que la fonction soit stricte ou non.

46.2.3. Tableaux, Listes

Les valeurs de type tableaux SQL sont passées via PL/Python comme des listes Python. Pour renvoyer une valeur de type tableau SQL par une fonction PL/Python, renvoyez une liste Python :

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return [1, 2, 3, 4, 5]
$$ LANGUAGE plpython3u;

SELECT return_arr();
 return_arr
-------------
 {1,2,3,4,5}
(1 row)
```

Les tableaux multi-dimensionnels sont passés dans PL/Python en tant que listes Python imbriquées. Un tableau à 2 dimensions est une liste de liste, par exemple. Quand une fonction PL/Python renvoie un tableau SQL multi-dimensionnel, les listes internes doivent avoir la même taille à chaque niveau. Par exemple :

```
CREATE FUNCTION test_type_conversion_array_int4(x int4[]) RETURNS
 int4[] AS $$
plpy.info(x, type(x))
return x
$$ LANGUAGE plpython3u;

SELECT * FROM test_type_conversion_array_int4(ARRAY[[1,2,3],
[4,5,6]]);
INFO:  ([[1, 2, 3], [4, 5, 6]], <type 'list'>)
 test_type_conversion_array_int4
---------------------------------
 {{1,2,3},{4,5,6}}
(1 row)
```

Les autres séquences Python, comme les tuples, sont également acceptées pour compatibilité descendante avec les versions 9.6 et inférieures de PostgreSQL, quand les tableaux multi-dimensionnels n'étaient pas supportés. Cependant, ils sont toujours traités comme des tableaux à une dimension, car ils sont ambigus avec les types composites. Pour la même raison, quand un type composite est utilisé dans un tableau multi-dimensionnel, il doit être représenté par un tuple, plutôt que par une liste.

Notez que, avec Python, les chaînes sont des séquences, ce qui peut avoir des effets indésirables qui peuvent être familiers aux codeurs Python :

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpython3u;

SELECT return_str_arr();
 return_str_arr
----------------
 {h,e,l,l,o}
(1 row)
```

46.2.4. Types composites

Les arguments de type composite sont passés à la fonction via une correspondance Python. Les noms d'élément de la correspondance sont les noms d'attribut du type composite. Si un attribut a une valeur NULL dans la ligne traitée; il a la valeur NULL dans sa correspondance. Voici un exemple :

```
CREATE TABLE employe (
  nom text,
  salaire integer,
  age integer
);

CREATE FUNCTION trop_paye (e employe)
  RETURNS boolean
AS $$
  if e["salaire"] > 200000:
    return True
  if (e["age"] < 30) and (e["salaire"] > 100000):
    return True
  return False
$$ LANGUAGE plpython3u;
```

Il existe plusieurs façons de renvoyer une ligne ou des types composites à partir d'une fonction Python. Les exemples suivants supposent que nous avons :

```
CREATE TABLE valeur_nommee (
  nom   text,
  valeur  integer
);
```

ou

```
CREATE TYPE valeur_nommee AS (
  nom   text,
  valeur  integer
);
```

Une valeur composite peut être renvoyée comme :

Un type séquence (ligne ou liste), mais pas un ensemble parce que ce n'est pas indexable

> Les objets séquences renvoyés doivent avoir le même nombre d'éléments que le type composite a de champs. L'élément d'index 0 est affecté au premier champ du type composite, 1 au second et ainsi de suite. Par exemple :

```
CREATE FUNCTION cree_paire (nom text, valeur integer)
  RETURNS valeur_nommee
AS $$
  return ( nom, valeur )
  # ou autrement, en tant que liste : return [ nom, valeur ]
$$ LANGUAGE plpython3u;
```

Pour renvoyer NULL dans une colonne, insérez `None` à la position correspondante.

Quand un tableau de types composites est retourné, il ne peut pas être retourné comme une liste, car il est ambigu de savoir si la liste Python représente un type composite ou une autre dimension de tableau.

Correspondance (dictionnaire)

La valeur de chaque colonne du type résultat est récupérée à partir de la correspondance avec le nom de colonne comme clé. Exemple :

```
CREATE FUNCTION cree_paire (nom text, valeur integer)
  RETURNS valeur_nommee
AS $$
  return { "nom": nom, "valeur": valeur }
$$ LANGUAGE plpython3u;
```

Des paires clés/valeurs supplémentaires du dictionnaire sont ignorées. Les clés manquantes sont traitées comme des erreurs. Pour renvoyer NULL comme une colonne, insérez `None` avec le nom de la colonne correspondante comme clé.

Objet (tout objet fournissant la méthode `__getattr__`)

Ceci fonctionne de la même façon qu'une correspondance. Exemple :

```
CREATE FUNCTION cree_paire (nom text, valeur integer)
  RETURNS valeur_nommee
AS $$
  class valeur_nommee:
    def __init__ (self, n, v):
      self.nom = n
      self.valeur = v
  return valeur_nommee(nom, valeur)

  # ou simplement
  class nv: pass
  nv.nom = nom
  nv.valeur = valeur
  return nv
$$ LANGUAGE plpython3u;
```

Les fonctions ayant des paramètres `OUT` sont aussi supportées. Par exemple :

```
CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
return (1, 2)
$$ LANGUAGE plpython3u;

SELECT * FROM multiout_simple();
```

Les paramètres en sortie de procédures sont renvoyés de la même façon. Par exemple :

```
CREATE PROCEDURE python_triple(INOUT a integer, INOUT b integer) AS
 $$
return (a * 3, b * 3)
$$ LANGUAGE plpython3u;

CALL python_triple(5, 10);
```

46.2.5. Fonctions renvoyant des ensembles

Une fonction PL/Python peut aussi renvoyer des ensembles scalaires ou des types composites. Il existe plusieurs façon de faire ceci parce que l'objet renvoyé est transformé en interne en itérateur. Les exemples suivants supposent que nous avons le type composite :

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

Un résultat ensemble peut être renvoyé à partir de :

Un type séquence (ligne, liste, ensemble)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # renvoie la ligne contenant les listes en tant que types
 composites
  # toutes les autres combinaisons fonctionnent aussi
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/
Python" ] )
$$ LANGUAGE plpython3u;
```

L'itérateur (tout objet fournissant les méthodes `__iter__` et `__next__`)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def __next__ (self):
      self.ndx += 1
      if self.ndx == len(self.who):
        raise StopIteration
      return ( self.how, self.who[self.ndx] )

  return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
```

```
$$ LANGUAGE plpython3u;
```

Le générateur (`yield`)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  for who in [ "World", "PostgreSQL", "PL/Python" ]:
    yield ( how, who )
$$ LANGUAGE plpython3u;
```

Les fonctions renvoyant des ensembles et ayant des paramètres `OUT` (en utilisant `RETURNS SETOF record`) sont aussi supportées. Par exemple :

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT
 integer) RETURNS SETOF record AS $$
return [(1, 2)] * n
$$ LANGUAGE plpython3u;

SELECT * FROM multiout_simple_setof(3);
```

46.3. Partage de données

Le dictionnaire global `SD` est disponible pour stocker des données privées entre les appels répétés à la même fonction. Cette variable est une donnée statique privée. Le dictionnaire global `GD` est une donnée publique disponible pour toutes les fonctions Python à l'intérieur d'une session. À utiliser avec précaution.

Chaque fonction obtient son propre environnement d'exécution dans l'interpréteur Python, de façon à ce que les données globales et les arguments de fonction provenant de `ma_fonction` ne soient pas disponibles depuis `ma_fonction2`. L'exception concerne les données du dictionnaire `GD` comme indiqué ci-dessus.

46.4. Blocs de code anonymes

PL/Python accepte aussi les blocs de code anonymes appelés avec l'instruction DO :

```
DO $$
    # Code PL/Python
$$ LANGUAGE plpython3u;
```

Un bloc de code anonyme ne reçoit aucun argument et, quelque soit la valeur renvoyée, elle est ignorée. Sinon, ce bloc se comporte exactement comme n'importe quelle fonction.

46.5. Fonctions de triggers

Quand une fonction est utilisée par un trigger, le dictionnaire `TD` contient les valeurs relatives au trigger :

`TD["event"]`

contient l'événement sous la forme d'une chaîne : `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`.

`TD["when"]`

contient une chaîne valant soit `BEFORE`, soit `AFTER` soit `INSTEAD OF`.

`TD["level"]`

contient une chaîne valant soit `ROW` soit `STATEMENT`.

`TD["new"]`
`TD["old"]`

pour un trigger au niveau ligne, ces champs contiennent les lignes du trigger, l'ancienne version et la nouvelle version ; les deux champs ne sont pas forcément disponibles, ceci dépendant de l'événement qui a déclenché le trigger

`TD["name"]`

contient le nom du trigger.

`TD["table_name"]`

contient le nom de la table sur laquelle le trigger a été déclenché

`TD["table_schema"]`

contient le schéma de la table sur laquelle le trigger a été déclenché

`TD["relid"]`

contient l'OID de la table sur laquelle le trigger a été déclenché

`TD["args"]`

si la commande `CREATE TRIGGER` comprend des arguments, ils sont disponibles dans les variables allant de `TD["args"][0]` à `TD["args"][`*n*`-1]`.

Si `TD["when"]` vaut `BEFORE` ou `INSTEAD OF` et si `TD["level"]` vaut `ROW`, vous pourriez renvoyer `None` ou `"OK"` à partir de la fonction Python pour indiquer que la ligne n'est pas modifiée, `"SKIP"` pour annuler l'événement ou si `TD["event"]` vaut `INSERT` ou `UPDATE`, vous pouvez renvoyer `"MODIFY"` pour indiquer que vous avez modifié la ligne. Sinon la valeur de retour est ignorée.

46.6. Accès à la base de données

Le module du langage PL/Python importe automatiquement un module Python appelé `plpy`. Les fonctions et constantes de ce module vous sont accessibles dans le code Python via `plpy.`*`foo`*.

46.6.1. Fonctions d'accès à la base de données

Le module `plpy` fournit plusieurs fonctions pour exécuter des commandes sur la base de données :

`plpy.execute(`*`query`* `[,` *`limit`*`])`

L'appel à `plpy.execute` avec une chaîne pour la requête et une limite de ligne optionnelle permet d'exécuter la requête et de retourner le résultat dans un objet résultant.

Si *`limit`* est indiqué et supérieur à zéro, alors `plpy.execute` récupère au plus *`limit`* lignes, tout comme si la requête incluait une clause `LIMIT`. Omettre *`limit`* ou le configurer à zéro fait qu'il n'y a pas de limite de lignes.

L'objet résultant émule une liste ou un objet dictionnaire. L'objet résultant peut être accédé par le numéro de ligne et le nom de colonne. Par exemple :

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

retourne jusqu'à 5 lignes de `my_table`. Si `my_table` possède une colonne `my_column`, elle pourra être accédée ainsi :

```
foo = rv[i]["my_column"]
```

Le nombre de lignes retournées peut être obtenu en utilisant la fonction intégrée `len`.

L'objet résultant contient ces méthodes additionnelles :

`nrows()`

Retourne le nombre de lignes traitées par cette commande. Notez que cela n'est pas nécessairement identique au nombre de lignes retournées. Par exemple, une commande `UPDATE` fixera cette valeur mais ne retournera aucune ligne (sauf si `RETURNING` est utilisé).

`status()`

La valeur retournée par `SPI_execute()`.

`colnames()`
`coltypes()`
`coltypmods()`

Retourne respectivement une liste de noms de colonne, une liste de type OID de colonne et une liste de type de modifieurs spécifiques à un type pour les colonnes.

Ces méthodes lèvent une exception quand elles sont appelées sur un objet résultant d'une commande n'ayant pas produit d'ensemble de résultat, par ex, `UPDATE` sans `RETURNING`, ou `DROP TABLE`. Il est cependant normal d'utiliser ces méthodes sur un ensemble de résultat ne contenant aucune ligne.

`__str__()`

La méthode standard `__str__` est définie pour qu'il soit possible de débugger les résultats de l'exécution d'une requête en utilisant `plpy.debug(rv)`.

L'objet résultant peut être modifié.

Notez que l'appel à `plpy.execute` provoquera la lecture de tout l'ensemble de résultat en mémoire. N'utilisez cette fonction que lorsque vous êtes surs que l'ensemble de résultat sera relativement petit. Si vous ne voulez pas risquer une utilisation excessive de mémoire pour récupérer de gros ensembles, préférez `plpy.cursor` à `plpy.execute`.

`plpy.prepare(`*`query`* `[, `*`argtypes`*`])`
`plpy.execute(`*`plan`* `[, `*`arguments`* `[, `*`limit`*`]])`

`plpy.prepare` prépare le plan d'exécution pour une requête. Il faut l'appeler avec une chaîne contenant la requête et une liste de types de paramètres, si vous avez des références à des paramètres dans cette requête. Par exemple :

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE
 first_name = $1", ["text"])
```

`text` est le type de variable qui sera passé à `$1`. Le second paramètre est optionel si vous ne voulez pas fournir de paramètre à la requête.

Après avoir préparé une requête, il faut utiliser une variante de la fonction `plpy.execute` pour l'exécuter :

```
rv = plpy.execute(plan, ["name"], 5)
```

Il faut fournir le plan comme premier argument (à la place de la chaîne), et une liste de valeurs à substituer dans la requête comme second argument. Le deuxième argument est optionnel si la requête n'attend pas de paramètre. Le troisième argument est la limite de ligne optionnelle comme auparavant.

De manière alternative, vous pouvez appeler la méthode `execute` sur l'objet plan :

```
rv = plan.execute(["name"], 5)
```

Les paramètres de requête ainsi que les champs des lignes de résultat sont converties entre les types de données de PostgreSQL et de Python comme décrit dans Section 46.2.

Quand un plan est préparé en utilisant le module PL/Python, il est automatiquement sauvegardé. Voir la documentation de SPI (Chapitre 47) pour une description de ce que cela signifie. Afin d'utiliser efficacement ces appels de fonction, il faut utiliser un des dictionnaires de stockage persistant `SD` ou `GD` (voir Section 46.3). Par exemple :

```
 CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    if "plan" in SD:
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
     # reste de la fonction
 $$ LANGUAGE plpython3u;
```

```
plpy.cursor(query)
plpy.cursor(plan [, arguments])
```

La fonction `plpy.cursor` accepte les mêmes arguments que `plpy.execute` (à l'exception de la limite de lignes) et retourne un objet curseur, qui permet de traiter de gros ensembles de résultats en plus petits morceaux. Comme avec `plpy.execute`, une chaîne de caractère ou un objet plan accompagné d'une liste d'arguments peuvent être utilisés, ou la fonction `cursor` peut être appelée comme une méthode de l'objet plan.

L'objet curseur fournit une méthode `fetch` qui requiert en entrée un paramètre entier et retourne un objet résultat. À chaque appel de `fetch`, l'objet retourné contiendra la prochaine série de lignes, mais jamais plus que la valeur passée en paramètre. Une fois que toutes les lignes ont été épuisées, `fetch` se met à retourner des objets résultat vides. Les objets curseurs fournissent également une interface d'itérateur[2], fournissant les lignes une par une jusqu'à épuisement. Les données récupérées de cette façon ne sont pas retournées dans des objets résultat, mais plutôt dans des dictionnaires, chacun correspondant à une unique ligne de résultat.

Un exemple montrant deux façons de traiter des données dans une large table est:

[2] https://docs.python.org/library/stdtypes.html#iterator-types

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
         odd += 1
return odd
$$ LANGUAGE plpython3u;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS
 integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
        if row['num'] % 2:
            odd += 1
return odd
$$ LANGUAGE plpython3u;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
odd = 0
plan = plpy.prepare("select num from largetable where num % $1
 <> 0", ["integer"])
rows = list(plpy.cursor(plan, [2]))  # or: =
 list(plan.cursor([2]))

return len(rows)
$$ LANGUAGE plpython3u;
```

Les curseurs sont automatiquement libérés. Mais si vous voulez libérer explicitement toutes les ressources retenues par un curseur, il faut utiliser la méthode `close`. Une fois fermé, un curseur ne peut plus être utilisé pour retourner des lignes.

> **Astuce**
>
> Il ne faut pas confondre les objets créés par `plpy.cursor` avec les curseurs DB-API comme définis par la spécification Python Database API[3]. Ils n'ont rien en commun si ce n'est le nom.

46.6.2. Récupérer les erreurs

Les fonctions accédant à la base de données peuvent rencontrer des erreurs, qui forceront leur annulation et lèveront une exception. `plpy.execute` et `plpy.prepare` peuvent lancer une instance d'une sous-classe de `plpy.SPIError`, qui terminera par défaut la fonction. Cette erreur peut être gérée comme toutes les autres exceptions Python, en utilisant la construction `try/except`. Par exemple :

```
CREATE FUNCTION essaie_ajout_joe() RETURNS text AS $$
```

[3] https://www.python.org/dev/peps/pep-0249/

```
    try:
        plpy.execute("INSERT INTO utilisateurs(nom) VALUES
 ('joe')")
    except plpy.SPIError:
        return "quelque chose de mal est arrivé"
    else:
        return "Joe ajouté"
$$ LANGUAGE plpython3u;
```

La classe réelle de l'exception levée correspond à la condition spécifique qui a causé l'erreur. Référez-vous à Tableau A.1 pour une liste des conditions possibles. Le module `plpy.spiexceptions` définit une classe d'exception pour chaque condition PostgreSQL, dérivant leur noms du nom de la condition. Par exemple, `division_by_zero` devient `DivisionByZero`, `unique_violation` devient `UniqueViolation`, `fdw_error` devient `FdwError`, et ainsi de suite. Chacune de ces classes d'exception hérite de `SPIError`. Cette séparation rend plus simple la gestion des erreurs spécifiques. Par exemple :

```
CREATE FUNCTION insere_fraction(numerateur int, denominateur int)
 RETURNS text AS $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 /
 $2)", ["int", "int"])
    plpy.execute(plan, [numerateur, denominateur])
except spiexceptions.DivisionByZero:
    return "denominateur doit être différent de zéro"
except spiexceptions.UniqueViolation:
    return "a déjà cette fraction"
except plpy.SPIError as e:
    return "autre erreur, SQLSTATE %s" % e.sqlstate
else:
    return "fraction insérée"
$$ LANGUAGE plpython3u;
```

Notez que, comme toutes les exceptions du module `plpy.spiexceptions` héritent de `SPIError`, une clause `except` la gérant récupèrera toutes les erreurs d'accès aux bases.

Comme alternative à la gestion des différentes conditions d'erreur, vous pouvez récupérer l'exception `SPIError` et déterminer la condition d'erreur spécifique dans le bloc `except` en recherchant l'attribut `sqlstate` de l'objet exception. Cet attribut est une chaîne contenant le code d'erreur « SQLSTATE ». Cette approche fournit approximativement la même fonctionnalité.

46.7. Sous-transactions explicites

La récupération d'erreurs causées par l'accès à la base de données, comme décrite dans Section 46.6.2, peut amener à une situation indésirable où certaines opérations réussissent avant qu'une d'entre elles échoue et, après récupération de cette erreur, les données sont laissées dans un état incohérent. PL/Python propose une solution à ce problème sous la forme de sous-transactions explicites.

46.7.1. Gestionnaires de contexte de sous-transaction

Prenez en considération une fonction qui implémente un transfert entre deux comptes :

```
CREATE FUNCTION transfert_fonds() RETURNS void AS $$
```

```
try:
    plpy.execute("UPDATE comptes SET balance = balance - 100 WHERE
 nom = 'joe'")
    plpy.execute("UPDATE comptes SET balance = balance + 100 WHERE
 nom = 'mary'")
except plpy.SPIError as e:
    result = "erreur lors du transfert de fond : %s" % e.args
else:
    result = "fonds transféré correctement"
plan = plpy.prepare("INSERT INTO operations (resultat) VALUES
 ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpython3u;
```

Si la deuxième instruction `UPDATE` se termine avec la levée d'une exception, cette fonction renverra l'erreur mais le résultat du premier `UPDATE` sera validé malgré tout. Autrement dit, les fonds auront été débités du compte de Joe mais ils n'auront pas été crédités sur le compte de Mary.

Pour éviter ce type de problèmes, vous pouvez intégrer vos appels à `plpy.execute` dans une sous-transaction explicite. Le module `plpy` fournit un objet d'aide à la gestion des sous-transactions explicites qui sont créées avec la fonction `plpy.subtransaction()`. Les objets créés par cette fonction implémentent l' interface de gestion du contexte[4]. Nous pouvons réécrire notre fonction en utilisant les sous-transactions explicites :

```
CREATE FUNCTION transfert_fonds2() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE comptes SET balance = balance - 100
 WHERE nom = 'joe'")
        plpy.execute("UPDATE comptes SET balance = balance + 100
 WHERE nom = 'mary'")
except plpy.SPIError as e:
    result = "erreur lors du transfert de fond : %s" % e.args
else:
    result = "fonds transféré correctement"
plan = plpy.prepare("INSERT INTO operations (resultat) VALUES
 ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpython3u;
```

Notez que l'utilisation de `try`/`catch` est toujours requis. Sinon, l'exception se propagerait en haut de la pile Python et causerait l'annulation de la fonction entière avec une erreur PostgreSQL, pour que la table `operations` ne contienne aucune des lignes insérées. Le gestionnaire de contexte des sous-transactions ne récupère pas les erreurs, il assure seulement que toutes les opérations de bases de données exécutées dans son cadre seront validées ou annulées de façon atomique. Une annulation d'un bloc de sous-transaction survient à la sortie de tout type d'exception, pas seulement celles causées par des erreurs venant de l'accès à la base de données. Une exception standard Python levée dans un bloc de sous-transaction explicite causerait aussi l'annulation de la sous-transaction.

46.8. Gestion des transactions

Dans une procédure ou dans un bloc de code anonyme (commande `DO`), appelé directement, il est possible de contrôler les transactions. Pour valider la transaction en cours, appelez `plpy.commit()`.

[4] https://docs.python.org/library/stdtypes.html#context-manager-types

Pour annuler la transaction en cours, appelez `plpy.rollback()`. (Notez qu'il n'est pas possible d'exécuter les commandes SQL `COMMIT` ou `ROLLBACK` via `plpy.execute` ou une fonction similaire. Cela doit se faire en utilisant ces fonctions.) Après la fin d'une transaction, une nouvelle transaction est démarrée automatiquement, donc il n'y a pas de fonction séparée pour cela.

Voici un exemple :

```
CREATE PROCEDURE transaction_test1()
LANGUAGE plpythonu
AS $$
for i in range(0, 10):
    plpy.execute("INSERT INTO test1 (a) VALUES (%d)" % i)
    if i % 2 == 0:
        plpy.commit()
    else:
        plpy.rollback()
$$;

CALL transaction_test1();
```

Les transactions ne peuvent être terminées quand une sous-transaction explicite est active.

46.9. Fonctions outils

Le module `plpy` fournit aussi les fonctions

```
plpy.debug(msg, **kwargs)
plpy.log(msg, **kwargs)
plpy.info(msg, **kwargs)
plpy.notice(msg, **kwargs)
plpy.warning(msg, **kwargs)
plpy.error(msg, **kwargs)
plpy.fatal(msg, **kwargs)
```

`plpy.error` et `plpy.fatal` lèvent une exception Python qui, si non attrapée, se propage à la requête appelante causant l'annulation de la transaction ou sous-transaction en cours. `raise plpy.Error(msg)` et `raise plpy.Fatal(msg)` sont équivalent à appeler, respectivement, `plpy.error(msg)` et `plpy.fatal(msg)`, mais la forme `raise` n'autorise pas de passer des arguments par mot clé. Les autres fonctions génèrent uniquement des messages de niveaux de priorité différents. Que les messages d'une priorité particulière soient reportés au client, écrit dans les journaux du serveur ou les deux, cette configuration est contrôlée par les variables log_min_messages et client_min_messages. Voir le Chapitre 20 pour plus d'informations.

L'argument *msg* est donné en tant qu'argument de position. Pour des raisons de compatibilité descendante, plus d'un argument de position doit être donné. Dans ce cas, la représentation en chaîne de caractères de la ligne des arguments de position devient le message rapporté au client.

Les arguments suivant par mot clé seulement sont acceptés :

```
detail
hint
sqlstate
schema_name
table_name
column_name
datatype_name
constraint_name
```

La représentation en chaine des objets passés en argument par mot clé seulement est utilisé pour enrichir les messages rapportés au client. Par exemple :

```
CREATE FUNCTION raise_custom_exception() RETURNS void AS $$
plpy.error("custom exception message",
           detail="some info about exception",
           hint="hint for users")
$$ LANGUAGE plpython3u;

=# SELECT raise_custom_exception();
ERROR:  plpy.Error: custom exception message
DETAIL:  some info about exception
HINT:  hint for users
CONTEXT:  Traceback (most recent call last):
 PL/Python function "raise_custom_exception", line 4, in <module>
   hint="hint for users")
 PL/Python function "raise_custom_exception"
```

Voici un autre ensemble de fonctions outils : `plpy.quote_literal(`*`string`*`)`, `plpy.quote_nullable(`*`string`*`)` et `plpy.quote_ident(`*`string`*`)`. Elles sont équivalentes aux fonctions internes de mise entre guillemets décrites dans Section 9.4. Elles sont utiles lors de la construction de requêtes. Un équivalent PL/Python d'une requête SQL dynamique pour Exemple 43.1 serait :

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

46.10. Python 2 versus Python 3

PL/Python accepte seulement Python 3. Les anciennes versions de PostgreSQL acceptaient Python 2, en utilisant les noms de language `plpythonu` et `plpython2u`.

46.11. Variables d'environnement

Certaines des variables d'environnement qui sont acceptées par l'interpréteur Python peuvent aussi être utilisées pour modifier le comportement de PL/Python. Elles doivent être configurées dans l'environnement du processus serveur PostgreSQL principal, par exemple dans le script de démarrage. Les variables d'environnement disponibles dépendent de la version de Python ; voir la documentation de Python pour les détails. Au moment de l'écriture de ce chapitre, les variables d'environnement suivantes avaient un comportement sur PL/Python, à condition d'utiliser une version adéquate de Python :

- `PYTHONHOME`
- `PYTHONPATH`
- `PYTHONY2K`
- `PYTHONOPTIMIZE`
- `PYTHONDEBUG`
- `PYTHONVERBOSE`

- PYTHONCASEOK
- PYTHONDONTWRITEBYTECODE
- PYTHONIOENCODING
- PYTHONUSERBASE
- PYTHONHASHSEED

(Cela semble être un détail d'implémentation de Python, en dehors du contrôle de PL/Python, qui fait que certaines variables d'environnement listées dans la page man de `python` sont seulement utilisables avec l'interpréteur en ligne de commande et non avec un interpréteur Python embarqué.)

Chapitre 47. Interface de programmation serveur

L'*interface de programmation serveur* (SPI) donne aux auteurs de fonctions C la capacité de lancer des commandes SQL au sein de leurs fonctions ou procédures. SPI est une série de fonctions d'interface simplifiant l'accès à l'analyseur, au planificateur et au lanceur. SPI fait aussi de la gestion de mémoire.

> **Note**
>
> Les langages procéduraux disponibles donnent plusieurs moyens de lancer des commandes SQL à partir de fonctions. La plupart est basée à partir de SPI. Cette documentation présente donc également un intérêt pour les utilisateurs de ces langages.

Notez que si une commande appelée via SPI échoue, alors le contrôle ne sera pas rendu à votre fonction C. Au contraire, la transaction ou sous-transaction dans laquelle est exécutée votre fonction C sera annulée. (Ceci pourrait être surprenant étant donné que les fonctions SPI ont pour la plupart des conventions documentées de renvoi d'erreur. Ces conventions s'appliquent seulement pour les erreurs détectées à l'intérieur des fonctions SPI.) Il est possible de récupérer le contrôle après une erreur en établissant votre propre sous-transaction englobant les appels SPI qui pourraient échouer.

Les fonctions SPI renvoient un résultat positif en cas de succès (soit par une valeur de retour entière, soit dans la variable globale `SPI_result` comme décrit ci-dessous). En cas d'erreur, un résultat négatif ou `NULL` sera retourné.

Les fichiers de code source qui utilisent SPI doivent inclure le fichier d'en-tête `executor/spi.h`.

47.1. Fonctions d'interface

SPI_connect_ext

SPI_connect_ext — connecter une fonction C au gestionnaire SPI

Synopsis

```
int SPI_connect(void)

int SPI_connect_ext(int options)
```

Description

`SPI_connect` ouvre une connexion au gestionnaire SPI lors de l'appel d'une fonction C. Vous devez appeler cette fonction si vous voulez lancer des commandes au travers du SPI. Certaines fonctions SPI utilitaires peuvent être appelées à partir de fonctions C non connectées.

`SPI_connect_ext` fait la même chose mais dispose d'un argument permettant de passer les options. Actuellement, les valeurs possibles des options sont :

`SPI_OPT_NONATOMIC`

Configure la connexion SPI comme *non atomique*, ce qui signifie que les appels de contrôle de transaction, (`SPI_commit`, `SPI_rollback`) sont autorisés. Dans le cas contraire, un appel à ces fonctions renverra immédiatement une erreur.

`SPI_connect()` est équivalent à `SPI_connect_ext(0)`.

Valeur de retour

`SPI_OK_CONNECT`

en cas de succès

`SPI_ERROR_CONNECT`

en cas d'échec

SPI_finish

SPI_finish — déconnecter une fonction C du gestionnaire SPI

Synopsis

```
int SPI_finish(void)
```

Description

`SPI_finish` ferme une connexion existante au gestionnaire SPI. Vous devez appeler cette fonction après avoir terminé les opérations SPI souhaitées pendant l'invocation courante de votre fonction C. Vous n'avez pas à vous préoccuper de ceci, sauf si vous terminez la transaction via `elog(ERROR)`. Dans ce cas, SPI terminera automatiquement.

Valeur de retour

`SPI_OK_FINISH`

si déconnectée correctement

`SPI_ERROR_UNCONNECTED`

si appel à partir d'une fonction C non connectée

SPI_execute

SPI_execute — exécute une commande

Synopsis

```
int SPI_execute(const char * command, bool read_only, long count)
```

Description

`SPI_exec` lance la commande SQL spécifiée pour *`count`* lignes. Si *`read_only`* est `true`, la commande doit être en lecture seule et la surcharge de l'exécution est quelque peu réduite.

Cette fonction ne devrait être appelée qu'à partir d'une fonction C connectée.

Si *`count`* vaut zéro, alors la commande est exécutée pour toutes les lignes auxquelles elle s'applique. Si *`count`* est supérieur à 0, alors pas plus de *`count`* lignes seront récupérées. L'exécution s'arrêtera quand le compte est atteint, un peu comme l'ajout d'une clause `LIMIT` à une requête. Par exemple :

```
SPI_execute("SELECT * FROM foo", true, 5);
```

récupérera 5 lignes tout au plus à partir de la table. Notez qu'une telle limite n'est efficace qu'à partir du moment où la requête renvoie des lignes. Par exemple :

```
 SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

insérera toutes les lignes de `bar`, en ignorant le paramètre *`count`*. Cependant, avec

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false,
 5);
```

au plus cinq lignes seront insérées car l'exécution s'arrêtera après la cinquième ligne renvoyée par `RETURNING`.

Vous pourriez passer plusieurs commandes dans une chaîne. `SPI_execute` renvoie le résultat pour la dernière commande exécutée. La limite *`count`* s'applique à chaque commande séparément (même si seul le dernier résultat sera renvoyé). La limite n'est pas appliquée à toute commande cachée générée par les règles.

Quand *`read_only`* vaut `false`, `SPI_execute` incrémente le compteur de la commande et calcule une nouvelle *image* avant d'exécuter chaque commande dans la chaîne. L'image n'est pas réellement modifiée si le niveau d'isolation de la transaction en cours est `SERIALIZABLE` ou `REPEATABLE READ` mais, en mode `READ COMMITTED`, la mise à jour de l'image permet à chaque commande de voir les résultats des transactions nouvellement validées à partir des autres sessions. Ceci est essentiel pour un comportement cohérent quand les commandes modifient la base de données.

Quand *`read_only`* vaut `true`, `SPI_execute` ne met à jour ni l'image ni le compteur de commandes, et il autorise seulement les commandes `SELECT` dans la chaîne des commandes. Elles sont exécutées en utilisant l'image précédemment établie par la requête englobante. Ce mode d'exécution est un peu plus rapide que le mode lecture/écriture à cause de l'élimination de la surcharge par commande. Il autorise aussi directement la construction des fonctions *stable* comme les exécutions successives utiliseront toutes la même image, il n'y aura aucune modification dans les résultats.

Il n'est généralement pas conseillé de mixer les commandes en lecture seule et les commandes en lecture/écriture à l'intérieur d'une seule fonction utilisant SPI ; ceci pourrait causer un comportement portant confusion car les requêtes en mode lecture seule devraient ne pas voir les résultats de toute mise à jour de la base de données effectuées par les requêtes en lecture/écriture.

Le nombre réel de lignes pour lesquelles la (dernière) commande a été lancée est retourné dans la variable globale `SPI_processed`. Si la valeur de retour de la fonction est `SPI_OK_SELECT`, `SPI_OK_INSERT_RETURNING`, `SPI_OK_DELETE_RETURNING` ou `SPI_OK_UPDATE_RETURNING`, alors vous pouvez utiliser le pointeur global `SPITupleTable *SPI_tuptable` pour accéder aux lignes de résultat. Quelques commandes (comme `EXPLAIN`) renvoient aussi des ensembles de lignes et `SPI_tuptable` contiendra aussi le résultat dans ces cas. Certaines commandes utilitaires (`COPY`, `CREATE TABLE AS`) ne renvoient pas un ensemble de lignes, donc `SPI_tuptable` est NULL, mais elles renvoient malgré tout le nombre de lignes traitées dans `SPI_processed`.

La structure `SPITupleTable` est définie comme suit :

```
+typedef struct SPITupleTable
 {
    /* Membres publics */
    TupleDesc   tupdesc;        /* descripteur de ligne */
    HeapTuple  *vals;           /* tableau de lignes */
    uint64      numvals;        /* nombre de lignes valides */

    /* Membres privés, non destinés aux appelants externes */
    uint64      alloced;        /* longueur alloué au tableau de
 valeurs */
    MemoryContext tuptabcxt;    /* contexte mémoire de la table
 résultante */
    slist_node  next;           /* lien pour la mécanique interne
 */
    SubTransactionId subid;     /* sous-transaction dans laquelle
 le tuptable a été créé */
 } SPITupleTable;
```

Les champs `tupdesc`, `vals`, et `numvals` peuvent être utilisés par les appelants SPI ; les champs restants sont internes. `vals` est un tableau de pointeurs vers des lignes. Le nombre de lignes est donné par `numvals` (pour des raisons un peu historiques, ce nombre est aussi renvoyé dans `SPI_processed`). `tupdesc` est un descripteur de ligne que vous pouvez passer aux fonctions SPI qui traitent des lignes.

`SPI_finish` libère tous les `SPITupleTable`s allouées pendant la fonction C courante. Vous pouvez libérer une table de résultats donnée plus tôt, si vous en avez terminé avec elle, en appelant `SPI_freetuptable`.

Arguments

`const char * *command*`

chaîne contenant la commande à exécuter

`bool *read_only*`

`true` en cas d'exécution en lecture seule

`long *count*`

nombre maximum de lignes à traiter ou `0` pour aucune limite

Valeur de retour

Si l'exécution de la commande a réussi, alors l'une des valeurs (positives) suivantes sera renvoyée :

`SPI_OK_SELECT`

si un `SELECT` (mais pas `SELECT INTO`) a été lancé

`SPI_OK_SELINTO`

si un `SELECT INTO` a été lancé

`SPI_OK_INSERT`

si un `INSERT` a été lancé

`SPI_OK_DELETE`

si un `DELETE` a été lancé

`SPI_OK_UPDATE`

si un `UPDATE` a été lancé

`SPI_OK_MERGE`

si un `MERGE` a été lancé

`SPI_OK_INSERT_RETURNING`

si un `INSERT RETURNING` a été lancé

`SPI_OK_DELETE_RETURNING`

si un `DELETE RETURNING` a été lancé

`SPI_OK_UPDATE_RETURNING`

si un `UPDATE RETURNING` a été lancé

`SPI_OK_UTILITY`

si une commande utilitaire (c'est-à-dire `CREATE TABLE`) a été lancée

`SPI_OK_REWRITTEN`

si la commande a été réécrite dans un autre style de commande (c'est-à-dire que `UPDATE` devient un `INSERT`) par une règle.

Sur une erreur, l'une des valeurs négatives suivantes est renvoyée :

`SPI_ERROR_ARGUMENT`

si *`command`* est `NULL` ou *`count`* est inférieur à 0

`SPI_ERROR_COPY`

si `COPY TO stdout` ou `COPY FROM stdin` ont été tentés

`SPI_ERROR_TRANSACTION`

Si une commande de manipulation de transaction a été tentée (`BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, `PREPARE TRANSACTION`, `COMMIT PREPARED`, `ROLLBACK PREPARED` ou toute variante de ces dernières)

`SPI_ERROR_OPUNKNOWN`

si le type de commande est inconnu (ce qui ne devrait pas arriver)

`SPI_ERROR_UNCONNECTED`

si appel à partir d'une fonction C non connectée

Notes

Toutes les fonctions d'exécution de requêtes SPI changent à la fois `SPI_processed` et `SPI_tuptable` (juste le pointeur, pas le contenu de la structure). Sauvegardez ces deux variables globales dans des variables locales de fonctions C si vous voulez accéder à la table des résultats de `SPI_execute` ou d'une fonction d'exécution de requêtes sur plusieurs appels.

SPI_exec

SPI_exec — exécute une commande en lecture/écriture

Synopsis

```
int SPI_exec(const char * command, long count)
```

Description

`SPI_exec` est identique à `SPI_execute`, mais le paramètre *`read_only`* de ce dernier est bloqué sur la valeur `false`.

Arguments

`const char * command`

chaîne contenant la commande à exécuter

`long count`

nombre maximum de lignes à renvoyer ou `0` pour aucune limite

Valeur de retour

Voir `SPI_execute`.

SPI_execute_extended

SPI_execute_extended — Exécute une commande avec des paramètres hors ligne

Synopsis

```
int SPI_execute_extended(const char *command,
                         const SPIExecuteOptions * options)
```

Description

`SPI_execute_extended` exécute une commande qui peut inclure des références à des paramètres fournis en externe. Le texte de commande se réfère à un paramètre sous la forme `$n`, et l'objet *`options->params`* (si fourni) donne des informations sur les valeurs et le type de chacun des symboles. De nombreuses options d'exécution peuvent aussi être spécifiées dans la structure *`options`*

L'objet *`options->params`* doit normalement marquer chaque paramètre avec le drapeau `PARAM_FLAG_CONST`, dès qu'un plan unique est toujours utilisé pour cette requête.

Si *`options->dest`* n'est pas NULL, alors les lignes de résultats sont passées à cet objet lors de leur génération par l'exécuteur, au lieu d'être accumulées dans `SPI_tuptable`. Employer un objet `DestReceiver` fourni par l'appelant est particulièrement utile pour les requêtes qui peuvent générer de nombreuses lignes, dès lors que les données sont traitées à la volée au lieu de s'accumuler en mémoire.

Arguments

`const char * command`

Chaîne de caractères de la commande

`const SPIExecuteOptions * options`

Structure contenant les arguments optionnels

Les codes appelants doivent toujours mettre à zéro l'entièreté de la structure *`options`*, et ensuite remplir les champs souhaités. Ceci permet d'assurer la compatibilité ascendante du code, car tous les champs ajoutés à la structure dans le futur seront définis comme étant rétrocompatibles s'ils sont mis à zéro. Les champs de *`options`* actuellement disponibles sont :

`ParamListInfo params`

Structure de données contenant les types et valeurs des paramètres de requêtes ; NULL si aucun.

`bool read_only`

`true` pour les exécutions en lecture seule

`bool no_snapshots`

à `true`, autorise l'exécution non atomique des instructions CALL et DO (mais ce champ est ignoré jusqu'à ce que le drapeau `SPI_OPT_NONATOMIC` soit passé à `SPI_connect_ext`)

`bool must_return_tuples`

si `true`, lève une erreur si la requête n'est pas d'un type qui renvoie des lignes (ceci n'interdit pas le cas où elle pourrait renvoyer zéro ligne)

`uint64` *`tcount`*

Nombre maximum de lignes à retourner, ou `0` pour aucune limite

`DestReceiver *` *`dest`*

L'objet `DestReceiver` qui recevra toutes les lignes émises par la requête ; si NULL, les lignes de résultats sont accumulés dans une structure `SPI_tuptable`, comme avec `SPI_execute`

`ResourceOwner` *`owner`*

Ce champ est présent pour la cohérence avec `SPI_execute_plan_extended`, mais est ignoré, puisque le plan utilisé par `SPI_execute_extended` n'est jamais sauvegardé.

Valeur de retour

La valeur de retour est la même que pour `SPI_execute`.

Quand *`options->dest`* est NULL, `SPI_processed` et `SPI_tuptable` sont affectés comme dans `SPI_execute`. Quand *`options->dest`* n'est pas NULL, `SPI_processed` est mis à zéro et `SPI_tuptable` est mis à NULL. Si un comptage de lignes est requis, l'objet `DestReceiver` appelant doit s'en occuper.

SPI_execute_with_args

SPI_execute_with_args — exécute une commande avec des paramètres hors ligne

Synopsis

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

Description

`SPI_execute_with_args` exécute une commande qui pourrait inclure des références à des paramètres fournis en externe. Le texte de commande fait référence à un paramètre avec `$`*`n`* et l'appel spécifie les types et valeurs des données pour chaque symbole de ce type. *`read_only`* et *`count`* ont la même interprétation que dans `SPI_execute`.

Le principal avantage de cette routine comparé à `SPI_execute` est que les valeurs de données peuvent être insérées dans la commande sans mise entre guillemets et échappements, et donc avec beaucoup moins de risques d'attaques du type injection SQL.

Des résultats similaires peuvent être réalisés avec `SPI_prepare` suivi par `SPI_execute_plan` ; néanmoins, lors de l'utilisation de cette fonction, le plan de requête est toujours personnalisé avec les valeurs de paramètres spécifiques fournies. Pour une exécution simple, cette fonction doit être préférée. Si la même commande doit être exécutée avec plusieurs paramètres différents, chaque méthode peut être la plus rapide, le coût de la planification pouvant contrebalancer les bénéfices des plans personnalisés.

Arguments

`const char * command`

chaîne de commande

`int nargs`

nombre de paramètres en entrée (`$1`, `$2`, etc.)

`Oid * argtypes`

un tableau de longueur *`nargs`*, contenant les OID des types de données des paramètres

`Datum * values`

un tableau de longueur *`nargs`*, containing des valeurs réelles des paramètres

`const char * nulls`

un tableau décrivant les paramètres NULL

Si *`nulls`* vaut `NULL`, alors `SPI_execute_with_args` suppose qu'aucun paramètre n'est NULL. Dans le cas contraire, chaque entrée du tableau *`nulls`* doit valoir `' '` si le paramètre correspondant est non NULL et `'n'` si le paramètre correspondant est NULL (dans ce dernier cas, la valeur réelle de l'entrée *`values`* correspondante n'a pas d'importance). Notez que *`nulls`* n'est pas une chaîne de texte. C'est un tableau et, de ce fait, il n'a pas besoin d'un caractère de fin `'\0'`.

`bool` *`read_only`*

`true` pour les exécutions en lecture seule

`long` *`count`*

nombre maximum de lignes à renvoyer ou `0` pour aucune limite

Valeur de retour

La valeur de retour est identique à celle de `SPI_execute`.

`SPI_processed` et `SPI_tuptable` sont configurés comme dans `SPI_execute` en cas de succès.

SPI_prepare

SPI_prepare — prépare une instruction sans l'exécuter tout de suite

Synopsis

```
SPIPlanStr SPI_prepare(const char * command, int nargs, Oid
 * argtypes)
```

Description

`SPI_prepare` crée et retourne une requête préparée pour la commande spécifiée mais ne lance pas la commande. La requête préparée peut être appelée plusieurs fois en utilisant `SPI_execute_plan`.

Lorsque la même commande ou une commande semblable doit être lancée à plusieurs reprises, il est généralement avantageux de réaliser une analyse du plan d'exécution une fois et de réutiliser le plan d'exécution pour la commande. `SPI_prepare` convertit une chaîne de commande en une requête préparée qui encapsule le résultat de l'analyse du plan. La requête préparée fournit aussi une place pour mettre en cache un plan d'exécution s'il s'avère que la génération d'un plan personnalisé pour chaque exécution n'est pas utile.

Une commande préparée peut être généralisée en utilisant les paramètres (`$1`, `$2`, etc.) en lieu et place de ce qui serait des constantes dans une commande normale. Les valeurs actuelles des paramètres sont alors spécifiées lorsque `SPI_executeplan` est appelée. Ceci permet à la commande préparée d'être utilisée sur une plage plus grande de situations que cela ne serait possible sans paramètres.

La requête renvoyée par `SPI_prepare` ne peut être utilisé que dans l'invocation courante de la fonction C puisque `SPI_finish` libère la mémoire allouée pour la requête. Mais l'instruction peut être sauvegardée plus longtemps par l'utilisation des fonctions `SPI_keepplan` ou `SPI_saveplan`.

Arguments

`const char * command`

chaîne contenant la commande à planifier

`int nargs`

nombre de paramètres d'entrée (`$1`, `$2`, etc.)

`Oid * argtypes`

pointeur vers un tableau contenant les OID des types de données des paramètres

Valeurs de retour

`SPI_prepare` retourne un pointeur non nul vers un plan d'exécution. En cas d'erreur, `NULL` sera retourné et `SPI_result` sera positionnée à un des mêmes codes d'erreur utilisés par `SPI_execute` sauf qu'il est positionné à `SPI_ERROR_ARGUMENT` si *`command`* est `NULL` ou si *`nargs`* est inférieur à 0 ou si *`nargs`* est supérieur à 0 et *`typesargs`* est `NULL`.

Notes

Si aucun paramètre n'est défini, un plan générique sera créé lors de la première utilisation de `SPI_execute_plan`, et utilisé aussi pour toutes les exécutions suivantes. Si des paramètres sont

fournis, les premières utilisations de `SPI_execute_plan` généreront des plans personnalisés qui sont spécifiques aux valeurs fournies pour les paramètres. Après suffisamment d'utilisation de la même requête préparée, `SPI_execute_plan` construira un plan générique et, si ce n'est pas beaucoup plus coûteux que les plans personnalisés, cette fonction commencera à utiliser le plan générique au lieu de re-planifier à chaque fois. Si le comportement par défaut n'est pas tenable, vous pouvez le modifier en passant le drapeau `CURSOR_OPT_GENERIC_PLAN` ou `CURSOR_OPT_CUSTOM_PLAN` à `SPI_prepare_cursor` pour forcer l'utilisation, respectivement, de plans génériques ou personnalisés.

Bien que le but principal d'une requête préparée est d'éviter les étapes d'analyse et de planification d'une requête, PostgreSQL forcera l'analyse et la planification de la requête avant de l'utiliser quand les objets de la base utilisés dans la requête ont subi des changements de définition (à partir de requêtes DDL) depuis la dernière utilisation de la requête préparée. De plus, si la valeur de search_path change d'une exécution à une autre, la requête sera de nouveau planifiée en utilisant le nouveau `search_path` (ce dernier comportement est une nouveauté de la version 9.3 de PostgreSQL). Voir PREPARE pour plus d'informations sur le comportement des requêtes préparées.

Cette fonction doit seulement être appelée à partir d'une fonction C connectée.

`SPIPlanPtr` est déclaré comme un pointeur vers un type de structure opaque dans `spi.h`. Il est déconseillé d'essayer d'accéder à son contenu directement car cela rend votre code plus fragile aux futures versions de PostgreSQL.

Le nom `SPIPlanPtr` est historique principalement car la structure des données ne contient plus nécessairement un plan d'exécution.

SPI_prepare_cursor

SPI_prepare_cursor — prépare une requête, sans l'exécuter pour l'instant

Synopsis

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs, Oid
 * argtypes, int cursorOptions)
```

Description

`SPI_prepare_cursor` est identique à `SPI_prepare`, sauf qu'il permet aussi la spécification du paramètre des « options du curseur » du planificateur. Il s'agit d'un champ de bits dont les valeurs sont indiquées dans `nodes/parsenodes.h` pour le champ `options` de `DeclareCursorStmt`. `SPI_prepare` utilise zéro pour les options du curseur.

Cette fonction est dépréciée en faveur de `SPI_prepare_extended`.

Arguments

`const char * command`

chaîne commande

`int nargs`

nombre de paramètres en entrée (`$1`, `$2`, etc.)

`Oid * argtypes`

pointeur vers un tableau contenant l'OID des types de données des paramètres

`int cursorOptions`

champ de bits précisant les options du curseur ; zéro est le comportement par défaut

Valeur de retour

`SPI_prepare_cursor` a les mêmes conventions pour la valeur de retour que `SPI_prepare`.

Notes

Les bits utiles pour *`cursorOptions`* incluent `CURSOR_OPT_NO_SCROLL`, `CURSOR_OPT_FAST_PLAN`, `CURSOR_OPT_GENERIC_PLAN` et `CURSOR_OPT_CUSTOM_PLAN`. Notez en particulier que `CURSOR_OPT_HOLD` est ignoré.

SPI_prepare_extended

SPI_prepare_extended — prépare une instruction, sans l'exécuter tout de suite

Synopsis

```
SPIPlanPtr SPI_prepare_extended(const char * command,
                                const SPIPrepareOptions * options)
```

Description

`SPI_prepare_extended` crée et retourne une requête préparée pour une commande donnée, mais n'exécute pas la commande. Cette fonction est équivalente à `SPI_prepare`, avec en plus le fait que l'appelant peut spécifier des options pour contrôler l'analyse (« parsing ») des paramètres en référence externes, aussi bien que d'autres aspects de l'analyse et de la planification des requêtes.

Arguments

`const char * command`

Chaîne de la commande

`const SPIPrepareOptions * options`

Structure contenant les arguments optionnels

Les codes appelants doivent toujours mettre à zéro la structure *`options`* entière, puis remplir les champs souhaités avec les valeurs à affecter. Ceci permet d'assurer une compatibilité ascendante du code, car tous les champs ajoutés à la structure dans le futur seront définis comme étant rétrocompatibles s'ils sont mis à zéro. Les champs de *`options`* actuellement disponibles sont :

`ParserSetupHook parserSetup`

Fonction de configuration de l'analyseur

`void * parserSetupArg`

Argument passé à *`parserSetup`*

`RawParseMode parseMode`

Mode pour l'analyse brute ; `RAW_PARSE_DEFAULT` (zéro) génère le comportement par défaut

`int cursorOptions`

Masque de bits des options du curseur, sous la forme d'un entier ; zéro indique le comportement par défaut

Valeur retournée

`SPI_prepare_extended` a la même convention de retour que `SPI_prepare`.

SPI_prepare_params

SPI_prepare_params — prépare une requête, mais sans l'exécuter

Synopsis

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

Description

`SPI_prepare_params` crée et renvoie une requête préparée pour la commande indiquée mais n'exécute pas la commande. Cette fonction est équivalente à `SPI_prepare_cursor` avec en plus le fait que l'appelant peut indiquer des fonctions pour contrôler l'analyse de références de paramètres externes.

Cette fonction est maintenant dépréciée en faveur de `SPI_prepare_extended`.

Arguments

`const char * command`

chaîne correspondant à la commande

`ParserSetupHook parserSetup`

fonction de configuration de l'analyseur

`void * parserSetupArg`

argument passé à *parserSetup*

`int cursorOptions`

masque de bits des options du curseur, sous la forme d'un entier ; zéro indique le comportement par défaut

Code de retour

`SPI_prepare_params` a les mêmes conventions de retour que `SPI_prepare`.

SPI_getargcount

SPI_getargcount — renvoie le nombre d'arguments nécessaire à une requête par `SPI_prepare`

Synopsis

```
int SPI_getargcount(SPIPlanPtr plan)
```

Description

`SPI_getargcount` renvoie le nombre d'arguments nécessaires pour exécuter une requête préparée par `SPI_prepare`.

Arguments

`SPIPlanPtr` *`plan`*

requête préparée (renvoyée par `SPI_prepare`)

Code de retour

Le nombre d'arguments attendus par le *`plan`*. Si *`plan`* est `NULL` ou invalide, `SPI_result` est initialisé à `SPI_ERROR_ARGUMENT` et -1 est renvoyé.

SPI_getargtypeid

SPI_getargtypeid — renvoie l'OID du type de données pour un argument de la requête préparée par SPI_prepare

Synopsis

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

Description

SPI_getargtypeid renvoie l'OID représentant le type pour le *argIndex*-ième argument d'une requête préparée par SPI_prepare. Le premier argument se trouve à l'index zéro.

Arguments

SPIPlanPtr *plan*

requête préparée (renvoyée par SPI_prepare)

int *argIndex*

index de l'argument (à partir de zéro)

Code de retour

L'OID du type de l'argument à l'index donné. Si le *plan* est NULL ou invalide, ou *argIndex* inférieur à 0 ou pas moins que le nombre d'arguments déclaré pour le *plan*, SPI_result est initialisé à SPI_ERROR_ARGUMENT et InvalidOid est renvoyé.

SPI_is_cursor_plan

SPI_is_cursor_plan — renvoie `true` si la requête préparée par `SPI_prepare` peut être utilisé avec `SPI_cursor_open`

Synopsis

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

Description

`SPI_is_cursor_plan` renvoie `true` si une requête préparée par `SPI_prepare` peut être passé comme un argument à `SPI_cursor_open` ou `false` si ce n'est pas le cas. Les critères sont que le *`plan`* représente une seule commande et que cette commande renvoie des lignes à l'appelant ; par l'exemple, `SELECT` est autorisé sauf s'il contient une clause `INTO` et `UPDATE` est autorisé seulement s'il contient un `RETURNING`

Arguments

`SPIPlanPtr` *`plan`*

requête préparée (renvoyée par `SPI_prepare`)

Valeur de retour

`true` ou `false` pour indiquer si *`plan`* peut produire un curseur ou non, avec `SPI_result` initialisé à zéro. S'il nest pas possible de déterminer la réponse (par exemple, si le *`plan`* vaut `NULL` ou est invalide, ou s'il est appelé en étant déconnecté de SPI), alors `SPI_result` est configuré avec un code d'erreur convenable et `false` est renvoyé.

SPI_execute_plan

SPI_execute_plan — exécute une requête préparée par SPI_prepare

Synopsis

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char
 * nulls, bool read_only, long count)
```

Description

SPI_execute_plan exécute une requête préparée par SPI_prepare ou une fonction du même type. *read_only* et *count* ont la même interprétation que dans SPI_execute.

Arguments

SPIPlanPtr *plan*

requête préparée (retournée par SPI_prepare)

Datum **values*

Un tableau des vraies valeurs des paramètres. Doit avoir la même longueur que le nombre d'arguments de la requête.

const char * *nulls*

Un tableau décrivant les paramètres nuls. Doit avoir la même longueur que le nombre d'arguments de la requête.

Si *nulls* vaut NULL, alors SPI_execute_plan suppose qu'aucun paramètre n'est NULL. Dans le cas contraire, chaque entrée du tableau *nulls* doit valoir ' ' si le paramètre correspondant est non NULL et 'n' si le paramètre correspondant est NULL (dans ce dernier cas, la valeur réelle de l'entrée *values* correspondante n'a pas d'importance). Notez que *nulls* n'est pas une chaîne de texte. C'est un tableau et, de ce fait, il n'a pas besoin d'un caractère de fin '\0'.

bool *read_only*

true pour une exécution en lecture seule

long *count*

nombre maximum de lignes à renvoyer ou 0 pour aucune ligne à renvoyer

Valeur de retour

La valeur de retour est la même que pour SPI_execute avec les résultats d'erreurs (négatif) possibles :

SPI_ERROR_ARGUMENT

si *plan* est NULL ou invalide ou *count* est inférieur à 0

SPI_ERROR_PARAM

si *values* est NULL et *plan* est préparé avec des paramètres

SPI_processed et SPI_tuptable sont positionnés comme dans SPI_execute en cas de réussite.

SPI_execute_plan_extended

SPI_execute_plan_extended — exécute une requête préparée par SPI_prepare

Synopsis

```
int SPI_execute_plan_extended(SPIPlanPtr plan,
                              const SPIExecuteOptions * options)
```

Description

SPI_execute_plan_extended exécute une requête préparée par SPI_prepare ou une similaire. La fonction est l'équivalent de SPI_execute_plan, à l'exception que l'information sur les valeurs des paramètres passées à la requête est présentée différemment, et des options supplémentaires de contrôle d'exécution peuvent être fournies.

Les valeurs des paramètres de requêtes sont représentées par une structure ParamListInfo, ce qui est pratique pour fournir les valeurs qui sont déjà disponibles dans ce format. Des ensembles de paramètres dynamiques peuvent aussi être utilisées, à travers des fonctions d'accrochage spécifiées dans ParamListInfo.

De plus, au lieu de toujours accumuler les lignes de résultats dans une structure SPI_tuptable, les lignes peuvent être passées à un objet DestReceiver fourni par l'appelant au fur et à mesure de leur génération par l'exécuteur de requête. Ceci est particulièrement utile pour les requêtes qui peuvent générer de nombreuses lignes, comme les données peuvent être traitées au fur et à mesure au lieu d'être accumulées en mémoire.

Arguments

SPIPlanPtr *plan*

Expression préparée (retournée par SPI_prepare)

const SPIExecuteOptions * *options*

Structure contenant les arguments optionnels

Les appelants doivent toujours mettre l'entièreté de la structure *options* à zéro, puis remplir les champs nécessaires à affecter. Ceci assure une compatibilité ascendante du code, car tous les champs qui seront ajoutés à la structure dans le futur seront rétro-compatibles s'ils sont mis à zéro. Les champs de *options* actuellement disponbiles sont :

ParamListInfo *params*

Structure de données contenant les types et valeurs des paramètres de requêtes ; NULL si aucun

bool *read_only*

true pour les exécutions en lecture seule

bool *no_snapshots*

true autorise l'exécution non atomique des instructions CALL et DO (mais ce champ est ignoré jusqu'à ce que le drapeau SPI_OPT_NONATOMIC soit passé à SPI_connect_ext)

bool *must_return_tuples*

si true, lève une erreur si la requête n'est pas d'un type qui renvoie des lignes (ceci n'interdit pas le cas où elle pourrait renvoyer zéro ligne)

`uint64` *`tcount`*

Nombre maximal de lignes à retourner ou `0` pour illimité

`DestReceiver *` *`dest`*

L'objet `DestReceiver` qui va recevoir toutes les lignes émises par la requête ; si NULL, les lignes des résultats sont accumulées dans une structure `SPI_tuptable`, comme dans `SPI_execute_plan`

`ResourceOwner` *`owner`*

Le propriétaire de ressources qui va garder un compteur de référence sur le plan pendant qu'il est exécuté. Si NULL, CurrentResourceOwner est utilisé. Ce champ est ignoré pour les plans non sauvegardés, comme SPI n'acquiert pas de compteur de référence dessus.

Valeur retournée

La valeur retournée est la même que pour `SPI_execute_plan`.

Quand *`options->dest`* est NULL, `SPI_processed` et `SPI_tuptable` sont affectés comme dans `SPI_execute_plan`. Quand *`options->dest`* n'est pas NULL, `SPI_processed` est mis à zéro et `SPI_tuptable` est mis à NULL. Si un comptage de lignes est requis, l'appellant de l'objet `DestReceiver` doit s'en charger.

SPI_execute_plan_with_paramlist

SPI_execute_plan_with_paramlist — exécute une requête préparée par SPI_prepare

Synopsis

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                    ParamListInfo params,
                                    bool read_only,
                                    long count)
```

Description

SPI_execute_plan_with_paramlist exécute une requête préparée par SPI_prepare. Cette fonction est l'équivalent de SPI_execute_plan, sauf que les informations sur les valeurs des paramètres à passer à la requête sont présentées différemment. La représentation ParamListInfo peut être utilisée pour passer des valeurs qui sont déjà disponibles dans ce format. Elle supporte aussi l'utilisation d'ensemble de paramètres dynamiques indiqués via des fonctions dans ParamListInfo.

This function is now deprecated in favor of SPI_execute_plan_extended.

Arguments

SPIPlanPtr *plan*

requête préparée (renvoyée par SPI_prepare)

ParamListInfo *params*

structure de données contenant les types et valeurs de paramètres ; NULL si aucune structure

bool *read_only*

true pour une exécution en lecture seule

long *count*

nombre maximum de lignes à renvoyer ou 0 pour aucune ligne à renvoyer

Code de retour

La valeur de retour est identique à celle de SPI_execute_plan.

SPI_processed et SPI_tuptable sont initialisés de la même façon que pour SPI_execute_plan en cas de réussite.

SPI_execp

SPI_execp — exécute une requête en mode lecture/écriture

Synopsis

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls,
 long count)
```

Description

`SPI_execp` est identique à `SPI_execute_plan` mais le paramètre *`read_only`* de ce dernier vaut toujours `false`.

Arguments

`SPIPlanPtr` *`plan`*

requête préparée (renvoyée par `SPI_prepare`)

`Datum *` *`values`*

Un tableau des vraies valeurs de paramètre. Doit avoir la même longueur que le nombre d'arguments de la requête.

`const char *` *`nulls`*

Un tableau décrivant les paramètres NULL. Doit avoir la même longueur que le nombre d'arguments de la requête.

Si *`nulls`* vaut `NULL`, alors `SPI_executep` suppose qu'aucun paramètre n'est NULL. Dans le cas contraire, chaque entrée du tableau *`nulls`* doit valoir `' '` si le paramètre correspondant est non NULL et `'n'` si le paramètre correspondant est NULL (dans ce dernier cas, la valeur réelle de l'entrée *`values`* correspondante n'a pas d'importance). Notez que *`nulls`* n'est pas une chaîne de texte. C'est un tableau et, de ce fait, il n'a pas besoin d'un caractère de fin `'\0'`.

`long` *`count`*

nombre maximum de lignes à renvoyer ou `0` pour aucune ligne à renvoyer

Valeur de retour

Voir `SPI_execute_plan`.

`SPI_processed` et `SPI_tuptable` sont initialisés comme dans `SPI_execute` en cas de succès.

SPI_cursor_open

SPI_cursor_open — met en place un curseur en utilisant une requête créée avec SPI_prepare

Synopsis

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
Datum * values, const char * nulls,
bool read_only)
```

Description

SPI_cursor_open met en place un curseur (en interne, un portail) qui lancera une requête préparée par SPI_prepare. Les paramètres ont la même signification que les paramètres correspondant à SPI_execute_plan.

Utiliser un curseur au lieu de lancer une requête directement à deux avantages. Premièrement, les lignes de résultats peuvent être récupérées un certain nombre à la fois, évitant la saturation de mémoire pour les requêtes qui retournent trop de lignes. Deuxièmement, un portail peut survivre à la fonction C courante (elle peut, en fait, vivre jusqu'à la fin de la transaction courante). Renvoyer le nom du portail à l'appelant de la fonction C donne un moyen de retourner une série de lignes en tant que résultat.

Les données passées seront copiées dans le portail du curseur, donc il peut être libéré alors que le curseur existe toujours.

Cette fonction est maintenant dépréciée en faveur de SPI_cursor_parse_open, qui fournit une fonctionnalité équivalente en utilisant une API plus moderne pour gérer les paramètres de requêtes.

Arguments

`const char * name`

nom pour le portail ou NULL pour laisser le système choisir un nom

`SPIPlanPtr plan`

requête préparée (retournée par SPI_prepare)

`Datum * values`

Un tableau des valeurs de paramètres actuelles. Doit avoir la même longueur que le nombre d'arguments de la requête.

`const char *nulls`

Un tableau décrivant quels paramètres sont NULL. Doit avoir la même longueur que le nombre d'arguments de la requête.

Si *nulls* vaut NULL, alors SPI_cursor_open suppose qu'aucun paramètre n'est NULL. Dans le cas contraire, chaque entrée du tableau *nulls* doit valoir ' ' si le paramètre correspondant est non NULL et 'n' si le paramètre correspondant est NULL (dans ce dernier cas, la valeur réelle de l'entrée *values* correspondante n'a pas d'importance). Notez que *nulls* n'est pas une chaîne de texte. C'est un tableau et, de ce fait, il n'a pas besoin d'un caractère de fin '\0'.

`bool read_only`

true pour les exécutions en lecture seule

Valeur de retour

Pointeur vers le portail contenant le curseur. Notez qu'il n'y a pas de convention pour le renvoi d'une erreur ; toute erreur sera rapportée via `elog`.

SPI_cursor_open_with_args

SPI_cursor_open_with_args — ouvre un curseur en utilisant une requête et des paramètres

Synopsis

```
Portal SPI_cursor_open_with_args(const char *name,
                                 const char *command,
                                 int nargs, Oid *argtypes,
                                 Datum *values, const char *nulls,
                                 bool read_only, int cursorOptions)
```

Description

`SPI_cursor_open_with_args` initialise un curseur (en interne, un portail) qui exécutera la requête spécifiée. La plupart des paramètres ont la même signification que les paramètres correspondant de `SPI_prepare_cursor` et `SPI_cursor_open`.

Pour une exécution seule, cette fonction sera préférée à `SPI_prepare_cursor` suivie de `SPI_cursor_open`. Si la même commande doit être exécutée avec plusieurs paramètres différents, il n'y a pas de différences sur les deux méthodes, la replanification a un coût, mais bénéficie de plans personnalisés.

Les données passées seront copiées dans le portail du curseur, donc elles seront libérées alors que le curseur existe toujours.

Arguments

`const char * name`

nom du portail, ou `NULL` pour que le système sélectionne un nom de lui-même

`const char * command`

chaîne de commande

`int nargs`

nombre de paramètres en entrée (`$1`, `$2`, etc.)

`Oid * argtypes`

un tableau de longueur *`nargs`*, contenant les OID des types de données des paramètres

`Datum * values`

un tableau de longueur *`nargs`*, contenant les valeurs actuelles des paramètres

`const char * nulls`

un tableau de longueur *`nargs`*, décrivant les paramètres NULL

Si *`nulls`* vaut `NULL`, alors `SPI_cursor_open_with_args` suppose qu'aucun paramètre n'est NULL. Dans le cas contraire, chaque entrée du tableau *`nulls`* doit valoir `' '` si le paramètre correspondant est non NULL et `'n'` si le paramètre correspondant est NULL (dans ce dernier cas,

la valeur réelle de l'entrée *values* correspondante n'a pas d'importance). Notez que *nulls* n'est pas une chaîne de texte. C'est un tableau et, de ce fait, il n'a pas besoin d'un caractère de fin '\0'.

bool *read_only*

true pour une exécution en lecture seule

int *cursorOptions*

masque de bits des options du curseur : zéro cause le comportement par défaut

Valeur de retour

Pointeur du portail contenant le curseur. Notez qu'il n'y a pas de convention pour le renvoi des erreurs ; toute erreur sera rapportée par elog.

SPI_cursor_open_with_paramlist

SPI_cursor_open_with_paramlist — ouvre un curseur en utilisant les paramètres

Synopsis

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

Description

`SPI_cursor_open_with_paramlist` prépare un curseur (en interne un portail), qui exécutera une requête préparée par `SPI_prepare`. Cette fonction est équivalente à `SPI_cursor_open` sauf que les informations sur les valeurs des paramètres passées à la requête sont présentées différemment. La représentation de `ParamListInfo` peut être utile pour fournir des valeurs déjà disponibles dans ce format. Elle supporte aussi l'utilisation d'ensemble de paramètres dynamiques via des fonctions spécifiées dans `ParamListInfo`.

Les données passées en paramètre seront copiées dans le portail du curseur et peuvent donc être libérées alors que le curseur existe toujours.

Arguments

`const char * name`

nom d'un portail ou `NULL` pour que le système en choisisse un lui-même

`SPIPlanPtr plan`

requête préparée (renvoyée par `SPI_prepare`)

`ParamListInfo params`

structure de données contenant les types et valeurs de paramètres ; NULL sinon

`bool read_only`

`true` pour une exécution en lecture seule

Valeur de retour

Pointeur vers le portail contenant le curseur. Notez qu'il n'existe pas de convention pour le retour d'erreur ; toute erreur sera renvoyée via `elog`.

SPI_cursor_parse_open

SPI_cursor_parse_open — provisionne un curseur en utilisant une requête sous forme chaîne et des paramètres

Synopsis

```
Portal SPI_cursor_parse_open(const char *name,
                             const char *command,
                             const SPIParseOpenOptions * options)
```

Description

La fonction `SPI_cursor_parse_open` provisionne un curseur (en interne, un portail) qui exécutera une requête spécifiée dans une chaine. Elle est comparable à l'appel de `SPI_prepare_cursor` suivie de `SPI_cursor_open_with_paramlist`, à l'exception que les références des paramètres dans la chaine de la requête sont entièrement gérées en fournissant un objet `ParamListInfo`.

Pour une exécution unique d'une requête, cette fonction devrait être préférée à l'appel de `SPI_prepare_cursor` suivie par `SPI_cursor_open_with_paramlist`. Si la même commande doit être exécutée avec de nombreux paramètres différents, l'une ou l'autre méthode pourrait être plus rapide, en fonction du coût de recalcul du plan par rapport aux bénéfices des plans personnalisés.

L'objet *`options->params`* doit normalement marqué chaque paramètre avec le drapeau `PARAM_FLAG_CONST`, comme un plan unique est toujours utilisé par la requête.

Le paramètre passé en entrée sera copié dans le portail du curseur, ainsi il peut être libéré alors que le curseur existe toujours.

Arguments

`const char * name`

Nom du portail, ou `NULL` pour laisser le système choisir un nom

`const char * command`

Chaîne de la commande

`const SPIParseOpenOptions * options`

Structure contenant les arguments optionnels

Les appelants doivent toujours mettre à zéro la structure entière *`options`*, puis remplir les champs nécessaires à affecter. Ceci assure la compatibilité ascendante du code, comme tout champ ajouté à la structure dans le futur sera défini comme rétro-compatible s'il est mis à zéro. Les champs de *`options`* actuellement disponibles sont :

`ParamListInfo params`

Structure de données contenant les types et valeurs de paramètres de la requête ; NULL si aucun

`int cursorOptions`

Masque de bits entiers des options de curseur ; zéro génère le comportement par défaut

`bool` *`read_only`*

`true` pour une exécution en lecture seule

Valeur de retour

Pointeur de portail contenant le curseur. Notez qu'il n'y a pas de convention de retour d'erreur ; n'importe quelle erreur sera affichée via `elog`.

SPI_cursor_find

SPI_cursor_find — recherche un curseur existant par nom

Synopsis

```
Portal SPI_cursor_find(const char * name)
```

Description

`SPI_cursor_find` recherche un portail par nom. Ceci est principalement utile pour résoudre un nom de curseur renvoyé en tant que texte par une autre fonction.

Arguments

`const char * name`

nom du portail

Valeur de retour

Pointeur vers le portail portant le nom spécifié ou `NULL` si aucun n'a été trouvé

Notes

Attention, cette fonction peut renvoyer un objet `Portal` qui n'a pas de propriétés identiques aux curseurs ; par exemple, elle pourrait ne pas renvoyer de lignes. Si vous lui passez simplement un pointeur `Portal` vers d'autres fonctions SPI, elles peuvent se défendre de tels cas mais une attention est nécessaire lors d'une inspection directe du `Portal`.

SPI_cursor_fetch

SPI_cursor_fetch — extrait des lignes à partir d'un curseur

Synopsis

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

Description

`SPI_cursor_fetch` extrait des lignes à partir d'un curseur. Ceci est équivalent à un sous-ensemble de la commande SQL `FETCH` (voir `SPI_scroll_cursor_fetch` pour plus de détails).

Arguments

`Portal` *`portal`*

portail contenant le curseur

`bool` *`forward`*

vrai pour une extraction en avant, faux pour une extraction en arrière

`long` *`count`*

nombre maximum de lignes à récupérer

Valeur de retour

`SPI_processed` et `SPI_tuptable` sont positionnés comme dans `SPI_execute` en cas de réussite.

Notes

Récupérer en sens inverse pourrait échouer si le plan du curseur n'était pas créé avec l'option `CURSOR_OPT_SCROLL`.

SPI_cursor_move

SPI_cursor_move — déplace un curseur

Synopsis

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

Description

`SPI_cursor_move` saute un certain nombre de lignes dans un curseur. Ceci est équivalent à un sous-ensemble de la commande SQL `MOVE` (voir `SPI_scroll_cursor_move` pour plus de détails).

Arguments

`Portal` *`portal`*

portail contenant le curseur

`bool` *`forward`*

vrai pour un saut en avant, faux pour un saut en arrière

`long` *`count`*

nombre maximum de lignes à déplacer

Notes

Se déplacer en sens inverse pourrait échouer si le plan du curseur n'a pas été créé avec l'option `CURSOR_OPT_SCROLL` option.

SPI_scroll_cursor_fetch

SPI_scroll_cursor_fetch — récupère quelques lignes à partir d'un curseur

Synopsis

```
void SPI_scroll_cursor_fetch(Portal portal,
 FetchDirection direction, long count)
```

Description

`SPI_scroll_cursor_fetch` récupère quelques lignes à partir d'un curseur. C'est équivalent à la commande SQL `FETCH`.

Arguments

`Portal` *`portal`*

portail contenant le curseur

`FetchDirection` *`direction`*

un parmi `FETCH_FORWARD`, `FETCH_BACKWARD`, `FETCH_ABSOLUTE` ou `FETCH_RELATIVE`

`long` *`count`*

nombre de lignes à récupérer pour `FETCH_FORWARD` ou `FETCH_BACKWARD` ; nombre de lignes absolu à récupérer pour `FETCH_ABSOLUTE` ; ou nombre de lignes relatif à récupérer pour `FETCH_RELATIVE`

Valeur de retour

`SPI_processed` et `SPI_tuptable` sont configurés comme `SPI_execute` en cas de succès.

Notes

Voir la commande SQL FETCH pour des détails sur l'interprétation des paramètres *`direction`* et *`count`*.

Les valeurs de direction autres que `FETCH_FORWARD` peuvent échouer si le plan du curseur n'a pas été créé avec l'option `CURSOR_OPT_SCROLL`.

SPI_scroll_cursor_move

SPI_scroll_cursor_move — déplacer un curseur

Synopsis

```
void SPI_scroll_cursor_move(Portal portal,
 FetchDirection direction, long count)
```

Description

`SPI_scroll_cursor_move` ignore un certain nombre de lignes dans un curseur. C'est l'équivalent de la commande SQL `MOVE`.

Arguments

`Portal` *`portal`*

portail contenant le curseur

`FetchDirection` *`direction`*

un parmi `FETCH_FORWARD`, `FETCH_BACKWARD`, `FETCH_ABSOLUTE` et `FETCH_RELATIVE`

`long` *`count`*

nombre de lignes à déplacer pour `FETCH_FORWARD` ou `FETCH_BACKWARD` ; nombre de lignes absolu à déplacer pour `FETCH_ABSOLUTE` ; ou nombre de lignes relatif à déplacer pour `FETCH_RELATIVE`

Valeur de retour

`SPI_processed` est configuré comme `SPI_execute` en cas de succès. `SPI_tuptable` est configuré à `NULL` car aucune ligne n'est renvoyée par cette fonction.

Notes

Voir la commande SQL FETCH pour des détails sur l'interprétation des paramètres *`direction`* et *`count`*.

Les valeurs de direction autres que `FETCH_FORWARD` peuvent échouer si le plan du curseur n'a pas été créé avec l'option `CURSOR_OPT_SCROLL`.

SPI_cursor_close

SPI_cursor_close — ferme un curseur

Synopsis

```
void SPI_cursor_close(Portal portal)
```

Description

`SPI_cursor_close` ferme un curseur créé précédemment et libère la mémoire du portail.

Tous les curseurs ouverts sont fermés automatiquement à la fin de la transaction. `SPI_cursor_close` n'a besoin d'être invoqué que s'il est désirable de libérer les ressources plus tôt.

Arguments

`Portal portal`

portail contenant le curseur

SPI_keepplan

SPI_keepplan — sauvegarde une instruction préparée

Synopsis

```
int SPI_keepplan(SPIPlanPtr plan)
```

Description

`SPI_keepplan` sauvegarde une instruction passée (préparée par `SPI_prepare`) pour qu'elle ne soit pas libérée par `SPI_finish` ou par le gestionnaire des transactions. Cela vous donne la possibilité de réutiliser les instructions préparées dans les prochains appels à votre fonction C dans la session courante.

Arguments

`SPIPlanPtr` *`plan`*

l'instruction préparée à sauvegarder

Valeur de retour

0 en cas de succès ; `SPI_ERROR_ARGUMENT` si *`plan`* vaut `NULL` ou est invalide

Notes

L'instruction passée est relocalisée dans un stockage permanent par l'ajustement de pointeur (pas de copie de données requise). Si vous souhaitez la supprimer plus tard, utilisez `SPI_freeplan`.

SPI_saveplan

SPI_saveplan — sauvegarde une requête préparée

Synopsis

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

Description

`SPI_saveplan` copie une instruction passée (préparée par `SPI_prepare`) en mémoire qui ne serait pas libérée par `SPI_finish` ou par le gestionnaire de transactions, et renvoie un pointeur vers l'instruction copiée. Cela vous donne la possibilité de réutiliser des instructions préparées dans les appels suivants de votre fonction C dans la session courante.

Arguments

`SPIPlanPtr` *`plan`*

la requête préparée à sauvegarder

Valeur de retour

Pointeur vers la requête copiée ; `NULL` en cas d'échec. En cas d'erreur, `SPI_result` est positionnée comme suit :

`SPI_ERROR_ARGUMENT`

si *`plan`* est `NULL` ou invalide

`SPI_ERROR_UNCONNECTED`

si appelé d'une fonction C non connectée

Notes

La requête passée n'est pas libérée, donc vous pouvez souhaiter exécuter `SPI_freeplan` sur ce dernier pour éviter des pertes mémoire jusqu'à `SPI_finish`.

Dans la plupart des cas, `SPI_keepplan` est préférée à cette fonction car elle accomplit largement le même résultat sans avoir besoin de copier physiquement la structure de données des instructions préparées.

SPI_register_relation

SPI_register_relation — rend une relation nommée éphémère disponible par son nom dans les requêtes SPI

Synopsis

```
int SPI_register_relation(EphemeralNamedRelation enr)
```

Description

`SPI_register_relation` rends une relation nommée éphémère - tout comme son information associée - disponible aux requêtes planifiées et exécutées par la connexion SPI en cours.

Arguments

`EphemeralNamedRelation` *`enr`*

l'entrée du registre de la relation nommée éphémère

Valeur de retour

Si l'exécution de la commande a réussi, alors la valeur (non négative) suivante sera retournée :

`SPI_OK_REL_REGISTER`

si la relation a bien été enregistrée avec succès par son nom

En cas d'erreur, une des valeurs négatives suivantes sera retournée :

`SPI_ERROR_ARGUMENT`

si *`enr`* est `NULL` ou si son champ `name` est `NULL`

`SPI_ERROR_UNCONNECTED`

en cas d'appel par une fonction C non connectée

`SPI_ERROR_REL_DUPLICATE`

si le nom spécifié dans le champ `name` de *`enr`* est déjà enregistré pour cette connexion

SPI_unregister_relation

SPI_unregister_relation — supprime une relation nommée éphémère du registre

Synopsis

```
int SPI_unregister_relation(const char * name)
```

Description

`SPI_unregister_relation` supprime une relation nommée éphémère du registre pour la connexion courante

Arguments

`const char * name`

le nom de l'entrée du registre de la relation

Valeur de retour

Si l'exécution de la commande a été réussie, alors la valeur (non négative) suivante sera retournée :

`SPI_OK_REL_UNREGISTER`

si le tuplestore a été correctement supprimé du registre

En cas d'erreur, une valeur négative sera retournée, parmi :

`SPI_ERROR_ARGUMENT`

si *name* est `NULL`

`SPI_ERROR_UNCONNECTED`

si appelé par une fonction C non connectée

`SPI_ERROR_REL_NOT_FOUND`

si *name* est absent du registre pour la connexion courante

SPI_register_trigger_data

SPI_register_trigger_data — rends les données de triggers disponibles dans les requêtes SPI

Synopsis

```
int SPI_register_trigger_data(TriggerData *tdata)
```

Description

Avec `SPI_register_trigger_data`, toutes les relations éphémères capturées par un trigger sont disponibles pour les requêtes planifiées et exécutées par la connexion SPI courante. Actuellement, cela concerne les tables de transition capturées par un trigger `AFTER` défini avec une clause `REFERENCING OLD/NEW TABLE AS` ... Cette fonction doit être appelée par une fonction PL de gestion de trigger après connexion.

Arguments

`TriggerData *tdata`

L'objet `TriggerData` passé à une fonction de gestion de trigger en tant que `fcinfo->context`

Valeur de retour

Si l'exécution de la commande est réussie, alors la valeur (non négative) suivante sera retournée :

`SPI_OK_TD_REGISTER`

si les données capturées par le trigger (s'il y en a) ont été correctement enregistrées

En cas d'erreur, une valeur négative sera retournée, parmi :

`SPI_ERROR_ARGUMENT`

si *`tdata`* est `NULL`

`SPI_ERROR_UNCONNECTED`

si appelé par une fonction C non connectée

`SPI_ERROR_REL_DUPLICATE`

si le nom d'une relation éphémère de données de trigger est déjà enregistré pour cette connexion

47.2. Fonctions de support d'interface

Les fonctions décrites ici donnent une interface pour extraire les informations des séries de résultats renvoyés par `SPI_execute` et les autres fonctions SPI.

Toutes les fonctions décrites dans cette section peuvent être utilisées par toutes les fonctions C, connectées et non connectées.

SPI_fname

SPI_fname — détermine le nom de colonne pour le numéro de colonne spécifié

Synopsis

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

Description

`SPI_fname` retourne une copie du nom de colonne d'une colonne spécifiée (vous pouvez utiliser `pfree` pour libérer la copie du nom lorsque vous n'en avez plus besoin).

Arguments

`TupleDesc` *`rowdesc`*

description de rangée d'entrée

`int` *`colnumber`*

nombre de colonne (le compte commence à 1)

Valeur de retour

Le nom de colonne ; `NULL` si *`colnumber`* est hors de portée. `SPI_result` est positionnée à `SPI_ERROR_NOATTRIBUTE` en cas d'échec.

SPI_fnumber

SPI_fnumber — détermine le numéro de colonne pour le nom de colonne spécifiée

Synopsis

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

Description

`SPI_fnumber` renvoie le numéro de colonne pour la colonne portant le nom spécifié.

Si *`colname`* réfère à une colonne système (c'est-à-dire `ctid`), alors le numéro de colonne négatif approprié sera renvoyé. L'appelant devra faire attention à tester la valeur de retour pour égalité exacte à `SPI_ERROR_NOATTRIBUTE` pour détecter une erreur ; tester le résultat pour une valeur inférieure ou égale à 0 n'est pas correcte sauf si les colonnes systèmes doivent être rejetées.

Arguments

`TupleDesc` *`rowdesc`*

description de la rangée d'entrée

`const char *` *`colname`*

nom de colonne

Valeur de retour

Numéro de colonne (le compte commence à 1 pour les colonnes utilisateurs) ou `SPI_ERROR_NOATTRIBUTE` si la colonne nommée n'est pas trouvée.

SPI_getvalue

SPI_getvalue — renvoie la valeur de chaîne de la colonne spécifiée

Synopsis

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc,
 int colnumber)
```

Description

`SPI_getvalue` retourne la représentation chaîne de la valeur de la colonne spécifiée.

Le résultat est retourné en mémoire allouée en utilisant `palloc` (vous pouvez utiliser `pfree` pour libérer la mémoire lorsque vous n'en avez plus besoin).

Arguments

`HeapTuple` *`row`*

ligne d'entrée à examiner

`TupleDesc` *`rowdesc`*

description de la ligne en entrée

`int` *`colnumber`*

numéro de colonne (le compte commence à 1)

Valeur de retour

Valeur de colonne ou `NULL` si la colonne est NULL, si *`colnumber`* est hors de portée (`SPI_result` est positionnée à `SPI_ERROR_NOATTRIBUTE`) ou si aucune fonction de sortie n'est disponible (`SPI_result` est positionnée à `SPI_ERROR_NOOUTFUNC`).

SPI_getbinval

SPI_getbinval — retourne la valeur binaire de la colonne spécifiée

Synopsis

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc,
 int colnumber, bool * isNULL)
```

Description

`SPI_getbinval` retourne la valeur de la colonne spécifiée dans le format interne (en tant que type `Datum`).

Cette fonction n'alloue pas de nouvel espace pour le datum. Dans le cas d'un type de données passé par référence, la valeur de retour sera un pointeur dans la ligne passée.

Arguments

`HeapTuple` *`row`*

ligne d'entrée à examiner

`TupleDesc` *`rowdesc`*

description de la ligne d'entrée

`int` *`colnumber`*

numéro de colonne (le compte commence à 1)

`bool *` *`isNULL`*

indique une valeur NULL dans la colonne

Valeur de retour

La valeur binaire de la colonne est retournée. La variable vers laquelle pointe *`isNULL`* est positionnée à vrai si la colonne est NULL et sinon à faux.

`SPI_result` est positionnée à `SPI_ERROR_NOATTRIBUTE` en cas d'erreur.

SPI_gettype

SPI_gettype — retourne le nom du type de donnée de la colonne spécifiée

Synopsis

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

Description

`SPI_gettype` retourne une copie du nom du type de donnée de la colonne spécifiée (vous pouvez utiliser `pfree` pour libérer la copie du nom lorsque vous n'en avez plus besoin).

Arguments

`TupleDesc` *`rowdesc`*

description de ligne d'entrée

`int` *`colnumber`*

numéro de colonne (le compte commence à 1)

Valeur de retour

Le nom de type de donnée de la colonne spécifiée ou `NULL` en cas d'erreur. `SPI_result` est positionnée à `SPI_ERROR_NOATTRIBUTE` en cas d'erreur.

SPI_gettypeid

SPI_gettypeid — retourne l'OID de type de donnée de la colonne spécifiée

Synopsis

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

Description

`SPI_gettypeid` retourne l'OID du type de donnée de la colonne spécifiée.

Arguments

`TupleDesc` *`rowdesc`*

description de ligne d'entrée

`int` *`colnumber`*

numéro de colonne (le compte commence à 1)

Valeur de retour

L'OID du type de donnée de la colonne spécifiée ou `InvalidOid` en cas d'erreur. En cas d'erreur, `SPI_result` est positionnée à `SPI_ERROR_NOATTRIBUTE`.

SPI_getrelname

SPI_getrelname — retourne le nom de la relation spécifiée

Synopsis

```
char * SPI_getrelname(Relation rel)
```

Description

`SPI_getrelname` retourne une copie du nom de la relation spécifiée (vous pouvez utiliser `pfree` pour libérer la copie du nom lorsque vous n'en avez plus besoin).

Arguments

`Relation rel`

relation d'entrée

Valeur de retour

Le nom de la relation spécifiée.

SPI_getnspname

SPI_getnspname — renvoie l'espace de noms de la relation spécifiée

Synopsis

```
char * SPI_getnspname(Relation rel)
```

Description

`SPI_getnspname` renvoie une copie du nom de l'espace de nom auquel appartient la `Relation` spécifiée. Ceci est équivalent au schéma de la relation. Vous devriez libérer (`pfree`) la valeur de retour de cette fonction lorsque vous en avez fini avec elle.

Arguments

`Relation rel`

relation en entrée

Valeur de retour

Le nom de l'espace de noms de la relation spécifiée.

SPI_result_code_string

SPI_result_code_string — renvoie un code d'erreur sous la forme d'une chaîne de caractères

Synopsis

```
const char * SPI_result_code_string(int code);
```

Description

`SPI_result_code_string` renvoie une chaîne contenant la représentation du code résultat renvoyé par différentes fonctions ou procédures SPI dans `SPI_result`.

Arguments

`int code`

code résultat

Valeur de retour

Une chaîne représentant le code résultat.

47.3. Gestion de la mémoire

PostgreSQL alloue de la mémoire dans des *contextes mémoire* qui donnent une méthode pratique pour gérer les allocations faites dans plusieurs endroits qui ont besoin de vivre pour des durées différentes. Détruire un contexte libère toute la mémoire qui y était allouée. Donc, il n'est pas nécessaire de garder la trace des objets individuels pour éviter les fuites de mémoire ; à la place, seul un petit nombre de contextes doivent être gérés. `palloc` et les fonctions liées allouent de la mémoire du contexte « courant ».

`SPI_connect` crée un nouveau contexte mémoire et le rend courant. `SPI_finish` restaure le contexte mémoire précédant et détruit le contexte créé par `SPI_connect`. Ces actions garantissent que les allocations temporaires de mémoire faites dans votre fonction C soient réclamées lors de la sortie de la fonction C, évitant les fuites de mémoire.

En revanche, si votre fonction C a besoin de renvoyer un objet dans de la mémoire allouée (tel que la valeur d'un type de donnée passé par référence), vous ne pouvez pas allouer cette mémoire en utilisant `palloc`, au moins pas tant que vous êtes connecté à SPI. Si vous essayez, l'objet sera désalloué par `SPI_finish` et votre fonction C ne fonctionnera pas de manière fiable. Pour résoudre ce problème, utilisez `SPI_palloc` pour allouer de la mémoire pour votre objet de retour. `SPI_palloc` alloue de la mémoire dans le « contexte de mémoire courant », c'est-à-dire le contexte de mémoire qui était courant lorsque `SPI_connect` a été appelée, ce qui est précisément le bon contexte pour une valeur renvoyée à partir de votre fonction C. Plusieurs autres procédures utilitaires décrites dans cette section renvoient également des objets créés dans le contexte mémoire de l'appelant.

Quand `SPI_connect` est appelée, le contexte privé de la fonction C, qui est créé par `SPI_connect`, est nommé le contexte courant. Toute allocation faîte par `palloc`, `repalloc` ou une fonction utilitaire SPI (à part celles décrites dans cette section) sont faites dans ce contexte. Quand une fonction C se déconnecte du gestionnaire SPI (via `SPI_finish`), le contexte courant est restauré au contexte de mémoire courant et toutes les allocations faites dans le contexte de mémoire de la fonction C sont libérées et ne peuvent plus être utilisées.

SPI_palloc

SPI_palloc — alloue de la mémoire dans le contexte de mémoire courant

Synopsis

```
void * SPI_palloc(Size size)
```

Description

`SPI_palloc` alloue de la mémoire dans le contexte de mémoire courant.

Cette fonction peut seulement être utilisée durant une connexion SPI. Sinon, elle renvoie une erreur.

Arguments

`Size size`

taille en octets du stockage à allouer

Valeur de retour

Pointeur vers le nouvel espace de stockage de la taille spécifiée

SPI_repalloc

SPI_repalloc — ré-alloue de la mémoire dans le contexte de mémoire courant

Synopsis

```
void * SPI_repalloc(void * pointer, Size size)
```

Description

`SPI_repalloc` change la taille d'un segment de mémoire alloué auparavant en utilisant `SPI_palloc`.

Cette fonction n'est plus différente du `repalloc` standard. Elle n'est gardée que pour la compatibilité du code existant.

Arguments

`void * pointer`

pointeur vers l'espace de stockage à modifier

`Size size`

taille en octets du stockage à allouer

Valeur de retour

Pointeur vers le nouvel espace de stockage de taille spécifiée avec le contenu copié de l'espace existant

SPI_pfree

SPI_pfree — libère de la mémoire dans le contexte de mémoire courant

Synopsis

```
void SPI_pfree(void * pointer)
```

Description

`SPI_pfree` libère de la mémoire allouée auparavant par `SPI_palloc` ou `SPI_repalloc`.

Cette fonction n'est plus différente du `pfree` standard. Elle n'est conservée que pour la compatibilité du code existant.

Arguments

`void * pointer`

pointeur vers l'espace de stockage à libérer

SPI_copytuple

SPI_copytuple — effectue une copie d'une ligne dans le contexte de mémoire courant

Synopsis

```
HeapTuple SPI_copytuple(HeapTuple row)
```

Description

`SPI_copytuple` crée une copie d'une ligne dans le contexte de mémoire courant. Ceci est normalement utilisé pour renvoyer une ligne modifiée à partir d'un trigger. Dans une fonction déclarée pour renvoyer un type composite, utilisez `SPI_returntuple` à la place.

Cette fonction peut seulement être utilisée durant une connexion SPI. Sinon, elle renvoie NULL et affecte `SPI_ERROR_UNCONNECTED` à `SPI_result`.

Arguments

`HeapTuple` *`row`*

ligne à copier

Valeur de retour

la ligne copiée ou `NULL` en cas d'erreur (voir `SPI_result` pour une indication sur l'erreur)

SPI_returntuple

SPI_returntuple — prépare le renvoi d'une ligne en tant que Datum

Synopsis

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

Description

`SPI_returntuple` crée une copie d'une ligne dans le contexte de l'exécuteur supérieur, la renvoyant sous la forme d'une ligne de type `Datum`. Le pointeur renvoyé a seulement besoin d'être converti en `Datum` via `PointerGetDatum` avant d'être renvoyé.

Cette fonction ne peut être utilisée que pendant une connexion SPI. Sinon, elle renvoie NULL et affecte `SPI_ERROR_UNCONNECTED` à `SPI_result`.

Notez que ceci devrait être utilisé pour les fonctions qui déclarent renvoyer des types composites. Ce n'est pas utilisé pour les triggers ; utilisez `SPI_copytuple` pour renvoyer une ligne modifiée dans un trigger.

Arguments

`HeapTuple` *`row`*

ligne à copier

`TupleDesc` *`rowdesc`*

descripteur pour la ligne (passez le même descripteur chaque fois pour un cache plus efficace)

Valeur de retour

`HeapTupleHeader` pointant vers la ligne copiée ou `NULL` en cas d'erreur (voir `SPI_result` pour une indication sur l'erreur)

SPI_modifytuple

SPI_modifytuple — crée une ligne en remplaçant les champs sélectionnés d'une ligne donnée

Synopsis

```
HeapTuple SPI_modifytuple(Relation rel,
 HeapTuple row, ncols, colnum, Datum * values, const char * nulls)
```

Description

`SPI_modifytuple` crée une nouvelle ligne en retirant les nouvelles valeurs pour les colonnes sélectionnées et en copiant les colonnes de la ligne d'origine à d'autres positions. La ligne d'entrée n'est pas modifiée. La nouvelle ligne est retournée dans le contexte de l'exécuteur supérieur.

Cette fonction ne peut être utilisée que pendant une connexion SPI. Sinon, elle renvoie NULL et affecte `SPI_ERROR_UNCONNECTED` à `SPI_result`.

Arguments

`Relation` *`rel`*

Utilisé seulement en tant que source du descripteur de ligne pour la ligne (passez une relation plutôt qu'un descripteur de ligne est une erreur).

`HeapTuple` *`row`*

rangée à modifier

`int` *`ncols`*

nombre de numéros de colonnes à changer

`int *` *`colnum`*

tableau de longueur *`ncols`*, contenant les numéros de colonnes à modifier (le numéro des colonnes commence à 1)

`Datum *` *`values`*

tableau de longueur *`ncols`*, contenant les nouvelles valeurs pour les colonnes spécifiées

`const char *` *`nulls`*

tableau de longueur *`ncols`*, décrivant les nouvelles valeurs NULL

Si *`nulls`* vaut `NULL`, alors `SPI_modifytuple` suppose qu'aucune valeur n'est NULL. Dans le cas contraire, chaque entrée du tableau *`nulls`* doit valoir `' '` si la nouvelle valeur correspondante est non NULL et `'n'` si la nouvelle valeur correspondante est NULL (dans ce dernier cas, la valeur réelle de l'entrée *`values`* correspondante n'a pas d'importance). Notez que *`nulls`* n'est pas une chaîne de texte. C'est un tableau et, de ce fait, il n'a pas besoin d'un caractère de fin `'\0'`.

Valeur de retour

nouvelle ligne avec modifications, allouée dans le contexte de mémoire courant, ou `NULL` en cas d'erreur (voir `SPI_result` pour une indication de l'erreur)

En cas d'erreur, `SPI_result` est positionnée comme suit :

`SPI_ERROR_ARGUMENT`

si *`colnum`* contient un numéro de colonne invalide (0 ou moins, ou plus que le nombre de colonnes dans *`row`*) *`row`*)

`SPI_ERROR_NOATTRIBUTE`

si *`nocolonne`* contient un numéro de colonne invalide (inférieur ou égal à 0 ou supérieur au numéro de colonne dans *`row`*)

`SPI_ERROR_UNCONNECTED`

si SPI n'est pas actif

SPI_freetuple

SPI_freetuple — libère une ligne allouée dans le contexte de mémoire courant

Synopsis

```
void SPI_freetuple(HeapTuple row)
```

Description

`SPI_freetuple` libère une rangée allouée auparavant dans le contexte de mémoire courant.

Cette fonction n'est plus différente du standard `heap_freetuple`. Elle est gardée juste pour la compatibilité du code existant.

Arguments

`HeapTuple` *`row`*

rangée à libérer

SPI_freetuptable

SPI_freetuptable — libère une série de lignes créée par `SPI_execute` ou une fonction semblable

Synopsis

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

Description

`SPI_freetuptable` libère une série de lignes créée auparavant par une fonction d'exécution de commandes SPI, tel que `SPI_execute`. Par conséquent, cette fonction est souvent appelée avec la variable globale `SPI_tupletable` comme argument.

Cette fonction est utile si une fonction C SPI a besoin d'exécuter de multiples commandes et ne veut pas garder les résultats de commandes précédentes en mémoire jusqu'à sa fin. Notez que toute série de lignes non libérées est libérée quand même lors de `SPI_finish`. De plus, si une sous-transaction est commencée puis annulée lors de l'exécution d'une fonction C SPI, SPI libère automatiquement tous les ensembles de lignes créés lors de l'exécution de la sous-transaction.

À partir de PostgreSQL 9.3, `SPI_freetuptable` contient la logique de sécurité pour protéger contre les demandes dupliquées de suppression à partir du même ensemble de lignes. Avec les versions précédentes, les suppressions dupliquées auraient amené à des crashs.

Arguments

`SPITupleTable * tuptable`

pointeur vers la série de lignes à libérer, ou NULL pour ne rien faire

SPI_freeplan

SPI_freeplan — libère une requête préparée sauvegardée auparavant

Synopsis

```
int SPI_freeplan(SPIPlanPtr plan)
```

Description

`SPI_freeplan` libère une requête préparée retournée auparavant par `SPI_prepare` ou sauvegardée par `SPI_keepplan` ou `SPI_saveplan`.

Arguments

`SPIPlanPtr plan`

pointeur vers la requête à libérer

Valeur de retour

0 en cas de succès ; `SPI_ERROR_ARGUMENT` si *`plan`* est `NULL` ou invalide.

47.4. Gestion des transactions

Il n'est pas possible d'exécuter des commandes de contrôle des transactions, telles que `COMMIT` et `ROLLBACK` via une fonction SPI comme `SPI_execute`. Cependant, il existe des fonctions d'interface séparées qui permettent le contrôle des transactions via SPI.

Il n'est généralement pas sûr et sensible de démarrer et terminer des transactions dans des fonctions définies par l'utilisateur et appelables en SQL sans prendre en compte le contexte dans lequel elles sont appelées. Par exemple, une limite de transaction dans le milieu d'une fonction qui fait partie d'une expression SQL complexe, elle-même partie d'une commande SQL, aura probablement comme résultat des erreurs internes obscures ou des crashs. Les fonctions d'interface présentées ici ont principalement comme but d'être utilisées par les implémentations de langage de procédure pour supporter la gestion des transactions dans les procédures niveau SQL appelées par la commande `CALL`, en prenant en compte le contexte de l'appel à `CALL`. Les procédures utilisant SPI implémentées en C peuvent implémenter la même logique mais les détails de cette implémentation dépassent le cadre de cette documentation.

SPI_commit

SPI_commit — valider la transaction courante

Synopsis

Description

`SPI_commit` valide la transaction en cours. C'est approximativement équivalent à exécuter la commande SQL `COMMIT`. Une fois la transaction validée, une nouvelle transaction est automatiquement démarrée en utilisant les caractéristiques par défaut de la transaction, pour que l'appelant puisse continuer à utiliser les possibilités offertes par SPI. Si la validation échoue, la transaction en cours est en fait annulée et une nouvelle transaction est lancée, et enfin une erreur est levée de la façon habituelle.

`SPI_commit_and_chain` est identique mais la nouvelle transaction est démarrée avec les mêmes caractéristiques que la transaction qui vient de se terminer, tout comme avec la commande SQL `COMMIT AND CHAIN`.

Ces fonctions peuvent seulement être exécutées si la connexion SPI a été configurée comme non atomique dans l'appel à `SPI_connect_ext`.

SPI_rollback

SPI_rollback, SPI_rollback_and_chain — annuler la transaction courante

Synopsis

Description

`SPI_rollback` annule la transaction en cours. C'est approximativement équivalent à exécuter la commande SQL `ROLLBACK`. Après l'annulation de la transaction, une nouvelle transaction est automatiquement démarrée en utilisant les caractéristiques par défaut de la transaction, pour que l'appelant puisse continuer à utiliser les possibilités offertes par SPI.

`SPI_rollback_and_chain` est identique mais la nouvelle transaction est démarrée avec les mêmes caractéristiques que la transaction qui vient de se terminer, tout comme avec la commande SQL `ROLLBACK AND CHAIN`.

Ces fonctions peuvent seulement être exécutées si la connexion SPI a été configurée comme non atomique dans l'appel à `SPI_connect_ext`.

SPI_start_transaction

SPI_start_transaction — fonction obsolète

Synopsis

Description

`SPI_start_transaction` ne fait rien. Elle n'existe que pour la compatibilité du code avec les versions précédentes de PostgreSQL. Elle était requise après un appel à `SPI_commit` ou ou `SPI_rollback`, mais maintenant, ces fonctions lancent automatiquement une nouvelle transaction.

47.5. Visibilité des modifications de données

Les règles suivantes gouvernent la visibilité des modifications de données dans les fonctions qui utilisent SPI (ou tout autre fonction C) :

- Pendant l'exécution de la commande SQL, toute modification de données faite par la commande est invisible à la commande. Par exemple, dans la commande :

  ```
  INSERT INTO a SELECT * FROM a;
  ```

  les lignes insérées sont invisibles à la partie `SELECT`.

- Les modifications effectuées par une commande C sont visibles par toutes les commandes qui sont lancées après C, peu importe qu'elles soient lancées à l'intérieur de C (pendant l'exécution de C) ou après que C soit terminée.

- Les commandes exécutées via SPI à l'intérieur d'une fonction appelée par une commande SQL (soit une fonction ordinaire, soit un trigger) suivent une des règles ci-dessus suivant le commutateur lecture/écriture passé à SPI. Les commandes exécutées en mode lecture seule suivent la première règle : elles ne peuvent pas voir les modifications de la commande appelante. Les commandes exécutées en mode lecture/écriture suivent la deuxième règle : elles peuvent voir toutes les modifications réalisées jusqu'à maintenant.

- Tous les langages standards de procédures initialisent le mode lecture/écriture suivant l'attribut de volatilité de la fonction. Les commandes des fonctions `STABLE` et `IMMUTABLE` sont réalisées en mode lecture seule alors que les fonctions `VOLATILE` sont réalisées en mode lecture/écriture. Alors que les auteurs de fonctions C sont capables de violer cette convention, il est peu probable que cela soit une bonne idée de le faire.

La section suivante contient un exemple qui illustre l'application de ces règles.

47.6. Exemples

Cette section contient un exemple très simple d'utilisation de SPI. La fonction C `execq` prend une commande SQL comme premier argument et un compteur de lignes comme second, exécute la commande en utilisant `SPI_exec` et renvoie le nombre de lignes qui ont été traitées par la commande.

Vous trouverez des exemples plus complexes pour SPI dans l'arborescence source dans `src/test/regress/regress.c` et dans le module spi.

```
#include "postgres.h"

#include "executor/spi.h"
#include "utils/builtins.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(execq);

Datum
execq(PG_FUNCTION_ARGS)
{
    char *command;
    int cnt;
    int ret;
    uint64 proc;

    /* Convert given text object to a C string */
    command = text_to_cstring(PG_GETARG_TEXT_PP(0));
    cnt = PG_GETARG_INT32(1);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;

    /*
     * Si des lignes ont été récupérées,
     * alors les afficher via elog(INFO).
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        SPITupleTable *tuptable = SPI_tuptable;
        TupleDesc tupdesc = tuptable->tupdesc;
        char buf[8192];
        int64 j;

        for (j = 0; j < tuptable->numvals; j++)
        {
            HeapTuple tuple = tuptable->vals[j];
            int i;

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen(buf), sizeof(buf) -
 strlen(buf), " %s%s",
                         SPI_getvalue(tuple, tupdesc, i),
                         (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);
```

```
    PG_RETURN_INT64(proc);
}
```

Voici comment déclarer la fonction après l'avoir compilée en une bibliothèque partagée (les détails sont dans Section 38.10.5) :

```
CREATE FUNCTION execq(text, integer) RETURNS int8
    AS 'filename'
    LANGUAGE C STRICT;
```

Voici une session d'exemple :

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- inséré par execq
INFO:  EXECQ:  1    -- retourné par execq et inséré par l'INSERT
 précédent

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a RETURNING *',
 1);
INFO:  EXECQ:  2    -- 0 + 2, puis exécution arr#té par décompte
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2

 execq
-------
     3              -- 10 est seulement la valeur max, 3 est le
 nombre réel de lignes
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 10 FROM a', 1);
 execq
-------
     3              -- toutes les lignes traitées ; le nombre ne le
 stoppe pas car rien n'est renvoyé
(1 row)

=> SELECT * FROM a;
```

```
 x
----
  0
  1
  2
 10
 11
 12
(6 rows)

=> DELETE FROM a;
DELETE 6
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1                  -- 0 (aucune ligne dans a) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  1
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- 1 (il y avait une ligne dans a) + 1
(2 rows)

-- Ceci montre la règle de visibilité de modifications de données.
-- execq est appelé deux fois et voir un nombre différent de lignes
 à chaque fois :

=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1    -- résultat du premier execq
INFO:  EXECQ:  2
INFO:  EXECQ:  1    -- résultat du deuxième execq
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                  -- 2 lignes * 1 (x dans la première ligne)
 6                  -- 3 lignes (2 + 1 juste inséré) * 2 (x dans la
 deuxième ligne)
(4 rows)
```

Chapitre 48. Processus en tâche de fond (background worker)

PostgreSQL peut être étendu pour lancer du code utilisateur dans des processus séparés. Ces processus sont démarrés, arrêtés et supervisés par `postgres`, ce qui leur permet d'avoir un cycle de vie étroitement lié au statut du serveur. Ces processus sont attachés à la zone de mémoire partagée de PostgreSQL et ont l'option de se connecter aux bases de manière interne ; ils peuvent également exécuter de multiples transactions séquentiellement, comme n'importe quel processus client standard connecté au serveur. De plus, en se liant avec la bibliothèque libpq, ils peuvent se connecter au serveur et se comporter comme une application cliente standard.

> **Avertissement**
>
> Il y a de considérables risques de robustesse et sécurité lorsque l'on utilise des processus background worker. En effet, ceux-ci étant écrits en langage `C`, ils ont un accès total aux données. Les administrateurs désirant activer des modules incluant des processus background worker devraient prendre énormément de précautions. Seuls les modules soigneusement testés devraient être autorisés à lancer des processus background worker.

Les processus en tâche de fond peuvent être initialisés au moment où PostgreSQL est démarré en incluant le nom du module dans `shared_preload_libraries`. Un module qui souhaite fonctionner comme un processus en tâche de fond peut s'enregistrer en appelant `RegisterBackgroundWorker(BackgroundWorker *worker)` dans son `_PG_init()`. Les processus en tâche de fond peuvent également être démarrés après que le système ait démarré et soit en fonctionnement en appelant la fonction `RegisterDynamicBackgroundWorker(BackgroundWorker *worker, BackgroundWorkerHandle **handle)`. À la différence de `RegisterBackgroundWorker`, qui ne peut être appelée que depuis le processus postmaster,`RegisterDynamicBackgroundWorker` doit être appelée depuis un processus client standard ou un processus en tâche de fond.

La structure `BackgroundWorker` est définie ainsi :

```
typedef void (*bgworker_main_type)(Datum main_arg);
typedef struct BackgroundWorker
{
    char        bgw_name[BGW_MAXLEN];
    char        bgw_type[BGW_MAXLEN];
    int         bgw_flags;
    BgWorkerStartTime bgw_start_time;
    int         bgw_restart_time;       /* in seconds, or
 BGW_NEVER_RESTART */
    char        bgw_library_name[BGW_MAXLEN];
    char        bgw_function_name[BGW_MAXLEN];
    Datum       bgw_main_arg;
    char        bgw_extra[BGW_EXTRALEN];
    int         bgw_notify_pid;
} BackgroundWorker;
```

`bgw_name` et `bgw_type` sont des chaînes de caractères à utiliser dans les messages de trace, liste de processus et autres listes similaires. `bgw_type` devrait être identique pour tous les processus en tâche de fond du même type pour qu'il soit possible de grouper ces processus avec une liste des processus par exemple. Par contre, `bgw_name` peut contenir des informations supplémentaires sur ce processus

spécifique. (Typiquement, la chaîne de bgw_name contiendra le type en quelque sort, mais ce n'est pas requis strictement.)

bgw_flags est un masque de bit OR indiquant les capacités que veut le module. Les valeurs possibles sont

BGWORKER_SHMEM_ACCESS

Réclame un accès à la mémoire partagée. C'est requis.

BGWORKER_BACKEND_DATABASE_CONNECTION

Réclame la capacité à établir une connexion à une base à partir de laquelle il peut ensuite exécuter des transactions et des requêtes. Un processus en tâche de fond utilisant BGWORKER_BACKEND_DATABASE_CONNECTION pour se connecter à une base doit aussi s'attacher à la mémoire partagée en utilisant BGWORKER_SHMEM_ACCESS. Dans le cas contraire, son démarrage échouera.

bgw_start_time spécifie l'état du serveur dans lequel postgres devrait démarrer le processus ; les valeurs possibles sont BgWorkerStart_PostmasterStart (démarrer dès que postgres lui-même a fini sa propre initialisation ; les processus réclamant celà ne sont pas éligibles à une connexion à la base de données), BgWorkerStart_ConsistentState (démarrer dès qu'un état cohérent a été atteint sur un serveur esclave en lecture seule, permettant aux processus de se connecter aux bases et d'exécuter des requêtes en lecture seule), et BgWorkerStart_RecoveryFinished (démarrer dès que le système est entré dans un état de lecture-écriture normal). Notez que les deux dernières valeurs sont équivalentes sur un serveur qui n'est pas un esclave en lecture seule. Notez également que ces valeurs indiquent uniquement quand les processus doivent être démarrés ; ils ne s'arrêtent pas quand un état différent est atteint.

bgw_restart_time est un intervalle, en secondes, que postgres doit attendre avant de redémarrer un processus, si celui-ci a subi un arrêt brutal. Cet intervalle peut être une valeur positive ou BGW_NEVER_RESTART, indiquant de ne pas redémarrer le processus suite à un arrêt brutal.

bgw_library_name est le nom d'une bibliothèque dans laquelle le point d'entrée initial pour le processus en tâche de fond devrait être recherché. La bibliothèque nommée sera chargée dynamiquement par le processus en tâche de fond et bgw_function_name sera utiliser pour identifier la fonction à appeler. S'il charge une fonction du code du moteur, il faudrait plutôt le configurer à « postgres ».

bgw_function_name est le nom d'une fonction dans une bibliothèque chargée dynamiquement qui devrait être utilisée comme point d'entrée initial pour un nouveau processus en tâche de fond.

bgw_main_arg est l'argument Datum de la fonction principale du processus. Cette fonction principale devrait prendre un seul argument de type Datum et renvoyer void. bgw_main_arg sera passé comme argument. De plus, la variable globale MyBgworkerEntry pointe vers une copie de la structure BackgroundWorker passé au moment de l'enregistrement ; le processus pourrait trouver utile d'examiner cette structure.

Sur Windows (et partout où EXEC_BACKEND est défini) ou dans des processus en tâche de fond dynamiques, il n'est pas sûr de passer un Datum par référence, il faut le passer par valeur. Si un argument est requis, il est plus sûr de passer un int32 ou toute autre petite valeur et l'utiliser comme un index d'un tableau alloué en mémoire partagée. Si une valeur comme un cstring ou un text est passée, alors le pointeur ne sera pas valide à partir du nouveau processus en tâche de fond.

bgw_extra peut contenir des données supplémentaires à fournir au background worker. Contrairement à bgw_main_arg, cette donnée n'est pas fourni comme argument de la fonction principale du processus. Elle est accessible via la variable MyBgworkerEntry, comme discuté ci-dessus.

bgw_notify_pid est le PID d'un processus client PostgreSQL auquel le postmaster devrait envoyer un signal SIGUSR1 quand le processus est démarré ou quitte. Il devrait valoir 0 pour les processus en tâche de fond enregistrés lors du démarrage du postmaster, ou quand le processus client enregistrant

le processus en tâche de fond ne souhaite pas attendre que le processus en tâche de fond ne démarre. Sinon, il devrait être initialisé à `MyProcPid`.

Une fois démarré, le processus peut se connecter à une base en appelant `BackgroundWorkerInitializeConnection(char *dbname, char *username, uint32 flags)` ou `BackgroundWorkerInitializeConnectionByOid(Oid dboid, Oid useroid, uint32 flags)`. Cela autorise le processus à exécuter des transactions et des requêtes en utilisant l'interface `SPI`. Si `dbname` vaut NULL ou que `dboid` vaut `InvalidOid`, la session n'est pas connectée à une base en particulier, mais les catalogues partagés peuvent être accédés. Si `username` vaut NULL ou que `useroid` vaut `InvalidOid`, le processus sera démarré avec le superutilisateur créé durant `initdb`. Si `BGWORKER_BYPASS_ALLOWCONN` est indiqué pour le paramètre `flags`, il est possible de contourner la restriction de se connecter aux bases de données ne permettant pas une connexions des utilisateurs. Un background worker ne peut être appelé que par une de ces deux fonctions, et seulement une fois. Il n'est pas possible de changer de base de données.

Les signaux sont initialement bloqués jusqu'à ce que le contrôle atteigne la fonction principale du background worker, et doivent être débloqués par elle ; cela permet une personnalisation des gestionnaires de signaux du processus, si nécessaire. Les signaux peuvent être débloqués dans le nouveau processus en appellant `BackgroundWorkerUnblockSignals` et bloqués en appelant `BackgroundWorkerBlockSignals`.

Si `bgw_restart_time` est configuré à `BGW_NEVER_RESTART` pour un processus en tâche de fond ou s'il quitte avec un code de sortie 0, ou encore s'il est terminé par `TerminateBackgroundWorker`, il sera automatiquement désenregistré par le postmaster lors de sa sortie. Sinon, il sera redémarré après que la période de temps configurée via `bgw_restart_time`, ou immédiatement si le postmaster réinitialise l'instance à cause d'une défaillance d'un processus client. Les processus en tâche de fond qui nécessitent de suspendre leur exécution seulement temporairement devraient utiliser un sommeil interruptible plutôt que de quitter. Vérifiez que le drapeau `WL_POSTMASTER_DEATH` est positionné lors de l'appel à cette fonction, et vérifiez le code retour pour une sortie rapide dans le cas d'urgence où `postgres` lui-même se termine.

Quand un processus en tâche de fond est enregistré en utilisant la fonction `RegisterDynamicBackgroundWorker`, le processus client effectuant cet enregistrement peut obtenir des informations concernant le statut du processus en tâche de fond. Les processus clients souhaitant faire cela devraient fournir l'adresse d'un `BackgroundWorkerHandle *` comme second argument pour `RegisterDynamicBackgroundWorker`. Si l'enregistrement du processus en tâche de fond est réussi, ce pointeur sera initialisé avec un handle opaque qui peut alors être fourni à `GetBackgroundWorkerPid(BackgroundWorkerHandle *, pid_t *)` ou `TerminateBackgroundWorker(BackgroundWorkerHandle *)`. `GetBackgroundWorkerPid` peut être utilisé pour interroger le statut du processus en tâche de fond : une valeur de retour valant `BGWH_NOT_YET_STARTED` indique que le processus en tâche de fond n'a pas encore été démarré par le postmaster; `BGWH_STOPPED` indique qu'il a été démarré mais n'est plus en fonctionnement; et `BGWH_STARTED` indique qu'il est actuellement en fonctionnement. Dans le dernier cas, le PID sera également renvoyé via le deuxième argument. `TerminateBackgroundWorker` demande postmaster d'envoyer un signal `SIGTERM` au processus en tâche de fond s'il est en train de fonctionner, et de le désenregistrer dès qu'il ne sera plus en fonctionnement.

Dans certains cas, un processus qui enregistre un processus en tâche de fond peut souhaiter attendre le démarrage du processus en tâche de fond. Ceci peut être fait en initialisant `bgw_notify_pid` à `MyProcPid` et en fournissant ensuite le `BackgroundWorkerHandle *` obtenu au moment de l'enregistrement à la fonction `WaitForBackgroundWorkerStartup(BackgroundWorkerHandle *handle, pid_t *)`. Cette fonctionne bloquera jusqu'à ce que le postmaster ait tenté de démarrer le processus en tâche de fond, ou jusqu'à l'arrêt du postmaster. Si le processus en tâche de fond est en fonctionnement, la valeur retournée sera `BGWH_STARTED`, et le PID sera écrit à l'adresse fournie. Sinon, la valeur de retour sera `BGWH_STOPPED` ou `BGWH_POSTMASTER_DIED`.

Un processus peut aussi attendre l'arrêt d'un autre processus en tâche de fond, en utilisant la fonction `WaitForBackgroundWorkerShutdown(BackgroundWorkerHandle *handle)` et en

passant le `BackgroundWorkerHandle *` obtenu à l'enregistrement. Cette fonction bloquera l'exécution jusqu'à l'arrêt de l'autre processus ou jusqu'à la mort de postmaster. Quand le processus en tâche de fond quitte, la valeur de retour est `BGWH_STOPPED`. Si postmaster meurt, il renverra `BGWH_POSTMASTER_DIED`.

Les processus en tâche de fond peuvent envoyer des messages de notificatio asynchrones, soit en utilisant la commande `NOTIFY` via SPI, soit directement avec `Async_Notify ()`. De telles notifications seront envoyées au moment du commit. Les processus en tâche de fond ne doivent pas s'enregistrer pour recevoir des notifications asynchrones avec la commande `LISTEN`, car il n'existe pas d'infrastructure pour qu'un tel processus puisse consommer ce type de notifications.

Le module contrib `src/test/modules/worker_spi` contient un exemple fonctionnel, qui démontre quelques techniques utiles.

Le nombre maximum de processus en tâche de fond enregistré est limité par max_worker_processes.

Chapitre 49. Décodage logique (Logical Decoding)

PostgreSQL fournit une infrastructure pour envoyer par flux les modifications effectuées en SQL à des consommateurs externes. Cette fonctionnalité peut être utilisée dans plusieurs buts, y compris pour des solutions de réplication ou d'audit.

Les changements sont envoyés dans des flux identifiés par des slots de réplication logique.

Le format dans lequel ces changements sont envoyés est déterminé par le plugin de sortie utilisé. Un plugin d'exemple est fourni dans la distribution de PostgreSQL. Des plugins additionnels peuvent être écrits pour étendre le choix des formats de sortie disponibles sans modifier une seule ligne de code du moteur. Chaque plugin de sortie a accès à chaque nouvelle ligne individuelle produite par les commandes `INSERT`, ainsi que les nouvelles versions de lignes créées par les commandes `UPDATE`. La disponibilité des anciennes versions des lignes dépend de l'identité de réplica configurée (voir `REPLICA IDENTITY`).

Les changements peuvent être consommés soit en utilisant le protocole de réplication par flux (*streaming*, voir Section 55.4 et Section 49.3), soit en appelant des fonctions en SQL (voir Section 49.4). Il est également possible d'écrire de nouvelles méthodes de consommation de sortie d'un slot de réplication sans modifier le code du moteur (voir Section 49.7).

49.1. Exemples de décodage logique

L'exemple suivant explique le contrôle du décodage logique en utilisant l'interface SQL.

Avant de pouvoir utiliser le décodage logique, il est nécessaire de positionner le paramètre wal_level à la valeur `logical` et le paramètre max_replication_slots à 1 au moins. Il sera alors possible de se connecter à la base de données cible (dans l'exemple suivant, `postgres`) en tant que superutilisateur.

```
postgres=# -- Créer un slot nommé 'regression_slot' utilisant le
 plugin de sortie 'test_decoding'
postgres=# SELECT * FROM
 pg_create_logical_replication_slot('regression_slot',
 'test_decoding', false, true);
    slot_name    |    lsn
-----------------+--------------
 regression_slot | 0/16B1970
(1 row)

postgres=# SELECT slot_name, plugin, slot_type, database, active,
 restart_lsn, confirmed_flush_lsn FROM pg_replication_slots;
    slot_name    |    plugin     | slot_type | database | active |
 restart_lsn | confirmed_flush_lsn
-----------------+--------------+-----------+----------+--------
+-------------+-----------------
 regression_slot | test_decoding | logical   | postgres | f      |
 0/16A4408   | 0/16A4440
(1 row)

postgres=# -- Il n'y a pas encore de changement à voir
postgres=# SELECT * FROM
 pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)
```

```
postgres=# CREATE TABLE data(id serial primary key, data text);
CREATE TABLE

postgres=# -- le DDL n'est pas répliqué, donc seule la transaction
 est visible
postgres=# SELECT * FROM
 pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |     data
-----------+-------+--------------
 0/BA2DA58 | 10297 | BEGIN 10297
 0/BA5A5A0 | 10297 | COMMIT 10297
(2 rows)

postgres=# -- Une fois les changements lus, ils sont consommés et
 ne seront pas renvoyés
postgres=# -- dans un appel ultérieur :
postgres=# SELECT * FROM
 pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 lsn | xid | data
-----+-----+------
(0 rows)

postgres=# BEGIN;
postgres=*# INSERT INTO data(data) VALUES('1');
postgres=*# INSERT INTO data(data) VALUES('2');
postgres=*# COMMIT;

postgres=# SELECT * FROM
 pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+
+-------------------------------------------------------------
 0/BA5A688 | 10298 | BEGIN 10298
 0/BA5A6F0 | 10298 | table public.data: INSERT: id[integer]:1
 data[text]:'1'
 0/BA5A7F8 | 10298 | table public.data: INSERT: id[integer]:2
 data[text]:'2'
 0/BA5A8A8 | 10298 | COMMIT 10298
(4 rows)

postgres=# INSERT INTO data(data) VALUES('3');

postgres=# -- Vous pouvez aussi jeter un œil dans le flux sans
 consommer les changements
postgres=# SELECT * FROM
 pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------+
+-------------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3
 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- L'appel suivant va renvoyer les mêmes changements
```

```
postgres=# SELECT * FROM
 pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
    lsn    |  xid  |                          data
-----------+-------
+-------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3
 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299
(3 rows)

postgres=# -- des options peuvent être passées au plugin de sortir
 pour influer sur le formatage
postgres=# SELECT * FROM
 pg_logical_slot_peek_changes('regression_slot', NULL, NULL,
 'include-timestamp', 'on');
    lsn    |  xid  |                          data
-----------+-------
+-------------------------------------------------------
 0/BA5A8E0 | 10299 | BEGIN 10299
 0/BA5A8E0 | 10299 | table public.data: INSERT: id[integer]:3
 data[text]:'3'
 0/BA5A990 | 10299 | COMMIT 10299 (at 2017-05-10
 12:07:21.272494-04)
(3 rows)

postgres=# -- Il ne faut pas oublier de détruire un slot une fois
 qu'on n'en a plus besoin
postgres=# -- afin qu'il ne consomme plus de ressources sur le
 serveur :
postgres=# SELECT pg_drop_replication_slot('regression_slot');
 pg_drop_replication_slot
----------------------

(1 row)
```

Les exemples suivants montrent comment le décodage logique est contrôlé avec le protocole de réplication en flux, en utilisant l'outil pg_recvlogical fourni avec la distribution PostgreSQL. Il requiert que l'authentification du client soit configuré pour autoriser une connexion de réplication (voir Section 27.2.5.1) et que le paramètre `max_wal_senders` soit configuré suffisamment haut pour qu'une nouvelle connexion soit acceptée. Le deuxième exemple montre comment envoyer en flux les transactions en deux phases. Avant d'utiliser des commandes à deux phases, vous devez configurer le paramètre max_prepared_transactions à au moins 1.

```
Exemple 1 :
$ pg_recvlogical -d postgres --slot=test --create-slot
$ pg_recvlogical -d postgres --slot=test --start -f -
Control+Z
$ psql -d postgres -c "INSERT INTO data(data) VALUES('4');"
$ fg
 BEGIN 693
 table public.data: INSERT: id[integer]:4 data[text]:'4'
 COMMIT 693
Control+C
$ pg_recvlogical -d postgres --slot=test --drop-slot

Exemple 2 :
$ pg_recvlogical -d postgres --slot=test --create-slot --two-phase
```

```
$ pg_recvlogical -d postgres --slot=test --start -f -
Control+Z
$ psql -d postgres -c "BEGIN;INSERT INTO data(data)
 VALUES('5');PREPARE TRANSACTION 'test';"
$ fg
BEGIN 694
table public.data: INSERT: id[integer]:5 data[text]:'5'
PREPARE TRANSACTION 'test', txid 694
Control+Z
$ psql -d postgres -c "COMMIT PREPARED 'test';"
$ fg
COMMIT PREPARED 'test', txid 694
Control+C
$ pg_recvlogical -d postgres --slot=test --drop-slot
```

L'exemple suivant montre l'interface SQL pouvant être utilisée pour décoder les transactions préparées. Avant d'utiliser les commandes du Two-Phase Commit, vous devez configurer le paramètre `max_prepared_transactions` à la valeur 1 au minimum. Vous devez aussi avoir configuré le paramètre two-phase à la valeur true lors de la création du slot en utilisant `pg_create_logical_replication_slot`. Notez que la transaction entière sera envoyée dans le flux après la validation de la transaction si elle n'est pas déjà décodée.

```
postgres=# BEGIN;
postgres=*# INSERT INTO data(data) VALUES('5');
postgres=*# PREPARE TRANSACTION 'test_prepared1';

postgres=# SELECT * FROM
 pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    | xid |                          data
-----------+-----
+-----------------------------------------------------------
 0/1689DC0 | 529 | BEGIN 529
 0/1689DC0 | 529 | table public.data: INSERT: id[integer]:3
 data[text]:'5'
 0/1689FC0 | 529 | PREPARE TRANSACTION 'test_prepared1', txid 529
(3 rows)

postgres=# COMMIT PREPARED 'test_prepared1';
postgres=# select * from
 pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    | xid |                    data
-----------+-----+-------------------------------------------
 0/168A060 | 529 | COMMIT PREPARED 'test_prepared1', txid 529
(4 row)

postgres=#-- On peut aussi faire un rollback de la transaction
 préparée
postgres=# BEGIN;
postgres=*# INSERT INTO data(data) VALUES('6');
postgres=*# PREPARE TRANSACTION 'test_prepared2';
postgres=# select * from
 pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    | xid |                          data
-----------+-----
+-----------------------------------------------------------
 0/168A180 | 530 | BEGIN 530
 0/168A1E8 | 530 | table public.data: INSERT: id[integer]:4
 data[text]:'6'
```

```
 0/168A430 | 530 | PREPARE TRANSACTION 'test_prepared2', txid 530
(3 rows)

postgres=# ROLLBACK PREPARED 'test_prepared2';
postgres=# select * from
 pg_logical_slot_get_changes('regression_slot', NULL, NULL);
    lsn    | xid |                      data
-----------+-----+-----------------------------------------------
 0/168A4B8 | 530 | ROLLBACK PREPARED 'test_prepared2', txid 530
(1 row)
```

49.2. Concepts de décodage logique

49.2.1. Décodage logique

Le décodage logique est le processus d'extraction de tous les changements persistants sur des tables d'une base de données, dans un format cohérent et simple à comprendre, et qui peut être interprété sans une connaissance détaillée de l'état interne de la base de données.

Dans PostgreSQL, le décodage logique est implémenté en décodant le contenu des journaux de transaction (WAL), qui décrivent les changements au niveau stockage, dans un format spécifique tel que le flux de lignes ou des ordres SQL.

49.2.2. Slots de réplication

Dans le contexte de la réplication logique, un slot représente un flux de changements qui peut être rejoué par un client, dans l'ordre dans lequel ils ont été effectués sur le serveur d'origine. Chaque slot envoie dans ce flux une séquence de changements d'une unique base.

> **Note**
>
> PostgreSQL possède également des slots de réplication (voir Section 27.2.5), mais ceux-ci sont utilisés de manière un peu différente ici.

Les slots de réplication ont un identifiant qui est unique à travers toutes les bases d'une instance PostgreSQL. Les slots persistent indépendamment de la connexion les utilisant et sont résistants à un arrêt brutal.

Un slot logique émettra chaque modification une fois en temps normal. La position actuelle de chaque slot est enregistrée seulement lors d'un checkpoint, donc dans le cas d'un crash, le slot pourrait revenir à un ancien LSN, qui sera la cause d'un renvoi des changements récents au redémarrage du serveur. Les clients de décodage logique sont responsables de la bonne gestion de ce fait et doivent éviter les mauvais effets dûs à la gestion du même message plusieurs fois. Les clients peuvent souhaiter enregistrer le dernier LSN qu'ils ont vu lors du décodage pour ignorer toute donnée répétée ou (lors de l'utilisation du protocole de réplication) demander que le décodage commence à partir de ce LSN plutôt que de laisser le serveur déterminer le point de démarrage. La fonctionnalité *Replication Progress Tracking* est conçue dans ce but, voir les origines de réplication.

De nombreux slots indépendants peuvent exister pour une même base. Chaque slot possède son propre état, autorisant différents consommateurs à recevoir des changements depuis différents points dans le flux de changement de la base. Pour la plupart des utilisations, un slot séparé sera requis pour chaque consommateur.

Un slot de réplication logique ne sait rien sur l'état du ou des destinataire(s). Il est même possible d'avoir plusieurs destinataires différents utilisant un même slot à des moments différents ; ils ne recevront que les changements à partir du moment où le dernier destinataire a arrêté de les consommer. Un seul destinataire peut consommer les changements d'un slot à un instant donné.

Attention

Les slots de réplication persistent après un arrêt brutal et ne connaissent rien de l'état de leur(s) consommateur(s). Ils empêcheront la suppression automatique des ressources nécessaires même si aucune connexion ne les utilise. Cela consomme de l'espace car aucun des journaux de transactions et aucune des lignes des catalogues systèmes requis ne peuvent être supprimés par `VACUUM` tant qu'ils sont requis par un slot de réplication. Dans les cas extrêmes, cela pourrait causer l'arrêt de la base pour empêcher une réutilisation des identifiants de transactions (voir Section 25.1.5). Par conséquent, si un slot n'est plus nécessaire, il devrait être supprimé.

49.2.3. Plugins de sortie

Les plugins de sortie transforment les données depuis la représentation interne dans les journaux de transactions (WAL) vers le format dont le consommateur d'un slot de réplication a besoin.

49.2.4. Instantanés exportés

Quand un nouveau slot de réplication est créé avec l'interface de la réplication en flux (voir create_replication_slot), un instantané est exporté (voir Section 9.27.5), qui montrera exactement l'état de la base de données après lequel tous les changements seront inclus dans le flux de changement. Cela peut être utilisé pour créer un nouveau réplica en utilisant `SET TRANSACTION SNAPSHOT` pour lire l'état de la base au moment où le slot a été créé. Cette transaction peut alors être utilisée pour exporter l'état de la base à ce point dans le temps, lequel peut ensuite être mis à jour en utilisant le contenu des slots sans perdre le moindre changement.

Les applications qui ne nécessitent pas d'instantané exporté peuvent les supprimer avec l'option `SNAPSHOT 'nothing'`.

49.3. Interface du protocole de réplication par flux

Les commandes

- `CREATE_REPLICATION_SLOT` *`nom_slot`* `LOGICAL` *`plugin_sortie`*
- `DROP_REPLICATION_SLOT` *`nom_slot`* [`WAIT`]
- `START_REPLICATION SLOT` *`nom_slot`* `LOGICAL ...`

sont utilisées pour, respectivement, créer, supprimer et envoyer les modifications à partir d'un slot de réplication. Ces commandes sont seulement disponibles à partir d'une connexion de réplication ; elles ne peuvent pas être utilisées sur une connexion standard, qui n'accepte que les commandes SQL. Voir Section 55.4 pour les détails sur ces commandes.

L'outil pg_recvlogical peut être utilisé pour commander le décodage logique sur une connexion de réplication en flux. (Il utilise ces commandes en interne.)

49.4. Interface SQL de décodage logique

Voir Section 9.27.6 pour une documentation détaillée sur l'API de niveau SQL afin d'interagir avec le décodage logique.

La réplication synchrone (voir Section 27.2.8) est uniquement supportée sur des slots de réplication utilisés au travers de l'interface de réplication en flux. L'interface de fonction et autres interfaces additionnelles ne faisant pas partie du moteur, ils ne gèrent pas la réplication synchrone.

49.5. Catalogues systèmes liés au décodage logique

Les vues `pg_replication_slots` et `pg_stat_replication` fournissent respectivement des informations sur l'état courant des slots de réplication et des connexions de réplication en flux. Ces vues s'appliquent à la fois à la réplication physique et logique. La vue `pg_stat_replication_slots` fournit des statistiques sur les slots de réplication logiques.

49.6. Plugins de sortie de décodage logique

Un exemple de plugin de sortie peut être trouvé dans le sous-répertoire `contrib/test_decoding` de l'arborescence du code source de PostgreSQL.

49.6.1. Fonction d'initialisation

Un plugin de sortie est initialisé en chargeant dynamiquement une bibliothèque partagée avec comme nom de base le nom du plugin de sortie. Le chemin de recherche de bibliothèque habituel est utilisé pour localiser cette bibliothèque. Pour fournir les callbacks de plugins de sortie requis et pour indiquer que la bibliothèque est effectivement un plugin de sortie, elle doit fournir une fonction nommée `_PG_output_plugin_init`. Une structure est passée à cette fonction qui doit la remplir avec les pointeurs des fonctions de callback pour chaque action individuelle.

```
typedef struct OutputPluginCallbacks
{
    LogicalDecodeStartupCB startup_cb;
    LogicalDecodeBeginCB begin_cb;
    LogicalDecodeChangeCB change_cb;
    LogicalDecodeTruncateCB truncate_cb;
    LogicalDecodeCommitCB commit_cb;
    LogicalDecodeMessageCB message_cb;
    LogicalDecodeFilterByOriginCB filter_by_origin_cb;
    LogicalDecodeShutdownCB shutdown_cb;
    LogicalDecodeFilterPrepareCB filter_prepare_cb;
    LogicalDecodeBeginPrepareCB begin_prepare_cb;
    LogicalDecodePrepareCB prepare_cb;
    LogicalDecodeCommitPreparedCB commit_prepared_cb;
    LogicalDecodeRollbackPreparedCB rollback_prepared_cb;
    LogicalDecodeStreamStartCB stream_start_cb;
    LogicalDecodeStreamStopCB stream_stop_cb;
    LogicalDecodeStreamAbortCB stream_abort_cb;
    LogicalDecodeStreamPrepareCB stream_prepare_cb;
    LogicalDecodeStreamCommitCB stream_commit_cb;
    LogicalDecodeStreamChangeCB stream_change_cb;
    LogicalDecodeStreamMessageCB stream_message_cb;
    LogicalDecodeStreamTruncateCB stream_truncate_cb;
} OutputPluginCallbacks;

typedef void (*LogicalOutputPluginInit) (struct
 OutputPluginCallbacks *cb);
```

Les callbacks `begin_cb`, `change_cb` et `commit_cb` sont obligatoires, alors que `startup_cb`, `filter_by_origin_cb`, `truncate_cb` et `shutdown_cb` sont facultatifs. Si `truncate_cb` n'est pas configuré mais que `TRUNCATE` doit être décodé, l'action sera ignorée.

Un plugin de sortie peut aussi définir les fonctions pour accepter le flux de grosses transactions en cours. Les callbacks `stream_start_cb`, `stream_stop_cb`, `stream_abort_cb`,

`stream_commit_cb`, `stream_change_cb` et `stream_prepare_cb` sont obligatoires, alors que `stream_message_cb` et `stream_truncate_cb` sont facultatifs. De plus, si l'envoi en flux des commandes à deux phases doit être supporté, alors des fonctions de support supplémentaires doivent être fournies. (Voir Section 49.10 pour les détails).

Un plugin de sortie peut aussi définir les fonctions pour accepter les validations en deux phases, ce qui permet de décoder un `PREPARE TRANSACTION`. Les callbacks `begin_prepare_cb`, `prepare_cb`, `stream_prepare_cb`, `commit_prepared_cb` et `rollback_prepared_cb` sont obligatoires, alors que `filter_prepare_cb` est facultatif.

49.6.2. Capacités

Pour décoder, formater et afficher les changements, les plugins de sortie peuvent utiliser une grande partie de l'infrastructure habituelle des processus clients, y compris l'appel aux fonctions de sortie. Les accès en lecture seule aux relations est permis du moment que les relations accédées ont été créées par `initdb` dans le schéma `pg_catalog`, ou ont été marquées comme tables du catalogue pour l'utilisateur en utilisant :

```
ALTER TABLE table_catalogue_utilisateur SET (user_catalog_table =
 true);
CREATE TABLE autre_table_catalogue(data text) WITH
 (user_catalog_table = true);
```

Notez que l'accès aux tables utilisateurs ou aux tables systèmes dans le plugin de sortie doit se faire uniquement via les API de parcours `systable_*`. L'accès via les API de parcours `heap_*` renverra une erreur. De plus, toute action amenant à l'affectation d'un identifiant de transaction est prohibée. Ceci, parmi d'autres choses, inclut l'écriture dans les tables, la réalisation de modifications DDL, et l'appel à `pg_current_xact_id()`.

49.6.3. Modes de sortie

Les fonctions callbacks des plugins en sortie peuvent renvoyer des données au consommateur dans des formats pratiquement arbitraires. Pour certains cas d'utilisation, comme la visualisation des changements en SQL, le renvoi des données dans un type de données qui peut contenir des données arbitraires (par exemple du `bytea`) est complexe. Si le plugin en sortie renvoie seulement les données au format texte dans l'encodage du serveur, il peut déclarer cela en configurant `OutputPluginOptions.output_type` à `OUTPUT_PLUGIN_TEXTUAL_OUTPUT` au lieu de `OUTPUT_PLUGIN_BINARY_OUTPUT` dans la fonction callback de démarrage. Dans ce cas, toutes les données doivent être dans l'encodage du serveur pour qu'un champ de type `text` puisse les contenir. Ceci est vérifié dans les constructions comprenant les assertions.

49.6.4. Callbacks de plugin de sortie

Un plugin de sortie est notifié des changements arrivant au travers de différents callbacks qu'il doit fournir.

Les transactions concurrentes sont décodées dans l'ordre dans lequel elles sont validées, et seuls les changements appartenant à une transaction spécifique sont décodés entre les callbacks `begin` et `commit`. Les transactions qui ont été explicitement ou implicitement annulées ne sont jamais décodées. Les savepoints validés sont inclus dans la transaction les contenant, dans l'ordre dans lequel ils ont été effectués dans la transaction. Une transaction qui est préparée pour une validation en deux phases avec `PREPARE TRANSACTION` sera aussi décodée si les callbacks du plugin de sortie nécessaires sont fournies pour le décodage. Il est possible que la transaction préparée courante en cours de décodage soit annulée en parallèle via une commande `ROLLBACK PREPARED`. Dans ce cas, le décodage logique de cette transaction sera lui-aussi annulé. Tous les changements d'une telle transaction seront ignorés une fois que l'annulation est détectée et que la fonction callback `prepare_cb` est appelée. De ce fait, même dans le cas d'une annulation en parallèle, suffisamment

d'informations sont fournies au plugin de sortie pour gérer correctement un `ROLLBACK PREPARED` une fois qu'il est décodé.

> **Note**
>
> Seules les transactions qui ont été synchronisées sur disque de manière sûre seront décodées. Cela peut amener à ce qu'un `COMMIT` ne soit pas immédiatement décodé lors d'un appel à `pg_logical_slot_get_changes()` juste après celui-ci quand `synchronous_commit` est positionné à `off`.

49.6.4.1. Callback de démarrage

Le callback facultatif `startup_cb` est appelé chaque fois qu'un slot de réplication est créé ou qu'on lui demande de fournir les flux de changement, indépendamment du nombre de changements qui sont prêt à être fournis.

```
typedef void (*LogicalDecodeStartupCB) (struct
 LogicalDecodingContext *ctx,
                                         OutputPluginOptions
 *options,
                                         bool is_init);
```

Le paramètre `is_init` sera positionné à true quand le slot de réplication est créé, et à false sinon. *`options`* pointe vers une structure d'options que le plugin de sortie peut positionner :

```
typedef struct OutputPluginOptions
{
    OutputPluginOutputType output_type;
    bool        receive_rewrites;
} OutputPluginOptions;
```

`output_type` doit être positionné soit à `OUTPUT_PLUGIN_TEXTUAL_OUTPUT` ou à `OUTPUT_PLUGIN_BINARY_OUTPUT`. Voir aussi Section 49.6.3. Si `receive_rewrites` vaut true, le plugin de sortie sera aussi appelé pour les modifications réalisées par des réécritures du fichier HEAP lors de certaines opérations DDL. Ceci est intéressant pour les plugins qui gèrent la réplication DDL mais ils nécessitent une gestion particulière.

Le callback de démarrage devrait valider les options présentes dans `ctx->output_plugin_options`. Si le plugin de sortie a besoin d'avoir un état, il peut utiliser `ctx->output_plugin_private` pour le stocker.

49.6.4.2. Callback d'arrêt

Le callback facultatif `shutdown_cb` est appelé chaque fois qu'un slot de réplication anciennement actif n'est plus utilisé et peut être utilisé pour désallouer les ressources privées du plugin de sortie. Le slot n'est pas nécessairement supprimé, le flux est juste arrêté.

```
typedef void (*LogicalDecodeShutdownCB) (struct
 LogicalDecodingContext *ctx);
```

49.6.4.3. Callback de début de transaction

Le callback obligatoire `begin_cb` est appelé chaque fois que le début d'une transaction validée a été décodé. Les transactions annulées et leur contenu ne sont pas décodés.

```
typedef void (*LogicalDecodeBeginCB) (struct LogicalDecodingContext
 *ctx, ReorderBufferTXN *txn);
```

Le paramètre *txn* contient des métadonnées sur la transaction, comme l'heure à laquelle elle a été validée et son XID.

49.6.4.4. Callback de fin de transaction

Le callback obligatoire commit_cb est appelé chaque fois qu'une transaction validée a été décodée. Le callback change_cb aura été appelé avant cela pour chacune des lignes modifiées, s'il y en a eu.

```
typedef void (*LogicalDecodeCommitCB) (struct
 LogicalDecodingContext *ctx,
                                        ReorderBufferTXN *txn,
                                        XLogRecPtr commit_lsn);
```

49.6.4.5. Callback de modification

Le callback obligatoire change_cb est appelé pour chacune des modifications de ligne au sein d'une transaction, qu'il s'agisse d'un INSERT, UPDATE ou DELETE. Même si la commande d'origine a modifié plusieurs ligne en une seule instruction, le callback sera appelé pour chaque ligne individuellement. Le callback change_cb pourrait accéder aux tables systèmes et utilisateurs pour aider au traitement en sortie des détails de modification de la ligne. Dans le cas du décodage d'une transaction préparée (mais pas encore validée) ou du décodage d'une transaction non validée, ce callback de modification pourrait aussi renvoyer une erreur à cause de l'annulation simultanée de cette même transaction. Dans ce cas, le décodage logique de cette transaction annulée est arrêté proprement.

```
typedef void (*LogicalDecodeChangeCB) (struct
 LogicalDecodingContext *ctx,
                                        ReorderBufferTXN *txn,
                                        Relation relation,
                                        ReorderBufferChange
 *change);
```

Les paramètres *ctx* et *txn* ont le même contenu que pour les callbacks begin_cb et commit_cb, mais en plus le descripteur de relation *relation* pointe vers la relation à laquelle appartient la ligne et une structure *change* décrivant les modifications de la ligne y est passée.

> **Note**
>
> Seuls les changements dans les tables définies par les utilisateurs qui sont journalisées (voir UNLOGGED) et non temporaires (voir TEMPORARY ou TEMP) peuvent être extraits avec le décodage logique.

49.6.4.6. Callback Truncate

La fonction callback truncate_cb est appelée pour la commande TRUNCATE.

```
typedef void (*LogicalDecodeTruncateCB) (struct
 LogicalDecodingContext *ctx,
                                          ReorderBufferTXN *txn,
                                          int nrelations,
                                          Relation relations[],
                                          ReorderBufferChange
 *change);
```

Les paramètres sont identiques à ceux du callback change_cb. Néanmoins, comme les actions du TRUNCATE sur les tables liées par clés étrangères doivent être exécutées ensemble, ce callback reçoit un tableau de relations au lieu d'une seule relation. Voir la description de l'instruction TRUNCATE pour les détails.

49.6.4.7. Fonction de filtre sur l'origine

La fonction optionnelle `filter_by_origin_cb` est appelée pour déterminer si les données rejouées à partir de *`origin_id`* ont un intérêt pour le plugin de sortie.

```
typedef bool (*LogicalDecodeFilterByOriginCB) (
    struct LogicalDecodingContext *ctx,
    RepNodeId origin_id
);
```

Le paramètre *`ctx`* a le même contenu que pour les autres fonctions. Aucune information n'est disponible à part l'origine. Pour signaler que les changements provenant du nœud sont hors de propos, elle renvoie true, ce qui permet de les filtrer. Elle renvoie false dans les autres cas. Les autres fonctions ne seront pas appelées pour les transactions et changements qui ont été filtrés.

Ceci est utile pour implémenter des solutions de réplication en cascade ou des solutions de réplication multi-directionnelles. Filtrer par rapport à l'origine permet d'empêcher la réplication dans les deux sens des mêmes modifications dans ce type de configuration. Quand les transactions et les modifications contiennent aussi des informations sur l'origine, le filtre via cette fonction est beaucoup plus efficace.

49.6.4.8. Fonctions personnalisées de message générique

La fonction (callback) `message_cb` est appelée quand un message de décodage logique a été décodé.

```
typedef void (*LogicalDecodeMessageCB) (
    struct LogicalDecodingContext *,
    ReorderBufferTXN *txn,
    XLogRecPtr message_lsn,
    bool transactional,
    const char *prefix,
    Size message_size,
    const char *message
);
```

Le paramètre *`txn`* contient des méta-informations sur la transaction, comme l'horodatage de la validation de la transaction et son identifiant (XID). Notez néanmoins qu'il peut être NULL quand le message n'est pas transactionnel et que le XID n'a pas encore été affecté dans la transaction qui a tracé le message. Le *`lsn`* a la position du message dans les WAL. Le paramètre *`transactional`* indique si le message a été envoyé de façon transactionnelle ou non. De façon similaire au callback de changement, dans le cas du décodage d'une transaction préparée (mais pas encore validée) ou du décodage d'une transaction non validée, ce callback de message pourrait renvoyer une erreur en cas d'annulation simultanée de cette même transaction. Dans ce cas, le décodage logique de cette transaction annulée est stoppé proprement. Le paramètre *`prefix`* est un préfixe arbitraire terminé par un caractère nul qui peut être utilisé pour identifier les messages intéressants pour le plugin courant. Et enfin, le paramètre *`message`* contient le message réel de taille *`message_size`*.

Une attention particulière doit être portée à l'unicité du préfixe que le plugin de sortie trouve intéressant. Utiliser le nom de l'extension ou du plugin de sortie est souvent un bon choix.

49.6.4.9. Fonction de filtre du PREPARE

La fonction callback facultative `filter_prepare_cb` est appelée pour déterminer si les données qui font partie de la transaction de validation en deux phases en cours doivent être considérées pour le décoage à cette étape préparatoire ou plus tard comme une transaction standard en une phase au moment du `COMMIT PREPARED`. Pour signaler que le décodage doit être ignoré, renvoyez `true` ; `false` dans le cas contraire. Quand la fonction callback est définie, `false` est supposé (autrement dit, il n'y a pas de filtrage et toutes les transactions utilisant la validation en deux phases sont décodées aussi en deux phases).

```
typedef bool (*LogicalDecodeFilterPrepareCB) (struct
 LogicalDecodingContext *ctx,
                                               TransactionId xid,
                                               const char *gid);
```

Le paramètre *ctx* a le même contenu que pour les autres fonctions callbacks. Les paramètres *xid* et *gid* fournissent deux façons différentes d'identifier la transaction. La commande COMMIT PREPARED ou ROLLBACK PREPARED ultérieure intègre les identifiants, autorisant au plugin de sortie le choix de celui à utiliser.

La fonction callback pourrait être appelée plusieurs fois par transaction pour décoder, et doit fournir la même réponse statique pour une paire donnée de *xid* et de *gid* à chaque fois qu'elle est appelée.

49.6.4.10. Fonction de préparation de début de la transaction

La fonction callback requise begin_prepare_cb est appelée à chaque fois que le début d'une transaction préparée a été décodée. Le champ *gid*, qui fait partie du paramètre *txn*, peut être utilisé dans cette fonction callback pour vérifier si le plugin a déjà reçu cette commande PREPARE, auquel cas elle peut renvoyer une erreur ou ignorer les changements restants de la transaction.

```
typedef void (*LogicalDecodeBeginPrepareCB) (struct
 LogicalDecodingContext *ctx,
                                              ReorderBufferTXN
 *txn);
```

49.6.4.11. Fonction de transaction préparée

La fonction callback requise prepare_cb est appelée à chaque fois qu'une transaction préparée pour une validation en deux phrases a été décodée. La fonction callback change_cb sera appelée avant celle-ci pour toutes les lignes modifiées, à condition que des lignes aient été modifiées. Le champ *gid*, qui fait partie du paramètre *txn*, peut être utilisé dans cette fonction callback.

```
typedef void (*LogicalDecodePrepareCB) (struct
 LogicalDecodingContext *ctx,
                                         ReorderBufferTXN *txn,
                                         XLogRecPtr prepare_lsn);
```

49.6.4.12. Fonction pour le COMMIT PREPARED d'une transaction

La fonction callback requise commit_prepared_cb est appelée quand une commande COMMIT PREPARED d'une transaction a été décodée. Le champ *gid*, qui fait partie du paramètre *txn*, peut être utilisé dans cette fonction callback.

```
typedef void (*LogicalDecodeCommitPreparedCB) (struct
 LogicalDecodingContext *ctx,
                                                ReorderBufferTXN
 *txn,
                                                XLogRecPtr
 commit_lsn);
```

49.6.4.13. Fonction d'annulation de transaction préparée

La fonction callback requise rollback_prepared_cb est appelée à chaque fois qu'une transaction ROLLBACK PREPARED a été décodée. Le champ *gid*, qui fait partie du paramètre *txn*, peut être utilisé dans cette fonction callback. Les paramètres *prepare_end_lsn* et *prepare_time* peuvent être utilisés pour vérifier si le plugin a reçu la commande PREPARE TRANSACTION auquel cas il peut appliquer l'annulation. Sinon, il peut ignorer l'opération d'annulation. Le paramètre *gid* seul n'est pas suffisant parce que le nœud en amont peut avoir une transaction préparée avec le même identifiant.

```
typedef void (*LogicalDecodeRollbackPreparedCB) (struct
 LogicalDecodingContext *ctx,
                                                  ReorderBufferTXN
 *txn,
                                                  XLogRecPtr
 prepare_end_lsn,
                                                  TimestampTz
 prepare_time);
```

49.6.4.14. Fonction de début de flux

La fonction callback `stream_start_cb` est appelée à l'ouverture d'un bloc de changements en flux à partir d'une transaction en cours.

```
typedef void (*LogicalDecodeStreamStartCB) (struct
 LogicalDecodingContext *ctx,
                                             ReorderBufferTXN *txn);
```

49.6.4.15. Fonction d'arrêt de flux

La fonction callback `stream_stop_cb` est appelée lors de la fermeture d'un bloc de changements en flux provenant d'une transaction en cours.

```
typedef void (*LogicalDecodeStreamStopCB) (struct
 LogicalDecodingContext *ctx,
                                            ReorderBufferTXN *txn);
```

49.6.4.16. Fonction d'annulation du flux

La fonction callback `stream_abort_cb` est appelée lors de l'annulation d'une transaction en flux précédente.

```
typedef void (*LogicalDecodeStreamAbortCB) (struct
 LogicalDecodingContext *ctx,
                                             ReorderBufferTXN *txn,
                                             XLogRecPtr abort_lsn);
```

49.6.4.17. Fonction de préparation du flux

La fonction callback `stream_prepare_cb` est appelée pour préparer une transaction en flux précédente pour faire partie d'une validation en deux phases.

```
typedef void (*LogicalDecodeStreamPrepareCB) (struct
 LogicalDecodingContext *ctx,
                                               ReorderBufferTXN
 *txn,
                                               XLogRecPtr
 prepare_lsn);
```

49.6.4.18. Fonction de validation d'un flux

La fonction callback `stream_commit_cb` est appelée pour valider une transaction en flux précédente.

```
typedef void (*LogicalDecodeStreamCommitCB) (struct
 LogicalDecodingContext *ctx,
                                              ReorderBufferTXN *txn,
                                              XLogRecPtr
 commit_lsn);
```

49.6.4.19. Fonction de changement du flux

La fonction callback `stream_change_cb` est appelée pour un changement dans un bloc de changements de flux (démarqué par des appels à `stream_start_cb` et `stream_stop_cb`). Les modifications réelles ne sont pas affichées car la transaction peut annuler plus tard et que nous ne décodons par les changements pour les transactions annulées.

```
typedef void (*LogicalDecodeStreamChangeCB) (struct
 LogicalDecodingContext *ctx,
                                              ReorderBufferTXN *txn,
                                              Relation relation,
                                              ReorderBufferChange
 *change);
```

49.6.4.20. Fonction de message du flux

La fonction callback `stream_message_cb` est appelée lors de l'envoi d'un message générique dans un bloc de changements en flux (démarqué par des appels à `stream_start_cb` et `stream_stop_cb`). Le contenu du message pour des messages transactionnels n'est pas affiché car la transaction peut s'annuler plus tard et que nous ne décodons pas les changements pour les transactions annulées.

```
typedef void (*LogicalDecodeStreamMessageCB) (struct
 LogicalDecodingContext *ctx,
                                               ReorderBufferTXN
 *txn,
                                               XLogRecPtr
 message_lsn,
                                               bool transactional,
                                               const char *prefix,
                                               Size message_size,
                                               const char *message);
```

49.6.4.21. Fonction troncage du flux

La fonction callback `stream_truncate_cb` est appelée pour une commande `TRUNCATE` dans un bloc de changements en flux (démarqué par des appels à `stream_start_cb` et `stream_stop_cb`).

```
typedef void (*LogicalDecodeStreamTruncateCB) (struct
 LogicalDecodingContext *ctx,
                                                ReorderBufferTXN
 *txn,
                                                int nrelations,
                                                Relation
 relations[],
                                                ReorderBufferChange
 *change);
```

Les paramètres sont analogues à ceux de la fonction callback `stream_change_cb`. Néanmoins, comme les actions de la commande `TRUNCATE` sur des tables connectées par des clés étrangères doivent s'exécuter ensemble, cette fonction callback reçoit un tableau de relations au lieu d'une seule relation. Voir la description de l'instruction TRUNCATE pour les détails.

49.6.5. Fonction pour produire une sortie

Pour pouvoir produire une sortie, les plugins de sortie peuvent écrire des données dans le tampon de sortie `StringInfo` dans `ctx->out` dans les callbacks `begin_cb`, `commit_cb` ou `change_cb`.

Avant d'écrire dans le tampon de sortie, `OutputPluginWrite(ctx, last_write)` doit avoir été appelé pour effectuer l'écriture. *`last_write`* indique si une écriture particulière était la dernière écriture du callback.

L'exemple suivant montre comment sortir des données pour le consommateur d'un plugin de sortie :

```
OutputPluginPrepareWrite(ctx, true);
appendStringInfo(ctx->out, "BEGIN %u", txn->xid);
OutputPluginWrite(ctx, true);
```

49.7. Écrivains de sortie de décodage logique

Il est possible d'ajouter d'autres méthodes de sortie pour le décodage logique. Pour plus de détails, voir `src/backend/replication/logical/logicalfuncs.c`. Principalement, trois fonctions doivent être fournies : une pour lire les journaux de transactions, une pour préparer l'écriture de sortie et une pour préparer la sortie (voir Section 49.6.5).

49.8. Support de la réplication synchrone pour le décodage logique

49.8.1. Overview

Le décodage logique peut être utilisé pour construire des solutions de réplication synchrone avec la même interface utilisateur que la réplication synchrone de la réplication par flux. Pour cela, l'interface de réplication en flux (voir Section 49.3) doit être utilisée pour renvoyer par flux les données. Les clients doivent envoyer des messages `Standby status update (F)` (voir Section 55.4), tout comme le font les clients de réplication par flux.

> **Note**
>
> Un réplica synchrone recevant des changements grâce au décodage logique fonctionnera dans le cadre d'une seule base de données. Puisque, à l'opposé de cela, *`synchronous_standby_names`* est actuellement commun à toutes les instances, cela signifie que cette technique ne marchera pas convenablement si plus d'une base de l'instance est utilisée activement.

49.8.2. Mises en garde

Dans une configuration de réplication synchrone, un verrou deadlock peut survenir si la transaction a verrouillé les tables du catalogue de façon exclusive. Voir Section 49.6.2 pour des informations sur les tables du catalogue utilisateur catalog tables. Ceci survient parce que le décodage logique des transactions peut verrouiller les tables du catalogue pour y accéder. Pour éviter ceci, les utilisateurs doivent s'empêcher de prendre un verrou exclusif sur les tables du catalogue. Ceci peut arriver dans les cas suivants :

- Lancer une commande `LOCK` exclusive sur `pg_class` dans une transaction ;
- Exécuter une commande `CLUSTER` sur `pg_class` dans une transaction ;
- `PREPARE TRANSACTION` après une commande `LOCK` sur `pg_class` et autoriser le décodage logique des transactions en deux phases ;
- `PREPARE TRANSACTION` après une commande `CLUSTER` sur `pg_trigger` et autoriser le décodage logique des transactions en deux phases. Ceci amènera un deadlock seulement quand la table publiée a un trigger ;

- Exécuter une commande `TRUNCATE` sur une table du catalogue dans une transaction.

Notez que ces commandes pouvant causer des deadlocks s'appliquent non seulement aux tables du catalogue système indiquées explicitement mais aussi aux autres tables du catalogue utilisateur.

49.9. Envoi en flux de grosses transactions pour le décodage logique

Les fonctions callbacks simples pour le plugin de sortie (soit `begin_cb`, `change_cb`, `commit_cb` et `message_cb`) sont uniquement appelées quand la transaction se termine avec un COMMIT. Les changements sont toujours décodés à partir des journaux de transactions, mais sont seulement passés au plugin de sortie à la validation de la transaction (et ignorés si la transaction est annulée).

Ceci signifie que, bien que le décodage survient de façon incrémental et pourrait déborder sur disque pour que l'utilisation de la mémoire reste sous contrôle, tous les changements décodés seront transmis au moment de la validation de la transaction (ou plus précisément quand le COMMIT sera décodé du journal de transaction). Suivant la taille de la transaction et la bande passante du réseau, le temps de transfert pourrait augmenter de façon significative le délai du rejeu.

Pour réduire le délai de rejeu causé par les grosses transactions, un plugin de sortie peut fournir des fonctions callback supplémentaires pour accepter un flux incrémental de transactions en cours. Il existe plusieurs fonctions callback requises pour un flux (`stream_start_cb`, `stream_stop_cb`, `stream_abort_cb`, `stream_commit_cb` et `stream_change_cb`) et deux fonctions callback optionnelles (`stream_message_cb` et `stream_truncate_cb`).

Lors du flux d'une transaction en cours, les changements (et les messages) sont envoyés en blocs démarqués par les fonctions `stream_start_cb` et `stream_stop_cb`. Une fois que tous les changements décodés ont été transmis, la transaction peut être validée en utilisant la fonction callback `stream_commit_cb`(ou possiblement annulée en utilisant la fonction callback `stream_abort_cb`). Si les validations en deux phases sont supportées, la transaction peut être préparée en utilisant la fonction callback `stream_prepare_cb`, la commande de validation `COMMIT PREPARED` en utilisant la fonction callback `commit_prepared_cb`, et la commande d'annulation en utilisant la fonction callback `rollback_prepared_cb`.

Une séquence d'exemple d'appels de callbacks pour un flux sur une transaction pourrait ressembler à ceci :

```
stream_start_cb(...);   <-- début du premier bloc de changements
  stream_change_cb(...);
  stream_change_cb(...);
  stream_message_cb(...);
  stream_change_cb(...);
  ...
  stream_change_cb(...);
stream_stop_cb(...);    <-- fin du premier bloc de changements

stream_start_cb(...);   <-- début du deuxième bloc de changements
  stream_change_cb(...);
  stream_change_cb(...);
  stream_change_cb(...);
  ...
  stream_message_cb(...);
  stream_change_cb(...);
stream_stop_cb(...);    <-- fin du deuxième bloc de changements


[a. when using normal commit]
```

```
stream_commit_cb(...);  <-- validation de la transaction en flux

[b. when using two-phase commit]
stream_prepare_cb(...);   <-- prépare la transaction en flux
commit_prepared_cb(...);  <-- commit de la transaction préparée
```

La séquence réelle d'appels de fonctions callback pourrait être plus compliquée bien sûr. Il pourrait y avoir des blocs pour plusieurs transactions en flux, certaines des transactions pourraient être annulées, etc.

De façon similaire au comportement de débordement sur disque, le flux est déclenché quand la quantité totale de changements décodés du WAL (pour toutes les transactions en cours) dépasse la limite définie par le paramètre `logical_decoding_work_mem`. À ce moment-là, la transaction de haut niveau la plus large (mesuré par la quantité de mémoire actuellement utilisée pour les changements décodés) est sélectionnée et envoyée en flux. Néanmoins, dans certains cas, nous avons encore besoin de déborder sur disque même si le flux est activé parce que nous dépassons la limite de mémoire, mais nous n'avons pas encore décodé la ligne complète, par exemple seulement le décodage de l'insertion d'un TOAST, mais pas celle de la table principale.

Même lors de flux de grosses transactions, les changements sont toujours appliqués dans l'ordre des validations, préservant les mêmes garanties que le mode sans flux.

49.10. Support du Two-phase commit pour le décodage logique

Avec les fonctions callbacks simples pour le plugin de sortie (donc `begin_cb`, `change_cb`, `commit_cb` et `message_cb`), les commandes du two-phase commit comme `PREPARE TRANSACTION`, `COMMIT PREPARED` et `ROLLBACK PREPARED` ne sont pas décodées. Alors que `PREPARE TRANSACTION` est ignoré, `COMMIT PREPARED` est décodé comme un `COMMIT` et `ROLLBACK PREPARED` est décodé comme un `ROLLBACK`.

Pour accepter le flux des commandes de la validation en deux phases, un plugin de sortie doit fournir des fonctions callbacks supplémentaires. Il existe plusieurs fonctions callbacks requises pour le two-phase commit (`begin_prepare_cb`, `prepare_cb`, `commit_prepared_cb`, `rollback_prepared_cb` et `stream_prepare_cb`) et une fonction callback facultative(`filter_prepare_cb`).

Si les fonctions callbacks du plugin de sortie sont fournies pour le décodage des commandes de validation en deux phases, alors, sur un `PREPARE TRANSACTION`, les changements de cette transaction sont décodés, passés au plugin de sortie, et la fonction callback `prepare_cb` est appelée. Ceci change de la configuration basique de décodage où les changements sont seulement passés au plugin de sortie quand une transaction est validée. Le début d'une transaction préparée est indiqué par la fonction callback `begin_prepare_cb`.

Quand une transaction préparée est annulée en utilisant `ROLLBACK PREPARED`, alors la fonction callback `rollback_prepared_cb` est appelée, et quand la transaction préparée est validée en utilisant `COMMIT PREPARED`, alors la fonction callback `commit_prepared_cb` est appelée.

En option, le plugin de sortie peut définir des règles de filtres via `filter_prepare_cb` pour décoder uniquement les transactions en deux phases. Ceci peut se faire avec la correspondance de motif sur *`gid`* ou via des recherches en utilisant *`xid`*.

Les utilisateurs qui veulent décoder les transactions préparées doivent faire attention aux points mentionnés ci-dessous :

- Si la transaction préparée a verrouillé des tables systèmes en mode exclusif, alors le décodage de la préparation peut bloquer jusqu'à la validation de la transaction principale.

- La solution de réplication logique qui construit une validation en deux phases distribuée en utilisant cette fonctionnalité peut provoquer un deadlock si la transaction préparée a verrouillé en exclusif des tables systèmes. Pour éviter ceci, les utilisateurs doivent éviter d'avoir des verrous sur des tables systèmes (comme une commande `LOCK` explicite) dans de telles transactions. Voir Section 49.8.2 pour les détails.

Chapitre 50. Tracer la progression de la réplication

Les origines de réplication ont pour but de rendre plus simple les solutions de réplication logique utilisant le décodage logique. Elles fournissent une solution à deux problèmes habituels :

- comment suivre la progression de la réplication de manière fiable ;
- comment modifier le comportement de la réplication basée sur l'origine d'une ligne ; par exemple pour empêcher les boucles dans les configurations de réplication bidirectionnelle.

Les origines de réplication n'ont que deux propriétés, un nom et un identifiant. Le nom, qui doit être utilisé pour faire référence à l'origine entre les systèmes, est une donnée libre de type `text`. Il doit être utilisé d'une façon qui rend improbable les conflits entre des origines de réplication créées par différentes solutions de réplication, par exemple en préfixant le nom avec celui de la solution de réplication. L'identifiant est utilisé seulement pour éviter d'avoir à stocker la version longue dans les situations où l'espace consommé est critique. Il ne doit jamais être partagé entre plusieurs systèmes.

Les origines de réplication peuvent être créées en utilisant la fonction `pg_replication_origin_create()`, supprimées avec la fonction `pg_replication_origin_drop()` et consultées dans le catalogue système `pg_replication_origin`.

Une partie non triviale de la construction d'une solution de réplication est le suivi de la progression de la réplication d'une manière fiable. Quand le processus d'application des modifications ou l'instance complète meurt, il doit être possible de savoir jusqu'où les données ont été répliquées. Les solutions naïves, comme la mise à jour d'une ligne pour chaque transaction rejouée, ont leurs problèmes, comme une surcharge à l'exécution et une fragmentation de la base de données.

En utilisant l'infrastructure d'origine de réplication, une session peut être marquée comme rejouant depuis un nœud distant (en utilisant la fonction `pg_replication_origin_session_setup()`). De plus, le LSN et l'horodatage de la validation de toute transaction source peuvent être configurés, transaction par transaction, en utilisant `pg_replication_origin_xact_setup()`. Si cela est fait, la progression de la réplication sera conservée de manière pérenne, même en cas de crash. La progression du rejeu pour toutes les origines de réplication peut être visualisée dans la vue `pg_replication_origin_status`. Le progrès d'une origine précise, par exemple lors de la reprise de la réplication, peut se faire en utilisant la fonction `pg_replication_origin_progress()` pour toute origine ou la fonction `pg_replication_origin_session_progress()` pour l'origine configurée dans la session courante.

Dans les topologies de réplication plus complexes que la réplication d'un système vers un autre système, un autre problème peut être la difficulté d'éviter la réplication de lignes déjà rejouées. Ceci peut mener à des cycles et une mauvaise efficacité dans la réplication. Les origines de réplication fournissent un mécanisme optionnel pour reconnaître et empêcher cela. Lorsqu'elles sont configurées en utilisant les fonctions évoquées dans le paragraphe précédent, chaque modification et chaque transaction passée aux fonctions de rappel (*callbacks*) des plugins en sortie (voir Section 49.6) générées par la session sont tracées avec l'origine de réplication de la session qui les a générées. Ceci permet de les traiter différemment par le plugin de sortie, et par exemple d'ignorer toutes les lignes qui ne proviennent pas de l'origine. De plus, la fonction de rappel `filter_by_origin_cb` peut être utilisée pour filtrer le flux de modifications de décodage logique basé sur la source. Bien que moins flexible, le filtre via cette fonction est considérablement plus efficace que le filtre d'un plugin de sortie.

Chapitre 51. Modules d'archivage

PostgreSQL fournit l'infrastructure nécessaire à la création de modules personnalisés pour l'archivage continu (voir Section 26.3). Alors que l'archivage via une commande shell (par exemple, archive_command) est bien plus simple, un module d'archivage personnalisé sera souvent considéré plus robuste et plus performant.

Quand un archive_library personnalisé est configuré, PostgreSQL soumettra les fichiers segment WAL complets au module, et le serveur évitera de recycler ou supprimer ces fichiers segment WAL jusqu'à ce que le module indique qu'ils ont été archivés avec succès. C'est au final au module de décider quoi faire avec chaque fichier segment WAL, mais de nombreuses recommandations sont listées sur Section 26.3.1.

Les modules d'archivage doivent au moins comporter une fonction d'initialisation (voir Section 51.1) ainsi que les fonctions de support requises (voir Section 51.2). Cependant, les modules d'archivage peuvent également faire bien plus (par exemple, déclarer des paramètres de configuration et démarrer des processus d'arrière-plan).

Le module `contrib/basic_archive` contient un exemple fonctionnel, qui démontre quelques techniques utiles.

51.1. Fonctions d'initialisation

Une bibliothèque d'archivage est chargée en appelant dynamiquement une bibliothèque partagée dont le nom de base est fourni par le paramètre archive_library. Le chemin habituel de recherche des bibliothèques est utilisé pour localiser cette bibliothèque. Pour fournir les fonctions de support nécessaires au module d'archivage et pour indiquer que la bibliothèque est en fait un module d'archivage, une fonction `_PG_archive_module_init` doit y être fournie. Cette fonction reçoit une structure définie par les pointeurs des fonctions de support pour chaque action individuelle.

```
typedef struct ArchiveModuleCallbacks
{
    ArchiveCheckConfiguredCB check_configured_cb;
    ArchiveFileCB archive_file_cb;
    ArchiveShutdownCB shutdown_cb;
} ArchiveModuleCallbacks;
typedef void (*ArchiveModuleInit) (struct ArchiveModuleCallbacks
 *cb);
```

Seule la fonction de support `archive_file_cb` est requise. Les autres sont optionnelles.

51.2. Fonctions de support d'un module d'archivage

Les fonctions de support d'archivage définissent le comportement d'archivage proprement dit du module. Le serveur les appellera suivant les besoins pour traiter chaque fichier segment WAL individuellement.

51.2.1. Fonction de vérification

La fonction de support `check_configured_cb` est appelée pour déterminer si le module est pleinement configuré et prêt à traiter des fichiers segment WAL (par exemple, si ses paramètres de configuration sont définis avec des valeurs valides). Si la fonction `check_configured_cb` n'est pas définie, le serveur présumera que le module est configuré.

```
typedef bool (*ArchiveCheckConfiguredCB) (void);
```

Si `true` est renvoyé, le serveur procédera à l'archivage en appelant la fonction `archive_file_cb`. Si `false` est renvoyé, l'archivage ne sera pas effectué, et l'archiver émettra le message suivant dans les traces du serveur :

```
WARNING:  archive_mode enabled, yet archiving is not configured
```

Dans ce dernier cas, le serveur appellera périodiquement cette fonction, et l'archivage ne sera effectué que lorsque le booléen `true` sera renvoyé.

51.2.2. Fonction d'archivage

La fonction `archive_file_cb` est appelée pour archiver un fichier segment WAL individuel.

```
typedef bool (*ArchiveFileCB) (const char *file, const char *path);
```

Si la valeur `true` est renvoyée, le serveur procédera comme si le fichier était archivé avec succès, ce qui peut inclure de recycler ou supprimer le fichier segment WAL original. Si la valeur `false` est renvoyée, le serveur conservera le fichier segment WAL original et réessayera de l'archiver plus tard. *`file`* contiendra juste le nom du fichier du segment WAL à archiver, alors que `path` contiendra le chemin complet du fichier segment WAL (nom du fichier inclus).

51.2.3. Fonction d'arrêt

La fonction `shutdown_cb` est appelée quand le processus archiver s'arrête (par exemple, après une erreur) ou si la valeur de archive_library change. Si la fonction `shutdown_cb` n'est pas définie, aucune action spécifique ne sera entreprise lors de ces situations.

```
typedef void (*ArchiveShutdownCB) (void);
```

Partie VI. Référence

Les points abordés dans ce référentiel ont pour objectif de fournir, de manière concise, un résumé précis, complet, formel et faisant autorité sur leurs sujets respectifs. Des informations complémentaires sur l'utilisation de PostgreSQL sont présentées, dans d'autres parties de cet ouvrage, sous la forme de descriptions, de tutoriels ou d'exemples. On pourra se reporter à la liste de références croisées disponible sur chaque page de référence.

Les entrées du référentiel sont également disponibles sous la forme de pages « man » traditionnelles.

Table des matières

I. Commandes SQL 1587
ABORT 1591
ALTER AGGREGATE 1592
ALTER COLLATION 1594
ALTER CONVERSION 1597
ALTER DATABASE 1599
ALTER DEFAULT PRIVILEGES 1602
ALTER DOMAIN 1606
ALTER EVENT TRIGGER 1610
ALTER EXTENSION 1611
ALTER FOREIGN DATA WRAPPER 1615
ALTER FOREIGN TABLE 1617
ALTER FUNCTION 1622
ALTER GROUP 1626
ALTER INDEX 1628
ALTER LANGUAGE 1631
ALTER LARGE OBJECT 1632
ALTER MATERIALIZED VIEW 1633
ALTER OPERATOR 1635
ALTER OPERATOR CLASS 1637
ALTER OPERATOR FAMILY 1638
ALTER POLICY 1642
ALTER PROCEDURE 1644
ALTER PUBLICATION 1647
ALTER ROLE 1650
ALTER ROUTINE 1654
ALTER RULE 1656
ALTER SCHEMA 1657
ALTER SEQUENCE 1658
ALTER SERVER 1662
ALTER STATISTICS 1664
ALTER SUBSCRIPTION 1666
ALTER SYSTEM 1669
ALTER TABLE 1671
ALTER TABLESPACE 1690
ALTER TEXT SEARCH CONFIGURATION 1692
ALTER TEXT SEARCH DICTIONARY 1694
ALTER TEXT SEARCH PARSER 1696
ALTER TEXT SEARCH TEMPLATE 1697
ALTER TRIGGER 1698
ALTER TYPE 1700
ALTER USER 1705
ALTER USER MAPPING 1706
ALTER VIEW 1708
ANALYZE 1711
BEGIN 1714
CALL 1716
CHECKPOINT 1718
CLOSE 1719
CLUSTER 1720
COMMENT 1723
COMMIT 1728
COMMIT PREPARED 1729
COPY 1730
CREATE ACCESS METHOD 1741

CREATE AGGREGATE 1742
CREATE CAST 1750
CREATE COLLATION 1755
CREATE CONVERSION 1758
CREATE DATABASE 1760
CREATE DOMAIN 1765
CREATE EVENT TRIGGER 1768
CREATE EXTENSION 1770
CREATE FOREIGN DATA WRAPPER 1773
CREATE FOREIGN TABLE 1775
CREATE FUNCTION 1780
CREATE GROUP 1790
CREATE INDEX 1791
CREATE LANGUAGE 1801
CREATE MATERIALIZED VIEW 1804
CREATE OPERATOR 1806
CREATE OPERATOR CLASS 1809
CREATE OPERATOR FAMILY 1812
CREATE POLICY 1813
CREATE PROCEDURE 1820
CREATE PUBLICATION 1824
CREATE ROLE 1829
CREATE RULE 1834
CREATE SCHEMA 1837
CREATE SEQUENCE 1840
CREATE SERVER 1844
CREATE STATISTICS 1846
CREATE SUBSCRIPTION 1850
CREATE TABLE 1854
CREATE TABLE AS 1878
CREATE TABLESPACE 1881
CREATE TEXT SEARCH CONFIGURATION 1883
CREATE TEXT SEARCH DICTIONARY 1885
CREATE TEXT SEARCH PARSER 1887
CREATE TEXT SEARCH TEMPLATE 1889
CREATE TRANSFORM 1891
CREATE TRIGGER 1894
CREATE TYPE 1902
CREATE USER 1912
CREATE USER MAPPING 1913
CREATE VIEW 1915
DEALLOCATE 1921
DECLARE 1922
DELETE 1926
DISCARD 1929
DO 1931
DROP ACCESS METHOD 1933
DROP AGGREGATE 1934
DROP CAST 1936
DROP COLLATION 1937
DROP CONVERSION 1938
DROP DATABASE 1939
DROP DOMAIN 1941
DROP EVENT TRIGGER 1942
DROP EXTENSION 1943
DROP FOREIGN DATA WRAPPER 1945
DROP FOREIGN TABLE 1946
DROP FUNCTION 1948

DROP GROUP 1950
DROP INDEX 1951
DROP LANGUAGE 1953
DROP MATERIALIZED VIEW 1954
DROP OPERATOR 1955
DROP OPERATOR CLASS 1957
DROP OPERATOR FAMILY 1959
DROP OWNED 1961
DROP POLICY 1962
DROP PROCEDURE 1963
DROP PUBLICATION 1966
DROP ROLE 1967
DROP ROUTINE 1968
DROP RULE 1970
DROP SCHEMA 1971
DROP SEQUENCE 1973
DROP SERVER 1974
DROP STATISTICS 1975
DROP SUBSCRIPTION 1976
DROP TABLE 1978
DROP TABLESPACE 1979
DROP TEXT SEARCH CONFIGURATION 1980
DROP TEXT SEARCH DICTIONARY 1981
DROP TEXT SEARCH PARSER 1982
DROP TEXT SEARCH TEMPLATE 1983
DROP TRANSFORM 1984
DROP TRIGGER 1986
DROP TYPE 1987
DROP USER 1988
DROP USER MAPPING 1989
DROP VIEW 1990
END 1991
EXECUTE 1992
EXPLAIN 1993
FETCH 1999
GRANT 2003
IMPORT FOREIGN SCHEMA 2009
INSERT 2011
LISTEN 2019
LOAD 2021
LOCK 2022
MERGE 2025
MOVE 2031
NOTIFY 2033
PREPARE 2036
PREPARE TRANSACTION 2039
REASSIGN OWNED 2041
REFRESH MATERIALIZED VIEW 2042
REINDEX 2044
RELEASE SAVEPOINT 2050
RESET 2052
REVOKE 2053
ROLLBACK 2057
ROLLBACK PREPARED 2058
ROLLBACK TO SAVEPOINT 2059
SAVEPOINT 2061
SECURITY LABEL 2063
SELECT 2066

SELECT INTO 2089
SET 2091
SET CONSTRAINTS 2094
SET ROLE 2096
SET SESSION AUTHORIZATION 2098
SET TRANSACTION 2100
SHOW 2103
START TRANSACTION 2105
TRUNCATE 2106
UNLISTEN 2109
UPDATE 2111
VACUUM 2116
VALUES 2121
II. Applications client de PostgreSQL 2124
clusterdb 2125
createdb 2128
createuser 2132
dropdb 2137
dropuser 2140
ecpg 2143
pg_amcheck 2146
pg_basebackup 2152
pgbench 2162
pg_config 2188
pg_dump 2191
pg_dumpall 2206
pg_isready 2214
pg_receivewal 2216
pg_recvlogical 2221
pg_restore 2225
pg_verifybackup 2235
psql 2238
reindexdb 2285
vacuumdb 2289
III. Applications relatives au serveur PostgreSQL 2295
initdb 2296
pg_archivecleanup 2301
pg_checksums 2303
pg_controldata 2305
pg_ctl 2306
pg_resetwal 2312
pg_rewind 2316
pg_test_fsync 2320
pg_test_timing 2321
pg_upgrade 2325
pg_waldump 2335
postgres 2338
postmaster 2346

Commandes SQL

Cette partie regroupe les informations de référence concernant les commandes SQL reconnues par PostgreSQL. Généralement, on désigne par « SQL » le langage ; toute information sur la structure et la compatibilité standard de chaque commande peut être trouvée sur les pages référencées.

Table des matières

ABORT 1591
ALTER AGGREGATE 1592
ALTER COLLATION 1594
ALTER CONVERSION 1597
ALTER DATABASE 1599
ALTER DEFAULT PRIVILEGES 1602
ALTER DOMAIN 1606
ALTER EVENT TRIGGER 1610
ALTER EXTENSION 1611
ALTER FOREIGN DATA WRAPPER 1615
ALTER FOREIGN TABLE 1617
ALTER FUNCTION 1622
ALTER GROUP 1626
ALTER INDEX 1628
ALTER LANGUAGE 1631
ALTER LARGE OBJECT 1632
ALTER MATERIALIZED VIEW 1633
ALTER OPERATOR 1635
ALTER OPERATOR CLASS 1637
ALTER OPERATOR FAMILY 1638
ALTER POLICY 1642
ALTER PROCEDURE 1644
ALTER PUBLICATION 1647
ALTER ROLE 1650
ALTER ROUTINE 1654
ALTER RULE 1656
ALTER SCHEMA 1657
ALTER SEQUENCE 1658
ALTER SERVER 1662
ALTER STATISTICS 1664
ALTER SUBSCRIPTION 1666
ALTER SYSTEM 1669
ALTER TABLE 1671
ALTER TABLESPACE 1690
ALTER TEXT SEARCH CONFIGURATION 1692
ALTER TEXT SEARCH DICTIONARY 1694
ALTER TEXT SEARCH PARSER 1696
ALTER TEXT SEARCH TEMPLATE 1697
ALTER TRIGGER 1698
ALTER TYPE 1700
ALTER USER 1705
ALTER USER MAPPING 1706
ALTER VIEW 1708
ANALYZE 1711
BEGIN 1714
CALL 1716
CHECKPOINT 1718
CLOSE 1719

CLUSTER 1720
COMMENT 1723
COMMIT 1728
COMMIT PREPARED 1729
COPY 1730
CREATE ACCESS METHOD 1741
CREATE AGGREGATE 1742
CREATE CAST 1750
CREATE COLLATION 1755
CREATE CONVERSION 1758
CREATE DATABASE 1760
CREATE DOMAIN 1765
CREATE EVENT TRIGGER 1768
CREATE EXTENSION 1770
CREATE FOREIGN DATA WRAPPER 1773
CREATE FOREIGN TABLE 1775
CREATE FUNCTION 1780
CREATE GROUP 1790
CREATE INDEX 1791
CREATE LANGUAGE 1801
CREATE MATERIALIZED VIEW 1804
CREATE OPERATOR 1806
CREATE OPERATOR CLASS 1809
CREATE OPERATOR FAMILY 1812
CREATE POLICY 1813
CREATE PROCEDURE 1820
CREATE PUBLICATION 1824
CREATE ROLE 1829
CREATE RULE 1834
CREATE SCHEMA 1837
CREATE SEQUENCE 1840
CREATE SERVER 1844
CREATE STATISTICS 1846
CREATE SUBSCRIPTION 1850
CREATE TABLE 1854
CREATE TABLE AS 1878
CREATE TABLESPACE 1881
CREATE TEXT SEARCH CONFIGURATION 1883
CREATE TEXT SEARCH DICTIONARY 1885
CREATE TEXT SEARCH PARSER 1887
CREATE TEXT SEARCH TEMPLATE 1889
CREATE TRANSFORM 1891
CREATE TRIGGER 1894
CREATE TYPE 1902
CREATE USER 1912
CREATE USER MAPPING 1913
CREATE VIEW 1915
DEALLOCATE 1921
DECLARE 1922
DELETE 1926
DISCARD 1929
DO 1931
DROP ACCESS METHOD 1933
DROP AGGREGATE 1934
DROP CAST 1936
DROP COLLATION 1937
DROP CONVERSION 1938
DROP DATABASE 1939

DROP DOMAIN 1941
DROP EVENT TRIGGER 1942
DROP EXTENSION 1943
DROP FOREIGN DATA WRAPPER 1945
DROP FOREIGN TABLE 1946
DROP FUNCTION 1948
DROP GROUP 1950
DROP INDEX 1951
DROP LANGUAGE 1953
DROP MATERIALIZED VIEW 1954
DROP OPERATOR 1955
DROP OPERATOR CLASS 1957
DROP OPERATOR FAMILY 1959
DROP OWNED 1961
DROP POLICY 1962
DROP PROCEDURE 1963
DROP PUBLICATION 1966
DROP ROLE 1967
DROP ROUTINE 1968
DROP RULE 1970
DROP SCHEMA 1971
DROP SEQUENCE 1973
DROP SERVER 1974
DROP STATISTICS 1975
DROP SUBSCRIPTION 1976
DROP TABLE 1978
DROP TABLESPACE 1979
DROP TEXT SEARCH CONFIGURATION 1980
DROP TEXT SEARCH DICTIONARY 1981
DROP TEXT SEARCH PARSER 1982
DROP TEXT SEARCH TEMPLATE 1983
DROP TRANSFORM 1984
DROP TRIGGER 1986
DROP TYPE 1987
DROP USER 1988
DROP USER MAPPING 1989
DROP VIEW 1990
END 1991
EXECUTE 1992
EXPLAIN 1993
FETCH 1999
GRANT 2003
IMPORT FOREIGN SCHEMA 2009
INSERT 2011
LISTEN 2019
LOAD 2021
LOCK 2022
MERGE 2025
MOVE 2031
NOTIFY 2033
PREPARE 2036
PREPARE TRANSACTION 2039
REASSIGN OWNED 2041
REFRESH MATERIALIZED VIEW 2042
REINDEX 2044
RELEASE SAVEPOINT 2050
RESET 2052
REVOKE 2053

ROLLBACK 2057
ROLLBACK PREPARED 2058
ROLLBACK TO SAVEPOINT 2059
SAVEPOINT 2061
SECURITY LABEL 2063
SELECT 2066
SELECT INTO 2089
SET 2091
SET CONSTRAINTS 2094
SET ROLE 2096
SET SESSION AUTHORIZATION 2098
SET TRANSACTION 2100
SHOW 2103
START TRANSACTION 2105
TRUNCATE 2106
UNLISTEN 2109
UPDATE 2111
VACUUM 2116
VALUES 2121

ABORT

ABORT — Interrompre la transaction en cours

Synopsis

```
ABORT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Description

`ABORT` annule la transaction en cours et toutes les mises à jour effectuées pendant cette transaction. Cette commande a un comportement identique à la commande SQL `ROLLBACK`. Elle n'est présente que pour des raisons historiques.

Paramètres

`WORK`
`TRANSACTION`

Mots-clé optionnels. Ils n'ont aucun effet.

`AND CHAIN`

Si la clause `AND CHAIN` est indiquée, une nouvelle transaction est immédiatement démarrée avec les mêmes caractéristiques de transaction (voir `SET TRANSACTION`) que celle qui vient de se terminer. Sinon aucune nouvelle transaction n'est démarrée.

Notes

`COMMIT` est utilisé pour terminer avec succès une transaction.

Exécuter `ABORT` à l'extérieur de toute transaction provoque un message d'avertissement mais ne cause aucun dégât.

Exemples

Annuler toutes les modifications :

```
ABORT;
```

Compatibilité

Cette commande est une extension PostgreSQL présente pour des raisons historiques. `ROLLBACK` est la commande équivalente du standard SQL.

Voir aussi

BEGIN, COMMIT, ROLLBACK

ALTER AGGREGATE

ALTER AGGREGATE — Modifier la définition d'une fonction d'agrégat

Synopsis

```
ALTER AGGREGATE nom ( signature_agrégat ) RENAME TO nouveau_nom
ALTER AGGREGATE nom ( signature_agrégat )
                OWNER TO { nouveau_propriétaire | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER }
ALTER AGGREGATE nom ( signature_agrégat ) SET SCHEMA nouveau_schéma

  où signature_agrégat correspond à :

* |
[ mode_arg ] [ nom_arg ] type_arg [ , ... ] |
[ [ mode_arg ] [ nom_arg ] type_arg [ , ... ] ] ORDER BY [ mode_arg
 ] [ nom_arg ] type_arg [ , ... ]
```

Description

`ALTER AGGREGATE` change la définition d'une fonction d'agrégat.

Seul le propriétaire de la fonction d'agrégat peut utiliser `ALTER AGGREGATE`. Pour modifier le schéma d'une fonction d'agrégat, il est nécessaire de posséder le droit `CREATE` sur le nouveau schéma. Pour modifier le propriétaire de la fonction, il faut être un membre direct ou indirect du nouveau rôle propriétaire, rôle qui doit en outre posséder le droit `CREATE` sur le schéma de la fonction d'agrégat. Ces restrictions assurent que la modification du propriétaire ne permet pas d'aller au-delà de ce que permet la suppression et la recréation d'une fonction d'agrégat. Toutefois, un superutilisateur peut modifier le propriétaire de n'importe quelle fonction d'agrégat.

Paramètres

nom

Le nom (éventuellement qualifié du nom du schéma) de la fonction d'agrégat.

mode_arg

Le mode d'un argument : `IN` ou `VARIADIC`. La valeur par défaut est `IN`.

nom_arg

Le nom d'un argument. Notez que `ALTER AGGREGATE` ne fait pas réellement attention aux noms des arguments car seuls les types de données des arguments sont nécessaires pour déterminer l'identité de la fonction d'agrégat.

type_arg

Un type de données en entrée sur lequel la fonction d'agrégat opère. Pour référencer une fonction d'agrégat sans argument, écrivez `*` à la place de la liste des arguments. Pour référencer une fonction d'agrégat avec ensemble trié, ajoutez `ORDER BY` entre les spécifications des arguments direct et agrégé.

nouveau_nom

Le nouveau nom de la fonction d'agrégat.

nouveau_propriétaire

Le nouveau propriétaire de la fonction d'agrégat.

nouveau_schema

Le nouveau schéma de la fonction d'agrégat.

Notes

La syntaxe recommandée pour référencer un agrégat dont l'ensemble est trié revient à écrire `ORDER BY` entre les spécifications de l'argument direct et de l'argument agrégé, dans le même style que `CREATE AGGREGATE`. Néanmoins, cela fonctionnera aussi d'omettre `ORDER BY` en plaçant uniquement les spécifications de l'argument direct et de l'argument agrégé. Dans cette forme abréviée, si `VARIADIC "any"` a été utilisé à la fois dans l'argument direct et l'argument agrégé, écrire `VARIADIC "any"` seulement une fois.

Exemples

Renommer la fonction d'agrégat `mamoyenne` de type `integer` en `ma_moyenne` :

```
ALTER AGGREGATE mamoyenne(integer) RENAME TO ma_moyenne;
```

Changer le propriétaire de la fonction d'agrégat `mamoyenne` de type `integer` en `joe` :

```
ALTER AGGREGATE mamoyenne(integer) OWNER TO joe;
```

Pour déplacer l'agrégat `mon_pourcentage` dont l'argument direct est de type `float8` et l'argument agrégé de type `integer` dans le schéma `mon_schema` :

```
ALTER AGGREGATE mamoyenne(integer) SET SCHEMA mon_schema;
ALTER AGGREGATE mon_pourcentage(float8 ORDER BY integer) SET SCHEMA
 mon_schema;
```

Ceci fonctionne aussi :

```
ALTER AGGREGATE mon_pourcentage(float8, integer) SET SCHEMA
 mon_schema;
```

Compatibilité

Il n'y a pas de commande `ALTER AGGREGATE` dans le standard SQL.

Voir aussi

CREATE AGGREGATE, DROP AGGREGATE

ALTER COLLATION

ALTER COLLATION — modifie la définition d'une collation

Synopsis

```
ALTER COLLATION name REFRESH VERSION

ALTER COLLATION nom RENAME TO nouveau_nom
ALTER COLLATION nom OWNER TO { nouveau_propriétaire | CURRENT_ROLE
 | CURRENT_USER | SESSION_USER }
ALTER COLLATION nom SET SCHEMA nouveau_schéma
```

Description

`ALTER COLLATION` modifie la définition d'une collation.

Vous devez être propriétaire de la collation pour utiliser `ALTER COLLATION`. Pour en modifier le propriétaire, vous devez également être un membre direct ou indirect du nouveau rôle propriétaire, et ce rôle doit détenir le droit `CREATE` sur le schéma de la collation. (Ces restrictions garantissent que, en modifiant le propriétaire, vous ne pouvez rien faire qui ne soit impossible en supprimant et en recréant la collation. De toute manière, un superutilisateur peut modifier le propriétaire de n'importe quelle collation.)

Paramètres

nom

Le nom (éventuellement précédé par le schéma) d'une collation existante.

nouveau_nom

Le nouveau nom de la collation.

nouveau_propriétaire

Le nouveau propriétaire de la collation.

nouveau_schéma

Le nouveau schéma de la collation.

`REFRESH VERSION`

Met à jour la version de la collation. Voir Notes ci-dessous.

Notes

Quand un objet de collation est créé, la version spécifique du fournisseur de la collation est enregistrée dans le catalogue système. Quand la collation est utilisée, la version courante est comparée à la version enregistrée, et un avertissement est émis en cas d'incompatibilité, par exemple :

```
WARNING:  collation "xx-x-icu" has version mismatch
```

```
DETAIL:  The collation in the database was created using version
 1.2.3.4, but the operating system provides version 2.3.4.5.
HINT:  Rebuild all objects affected by this collation and run
 ALTER COLLATION pg_catalog."xx-x-icu" REFRESH VERSION, or build
 PostgreSQL with the right library version.
```

Un changement dans des définitions de collations peut entraîner des corruptions d'index et d'autres problèmes, car le moteur de la base de données s'appuie sur le fait que les objets stockées ont un certain ordre. En général, cela devrait être évité, mais cela peut néanmoins arriver dans certaines circonstances légitimes, telles que la mise à jour d'un système d'exploitation vers une nouvelle version majeure ou comme l'utilisation de `pg_upgrade` pour mettre à jour vers des binaires du serveur liés à une version plus récente d'ICU. Dans ces circonstances, tous les objets dépendants de cette collation doivent être reconstruits, par exemple avec `REINDEX`. Cette opération terminée, la version de la collation peut être rafraîchie avec la commande `ALTER COLLATION ... REFRESH VERSION`. Elle mettra à jour le catalogue système avec la version courante de la collation et l'avertissement ne s'affichera plus. Notez que la commande ne vérifie pas que tous les objets affectés ont été reconstruits correctement.

Avec les collations fournies par la `libc`, l'information sur la version est enregistrée sur les systèmes utilisant la bibliothèque C GNU (soit la plupart des systèmes sous Linux), sur FreeBSD, et Windows. Lors de l'utilisation de collations fournies par ICU, l'information de version est fournie par la bibliothèque ICU et est disponible sur toutes les plateformes.

> **Note**
>
> Si les collations utilisent la bibliothèque C GNU, la version de la bibliothèque C est utilisée comme proxy pour la version de la collation. Beaucoup de distributions Linux ne changent les définitions de collation que lors des mises à jour de la bibliothèque C. Cette approche pose problème, car les mainteneurs sont libres de rétroporter des définitions de collation récentes vers des versions anciennes de la bibliothèque C.
>
> Si les collations utilisent celles de Windows, l'information de version n'est disponible que pour celles définies avec des étiquettes d'identification de langue BCP 47, par exemple `en-US`.

Pour la collation par défaut de la base, il existe une commande analogue appelée `ALTER DATABASE ... REFRESH COLLATION VERSION`.

La requête suivante peut être utilisée pour identifier toutes les collations de la base de données courante qui nécessitent d'être rafraîchies, ainsi que tous les objets qui en dépendent :

```
SELECT pg_describe_object(refclassid, refobjid, refobjsubid) AS
 "Collation",
       pg_describe_object(classid, objid, objsubid) AS "Object"
  FROM pg_depend d JOIN pg_collation c
       ON refclassid = 'pg_collation'::regclass AND refobjid =
 c.oid
  WHERE c.collversion <> pg_collation_actual_version(c.oid)
  ORDER BY 1, 2;
```

Exemples

Pour renommer la collation `de_DE` en `german` :

```
ALTER COLLATION "de_DE" RENAME TO german;
```

Pour changer le propriétaire la propriété de la collation `en_US` par `joe` :

```
ALTER COLLATION "en_US" OWNER TO joe;
```

Compatibilité

Il n'y a pas de commande `ALTER COLLATION` dans le standard SQL.

Voir également

CREATE COLLATION, DROP COLLATION

ALTER CONVERSION

ALTER CONVERSION — Modifier la définition d'une conversion

Synopsis

```
ALTER CONVERSION nom RENAME TO nouveau_nom
ALTER CONVERSION nom OWNER TO { nouveau_propriétaire | CURRENT_ROLE
 | CURRENT_USER | SESSION_USER }
ALTER CONVERSION nom SET SCHEMA nouveau_schéma
```

Description

`ALTER CONVERSION` modifie la définition d'une conversion.

Seul le propriétaire de la conversion peut utiliser `ALTER CONVERSION`. Pour changer le propriétaire, il faut aussi être un membre direct ou indirect du nouveau rôle propriétaire et ce rôle doit avoir le droit `CREATE` sur le schéma de la conversion. Ces restrictions assurent que le changement de propriétaire ne va pas au-delà de ce qui peut être obtenu en supprimant et en re-créant la conversion. Toutefois, un superutilisateur peut changer le propriétaire de n'importe quelle conversion.

Paramètres

nom

Le nom de la conversion.

nouveau_nom

Le nouveau nom de la conversion.

nouveau_propriétaire

Le nouveau propriétaire de la conversion.

nouveau_schéma

Le nouveau schéma de la conversion.

Exemples

Renommer la conversion `iso_8859_1_to_utf8` en `latin1_to_unicode` :

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

Changer le propriétaire de la conversion `iso_8859_1_to_utf8` en `joe` :

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

Compatibilité

Il n'y a pas d'instruction `ALTER CONVERSION` dans le standard SQL.

Voir aussi

CREATE CONVERSION, DROP CONVERSION

ALTER DATABASE

ALTER DATABASE — Modifier une base de données

Synopsis

```
ALTER DATABASE nom [ [ WITH ] option [ ... ] ]

où option peut être :

    ALLOW_CONNECTIONS connexions_autorisées
    CONNECTION LIMIT limite_connexion
    IS_TEMPLATE est_modèle

ALTER DATABASE nom RENAME TO nouveau_nom
ALTER DATABASE nom OWNER TO { nouveau_propriétaire | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER }

ALTER DATABASE nom SET TABLESPACE nouveau_tablespace

ALTER DATABASE name REFRESH COLLATION VERSION

ALTER DATABASE nom SET paramètre { TO | = } { valeur | DEFAULT }
ALTER DATABASE nom SET paramètre FROM CURRENT
ALTER DATABASE nom RESET paramètre
ALTER DATABASE nom RESET ALL
```

Description

`ALTER DATABASE` modifie la définition d'une base de données.

La première forme modifie certains paramètres d'une base de données (voir ci-dessous pour les détails). Seul le propriétaire de la base de données ou un superutilisateur peut modifier ces paramètres.

La deuxième forme permet de renommer la base. Seul le propriétaire ou un superutilisateur peut renommer une base. Un propriétaire qui n'est pas superutilisateur doit en outre posséder le droit `CREATEDB`. La base en cours d'utilisation ne peut pas être renommée (on se connectera à une autre base pour réaliser cette opération).

La troisième forme change le propriétaire de la base de données. Pour changer le propriétaire, il faut être propriétaire de la base de données et membre direct ou indirect du nouveau rôle propriétaire. Le droit `CREATEDB` est également requis (les superutilisateurs ont automatiquement tous ces droits).

La quatrième forme change le tablespace par défaut de la base de données. Seuls le propriétaire de la base de données et un superutilisateur peuvent le faire. Vous devez aussi avoir le droit `CREATE` pour le nouveau tablespace. Cette commande déplace physiquement toutes les tables et tous les index actuellement dans l'ancien tablespace par défaut de la base de données vers le nouveau tablespace. Le nouveau tablespace par défaut doit être vide pour cette base de données, et personne ne peut être connecté à la base de données. Les tables et index placés dans d'autres tablespaces ne sont pas affectés.

Les formes restantes modifient la valeur par défaut d'un paramètre de configuration pour une base PostgreSQL. Par la suite, à chaque fois qu'une nouvelle session est lancée, la valeur spécifique devient la valeur par défaut de la session. Les valeurs par défaut de la base deviennent les valeurs par défaut de la session. En fait, elles surchargent tout paramètre présent dans `postgresql.conf` ou indiqué sur la ligne de commande de `postgres`. Seul le propriétaire de la base de données ou un superutilisateur

peut modifier les valeurs par défaut de la session pour une base. Certaines variables ne peuvent pas être configurées de cette façon pour une base de données ou peuvent seulement être configurées par un superutilisateur.

Paramètres

nom

Le nom de la base dont les attributs sont à modifier.

connexions_autorisées

Personne ne peut se connecter à cette base de données lorsque cette option est à false.

limite_connexion

Le nombre de connexions concurrentes sur la base de données. -1 signifie aucune limite.

est_modèle

Avec cette option à true, cette base peut être clonée par tout utilisateur ayant l'attribut CREATEDB. À false, seuls les superutilisateurs et le propriétaire de la base de données peuvent la cloner.

nouveau_nom

Le nouveau nom de la base.

nouveau_propriétaire

Le nouveau propriétaire de la base.

nouveau_tablespace

Le nouveau tablespace par défaut de la base de données.

Cette clause ne peut pas être exécutée dans un bloc de transaction.

REFRESH COLLATION VERSION

Met à jour la version de la collation de la base. Voir Notes pour l'historique.

paramètre
valeur

Configure cette valeur comme valeur par défaut de la base pour le paramètre de configuration précisée. Si *valeur* indique DEFAULT ou, de façon équivalente, si RESET est utilisé, la configuration en cours pour cette base est supprimée, donc la valeur système est utilisée pour les nouvelles sessions. Utiliser RESET ALL permet de supprimer tous les paramètres spécifiques de cette base. SET FROM CURRENT sauvegarde la valeur actuelle du paramètre en tant que valeur spécifique de la base.

Voir SET et Chapitre 20 pour plus d'informations sur les noms de paramètres et valeurs autorisées.

Notes

Il est possible de lier une valeur de session par défaut à un rôle plutôt qu'à une base. Voir ALTER ROLE à ce propos. En cas de conflit, les configurations spécifiques au rôle l'emportent sur celles spécifiques à la base.

Exemples

Désactiver les parcours d'index par défaut de la base `test` ::

```
ALTER DATABASE test SET enable_indexscan TO off;
```

Compatibilité

La commande `ALTER DATABASE` est une extension PostgreSQL.

Voir aussi

CREATE DATABASE, DROP DATABASE, SET, CREATE TABLESPACE

ALTER DEFAULT PRIVILEGES

ALTER DEFAULT PRIVILEGES — définit les droits d'accès par défaut

Synopsis

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } cible_rôle [, ...] ]
    [ IN SCHEMA nom_schéma [, ...] ]
    grant_ou_revoke_réduit

où grant_ou_revoke_réduit
peut être :

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES
 | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT
 OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT
 OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT
 OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT
 OPTION ]

GRANT { { USAGE | CREATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    TO { [ GROUP ] nom_rôle | PUBLIC } [, ...] [ WITH GRANT
 OPTION ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
 TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
```

```
    ON SEQUENCES
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | CREATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    FROM { [ GROUP ] nom_rôle | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

Description

`ALTER DEFAULT PRIVILEGES` vous permet de configurer les droits qui seront appliqués aux objets qui seront créés dans le futur. (Cela ne modifie pas les droits affectés à des objets déjà existants.) Actuellement, seuls les droits pour les schémas, les tables (ceci incluant les vues et les tables distantes), les séquences, les fonctions et les types (domaines inclus) peuvent être modifiés. Pour cette commande, les fonctions incluent les agrégats et les procédures. Les mots `FUNCTIONS` et `ROUTINES` sont équivalents sur cette commande. (`ROUTINES` est préféré à partir de maintenant en tant que terme standard pour fonctions et procédures.) Dans les versions antérieures de PostgreSQL, seul le mot `FUNCTIONS` était autorisé. Il n'est pas possible de configurer les droits par défaut séparément pour les fonctions et les procédures.

Vous pouvez modifier les droits par défaut seulement pour les objets qui seront créés par vous ou par des rôles dont vous êtes membres. Les droits peuvent être configurés de manière globale (c'est-à-dire pour tous les objets de la base de données) ou pour les objets des schémas indiqués.

Comme indiqué dans Section 5.7, les droits par défaut de tout type d'objet donnent tous les droits au propriétaire de l'objet et peut aussi donner certains droits à `PUBLIC`. Néanmoins, ce comportement peut être changé par une modification des droits par défaut globaux avec `ALTER DEFAULT PRIVILEGES`.

Les droits par défaut indiqués par schéma sont ajoutés aux droits par défaut globaux pour un type d'objet particulier. Ceci signifie que vous ne pouvez pas supprimer des droits par schéma s'ils sont donnés globalement (soit par défaut soit d'après une commande précédente `ALTER DEFAULT PRIVILEGES` qui n'indiquait pas de schéma). Le `REVOKE` par schéma est seulement utile pour inverser les effets d'un ancien `GRANT` par défaut sur un schéma.

Paramètres

`cible_rôle`

Le nom d'un rôle existant dont le rôle actuel est membre. Les droits d'accès par défaut ne sont pas hérités, donc les rôles membres doivent utiliser `SET ROLE` pour bénéficier de leurs droits ou `ALTER DEFAULT PRIVILEGES` doit être exécuter pour chaque rôle membre. Si `FOR ROLE` est omis, le rôle courant est supposé.

nom_schéma

Le nom d'un schéma existant. Si précisé, les droits par défaut sont modifiés pour les objets créés ultérieurement dans ce schéma. Si `IN SCHEMA` est omis, les droits globaux par défaut sont modifiés. `IN SCHEMA` n'est pas autorisé si `ON SCHEMAS` est utilisé puisque les schémas ne peuvent pas être imbriqués.

nom_rôle

Le nom d'un rôle existant pour donner ou reprendre les droits. Ce paramètre, et tous les autres paramètres dans *grant_ou_revoke_réduit*, agissent de la façon décrite dans GRANT ou REVOKE, sauf qu'il permet de configurer les droits pour une classe complète d'objets plutôt que pour des objets nommés spécifiques.

Notes

Utilisez la commande `\ddp` de psql pour obtenir des informations sur les droits par défaut. La signification des valeurs de droit est identique à celles utilisées par `\dp` et est expliquée dans Section 5.7.

Si vous souhaitez supprimer un rôle dont les droits par défaut ont été modifiés, il est nécessaire d'inverser les modifications dans ses droits par défaut ou d'utiliser `DROP OWNED BY` pour supprimer l'entrée des droits par défaut pour le rôle.

Exemples

Donner le droit SELECT à tout le monde pour toutes les tables (et vues) que vous pourriez créer plus tard dans le schéma `mon_schema`, et permettre au rôle `webuser` d'utiliser en plus INSERT :

```
ALTER DEFAULT PRIVILEGES IN SCHEMA mon_schema GRANT SELECT ON
 TABLES TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA mon_schema GRANT INSERT ON
 TABLES TO webuser;
```

Annuler ce qui a été fait ci-dessus, pour que les tables créées par la suite n'aient pas plus de droits qu'en standard :

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES
 FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES
 FROM webuser;
```

Supprimer le droit publique EXECUTE qui est normalement donné aux fonctions, pour toutes les fonctions créées après coup par le rôle `admin` :

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS
 FROM PUBLIC;
```

Notez néanmoins que vous ne pouvez *pas* obtenir cet effet avec une commande limitée à un seul schéma. Cette commande n'a pas d'effet sauf si elle annule un `GRANT` existant :

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public REVOKE EXECUTE ON
 FUNCTIONS FROM PUBLIC;
```

Ceci est dû au fait que les droits par défaut par schéma peuvent seulement ajouter des droits au paramétrage global, et non pas en supprimer.

Compatibilité

Il n'existe pas d'instruction `ALTER DEFAULT PRIVILEGES` dans le standard SQL.

Voir aussi

GRANT, REVOKE

ALTER DOMAIN

ALTER DOMAIN — Modifier la définition d'un domaine

Synopsis

```
ALTER DOMAIN nom
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN nom
    { SET | DROP } NOT NULL
ALTER DOMAIN nom
    ADD contrainte_de_domaine [ NOT VALID ]
ALTER DOMAIN nom
    DROP CONSTRAINT [ IF EXISTS ] nom_de_contrainte [ RESTRICT |
 CASCADE ]
ALTER DOMAIN nom
     RENAME CONSTRAINT nom_de_contrainte
 TO nouveau_nom_de_contrainte
ALTER DOMAIN nom
    VALIDATE CONSTRAINT nom_de_contrainte
ALTER DOMAIN nom
    OWNER TO { nouveau_propriétaire | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER DOMAIN nom
    RENAME TO nouveau_nom
ALTER DOMAIN nom
    SET SCHEMA nouveau_schéma

où contrainte_domaine
vaut :

[ CONSTRAINT nom_contrainte ]
{ NOT NULL | CHECK (expression) }
```

Description

`ALTER DOMAIN` modifie la définition d'un domaine. Il existe plusieurs sous-formes :

`SET/DROP DEFAULT`

Ces formes positionnent ou suppriment la valeur par défaut d'un domaine. Les valeurs par défaut ne s'appliquent qu'aux commandes `INSERT` ultérieures ; les colonnes d'une table qui utilisent déjà le domaine ne sont pas affectées.

`SET/DROP NOT NULL`

Ces formes agissent sur l'acceptation ou le rejet des valeurs NULL par un domaine. `SET NOT NULL` ne peut être utilisé que si les colonnes qui utilisent le domaine contiennent des valeurs non nulles.

`ADD contrainte_domaine [ NOT VALID ]`

Cette forme ajoute une nouvelle contrainte à un domaine. Lorsqu'une nouvelle contrainte est ajoutée à un domaine, toutes les colonnes utilisant ce domaine seront vérifiées avec cette nouvelle contrainte. Cette vérification initiale peut être annulée en ajoutant l'option `NOT VALID` lors de l'ajout de la nouvelle contrainte ; la contrainte pourra à nouveau être activée en utilisant la

commande `ALTER DOMAIN ... VALIDATE CONSTRAINT`. Les lignes nouvellement ajoutées ou modifiées sont toujours vérifiées pour l'ensemble des contraintes, y compris celles marquées `NOT VALID`. À noter enfin que l'option `NOT VALID` n'est acceptée que pour les contraintes de type `CHECK`.

`DROP CONSTRAINT [ IF EXISTS ]`

Cette forme supprime les contraintes sur un domaine. Si l'option `IF EXISTS` est spécifiée et que la contrainte n'existe pas, aucune erreur n'est retournée. Dans ce cas, un simple message d'avertissement est retourné.

`RENAME CONSTRAINT`

Cette forme modifie le nom de la contrainte d'un domaine.

`VALIDATE CONSTRAINT`

Cette forme valide une contrainte ajoutée précédemment avec l'option `NOT VALID`, c'est-à-dire qu'elle vérifie que les valeurs de chaque colonne utilisant ce type domaine satisfont la contrainte spécifiée.

`OWNER`

Cette forme change le propriétaire du domaine.

`RENAME`

Cette forme modifie le nom du domaine.

`SET SCHEMA`

Cette forme change le schéma du domaine. Toute contrainte associée au domaine est déplacée dans le nouveau schéma.

Seul le propriétaire du domaine peut utiliser `ALTER DOMAIN`. Pour modifier le schéma d'un domaine, le droit `CREATE` sur le nouveau schéma est également requis. Pour modifier le propriétaire, il faut être un membre direct ou indirect du nouveau rôle propriétaire et ce rôle doit avoir le droit `CREATE` sur le schéma du domaine. Ces restrictions assurent que la modification du propriétaire n'agissent pas au-delà de ce qui est réalisable en supprimant et en re-créant le domaine. Toutefois, un superutilisateur peut modifier le propriétaire de n'importe quel domaine.

Paramètres

nom

Le nom du domaine à modifier.

contrainte_de_domaine

Nouvelle contrainte de domaine pour le domaine.

nom_de_contrainte

Le nom d'une contrainte à supprimer ou renommer.

`NOT VALID`

Ne vérifie pas la validité de la contrainte appliquée aux valeurs des colonnes existantes.

`CASCADE`

Les objets qui dépendent de la contrainte sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

La contrainte n'est pas supprimée si des objets en dépendent. C'est le comportement par défaut.

`nouveau_nom`

Le nouveau nom du domaine.

`nouveau_nom_de_contrainte`

Le nouveau nom de la contrainte.

`nouveau_propriétaire`

Le nom de l'utilisateur nouveau propriétaire du domaine.

`nouveau_schéma`

Le nouveau schéma du domaine.

Notes

Bien que `ALTER DOMAIN ADD CONSTRAINT` essaie de vérifier que les données existantes satisfont la nouvelle contrainte, le test n'est pas absolu, parce que la commande ne peut pas « voir » les lignes de la table qui sont nouvellement insérées ou mises à jour et dont la transaction n'est pas encore validée. S'il y a un risque que des opérations concurrentes puissent insérer des données ne satisfaisant pas la contrainte, la manière de procéder consiste à ajouter la contrainte avec l'option `NOT VALID`, valider (commit) cette commande, attendre jusqu'à ce que toutes les transactions démarrées avant cette validation aient terminé, et alors exécuter `ALTER DOMAIN VALIDATE CONSTRAINT` pour chercher les données qui violeraient la contrainte. Cette méthode est fiable parce qu'une fois que la création de contrainte a fait l'objet d'un commit, il est garanti que les nouvelles transactions sont soumises à cette contrainte pour toute nouvelle valeur du type domaine.

Actuellement, `ALTER DOMAIN ADD CONSTRAINT`, `ALTER DOMAIN VALIDATE CONSTRAINT` et `ALTER DOMAIN SET NOT NULL` échoueront si le domaine nommé ou tout domaine dérivé est utilisé pour une colonne de type conteneur (type composé, tableau ou intervalle) dans toute table de la base de données. Il se pourrait que cela soit amélioré pour vérifier la nouvelle contrainte sur ce type de colonnes intégrées.

Exemples

Ajouter une contrainte `NOT NULL` à un domaine :

```
ALTER DOMAIN codezip SET NOT NULL;
```

Supprimer une contrainte `NOT NULL` d'un domaine :

```
ALTER DOMAIN codezip DROP NOT NULL;
```

Ajouter une contrainte de vérification à un domaine :

```
ALTER DOMAIN codezip ADD CONSTRAINT verif_zip CHECK
 (char_length(VALUE) = 5);
```

Supprimer une contrainte de vérification d'un domaine :

```
ALTER DOMAIN codezip DROP CONSTRAINT verif_zip;
```

Pour renommer une contrainte de vérification d'un domaine :

```
ALTER DOMAIN codezip RENAME CONSTRAINT verif_zip TO zip_verif;
```

Déplacer le domaine dans un schéma différent :

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

Compatibilité

`ALTER DOMAIN` se conforme au standard SQL, à l'exception des variantes `OWNER`, `RENAME`, `SET SCHEMA` et `VALIDATE CONSTRAINT`, qui sont des extensions PostgreSQL. L'option `NOT VALID` de la variante `ADD CONSTRAINT` est elle-aussi une extension de PostgreSQL.

Voir aussi

CREATE DOMAIN, DROP DOMAIN

ALTER EVENT TRIGGER

ALTER EVENT TRIGGER — modifier la définition d'un trigger sur un événement

Synopsis

```
ALTER EVENT TRIGGER nom DISABLE
ALTER EVENT TRIGGER nom ENABLE [ REPLICA | ALWAYS ]
ALTER EVENT TRIGGER nom OWNER TO { nouveau_propriétaire |
 CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER EVENT TRIGGER nom RENAME TO nouveau_nom
```

Description

`ALTER EVENT TRIGGER` modifie les propriétés d'un trigger sur événement existant.

Vous devez être superutilisateur pour modifier un trigger sur événement.

Paramètres

nom

Le nom d'un trigger existant à modifier.

nouveau_propriétaire

Le nom d'utilisateur du nouveau propriétaire du trigger sur événement.

nouveau_nom

Le nouveau nom du trigger sur événement.

`DISABLE`/`ENABLE [ REPLICA | ALWAYS ]`

Ces formes configurent le déclenchement des triggers sur événement. Un trigger désactivé est toujours connu du système mais il n'est pas exécuté si un événement intervient. Voir aussi session_replication_role.

Compatibilité

Il n'existe pas de commande `ALTER EVENT TRIGGER` dans le standard SQL.

Voir aussi

CREATE EVENT TRIGGER, DROP EVENT TRIGGER

ALTER EXTENSION

ALTER EXTENSION — modifie la définition d'une extension

Synopsis

```
ALTER EXTENSION nom UPDATE [ TO nouvelle_version ]
ALTER EXTENSION nom SET SCHEMA nouveau_schéma
ALTER EXTENSION nom ADD objet_membre
ALTER EXTENSION nom DROP objet_membre

où objet_membre peut être :

  ACCESS METHOD nom_objet |
  AGGREGATE nom_agrégat ( signature_agrégat ) |
  CAST (type_source AS type_cible) |
  COLLATION nom_objet |
  CONVERSION nom_objet |
  DOMAIN nom_objet |
  EVENT TRIGGER nom_objet |
  FOREIGN DATA WRAPPER nom_objet |
  FOREIGN TABLE nom_objet |
  FUNCTION nom_fonction [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ] |
  MATERIALIZED VIEW nom_objet |
  OPERATOR nom_opérateur (type_gauche, type_droit) |
  OPERATOR CLASS nom_objet USING méthode_indexage |
  OPERATOR FAMILY nom_objet USING méthode_indexage |
  [ PROCEDURAL ] LANGUAGE nom_objet |
  PROCEDURE nom_procédure [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ] |
  ROUTINE nom_routine [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ] |
  SCHEMA nom_objet |
  SEQUENCE nom_objet |
  SERVER nom_objet |
  TABLE nom_objet |
  TEXT SEARCH CONFIGURATION nom_objet |
  TEXT SEARCH DICTIONARY nom_objet |
  TEXT SEARCH PARSER nom_objet |
  TEXT SEARCH TEMPLATE nom_objet |
  TRANSFORM FOR nom_type LANGUAGE nom_langage |
  TYPE nom_objet |
  VIEW nom_objet

et signature_agrégat est :

* |
[ mode_arg ] [ nom_arg ] type_arg [ , ... ] |
[ [ mode_arg ] [ nom_arg ] type_arg [ , ... ] ] ORDER BY [ mode_arg
 ] [ nom_arg ] type_arg [ , ... ]
```

Description

`ALTER EXTENSION` modifie la définition d'une extension. Il existe plusieurs variantes :

`UPDATE`

Met à jour l'extension avec une nouvelle version. L'extension doit fournir le script de mise à jour adéquat (voire un ensemble de scripts) qui peut modifier la version en cours vers la version demandée.

`SET SCHEMA`

Déplace les objets de l'extension vers un autre schéma. L'extension doit permettre que ses objets soient déplacés pour que cette commande fonctionne.

`ADD` *`objet_membre`*

Ajoute un objet existant à l'extension. Cette commande est utilisée principalement dans les scripts de mise à jour d'extensions. L'objet concerné sera alors considéré comme appartenant à l'extension. Cela signifie principalement que l'objet ne pourra être supprimé qu'en supprimant l'extension.

`DROP` *`objet_membre`*

Supprime un objet de l'extension. Cette commande est utilisée principalement dans les scripts de mise à jour d'extensions. L'objet n'est pas supprimé : il n'appartient simplement plus à l'extension.

Voir aussi Section 38.17 pour des informations complémentaires sur les extensions.

Seul le propriétaire de l'extension peut utiliser la commande `ALTER EXTENSION` pour supprimer l'extension. Les options `ADD` ou `DROP` nécessitent en complément d'être le propriétaire de l'objet concerné par l'ajout ou la suppression.

Paramètres

`nom`

Le nom de l'extension concernée.

`nouvelle_version`

La nouvelle version de l'extension à installer. Il peut autant s'agir d'un identifiant que d'une chaîne de caractère. Si cette version n'est pas spécifiée, la commande `ALTER EXTENSION UPDATE` va utiliser tous les éléments de la version par défaut mentionnés dans le fichier de contrôle de l'extension.

`nouveau_schéma`

Le nouveau schéma vers lequel déplacer l'extension.

`nom_objet`
`nom_agregat`
`nom_fonction`
`nom_opérateur`
`nom_procédure`
`nom_routine`

Le nom d'un objet qui sera ajouté ou retiré de l'extension. Les noms de tables, agrégats, domaines, tables distantes, fonctions, opérateurs, classes d'opérateurs, familles d'opérateurs, procédures, routines, séquences, objets de recherche de texte, types et vues peuvent être qualifiés du nom du schéma.

type_source

Le nom d'un type de données source d'un transtypage.

type_cible

Le nom du type de donnée cible d'un transtypage.

mode_arg

Le mode du paramètre d'une fonction, d'une procédure ou d'un agrégat : `IN`, `OUT`, `INOUT` ou `VARIADIC`. La valeur par défaut est `IN`. Notez que la commande `ALTER EXTENSION` ne tient en réalité pas compte des paramètres dont le mode est `OUT` car les paramètres en entrée sont suffisants pour déterminer la signature de la fonction. Il est ainsi possible de ne spécifier que les paramètres de mode `IN`, `INOUT` et `VARIADIC`.

nom_arg

Le nom du paramètre de la fonction, de la procédure, ou de l'agrégat concerné. Notez que la commande `ALTER EXTENSION` ne tient pas compte en réalité des noms de paramètre, car les types de données sont suffisants pour déterminer la signature de la méthode.

type_arg

Le(s) type(s) de donnée des paramètres de la fonction, de la procédure ou de l'agrégat concerné (éventuellement qualifié du nom du schéma).

type_gauche
type_droit

Le type de données des arguments (éventuellement qualifié du nom du schéma) d'une fonction, d'une procédure ou d'un agrégat. Écrire `NONE` pour l'argument manquant d'un opérateur préfixé.

`PROCEDURAL`

Le mot clé `PROCEDURAL` n'est pas nécessaire. Il peut être omis.

nom_type

Le nom du type de données pour la transformation.

nom_language

Le nom du langage pour la transformation.

Exemples

Pour mettre à jour l'extension `hstore` à la version 2.0 :

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

Pour modifier le schéma de l'extension `hstore` par `utils` :

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

Pour ajouter une procédure stockée existante à l'extension `hstore` :

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement,
 hstore);
```

Compatibilité

`ALTER EXTENSION` est une extension de PostgreSQL.

Voir aussi

CREATE EXTENSION, DROP EXTENSION

ALTER FOREIGN DATA WRAPPER

ALTER FOREIGN DATA WRAPPER — modifier la définition d'un wrapper de données distantes

Synopsis

```
ALTER FOREIGN DATA WRAPPER nom
    [ HANDLER fonction_handler | NO HANDLER ]
    [ VALIDATOR fonction_validation | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ]) ]
ALTER FOREIGN DATA WRAPPER nom OWNER TO { nouveau_propriétaire |
 CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER FOREIGN DATA WRAPPER nom RENAME TO nouveau_nom
```

Description

`ALTER FOREIGN DATA WRAPPER` modifie la définition d'un wrapper de données distantes. La première forme de la commande modifie les fonctions de support ou les options génériques du wrapper de données distantes (au moins une clause est nécessaire). La seconde forme modifie le propriétaire du wrapper de données distantes.

Seuls les superutilisateurs peuvent modifier les wrappers de données distantes. De plus, seuls les superutilisateurs peuvent être propriétaires de wrappers de données distantes.

Paramètres

nom

Le nom d'un wrapper de données distantes existant.

`HANDLER` *fonction_handler*

Spécifie une nouvelle fonction de gestion pour le wrapper de données distantes.

`NO HANDLER`

Cette clause est utilisée pour spécifier que le wrapper de données distantes ne doit plus avoir de fonction de gestion.

Notez que les tables distantes qui utilisent un wrapper de données distantes, sans fonction de gestion, ne peuvent pas être utilisées.

`VALIDATOR` *fonction_validation*

Indique une fonction de validation pour le wrapper de données distantes.

Notez qu'il est possible que des options pré-existantes du wrapper de données distantes, ou de ses serveurs, correspondances d'utilisateurs ou tables distantes, soient invalides d'après le nouveau validateur. PostgreSQL ne vérifie pas ça. C'est à l'utilisateur de s'assurer que ces options sont correctes avant d'utiliser le wrapper de données distantes modifié. Néanmoins, toute option précisée dans cette commande `ALTER FOREIGN DATA WRAPPER` sera vérifiée en utilisant le nouveau validateur.

`NO VALIDATOR`

Cette option est utilisée pour spécifier que le wrapper de données distantes n'aura plus de fonction de validation.

`OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ] )`

Modifie les options du wrapper de données distantes. `ADD`, `SET` et `DROP` spécifient l'action à réaliser. `ADD` est pris par défaut si aucune opération n'est explicitement spécifiée. Les noms des options doivent être uniques ; les noms et valeurs sont validés en utilisant la fonction de validation du wrapper de données distantes.

nouveau_propriétaire

Le nom du nouveau propriétaire du wrapper de données distantes.

nouveau_nom

Le nouveau nom du wrapper de données distantes.

Exemples

Modifier le wrapper de données distantes `dbi`, ajouter l'option `foo`, supprimer `bar` :

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP bar);
```

Modifier la fonction de validation du wrapper de données distantes `dbi` en `bob.myvalidator` :

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

Compatibilité

`ALTER FOREIGN DATA WRAPPER` se conforme à ISO/IEC 9075-9 (SQL/MED). Néanmoins, les clauses `HANDLER`, `VALIDATOR`, `OWNER TO` et `RENAME` sont des extensions.

Voir aussi

CREATE FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER

ALTER FOREIGN TABLE

ALTER FOREIGN TABLE — modifie la définition de la table distante

Synopsis

```
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] nom [ * ]
    action [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] nom [ * ]
    RENAME [ COLUMN ] nom_colonne TO nouveau_nom_colonne
ALTER FOREIGN TABLE [ IF EXISTS ] nom
    RENAME TO nouveau_nom
ALTER FOREIGN TABLE [ IF EXISTS ] nom
    SET SCHEMA nouveau_schéma

où action peut être :

    ADD [ COLUMN ] [ IF NOT EXISTS ] nom_colonne type_données
 [ COLLATE collation ] [ contrainte_colonne [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] nom_colonne [ RESTRICT |
 CASCADE ]
    ALTER [ COLUMN ] nom_colonne [ SET DATA ] TYPE type_données
 [ COLLATE collation ]
    ALTER [ COLUMN ] nom_colonne SET DEFAULT expression
    ALTER [ COLUMN ] nom_colonne DROP DEFAULT
    ALTER [ COLUMN ] nom_colonne { SET | DROP } NOT NULL
    ALTER [ COLUMN ] nom_colonne SET STATISTICS integer
    ALTER [ COLUMN ] nom_colonne SET ( option_attribut = valeur
 [, ... ] )
    ALTER [ COLUMN ] nom_colonne RESET ( option_attribut [, ... ] )
    ALTER [ COLUMN ] nom_colonne SET STORAGE { PLAIN | EXTERNAL |
 EXTENDED | MAIN }
    ALTER [ COLUMN ] nom_colonne OPTIONS ( [ ADD | SET |
 DROP ] option ['valeur'] [, ... ])
    ADD contrainte_table [ NOT VALID ]
    VALIDATE CONSTRAINT nom_contrainte
    DROP CONSTRAINT [ IF EXISTS ]  nom_contrainte [ RESTRICT |
 CASCADE ]
    DISABLE TRIGGER [ nom_trigger | ALL | USER ]
    ENABLE TRIGGER [ nom_trigger | ALL | USER ]
    ENABLE REPLICA TRIGGER nom_trigger
    ENABLE ALWAYS TRIGGER nom_trigger
    SET WITHOUT OIDS
    INHERIT table_parent
    NO INHERIT table_parent
    OWNER TO { nouveau_propriétaire | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
    OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ])
```

Description

`ALTER FOREIGN TABLE` modifie la définition d'une table distante existante. Il existe plusieurs variantes :

`ADD [ COLUMN ] [ IF EXISTS ]`

Ajoute une nouvelle colonne à la table distante en utilisant une syntaxe identique à celle de `CREATE FOREIGN TABLE`. Si `IF NOT EXISTS` est précisé et qu'une colonne existe déjà avec ce nom, aucune erreur n'est renvoyée. Contrairement au comportement de l'ajout d'une colonne à une table, rien ne se passe au niveau stockage : cette action déclare simplement qu'une nouvelle colonne est accessible via la table distante.

`DROP [ COLUMN ] [ IF EXISTS ]`

Supprime une colonne de la table. L'option `CASCADE` doit être utilisée lorsque des objets en dehors de la table dépendent de cette colonne, comme par exemple des références de clés étrangères ou des vues. Si `IF EXISTS` est indiqué et que la colonne n'existe pas, aucune erreur n'est renvoyée. Dans ce cas, un message d'avertissement est envoyé à la place.

`SET DATA TYPE`

Change le type d'une colonne de la table. Là-aussi, cela n'a aucun effet sur le stockage sous-jacent : cette action change simplement le type de la colonne, d'après PostgreSQL.

`SET/DROP DEFAULT`

Ces clauses ajoutent ou suppriment une valeur par défaut pour une colonne. Les valeurs par défaut s'appliquent seulement pour les prochaines commandes `INSERT` et `UPDATE` ; elles ne changent rien aux lignes déjà présentes dans la table.

`SET/DROP NOT NULL`

Autorise / refuse l'ajout de valeurs NULL dans la colonne. `SET NOT NULL` ne peut être utilisé que si la colonne ne contient pas de valeurs NULL.

`SET STATISTICS`

Cette clause définit pour chaque colonne l'objectif de collecte de statistiques pour les opérations d'`ANALYZE` ultérieures. Voir les clauses correspondantes de l'instruction `ALTER TABLE` pour plus de détails.

`SET ( option_attribut = valeur [, ... ] )`
`RESET ( option_attribut [, ... ] )`

Cette clause définit ou met à zéro des options propres à une colonne. Voir les clauses correspondantes de l'instruction `ALTER TABLE` pour plus de détails.

`SET STORAGE`

Cette clause configure le mode de stockage pour une colonne. Voir la clause similaire de `ALTER TABLE` pour plus de détails. Notez que le mode de stockage n'a d'effet que si le wrapper de données distantes choisit de le prendre en compte.

`ADD contrainte_table [ NOT VALID ]`

Cette clause ajoute une nouvelle contrainte à une table distante, en utilisant la même syntaxe que `CREATE FOREIGN TABLE`. Seules les contraintes `CHECK` sont actuellement supportées.

Contrairement à l'ajout d'une contrainte sur une table standard, rien n'est tenté pour vérifier que la contrainte est vraie ; en fait, cette action déclare uniquement certaines conditions qui seront supposées vraies pour toutes les lignes de la table distante. (Voir la discussion dans `CREATE FOREIGN TABLE`.) Si la contrainte est marquée `NOT VALID`, alors elle n'est pas considérée vraie mais est enregistrée pour une utilisation future.

`VALIDATE CONSTRAINT`

Cette clause marque la validité d'une contrainte précédemment invalide (`NOT VALID`). Aucune vérification n'est effectuée pour s'assurer de la véracité de cette indication. Les prochaines requêtes supposeront que les données respectent cette contrainte.

`DROP CONSTRAINT [ IF EXISTS ]`

Cette clause supprime la contrainte indiquée sur la table distante. Si la clause `IF EXISTS` est précisée et que la contrainte n'existe pas, aucune erreur n'est renvoyée mais un message d'information apparait.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER`

Ces syntaxes configurent le déclenchement des triggers positionnés sur la table distante. Voir la syntaxe similaire de `ALTER TABLE` pour plus de détails.

`SET WITHOUT OIDS`

Procure une compatibilité de syntaxe dans le cadre de la suppression de la colonne système `oid`. Comme les colonnes système `oid` ne peuvent plus être ajoutées, cela n'a aucun effet.

`INHERIT` *`parent_table`*

Cette forme ajoute la table distante cible comme nouvel enfant de la table parente spécifiée. Pour plus de détails, voir la forme similaire dans `ALTER TABLE`.

`NO INHERIT` *`parent_table`*

Cette forme supprime la table cible distante de la liste des enfants de la table parente spécifiée.

`OWNER`

Change le propriétaire d'une table distante. Le nouveau propriétaire est celui passé en paramètre.

`OPTIONS ( [ ADD | SET | DROP ]` *`option`* `['`*`value`*`'] [, ... ] )`

Modifie les options de la table distante et de ses colonnes. L'action à effectuer est spécifiée par `ADD` (ajout), `SET` (définition) ou `DROP` (suppression). Si aucune action n'est mentionnée, `ADD` est utilisée.

Les noms des options autorisées et leurs valeurs sont spécifiques à chaque wrapper de données distantes. L'utilisation répétée de la même option n'est pas autorisée (bien qu'il soit possible qu'une option de table et de colonne aie le même nom). Les noms d'option et leur valeur sont en outre validées par la bibliothèque du wrapper de données distantes.

`RENAME`

Change le nom d'une table distante ou le nom d'une colonne individuelle de la table distante. Cela n'a aucun effet sur la donnée stockée.

`SET SCHEMA`

Déplace la table distante dans un autre schéma.

À l'exception de `RENAME` et `SET SCHEMA`, toutes les actions peuvent être combinées en une liste de modifications appliquées parallèlement. Par exemple, il est possible d'ajouter plusieurs colonnes et/ou de modifier plusieurs colonnes en une seule commande.

Si la commande est écrite avec `ALTER FOREIGN TABLE IF EXISTS ...` et que la table distante n'existe pas, aucune erreur n'est émise. Un message d'avertissement est émis dans ce cas.

Il faut être propriétaire de la table pour utiliser `ALTER FOREIGN TABLE`. Pour modifier le schéma d'une table, le droit `CREATE` sur le nouveau schéma est requis. Pour modifier le propriétaire de la table, il est nécessaire d'être un membre direct ou indirect du nouveau rôle et ce dernier doit avoir le droit `CREATE` sur le schéma de la table (ces restrictions assurent que la modification du propriétaire ne diffère en rien de ce qu'il est possible de faire par la suppression et la re-création de la table. Néanmoins, dans tous les cas, un superutilisateur peut modifier le propriétaire de n'importe quelle table). Pour ajouter une colonne ou modifier un type de colonne, vous devez aussi détenir le droit `USAGE` sur le type de donnée.

Paramètres

nom

Le nom (éventuellement qualifié du nom du schéma) de la table à modifier. Si `ONLY` est spécifié avant le nom de la table, la table seule est modifiée. Si `ONLY` n'est pas spécifié, la table et toutes ses tables filles (s'il y en a) sont modifiées. En option, `*` peut être spécifié après le nom de la table pour indiquer explicitement que les tables filles sont incluses.

nom_colonne

Le nom d'une colonne, existante ou nouvelle.

nouveau_nom_colonne

Le nouveau nom d'une colonne existante.

nouveau_nom

Le nouveau nom de la table.

type_données

Le type de données de la nouvelle colonne, ou le nouveau type de données d'une colonne existante.

table_constraint

Nouvelle contrainte au niveau table pour la table distante.

constraint_name

Nom d'une contrainte existante à supprimer.

`CASCADE`

Les objets qui dépendent de la colonne ou de la contrainte supprimée sont automatiquement supprimés (par exemple, les vues référençant la colonne), ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

La colonne ou la contrainte n'est pas supprimée si des objets en dépendent. C'est le comportement par défaut.

nom_trigger

Nom du trigger à activer ou désactiver.

`ALL`

Désactive ou active tous les triggers appartenant à la table distante. (Ceci requiert l'attribut superutilisateur si un des triggers est un trigger interne. Le moteur n'ajoute pas de tels triggers sur les tables distantes mais du code externe pourrait le faire.)

`USER`

Désactive ou active tous les triggers appartenant à la table distante, sauf pour les triggers internes.

`parent_table`

Une table parente à associer ou dissocier de cette table distante.

`nouveau_propriétaire`

Le nom d'utilisateur du nouveau propriétaire de la table distante.

`nouveau_schéma`

Le nom du schéma vers lequel la table distante sera déplacée.

Notes

Le mot clé `COLUMN` n'est pas nécessaire. Il peut être omis.

La cohérence avec le serveur distant n'est pas vérifiée lorsqu'une colonne est ajoutée ou supprimée avec la commande `ADD COLUMN` ou `DROP COLUMN`, lorsqu'une contrainte `CHECK` ou `NOT NULL` est ajoutée, ou encore lorsqu'un type de colonne est modifié avec l'action `SET DATA TYPE`. Il est ainsi de la responsabilité de l'utilisateur de s'assurer que la définition de la table distante est compatible avec celle du serveur distant.

Voir la commande `CREATE FOREIGN TABLE` pour une description plus complète des paramètres valides.

Exemples

Pour interdire les valeurs NULL sur une colonne :

```
ALTER FOREIGN TABLE distributeurs ALTER COLUMN rue SET NOT NULL;
```

Pour modifier les options d'une table distante :

```
ALTER FOREIGN TABLE mon_schema.distributeurs OPTIONS (ADD opt1
 'valeur', SET opt2 'valeur2', DROP opt3);
```

Compatibilité

Les actions `ADD`, `DROP`, et `SET DATA TYPE` sont conformes au standard SQL. Les autres actions sont des extensions PostgreSQL du standard SQL. De plus, la possibilité de combiner de multiples modifications en une seule commande `ALTER FOREIGN TABLE` est une extension PostgreSQL.

La commande `ALTER FOREIGN TABLE DROP COLUMN` peut être utilisée pour supprimer jusqu'à la dernière colonne d'une table distante, permettant ainsi d'obtenir une table sans colonne. Il s'agit d'une extension du standard SQL, qui ne permet pas de gérer des tables sans colonnes.

Voir aussi

CREATE FOREIGN TABLE, DROP FOREIGN TABLE

ALTER FUNCTION

ALTER FUNCTION — Modifier la définition d'une fonction

Synopsis

```
ALTER FUNCTION nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER FUNCTION nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    REnom TO nouveau_nom
ALTER FUNCTION nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    OWNER TO { nouveau_propriétaire | CURRENT_ROLE | CURRENT_USER
 | SESSION_USER }
ALTER FUNCTION nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    SET SCHEMA nouveau_schéma
ALTER FUNCTION nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    [ NO ] DEPENDS ON EXTENSION nom_extension

où action peut être :

    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE
    [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST cout_exécution
    ROWS nb_lignes_résultat
    SUPPORT support_function
    SET paramètre { TO | = } { valeur | DEFAULT }
    SET paramètre FROM CURRENT
    RESET paramètre
    RESET ALL
```

Description

`ALTER FUNCTION` modifie la définition d'une fonction.

Seul le propriétaire de la fonction peut utiliser `ALTER FUNCTION`. Le droit `CREATE` sur le nouveau schéma est requis pour pouvoir changer le schéma de la fonction. Pour modifier le propriétaire, il est nécessaire d'être membre direct ou indirect du nouveau rôle propriétaire. Ce dernier doit posséder le droit `CREATE` sur le schéma de la fonction. Ces restrictions assurent que la modification du propriétaire n'a pas d'effets autres que ceux obtenus par la suppression et la re-création de la fonction ; toutefois, un superutilisateur peut modifier le propriétaire de n'importe quelle fonction.

Paramètres

nom

Le nom de la fonction (potentiellement qualifié du nom du schéma). Si aucune liste d'argument n'est spécifiée, le nom doit être unique dans son schéma.

modearg

Le mode d'un argument : `IN`, `OUT`, `INOUT` ou `VARIADIC`. En cas d'omission, la valeur par défaut est `IN`. `ALTER FUNCTION` ne tient pas compte des arguments `OUT`, car seuls les arguments en entrée sont nécessaire pour déterminer l'identité de la fonction. Les arguments `IN`, `INOUT` et `VARIADIC` sont donc suffisants.

nomarg

Le nom d'un argument. `ALTER FUNCTION` ne tient pas compte des noms des arguments, car seuls les types de données des arguments sont nécessaires pour déterminer l'identité d'une fonction.

typearg

Le type de données d'un argument de la fonction (éventuellement qualifié du nom du schéma).

nouveau_nom

Le nouveau nom de la fonction.

nouveau_propriétaire

Le nouveau propriétaire de la fonction. Si cette fonction est marquée `SECURITY DEFINER`, elle s'exécute par la suite sous cette identité.

nouveau_schéma

Le nouveau schéma de la fonction.

`DEPENDS ON EXTENSION` *nom_extension*
`NO DEPENDS ON EXTENSION` *nom_extension*

Cette clause marque la fonction comme dépendante de l'extension ou comme indépendante de l'extension si le mot-clé `NO` est ajouté. Une fonction dépendante d'une extension est supprimée quand l'extension est supprimée, même si la clause `CASCADE` n'est pas indiquée. Une fonction peut dépendre de plusieurs extensions, et est supprimé quand une de ces extensions est supprimée.

`CALLED ON NULL INPUT`
`RETURNS NULL ON NULL INPUT`
`STRICT`

`CALLED ON NULL INPUT` modifie la fonction pour qu'elle puisse être appelée avec des arguments NULL. `RETURNS NULL ON NULL INPUT` et `STRICT` modifie la fonction pour qu'elle ne soit pas appelée si un des arguments est NULL ; un résultat NULL est alors automatiquement déterminé. Voir CREATE FUNCTION pour plus d'informations.

`IMMUTABLE`
`STABLE`
`VOLATILE`

Modifie la volatilité de la fonction. Voir CREATE FUNCTION pour plus d'informations.

```
[ EXTERNAL ] SECURITY INVOKER
[ EXTERNAL ] SECURITY DEFINER
```

Précise si la fonction doit être appelée avec les droits de l'utilisateur qui l'a créée. Le mot clé `EXTERNAL`, ignoré, existe pour des raisons de compatibilité SQL. Voir CREATE FUNCTION pour plus d'informations.

`PARALLEL`

Indique si la fonction peut être exécutée en parallèle. Voir CREATE FUNCTION pour les détails.

`LEAKPROOF`

Indique si la fonction doit être considérée comme étant étanche (*leakproof*). Voir CREATE FUNCTION pour plus d'informations.

`COST` *`cout_execution`*

Modifie l'estimation du coût d'exécution de la fonction. Voir CREATE FUNCTION pour plus d'informations.

`ROWS` *`nb_lignes_resultat`*

Modifie l'estimation du nombre de lignes renvoyées par une fonction SRF. Voir CREATE FUNCTION pour plus d'informations.

`SUPPORT` *`support_function`*

Fournit une fonction de support de planification utilisable pour cette fonction. Voir Section 38.11 pour plus d'informations. Il faut être superutilisateur pour utiliser cette option.

Cette option ne peut pas être utilisée pour supprimer la fonction de support, parce qu'elle doit indiquer une nouvelle fonction de support. Si vous avez besoin de faire cela, utilisez `CREATE OR REPLACE FUNCTION`.

`paramètre`
`valeur`

Ajoute ou modifie l'initialisation d'un paramètre de configuration lorsque la fonction est appelée. Si *`valeur`* est `DEFAULT` ou, de façon équivalente, si `RESET` est utilisé, le paramètre local de la fonction est supprimée pour que la fonction s'exécute avec la valeur par défaut du paramètre. Utiliser `RESET ALL` supprime tous les valeurs spécifiques des paramètres pour cette fonction. `SET FROM CURRENT` sauvegarde la valeur actuelle du paramètre quand `ALTER FUNCTION` est exécuté comme valeur à appliquer lors de l'exécution de la fonction.

Voir SET et Chapitre 20 pour plus d'informations sur les noms des paramètres et les valeurs autorisés.

`RESTRICT`

Ignoré, présent pour des raisons de conformité avec le standard SQL.

Exemples

Renommer la fonction `sqrt` pour le type `integer` en `square_root` :

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

Changer le propriétaire de la fonction `sqrt` pour le type `integer` en `joe` :

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

Modifier le schéma de la fonction `sqrt` du type `integer` par `maths` :

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

Pour marquer la fonction `sqrt` du type `integer` comme dépendant de l'extension `mathlib` :

```
ALTER FUNCTION sqrt(integer) DEPENDS ON EXTENSION mathlib;
```

Pour ajuster automatiquement le chemin de recherche des schémas pour une fonction :

```
ALTER FUNCTION verifie_motdepasse(text) SET search_path = admin,
 pg_temp;
```

Pour désactiver le paramètre `search_path` d'une fonction :

```
ALTER FUNCTION verifie_motdepasse(text) RESET search_path;
```

La fonction s'exécutera maintenant avec la valeur de la session pour cette variable.

Compatibilité

La compatibilité de cette instruction avec l'instruction `ALTER FUNCTION` du standard SQL est partielle. Le standard autorise la modification d'un plus grand nombre de propriétés d'une fonction mais ne laisse pas la possibilité de renommer une fonction, de placer le commutateur `SECURITY DEFINER` sur la fonction, d'y attacher des valeurs de paramètres ou d'en modifier le propriétaire, le schéma ou la volatilité. Le standard requiert le mot clé `RESTRICT` ; il est optionnel avec PostgreSQL.

Voir aussi

CREATE FUNCTION, DROP FUNCTION

ALTER GROUP

ALTER GROUP — Modifier le nom d'un rôle ou la liste de ses membres

Synopsis

```
ALTER GROUP spécification_role ADD USER nom_utilisateur [, ... ]
ALTER GROUP spécification_role DROP USER nom_utilisateur [, ... ]

où spécification_role peut
valoir :

    nom_rôle
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER

ALTER GROUP nom_groupe RENAME TO nouveau_nom
```

Description

`ALTER GROUP` modifie les attributs d'un groupe d'utilisateurs Cette commande est obsolète, mais toujours acceptée pour des raisons de compatibilité ascendante. Les groupes (et les utilisateurs) ont été remplacés par le concept plus général de rôles.

Les deux premières formes ajoutent des utilisateurs à un groupe ou en suppriment. Tout rôle peut être ici « utilisateur » ou « groupe ». Ces variantes sont réellement équivalentes à la promotion ou la révocation de l'appartenance au rôle nommé « groupe » ; il est donc préférable d'utiliser `GRANT` et `REVOKE` pour le faire.

La troisième forme change le nom du groupe. Elle est strictement équivalente au renommage du rôle par `ALTER ROLE`.

Paramètres

nom_groupe

Le nom du groupe (rôle) à modifier.

nom_utilisateur

Les utilisateurs (rôles) à ajouter au groupe ou à en enlever. Les utilisateurs doivent préalablement exister ; `ALTER GROUP` ne crée pas et ne détruit pas d'utilisateur.

nouveau_nom

Le nouveau nom du groupe.

Exemples

Ajouter des utilisateurs à un groupe :

```
ALTER GROUP staff ADD USER karl, john;
```

Supprimer des utilisateurs d'un groupe :

```
ALTER GROUP workers DROP USER beth;
```

Compatibilité

Il n'existe pas de relation `ALTER GROUP` en SQL standard.

Voir aussi

GRANT, REVOKE, ALTER ROLE

ALTER INDEX

ALTER INDEX — Modifier la définition d'un index

Synopsis

```
ALTER INDEX [ IF EXISTS ] nom RENAME TO nouveau_nom
ALTER INDEX [ IF EXISTS ] nom SET TABLESPACE nom_tablespace
ALTER INDEX nom ATTACH PARTITION nom_index
ALTER INDEX nom [ NO ] DEPENDS ON EXTENSION nom_extension
ALTER INDEX [ IF EXISTS ] nom SET ( paramètre_stockage [= valeur]
 [, ... ] )
ALTER INDEX [ IF EXISTS ] nom RESET ( paramètre_stockage [, ... ] )
ALTER INDEX [ IF EXISTS ] nom ALTER [ COLUMN ] numéro_colonne
    SET STATISTICS integer
ALTER INDEX ALL IN TABLESPACE nom [ OWNED BY nom_rôle [, ... ] ]
    SET TABLESPACE nouveau_tablespace [ NOWAIT ]
```

Description

`ALTER INDEX` modifie la définition d'un index. Il y a plusieurs variantes décrites ci-dessous. Notez que le niveau de verrouillage requis peut différer entre chaque. Un verrou de niveau `ACCESS EXCLUSIVE` est pris, sauf exception dûment précisée. Lorsque plusieurs sous-commandes sont listées, le verrou pris sera le plus strict de l'ensemble des sous-commandes.

`RENAME`

La forme `RENAME` modifie le nom de l'index. Si l'index est associé avec une contrainte de table (soit `UNIQUE`, soit `PRIMARY KEY`, soit `EXCLUDE`), la contrainte est elle-aussi renommée. Cela n'a aucun effet sur les données stockées.

Le renommage d'index acquiert un verrou de niveau `SHARE UPDATE EXCLUSIVE`.

`SET TABLESPACE`

Cette forme remplace le tablespace de l'index par le tablespace spécifié et déplace les fichiers de données associés à l'index dans le nouveau tablespace. Pour modifier le tablespace d'un index, vous devez être le propriétaire de l'index et avoir le droit `CREATE` sur le nouveau tablespace. Toutes les index d'un tablespace de la base de données actuelle peuvent être déplacés en utilisant la forme `ALL IN TABLESPACE`, qui verrouillera tous les index à déplacer, puis les déplacera un par un. Cette forme supporte aussi la clause `OWNED BY`, qui ne déplacera que les index dont les propriétaires sont indiqués. Si l'option `NOWAIT` est spécifié, alors la commande échouera si elle est incapable de récupérer immédiatement tous les verrous requis. Notez que les catalogues systèmes ne seront pas déplacés par cette commande. Dans ce cas, il faut utiliser `ALTER DATABASE` ou `ALTER INDEX`. Voir aussi `CREATE TABLESPACE`.

`ATTACH PARTITION` *`nom_index`*

Attache l'index nommé (potentiellement qualifié du schéma) à l'index modifié. L'index nommé doit être sur une partition de la table contenant l'index à modifier et avoir une définition équivalente. Un index attaché ne peut pas être lui-même supprimé. Il sera automatiquement supprimé si son index parent est supprimé.

Si l'index nommé est déjà attaché à l'index modifié, la commande tentera de valider l'index parent si le parent n'est pas actuellement valide.

`DEPENDS ON EXTENSION` *`nom_extension`*
`NO DEPENDS ON EXTENSION` *`nom_extension`*

Cette clause marque l'index comme dépendant de l'extension ou comme indépendant de l'extension si le mot-clé `NO` est ajouté. Un index dépendant d'une extension est automatiquement supprimé quand l'extension est supprimée.

`SET (` *`paramètre_stockage`* `[=` *`valeur`*`] [, ... ] )`

Cette forme modifie un ou plusieurs paramètres spécifiques à la méthode d'indexage de cet index. Voir `CREATE INDEX` pour les détails sur les paramètres disponibles. Notez que le contenu de l'index ne sera pas immédiatement modifié par cette commande ; suivant le paramètre, vous pourrez avoir besoin de reconstruire l'index avec `REINDEX` pour obtenir l'effet désiré.

`ALTER [ COLUMN ]` *`numéro_colonne`* `SET STATISTICS` *`integer`*

Cette syntaxe configure la cible de récupération des statistiques par colonne pour les opérations `ANALYZE` qui suivront, mais peut être utilisé seulement sur les colonnes d'index définies sous la forme d'une expression. Comme les expressions n'ont pas de nom unique, nous faisons référence à elles en utilisant le numéro ordinal de la colonne d'index. La cible peut être configurée sur l'intervalle 0 à 10000. Une configuration à -1 annule l'ancienne configuration pour revenir à l'utilisation de la cible statistique par défaut du système (default_statistics_target). Pour plus d'informations sur l'utilisation de statistiques par l'optimiseur de requêtes de PostgreSQL, référez-vous à Section 14.2.

`SET (` *`paramètre_stockage`* `[=` *`valeur`*`] [, ... ] )`

Cette forme modifie un ou plusieurs paramètres spécifiques à la méthode d'indexage de cet index. Voir CREATE INDEX pour les détails sur les paramètres disponibles. Notez que le contenu de l'index ne sera pas immédiatement modifié par cette commande ; suivant le paramètre, vous pouvez avoir besoin de reconstruire l'index avec REINDEX pour obtenir l'effet désiré.

`RESET (` *`paramètre_stockage`* `[, ... ] )`

Cette forme réinitialise un ou plusieurs paramètres de stockage spécifiques à la méthode d'indexage à leurs valeurs par défaut. Comme avec `SET`, un `REINDEX` peut être nécessaire pour mettre à jour l'index complètement.

Paramètres

`IF EXISTS`

Ne retourne par d'erreur si l'index n'existe pas. Seul un message d'avertissement est retourné dans ce cas.

`numéro_colonne`

Ce numéro fait référence à la position ordinale (de gauche à droite) de la colonne de l'index.

`nom`

Le nom de l'index à modifier (éventuellement qualifié du nom du schéma).

`nouveau_nom`

Le nouveau nom de l'index.

`nom_tablespace`

Le nom du tablespace dans lequel déplacer l'index.

nom_extension

Le nom de l'extension dont l'index dépend.

paramètre_stockage

Le nom du paramètre de stockage spécifique à la méthode d'indexage.

valeur

La nouvelle valeur du paramètre de stockage spécifique à la méthode d'indexage. Cette valeur peut être un nombre ou une chaîne suivant le paramètre.

Notes

Ces opérations sont aussi possibles en utilisant `ALTER TABLE`. `ALTER INDEX` n'est en fait qu'un alias pour les formes d'`ALTER TABLE` qui s'appliquent aux index.

Auparavant, il existait une variante `ALTER INDEX OWNER` mais elle est maintenant ignorée (avec un message d'avertissement). Un index ne peut pas avoir un propriétaire différent de celui de la table. Modifier le propriétaire de la table modifie automatiquement celui de l'index.

Il est interdit de modifier toute partie d'un index du catalogue système.

Exemples

Renommer un index existant :

```
ALTER INDEX distributeurs RENAME TO fournisseurs;
```

Déplacer un index dans un autre tablespace :

```
ALTER INDEX distributeurs SET TABLESPACE espacelogiquerapide;
```

Pour modifier le facteur de remplissage d'un index (en supposant que la méthode d'indexage le supporte) :

```
ALTER INDEX distributeurs SET (fillfactor = 75);
REINDEX INDEX distributeurs;
```

Configure la cible de récupération des statistiques pour un index sur expression :

```
CREATE INDEX coord_idx ON measured (x, y, (z + t));
ALTER INDEX coord_idx ALTER COLUMN 3 SET STATISTICS 1000;
```

Compatibilité

`ALTER INDEX` est une extension PostgreSQL.

Voir aussi

CREATE INDEX, REINDEX

ALTER LANGUAGE

ALTER LANGUAGE — Modifier la définition d'un langage procédural

Synopsis

```
ALTER LANGUAGE nom RENAME TO nouveau_nom

ALTER LANGUAGE nom OWNER TO { nouveau_propriétaire | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER }
```

Description

`ALTER LANGUAGE` modifie la définition d'un langage. Les seules fonctionnalités disponibles sont le changement de nom et de propriétaire du langage. Vous devez être soit un superutilisateur soit le propriétaire du langage pour utiliser `ALTER LANGUAGE`.

Paramètres

nom

Le nom du langage.

nouveau_nom

Le nouveau nom du langage.

new_owner

Le nouveau propriétaire du langage

Compatibilité

Il n'existe pas d'instruction `ALTER LANGUAGE` dans le standard SQL.

Voir aussi

CREATE LANGUAGE, DROP LANGUAGE

ALTER LARGE OBJECT

ALTER LARGE OBJECT — Modifier la définition d'un Large Object

Synopsis

```
ALTER LARGE OBJECT oid_large_object OWNER TO { nouveau_propriétaire
 | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
```

Description

`ALTER LARGE OBJECT` modifie la définition d'un Large Object.

Le Large Object doit vous appartenir pour utiliser `ALTER LARGE OBJECT`. Pour modifier le propriétaire, vous devez être le membre direct ou indirect du nouveau rôle propriétaire. (Néanmoins, un superutilisateur peut modifier tout Large Object.) Actuellement, la seule fonctionnalité de cette instruction est le changement du propriétaire, donc ces deux restrictions s'appliquent toujours.

Paramètres

oid_large_object

OID d'un Large Object à modifier

nouveau_propriétaire

Le nouveau propriétaire du Large Object

Compatibilité

Il n'existe pas d'instruction `ALTER LARGE OBJECT` dans le standard SQL.

Voir aussi

Chapitre 35

ALTER MATERIALIZED VIEW

ALTER MATERIALIZED VIEW — modifier la définition d'une vue matérialisée

Synopsis

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] nom
    action [, ... ]
ALTER MATERIALIZED VIEW nom
    [ NO ] DEPENDS ON EXTENSION nom_extension
ALTER MATERIALIZED VIEW [ IF EXISTS ] nom
    RENAME [ COLUMN ] nom_colonne TO nouveau_nom_colonne
ALTER MATERIALIZED VIEW [ IF EXISTS ] nom
    RENAME TO nouveau_nom
ALTER MATERIALIZED VIEW [ IF EXISTS ] nom
    SET SCHEMA nouveau_schéma
ALTER MATERIALIZED VIEW ALL IN TABLESPACE nom [ OWNED BY nom_rôle
 [, ... ] ]
    SET TABLESPACE nouveau_tablespace [ NOWAIT ]

où action fait partie
de :

    ALTER [ COLUMN ] nom_colonne SET STATISTICS integer
    ALTER [ COLUMN ] nom_colonne SET ( option_colonne [= valeur]
 [, ... ] )
    ALTER [ COLUMN ] nom_colonne RESET ( option_colonne [, ... ] )
    ALTER [ COLUMN ] nom_colonne SET STORAGE { PLAIN | EXTERNAL |
 EXTENDED | MAIN }
    ALTER [ COLUMN ] nom_colonne SET
 COMPRESSION méthode_compression
    CLUSTER ON nom_index
    SET WITHOUT CLUSTER
    SET ACCESS METHOD nouvelle_méthode_accès
    SET TABLESPACE nouveau_tablespace
    SET ( paramètre_stockage = valeur [, ... ] )
    RESET ( paramètre_stockage [, ... ] )
    OWNER TO { nouveau_propriétaire | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
```

Description

`ALTER MATERIALIZED VIEW` modifie les différentes propriétés d'une vue matérialisée existante.

Vous devez être le propriétaire d'une vue matérialisée pour utiliser `ALTER MATERIALIZED VIEW`. Pour changer le schéma d'une vue matérialisée, vous devez aussi avoir le droit `CREATE` sur le nouveau schéma. Pour modifier le propriétaire, vous devez aussi être un membre direct ou indirect du nouveau rôle propriétaire et ce rôle doit avoir le droit `CREATE` sur le schéma de la vue matérialisée. (Ces restrictions assurent que la modification du propriétaire ne vous permet pas plus que ce que vous pourriez faire en supprimant puis récréant la vue matérialisée. Néanmoins, un superutilisateur peut modifier le propriétaire d'une vue.)

Les différentes formes et actions disponibles pour `ALTER MATERIALIZED VIEW` sont un sous-ensemble de celles disponibles pour `ALTER TABLE`, et ont la même signification quand elles sont utilisées pour les vues matérialisées. Pour plus de détails, voir les descriptions sur `ALTER TABLE`.

Paramètrs

nom

Nom, potentiellement qualifié du nom du schéma, d'une vue matérialisée existante.

nom_colonne

Nom d'une colonne déjà existante.

nom_extension

Nom de l'extension dont dépend la vue matérialisée (ou dont elle ne dépend plus si le mot-clé `NO` est ajouté). Une vue matérialisée dépendante d'une extension est automatiquement supprimée quand l'extension est supprimée.

nouveau_nom_colonne

Nouveau nom d'une colonne existante.

nouveau_propriétaire

Nom utilisateur du nouveau propriétaire de la vue matérialisée.

nouveau_nom

Nouveau nom de la vue matérialisée.

nouveau_schéma

Nouveau schéma de la vue matérialisée.

Exemples

Renommer la vue matérialisée `truc` en `chose` :

```
ALTER MATERIALIZED VIEW truc RENAME TO chose;
```

Compatibilité

`ALTER MATERIALIZED VIEW` est une extension PostgreSQL.

Voir aussi

CREATE MATERIALIZED VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

ALTER OPERATOR

ALTER OPERATOR — Modifier la définition d'un opérateur

Synopsis

```
ALTER OPERATOR nom ( { type_gauche | NONE } , type_droit )
    OWNER TO { nouveau_propriétaire | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }

ALTER OPERATOR nom ( { type_gauche | NONE } , type_droit )
    SET SCHEMA nouveau_schema

ALTER OPERATOR nom ( { type_gauche | NONE } , type_droit )
    SET ( {  RESTRICT = { proc_res | NONE }
           | JOIN = { proc_join | NONE }
         } [, ... ] )
```

Description

`ALTER OPERATOR` modifie la définition d'un opérateur.

Seul le propriétaire de l'opérateur peut utiliser `ALTER OPERATOR`. Pour modifier le propriétaire, il est nécessaire d'être un membre direct ou indirect du nouveau rôle propriétaire, et ce rôle doit avoir le droit `CREATE` sur le schéma de l'opérateur. Ces restrictions assurent que la modification du propriétaire produise le même résultat que la suppression et la re-création de l'opérateur ; néanmoins, un superutilisateur peut modifier le propriétaire de n'importe quel opérateur.

Paramètres

nom

Le nom de l'opérateur (éventuellement qualifié du nom du schéma).

type_gauche

Le type de données de l'opérande gauche de l'opérateur ; `NONE` si l'opérateur n'a pas d'opérande gauche.

type_droit

Le type de données de l'opérande droit de l'opérateur.

nouveau_propriétaire

Le nouveau propriétaire de l'opérateur.

nouveau_schéma

Le nouveau schéma de l'opérateur.

proc_res

La fonction d'estimation de la sélectivité de restriction pour cet opérateur ; écrire NONE pour supprimer cet estimateur.

join_proc

La fonction d'estimation de la sélectivité de jointure pour cet opérateur ; écrire NONE pour supprimer cet estimateur.

Exemples

Modifier le propriétaire d'un opérateur personnalisé `a @@ b` pour le type `text` :

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

Modifier les fonctions de sélectivité de restriction et de jointure pour un opérateur personnalisé `a && b` pour le type `int[]` ::

```
ALTER OPERATOR && (_int4, _int4) SET (RESTRICT = _int_contsel, JOIN
 = _int_contjoinsel);
```

Compatibilité

Il n'existe pas d'instructions `ALTER OPERATOR` dans le standard SQL.

Voir aussi

CREATE OPERATOR, DROP OPERATOR

ALTER OPERATOR CLASS

ALTER OPERATOR CLASS — Modifier la définition d'une classe d'opérateurs

Synopsis

```
ALTER OPERATOR CLASS nom USING méthode_indexage
    RENAME TO nouveau_nom

ALTER OPERATOR CLASS nom USING méthode_indexage
    OWNER TO { nouveau_propriétaire | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }

ALTER OPERATOR CLASS nom USING méthode_indexage
    SET SCHEMA nouveau_schéma
```

Description

`ALTER OPERATOR CLASS` modifie la définition d'une classe d'opérateur.

Seul le propriétaire de la classe d'opérateur peut utiliser `ALTER OPERATOR CLASS`. Pour modifier le propriétaire, il est obligatoire d'être un membre direct ou indirect du nouveau rôle propriétaire. Ce rôle doit avoir le droit `CREATE` sur le schéma de la classe d'opérateur. Ces restrictions assurent que la modification du propriétaire produise le même effet que celui obtenu par la suppression et la re-création de la classe d'opérateur ; néanmoins, un superutilisateur peut modifier le propriétaire de n'importe quelle classe d'opérateur.

Paramètres

nom

Le nom d'une classe d'opérateur.

méthode_indexage

Le nom de la méthode d'indexage à laquelle associer la classe d'opérateur.

nouveau_nom

Le nouveau nom de la classe d'opérateur.

nouveau_propriétaire

Le nouveau propriétaire de la classe d'opérateur.

nouveau_schéma

Le nouveau schéma de la classe d'opérateur.

Compatibilité

Il n'existe pas d'instruction `ALTER OPERATOR CLASS` dans le standard SQL.

Voir aussi

CREATE OPERATOR CLASS, DROP OPERATOR CLASS, ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY — Modifier la définition d'une famille d'opérateur

Synopsis

```
ALTER OPERATOR FAMILY nom USING methode_indexage ADD
  {  OPERATOR numéro_stratégie nom_opérateur ( type_op, type_op )
             [ FOR SEARCH | FOR ORDER BY nom_famille_tri ]
   | FUNCTION numéro_support [ ( type_op [ , type_op ] ) ]
             nom_fonction [ ( type_argument [, ...] ) ]
  } [, ... ]
ALTER OPERATOR FAMILY nom USING methode_indexage DROP
  {  OPERATOR numero_strategie ( type_op [ , type_op ] )
   | FUNCTION numero_support ( type_op [ , type_op ] )
  } [, ... ]

ALTER OPERATOR FAMILY nom USING methode_indexation
    RENAME TO nouveau_nom

ALTER OPERATOR FAMILY nom USING methode_indexation
    OWNER TO { nouveau_propriétaire | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }

ALTER OPERATOR FAMILY nom USING methode_indexation
    SET SCHEMA nouveau_schéma
```

Description

`ALTER OPERATOR FAMILY` modifie la définition d'une famille d'opérateur. Vous pouvez ajouter des opérateurs et des fonctions du support à la famille, les supprimer ou modifier le nom et le propriétaire de la famille.

Quand les opérateurs et fonctions de support sont ajoutés à une famille avec la commande `ALTER OPERATOR FAMILY`, ils ne font partie d'aucune classe d'opérateurs spécifique à l'intérieur de la famille. Ils sont « lâches » dans la famille. Ceci indique que ces opérateurs et fonctions sont compatibles avec la sémantique de la famille mais qu'ils ne sont pas requis pour un fonctionnement correct d'un index spécifique. (Les opérateurs et fonctions qui sont ainsi nécessaires doivent être déclarés comme faisant partie d'une classe d'opérateurs ; voir CREATE OPERATOR CLASS.) PostgreSQL autorise la suppression des membres lâches d'une famille à tout moment, mais les membres d'une classe d'opérateurs ne peuvent pas être supprimés sans supprimer toute la classe et les index qui en dépendent. Typiquement, les opérateurs et fonctions sur un seul type de données font partie des classes d'opérateurs car ils ont besoin de supporter un index sur ce type de données spécifique alors que les opérateurs et familles inter-types sont fait de membres lâches de la famille.

Vous devez être superutilisateur pour utiliser `ALTER OPERATOR FAMILY`. (Cette restriction est faite parce qu'une définition erronée d'une famille d'opérateur pourrait gêner, voire même arrêter brutalement le serveur.)

`ALTER OPERATOR FAMILY` ne vérifie pas encore si la définition de l'opérateur de famille inclut tous les opérateurs et fonctions requis par la méthode d'indexage, ni si les opérateurs et les fonctions forment un ensemble cohérent et suffisant. C'est de la responsabilité de l'utilisateur de définir une famille d'opérateur valide.

Voir Section 38.16 pour plus d'informations.

Paramètres

nom

Le nom d'une famille d'opérateur (pouvant être qualifié du schéma).

methode_indexage

Le nom de la méthode d'indexage.

numero_strategie

Le numéro de stratégie de la méthode d'indexage pour un opérateur associé avec la famille.

nom_operateur

Le nom d'un opérateur (pouvant être qualifié du schéma) associé avec la famille d'opérateur.

type_op

Dans une clause `OPERATOR`, les types de données en opérande de l'opérateur, ou `NONE` pour signifier un opérateur préfixe. Contrairement à la syntaxe comparable de `CREATE OPERATOR CLASS`, les types de données en opérande doivent toujours être précisés.

Dans une clause `ADD FUNCTION`, les types de données des opérandes que la fonction est sensée supporter, si différent des types de données en entrée de la fonction. Pour les fonctions de comparaison des index B-tree et hash, il n'est pas strictement nécessaire de spécifier *type_op* car les types de données en entrée de la fonction sont toujours les bons à utiliser. Pour les fonctions de tri des index B-tree, les fonctions d'égalité d'image B-Tree, ainsi que pour toutes les fonctions des classes d'opérateurs GIST, SP-GiST et GIN, il est nécessaire de spécifier le type de données en entrée qui sera utilisé par la fonction.

Dans une clause `DROP FUNCTION`, les types de données en opérande que la fonction est sensée supporter doivent être précisés. Pour les index GiST, SP-GiST et GIN, les types en question pourraient ne pas être identiques aux des arguments en entrée de la fonction.

nom_famille_tri

Le nom d'une famille d'opérateur `btree` (pouvant être qualifié du schéma) décrivant l'ordre de tri associé à l'opérateur de tri.

Si ni `FOR SEARCH` ni `FOR ORDER BY` ne sont indiqués, `FOR SEARCH` est la valeur par défaut.

numero_support

Le numéro de la fonction de support de la méthode d'indexage associé avec la famille d'opérateur.

nom_fonction

Le nom (en option qualifié du schéma) d'une fonction qui est une fonction de support d'une méthode d'index pour la famille d'opérateur. Si aucune liste d'argument n'est spécifiée, le nom doit être unique dans son schéma.

argument_types

Les types de données pour les arguments de la fonction.

nouveau_nom

Le nouveau nom de la famille d'opérateur

nouveau_propriétaire

Le nouveau propriétaire de la famille d'opérateur

nouveau_schéma

Le nouveau schéma de la famille d'opérateur.

Les clauses `OPERATOR` et `FUNCTION` peuvent apparaître dans n'importe quel ordre.

Notes

Notez que la syntaxe `DROP` spécifie uniquement le « slot » dans la famille d'opérateur, par stratégie ou numéro de support et types de données en entrée. Le nom de l'opérateur ou de la fonction occupant le slot n'est pas mentionné. De plus, pour `DROP FUNCTION`, les types à spécifier sont les types de données en entrée que la fonction doit supporter ; pour les index GIN et GiST, ceci pourrait ne rien avoir à faire avec les types d'argument en entrée de la fonction.

Comme le processus des index ne vérifie pas les droits sur les fonctions avant de les utiliser, inclure une fonction ou un opérateur dans une famille d'opérateur est équivalent à donner le droit d'exécution à public. Ceci n'est généralement pas un problème pour les tris de fonction qui sont utiles à une famille d'opérateur.

Les opérateurs ne doivent pas être définis par des fonctions SQL. Une fonction SQL risque d'être remplacée dans la requête appelante, ce qui empêchera l'optimiseur de savoir si la requête peut utiliser un index.

Avant PostgreSQL 8.4, la clause `OPERATOR` pouvait inclure une option `RECHECK`. Ce n'est plus supporté parce que le fait qu'un opérateur d'index soit « à perte » est maintenant déterminé à l'exécution. Cela permet une gestion plus efficace des cas où un opérateur pourrait ou non être à perte.

Exemples

La commande exemple suivant ajoute des opérateurs inter-type de données et ajoute les fonctions de support pour une famille d'opérateur qui contient déjà les classes d'opérateurs Btree pour les types de données `int4` et `int2`.

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

Pour supprimer de nouveau ces entrées :

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP

  -- int4 vs int2
  OPERATOR 1 (int4, int2) ,
  OPERATOR 2 (int4, int2) ,
  OPERATOR 3 (int4, int2) ,
  OPERATOR 4 (int4, int2) ,
  OPERATOR 5 (int4, int2) ,
  FUNCTION 1 (int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 (int2, int4) ,
  OPERATOR 2 (int2, int4) ,
  OPERATOR 3 (int2, int4) ,
  OPERATOR 4 (int2, int4) ,
  OPERATOR 5 (int2, int4) ,
  FUNCTION 1 (int2, int4) ;
```

Compatibilité

Il n'existe pas d'instruction `ALTER OPERATOR FAMILY` dans le standard SQL.

Voir aussi

CREATE OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

ALTER POLICY

ALTER POLICY — modifie la définition du niveau d'ordre de la politique de sécurité

Synopsis

```
ALTER POLICY nom ON nom_table RENAME TO nouveau_nom

ALTER POLICY nom ON nom_table
    [ TO { nom_rôle | PUBLIC | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER } [, ...] ]
    [ USING ( expression_USING ) ]
    [ WITH CHECK ( expression_CHECK ) ]
```

Description

`ALTER POLICY` modifie la définition du niveau d'ordre existant de la politique de sécurité. Il est à noter que `ALTER POLICY` autorise uniquement l'ensemble des rôles auquel la politique de sécurité s'applique et uniquement la modification des expressions `USING` et `WITH CHECK`. Pour changer d'autres propriétés d'une politique de sécurité, comme la commande à laquelle elle s'applique ou si elle est permissive ou restrictive, la politique de sécurité doit être supprimée et recrée.

Pour vous servir de la commande `ALTER POLICY`, vous devez être propriétaire de la table à laquelle cette politique s'applique.

Dans la deuxième forme de `ALTER POLICY`, la liste des rôles, *`expression_USING`* et *`expression_CHECK`* sont remplacés de manière indépendante s'ils sont spécifiés. Lorsqu'une des clauses n'est pas spécifiée, la partie correspondante dans la politique de sécurité n'est pas modifiée.

Paramètres

`nom`

Le nom de la politique existante à modifier.

`nom_table`

Le nom de la table sur laquelle la politique est appliquée (éventuellement qualifiée par le schéma).

`nouveau_nom`

Le nouveau nom de la politique.

`nom_rôle`

Le ou les rôle(s) auxquels la politique s'applique. Plusieurs rôles peuvent être spécifiés en une fois. Pour appliquer la politique à tous les rôles, vous pouvez utiliser `PUBLIC`.

`expression_USING`

Expression définie pour la clause `USING` de la politique. Voir CREATE POLICY pour plus de détails.

`expression_CHECK`

Expression définie pour la clause `WITH CHECK` de la politique. Voir CREATE POLICY pour plus de détails.

Compatibilité

`ALTER POLICY` est une extension PostgreSQL.

Voir aussi

CREATE POLICY, DROP POLICY

ALTER PROCEDURE

ALTER PROCEDURE — Modifier la définition d'une procédure

Synopsis

```
ALTER PROCEDURE nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]

ALTER PROCEDURE nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    RENAME TO nouveau_nom

ALTER PROCEDURE nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    OWNER TO { nouveau_propriétaire | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }

ALTER PROCEDURE nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    SET SCHEMA nouveau_schéma

ALTER PROCEDURE nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    [ NO ] DEPENDS ON EXTENSION nom_extension

où action fait partie
de :

    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    SET paramètre_de_configuration { TO | = } { valeur | DEFAULT }
    SET paramètre_de_configuration FROM CURRENT
    RESET paramètre_de_configuration
    RESET ALL
```

Description

`ALTER PROCEDURE` modifie la définition d'une procédure.

Seul le propriétaire de la procédure peut utiliser `ALTER PROCEDURE`. Le droit `CREATE` sur le nouveau schéma est requis pour pouvoir changer le schéma de la procédure. Pour modifier le propriétaire, il est nécessaire d'être membre direct ou indirect du nouveau rôle propriétaire. Ce dernier doit posséder le droit `CREATE` sur le schéma de la procédure. Ces restrictions assurent que la modification du propriétaire n'a pas d'effets autres que ceux obtenus par la suppression et la re-création de la procédure ; toutefois, un superutilisateur peut modifier le propriétaire de n'importe quelle procédure.

Paramètres

nom

Le nom d'une procédure existante (éventuellement qualifié par le schéma). Si aucune liste d'arguments n'est spécifiée, le nom doit être unique dans son schéma.

mode_arg

Le mode d'un argument : `IN`, `OUT`, `INOUT` ou `VARIADIC`. Si non précisé, le défaut est `IN`.

nom_arg

Le nom d'un argument. Notez que `ALTER PROCEDURE` ne fait pas vraiment attention aux noms des arguments, puisqu'il n'a besoin que des types des arguments pour déterminer la procédure.

type_arg

Les types de données des arguments de la procédure (éventuellement qualifiés par le schéma), s'il y en a. Voir DROP PROCEDURE pour les détails sur comment la procédure est recherchée en utilisant les types de données des arguments.

nouveau_nom

Le nouveau nom de la procédure.

nouveau_propriétaire

Le nouveau propriétaire de la procédure. Si cette procédure est marquée `SECURITY DEFINER`, elle s'exécute par la suite sous cette identité.

nouveau_schema

Le nouveau schéma de la procédure.

nom_extension

Cette forme marque la procédure comme dépendante de l'extension, ou comme indépendante de l'extension si `NO` est ajoutée. Une procédure marquée comme dépendante d'une extension est supprimée quand l'extension est supprimée même si la cascade n'est pas explicitement demandée. Une procédure peut dépendre de plusieurs extensions, et ne sera supprimée que quand une de ces extensions est supprimée.

```
[ EXTERNAL ] SECURITY INVOKER
[ EXTERNAL ] SECURITY DEFINER
```

Précise si la procédure doit être appelée avec les droits de l'utilisateur qui l'a créée. Le mot clé `EXTERNAL`, ignoré, existe pour des raisons de compatibilité SQL. Voir CREATE PROCEDURE pour plus d'informations.

paramètre_de_configuration
valeur

Ajoute ou modifie l'initialisation d'un paramètre de configuration lorsque la procédure est appelée. Si *valeur* est `DEFAULT` ou, de façon équivalente, si `RESET` est utilisé, le paramètre local de la procédure est supprimé pour que la procédure s'exécute avec la valeur par défaut du paramètre. Utiliser `RESET ALL` supprime toutes les valeurs spécifiques des paramètres pour cette procédure. `SET FROM CURRENT` sauvegarde la valeur actuelle du paramètre quand `ALTER PROCEDURE` est exécuté comme valeur à appliquer lors de l'exécution de la procédure.

Voir SET et Chapitre 20 pour plus d'informations sur les noms des paramètres et les valeurs autorisés.

`RESTRICT`

Ignoré, présent pour des raisons de conformité avec le standard SQL.

Exemples

Renommer la procédure `insert_data` ayant deux arguments de type `integer` vers `insert_record` :

```
ALTER PROCEDURE insert_data(integer, integer) RENAME TO
 insert_record;
```

Changer le propriétaire de la procédure `insert_data` ayant deux arguments de type `integer` vers `joe` :

```
ALTER PROCEDURE insert_data(integer, integer) OWNER TO joe;
```

Changer le schéma de la procédure `insert_data` ayant deux arguments de type `integer` par `accounting` :

```
ALTER PROCEDURE insert_data(integer, integer) SET SCHEMA
 accounting;
```

Marquer la procédure `insert_data(integer, integer)` comme dépendante de l'extension `myext` :

```
ALTER PROCEDURE insert_data(integer, integer) DEPENDS ON EXTENSION
 myext;
```

Pour ajuster automatiquement le chemin de recherche des schémas pour une procédure :

```
ALTER PROCEDURE check_password(text) SET search_path = admin,
 pg_temp;
```

Pour désactiver le paramètre `search_path` d'une procédure :

```
ALTER PROCEDURE check_password(text) RESET search_path;
```

La procédure s'exécutera maintenant avec la valeur de la session pour cette variable.

Compatibilité

La compatibilité de cette instruction avec l'instruction `ALTER PROCEDURE` du standard SQL est partielle. Le standard autorise la modification d'un plus grand nombre de propriétés d'une procédure mais ne laisse pas la possibilité de renommer une procédure, de placer l'option `SECURITY DEFINER` sur la procédure, d'y attacher des valeurs de paramètres ou d'en modifier le propriétaire, le schéma ou la volatilité. Le standard requiert le mot clé `RESTRICT` ; il est optionnel avec PostgreSQL.

Voir aussi

CREATE PROCEDURE, DROP PROCEDURE, ALTER FUNCTION, ALTER ROUTINE

ALTER PUBLICATION

ALTER PUBLICATION — change la définition d'une publication

Synopsis

```
ALTER PUBLICATION nom ADD objet_publication [, ...]
ALTER PUBLICATION nom SET objet_publication [, ...]
ALTER PUBLICATION nom DROP objet_supprime_publication [, ...]
ALTER PUBLICATION nom SET ( paramètre_publication [= valeur]
 [, ... ] )
ALTER PUBLICATION nom OWNER TO { nouveau_propriétaire |
 CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER PUBLICATION nom RENAME TO nouveau_nom

où objet_publication est
parmi :

    TABLE table_et_colonnes [, ... ]
    TABLES IN SCHEMA { nom_schéma | CURRENT_SCHEMA } [, ... ]

et objet_supprime_publication fait partie de :

    TABLE [ ONLY ] nom_table [ * ] [, ... ]
    TABLES IN SCHEMA { nom_schéma | CURRENT_SCHEMA } [, ... ]

et table_et_colonnes vaut :

    [ ONLY ] nom_table [ * ] [ ( nom_colonne [, ... ] ) ] [ WHERE
 ( expression ) ]
```

Description

La commande ALTER PUBLICATION peut modifier les attributs d'une publication.

Les trois premières variantes modifient les tables/schémas faisant partie de la publication. La clause SET remplacera la liste des tables/schémas de la publication avec celle indiquée ; les schémas/tables existant qui étaient présents dans la publication seront supprimés. Les clauses ADD et DROP, respectivement, ajouteront et supprimeront une ou plusieurs tables/schémas à la publication. Notez qu'ajouter des tables/schémas à une publication où des souscriptions ont déjà eu lieu nécessitera un ALTER SUBSCRIPTION ... REFRESH PUBLICATION du côté de l'abonné pour devenir réelle. Notez aussi que DROP TABLES IN SCHEMA ne supprimera aucune table du schéma qui a été ajoutée en utilisant FOR TABLE/ADD TABLE.

La quatrième variante de cette commande listée dans le synopsis peut changer toutes les propriétés de la publication spécifiées dans CREATE PUBLICATION. Les propriétés qui ne sont pas mentionnées dans la commande restent à leurs anciennes valeurs.

Les variantes suivantes modifient le propriétaire et le nom de la publication.

Vous devez être le propriétaire de la publication pour utiliser ALTER PUBLICATION. Ajouter une table à une publication requiert en plus d'être le propriétaire de cette table. Les clauses ADD TABLES IN SCHEMA et SET TABLES IN SCHEMA pour une publication requièrent que l'utilisateur exécutant la requête ait l'attribut SUPERUSER. Pour modifier le propriétaire, vous devez aussi être

un membre direct ou indirect du nouveau rôle propriétaire. Le nouveau propriétaire doit avoir le droit CREATE sur la base. De plus, le nouveau propriétaire d'une publication FOR ALL TABLES et FOR TABLES IN SCHEMA doit avoir l'attribut SUPERUSER. Cependant, un superutilisateur peut modifier le propriétaire d'une publication sans ces restrictions.

Ajouter/configurer tout schéma quand la publication quand la publication publie aussi une table avec une liste de colonnes et vice-verse n'est pas supporté.

Paramètres

name

Le nom d'une publication existante dont la définition doit être modifiée.

nom_table

Nom d'une table existante. Si ONLY est spécifié avant le nom de la table, seule cette table est affectée. Si ONLY n'est pas spécifié, la table et toutes les tables descendantes (s'il y en a) sont affectées. * peut être spécifié de manière facultative après le nom de la table pour indiquer explicitement que les tables descendantes doivent être inclues.

En supplément, une liste de colonnes peut être indiquée. Voir CREATE PUBLICATION pour les détails. Notez qu'une souscription ayant plusieurs publications pour lesquelles la même table a été publié avec des listes de colonnes différentes n'est pas supporté. Voir Attention : Combiner des listes de plusieurs publications pour des détails sur des problèmes potentiels lors de la modification des listes de colonnes.

Si la clause optionnelle WHERE est indiquée, les lignes pour lesquelles *expression* est évaluée à false ou NULL ne seront pas publiées. Notez que des parenthèses sont requises autour de l'expression. *expression* est évalué avec le rôle utilisé pour la connexion de réplication.

nom_schéma

Nom d'un schéma existant.

SET (*param_publication* [= *valeur*] [, ...])

Cette clause change les paramètres de la publication positionnés à l'origine par CREATE PUBLICATION. Consulter cette page pour plus d'information.

nouveau_proprietaire

Le nom d'utilisateur du nouveau propriétaire de la publication.

nouveau_nom

Le nouveau nom de la publication.

Exemples

Changer la publication pour ne publier que les suppressions et les mises à jour :

```
ALTER PUBLICATION noinsert SET (publish = 'update, delete');
```

Ajouter des tables à la publication :

```
ALTER PUBLICATION mypublication ADD TABLE users (user_id,
 firstname), departments;
```

Modifier l'ensemble de colonnes publiées pour une table :

```
ALTER PUBLICATION mypublication SET TABLE users (user_id,
 firstname, lastname), TABLE departments;
```

Ajouter les schémas `marketing` et `sales` à la publication `sales_publication` :

```
ALTER PUBLICATION sales_publication ADD TABLES IN SCHEMA marketing,
 sales;
```

Ajouter les tables `users`, `departments` et le schéma `production` à la publication `production_publication` :

```
ALTER PUBLICATION production_publication ADD TABLE users,
 departments, TABLES IN SCHEMA production;
```

Compatibilé

`ALTER PUBLICATION` est une extension PostgreSQL au standard SQL.

Voir aussi

CREATE PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

ALTER ROLE

ALTER ROLE — Modifier un rôle de base de données

Synopsis

```
ALTER ROLE spécification_rôle [ WITH ] option [ ... ]

où option peut être :

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT limiteconnexion
    | [ ENCRYPTED ] PASSWORD 'motdepasse' | PASSWORD NULL
    | VALID UNTIL 'dateheure'

ALTER ROLE nom RENAME TO nouveau_nom

ALTER ROLE { spécification_rôle | ALL } [ IN DATABASE nom_base ]
 SET paramètre_configuration { TO | = } { value | DEFAULT }
ALTER ROLE { spécification_rôle | ALL } [ IN DATABASE nom_base ]
 SET paramètre_configuration FROM CURRENT
ALTER ROLE { spécification_rôle | ALL } [ IN DATABASE nom_base ]
 RESET paramètre_configuration
ALTER ROLE { spécification_rôle | ALL } [ IN DATABASE nom_base ]
 RESET ALL

où spécification_rôle peut valoir :

    nom_rôle
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
```

Description

`ALTER ROLE` modifie les propriétés d'un rôle PostgreSQL.

La première variante listée dans le synopsis permet de modifier la plupart des attributs de rôle spécifiables dans la commande `CREATE ROLE`. (Tous les attributs possibles sont couverts, à l'exception de la gestion des appartenances ; `GRANT` et `REVOKE` sont utilisés pour cela.) Les attributs qui ne sont pas mentionnés dans la commande conservent leur paramétrage précédent. Tous ces attributs peuvent être modifiés par les superutilisateurs de base de données. Les rôles qui possèdent l'attribut `CREATEROLE` peuvent modifier ces paramètres à l'exception de `SUPERUSER`, `REPLICATION` et `BYPASSRLS`, mais uniquement pour les rôles qui ne sont pas superutilisateur. Les rôles ordinaires ne peuvent modifier que leur mot de passe.

La deuxième variante permet de modifier le nom du rôle. Les superutilisateurs peuvent renommer n'importe quel rôle. Les rôles disposant de l'attribut `CREATEROLE` peuvent renommer tout rôle qui n'est pas superutilisateur. L'utilisateur de la session en cours ne peut pas être renommé. (On se

connectera sous un autre utilisateur pour cela.) Comme les mots de passe chiffrés par MD5 utilisent le nom du rôle comme sel de chiffrement, renommer un rôle efface son mot de passe si ce dernier est chiffré avec MD5.

Les autres variantes modifient la valeur par défaut d'une variable de configuration de session pour un rôle, soit pour toutes les bases soit, quand la clause IN DATABASE est spécifiée, uniquement pour les sessions dans la base nommée. Si ALL est indiqué à la place d'un nom de rôle, ceci modifie le paramétrage de tous les rôles. Utiliser ALL avec IN DATABASE est en effet identique à utiliser la commande ALTER DATABASE ... SET ....

Quand le rôle lance une nouvelle session après cela, la valeur spécifiée devient la valeur par défaut de la session, surchargeant tout paramétrage présent dans postgresql.conf ou provenant de la ligne de commande de postgres. Ceci arrive seulement lors de la connexion ; exécuter SET ROLE ou SET SESSION AUTHORIZATION ne cause pas la configuration de nouvelles valeurs pour les paramètres. L'ensemble des paramètres pour toutes les bases est surchargé par les paramètres spécifiques à cette base attachés à un rôle. La configuration pour une base de données spécifique ou pour un rôle spécifique surcharge la configuration pour tous les rôles.

Les superutilisateurs peuvent modifier les valeurs de session de n'importe quel utilisateur. Les rôles disposant de l'attribut CREATEROLE peuvent modifier les valeurs par défaut pour les rôles ordinaires (non superutilisateurs et non réplication). Les rôles standards peuvent seulement configurer des valeurs par défaut pour eux-mêmes. Certaines variables ne peuvent être configurées de cette façon ou seulement par un superutilisateur. Seuls les superutilisateurs peuvent modifier un paramétrage pour tous les rôles dans toutes les bases de données.

Paramètres

nom

Le nom du rôle dont les attributs sont modifiés.

CURRENT_ROLE
CURRENT_USER

Modifie l'utilisateur actuel au lieu d'un rôle identifié explicitement.

SESSION_USER

Modifie l'utilisateur de la session courante au lieu d'un rôle identifié explicitement.

SUPERUSER
NOSUPERUSER
CREATEDB
NOCREATEDB
CREATEROLE
NOCREATEROLE
INHERIT
NOINHERIT
LOGIN
NOLOGIN
REPLICATION
NOREPLICATION
BYPASSRLS
NOBYPASSRLS
CONNECTION LIMIT *limite_connexion*
[ENCRYPTED] PASSWORD *mot_de_passe* | PASSWORD NULL
VALID UNTIL '*dateheure*'

Ces clauses modifient les attributs originairement configurés par CREATE ROLE. Pour plus d'informations, voir la page de référence CREATE ROLE.

nouveau_nom

Le nouveau nom du rôle.

nom_base

Le nom d'une base où se fera la configuration de la variable.

paramètre_configuration
valeur

Positionne la valeur de session par défaut à *valeur* pour le paramètre de configuration *paramètre*. Si DEFAULT est donné pour *valeur* ou, de façon équivalente, si RESET est utilisé, le positionnement spécifique de la variable pour le rôle est supprimé. De cette façon, le rôle hérite de la valeur système par défaut pour les nouvelles sessions. RESET ALL est utilisé pour supprimer tous les paramétrages du rôle. SET FROM CURRENT sauvegarde la valeur de la session de ce paramètre en tant que valeur du rôle. Si IN DATABASE est précisé, le paramètre de configuration est initialisé ou supprimé seulement pour le rôle et la base indiqués.

Les paramètres spécifiques au rôle ne prennent effet qu'à la connexion ; SET ROLE et SET SESSION AUTHORIZATION ne traitent pas les paramètres de rôles.

Voir SET et Chapitre 20 pour plus d'informations sur les noms et les valeurs autorisés pour les paramètres.

Notes

CREATE ROLE est utilisé pour ajouter de nouveaux rôles et DROP ROLE pour les supprimer.

ALTER ROLE ne peut pas modifier les appartenances à un rôle. GRANT et REVOKE sont conçus pour cela.

Faites attention lorsque vous précisez un mot de passe non chiffré avec cette commande. Le mot de passe sera transmis en clair au serveur. Il pourrait se trouver tracer dans l'historique des commandes du client et dans les traces du serveur. psql contient une commande \password qui peut être utilisé pour changer le mot de passe d'un rôle sans exposer le mot de passe en clair.

Il est également possible de lier une valeur de session par défaut à une base de données plutôt qu'à un rôle ; voir ALTER DATABASE. S'il y a un conflit, les paramètres spécifiques à la paire base de données/rôle surchargent ceux spécifiques au rôle, qui eux-même surchargent ceux spécifiques à la base de données.

Exemples

Modifier le mot de passe d'un rôle :

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

Supprimer le mot de passe d'un rôle :

```
ALTER ROLE davide WITH PASSWORD NULL;
```

Modifier la date d'expiration d'un mot de passe, en spécifiant que le mot de passe doit expirer à midi le 4 mai 2015 fuseau horaire UTC plus 1 heure :

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

Créer un mot de passe toujours valide :

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

Donner à un rôle la capacité de créer d'autres rôles et de nouvelles bases de données :

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

Donner à un rôle une valeur différente de celle par défaut pour le paramètre maintenance_work_mem :

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

Donner à un rôle une configuration différente, spécifique à une base de données, du paramètre client_min_messages :

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```

Compatibilité

L'instruction `ALTER ROLE` est une extension PostgreSQL.

Voir aussi

CREATE ROLE, DROP ROLE, ALTER DATABASE, SET

ALTER ROUTINE

ALTER ROUTINE — Modifier la définition d'une routine

Synopsis

```
ALTER ROUTINE nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER ROUTINE nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    RENAME TO nouveau_nom
ALTER ROUTINE nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    OWNER TO { nouveau_propriétaire | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
ALTER ROUTINE nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    SET SCHEMA nouveau_schéma
ALTER ROUTINE nom [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ]
    [ NO ] DEPENDS ON EXTENSION nom_extension

où action peut
être :

    IMMUTABLE | STABLE | VOLATILE
    [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST cout_exécution
    ROWS nb_lignes_résultat
    SET paramètre_configuration { TO | = } { valeur | DEFAULT }
    SET paramètre_configuration FROM CURRENT
    RESET paramètre_configuration
    RESET ALL
```

Description

`ALTER ROUTINE` modifie la définition d'une routine, qui peut être une fonction d'agrégat, une fonction normale ou une procédure. Voir ALTER AGGREGATE, ALTER FUNCTION, et ALTER PROCEDURE pour la description des paramètres, plus d'exemples et plus de détails.

Exemples

Pour renommer la routine `foo` ayant un argument de type `integer` en `foobar` :

```
ALTER ROUTINE foo(integer) RENAME TO foobar;
```

Cette commande fonctionnera indépendamment du fait que `foo` soit une fonction d'agrégat, une fonction ou une procédure.

Compatibilité

Cette commande est partiellement compatible avec la commande `ALTER ROUTINE` du standard SQL. Voir ALTER FUNCTION et ALTER PROCEDURE pour plus de détails. Autoriser les noms de routine à se référer à des noms de fonctions d'agrégat est une extension de PostgreSQL.

Voir aussi

ALTER AGGREGATE, ALTER FUNCTION, ALTER PROCEDURE, DROP ROUTINE

Veuillez noter qu'il n'existe pas de commande `CREATE ROUTINE`.

ALTER RULE

ALTER RULE — modifier la définition d'une règle

Synopsis

```
ALTER RULE nom ON nom_table RENAME TO nouveau_nom
```

Description

`ALTER RULE` modifie les propriétés d'une règle existante. Actuellement, la seule action disponible est de modifier le nom de la règle.

Pour utiliser `ALTER RULE`, vous devez être le propriétaire de la table ou de la vue sur laquelle s'applique la règle.

Paramètres

nom

Le nom d'une règle existante à modifier.

nom_table

Le nom (potentiellement qualifié du schéma) de la table ou de la vue sur laquelle s'applique la règle.

nouveau_nom

Le nouveau nom de la règle.

Exemples

Renommer une règle existante :

```
ALTER RULE tout_notifier ON emp RENAME TO notifie_moi;
```

Compatibilité

`ALTER RULE` est une extension de PostgreSQL, comme tout le système de réécriture des requêtes.

Voir aussi

CREATE RULE, DROP RULE

ALTER SCHEMA

ALTER SCHEMA — Modifier la définition d'un schéma

Synopsis

```
ALTER SCHEMA nom RENAME TO nouveau_nom
ALTER SCHEMA nom OWNER TO { nouveau_propriétaire | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER }
```

Description

`ALTER SCHEMA` modifie la définition d'un schéma.

Seul le propriétaire du schéma peut utiliser `ALTER SCHEMA`. Pour renommer le schéma, le droit `CREATE` sur la base est obligatoire. Pour modifier le propriétaire, il faut être membre, direct ou indirect, du nouveau rôle propriétaire, et posséder le droit `CREATE` sur la base (les superutilisateurs ont automatiquement ces droits).

Paramètres

nom

Le nom du schéma.

nouveau_nom

Le nouveau nom du schéma. Il ne peut pas commencer par `pg_`, noms réservés aux schémas système.

nouveau_propriétaire

Le nouveau propriétaire du schéma.

Compatibilité

Il n'existe pas de relation `ALTER SCHEMA` dans le standard SQL.

Voir aussi

CREATE SCHEMA, DROP SCHEMA

ALTER SEQUENCE

ALTER SEQUENCE — Modifier la définition d'un générateur de séquence

Synopsis

```
ALTER SEQUENCE [ IF EXISTS ] nom
    [ AS type_donnee ]
    [ INCREMENT [ BY ] incrément ]
    [ MINVALUE valeurmin | NO MINVALUE ] [ MAXVALUE valeurmax | NO
 MAXVALUE ]
    [ START [ WITH ] début ]
    [ RESTART [ [ WITH ] nouveau_début ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { nom_table.nom_colonne | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] nom SET { LOGGED | UNLOGGED }
ALTER SEQUENCE [ IF EXISTS ] nom OWNER TO { nouveau_propriétaire |
 CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] nom RENAME TO nouveau_nom
ALTER SEQUENCE [ IF EXISTS ] nom SET SCHEMA nouveau_schéma
```

Description

`ALTER SEQUENCE` modifie les paramètres d'un générateur de séquence. Tout paramètre non précisé dans la commande `ALTER SEQUENCE` conserve sa valeur précédente. Pour modifier le propriétaire, vous devez aussi être un membre direct ou indirect du nouveau rôle propriétaire, et ce rôle doit avoir le droit `CREATE` sur le schéma de la séquence (ces restrictions permettent de s'assurer que modifier le propriétaire ne fait rien de plus que ce que vous pourriez faire en supprimant puis recréant la séquence ; néanmoins un superutilisateur peut déjà modifier le propriétaire de toute séquence).

Seul le propriétaire de la séquence peut utiliser `ALTER SEQUENCE`. Pour modifier le schéma de la séquence, il faut posséder le droit `CREATE` sur le nouveau schéma.

Paramètres

nom

Le nom de la séquence à modifier (éventuellement qualifié du nom du schéma).

`IF EXISTS`

Ne retourne pas d'erreur si la séquence n'existe pas. Seul un message d'avertissement est retourné dans ce cas.

type_donnée

La clause facultative `AS` *type_donnee* change le type de données de la séquence. Les types valides sont `smallint`, `integer` et `bigint`.

Changer le type de donnée change automatiquement les valeurs minimales et maximales de la séquence si et seulement si les précédentes valeurs minimales et maximales étaient les valeurs minimales et maximales de l'ancien type de donnée (autrement dit, si la séquence avait été créée en utilisant `NO MINVALUE` ou `NO MAXVALUE`, de manière implicite ou explicite). Sinon les valeurs minimales et maximales sont préservées, à moins que de nouvelles valeurs soient spécifiées dans

la même commande. Si les nouvelles valeurs minimales et maximales ne rentrent pas dans le nouveau type de donnée, une erreur sera générée.

incrément

La clause INCREMENT BY *incrément* est optionnelle. Une valeur positive crée une séquence croissante, une valeur négative une séquence décroissante. Lorsque cette clause n'est pas spécifiée, la valeur de l'ancien incrément est conservée.

valeurmin
NO MINVALUE

La clause optionnelle MINVALUE *valeurmin*, détermine la valeur minimale de la séquence. Si NO MINVALUE est utilisé, les valeurs par défaut, 1 et la valeur minimale du type de donnée sont utilisées respectivement pour les séquences croissantes et décroissantes. Si aucune option n'est précisée, la valeur minimale courante est conservée.

valeurmax
NO MAXVALUE

La clause optionnelle MAXVALUE *valeurmax* détermine la valeur maximale de la séquence. Si NO MAXVALUE est utilisé, la valeur maximale du type de données et -1 sont utilisées respectivement pour les séquences croissantes et décroissantes comme valeurs par défaut. Si aucune option n'est précisée, la valeur maximale courante est conservée.

début

La clause optionnelle START WITH *début* modifie la valeur de départ enregistré pour la séquence. Cela n'a pas d'effet sur la valeur *actuelle* de celle-ci ; cela configure la valeur que les prochaines commandes ALTER SEQUENCE RESTART utiliseront.

restart

La clause optionnelle RESTART [WITH *restart*] modifie la valeur actuelle de la séquence. C'est équivalent à l'appel de la fonction setval avec is_called = false : la valeur spécifiée sera renvoyée par le *prochain* appel à nextval. Écrire RESTART sans valeur pour *restart* est équivalent à fournir la valeur de début enregistrée par CREATE SEQUENCE ou par ALTER SEQUENCE START WITH.

En contraste avec un appel à setval, une opération RESTART sur une séquence est transactionnelle et empêche les transactions concurrentes d'obtenir des nombres de la même séquence. Si ce n'est pas le mode désiré, setval doit être utilisé.

cache

La clause CACHE *cache* active la préallocation des numéros de séquences et leur stockage en mémoire pour en accélerer l'accès. 1 est la valeur minimale (une seule valeur est engendrée à la fois, donc pas de cache). Lorsque la clause n'est pas spécifiée, l'ancienne valeur est conservée.

CYCLE

Le mot clé optionnel CYCLE est utilisé pour autoriser la séquence à boucler lorsque *valeurmax* ou *valeurmin* est atteint par, respectivement, une séquence croissante ou décroissante. Lorsque la limite est atteinte, le prochain numéro engendré est, respectivement, *valeurmin* ou *valeurmax*.

NO CYCLE

Si le mot clé optionnel NO CYCLE est spécifié, tout appel à nextval alors que la séquence a atteint sa valeur maximale, dans le cas d'une séquence croissante, ou sa valeur minimale dans le

cas contraire, retourne une erreur. Lorsque ni `CYCLE` ni `NO CYCLE` ne sont spécifiés, l'ancien comportement est préservé.

`SET { LOGGED | UNLOGGED }`

Cette clause change la séquence de non journalisée à journalisée et vice-versa (voir CREATE SEQUENCE). Elle ne peut pas être utilisée avec une séquence temporaire.

`OWNED BY` *`nom_table.nom_colonne`*
`OWNED BY NONE`

L'option `OWNED BY` permet d'associer la séquence à une colonne spécifique d'une table pour que cette séquence soit supprimée automatiquement si la colonne (ou la table complète) est supprimée. Si cette option est spécifiée, cette association remplacera toute ancienne association de cette séquence. La table indiquée doit avoir le même propriétaire et être dans le même schéma que la séquence. Indiquer `OWNED BY NONE` supprime toute association existante, rendant à la séquence son « autonomie ».

`nouveau_propriétaire`

Le nom utilisateur du nouveau propriétaire de la séquence.

`nouveau_nom`

Le nouveau nom de la séquence.

`nouveau_schema`

Le nouveau schéma de la séquence.

Notes

`ALTER SEQUENCE` n'affecte pas immédiatement les résultats de `nextval` pour les sessions, à l'exception de la session courante, qui ont préalloué (placé en cache) des valeurs de la séquence. Elles épuisent les valeurs en cache avant de prendre en compte les modifications sur les paramètres de génération de la séquence. La session à l'origine de la commande est, quant à elle, immédiatement affectée.

`ALTER SEQUENCE` ne modifie pas le statut `currval` d'une séquence (avant PostgreSQL 8.3, c'était le cas quelque fois).

`ALTER SEQUENCE` bloque les appels concurrents à `nextval`, `currval`, `lastval`, et `setval`.

Pour des raisons historiques, `ALTER TABLE` peut aussi être utilisé avec les séquences, mais seules les variantes d'`ALTER TABLE` autorisées pour les séquences sont équivalentes aux formes affichées ci-dessus.

Exemples

Redémarrez la séquence `serial` à 105 :

```
ALTER SEQUENCE serial RESTART WITH 105;
```

Compatibilité

`ALTER SEQUENCE` est conforme au standard SQL, à l'exception des variantes `AS`, `START WITH`, `OWNED BY`, `OWNER TO`, `RENAME TO` et `SET SCHEMA` qui sont une extension PostgreSQL.

Voir aussi

CREATE SEQUENCE, DROP SEQUENCE

ALTER SERVER

ALTER SERVER — modifier la définition d'un serveur distant

Synopsis

```
ALTER SERVER nom [ VERSION 'nouvelle_version' ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ] ) ]
ALTER SERVER nom OWNER TO { nouveau_propriétaire | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER }
ALTER SERVER nom RENAME TO nouveau_nom
```

Description

`ALTER SERVER` modifie la définition d'un serveur distant. La première forme modifie la chaîne de version du serveur ou les options génériques du serveur (au moins une clause est nécessaire). La seconde forme modifie le propriétaire du serveur.

Pour modifier le serveur, vous devez être le propriétaire du serveur. De plus, pour modifier le propriétaire, vous devez posséder le serveur ainsi qu'être un membre direct ou indirect du nouveau rôle, et vous devez avoir le droit `USAGE` sur le wrapper de données distantes du serveur. (Notez que les superutilisateurs satisfont à tous ces critères automatiquement.)

Paramètres

nom

Le nom d'un serveur existant.

nouvelle_version

Nouvelle version du serveur.

`OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ] )`

Modifie des options pour le serveur. `ADD`, `SET` et `DROP` spécifient les actions à exécuter. Si aucune opération n'est spécifiée explicitement, l'action est `ADD`. Les noms d'options doivent être uniques ; les noms et valeurs sont aussi validés en utilisant la bibliothèque de wrapper de données distantes.

nouveau_propriétaire

Le nom du nouveau propriétaire du serveur distant.

nouveau_nom

Le nouveau nom du serveur distant.

Exemples

Modifier le serveur `foo` et lui ajouter des options de connexion :

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'dbfoo');
```

Modifier le serveur `foo`, modifier sa version, modifier son option `host` :

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

Compatibilité

`ALTER SERVER` est conforme à ISO/IEC 9075-9 (SQL/MED). Les clauses `OWNER TO` et `RENAME TO` sont des extensions PostgreSQL.

Voir aussi

CREATE SERVER, DROP SERVER

ALTER STATISTICS

ALTER STATISTICS — Modifier la définition d'un objet statistique étendu

Synopsis

```
ALTER STATISTICS nom OWNER TO { nouveau_propriétaire | CURRENT_ROLE
 | CURRENT_USER | SESSION_USER }
ALTER STATISTICS nom RENAME TO nouveau_nom
ALTER STATISTICS nom SET SCHEMA nouveau_schéma
ALTER STATISTICS name SET STATISTICS nouvelle_cible
```

Description

`ALTER STATISTICS` change les paramètres d'un objet statistiques étendu existant. Tous les paramètres qui n'ont pas été spécifiquement positionnés dans la commande `ALTER STATISTICS` conservent leurs précédentes valeurs.

Vous devez être propriétaire de l'objet statistique pour pouvoir utiliser `ALTER STATISTICS`. Pour changer le schéma d'un objet statistiques, vous devez également avoir le droit `CREATE` sur le nouveau schéma. Pour modifier le propriétaire, vous devez également être un membre direct ou indirect du nouveau rôle propriétaire, et ce rôle doit avoir le droit `CREATE` sur le schéma d'un objet statistique. (Ces restrictions assurent que la modification du propriétaire ne fasse rien que vous ne pourriez faire en supprimant et recréant l'objet statistiques. Néanmoins, un superutilisateur peut de toute façon modifier le propriétaire de n'importe quel objet statistique.)

Paramètres

nom

Le nom (éventuellement qualifié du nom du schéma) de l'objet statistique devant être modifié.

nouveau_proprietaire

Le nom d'utilisateur du nouveau propriétaire de l'objet statistique.

nouveau_nom

Le nouveau nom de l'objet statistique.

nouveau_schema

Le nouveau schéma de l'objet statistique.

nouvelle_cible

La cible de la récupération de statistiques pour cet objet lors des prochaines opérations `ANALYZE`. La cible est contenue dans l'intervalle 0 à 10000 ; sinon, configurez la à -1 pour revenir à l'utilisation du maximum de la cible statistique des colonnes référencées, si elles sont configurées, ou de la cible statistique par défaut du système (default_statistics_target). Pour plus d'informations sur l'utilisation des statistiques par l'optimiseur de PostgreSQL, référez-vous à Section 14.2.

Compatibilité

Il n'y a pas de commande `ALTER STATISTICS` dans le standard SQL.

Voir aussi

CREATE STATISTICS, DROP STATISTICS

ALTER SUBSCRIPTION

ALTER SUBSCRIPTION — modifier la définition d'une souscription

Synopsis

```
ALTER SUBSCRIPTION nom CONNECTION 'conninfo'
ALTER SUBSCRIPTION nom SET PUBLICATION nom_publication [, ...]
 [ WITH ( option_publication [= valeur] [, ... ] ) ]
ALTER SUBSCRIPTION nom ADD PUBLICATION nom_publication [, ...]
 [ WITH ( option_publication [= valeur] [, ... ] ) ]
ALTER SUBSCRIPTION nom DROP PUBLICATION nom_publication [, ...]
 [ WITH ( option_publication [= valeur] [, ... ] ) ]
ALTER SUBSCRIPTION nom REFRESH PUBLICATION [ WITH
 ( option_rafraichissement [= value] [, ... ] ) ]
ALTER SUBSCRIPTION nom ENABLE
ALTER SUBSCRIPTION nom DISABLE
ALTER SUBSCRIPTION nom SET ( subscription_parameter [= valeur]
 [, ... ] )
ALTER SUBSCRIPTION nom SKIP ( option_ignore = valeur )
ALTER SUBSCRIPTION nom OWNER TO { nouveau_proprietaire |
 CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER SUBSCRIPTION nom RENAME TO nouveau_nom
```

Description

`ALTER SUBSCRIPTION` peut changer la plupart des propriétés d'une souscription pouvant être spécifiées dans CREATE SUBSCRIPTION.

Vous devez être le propriétaire de la souscription pour utiliser `ALTER SUBSCRIPTION`. Pour modifier le propriétaire, vous devez également être un membre direct ou indirect du nouveau rôle propriétaire. Le nouveau propriétaire doit être un superutilisateur. (Actuellement, tous les propriétaires de souscription doivent être superutilisateurs, donc les vérifications du propriétaire seront en fait contournées. Mais ceci pourrait changer dans le futur.)

Lors du rafraichissement d'une publication, nous supprimons les relations qui ne font plus partie de la publication, et nous supprimons aussi les slots de synchronisation de table s'il en existe. Il est nécessaire de supprimer ces slots pour que les ressources allouées à la souscription sur l'hôte distant soient rendues. Si à cause d'erreurs réseau ou autres, PostgreSQL est incapable de supprimer les slots, un message d'erreur sera renvoyé. Pour continuer, l'utilisateur doit soit tenter une nouvelle fois l'opération, soit dissocier le slot de la souscription et supprimer la souscription comme expliquée dans DROP SUBSCRIPTION.

Les commandes `ALTER SUBSCRIPTION ... REFRESH PUBLICATION` et `ALTER SUBSCRIPTION ...{SET|ADD|DROP} PUBLICATION ...` avec l'option `refresh` activée ne peuvent pas être exécutées dans un bloc de transaction. Ces ne commandes peuvent pas non plus être exécutées quand la souscription a l'option `two_phase` activée, sauf si `copy_data` vaut `false`. Voir la colonne `subtwophasestate` de `pg_subscription`.

Paramètres

nom

Le nom de la souscription dont la propriété doit être modifiée.

`CONNECTION '*conninfo*'`

Cette clause remplace la propriété de connexion positionnée à l'origine par CREATE SUBSCRIPTION. S'y référer pour plus d'informations.

`SET PUBLICATION *nom_publication*`
`ADD PUBLICATION *nom_publication*`
`DROP PUBLICATION *nom_publication*`

Ces syntaxes changent la liste des publications souscrites. `SET` remplace la liste entière des publications avec une nouvelle liste, `ADD` ajoute des publications supplémentaires à la liste de publications, et `DROP` supprime des publications de la liste des publications. Nous permettons aux publications inexistantes d'être indiquées dans les variantes `ADD` et `SET` pour que les utilisateurs puissent les ajouter après. Voir CREATE SUBSCRIPTION pour plus d'informations. Par défaut, cette commande agira aussi comme `REFRESH PUBLICATION`.

`option_publication` indique des options supplémentaires pour cette opération. Les options supportées sont :

`refresh` (`boolean`)

Si false, la commande n'essaiera pas de rafraichir des informations des tables. `REFRESH PUBLICATION` devrait alors être exécutée séparément. La valeur par défaut est `true`.

De plus, les options décrites sous `REFRESH PUBLICATION` peuvent être spécifiées pour contrôler l'opération implicite de rafraichissement.

`REFRESH PUBLICATION`

Récupère les informations de table manquante depuis la publication. Cela commencera la réplication des tables qui avaient été ajoutées en tant que souscriptions aux publications depuis `CREATE SUBSCRIPTION` ou depuis la dernière exécution de `REFRESH PUBLICATION`.

`option_rafraichissement` spécifie les options supplémentaires pour l'opération de rafraîchissement. Les options supportées sont :

`copy_data` (`boolean`)

Indique de copier les données pré-existantes dans la publication en cours de souscription quand la réplication démarre. La valeur par défaut est `true`.

Les tables précédemment souscrites ne sont pas copiées, même si la clause `WHERE` de filtre de lignes d'une table a depuis été modifiée.

`ENABLE`

Active une souscription précédemment désactivée, démarrant le worker de réplication logique à la fin de la transaction.

`DISABLE`

Désactive une souscription en cours d'exécution, arrêtant le worker de réplication logique à la fin de la transaction.

`SET ( *subscription_parameter* [= *valeur*] [, ... ] )`

Cette clause change les paramètres initialement positionnés par CREATE SUBSCRIPTION. S'y référer pour plus d'informations. Les paramètres modifiables sont `slot_name`, `synchronous_commit`, `binary`, `streaming` et `disable_on_error`.

`SKIP ( *skip_option* = *valeur* )`

Ignore l'application de tous les changements de la transaction distante. Si des données entrantes violent des contraintes, la réplication logique s'arrêtera jusqu'à ce que le problème soit résolu.

En utilisant la commande `ALTER SUBSCRIPTION ... SKIP`, le processus worker de réplication logique ignore tous les changements de modification de données dans la transaction. Cette option n'a pas d'effet sur les transactions qui sont déjà préparées en activant l'option `two_phase` sur le souscripteur. Après que le processus worker de la réplication logique ait ignoré avec succès la transaction ou ait terminé une transaction, le LSN, enregistré dans `pg_subscription.subskiplsn`) est effacé. Voir Section 31.5 pour les détails des conflits sur la réplication logique. Utiliser cette commande requiert l'attribut `SUPERUSER`.

`skip_option` précise les options pour cette opération. La seule option acceptée est :

`lsn` (`pg_lsn`)

Indique le LSN de fin de la transaction distante dont les changements doivent être ignorés par le processus worker de réplication logique. Le LSN de fin est le LSN pour lequel la transaction est soit validée soit préparée. Ignorer les sous-transactions individuelles n'est pas supportée. Configurer à `NONE` réinitialise le LSN.

`nouveau_proprietaire`

Le nom d'utilisateur du nouveau propriétaire de la souscription.

`nouveau_nom`

Le nouveau nom de la souscription.

Exemples

Changer la publicatio souscrites par une publication en `insert_only` :

```
ALTER SUBSCRIPTION mysub SET PUBLICATION insert_only;
```

Désactive (stoppe) la souscription :

```
ALTER SUBSCRIPTION mysub DISABLE;
```

Compatibilité

`ALTER SUBSCRIPTION` est une extension PostgreSQL au standard SQL.

Voir aussi

CREATE SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

ALTER SYSTEM

ALTER SYSTEM — Modifier un paramètre de configuration du serveur

Synopsis

```
ALTER SYSTEM SET paramètre_configuration { TO | = } { valeur
 [, ...] | DEFAULT }

ALTER SYSTEM RESET paramètre_configuration
ALTER SYSTEM RESET ALL
```

Description

`ALTER SYSTEM` est utilisé pour modifier les paramètres de configuration du serveur pour l'instance complète. Cette méthode peut être plus pratique que la méthode traditionnelle revenant à éditer manuellement le fichier `postgresql.conf`. `ALTER SYSTEM` écrit la valeur du paramètre indiqué dans le fichier `postgresql.auto.conf`, qui est lu en plus du fichier `postgresql.conf`. Configurer un paramètre à `DEFAULT`, ou utiliser la variante `RESET`, supprime le paramètre du fichier `postgresql.auto.conf`. Utilisez `RESET ALL` pour supprimer tous les paramètres configurés dans ce fichier.

Les nouvelles valeurs des paramètres configurés avec `ALTER SYSTEM` seront prises en compte après le prochain rechargement de la configuration ou le prochain redémarrage du serveur dans le cas des paramètres nécessitant un redémarrage. Un rechargement de la configuration du serveur peut se faire en appelant la fonction SQL `pg_reload_conf()`, en exécutant la commande `pg_ctl reload` ou en envoyant un signal SIGHUP au processus principal du serveur.

Seuls les superutilisateurs et les utilisateurs disposant du droit `ALTER SYSTEM` sur un paramètre peuvent le modifier en utilisant `ALTER SYSTEM`. De plus, comme cette commande agit directement sur le système de fichiers et ne peut pas être annulée, elle n'est pas autorisée dans un bloc de transaction et dans une fonction.

Paramètres

`paramètre_configuration`

Nom d'un paramètre configurable. Les paramètres disponibles sont documentés dans Chapitre 20.

`valeur`

Nouvelle valeur du paramètre. Les valeurs peuvent être spécifiées en tant que constantes de chaîne, identifiants, nombres ou listes de valeurs séparées par des virgules, suivant le paramètre. Les valeurs qui ne sont ni des nombres ni des identifiants valides doivent être placées entre guillemets. `DEFAULT` peut être utilisé pour supprimer le paramètre et sa valeur du fichier `postgresql.auto.conf`.

Pour certains des paramètres acceptant des listes, des valeurs entre guillemets peuvent produire une sortie avec des guillemets doubles pour conserver espaces blancs et virgules ; pour les autres, les guillemets doubles doivent être utilisés à l'intérieur de chaînes comprises entre des guillemets simples pour obtenir cet effet.

Notes

Cette commande ne peut pas être utilisée pour configurer le paramètre data_directory ainsi que les paramètres qui ne sont pas autorisés dans le fichier `postgresql.conf` (donc les options préconfigurées).

Voir Section 20.1 pour d'autres façons de configurer les paramètres.

Exemples

Configurer le paramètre `wal_level` :

```
ALTER SYSTEM SET wal_level = replica;
```

Annuler cette configuration et restaurer le paramétrage indiqué dans le fichier `postgresql.conf` :

```
ALTER SYSTEM RESET wal_level;
```

Compatibilité

La commande `ALTER SYSTEM` est une extension PostgreSQL.

Voir aussi

SET, SHOW

ALTER TABLE

ALTER TABLE — Modifier la définition d'une table

Synopsis

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] nom [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] nom [ * ]
    RENAME [ COLUMN ] nom_colonne TO nouveau_nom_colonne
ALTER TABLE [ IF EXISTS ] [ ONLY ] nom [ * ]
    RENAME CONSTRAINT nom_contrainte TO nouveau_nom_contrainte
ALTER TABLE [ IF EXISTS ] nom
    RENAME TO nouveau_nom
ALTER TABLE [ IF EXISTS ] nom
    SET SCHEMA nouveau_schéma
ALTER TABLE ALL IN TABLESPACE nom [ OWNED BY nom_rôle [, ... ] ]
    SET TABLESPACE nouveau_tablespace [ NOWAIT ]
ALTER TABLE [ IF EXISTS ] nom
    ATTACH PARTITION nouvelle_partition { FOR
 VALUES spec_limite_partition | DEFAULT }
ALTER TABLE [ IF EXISTS ] nom
    DETACH PARTITION nouvelle_partition [ CONCURRENTLY | FINALIZE ]

où action fait partie
de :

    ADD [ COLUMN ] [ IF NOT EXISTS ] nom_colonne type_donnée
 [ COLLATE collation ] [ contrainte_colonne [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] nom_colonne [ RESTRICT |
 CASCADE ]
    ALTER [ COLUMN ] nom_colonne [ SET DATA ] TYPE type_donnée
 [ COLLATE collation ] [ USING expression ]
    ALTER [ COLUMN ] nom_colonne SET DEFAULT expression
    ALTER [ COLUMN ] nom_colonne DROP DEFAULT
    ALTER [ COLUMN ] nom_colonne { SET | DROP } NOT NULL
    ALTER [ COLUMN ] nom_colonne DROP EXPRESSION [ IF EXISTS ]
    ALTER [ COLUMN ] nom_colonne ADD GENERATED { ALWAYS | BY
 DEFAULT } AS IDENTITY [ ( options_séquence ) ]
    ALTER [ COLUMN ] nom_colonne { SET GENERATED
 { ALWAYS | BY DEFAULT } | SET option_séquence | RESTART
 [ [ WITH ] valeur_redémarrage ] } [...]
    ALTER [ COLUMN ] nom_colonne DROP IDENTITY [ IF EXISTS ]
    ALTER [ COLUMN ] nom_colonne SET STATISTICS integer
    ALTER [ COLUMN ] nom_colonne SET ( option_attribut = valeur
 [, ... ] )
    ALTER [ COLUMN ] nom_colonne RESET ( option_attribut [, ... ] )
    ALTER [ COLUMN ] nom_colonne SET STORAGE { PLAIN | EXTERNAL |
 EXTENDED | MAIN }
    ALTER [ COLUMN ] nom_colonne SET
 COMPRESSION méthode_compression
    ADD contrainte_table [ NOT VALID ]
    ADD contrainte_table_utilisant_index
    ALTER CONSTRAINT nom_contrainte [ DEFERRABLE | NOT DEFERRABLE ]
 [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
    VALIDATE CONSTRAINT nom_contrainte
```

```
    DROP CONSTRAINT [ IF EXISTS ]  nom_contrainte [ RESTRICT |
 CASCADE ]
    DISABLE TRIGGER [ nom_trigger | ALL | USER ]
    ENABLE TRIGGER [ nom_trigger | ALL | USER ]
    ENABLE REPLICA TRIGGER nom_trigger
    ENABLE ALWAYS TRIGGER nom_trigger
    DISABLE RULE nom_règle_réécriture
    ENABLE RULE nom_règle_réécriture
    ENABLE REPLICA RULE nom_règle_réécriture
    ENABLE ALWAYS RULE nom_règle_réécriture
    DISABLE ROW LEVEL SECURITY
    ENABLE ROW LEVEL SECURITY
    FORCE ROW LEVEL SECURITY
    NO FORCE ROW LEVEL SECURITY
    CLUSTER ON nom_index
    SET WITHOUT CLUSTER
    SET WITHOUT OIDS
    SET ACCESS METHOD nouvelle_méthode_accès
    SET TABLESPACE nouveau_tablespace
    SET { LOGGED | UNLOGGED }
    SET ( paramètre_stockage [= valeur] [, ... ] )
    RESET ( paramètre_stockage [, ... ] )
    INHERIT table_parent
    NO INHERIT table_parent
    OF nom_type
    NOT OF
    OWNER TO { nouveau_propriétaire | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
    REPLICA IDENTITY { DEFAULT | USING INDEX nom_index | FULL |
 NOTHING }

et spec_limite_partition
vaut :

IN ( expr_limite_partition [, ...] ) |
FROM ( { expr_limite_partition | MINVALUE | MAXVALUE } [, ...] )
  TO ( { expr_limite_partition | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS litéral_numérique, REMAINDER litéral_numérique )

et contrainte_colonne
vaut :

[ CONSTRAINT nom_contrainte ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT expression_par_défaut |
  GENERATED ALWAYS AS ( expr_génération ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
 [ ( options_séquence ) ] |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] paramètres_index |
  PRIMARY KEY paramètres_index |
  REFERENCES table_référencée [ ( colonne_référencée ) ] [ MATCH
 FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action_referentielle ] [ ON
 UPDATE action_referentielle ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
 IMMEDIATE ]
```

```
et contrainte_table
vaut :

[ CONSTRAINT nom_contrainte ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] ( nom_colonne
 [, ... ] ) index_parameters |
  PRIMARY KEY ( nom_colonne [, ... ] ) paramètres_index |
  EXCLUDE [ USING méthode_index ] ( élément_exclus WITH opérateur
 [, ... ] ) paramètres_index [ WHERE ( prédicat ) ] |
  FOREIGN KEY ( nom_colonne [, ... ] ) REFERENCES table_référencée
 [ ( colonne_référencée [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON
 DELETE action_referentielle ] [ ON UPDATE action_referentielle ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
 IMMEDIATE ]

et contrainte_table_utilisant_index vaut :

    [ CONSTRAINT nom_contrainte ]
    { UNIQUE | PRIMARY KEY } USING INDEX nom_index
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED |
 INITIALLY IMMEDIATE ]

paramètres_index dans les
contraintes UNIQUE, PRIMARY KEY
 et
EXCLUDE valent :

[ INCLUDE ( nom_colonne [, ... ] ) ]
[ WITH ( paramètre_stockage [= valeur] [, ... ] ) ]
[ USING INDEX TABLESPACE nom_tablespace ]

élément_exclus dans une
contrainte EXCLUDE vaut :

{ nom_colonne | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS
 { FIRST | LAST } ]
{ nom_colonne | ( expression ) } [ COLLATE collation ] [ opclass
 [ ( parametre_opclass = valeur [, ... ] ) ] ] [ ASC | DESC ]
 [ NULLS { FIRST | LAST } ]

referential_action dans une
contrainte FOREIGN
KEY/REFERENCES vaut :

{ NO ACTION | RESTRICT | CASCADE | SET NULL [ ( nom_colonne
 [, ... ] ) ] | SET DEFAULT [ ( nom_colonne [, ... ] ) ] }
```

Description

ALTER TABLE modifie la définition d'une table existante. Il existe plusieurs variantes décrites après. Il est à noter que le niveau de verrouillage requis peut changer pour chaque variante. Un verrou ACCESS EXCLUSIVE est utilisé à moins que le verrou ne soit explicitement noté. Quand de multiples sous-commandes sont listées, le verrou utilisé sera celui le plus strict requis pour l'ensemble des sous-commandes.

`ADD [ COLUMN ] [ IF NOT EXISTS ]`

Ajoute une nouvelle colonne à la table en utilisant une syntaxe identique à celle de `CREATE TABLE`. Si `IF NOT EXISTS` est précisée et qu'une colonne existe déjà avec ce nom, aucune erreur n'est renvoyée.

`DROP [ COLUMN ] [ IF EXISTS ]`

Supprime une colonne de la table. Les index et les contraintes de table référençant cette colonne sont automatiquement supprimés. Les statistiques multivarées référençant les colonnes supprimées seront également supprimées si la suppression de la colonne avait pour effet de réduire le nombre de colonnes dans la statistique à 1. L'option `CASCADE` doit être utilisée lorsque des objets en dehors de la table dépendent de cette colonne, comme par exemple des références de clés étrangères ou des vues. Si `IF EXISTS` est indiqué et que la colonne n'existe pas, aucune erreur n'est renvoyée. Dans ce cas, un message d'avertissement est envoyé à la place.

`SET DATA TYPE`

Change le type d'une colonne de la table. Les index et les contraintes simples de table qui impliquent la colonne sont automatiquement convertis pour utiliser le nouveau type de la colonne en ré-analysant l'expression d'origine. La clause optionnelle `COLLATE` spécifie une collation pour la nouvelle colonne. Si elle est omise, la collation utilisée est la collation par défaut pour le nouveau type de la colonne. La clause optionnelle `USING` précise comment calculer la nouvelle valeur de la colonne à partir de l'ancienne ; en cas d'omission, la conversion par défaut est identique à une affectation de transtypage de l'ancien type vers le nouveau. Une clause `USING` doit être fournie s'il n'existe pas de conversion implicite ou d'assignement entre les deux types.

Quand cette clause est utilisée, les statistiques de la colonne sont supprimées donc exécuter un `ANALYZE` sur cette table est recommandé après avoir utilisé cette clause.

`SET/DROP DEFAULT`

Ajoute ou supprime les valeurs par défaut d'une colonne (où la suppression est équivalente à ajouter une valeur par défaut à NULL). La nouvelle valeur par défaut s'appliquera uniquement aux prochaines requêtes `INSERT` ou `UPDATE`. Cela ne concerne donc pas les lignes déjà présentes dans la table.

`SET/DROP NOT NULL`

Modifie l'autorisation de valeurs NULL.

`SET NOT NULL` ne peut être utilisé que sur une colonne ne comportant aucune valeur `NULL`. Normalement, cela entraîne une vérification par un parcours complet de la table ; cependant, si une contrainte `CHECK` valide existe prouvant l'impossibilité d'avoir de donnée `NULL` dans cette colonne, alors le parcours de table n'est pas effectué.

Si cette table est une partition, il n'est pas possible d'effectuer de `DROP NOT NULL` sur une colonne qui est marquée `NOT NULL` dans la table parente. Pour supprimer la contrainte `NOT NULL` de toutes les partitions, effectuez un `DROP NOT NULL` sur la table parente. Même s'il n'y a pas de contrainte `NOT NULL` sur la table parente, une telle contrainte peut quand même être ajoutée à des partitions individuelles, si l'on veut ; ainsi, les enfants peuvent refuser les valeurs NULL même si le parent les autorise, mais l'inverse n'est pas possible.

`DROP EXPRESSION [ IF EXISTS ]`

Transforme une colonne générée stockée en une colonne de base. Les données existantes dans la colonne sont conservées mais les modifications futures n'appliqueront plus l'expression de génération.

Si `DROP EXPRESSION IF EXISTS` est spécifié et que la colonne n'est pas une colonne générée stockée, aucune erreur ne sera remontée. Dans ce cas, un message de notification sera émis à la place.

```
ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
SET GENERATED { ALWAYS | BY DEFAULT }
DROP IDENTITY [ IF EXISTS ]
```

Modifier une colonne en colonne d'identité ou changer les attributs de génération d'une colonne d'identité existante. Voir `CREATE TABLE` pour plus de détails. Comme pour le `SET DEFAULT`, cette modification s'appliquera uniquement aux prochaines requêtes `INSERT` et `UPDATE`. Cela ne concerne donc pas les lignes déjà présentes dans la table.

Si `DROP IDENTITY IF EXISTS` est spécifié et que la colonne n'est pas une colonne d'identité, aucune erreur n'est remontée. Dans ce cas, un message d'avertissement est affiché à la place.

`SET` *`option_séquence`*
`valeur_redémarrage`

Modifie la séquence associée à une colonne d'identité existante. *`option_sequence`* est une options supportée par `ALTER SEQUENCE` tout comme `INCREMENT BY`.

`SET STATISTICS`

Permet de modifier l'objectif de collecte de statistiques par colonne pour les opérations d'analyse (`ANALYZE`) ultérieures. L'objectif prend une valeur entre 0 et 10000. Il est positionné à -1 pour utiliser l'objectif de statistiques par défaut du système (default_statistics_target). Pour plus d'informations sur l'utilisation des statistiques par le planificateur de requêtes de PostgreSQL, voir Section 14.2.

`SET STATISTICS` acquiert un verrou `SHARE UPDATE EXCLUSIVE`.

`SET ( `*`option_attribut = valeur`*` [, ... ] )`
`RESET ( `*`option_attribut`*` [, ... ] )`

Cette syntaxe permet de configurer ou de réinitialiser des propriétés. Actuellement, les seules propriétés acceptées sont `n_distinct` et `n_distinct_inherited`, qui surchargent l'estimation du nombre de valeurs distinctes calculé par `ANALYZE`. `n_distinct` affecte les statistiques de la table elle-même alors que `n_distinct_inherited` affecte les statistiques récupérées pour la table et les tables en héritant, ainsi que pour les statistiques récupérées sur les tables partitionnées. Si la valeur indiquée est une valeur positive, l'optimiseur de requêtes supposera que la colonne contient exactement le nombre spécifié de valeurs distinctes non NULL. Des valeurs non entières peuvent aussi être indiquées en utilisant des valeurs entre 0 et une valeur supérieure ou égale à -1. Ceci indique à l'optimiseur que l'estimation du nombre de valeurs distinctes se fait en multipliant la valeur absolue du nombre indique par le nombre estimé de lignes dans la table. Par exemple, une valeur de -1 indique que toutes les valeurs de la colonne sont distinctes alors qu'une valeur de -0,5 indique que chaque valeur apparaît deux fois en moyenne. Ceci peut être utile quand la taille d'une table change dans le temps. Pour plus d'informations sur l'utilisation des statistiques par l'optimiseur de requêtes de PostgreSQL, référez-vous à Section 14.2.

Changer les options d'une propriété nécessite un verrou `SHARE UPDATE EXCLUSIVE`.

`SET STORAGE`
`SET STORAGE`

Modifie le mode de stockage pour une colonne. Cela permet de contrôler si cette colonne est conservée en ligne ou dans une deuxième table, appelée table TOAST, et si les données sont ou non compressées. `PLAIN`, en ligne, non compressé, est utilisé pour les valeurs de longueur fixe, comme les `integer`. `MAIN` convient pour les données en ligne, compressibles. `EXTERNAL` est fait pour les données externes non compressées, `EXTENDED` pour les données externes compressées. `EXTENDED` est la valeur par défaut pour la plupart des types qui supportent les stockages différents de `PLAIN`. L'utilisation d'`EXTERNAL` permet d'accélérer les opérations d'extraction de sous-chaînes sur les très grosses valeurs de types `text` et `bytea` mais utilise plus

d'espace de stockage. SET STORAGE ne modifie rien dans la table, il configure la stratégie à poursuivre lors des mises à jour de tables suivantes. Voir Section 73.2 pour plus d'informations.

Bien que la plupart des formes de ADD *contrainte_table* nécessite un verrou ACCESS EXCLUSIVE, ADD FOREIGN KEY nécessite seulement un verrou SHARE ROW EXCLUSIVE. Notez que ADD FOREIGN KEY nécessite aussi un verrou SHARE ROW EXCLUSIVE sur la table référencée, en plus du verrou sur la table où la contrainte est déclarée.

SET COMPRESSION *méthode_compression*

Cette clause configure la méthode de compression pour une colonne, déterminant comment les valeurs insérées dans le futur seront compressées (si le mode de stockage permet la compression). Ceci ne cause pas la réécriture de la table, donc les données existantes pourraient toujours être compressées avec les autres méthodes de compression. Si la table est restaurée avec pg_restore, alors toutes les valeurs sont réécrites avec la méthode de compression configurée. Néanmoins, quand les données sont insérées à partir d'une autre relation (par exemple par INSERT ... SELECT), les valeurs de la table source ne sont pas forcément détoastées, donc toute donnée précédemment compressée pourrait conserver sa méthode de compression existante, plutôt que d'être recompressée avec la méthode de compression de la colonne cible. Les méthodes de compression supportées sont pglz et lz4. (lz4 est disponible seulement si --with-lz4 a été utilisé lors de la compilation de PostgreSQL.) De plus, *méthode_compression* peut valoir default, ce qui sélectionne le comportement par défaut consistant à lire la valeur du paramètre default_toast_compression lors de l'insertion de données pour déterminer la méthode à utiliser.

ADD *contrainte_table* [NOT VALID]

Ajoute une nouvelle contrainte à une table en utilisant une syntaxe identique à CREATE TABLE, plus l'option NOT VALID, qui est actuellement seulement autorisée pour les contraintes de type clé étrangère et les contraintes CHECK.

Normalement, cette clause entraîne un parcours complet de la table pour vérifier que les lignes pré-existantes satisfont la nouvelle contrainte. Mais si l'option NOT VALID est utilisée, cette opération potentiellement longue n'est pas effectuée. Cette contrainte sera néanmoins toujours vérifiée pour les nouvelles insertions ou modifications de données (c'est-à-dire que ces ordres échoueront s'il n'y a pas de ligne correspondante dans la table de référence pour une clé étrangère ou si la nouvelle ligne ne satisfait pas la condition d'une contrainte de vérification). La base de données ne considérera pas la contrainte comme entièrement vérifiée pour toutes les lignes tant qu'une validation effectuée par l'ordre VALIDATE CONSTRAINT n'aura pas été effectuée. Voir Notes ci-dessous pour plus d'informations sur l'utilisation de l'option NOT VALID.

L'ajout d'une contrainte de clé étrangère requiert un verrou SHARE ROW EXCLUSIVE sur la table référencée, en plus du verrou sur la table bénéficiant de la nouvelle contrainte.

Des restrictions supplémentaires s'appliquent quand des contraintes d'unicité ou des clés primaires sont ajoutées à des tables partitionnées. Voir CREATE TABLE. De plus, les contraintes de clés étrangères sur les tables partitionnées ne peuvent bénéficier de l'option NOT VALID, pour l'instant.

ADD *contrainte_table_utilisant_index*

Cette forme ajoute une nouvelle contrainte PRIMARY KEY ou UNIQUE sur une table, basée sur un index unique existant auparavant. Toutes les colonnes de l'index sont incluses dans la contrainte.

Cet index ne peut pas être un index partiel, ni être sur des expressions de colonnes. De plus, il doit être un index b-tree avec un ordre de tri par défaut. Ces restrictions assurent que cet index soit équivalent à un index qui aurait été créé par une commande standard ADD PRIMARY KEY ou ADD UNIQUE.

Si vous précisez PRIMARY KEY, et que les colonnes de l'index ne sont pas déjà spécifiées comme NOT NULL, alors la commande va tenter d'appliquer la commande ALTER COLUMN SET NOT

NULL sur chacune de ces colonnes. Cela nécessite un parcours complet de la table pour vérifier que la ou les colonne(s) ne contiennent pas de NULL. Dans tous les autres cas, c'est une opération rapide.

Si un nom de contrainte est fourni, alors l'index sera renommé afin de correspondre au nom de la contrainte. Sinon la contrainte sera nommée comme l'index.

Une fois que la commande est exécutée, l'index est « possédé » par la contrainte, comme si l'index avait été construit par une commande ADD PRIMARY KEY ou ADD UNIQUE ordinaire. En particulier, supprimer la contrainte fait également disparaître l'index.

Cette syntaxe n'est actuellement pas supportée sur les tables partitionnées.

> **Note**
>
> Ajouter une contrainte en utilisant un index existant peut être utile dans les situations où il faut ajouter une nouvelle contrainte, sans bloquer les mises à jour de table trop longtemps. Pour faire cela, créez l'index avec CREATE INDEX CONCURRENTLY, puis installez-la en tant que contrainte officielle en utilisant cette syntaxe. Voir l'exemple ci-dessous.

ALTER CONSTRAINT

Cette forme modifie les propriétés d'une contrainte précédemment créée. Pour le moment, seules les contraintes de clés étrangères peuvent être modifiées.

VALIDATE CONSTRAINT

Cette forme valide une contrainte de type clé étrangère ou une contrainte CHECK qui a été précédemment créée avec la clause NOT VALID. Elle le fait en parcourant la table pour s'assurer qu'il n'existe aucune ligne pour laquelle la contrainte n'est pas satisfaite. Si la contrainte est déjà marquée valide, cette clause ne fait rien. (Voir Notes ci-dessous pour une explication sur l'utilisation de cette commande.)

Cette commande récupère un verrou de type SHARE UPDATE EXCLUSIVE.

DROP CONSTRAINT [IF EXISTS]

Supprime la contrainte de table précisée, ainsi que tout index sous-jacent de la contrainte. Si IF EXISTS est précisé et que la contrainte n'existe pas, aucune erreur n'est renvoyée. Par contre, un message d'avertissement est lancé.

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

Configure l'exécution des triggers définis sur la table. Un trigger désactivé est toujours connu par le système mais n'est plus exécuté lorsque l'événement trigger survient. Pour un trigger retardé, le statut d'activité est vérifié au moment où survient l'événement, et non quand la fonction du trigger est réellement exécutée. Il est possible de désactiver ou d'activer un trigger spécifique (précisé par son nom), tous les triggers d'une table ou seulement les triggers utilisateur de cette table (cette option exclut les triggers générés en interne pour gérer les contraintes comme ceux utilisés pour implanter les contraintes de clés étrangères ou les contraintes déferrés uniques ou d'exclusion). Désactiver ou activer les triggers implicites de contraintes requiert des droits de superutilisateur ; cela doit se faire avec précaution car l'intégrité de la contrainte ne peut pas être garantie si les triggers ne sont pas exécutés.

Le mécanisme de déclenchement des triggers est aussi affecté par la variable de configuration session_replication_role. Les triggers activés (ENABLE, par défaut) se déclencheront quand le rôle de réplication est « origin » (la valeur par défaut) ou « local ». Les triggers configurés ENABLE

REPLICA se déclencheront seulement si la session est en mode « replica » et les triggers ENABLE ALWAYS se déclencheront à chaque fois, quelque soit le rôle de réplication.

L'effet de ce mécanisme est que, dans la configuration par défaut, les triggers ne se déclenchent pas sur les replicas. Ceci est utile parce que si un trigger est utilisé sur l'origine pour propager des données entre des tables, alors le système de réplication va aussi répliquer les données propagées, et le trigger ne devrait pas être exécuté une deuxième fois sur le serveur secondaire car cela amènerait à une duplication. Néanmoins, si un trigger est utilisé pour une autre raison comme la création d'alertes externes, il pourrait être approprié de le configurer à ENABLE ALWAYS pour qu'il puisse être exécuté sur les serveurs secondaires.

Quand cette commande s'applique à une table partitionnée, les états des triggers clones correspondant dans les partitions sont aussi mis à jour, sauf si ONLY est indiqué.

Cette commande acquiert un verrou SHARE ROW EXCLUSIVE.

DISABLE/ENABLE [REPLICA | ALWAYS] RULE

Ces formes configurent le déclenchement des règles de réécriture appartenant à la table. Une règle désactivée est toujours connue par le système mais non appliquée lors de la réécriture de la requête. La sémantique est identique celles des triggers activés/désactivés. Cette configuration est ignorée pour les règles ON SELECT qui sont toujours appliqués pour conserver le bon fonctionnement des vues même si la session actuelle n'est pas dans le rôle de réplication par défaut.

Le mécanisme d'exécution d'une règle est aussi affecté par la variable de configuration session_replication_role, de façon identique aux triggers comme décrit ci-dessus.

DISABLE/ENABLE ROW LEVEL SECURITY

Ces clauses contrôlent l'application des politiques de sécurité de lignes appartenant à la table. Si activé et qu'aucune politique n'existe pour la table, alors une politique de refus est appliqué par défaut. Notez que les politiques peuvent exister pour une table même si la sécurité niveau ligne est désactivée. Dans ce cas, les politiques ne seront *pas* appliquées, elles seront ignorées. Voir aussi CREATE POLICY.

NO FORCE/FORCE ROW LEVEL SECURITY

Ces clauses contrôlent l'application des politiques de sécurité niveau ligne appartenant à la table quand l'utilisateur est le propriétaire de la table. Si activé, les politiques de sécurité au niveau ligne seront appliquées quand l'utilisateur est le propriétaire de la table. S'il est désactivé (ce qui est la configuration par défaut), alors la sécurité niveau ligne ne sera pas appliquée quand l'utilisateur est le propriétaire de la table. Voir aussi CREATE POLICY.

CLUSTER ON

Sélectionne l'index par défaut pour les prochaines opérations CLUSTER. La table n'est pas réorganisée.

Changer les options de cluster nécessite un verrou SHARE UPDATE EXCLUSIVE.

SET WITHOUT CLUSTER

Supprime de la table la spécification d'index CLUSTER la plus récemment utilisée. Cela agit sur les opérations de réorganisation suivantes qui ne spécifient pas d'index.

Changer les options de cluster nécessite un verrou SHARE UPDATE EXCLUSIVE.

SET WITHOUT OIDS

Cette syntaxe permet de supprimer les colonnes système oid. Comme les colonnes système oid ne peuvent plus être ajoutées, cette clause n'a aucun effet.

`SET ACCESS METHOD`

Cette clause modifie la méthode d'accès à la table en la réécrivant. Voir Chapitre 63 pour plus d'informations.

`SET TABLESPACE`

Cette clause remplace le tablespace de la table par le tablespace indiqué, et déplace les fichiers de données associés à la table vers le nouveau tablespace. Les index de la table, s'il y en a, ne sont pas déplacés mais ils peuvent l'être avec des commandes `SET TABLESPACE` séparées. Quand cette clause est appliquée à une table partitionnée, aucune partition pré-existante n'est déplacée. Cependant, toute partition créée ensuite avec l'ordre `CREATE TABLE PARTITION OF` utilisera ce tablespace, à moins qu'une autre clause `TABLESPACE` ne soit utilisée pour la remplacer.

Toutes les tables de la base de donnée d'un tablespace peuvent être déplacées en utilisant la clause `ALL IN TABLESPACE`, ce qui verrouillera toutes les tables pour les déplacer une par une. Cette clause supporte aussi `OWNED BY`, qui déplacera seulement les tables appartenant aux rôles spécifiées. Si l'option `NOWAIT` est précisée, alors la commande échouera si elle est incapable d'acquérir tous les verrous requis immédiatement. Notez que les catalogues systèmes ne sont pas déplacés par cette commande, donc utilisez `ALTER DATABASE` ou des appels explicites à `ALTER TABLE` si désiré. Les tables du schéma `information_schema` ne sont pas considérées comme faisant partie des catalogues systèmes et seront donc déplacées. Voir aussi `CREATE TABLESPACE`.

`SET { LOGGED | UNLOGGED }`

Cette clause modifie le statut journalisé/non journalisé d'une table (voir `UNLOGGED`). Cela ne peut pas s'appliquer à une table temporaire.

Ceci modifie aussi la persistence de toute séquence liée à la table (pour des colonnes d'identité ou des colonnes de type serial). Néanmoins, il est aussi possible de modifier la persistence de telles séquences séparément.

`SET ( `*`paramètre_stockage`*` [= `*`valeur`*`] [, ... ] )`

Cette forme modifie un ou plusieurs paramètres de stockage pour la table. Voir Paramètres de stockage dans la documentation de `CREATE TABLE` pour les détails sur les paramètres disponibles. Le contenu de la table ne sera pas modifié immédiatement par cette commande ; en fonction du paramètre, il pourra s'avérer nécessaire de réécrire la table pour obtenir les effets désirés. Ceci peut se faire avec `VACUUM FULL`, `CLUSTER` ou une des formes d'`ALTER TABLE` qui force une réécriture de la table. Pour les paramètres liés à l'optimiseur, les changements prendront effet à partir de la prochaine fois que la table est verrouillée, donc les requêtes en cours d'exécution ne seront pas affectées.

Un verrou de type `SHARE UPDATE EXCLUSIVE` sera acquis pour les paramètre de stockage fillfactor, toast et autovacuum, ainsi que le paramètre lié à l'optimiseur `parallel_workers`.

`RESET ( `*`paramètre_stockage`*` [, ... ] )`

Cette forme réinitialise un ou plusieurs paramètres de stockage à leur valeurs par défaut. Comme avec `SET`, une réécriture de table pourrait être nécessaire pour mettre à jour entièrement la table.

`INHERIT `*`table_parent`*

Cette forme ajoute la table cible comme nouvel enfant à la table parent indiquée. En conséquence, les requêtes concernant le parent ajouteront les enregistrements de la table cible. Pour être ajoutée en tant qu'enfant, la table cible doit déjà contenir toutes les colonnes de la table parent (elle peut avoir des colonnes supplémentaires). Les colonnes doivent avoir des types qui correspondent, et s'il y a des contraintes `NOT NULL` définies pour le parent, alors elles doivent aussi avoir les contraintes `NOT NULL` pour l'enfant.

Il doit y avoir aussi une correspondance des contraintes de tables enfants pour toutes les contraintes CHECK, sauf pour celles qui ont été définies comme non héritables (c'est-à-dire créées avec l'option ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT) par la table parente, qui sont donc ignorées. Les contraintes des tables filles en correspondance avec celles de la table parente ne doivent pas être définies comme non héritables. Actuellement, les contraintes UNIQUE, PRIMARY KEY et FOREIGN KEY ne sont pas prises en compte mais ceci pourrait changer dans le futur.

NO INHERIT *table_parent*

Cette forme supprime une table cible de la liste des enfants de la table parent indiquée. Les requêtes envers la table parent n'incluront plus les enregistrements de la table cible.

OF *nom_type*

Cette forme lie la table à un type composite comme si la commande CREATE TABLE OF l'avait créée. La liste des noms de colonnes et leurs types doivent correspondre précisément à ceux du type composite. La table ne doit pas hériter d'une autre table. Ces restrictions garantissent que la commande CREATE TABLE OF pourrait permettre la définition d'une table équivalente.

NOT OF

Cette forme dissocie une table typée de son type.

OWNER

Change le propriétaire d'une table, d'une séquence, d'une vue, d'une vue matérialisée ou d'une table distante. Le nouveau propriétaire est celui passé en paramètre.

REPLICA IDENTITY

Cette forme change l'information écrite dans les journaux de transactions permettant d'identifier les lignes qui sont mises à jour ou supprimées. Dans la plupart des cas, l'ancienne valeur de chaque colonne ne sera enregistrée que si elle diffère de la nouvelle valeur ; néanmoins, si l'ancienne valeur est enregistrée extérieurement, elle est toujours tracée qu'elle soit modifiée ou pas. Cette option n'a pas d'effet quand la réplication logique est utilisée.

DEFAULT

Enregistre les anciennes valeurs de toutes les colonnes de la clé primaire, si elle existe. C'est la valeur par défaut pour les tables non systèmes.

USING INDEX *nom_index*

Enregistre les anciennes valeurs des colonnes couvertes par l'index nommé, qui doit être d'unicité, non partiel, non déferrable, et inclure seulement des colonnes marquées NOT NULL. Si cet index est supprimé, le comportement est identique à NOTHING.

FULL

Enregistre les anciennes valeurs de toutes les colonnes de la ligne.

NOTHING

N'enregistre aucune information sur l'ancienne ligne. C'est la valeur par défaut pour les tables systèmes.

RENAME

Change le nom d'une table (ou d'un index, d'une séquence, d'une vue, d'une vue matérialisée ou d'une table distante) ou le nom d'une colonne individuelle de la table ou le nom d'une contrainte

de la table. Lors du renommage d'une contrainte qui dispose d'un index sous-jacent, l'index est aussi renommé. Cela n'a aucun effet sur la donnée stockée.

`SET SCHEMA`

Déplace la table dans un autre schéma. Les index, les contraintes et les séquences utilisées dans les colonnes de table sont également déplacés.

`ATTACH PARTITION` *`nom_partition`* `{ FOR VALUES` *`spec_limite_partition`* `| DEFAULT }`

Attache une table existante (qui peut elle-même être partitionnée) comme une partition de la table cible. La table peut être attachée comme partition pour des valeurs spécifiques en utilisant `FOR VALUES` ou comme partition par défaut en utilisant `DEFAULT`. Pour chaque index de la table cible, si un index équivalent valide existe dans la partition, il sera attaché à l'index de la table cible, comme si ALTER INDEX ATTACH PARTITION avait été exécuté ; sinon un nouvel index correspondant sera créé. Les index invalides sur la partition sont ignorés. Notez que si la table existante est une table distante, il n'est actuellement pas autorisé d'attacher la table comme partition de la table cible s'il existe des index `UNIQUE` sur la table cible (voir aussi CREATE FOREIGN TABLE). Pour chaque trigger de niveau ligne défini par l'utilisateur existant dans la table cible, un trigger correspondant est créé dans la table attachée.

Une partition utilisant `FOR VALUES` utilise la même syntaxe pour *`spec_limite_partition`* que `CREATE TABLE`. La spécification de limite de partition doit correspondre à la stratégie de partitionnement et à la clé de partition de la table cible. La table qui doit être attachée doit avoir la totalité des colonnes identiques à la table cible et ne doit pas en avoir plus ; de plus, les types de colonnes doivent également correspondre. De plus, elle doit avoir toutes les contraintes `NOT NULL` et `CHECK` de la table cible, non marquées `NO INHERIT`. Pour le moment, les contraintes `FOREIGN KEY` ne sont pas considérées. Les contraintes `UNIQUE` et `PRIMARY KEY` de la table parent seront créées dans la partition si elles n'existent pas déjà.

Si la nouvelle partition est une table standard, un parcours complet de la table est effectué pour vérifier que les lignes existantes ne violent pas la contrainte de partition. Il est possible d'éviter ce parcours en ajoutant une contrainte `CHECK` valide à la table qui n'autoriserait que les lignes satisfaisant la contrainte de partition désirée avant de lancer cette commande. La contrainte `CHECK` sera utilisée pour déterminer si le parcours de la table est nécessaire pour valider la contrainte de partition. Cependant, cela ne fonctionne pas si l'une des clés de la partition est une expression et que la partition n'accepte pas de valeurs `NULL`. Si une partition de type liste qui n'accepte pas de valeurs `NULL` est attachée, ajoutez également une contrainte `NOT NULL` à la colonne de la clé de partition, à moins qu'il s'agisse d'une expression.

Si la nouvelle partition est une table distante, rien ne sera fait pour vérifier que toutes les lignes de la table distante obéissent à la contrainte de partition. (Voir la discussion dans CREATE FOREIGN TABLE sur les contraintes sur les tables distantes.)

Quand une table a une partition par défaut, définir une nouvelle partition modifie la contrainte de la partition par défaut. Cette dernière ne peut pas contenir de lignes qui devraient être déplacées dans la nouvelle partition. Ce cas sera vérifié. Ce parcours, tout comme le parcours de la nouvelle partition, peut être évité si une contrainte `CHECK` appropriée est présente. De plus, comme pour le parcours de la nouvelle partition, c'est toujours ignoré quand la partition par défaut est une table distante.

Attacher une partition nécessite l'acquisition d'un verrou `SHARE UPDATE EXCLUSIVE` sur la table parent, en plus des verrous `ACCESS EXCLUSIVE` sur la table à attacher et sur la partition par défaut (s'il y en a une).

D'autres verrous pourraient être détenus sur toutes les sous-partitions si la table en cours d'attachement est elle-même une table partitionnée. Pareil si la partition est elle-même une table partitionnée. Le verrouillage des sous-partitions peut être évité en ajoutant une contrainte `CHECK` comme décrit dans Section 5.11.2.2.

`DETACH PARTITION` *`nom_partition`* `[ CONCURRENTLY | FINALIZE ]`

Cette syntaxe détache la partition spécifiée de la table cible. La partition détachée continue d'exister comme une table standard, mais n'a plus aucun lien avec la table dont elle vient d'être détachée. Tout index attaché aux index de la table cible est détaché. Tous les triggers créés comme clones de ceux disponibles dans la table cible sont supprimés. Un verrou `SHARE` est obtenu sur toute table qui référence cette table partitionnée dans les contraintes de type clé étrangère.

Si `CONCURRENTLY` est précisé, cela fonctionne en utilisant un niveau de verrou réduit pour éviter de bloquer les autres sessions qui pourraient être en train d'accéder à la table partitionnée. Dans ce mode, deux transactions sont utilisées en interne. Lors de la première transaction, un verrou `SHARE UPDATE EXCLUSIVE` est pris à la fois sur la table parent et sur la partition, et la partition est marquée comme en cours de détachement ; à ce moment là, la transaction est validée et la commande est en attente de toutes les autres transactions utilisant la table partitionnée. Une fois que toutes ces transactions ont terminé, la deuxième transaction acquiert un verrou `SHARE UPDATE EXCLUSIVE` sur la table partitionnée et un verrou `ACCESS EXCLUSIVE` sur la partition, et le processus de détachement se termine. Une contrainte `CHECK` qui duplique la contrainte de partitionnement est ajoutée à la partition. `CONCURRENTLY` ne peut pas être utilisé dans un bloc de transaction et n'est pas autorisé si la table partitionnée contient une partition par défaut.

Si `FINALIZE` est indiqué, un appel précédent à `DETACH CONCURRENTLY` qui a été annulé ou interrompu est terminé. Au plus une partition dans une table partitionnée peut être en attente de détachement.

Toutes les formes d'ALTER TABLE qui agissent sur une seule table, à l'exception de `RENAME`, `SET SCHEMA`, `ATTACH PARTITION`, et `DETACH PARTITION` peuvent être combinées dans une liste de plusieurs altérations à appliquer en parallèle. Par exemple, il est possible d'ajouter plusieurs colonnes et/ou de modifier le type de plusieurs colonnes en une seule commande. Ceci est particulièrement utile avec les grosses tables car une seule passe sur la table est alors nécessaire.

Il faut être propriétaire de la table pour utiliser `ALTER TABLE`. Pour modifier le schéma ou le tablespace d'une table, le droit `CREATE` sur le nouveau schéma est requis. Pour ajouter la table en tant que nouvel enfant d'une table parent, vous devez aussi être propriétaire de la table parent. De plus, pour attacher une table en tant que nouvelle partition de la table, vous devez être le propriétaire de la table qui est attachée. Pour modifier le propriétaire, il est nécessaire d'être un membre direct ou indirect du nouveau rôle et ce dernier doit avoir le droit `CREATE` sur le schéma de la table. (Ces restrictions assurent que la modification du propriétaire ne diffère en rien de ce qu'il est possible de faire par la suppression et la recréation de la table. Néanmoins, un superutilisateur peut modifier le propriétaire de n'importe quelle table.) Pour ajouter une colonne ou modifier un type de colonne ou utiliser la clause `OF`, vous devez avoir le droit `USAGE` sur le type de la donnée.

Paramètres

`IF EXISTS`

Ne renvoie pas une erreur si la table n'existe pas. Un message d'avertissement est renvoyé dans ce cas.

`nom`

Le nom (éventuellement qualifié du nom du schéma) de la table à modifier. Si `ONLY` est indiqué avant le nom de la table, seule cette table est modifiée. Dans le cas contraire, la table et toutes ses tables filles (s'il y en a) sont modifiées. En option, `*` peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles doivent être inclues.

`nom_colonne`

Le nom d'une colonne, existante ou nouvelle.

nouveau_nom_colonne

Le nouveau nom d'une colonne existante.

nouveau_nom

Le nouveau nom de la table.

type_données

Le type de données de la nouvelle colonne, ou le nouveau type de données d'une colonne existante.

contraintedetable

Une nouvelle contrainte de table pour la table.

nomdecontrainte

Le nom d'une nouvelle contrainte ou d'une contrainte existante à supprimer.

CASCADE

Les objets qui dépendent de la colonne ou de la contrainte supprimée sont automatiquement supprimés (par exemple, les vues référençant la colonne), ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

RESTRICT

La colonne ou la contrainte n'est pas supprimée si des objets en dépendent. C'est le comportement par défaut.

nom_trigger

Le nom d'un trigger isolé à désactiver ou activer.

ALL

Désactiver ou activer tous les triggers appartenant à la table. (Les droits de superutilisateur sont nécessaires si l'un des triggers est un trigger interne pour la gestion d'une contrainte comme ceux utilisés pour implanter les contraintes de type clés étrangères ou les contraintes déferrables comme les contraintes uniques et d'exclusion.

USER

Désactiver ou activer tous les triggers appartenant à la table sauf les triggers systèmes permettant de gérer en interne certaines contraintes, comme celles utilisées pour implanter les contraintes de type clés étrangères ou les contraintes déferrables comme les contraintes uniques et d'exclusion.

nomindex

Le nom d'un index existant.

paramètre_stockage

Le nom d'un paramètre de stockage de la table.

valeur

La nouvelle valeur d'un paramètre de stockage de la table. Cela peut être un nombre ou un mot suivant le paramètre.

table_parent

Une table parent à associer ou dissocier de cette table.

nouveau_propriétaire

Le nom du nouveau propriétaire de la table.

new_access_method

Le nom de la méthode d'accès dans laquelle la table sera convertie.

nouveau_tablespace

Le nom du tablespace où déplacer la table.

nouveau_schéma

Le nom du schéma où déplacer la table.

nom_partition

Le nom de la table à attacher comme nouvelle partition ou à détacher de cette table.

spec_limite_partition

La spécification de limite de partition pour une nouvelle partition. Se référer à CREATE TABLE pour plus de détails sur la syntaxe.

Notes

Le mot clé `COLUMN` n'est pas nécessaire. Il peut être omis.

Quand une colonne est ajoutée avec `ADD COLUMN` et qu'un `DEFAULT` non volatile est spécifié, la valeur par défaut est évaluée au moment de la requête et le résultat stocké dans les méta-données de la table. Cette valeur sera utilisée pour la colonne sur toutes les lignes existantes. Si aucune valeur par défaut (`DEFAULT`) n'est indiquée, NULL est utilisé. Une réécriture de la table n'est jamais requise.

Ajouter une colonne avec un `DEFAULT` volatile ou changer le type d'une colonne existante requiert une réécriture complète de la table et de ses index. Il existe une exception lors du changement du type de données d'une colonne existante si la clause `USING` ne change pas le contenu de la colonne, et que l'ancien type est compatible binairement avec le nouveau type ou vers le domaine non contraint sur le nouveau type. Dans ces cas, la réécriture de la table n'est pas nécessaire. Néanmoins, les index doivent toujours être reconstruits sauf si le système peut vérifier que le nouvel index sera équivalent logiquement à l'index existant. Par exemple, si la collation d'une colonne a été modifiée, une reconstruction des index est toujours nécessaire car le nouvel ordre de tri pourrait être différent. Néanmoins, dans l'absence d'un changement de collation, une colonne peut être changée de `text` à `varchar` (ou vice versa) sans reconstruire les index parce que ces types de données sont triés de la même façon.

Ajouter une contrainte `CHECK` ou `NOT NULL` requiert de parcourir la table pour vérifier que les lignes existantes respectent cette contrainte, mais ne requiert par une ré-écriture de la table.

Pareillement, quand une nouvelle partition est attachée, elle pourrait être parcourue pour vérifier que les lignes existantes vérifient la contrainte de partition.

La raison principale de la possibilité de spécifier des changements multiples à l'aide d'une seule commande `ALTER TABLE` est la combinaison en une seule passe sur la table de plusieurs parcours et réécritures.

Le parcours d'une grosse table pour vérifier une nouvelle contrainte de clé étrangère ou une nouvelle contrainte de vérification peut prendre beaucoup de temps, et toute modification de données est bloquée tant que la commande `ALTER TABLE ADD CONSTRAINT` n'est pas validée. Le but principal de l'option `NOT VALID` est de réduire l'impact de l'ajout d'une nouvelle contrainte sur les demandes de modifications concurrentes. Avec la clause `NOT VALID`, la commande `ADD CONSTRAINT` n'effectue pas de parcours complet de la table et peut donc être validée instantanément. Ensuite,

une commande VALIDATE CONSTRAINT peut être lancée pour vérifier que les lignes existantes satisfont la contrainte. L'étape de validation ne nécessite pas de verrou empêchant les modifications concurrentes car la contrainte sera vérifiée pour les lignes insérées ou modifiées ; seules les lignes antérieures à l'ajout de la contrainte doivent être vérifiées. Ainsi, la validation ne nécessite qu'un verrou SHARE UPDATE EXCLUSIVE sur la table indiquée. (Si la contrainte est une contrainte de clé étrangère, alors un verrou ROW SHARE est aussi nécessaire sur la table référencée par la contrainte.) En plus d'améliorer la gestion de la concurrence des accès, on peut utiliser la clause NOT VALID suivie d'un VALIDATE CONSTRAINT dans le cas où des violations de cette contrainte sont connues. Une fois que la contrainte est en place, aucune nouvelle violation ne pourra avoir lieu et le problème préexistant pourra être géré plus tard pour qu'enfin une commande VALIDATE CONSTRAINT puisse finalement aboutir.

La forme DROP COLUMN ne supprime pas physiquement la colonne, mais la rend simplement invisible aux opérations SQL. Par la suite, les ordres d'insertion et de mise à jour sur cette table stockent une valeur NULL pour la colonne. Ainsi, supprimer une colonne ne réduit pas immédiatement la taille de la table sur disque car l'espace occupé par la colonne n'est pas récupéré. Cet espace est récupéré au fur et à mesure des mises à jour des lignes de la table.

Pour forcer une réécriture immédiate de la table, vous pouvez utiliser VACUUM FULL, CLUSTER ou bien une des formes de la commande ALTER TABLE qui force une réécriture. Ceci ne cause pas de modifications visibles dans la table, mais élimine des données qui ne sont plus utiles.

Les formes d'ALTER TABLE qui ré-écrivent la table ne sont pas sûres au niveau MVCC. Après une ré-écriture de la table, elle apparaitra vide pour les transactions concurrentes si elles ont utilisé une image de la base prise avant la ré-écriture de la table. Voir Section 13.6 pour plus de détails.

L'option USING de SET DATA TYPE peut en fait utiliser une expression qui implique d'anciennes valeurs de la ligne ; c'est-à-dire qu'il peut être fait référence aussi bien aux autres colonnes qu'à celle en cours de conversion. Cela permet d'effectuer des conversions très générales à l'aide de la syntaxe SET DATA TYPE. À cause de cette flexibilité, l'expression USING n'est pas appliquée à la valeur par défaut de la colonne (s'il y en a une) : le résultat pourrait ne pas être une expression constante requise pour une valeur par défaut. Lorsqu'il n'existe pas de transtypage, implicite ou d'affectation, entre les deux types, SET DATA TYPE peut échouer à convertir la valeur par défaut alors même que la clause USING est spécifiée. Dans de ce cas, il convient de supprimer la valeur par défaut avec DROP DEFAULT, d'exécuter ALTER TYPE et enfin d'utiliser SET DEFAULT pour ajouter une valeur par défaut appropriée. Des considérations similaires s'appliquent aux index et contraintes qui impliquent la colonne.

Si une table a des tables descendantes, il n'est pas permis d'ajouter, renommer ou changer le type d'une colonne dans la table parente sans faire la même chose sur tous les descendants. Cela permet de s'assurer que les descendants ont toujours des colonnes qui correspondent au parent. De la même façon, une contrainte CHECK ne peut pas être renommée dans la table parente sans également la renommer dans tous les descendants, afin que toutes les contraintes CHECK soient également en correspondance avec celles du parent et de ses descendants. (Néanmoins, cette restriction ne s'applique pas aux contraintes basées sur des index.) De plus, puisque la sélection de ligne de la table parente sélectionne également des lignes de ses descendants, une contrainte sur le parent ne peut pas être marquée comme valide à moins qu'elle ne le soit également sur tous les descendants. Dans tous ces cas, ALTER TABLE ONLY sera rejeté.

Un appel récursif à DROP COLUMN supprime la colonne d'une table descendante si et seulement si cette table n'hérite pas de cette colonne d'une autre table et que la colonne n'y a pas été définie indépendamment de tout héritage. Une suppression non récursive de colonne (ALTER TABLE ONLY ... DROP COLUMN) ne supprime jamais les colonnes descendantes ; elles sont marquées comme définies de manière indépendante, plutôt qu'héritées. Une commande DROP COLUMN non récursive échouera pour une table partitionnée, puisque toutes les partitions d'une table doivent avoir les même colonnes que la racine de partitionnement.

Les actions pour les colonnes d'identité (ADD GENERATED, SET etc., DROP IDENTITY), ainsi que les actions CLUSTER, OWNER et TABLESPACE ne sont jamais appelées récursivement sur les tables

descendantes ; c'est-à-dire qu'elles agissent comme si `ONLY` est spécifié. Les actions affectant les états de trigger s'exécutent récursivement aux partitions des tables partitionnées (sauf si `ONLY` est spécifié), mais jamais aux descendants d'un partitionnement par héritage. Seules les contraintes `CHECK` sont propagées, et uniquement si elles ne sont pas marquées `NO INHERIT`.

Tout changement sur une table du catalogue système est interdit.

Voir la commande CREATE TABLE pour avoir une description plus complète des paramètres valides. Chapitre 5 fournit de plus amples informations sur l'héritage.

Exemples

Ajouter une colonne de type `varchar` à une table :

```
ALTER TABLE distributeurs ADD COLUMN adresse varchar(30);
```

Cette requête entraînera le remplissage par des valeurs NULL de toutes les lignes existantes de la table pour cette nouvelle colonne.

Pour ajouter une colonne avec une valeur par défaut non NULL :

```
ALTER TABLE measurements
  ADD COLUMN mtime timestamp with time zone DEFAULT now();
```

Les lignes existantes seront remplies avec la date actuelle comme valeur de la nouvelle colonne, puis les nouvelles lignes recevront l'heure de leur insertion.

Pour ajouter une colonne avec une valeur par défaut différente entre les anciennes lignes et les nouvelles :

```
ALTER TABLE transactions
  ADD COLUMN status varchar(30) DEFAULT 'ancien',
  ALTER COLUMN status SET default 'nouveau';
```

Les lignes existantes seront remplies avec la valeur `ancien`, mais ensuite la valeur par défaut pour les requêtes suivantes sera `nouveau`. L'effet est le même que si deux sous-commandes avaient été émises par des requêtes `ALTER TABLE` séparées.

Supprimer une colonne de table :

```
ALTER TABLE distributeurs DROP COLUMN adresse RESTRICT;
```

Changer les types de deux colonnes en une seule opération :

```
ALTER TABLE distributeurs
    ALTER COLUMN adresse TYPE varchar(80),
    ALTER COLUMN nom TYPE varchar(100);
```

Convertir une colonne de type integer (entier) contenant une estampille temporelle UNIX en `timestamp with time zone` à l'aide d'une clause `USING` :

```
ALTER TABLE truc
    ALTER COLUMN truc_timestamp SET DATA TYPE timestamp with time
 zone
```

```
    USING
        timestamp with time zone 'epoch' + truc_timestamp *
 interval '1 second';
```

La même, quand la colonne a une expression par défaut qui ne sera pas convertie automatiquement vers le nouveau type de données :

```
ALTER TABLE truc
    ALTER COLUMN truc_timestamp DROP DEFAULT,
    ALTER COLUMN truc_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + truc_timestamp *
 interval '1 second',
    ALTER COLUMN truc_timestamp SET DEFAULT now();
```

Renommer une colonne existante :

```
ALTER TABLE distributeurs RENAME COLUMN adresse TO ville;
```

Renommer une table existante :

```
ALTER TABLE distributeurs RENAME TO fournisseurs;
```

Pour renommer une contrainte existante :

```
ALTER TABLE distributeurs RENAME CONSTRAINT verif_cp TO
 verif_code_postal;
```

Ajouter une contrainte NOT NULL à une colonne :

```
ALTER TABLE distributeurs ALTER COLUMN rue SET NOT NULL;
```

Supprimer la contrainte NOT NULL d'une colonne :

```
ALTER TABLE distributeurs ALTER COLUMN rue DROP NOT NULL;
```

Ajouter une contrainte de vérification sur une table et tous ses enfants :

```
ALTER TABLE distributeurs ADD CONSTRAINT verif_cp CHECK
 (char_length(code_postal) = 5);
```

Pour ajouter une contrainte `CHECK` à une table, mais pas à ses filles :

```
ALTER TABLE distributeurs ADD CONSTRAINT verif_cp CHECK
 (char_length(code_postal) = 5) NO INHERIT;
```

(La contrainte de vérification ne sera également pas héritée par les futures filles.)

Supprimer une contrainte de vérification d'une table et de toutes ses tables filles :

```
ALTER TABLE distributeurs DROP CONSTRAINT verif_cp;
```

Pour enlever une contrainte de vérification d'une table seule (pas sur ses enfants)

```
ALTER TABLE ONLY distributeurs DROP CONSTRAINT verif_cp;
```

(La contrainte de vérification reste en place pour toutes les tables filles).

Ajouter une contrainte de clé étrangère à une table :

```
ALTER TABLE distributeurs ADD CONSTRAINT dist_fk FOREIGN KEY
 (adresse) REFERENCES adresses (adresse);
```

Pour ajouter une contrainte de clé étrangère à une table avec le moins d'impact sur le reste de l'activité

```
ALTER TABLE distributeurs ADD CONSTRAINT distfk FOREIGN KEY
 (address) REFERENCES adresses (adresse) NOT VALID;
ALTER TABLE distributeurs VALIDATE CONSTRAINT distfk;
```

Ajouter une contrainte unique (multicolonnes) à une table :

```
ALTER TABLE distributeurs ADD CONSTRAINT dist_id_codepostal_key
 UNIQUE (dist_id, code_postal);
```

Ajouter une clé primaire nommée automatiquement à une table. Une table ne peut jamais avoir qu'une seule clé primaire.

```
ALTER TABLE distributeurs ADD PRIMARY KEY (dist_id);
```

Déplacer une table dans un tablespace différent :

```
ALTER TABLE distributeurs SET TABLESPACE tablespacerapide;
```

Déplacer une table dans un schéma différent :

```
ALTER TABLE mon_schema.distributeurs SET SCHEMA votre_schema;
```

Recréer une contrainte de clé primaire sans bloquer les mises à jour pendant la reconstruction de l'index :

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributeurs
 (dist_id);
ALTER TABLE distributeurs DROP CONSTRAINT distributeurs_pkey,
    ADD CONSTRAINT distributeurs_pkey PRIMARY KEY USING INDEX
 dist_id_temp_idx;
```

Pour attacher une partition à une table partitionnée par intervalles :

```
ALTER TABLE measurement
```

```
    ATTACH PARTITION measurement_y2016m07 FOR VALUES FROM
 ('2016-07-01') TO ('2016-08-01');
```

Pour attacher une partition à une table partitionnée par liste :

```
ALTER TABLE cities
    ATTACH PARTITION cities_ab FOR VALUES IN ('a', 'b');
```

Pour attacher une partition à une table partitionnée par hachage :

```
ALTER TABLE orders
    ATTACH PARTITION orders_p4 FOR VALUES WITH (MODULUS 4,
 REMAINDER 3);
```

Pour attacher une partition par défaut à une table partitionnée :

```
ALTER TABLE cities
    ATTACH PARTITION cities_partdef DEFAULT;
```

Pour détacher une partition d'une table partitionnée :

```
ALTER TABLE measurement
    DETACH PARTITION measurement_y2015m12;
```

Compatibilité

Les formes `ADD` (sans `USING INDEX`), `DROP [COLUMN]`, `DROP IDENTITY`, `valeur_redémarrage`, `SET DEFAULT`, `SET DATA TYPE` (sans `USING`), `SET GENERATED`, et `SET` *`option_sequence`* se conforment au standard SQL. Les autres formes sont des extensions PostgreSQL, tout comme la possibilité de spécifier plusieurs manipulations en une seule commande `ALTER TABLE`.

`ALTER TABLE DROP COLUMN` peut être utilisé pour supprimer la seule colonne d'une table, laissant une table dépourvue de colonne. C'est une extension au SQL, qui n'autorise pas les tables sans colonne.

Voir aussi

CREATE TABLE

ALTER TABLESPACE

ALTER TABLESPACE — Modifier la définition d'un tablespace

Synopsis

```
ALTER TABLESPACE nom RENAME TO nouveau_nom
ALTER TABLESPACE nom OWNER TO { nouveau_propriétaire | CURRENT_ROLE
 | CURRENT_USER | SESSION_USER }
ALTER TABLESPACE nom SET ( option_tablespace = valeur [, ... ] )
ALTER TABLESPACE nom RESET ( option_tablespace [, ... ] )
```

Description

`ALTER TABLESPACE` modifie la définition d'un tablespace.

Seul le propriétaire du tablespace peut changer la définition d'un tablespace. Pour modifier le propriétaire, il est nécessaire d'être un membre direct ou indirect du nouveau rôle propriétaire (les superutilisateurs ont automatiquement tous ces droits).

Paramètres

nom

Le nom du tablespace.

nouveau_nom

Le nouveau nom du tablespace. Le nouveau nom ne peut pas débuter par `pg_` car ces noms sont réservés aux tablespaces système.

nouveau_propriétaire

Le nouveau propriétaire du tablespace.

option_tablespace

Un paramètre du tablespace à configurer ou réinitialiser. Actuellement, les seuls paramètres disponibles sont `seq_page_cost` `random_page_cost`, `effective_io_concurrency` et `maintenance_io_concurrency`. Configurer une valeur pour un tablespace particulier surchargera l'estimation habituelle du planificateur pour le coût de lecture de pages pour les tables du tablespace, et le comportement de lecture en avance de l'exécuteur, comme indiqué par les paramètres de configuration du même nom (voir seq_page_cost, random_page_cost, effective_io_concurrency, maintenance_io_concurrency). Ceci peut être utile si un tablespace se trouve sur un disque qui est plus rapide ou plus lent du reste du système d'entrées/sorties.

Exemples

Renommer le tablespace `espace_index` en `raid_rapide` :

```
ALTER TABLESPACE espace_index RENAME TO raid_rapide;
```

Modifier le propriétaire du tablespace `espace_index` :

```
ALTER TABLESPACE espace_index OWNER TO mary;
```

Compatibilité

Il n'existe pas d'instruction `ALTER TABLESPACE` dans le standard SQL.

Voir aussi

CREATE TABLESPACE, DROP TABLESPACE

ALTER TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH CONFIGURATION — modifier la définition d'une configuration de recherche plein texte

Synopsis

```
ALTER TEXT SEARCH CONFIGURATION nom
    ADD MAPPING FOR type_jeton [, ... ] WITH nom_dictionnaire
 [, ... ]
ALTER TEXT SEARCH CONFIGURATION nom
    ALTER MAPPING FOR type_jeton [, ... ] WITH nom_dictionnaire
 [, ... ]
ALTER TEXT SEARCH CONFIGURATION nom
    ALTER MAPPING REPLACE vieux_dictionnaire
 WITH nouveau_dictionnaire
ALTER TEXT SEARCH CONFIGURATION nom
    ALTER MAPPING FOR type_jeton [, ... ]
 REPLACE vieux_dictionnaire WITH nouveau_dictionnaire
ALTER TEXT SEARCH CONFIGURATION nom
    DROP MAPPING [ IF EXISTS ] FOR type_jeton [, ... ]
ALTER TEXT SEARCH CONFIGURATION nom RENAME TO nouveau_nom
ALTER TEXT SEARCH CONFIGURATION nom OWNER TO { nouveau_propriétaire
 | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER TEXT SEARCH CONFIGURATION nom SET SCHEMA nouveau_schéma
```

Description

`ALTER TEXT SEARCH CONFIGURATION` modifie la définition d'une configuration de recherche plein texte. Vous pouvez modifier les correspondances à partir des types de jeton vers des dictionnaires, ou modifier le nom ou le propriétaire de la configuration.

Vous devez être le propriétaire de la configuration pour utiliser `ALTER TEXT SEARCH CONFIGURATION`.

Paramètres

nom

Le nom de la configuration de recherche plein texte (pouvant être qualifié du schéma).

type_jeton

Le nom d'un type de jeton qui est émis par l'analyseur de configuration.

nom_dictionnaire

Le nom d'un dictionnaire de recherche plein texte à consulter pour le type de jeton spécifié. Si plusieurs dictionnaires sont listés, ils sont consultés dans l'ordre d'apparence.

ancien_dictionnaire

Le nom d'un dictionnaire de recherche plein texte à remplacer dans la correspondance.

nouveau_dictionnaire

Le nom d'un dictionnaire de recherche plein texte à substituer à *ancien_dictionnaire*.

nouveau_nom

Le nouveau nom de la configuration de recherche plein texte.

newowner

Le nouveau propriétaire de la configuration de recherche plein texte.

nouveau_schéma

Le nouveau schéma de la configuration de recherche plein texte.

La forme `ADD MAPPING FOR` installe une liste de dictionnaires à consulter pour les types de jeton indiqués ; il y a une erreur s'il y a déjà une correspondance pour un des types de jeton. La forme `ALTER MAPPING FOR` fait de même mais en commençant par supprimer toute correspondance existante avec ces types de jeton. Les formes `ALTER MAPPING REPLACE` substituent *nouveau_dictionnaire* par *ancien_dictionnaire* partout où ce dernier apparaît. Ceci se fait pour les seuls types de jeton indiqués quand `FOR` apparaît ou pour toutes les correspondances de la configuration dans le cas contraire. La forme `DROP MAPPING` supprime tous les dictionnaires pour les types de jeton spécifiés, faisant en sorte que les jetons de ces types soient ignorés par la configuration de recherche plein texte. Il y a une erreur s'il n'y a pas de correspondance pour les types de jeton sauf si `IF EXISTS` a été ajouté.

Exemples

L'exemple suivant remplace le dictionnaire `english` avec le dictionnaire `swedish` partout où `english` est utilisé dans `ma_config`.

```
ALTER TEXT SEARCH CONFIGURATION ma_config
  ALTER MAPPING REPLACE english WITH swedish;
```

Compatibilité

Il n'existe pas d'instructions `ALTER TEXT SEARCH CONFIGURATION` dans le standard SQL.

Voir aussi

CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH DICTIONARY — modifier la définition d'un dictionnaire de recherche plein texte

Synopsis

```
ALTER TEXT SEARCH DICTIONARY nom (
    option [ = valeur ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY nom RENAME TO nouveau_nom
ALTER TEXT SEARCH DICTIONARY nom OWNER TO { nouveau_propriétaire |
 CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER TEXT SEARCH DICTIONARY nom SET SCHEMA nouveau_schéma
```

Description

`ALTER TEXT SEARCH DICTIONARY` modifie la définition d'un dictionnaire de recherche plein texte. Vous pouvez modifier les options spécifiques au modèle d'un dictionnaire. Vous pouvez aussi modifier le nom du dictionnaire et son propriétaire.

Vous devez être superutilisateur pour utiliser `ALTER TEXT SEARCH DICTIONARY`.

Paramètres

nom

Le nom du dictionnaire de recherche plein texte (pouvant être qualifié du schéma).

option

Le nom d'une option, spécifique au modèle, à configurer pour ce dictionnaire.

valeur

La nouvelle valeur à utiliser pour une option spécifique au modèle. Si le signe égal et la valeur sont omises, alors toute valeur précédente de cette option est supprimée du dictionnaire, permettant ainsi l'utilisation de la valeur par défaut.

nouveau_nom

Le nouveau nom du dictionnaire de recherche plein texte.

nouveau_proprietaire

Le nouveau propriétaire du dictionnaire de recherche plein texte.

nouveau_schéma

Le nouveau schéma du dictionnaire de recherche plein texte.

Les options spécifiques au modèle peuvent apparaître dans n'importe quel ordre.

Exemples

La commande exemple suivante modifie la liste des mots d'arrêt par un dictionnaire basé sur Snowball. Les autres paramètres restent inchangés.

```
ALTER TEXT SEARCH DICTIONARY mon_dico ( StopWords = nouveaurusse );
```

La commande exemple suivante modifie la langue par le hollandais et supprime complètement l'option des mots d'arrêt.

```
ALTER TEXT SEARCH DICTIONARY mon_dico ( language = dutch,
 StopWords );
```

La commande exemple suivante « met à jour » la définition du dictionnaire sans rien modifier.

```
ALTER TEXT SEARCH DICTIONARY mon_dico ( dummy );
```

(Ceci fonctionne parce que le code de suppression de l'option ne se plaint pas s'il n'y a pas d'options.) Cette astuce est utile lors de la modification des fichiers de configuration pour le dictionnaire : la commande ALTER forcera les sessions existantes à relire les fichiers de configuration, ce qu'elles ne feraient jamais si elles les avaient déjà lus.

Compatibilité

Il n'existe pas d'instruction `ALTER TEXT SEARCH DICTIONARY` dans le standard SQL.

Voir aussi

CREATE TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH PARSER

ALTER TEXT SEARCH PARSER — modifier la définition d'un analyseur de recherche plein texte

Synopsis

```
ALTER TEXT SEARCH PARSER nom RENAME TO nouveau_nom
ALTER TEXT SEARCH PARSER nom SET SCHEMA nouveau_schéma
```

Description

`ALTER TEXT SEARCH PARSER` modifie la définition d'un analyseur de recherche plein texte. Actuellement, la seule fonctionnalité supportée est la modification du nom de l'analyseur.

Vous devez être superutilisateur pour utiliser `ALTER TEXT SEARCH PARSER`.

Paramètres

nom

Le nom de l'analyseur de recherche plein texte (pouvant être qualifié du schéma).

nouveau_nom

Le nouveau nom de l'analyseur de recherche plein texte.

nouveau_schéma

Le nouveau schéma de l'analyseur de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction `ALTER TEXT SEARCH PARSER` dans le standard SQL.

Voir aussi

CREATE TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

ALTER TEXT SEARCH TEMPLATE

ALTER TEXT SEARCH TEMPLATE — modifier la définition d'un modèle de recherche plein texte

Synopsis

```
ALTER TEXT SEARCH TEMPLATE nom RENAME TO nouveau_nom
ALTER TEXT SEARCH TEMPLATE nom SET SCHEMA nouveau_schéma
```

Description

`ALTER TEXT SEARCH TEMPLATE` modifie la définition d'un modèle de recherche plein texte. Actuellement, la seule fonctionnalité supportée est la modification du nom du modèle.

Vous devez être superutilisateur pour utiliser `ALTER TEXT SEARCH TEMPLATE`.

Paramètres

nom

Le nom du modèle de recherche plein texte (pouvant être qualifié du schéma).

nouveau_nom

Le nouveau nom du modèle de recherche plein texte.

nouveau_schéma

Le nouveau schéma du modèle de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction `ALTER TEXT SEARCH TEMPLATE` dans le standard SQL.

Voir aussi

CREATE TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

ALTER TRIGGER

ALTER TRIGGER — Modifier la définition d'un trigger

Synopsis

```
ALTER TRIGGER nom ON nom_table RENAME TO nouveau_nom
ALTER TRIGGER nom ON nom_table [ NO ] DEPENDS ON
 EXTENSION nom_extension
```

Description

`ALTER TRIGGER` modifie les propriétés d'un trigger.

La clause `RENAME` renomme le trigger sans en changer la définition. Si la table du trigger est une table partitionnée, alors les triggers clones correspondants dans les partitions sont aussi renommés.

La clause `DEPENDS ON EXTENSION` marque le trigger comme dépendance de l'extension, pour qu'en cas de suppression de l'extension, le trigger soit lui-aussi supprimé automatiquement.

Seul le propriétaire de la table sur laquelle le trigger agit peut modifier ses propriétés.

Paramètres

nom

Le nom du trigger à modifier.

nom_table

La table sur laquelle le trigger agit.

nouveau_nom

Le nouveau nom du trigger.

nom_extension

Le nom de l'extension dont le trigger dépend (ou dont le trigger ne dépend plus si le mot-clé `NO` est ajouté). Un trigger dépendant d'une extension est automatiquement supprimé quand l'extension est supprimée.

Notes

La possibilité d'activer ou de désactiver temporairement un trigger est offerte par `ALTER TABLE`, et non par `ALTER TRIGGER` qui ne permet pas d'agir sur tous les triggers d'une table en une seule opération.

Exemples

Renommer un trigger :

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

Pour marquer un trigger comme dépendant d'une extension :

```
ALTER TRIGGER emp_stamp ON emp DEPENDS ON EXTENSION emplib;
```

Compatibilité

`ALTER TRIGGER` est une extension PostgreSQL au standard SQL.

Voir aussi

ALTER TABLE

ALTER TYPE

ALTER TYPE — Modifier la définition d'un type

Synopsis

```
ALTER TYPE nom OWNER TO { nouveau_propriétaire | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER }
ALTER TYPE nom RENAME TO nouveau_nom
ALTER TYPE nom SET SCHEMA nouveau_schéma
ALTER TYPE name RENAME ATTRIBUTE attribute_name
 TO new_attribute_name [ CASCADE | RESTRICT ]
ALTER TYPE name action [, ... ]
ALTER TYPE nom ADD VALUE [ IF NOT EXISTS ] nouvelle_valeur_enumérée
 [ { BEFORE | AFTER } valeur_enumérée ]
ALTER TYPE nom ADD VALUE [ IF NOT EXISTS ] nouvelle_valeur_enum
 [ { BEFORE | AFTER } valeur_enum_voisine ]
ALTER TYPE nom RENAME VALUE valeur_enum_existante
 TO nouvelle_valeur_enum
ALTER TYPE name SET ( property = value [, ... ] )

où action fait partie de :

    ADD ATTRIBUTE nom_attribut type_de_donnée
 [ COLLATE collationnement ] [ CASCADE | RESTRICT ]
    DROP ATTRIBUTE [ IF EXISTS ] nom_attribut [ CASCADE |
 RESTRICT ]
    ALTER ATTRIBUTE nom_attribut [ SET DATA ] TYPE type_de_donnée
 [ COLLATE collationnement ] [ CASCADE | RESTRICT ]
```

Description

`ALTER TYPE` modifie la définition d'un type existant. Les variantes suivantes existent :

`OWNER`

Cette forme change le propriétaire du type.

`RENAME`

Cette forme change le nom du type.

`SET SCHEMA`

Cette forme déplace le type dans un autre schéma.

`RENAME ATTRIBUTE`

Cette forme n'est seulement utilisable que sur des types composites. Elle change le nom d'un attribut individuel du type.

`ADD ATTRIBUTE`

Cette forme ajoute un nouvel attribut à un type composite, en utilisant la même syntaxe que `CREATE TYPE`.

`DROP ATTRIBUTE [ IF EXISTS ]`

Cette forme supprime un attribut d'un type composite. Si `IF EXISTS` est précisé et que l'attribut n'existe pas, aucune erreur n'est renvoyée. Dans ce cas, un avertissement est affichée à la place.

`ALTER ATTRIBUTE ... SET DATA TYPE`

Cette forme change le type d'un attribut d'un type composite.

`ADD VALUE [ IF NOT EXISTS ] [ BEFORE | AFTER ]`

Cette forme ajoute une valeur à une énumération. L'emplacement de la nouvelle valeur dans l'énumération peut être spécifié comme étant avant (`BEFORE`) ou après (`AFTER`) une des valeurs existantes. Dans le cas contraire, le nouvel élément est ajouté à la fin de la liste de valeurs.

Si `IF NOT EXISTS` est précisé, l'existence d'une valeur de même nom ne constitue par une erreur : un message d'avertissement sera envoyé mais aucune action ne sera prise. Dans le cas contraire, une erreur est renvoyée si la nouvelle valeur est déjà présente.

`RENAME VALUE`

Renomme une valeur d'un type énumération. La place de la valeur dans l'ordre de l'énumération n'est pas affecté. Une erreur sera renvoyée si la valeur spécifiée n'est pas présente ou si le nouveau nom est déjà présent.

`SET ( `*`propriété`*` = `*`valeur`*` [, ... ] )`

Cette forme est applicable seulement sur les types de base. Elle permet l'ajustement d'un sous-ensemble des propriétés de type de base qui peuvent être définies dans `CREATE TYPE`. Ces propriétés peuvent être modifiées spécifiquement :

- `RECEIVE` peut se voir affecter le nom d'une fonction d'entrée binaire ou `NONE` pour supprimer la fonction d'entrée binaire du type. L'utilisation de cette option requiert l'attribut `SUPERUSER`.
- `SEND` peut se voir affecter le nom d'une fonction de sortie binaire ou `NONE` pour supprimer la fonction de sortie binaire du type. L'utilisation de cette option requiert l'attribut `SUPERUSER`.
- `TYPMOD_IN` peut se voir affecter le nom d'une fonction de modificateur d'entrée du type ou `NONE` pour supprimer la fonction de modificateur d'entrée du type. L'utilisation de cette option requiert l'attribut `SUPERUSER`.
- `TYPMOD_OUT` peut se voir affecter le nom d'une fonction de modificateur de sortie du type ou `NONE` pour supprimer la fonction de modificateur de sortie du type. L'utilisation de cette option requiert l'attribut `SUPERUSER`.
- `ANALYZE` peut se voir affecter le nom d'une fonction de récupération des statistique pour ce type ou `NONE` pour supprimer la fonction de récupération des statistiques L'utilisation de cette option requiert l'attribut `SUPERUSER`.
- `SUBSCRIPT` peut être initialisé avec le nom d'une fonction de gestion d'indices spécifique au type, ou `NONE` pour supprimer la fonction de gestion d'indices du type. Utiliser cette fonction nécessite l'attribut de `SUPERUSER`.
- `STORAGE` Peut être affecté à `plain`, `extended`, `external` ou `main` (voir Section 73.2 pour plus d'informations sur ce que cela signifie). Cependant, passer de `plain` à un autre paramètre nécessite l'attribut `SUPERUSER` (car cela demande que toutes les fonctions de type C soient prêtes pour TOAST), et passer à `plain` depuis un autre paramétrage n'est pas autorisé du tout (car des valeurs de type TOAST peuvent déjà être contenues dans la base de données). Notez que modifier cette option ne modifie pas d'elle même les données stockées, elle permet juste de modifier la stratégie TOAST par défaut pour les colonnes de tables créées dans le futur. Voir ALTER TABLE pour changer la stratégie TOAST pour des colonnes d'une table existante.

Voir CREATE TYPE pour plus de détail sur ces propriétés de type. Notez qu'une modification sur les propriétés de type de base sera propagée automatiquement sur les domaines basés sur ce type si cela s'avère nécessaire.

Les actions ADD ATTRIBUTE, DROP ATTRIBUTE, et ALTER ATTRIBUTE peuvent être combinées dans une liste de modifications multiples à appliquer en parallèle. Il est ainsi possible d'ajouter et/ou modifier plusieurs attributs par une seule et même commande.

Seul le propriétaire du type peut utiliser ALTER TYPE. Pour modifier le schéma d'un type, le droit CREATE sur le nouveau schéma est requis. Pour modifier le propriétaire, il faut être un membre direct ou indirect du nouveau rôle propriétaire et ce rôle doit avoir le droit CREATE sur le schéma du type (ces restrictions assurent que la modification du propriétaire ne va pas au-delà de ce qui est possible par la suppression et la recréation du type ; toutefois, un superutilisateur peut modifier le propriétaire de n'importe quel type). Pour ajouter un attribut ou pour modifier le type d'un attribut, vous devez aussi avoir le droit USAGE sur le type de données de la colonne.

Paramètres

nom

Le nom du type à modifier (éventuellement qualifié du nom du schéma).

nouveau_nom

Le nouveau nom du type.

nouveau_propriétaire

Le nom du nouveau propriétaire du type.

nouveau_schema

Le nouveau schéma du type.

nom_attribut

Le nom de l'attribut à ajouter, modifier ou supprimer.

nouveau_nom_attribute

Le nouveau nom de l'attribut à renommer.

type_de_donnée

Le type de donnée pour l'attribut à ajouter ou modifier.

nouvelle_valeur_enumérée

La nouvelle valeur à ajouter à la liste d'un type, ou le nouveau nom à donner à une valeur existante. Comme pour tous les littéraux, la valeur devra être délimitée par des guillemets simples.

valeur_enumérée_voisine

La valeur existante d'une énumération par rapport à laquelle la nouvelle valeur doit être ajoutée (permet de déterminer l'ordre de tri du type énuméré). Comme pour tous les littéraux, la valeur existante devra être délimitée par des guillemets simples.

valeur_enum_existante

La valeur existante de l'énumération qui doit être renommée. Comme toutes les littéraux d'énumération, elle doit être délimitée par des guillemets simples.

property

Le nom de la propriété du type à modifier ; voir ci-dessus pour les valeurs possibles.

CASCADE

Propage automatiquement les opération sur les tables typées du type étant modifié, ainsi que leur descendants.

RESTRICT

Refuse les opérations si le type étant modifié est le type d'une table typée. C'est le comportement par défaut.

Notes

Si ALTER TYPE ... ADD VALUE (la forme qui ajoute une nouvelle valeur à un type énuméré) est exécuté à l'intérieur d'un bloc transactionnel, la nouvelle valeur ne peut être utilisée qu'à partir d'après la validation de la transaction.

Les comparaisons faisant intervenir une valeur ajoutée à postériori peuvent quelquefois s'avérer plus lentes que celles portant uniquement sur les valeurs originales d'un type énuméré. Ce ralentissement ne devrait toutefois intervenir que si la position de la nouvelle valeur a été spécifiée en utilisant les options BEFORE ou AFTER, au lieu d'insérer la nouvelle valeur en fin de liste. Ce ralentissement peut également se produire, bien que la nouvelle valeur ait été insérée en fin d'énumération, en cas de « bouclage » du compteur des OID depuis la création du type énuméré. Le ralentissement est généralement peu significatif ; mais s'il s'avère important, il est toujours possible de retrouver les performances optimales par une suppression / recréation du type énuméré, ou encore par sauvegarde et rechargement de la base.

Exemples

Pour renommer un type de données :

```
ALTER TYPE courrier_electronique RENAME TO courriel;
```

Donner la propriété du type courriel à joe :

```
ALTER TYPE courriel OWNER TO joe;
```

Changer le schéma du type courriel en clients :

```
ALTER TYPE courriel SET SCHEMA clients;
```

Ajouter un nouvel attribut à un type composite :

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

Ajouter une nouvelle valeur à une énumération, en spécifiant sa position de tri :

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

Pour renommer une valeur d'une énumération :

```
ALTER TYPE colors RENAME VALUE 'purple' TO 'mauve';
```

Créer une fonction I/O binaire pour un type existant :

```
CREATE FUNCTION mytypesend(mytype) RETURNS bytea ...;
CREATE FUNCTION mytyperecv(internal, oid, integer) RETURNS
 mytype ...;
ALTER TYPE mytype SET (
    SEND = mytypesend,
    RECEIVE = mytyperecv
);
```

Compatibilité

Les variantes permettant d'ajouter et supprimer un attribut font partie du standard SQL ; les autres variantes sont des extensions spécifiques à PostgreSQL.

Voir aussi

CREATE TYPE, DROP TYPE

ALTER USER

ALTER USER — Modifier un rôle de la base de données

Synopsis

```
ALTER USER spécification_rôle [ WITH ] option [ ... ]

où option peut être :

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT limite_connexion
    | [ ENCRYPTED ] PASSWORD 'motdepasse' | PASSWORD NULL
    | VALID UNTIL 'dateheure'

ALTER USER nom RENAME TO nouveau_nom

ALTER USER { spécification_rôle | ALL } [ IN DATABASE nom_base ]
 SET paramètre_configuration { TO | = } { valeur | DEFAULT }
ALTER USER { spécification_rôle | ALL } [ IN DATABASE nom_base ]
 SET paramètre_configuration FROM CURRENT
ALTER USER { spécification_rôle | ALL } [ IN DATABASE nom_base ]
 RESET paramètre_configuration
ALTER USER { spécification_rôle | ALL } [ IN DATABASE nom_base ]
 RESET ALL

où spécification_rôle peut valoir :

    nom_rôle
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
```

Description

`ALTER USER` est désormais un alias de `ALTER ROLE`.

Compatibilité

La commande `ALTER USER` est une extension PostgreSQL. En effet, le standard SQL laisse le choix de la définition des utilisateurs au SGBD.

Voir aussi

ALTER ROLE

ALTER USER MAPPING

ALTER USER MAPPING — change la définition d'une correspondance d'utilisateurs (user mapping)

Synopsis

```
ALTER USER MAPPING FOR { nom_utilisateur | USER | CURRENT_ROLE |
 CURRENT_USER | SESSION_USER | PUBLIC }
    SERVER nom_serveur
    OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ] )
```

Description

`ALTER USER MAPPING` change la définition d'une correspondance d'utilisateur (user mapping).

Le propriétaire d'un serveur distant peut aussi altérer les correspondances d'utilisateurs pour ce serveur pour tout utilisateur. Par ailleurs, un utilisateur peut modifier une correspondance d'utilisateur pour son propre nom d'utilisateur s'il a reçu le droit `USAGE` sur le serveur distant.

Paramètres

nom_utilisateur

Nom d'utilisateur de la correspondance. `CURRENT_ROLE`, `CURRENT_USER` et `USER` correspondent au nom de l'utilisateur courant. `PUBLIC` est utilisé pour correspondre à tous les noms d'utilisateurs présents et futurs du système.

nom_serveur

Nom du serveur de la correspondance d'utilisateur.

`OPTIONS ( [ ADD | SET | DROP ] option ['valeur'] [, ... ] )`

Modifie l'option pour la correspondance d'utilisateur. La nouvelle option écrase toute option précédemment spécifiée. `ADD`, `SET` et `DROP` spécifient l'action à exécuter. Si aucune action n'est spécifiée, l'action est `ADD`. Les noms d'options doivent être uniques ; les options sont aussi validées par le wrapper de données distantes du serveur.

Exemples

Modifier le mot de passe pour la correspondance d'utilisateur `bob`, et le serveur `foo` :

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (SET password
 'public');
```

Compatibilité

`ALTER USER MAPPING` est conforme à la norme ISO/IEC 9075-9 (SQL/MED). Il y a un problème de syntaxe subtil : le standard omet le mot clé `FOR`. Puisque `CREATE USER MAPPING` et `DROP USER MAPPING` utilisent tous les deux `FOR` à un endroit analogue et que DB2 d'IBM (l'autre implémentation majeure de SQL/MED) l'impose aussi pour `ALTER USER MAPPING`, PostgreSQL diverge du standard pour des raisons de cohérence et de compatibilité.

Voir aussi

CREATE USER MAPPING, DROP USER MAPPING

ALTER VIEW

ALTER VIEW — modifier la définition d'une vue

Synopsis

```
ALTER VIEW [ IF EXISTS ] nom ALTER [ COLUMN ] nom_colonne SET
 DEFAULT expression
ALTER VIEW [ IF EXISTS ] nom ALTER [ COLUMN ] nom_colonne DROP
 DEFAULT
ALTER VIEW [ IF EXISTS ] nom OWNER TO { nouveau_propriétaire |
 CURRENT_ROLE | CURRENT_USER | SESSION_USER }
ALTER VIEW [ IF EXISTS ] name RENAME [ COLUMN ] nom_colonne
 TO nouveau_nom_colonne
ALTER VIEW [ IF EXISTS ] nom RENAME TO nouveau_nom
ALTER VIEW [ IF EXISTS ] nom SET SCHEMA nouveau_schéma
ALTER VIEW [ IF EXISTS ] nom SET ( nom_option [= valeur_option]
 [, ... ] )
ALTER VIEW [ IF EXISTS ] nom RESET ( nom_option [, ... ] )
```

Description

`ALTER VIEW` modifie différentes propriétés d'une vue. Si vous voulez modifier la requête définissant la vue, utilisez `CREATE OR REPLACE VIEW`.)

Vous devez être le propriétaire de la vue pour utiliser `ALTER VIEW`. Pour modifier le schéma d'une vue, vous devez aussi avoir le droit `CREATE` sur le nouveau schéma. Pour modifier le propriétaire, vous devez aussi être un membre direct ou indirect de nouveau rôle propriétaire, et ce rôle doit avoir le droit `CREATE` sur le schéma de la vue. Ces restrictions permettent de s'assurer que le changement de propriétaire ne fera pas plus que ce que vous pourriez faire en supprimant et en recréant la vue. Néanmoins, un superutilisateur peut changer le propriétaire de n'importe quelle vue.

Paramètres

nom

Le nom de la vue (pouvant être qualifié du schéma).

nom_colonne

Nom de la colonne existante.

nouveau_nom_colonne

NOuveau nom de la colonne existante.

`IF EXISTS`

Ne renvoie par d'erreur si la vue n'existe pas. Seul un message d'avertissement est renvoyé dans ce cas.

`SET/DROP DEFAULT`

Ces formes ajoutent ou suppriment la valeur par défaut pour une colonne. La valeur par défaut d'une colonne de la vue est substituée dans toute commande `INSERT` ou `UPDATE` dont la vue est la cible, avant d'appliquer les règles et triggers de la vue. Le comportement par défaut de la vue prendra précédence sur toute valeur par défaut à partir des relations sous-jacentes.

nouveau_propriétaire

Nom utilisateur du nouveau propriétaire de la vue.

nouveau_nom

Nouveau nom de la vue.

nouveau_schéma

Nouveau schéma de la vue.

```
SET ( nom_option [= valeur_option] [, ... ] )
RESET ( nom_option [, ... ] )
```

Configure ou annule la configuration d'une option d'une vue. Les options actuellement supportées sont :

`check_option` (`enum`)

Modifie l'option de vérification d'une valeur. Les valeurs autorisées sont `local` et `cascaded`.

`security_barrier` (`boolean`)

Modifie la propriété security_barrier de la vue. Il s'agit d'une valeur booléenne, `true` ou `false`.

`security_invoker` (`boolean`)

Modifie la propriété security-invoker de la vue. Il s'agit d'une valeur booléenne, `true` ou `false`.

Notes

Pour des raisons historiques, `ALTER TABLE` peut aussi être utilisé avec des vues ; mais seules les variantes de `ALTER TABLE` qui sont acceptées avec les vues sont équivalentes à celles affichées ci-dessus.

Exemples

Pour renommer la vue `foo` en `bar` :

```
ALTER VIEW foo RENAME TO bar;
```

Pour attacher une valeur par défaut à une colonne dans une vue modifiable :

```
CREATE TABLE table_base (id int, ts timestamptz);
CREATE VIEW une_view AS SELECT * FROM table_base;
ALTER VIEW une_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO table_base(id) VALUES(1);  -- ts recevra une valeur
 NULL
INSERT INTO une_view(id) VALUES(2);  -- ts recevra l'heure courante
```

Compatibilité

`ALTER VIEW` est une extensions PostgreSQL du standard SQL.

Voir aussi

CREATE VIEW, DROP VIEW

ANALYZE

ANALYZE — Collecter les statistiques d'une base de données

Synopsis

```
ANALYZE [ ( option [, ...] ) ] [ table_et_colonnes [, ...] ]
ANALYZE [ VERBOSE ] [ table_et_colonnes [, ...] ]

où option peut valoir :

    VERBOSE [ booléen ]
    SKIP_LOCKED [ booléen ]

et table_et_colonnes est :

    nom_table [ ( nom_colonne [, ...] ) ]
```

Description

ANALYZE collecte des statistiques sur le contenu des tables de la base de données et stocke les résultats dans le catalogue système pg_statistic. L'optimiseur de requêtes les utilise pour déterminer les plans d'exécution les plus efficaces.

Sans une liste de *table_et_colonnes*, ANALYZE examine chaque table et vue matérialisée de la base de données courante lisible par l'utilisateur courant. Avec cette liste, ANALYZE n'examine que les tables de cette liste. Il est également possible de donner une liste de noms de colonnes pour une table, auquel cas seules les statistiques concernant ces colonnes sont collectées.

Quand la liste d'options est entourée de parenthèses, les options peuvent être écrites dans n'importe quel ordre. La syntaxe avec parenthèses a été introduite dans la version 11 de PostgreSQL ; la syntaxe sans parenthèses devient obsolète.

Paramètres

VERBOSE

L'affichage de messages de progression est activé.

SKIP_LOCKED

Précise qu'une commande ANALYZE, quand elle commence à travailler sur une relation, ne doit pas attendre la libération de verrous en conflit :si une relation ne peut être verrouillée immédiatement et sans attente, la relation est ignorée. Notez que même avec cette option, ANALYZE peut se retrouver bloqué en ouvrant les index d'une relation, ou en récupérant des échantillons de lignes de partitions, de tables héritant d'une autre, et de certains types de tables étrangères. Notez aussi que ANALYZE traite habituellement toutes les partitions des tables partitionnées demandées, mais il ignorera toutes les partitions s'il y a un verrou en conflit sur la table partitionnée.

booléen

Indique si l'option sélectionnée doit être activée ou désactivée. Vous pouvez écrire TRUE, ON ou 1 pour activer l'option, et FALSE, OFF ou 0 pour la désactiver. La valeur *booléen* peut aussi être omise, auquel cas TRUE est supposé.

nom_table

Le nom (éventuellement qualifié du nom du schéma) de la table à analyser. S'il n'est pas spécifié, toutes les tables standards, les tables partitionnées et les vues matérialisées dans la base de données courante sont analysées (mais pas les tables distantes). Si la table spécifiée est une table partitionnée, les statistiques héritées de la table partitionnée dans son ensemble ainsi que les statistiques des partitions individuelles sont mises à jour.

nom_colonne

Le nom d'une colonne à analyser. Par défaut, toutes les colonnes le sont.

Sorties

Quand VERBOSE est spécifié, ANALYZE affiche des messages de progression pour indiquer la table en cours de traitement. Diverses statistiques sur les tables sont aussi affichées.

Notes

Pour analyser une table, l'utilisateur doit être le propriétaire de la table ou un superutilisateur. Néanmoins, les propriétaires des bases ont le droit d'analyser toutes les tables situées dans leur bases, sauf les catalogues partagés. (La restriction pour les catalogues partagées signifie qu'un ANALYZE sur une base complète peut seulement être réalisé par un superutilisateur.) ANALYZE ignorera toutes les tables pour lesquelles l'utilisateur n'a pas le droit d'analyse.

Les tables distantes sont analysées seulement lorsqu'elles sont explicitement ciblées. Certains wrappers de données distantes ne supportent pas encore ANALYZE. Si le wrapper de la table distante ne supporte pas ANALYZE, la commande affiche un message d'avertissement et ne fait rien de plus.

Dans la configuration par défaut de PostgreSQL, le démon autovacuum (voir Section 25.1.6) s'occupe de l'analyse automatique des tables lorsqu'elles sont chargées pour la première fois avec des données, puis à chaque fois qu'elles sont modifiées via les opérations habituelles. Quand l'autovacuum est désactivé, il est intéressant de lancer ANALYZE périodiquement ou juste après avoir effectué de grosses modifications sur le contenu d'une table. Des statistiques à jour aident l'optimiseur à choisir le plan de requête le plus approprié et améliorent ainsi la vitesse du traitement des requêtes. Une stratégie habituelle pour les bases de données principalement en lecture consiste à lancer VACUUM et ANALYZE une fois par jour, au moment où le serveur est le moins sollicité. (Cela ne sera pas suffisant en cas de grosse activité en mise à jour.)

ANALYZE ne requiert qu'un verrou en lecture sur la table cible. Il peut donc être lancé en parallèle à d'autres activités sur la table.

Les statistiques récupérées par ANALYZE incluent habituellement une liste des quelques valeurs les plus communes dans chaque colonne et un histogramme affichant une distribution approximative des données dans chaque colonne. L'un ou les deux peuvent être omis si ANALYZE les juge inintéressants (par exemple, dans une colonne à clé unique, il n'y a pas de valeurs communes) ou si le type de données de la colonne ne supporte pas les opérateurs appropriés. Il y a plus d'informations sur les statistiques dans le Chapitre 25.

Pour les grosses tables, ANALYZE prend aléatoirement plusieurs lignes de la table, au hasard, plutôt que d'examiner chaque ligne. Ceci permet à des tables très larges d'être examinées rapidement. Néanmoins, les statistiques ne sont qu'approximatives et changent légèrement à chaque fois qu'ANALYZE est lancé, même si le contenu réel de la table n'a pas changé. Cela peut résulter en de petites modifications dans les coûts estimés par l'optimiseur affichés par EXPLAIN. Dans de rares situations, ce non-déterminisme entraîne le choix par l'optimiseur d'un plan de requête différent entre deux lancements d'ANALYZE. Afin d'éviter cela, le nombre de statistiques récupérées par ANALYZE peut être augmenté, comme cela est décrit ci-dessous.

L'étendue de l'analyse est contrôlée par l'ajustement de la variable de configuration default_statistics_target ou colonne par colonne en initialisant la cible des statistiques par colonne

avec ALTER TABLE ... ALTER COLUMN ... SET STATISTICS. Cette valeur cible initialise le nombre maximum d'entrées dans la liste des valeurs les plus communes et le nombre maximum de points dans l'histogramme. La valeur cible par défaut est fixée à 100 mais elle peut être ajustée vers le haut ou vers le bas afin d'obtenir un bon compromis entre la précision des estimations de l'optimiseur, le temps pris par ANALYZE et l'espace total occupé dans pg_statistic. En particulier, initialiser la cible des statistiques à zéro désactive la collecte de statistiques pour cette colonne. Cela peut s'avérer utile pour les colonnes qui ne sont jamais utilisées dans les clauses WHERE, GROUP BY ou ORDER BY des requêtes puisque l'optimiseur ne fait aucune utilisation des statistiques de ces colonnes.

La plus grande cible de statistiques parmi les colonnes en cours d'analyse détermine le nombre de lignes testées pour préparer les statistiques de la table. Augmenter cette cible implique une augmentation proportionnelle du temps et de l'espace nécessaires à l'exécution d'ANALYZE.

Une des valeurs estimées par ANALYZE est le nombre de valeurs distinctes qui apparaissent dans chaque colonne. Comme seul un sous-ensemble des lignes est examiné, cette estimation peut parfois être assez inexacte, même avec la cible statistique la plus large possible. Si cette inexactitude amène de mauvais plans de requêtes, une valeur plus précise peut être déterminée manuellement, puis configurée avec ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...).

Si la table en cours d'analyse a des enfants en héritage, ANALYZE récupère deux ensembles de statistiques : un sur les lignes de la table parent seule et un autre incluant les lignes du parent et de tous les enfants. Le deuxième ensemble de statistiques est nécessaire lors de l'optimisation de requêtes traitant l'arbre de tables comme un ensemble. Les enfants eux-même ne sont pas analysés dans ce cas. Néanmoins, le démon autovacuum ne prendra en considération que les insertions et les mises à jour sur la table parent elle-même pour décider de déclencher une analyse automatique sur cette table. Si cette table a rarement des insertions ou des mises à jour, les statistiques sur l'arbre d'héritage complet ne seront pas à jour tant que vous n'exécuterez pas ANALYZE manuellement.

Pour les tables partitionnées, ANALYZE récupère les statistiques en échantillonnant les lignes à partir de toutes les partitions ; de plus, il va parcourir chaque partition récursivement et mettre à jour ses statistiques. Chaque partition feuille est analysée seulement une fois, y compris dans le cas d'un partitionnement à plusieurs niveaux. Aucune statistique n'est récupérée pour la table parent seule (sans les données de ces partitions), parce qu'avec le partitionnement, elle est garantie d'être vide.

Le démon autovacuum ne traite pas les tables partitionnées, pas plus qu'il ne traite les parents en héritage si seules les tables filles sont modifiées. Il est généralement nécessaire d'exécuter périodiquement un ANALYZE manuel pour conserver des statistiques à jour sur la hiérarchie de tables.

Si certaines tables filles ou partitions sont des tables externes dont les wrappers de données externes ne supportent pas ANALYZE, ces tables sont ignorées lors de la récupération de statistiques pour l'héritage.

Si la table en cours d'analyse est entièrement vide, ANALYZE n'enregistrera pas les nouvelles statistiques pour cette table. Toutes les statistiques existantes seront conservées.

Chaque processus exécutant ANALYZE indiquera sa progression dans la vue pg_stat_progress_analyze. Voir Section 28.4.1 pour les détails.

Compatibilité

Il n'existe pas d'instruction ANALYZE dans le standard SQL.

Voir aussi

VACUUM, vacuumdb, Section 20.4.4, Section 25.1.6, Section 28.4.1

BEGIN

BEGIN — Débuter un bloc de transaction

Synopsis

```
BEGIN [ WORK | TRANSACTION ] [ mode_transaction [, ...] ]

où mode_transaction peut être :

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ
 COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Description

`BEGIN` initie un bloc de transaction, c'est-à-dire que toutes les instructions apparaissant après la commande `BEGIN` sont exécutées dans une seule transaction jusqu'à ce qu'un `COMMIT` ou `ROLLBACK` explicite soit exécuté. Par défaut (sans `BEGIN`), PostgreSQL exécute les transactions en mode « autocommit », c'est-à-dire que chaque instruction est exécutée dans sa propre transaction et une validation (commit) est traitée implicitement à la fin de l'instruction (si l'exécution a réussi, sinon une annulation est exécutée).

Les instructions sont exécutées plus rapidement dans un bloc de transaction parce que la séquence début/validation de transaction demande une activité significative du CPU et du disque. L'exécution de plusieurs instructions dans une transaction est aussi utile pour s'assurer d'une cohérence lors de la réalisation de certains modifications liées : les autres sessions ne voient pas les états intermédiaires tant que toutes les mises à jour ne sont pas réalisées.

Si le niveau d'isolation, le mode lecture/écriture ou le mode différable sont spécifiés, la nouvelle transaction possède ces caractéristiques, comme si `SET TRANSACTION` était exécutée.

Paramètres

`WORK`
`TRANSACTION`

Mots clés optionnels. Ils n'ont pas d'effet.

SET TRANSACTION présente la signification des autres paramètres de cette instruction.

Notes

`START TRANSACTION` est un synonyme de `BEGIN`.

`COMMIT` et `ROLLBACK` sont utilisés pour terminer un bloc de transaction.

Lancer `BEGIN` en étant déjà dans un bloc de transaction provoque l'apparition d'un message d'avertissement, mais l'état de la transaction n'en est pas affecté. Pour intégrer des transactions à l'intérieur d'un bloc de transaction, les points de sauvegarde sont utilisés (voir SAVEPOINT).

Pour des raisons de compatibilité descendante, les virgules entre chaque *`mode_transaction`* peuvent être omises.

Exemples

Commencer un bloc de transaction :

```
BEGIN;
```

Compatibilité

`BEGIN`, qui est une extension PostgreSQL, est équivalent à la commande `START TRANSACTION` du standard SQL. La page de référence de cette commande contient des informations supplémentaires de compatibilité.

L'option `DEFERRABLE` de *`mode_transaction`* est une extension de PostgreSQL.

Le mot clé `BEGIN` est utilisé dans un but différent en SQL embarqué. La sémantique de la transaction doit être étudiée avec précaution lors du portage d'applications.

Voir aussi

COMMIT, ROLLBACK, START TRANSACTION, SAVEPOINT

CALL

CALL — Exécuter une procédure

Synopsis

```
CALL nom ( [ argument ] [, ...] )
```

Description

`CALL` exécute une procédure.

Si la procédure a des arguments en sortie, alors une ligne de résultat sera retournée, contenant les valeurs de ces paramètres.

Paramètres

nom

Le nom (potentiellement qualifié du schéma) de la procédure.

argument

Un argument en entrée pour l'appel de la procédure.

Les arguments peuvent inclure des noms de paramètres, en utilisant la syntaxe *nom* `=>` *valeur*. Ceci fonctionne de la même façon que des appels de fonctions ordinaires ; voir Section 4.3 pour les détails.

Les arguments doivent être fournis pour tous les paramètres de la procédure qui n'ont pas de valeurs par défaut, y compris pour les paramètres `OUT`. Néanmoins, les arguments correspondant aux paramètres `OUT` ne sont pas évalués, donc il est habituel de simplement écrire `NULL` pour eux. (Écrire autre chose pour un paramètre `OUT` pourrait causer des problèmes de compatibilité avec les prochaines versions de PostgreSQL.)

Notes

L'utilisateur doit avoir le droit `EXECUTE` sur la procédure pour être autorisé à l'exécuter.

Pour appeler une fonction (pas une procédure), utilisez `SELECT` à la place.

Si `CALL` est exécuté dans un bloc de transaction, alors la procédure appelée ne peut pas exécuter d'ordre de contrôle de transaction. Les ordres de contrôle de transaction ne sont autorisés que si `CALL` est exécuté dans sa propre transaction.

PL/pgSQL gère différemment des paramètres en sortie dans les commandes `CALL` ; voir Section 43.6.3.

Exemples

```
CALL faire_maintenance_bd();
```

Compatibilité

`CALL` est conforme au standard SQL, sauf pour la gestion des paramètres en sortie. Le standard dit que les utilisateurs devraient écrire des variables pour recevoir les valeurs des paramètres en sortie.

Voir aussi

CREATE PROCEDURE

CHECKPOINT

CHECKPOINT — Forcer un checkpoint dans le journal des transactions

Synopsis

```
CHECKPOINT
```

Description

Un checkpoint est un point dans la séquence du journal des transactions pour lequel tous les fichiers de données ont été mis à jour pour refléter l'information des journaux. Tous les fichiers de données sont écrits sur le disque. Il convient de se référer à Chapitre 30 pour plus d'informations sur ce qui se produit lors d'un checkpoint.

La commande `CHECKPOINT` force un checkpoint immédiat, sans attendre le checkpoint régulier planifié par le système et contrôlé par les paramètres de Section 20.5.2. `CHECKPOINT` n'est généralement pas utilisé en temps normal.

S'il est exécuté lors d'une restauration, la commande `CHECKPOINT` forcera un restartpoint (voir Section 30.5) plutôt que l'écriture d'un nouveau checkpoint.

Seuls les superutilisateurs et les utilisateurs disposant des droits du rôle `pg_checkpoint` peuvent exécuter `CHECKPOINT`.

Compatibilité

La commande `CHECKPOINT` est une extension PostgreSQL.

CLOSE

CLOSE — Fermer un curseur

Synopsis

```
CLOSE { nom | ALL }
```

Description

`CLOSE` libère les ressources associées à un curseur ouvert. Une fois le curseur fermé, aucune opération n'est autorisée sur celui-ci. Un curseur doit être fermé lorsqu'il n'est plus nécessaire.

Tout curseur volatil ouvert (en anglais `non-holdable cursor`, soit un curseur qui ne perdure pas au-delà de la transaction qui l'a créé) est fermé implicitement lorsqu'une transaction est terminée avec `COMMIT` ou `ROLLBACK`. Un curseur persistant (en anglais `holdable cursor`, soit un curseur qui perdure au-delà de la transaction initiale) est implicitement fermé si la transaction qui l'a créé est annulée via `ROLLBACK`. Si cette transaction est validée (avec succès), ce curseur reste ouvert jusqu'à ce qu'une commande `CLOSE` explicite soit lancée ou jusqu'à la déconnexion du client.

Paramètres

name

Le nom du curseur ouvert à fermer.

`ALL`

Ferme tous les curseurs ouverts.

Notes

PostgreSQL ne possède pas d'instruction explicite d'ouverture (`OPEN`) de curseur ; un curseur est considéré ouvert à sa déclaration. Un curseur est déclaré à l'aide de l'instruction `DECLARE`.

Vous pouvez voir tous les curseurs disponibles en exécutant une requête sur la vue système `pg_cursors`.

Si un curseur est fermé après un point de sauvegarde qui est annulé par la suite, la commande `CLOSE` n'est pas annulée ; autrement dit, le curseur reste fermé.

Exemples

Fermer le curseur `liahona` :

```
CLOSE liahona;
```

Compatibilité

`CLOSE` est totalement conforme au standard SQL. `CLOSE ALL` est une extension PostgreSQL.

Voir aussi

DECLARE, FETCH, MOVE

CLUSTER

CLUSTER — Réorganiser une table en fonction d'un index

Synopsis

```
CLUSTER [VERBOSE] nom_table [ USING nom_index ]
CLUSTER ( option [, ...] ) nom_table [ USING nom_index ]
CLUSTER [VERBOSE]

où option peut faire partie de :

    VERBOSE [ boolean ]
```

Description

`CLUSTER` réorganise (trie) la table *`nom_table`* en fonction de l'index *`nom_index`*. L'index doit avoir été préalablement défini sur *`nom_table`*.

Une table réorganisée est physiquement triée en fonction des informations de l'index. Ce tri est une opération ponctuelle : les actualisations ultérieures ne sont pas triées. C'est-à-dire qu'aucune tentative n'est réalisée pour stocker les nouvelles lignes ou les lignes mises à jour d'après l'ordre de l'index. (Une réorganisation périodique peut être obtenue en relançant la commande aussi souvent que souhaité. De plus, configurer le paramètre `FILLFACTOR` à moins de 100% peut aider à préserver l'ordre du tri lors des mises à jour car les lignes mises à jour sont conservées dans la même page si suffisamment d'espace est disponible ici.)

Quand une table est réorganisée, PostgreSQL enregistre l'index utilisé à cet effet. La forme `CLUSTER` *`nom_table`* réorganise la table en utilisant le même index que la dernière fois. Vous pouvez aussi utiliser les formes `CLUSTER` ou `SET WITHOUT CLUSTER` de `ALTER TABLE` pour initialiser l'index de façon à ce qu'il soit intégré aux prochaines opérations CLUSTER ou pour supprimer tout précédent paramètre.

`CLUSTER`, sans paramètre, réorganise toutes les tables de la base de données courante qui ont déjà été réorganisées et dont l'utilisateur est propriétaire, ou toutes les tables s'il s'agit d'un superutilisateur. Cette forme de `CLUSTER` ne peut pas être exécutée à l'intérieur d'une transaction.

Quand une table est en cours de réorganisation, un verrou `ACCESS EXCLUSIVE` est acquis. Cela empêche toute opération sur la table (à la fois en lecture et en écriture) pendant l'exécution de `CLUSTER`.

Paramètres

`nom_table`

Le nom d'une table (éventuellement qualifié du nom du schéma).

`nom_index`

Le nom d'un index.

`VERBOSE`

Affiche la progression pour chaque table traitée.

boolean

Indique si l'option sélectionnée doit être activée ou non. Vous pouvez écrire TRUE, ON ou 1 pour activer l'option, et FALSE, OFF ou 0 pour la désactiver. La valeur *boolean* peut aussi être omise, auquel cas TRUE est supposé.

Notes

Lorsque les lignes d'une table sont accédées aléatoirement et unitairement, l'ordre réel des données dans la table n'a que peu d'importance. Toutefois, si certaines données sont plus accédées que d'autres, et qu'un index les regroupe, l'utilisation de CLUSTER peut s'avérer bénéfique. Si une requête porte sur un ensemble de valeurs indexées ou sur une seule valeur pour laquelle plusieurs lignes de la table correspondent, CLUSTER est utile. En effet, lorsque l'index identifie la page de la table pour la première ligne correspondante, toutes les autres lignes correspondantes sont déjà probablement sur la même page de table, ce qui diminue les accès disque et accélère la requête.

CLUSTER peut trier de nouveau en utilisant soit un parcours de l'index spécifié soit (si l'index est un Btree) un parcours séquentiel suivi d'un tri. Il choisira la méthode qui lui semble la plus rapide, en se basant sur les paramètres de coût du planificateur et sur les statistiques disponibles.

Quand un parcours d'index est utilisé, une copie temporaire de la table est créée. Elle contient les données de la table dans l'ordre de l'index. Des copies temporaires de chaque index sur la table sont aussi créées. De ce fait, vous devez disposer d'un espace libre sur le disque d'une taille au moins égale à la somme de la taille de la table et des index.

Quand un parcours séquentiel suivi d'un tri est utilisé, un fichier de tri temporaire est aussi créé. Donc l'espace temporaire requis correspond à au maximum le double de la taille de la table et des index. Cette méthode est généralement plus rapide que le parcours d'index mais si le besoin en espace disque est trop important, vous pouvez désactiver ce choix en désactivant temporairement enable_sort (off).

Il est conseillé de configurer maintenance_work_mem à une valeur suffisamment large (mais pas plus importante que la quantité de mémoire que vous pouvez dédier à l'opération CLUSTER) avant de lancer la commande.

Puisque l'optimiseur enregistre les statistiques d'ordonnancement des tables, il est conseillé de lancer ANALYZE sur la table nouvellement réorganisée. Dans le cas contraire, les plans de requêtes peuvent être mal choisis par l'optimiseur.

Comme CLUSTER se rappelle les index utilisés pour cette opération, un utilisateur peut exécuter manuellement des commandes CLUSTER une première fois, puis configurer un script de maintenance périodique qui n'exécutera qu'un CLUSTER sans paramètres, pour que les tables soient fréquemment triées physiquement.

Chaque processus exécutant CLUSTER indiquera sa progression dans la vue pg_stat_progress_cluster. Voir Section 28.4.4 pour les détails.

Exécuter la commande CLUSTER sur une table partitionnée va exécuter CLUSTER sur chacune de ses partitions en utilisant la partition de l'index partitionné indiqué. Dans ce cas, l'index ne peut pas être omis.

Exemples

Réorganiser la table employes sur la base de son index employes_ind :

```
CLUSTER employes ON employes_ind;
```

Réorganiser la relation employes en utilisant le même index que précédemment :

```
CLUSTER employes;
```

Réorganiser toutes les tables de la base de données qui ont déjà été préalablement réorganisées :

```
CLUSTER;
```

Compatibilité

Il n'existe pas d'instruction `CLUSTER` dans le standard SQL.

La syntaxe

```
CLUSTER nom_index ON nom_table
```

est aussi supportée pour la compatibilité avec les versions de PostgreSQL antérieures à la 8.3.

Voir aussi

clusterdb, Section 28.4.4

COMMENT

COMMENT — Définir ou modifier le commentaire associé à un objet

Synopsis

```
COMMENT ON
{
  ACCESS METHOD nom_objet |
  AGGREGATE nom_agrégat ( signature_agrégat ) |
  CAST (type_source AS type_cible) |
  COLLATION nom_objet |
  COLUMN nom_relation.nom_colonne |
  CONSTRAINT nom_contrainte ON nom_table |
  CONSTRAINT nom_contrainte ON DOMAIN nom_domaine |
  CONVERSION nom_objet |
  DATABASE nom_objet |
  DOMAIN nom_objet |
  EXTENSION nom_objet |
  EVENT TRIGGER nom_objet |
  FOREIGN DATA WRAPPER nom_objet |
  FOREIGN TABLE nom_objet |
  FUNCTION nom_fonction [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ] |
  INDEX nom_objet |
  LARGE OBJECT oid_large_objet |
  MATERIALIZED VIEW nom_objet |
  OPERATOR op (type_opérande1, type_opérande2) |
  OPERATOR CLASS nom_objet USING méthode_indexage |
  OPERATOR FAMILY nom_objet USING méthode_index |
  POLICY nom_politique ON nom_table |
  PROCEDURE nom_procédure [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ] |
  PUBLICATION nom_objet |
  ROLE nom_objet |
  ROUTINE nom_routine [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ] |
  RULE nom_règle ON nom_table |
  SCHEMA nom_objet |
  SEQUENCE nom_objet |
  SERVER nom_objet |
  STATISTICS nom_objet |
  SUBSCRIPTION nom_objet |
  TABLE nom_objet |
  TABLESPACE nom_objet |
  TEXT SEARCH CONFIGURATION nom_objet |
  TEXT SEARCH DICTIONARY nom_objet |
  TEXT SEARCH PARSER nom_objet |
  TEXT SEARCH TEMPLATE nom_objet |
  TRANSFORM FOR nom_type LANGUAGE nom_langage |
  TRIGGER nom_trigger ON nom_table |
  TYPE nom_objet |
  VIEW nom_objet
} IS { texte | NULL }
```

où *signature_agrégat* est :

```
* |
[ mode_arg ] [ nom_arg ] type_arg [ , ... ] |
[ [ mode_arg ] [ nom_arg ] type_arg [ , ... ] ] ORDER BY [ mode_arg
 ] [ nom_arg ] type_arg [ , ... ]
```

Description

`COMMENT` enregistre, modifie ou supprime un commentaire sur un objet de la base de données.

Seule une chaîne de commentaire est stockée pour chaque objet. Exécuter une nouvelle fois `COMMENT` pour le même objet remplace le commentaire existant. Indiquer `NULL` ou une chaîne vide (`''`) supprime le commentaire. Les commentaires sont automatiquement supprimés quand leur objet est supprimé.

Un verrou `SHARE UPDATE EXCLUSIVE` est acquis sur l'objet concerné par le commentaire.

Pour la plupart des types d'objet, seul le propriétaire de l'objet peut configurer le commentaire. Les rôles n'ont pas de propriétaires, donc la règle pour `COMMENT ON ROLE` est que vous devez être superutilisateur pour commenter un rôle superutilisateur ou avoir l'attribut `CREATEROLE` pour commenter des rôles standards. De la même façon, les méthodes d'accès n'ont pas encore de propriétaire ; vous devez être superutilisateur pour modifier le commentaire d'une méthode d'accès. Bien sûr, un superutilisateur peut ajouter un commentaire sur n'importe quel objet.

Les commentaires sont visibles avec la famille de commandes `\d`, de psql. D'autres interfaces utilisateur de récupération des commentaires peuvent être construites au-dessus des fonctions intégrées qu'utilise psql, à savoir `obj_description`, `col_description` et `shobj_description`. (Voir Tableau 9.77.)

Paramètres

nom_objet
nom_relation.nom_colonne
nom_agrégat
nom_contrainte
nom_fonction
op
nom_opérateur
nom_politique
nom_procédure
nom_routine
nom_règle
nom_trigger

Le nom de l'objet à commenter. Les noms des objets qui résident dans des schémas (tables, fonctions, etc.) peuvent être qualifiés du nom du schéma. Lorsque le commentaire est placé sur une colonne, *nom_relation* doit faire référence à une table, une vue, un type composite ou une table distante.

nom_table
nom_domaine

Lors de l'ajout d'un commentaire sur une contrainte, un trigger, une règle ou une politique, ces paramètres spécifient le nom de la table ou du domaine sur lequel cet objet est défini.

type_source

Le nom du type de donnée source du transtypage.

type_cible

Le nom du type de données cible du transtypage.

mode_arg

Le mode d'un argument de la fonction, de la procédure ou de l'agrégat : `IN`, `OUT`, `INOUT` ou `VARIADIC`. En cas d'omission, la valeur par défaut est `IN`. `COMMENT` ne tient pas compte, à l'heure actuelle, des arguments `OUT` car seuls ceux en entrée sont nécessaires pour déterminer l'identité de la fonction. Lister les arguments `IN`, `INOUT` et `VARIADIC` est suffisant.

nom_arg

Le nom d'un argument de la fonction, de la procédure ou de l'agrégat. `COMMENT ON FUNCTION` ne tient pas compte, à l'heure actuelle, des noms des arguments, seuls les types de données des arguments sont nécessaires pour déterminer l'identité de la fonction.

type_arg

Le type de données d'un argument de la fonction, de la procédure ou de l'agrégat.

oid_objet_large

L'OID de l'objet large.

type_gauche
type_droit

Les types de données des arguments de l'opérateur (avec en option le nom du schéma). Écrire `NONE` pour l'argument manquant d'un opérateur préfixe.

`PROCEDURAL`

Inutilisé.

nom_type

Le nom du type de données de la transformation.

nom_langage

Le nom du langage de la transformation.

texte

Le nouveau commentaire, rédigé sous la forme d'une chaîne littérale. Une chaîne vide (`''`) supprime le commentaire.

`NULL`

Écrire `NULL` pour supprimer le commentaire.

Notes

Il n'existe pas de mécanisme de sécurité pour visualiser les commentaires : tout utilisateur connecté à une base de données peut voir les commentaires de tous les objets de la base. Pour les objets partagés comme les bases, les rôles et les tablespaces, les commentaires sont stockées globalement et tout utilisateur connecté à une base peut voir tous les commentaires pour les objets partagés. Du coup, ne placez pas d'informations critiques pour la sécurité dans vos commentaires.

Exemples

Attacher un commentaire à la table `matable` :

```
COMMENT ON TABLE matable IS 'Ceci est ma table.';
```

Suppression du commentaire précédent :

```
COMMENT ON TABLE matable IS NULL;
```

Quelques exemples supplémentaires :

```
+COMMENT ON ACCESS METHOD gin IS Méthode d''accès GIN';
COMMENT ON AGGREGATE mon_agregat (double precision) IS 'Calcul
 d''une variance type';
COMMENT ON CAST (text AS int4) IS 'Transtypage de text en int4';
COMMENT ON COLLATION "fr_CA" IS 'Canadian French';
COMMENT ON COLUMN ma_table.ma_colonne IS 'Numéro employé';
COMMENT ON CONVERSION ma_conv IS 'Conversion vers UTF8';
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Contrainte sur la
 colonne col';
COMMENT ON CONSTRAINT dom_col_constr ON DOMAIN dom IS 'Contrainte
 sur la colonne du domaine';
COMMENT ON DATABASE ma_base IS 'Base de données de développement';
COMMENT ON DOMAIN mon_domaine IS 'Domaine des adresses de
 courriel';
COMMENT ON EVENT TRIGGER abort_ddl IS 'Annule toute commande DDL';
COMMENT ON EXTENSION hstore IS 'implémente le type de données
 hstore';
COMMENT ON FOREIGN DATA WRAPPER mon_wrapper IS 'mon wrapper de
 données distantes';
COMMENT ON FOREIGN TABLE ma_table_distante IS 'Information employés
 dans une autre base';
COMMENT ON FUNCTION ma_fonction (timestamp) IS 'Retourner des
 chiffres romains';
COMMENT ON INDEX mon_index IS 'S'assurer de l'unicité de l'ID de
 l'employé';
COMMENT ON LANGUAGE plpython IS 'Support de Python pour les
 procedures stockées';
COMMENT ON LARGE OBJECT 346344 IS 'Document de planification';
COMMENT ON MATERIALIZED VIEW ma_vuemat IS 'Résumé de l\'historique
 des ordres';
COMMENT ON OPERATOR ^ (text, text) IS 'L\'intersection de deux
 textes';
COMMENT ON OPERATOR - (NONE, integer) IS 'Moins unaire';
COMMENT ON OPERATOR CLASS int4ops USING btree IS 'Opérateurs
 d'entiers sur quatre octets pour les index btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'Tous les
 opérateurs entiers pour les index btree';
COMMENT ON POLICY ma_politique ON ma_table IS 'Filtre des lignes
 par utilisateur';
COMMENT ON PROCEDURE ma_proc (integer, integer) IS 'Lance un
 rapport';
COMMENT ON PUBLICATION toutes_tables IS 'Publit toutes les
 opérations sur toutes les tables';
COMMENT ON ROLE mon_role IS 'Groupe d'administration pour les
 tables finance';
COMMENT ON ROUTINE ma_routine (integer, integer) IS 'Exécute une
 routine (qui est une fonction ou une procédure)';
COMMENT ON RULE ma_regle ON my_table IS 'Tracer les mises à jour
 des enregistrements d\'employé';
```

```
COMMENT ON SCHEMA mon_schema IS 'Données du département';
COMMENT ON SEQUENCE ma_sequence IS 'Utilisé pour engendrer des clés
 primaires';
COMMENT ON SERVER mon_serveur IS 'mon serveur distant';
COMMENT ON STATISTICS ma_statistique IS 'Améliore les estimations
 de ligne de l''optimiseur';
COMMENT ON SUBSCRIPTION toutes_tables IS 'Souscription pour toutes
 les opérations sur toutes les tables';
COMMENT ON TABLE mon_schema.ma_table IS 'Informations sur les
 employés';
COMMENT ON TABLESPACE mon_tablespace IS 'Tablespace pour les
 index';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Filtre des mots
 spéciaux';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Stemmer Snowball pour
 le Suédois';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Divise le texte en
 mot';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Stemmer Snowball';
COMMENT ON TRANSFORM FOR hstore LANGUAGE plpython3u IS
 'Transformation entre hstore et un dictionnaire Python';
COMMENT ON TRIGGER mon_trigger ON my_table IS 'Utilisé pour RI';
COMMENT ON TYPE complex IS 'Type de données pour les nombres
 complexes';
COMMENT ON VIEW ma_vue IS 'Vue des coûts départementaux';
COMMENT ON VIEW ma_vue IS NULL;
```

Compatibilité

Il n'existe pas de commande `COMMENT` dans le standard SQL.

COMMIT

COMMIT — Valider la transaction en cours

Synopsis

```
COMMIT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Description

`COMMIT` valide la transaction en cours. Tout le monde peut désormais voir les modifications réalisées au cours de la transaction. De plus, leur persistance est garantie en cas d'arrêt brutal du serveur.

Paramètres

`WORK`
`TRANSACTION`

Mots clés optionnels et sans effet.

`AND CHAIN`

Si `AND CHAIN` est précisé, une nouvelle transaction est immédiatement commencée avec les mêmes caractéristiques (voir SET TRANSACTION) que celle qui vient de se terminer. Sinon, aucune nouvelle transaction n'est démarrée.

Notes

ROLLBACK est utilisé pour annuler une transaction.

Lancer `COMMIT` à l'extérieur d'une transaction n'a aucune conséquence mais provoque l'affichage d'un message d'avertissement. `COMMIT AND CHAIN` renvoie une erreur quand elle est exécutée en dehors d'une transaction.

Exemples

Valider la transaction courante et rendre toutes les modifications persistantes :

```
COMMIT;
```

Compatibilité

La commande `COMMIT` est conforme au standard SQL. La syntaxe `COMMIT TRANSACTION` est une extension PostgreSQL.

Voir aussi

BEGIN, ROLLBACK

COMMIT PREPARED

COMMIT PREPARED — Valider une transaction préalablement préparée en vue d'une validation en deux phases

Synopsis

```
COMMIT PREPARED id_transaction
```

Description

`COMMIT PREPARED` valide une transaction préparée.

Paramètres

`id_transaction`

L'identifiant de la transaction à valider.

Notes

Seul l'utilisateur à l'origine de la transaction ou un superutilisateur peut valider une transaction préparée. Il n'est cependant pas nécessaire d'être dans la session qui a initié la transaction.

Cette commande ne peut pas être exécutée à l'intérieur d'un bloc de transaction. La transaction préparée est validée immédiatement.

Toutes les transactions préparées disponibles sont listées dans la vue système `pg_prepared_xacts`.

Exemples

Valider la transaction identifiée par `foobar` :

```
COMMIT PREPARED 'foobar';
```

Compatibilité

L'instruction `COMMIT PREPARED` est une extension PostgreSQL. Elle est destinée à être utilisée par des systèmes tiers de gestion des transactions, dont le fonctionnement est parfois standardisé (comme X/Open XA), mais la portion SQL de ces systèmes ne respecte pas le standard.

Voir aussi

PREPARE TRANSACTION, ROLLBACK PREPARED

COPY

COPY — Copier des données depuis/vers un fichier vers/depuis une table

Synopsis

```
COPY nom_table [ ( nom_colonne [, ...] ) ]
    FROM { 'nom_fichier' | PROGRAM 'commande' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]
    [ WHERE condition ]

COPY { nom_table [ ( nom_colonne [, ...] ) ] | ( requête ) }
    TO { 'nom_fichier' | PROGRAM 'commande' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]

où option fait partie
de :

    FORMAT nom_format
    FREEZE [ booléen ]
    DELIMITER 'caractère_délimiteur'
    NULL 'chaîne_null'
    HEADER [ boolean | MATCH ]
    QUOTE 'caractère_guillemet'
    ESCAPE 'caractère_échappement'
    FORCE_QUOTE { ( nom_colonne [, ...] ) | * }
    FORCE_NOT_NULL ( nom_colonne [, ...] )
    FORCE_NULL ( nom_colonne [, ...] )
    ENCODING 'nom_encodage'
```

Description

`COPY` transfère des données entre les tables de PostgreSQL et les fichiers du système de fichiers standard. `COPY TO` copie le contenu d'une table *vers* un fichier tandis que `COPY FROM` copie des données *depuis* un fichier vers une table (ajoutant les données à celles déjà présentes dans la table). `COPY TO` peut aussi copier le résultat d'une requête `SELECT`.

Si une liste de colonnes est indiquée, `COPY TO` copie seulement les données des colonnes spécifiées dans le fichier. Pour `COPY FROM`, chaque champ du fichier est inséré, dans l'ordre, dans la colonne spécifiée. Les colonnes de la table non spécifiées dans la liste de colonnes de `COPY FROM` recevront leur valeur par défaut.

La commande `COPY` avec un nom de fichier force PostgreSQL à lire ou écrire directement dans un fichier. Il doit être accessible par l'utilisateur PostgreSQL (l'utilisateur exécutant le serveur) et le nom doit être spécifié du point de vue du serveur. Quand `PROGRAM` est indiqué, le serveur exécute la commande donnée, et lit la sortie standard du programme ou écrit dans l'entrée standard du programme. La commande doit être spécifiée du point de vue du serveur, et être exécutable par l'utilisateur PostgreSQL. Si `STDIN` ou `STDOUT` est indiqué, les données sont transmises au travers de la connexion entre le client et le serveur.

Chaque processus exécutant `COPY` indiquera sa progression dans la vue `pg_stat_progress_copy`. Voir Section 28.4.6 pour les détails.

Paramètres

nom_table

Le nom de la table (éventuellement qualifié du nom du schéma).

nom_colonne

Une liste optionnelle de colonnes à copier. Sans précision, toutes les colonnes de la table seront copiées.

requête

Une commande SELECT, VALUES, INSERT, UPDATE, ou DELETE dont les résultats sont à copier. Notez que des parenthèses sont requises autour de la requête.

Pour les requêtes INSERT, UPDATE et DELETE, une clause RETURNING doit être fournie, et la relation cible ne doit avoir ni règle conditionnelle, ni règle ALSO, ni règle INSTEAD qui ajoute plusieurs requêtes.

nom_fichier

Le chemin vers le fichier en entrée ou en sortie. Un nom de fichier en entrée peut avoir un chemin absolu ou relatif mais un nom de fichier en sortie doit absolument avoir un chemin absolu. Les utilisateurs Windows peuvent avoir besoin d'utiliser la syntaxe E'' et de doubler tous les antislashs utilisés dans le nom du chemin.

PROGRAM

Une commande à exécuter. Avec COPY FROM, l'entrée est lue de la sortie standard de la commande alors qu'avec COPY TO, la sortie est écrite dans l'entrée standard de la commande.

Notez que la commande est appelée par le shell. Si vous avez besoin de passer à la commande shell des arguments qui viennent d'une source sans confiance, vous devez faire particulièrement attention à supprimer ou échapper tous les caractères spéciaux qui pourraient avoir une signification particulière pour le shell. Pour des raisons de sécurité, il est préférable d'utiliser une chaîne de commande fixe ou, tout du moins, d'éviter de lui passer une entrée utilisateurq.

STDIN

Les données en entrée proviennent de l'application cliente.

STDOUT

Les données en sortie vont sur l'application cliente.

boolean

Spécifie si l'option sélectionnée doit être activée ou non. Vous pouvez écrire TRUE, ON ou 1 pour activer l'option, et FALSE, OFF ou 0 pour la désactiver. La valeur *boolean* peut aussi être omise, auquel cas la valeur TRUE est prise en compte.

FORMAT

Sélectionne le format des données pour la lecture ou l'écriture : text, csv (valeurs séparées par des virgules), ou binary. la valeur par défaut est text.

FREEZE

Demande la copie des données dans des lignes déjà gelées (donc dans le même état qu'après un VACUUM FREEZE). Ceci est une option de performance pour un chargement initial des données.

Les lignes seront gelées seulement si la table en cours de chargement a été créée ou tronquée dans la même sous-transaction, qu'il n'y a pas de curseurs ouverts ou d'anciennes images de la base de données détenus par cette transaction. Il n'est actuellement pas possible de réaliser un `COPY FREEZE` sur une table partitionnée.

Notez que toutes les autres sessions seront immédiatement capables de voir les données une fois qu'elles auront été chargées. Ceci viole les règles habituelles de la visibilité d'après MVCC. Les utilisateurs intéressées par cette option doivent être conscients des problèmes potentiels que cela peut poser.

`DELIMITER`

Spécifie le caractère qui sépare les colonnes sur chaque ligne du fichier. La valeur par défaut est une tabulation dans le format texte et une virgule dans le format `CSV`. Il doit être un seul caractère sur un seul octet. Cette option n'est pas autorisée lors de l'utilisation du format `binary`.

`NULL`

Spécifie la chaîne qui représente une valeur NULL. La valeur par défaut est `\N` (antislash-N) dans le format texte et une chaîne vide sans guillemets dans le format `CSV`. Vous pouvez préférer une chaîne vide même dans le format texte pour les cas où vous ne voulez pas distinguer les valeurs NULL des chaînes vides. Cette option n'est pas autorisée lors de l'utilisation du format `binary`.

> **Note**
>
> Lors de l'utilisation de `COPY FROM`, tout élément de données qui correspond à cette chaîne est stocké comme valeur NULL. Il est donc utile de s'assurer que c'est la même chaîne que celle précisée pour le `COPY TO` qui est utilisée.

`HEADER`

Le fichier contient une ligne d'en-tête avec les noms de chaque colonne. En sortie, la première ligne contient les noms de colonne de la table. En entrée, la première ligne est ignorée quand cette option vaut `true` (ou une valeur booléenne équivalente). Si cette option vaut `MATCH`, le nombre et les noms des colonnes de la ligne d'en-tête doivent correspondre aux noms des colonnes de la table, dans l'ordre. Dans le cas contre, une erreur est renvoyée. Cette option n'est pas autorisée avec le format `binary`. L'option `MATCH` est uniquement valide pour les commandes `COPY FROM`.

`QUOTE`

Spécifie le caractère guillemet à utiliser lorsqu'une valeur doit être entre guillemets. Par défaut, il s'agit du guillemet double. Cela doit de toute façon être un seul caractère sur un seul octet. Cette option n'est autorisée que lors de l'utilisation du format `CSV`.

`ESCAPE`

Spécifie le caractère qui doit apparaître avant un caractère de données qui correspond à la valeur `QUOTE`. La valeur par défaut est la même que la valeur `QUOTE` (du coup, le caractère guillemet est doublé s'il apparaît dans les données). Cela doit être un seul caractère codé en un seul octet. Cette option n'est autorisée que lors de l'utilisation du format `CSV`.

`FORCE_QUOTE`

Force l'utilisation des guillemets pour toutes les valeurs non `NULL` dans chaque colonne spécifiée. La sortie `NULL` n'est jamais entre guillemets. Si `*` est indiqué, les valeurs non `NULL` seront entre guillemets pour toutes les colonnes. Cette option est seulement autorisée avec `COPY TO` et seulement quand le format `CSV` est utilisé.

FORCE_NOT_NULL

Ne fait pas correspondre les valeurs des colonnes spécifiées avec la chaîne nulle. Dans le cas par défaut où la chaîne nulle est vide, cela signifie que les valeurs vides seront lues comme des chaînes de longueur nulle plutôt que comme des NULL, même si elles ne sont pas entre guillemets. Cette option est seulement autorisée avec COPY FROM et seulement quand le format CSV est utilisé.

FORCE_NULL

Essaie d'établir une correspondance entre les valeurs des colonnes spécifiées avec la chaîne NULL, même si elle est entre guillemets. Si une correspondance est trouvée, configure la valeur à NULL. Dans le cas par défaut où la chaîne NULL est vide, cela convertit une chaîne vide entre guillemets en valeur NULL. Cette option est uniquement autorisée avec COPY FROM, et seulement avec le format CSV.

ENCODING

Spécifie que le fichier est dans l'encodage *nom_encodage*. Si cette option est omise, l'encodage client par défaut est utilisé. Voir la partie Notes ci-dessous pour plus de détails.

WHERE

La clause optionnelle WHERE a la forme générale

```
WHERE condition
```

La *condition* where est une expression qui, évaluée ramènera un résultat de type booléen. Toutes les lignes qui ne valident pas la condition ne seront pas insérées dans la table. On considère qu'une ligne qui valide la condition est une ligne qui renvoie true lorsque la valeur réelle dans la table est comparée avec la condition.

Pour le moment, les sous-requêtes et les colonnes générées ne sont pas autorisées dans l'expression du WHERE, et l'évaluation ne verra pas les changements réalisés par la commande COPY elle même. (Cela a son importance lorsque l'expression utilisée contient des appels à des fonctions VOLATILE)

Affichage

En cas de succès, une commande COPY renvoie une balise de la forme

```
COPY nombre
```

Le *nombre* correspond au nombre de lignes copiées.

> **Note**
>
> psql affichera cette balise de commande seulement si la commande n'est pas COPY ... TO STDOUT ou son équivalent sous psql (la méta-commande \copy ... to stdout). Ceci a pour but d'empêcher toute confusion entre la balise de commande et les données affichées.

Notes

COPY TO ne peut être utilisé qu'avec des tables réelles, pas avec des vues, et ne peut pas copier les lignes des tables enfants ou des partitions enfants. Par exemple, COPY *table* TO copie les mêmes lignes que SELECT * FROM ONLY *table*. La syntaxe COPY (SELECT * FROM *table*)

`TO ...` peut être utilisé pour sauvegarder toutes les lignes dans une hiérarchie d'héritage, dans une table partitionnée, ou une vue.

`COPY FROM` peut être utilisée avec une table standard et avec des vues ayant des triggers `INSTEAD OF INSERT`.

Le droit SELECT est requis sur la table dont les valeurs sont lues par `COPY TO` et le droit INSERT sur la table dont les valeurs sont insérées par `COPY FROM`. Il est suffisant d'avoir des droits sur les colonnes listées dans la commande.

Si la sécurité de niveau ligne est activée pour la table, les politiques `SELECT` associées seront exécutées pour les instructions `COPY` *`table`* `TO`. Actuellement, `COPY FROM` n'est pas supporté pour les tables ayant une sécurité au niveau ligne. Utilisez les instructions `INSERT` équivalentes à la place.

Les fichiers nommés dans une commande `COPY` sont lus ou écrits directement par le serveur, non par l'application cliente. De ce fait, la machine hébergeant le serveur de bases de données doit les héberger ou pouvoir y accéder. L'utilisateur PostgreSQL (l'identifiant de l'utilisateur qui exécute le serveur), et non pas le client, doit pouvoir y accéder et les lire ou les modifier. De la même façon, la commande qui utilise `PROGRAM` est exécutée directement par le serveur, et non pas par l'application cliente. Elle doit être exécutable par l'utilisateur PostgreSQL. L'utilisation de `COPY` avec un fichier n'est autorisé qu'aux superutilisateurs de la base de données ou aux utilisateurs membres des rôles `pg_read_server_files`, `pg_write_server_files` ou `pg_execute_server_program` car `COPY` autorise la lecture et l'écriture de tout fichier accessible au serveur.

Il ne faut pas confondre `COPY` et l'instruction `\copy` de psql. `\copy` appelle `COPY FROM STDIN` ou `COPY TO STDOUT`, puis lit/stocke les données dans un fichier accessible au client psql. L'accès au fichier et les droits d'accès dépendent alors du client et non du serveur.

Il est recommandé que le chemin absolu du fichier utilisé dans `COPY` soit toujours précisé. Ceci est assuré par le serveur dans le cas d'un `COPY TO` mais, pour les `COPY FROM`, il est possible de lire un fichier spécifié par un chemin relatif. Le chemin est interprété relativement au répertoire de travail du processus serveur (habituellement dans le répertoire des données), pas par rapport au répertoire de travail du client.

Exécuter une commande avec `PROGRAM` peut être restreint par des mécanismes de contrôle d'accès du système d'exploitation, comme par exemple SELinux.

`COPY FROM` appelle tous les triggers et contraintes de vérification sur la table de destination, mais pas les règles.

Pour les colonnes d'identité, la commande `COPY FROM` écrira toujours les valeurs des colonnes fournies dans les données en entrée, comme l'option `INSERT` pour `OVERRIDING SYSTEM VALUE`.

L'entrée et la sortie de `COPY` sont sensibles à `datestyle`. Pour assurer la portabilité vers d'autres installations de PostgreSQL qui éventuellement utilisent des paramétrages `datestyle` différents de ceux par défaut, il est préférable de configurer `datestyle` en `ISO` avant d'utiliser `COPY TO`. Éviter d'exporter les données avec le `IntervalStyle` configuré à `sql_standard` est aussi une bonne idée car les valeurs négatives d'intervalles pourraient être mal interprétées par un serveur qui a une autre configuration pour `IntervalStyle`.

Les données en entrée sont interprétées suivant la clause `ENCODING` ou suivant l'encodage actuel du client. Les données en sortie sont codées suivant la clause `ENCODING` ou suivant l'encodage actuel du client. Ceci est valable même si les données ne passent pas par le client, c'est-à-dire si elles sont lues et écrites directement sur un fichier du serveur.

`COPY` stoppe l'opération à la première erreur. Si cela ne porte pas à conséquence dans le cas d'un `COPY TO`, il en va différemment dans le cas d'un `COPY FROM`. Dans ce cas, la table cible a déjà reçu les lignes précédentes. Ces lignes ne sont ni visibles, ni accessibles, mais occupent de l'espace disque. Il peut en résulter une perte importante d'espace disque si l'échec se produit lors d'une copie volumineuse. L'espace perdu peut alors être récupéré avec la commande `VACUUM`.

FORCE_NULL et FORCE_NOT_NULL peuvent être utilisés simultanément sur la même colonne. Cela a pour résultat la conversion des chaînes NULL entre guillemets en valeurs NULL et la conversion de chaînes NULL sans guillemets en chaînes vides.

Les données en entrée sont interprétées suivant l'encodage actuel du client et les données en sortie sont encodées suivant l'encodage client même si les données ne passent pas par le client mais sont lues à partir d'un fichier ou écrites dans un fichier.

Formats de fichiers

Format texte

Quand le format text est utilisé, les données sont lues ou écrites dans un fichier texte, chaque ligne correspondant à une ligne de la table. Les colonnes sont séparées, dans une ligne, par le caractère de délimitation. Les valeurs des colonnes sont des chaînes, engendrées par la fonction de sortie ou utilisables par celle d'entrée, correspondant au type de données des attributs. La chaîne de spécification des valeurs NULL est utilisée en lieu et place des valeurs nulles. COPY FROM lève une erreur si une ligne du fichier ne contient pas le nombre de colonnes attendues.

La fin des données peut être représentée par une ligne ne contenant qu'un antislash et un point (\.). Ce marqueur de fin de données n'est pas nécessaire lors de la lecture d'un fichier, la fin du fichier tenant ce rôle. Il n'est réellement nécessaire que lors d'une copie de données vers ou depuis une application cliente qui utilise un protocole client antérieur au 3.0.

Les caractères antislash (\) peuvent être utilisés dans les données de COPY pour échapper les caractères qui, sans cela, seraient considérés comme des délimiteurs de ligne ou de colonne. Les caractères suivants, en particulier, *doivent* être précédés d'un antislash s'ils apparaissent dans la valeur d'une colonne : l'antislash lui-même, le saut de ligne, le retour chariot et le délimiteur courant.

La chaîne NULL spécifiée est envoyée par COPY TO sans ajout d'antislash ; au contraire, COPY FROM teste l'entrée au regard de la chaîne NULL avant la suppression des antislash. Ainsi, une chaîne NULL telle que \N ne peut pas être confondue avec la valeur de donnée réelle \N (représentée dans ce cas par \\N).

Les séquences spéciales suivantes sont reconnues par COPY FROM :

Séquence	**Représente**
\b	Retour arrière (*backspace*) (ASCII 8)
\f	Retour chariot (ASCII 12)
\n	Nouvelle ligne (ASCII 10)
\r	Retour chariot (ASCII 13)
\t	Tabulation (ASCII 9)
\v	Tabulation verticale (ASCII 11)
chiffres	Antislash suivi d'un à trois chiffres en octal représente l'octet qui possède ce code numérique
\x*digits*	Antislash x suivi d'un ou deux chiffres hexadécimaux représente l'octet qui possède ce code numérique

Actuellement, COPY TO n'émet pas de séquence octale ou hexadécimale mais utilise les autres séquences listées ci-dessus pour les caractères de contrôle.

Tout autre caractère précédé d'un antislash se représente lui-même. Cependant, il faut faire attention à ne pas ajouter d'antislash qui ne soit pas absolument nécessaire afin d'éviter le risque d'obtenir accidentellement une correspondance avec le marqueur de fin de données (\.) ou la chaîne NULL (\N par défaut) ; ces chaînes sont reconnues avant tout traitement des antislashs.

Il est fortement recommandé que les applications qui engendrent des données COPY convertissent les données de nouvelle ligne et de retour chariot par les séquences respectives \n et \r. A l'heure actuelle, il est possible de représenter un retour chariot par un antislash et un retour chariot, et une nouvelle ligne par un antislash et une nouvelle ligne. Cependant, il n'est pas certain que ces représentations soient encore acceptées dans les prochaines versions. Celles-ci sont, de plus, extrêmement sensibles à la corruption si le fichier de COPY est transféré sur d'autres plateformes (d'un Unix vers un Windows ou inversement, par exemple).

Toutes les séquences avec antislash sont interprétées après la conversion d'encodage. Les octets indiquées avec des séquences octales ou hexadécimales doivent former des caractères valides dans l'encodage de la base.

COPY TO termine chaque ligne par une nouvelle ligne de style Unix (« \n »). Les serveurs fonctionnant sous Microsoft Windows engendrent un retour chariot/nouvelle ligne (« \r\n »), mais uniquement lorsque les données engendrées par COPY sont envoyées dans un fichier sur le serveur. Pour des raisons de cohérence entre les plateformes, COPY TO STDOUT envoie toujours « \n » quelque soit la plateforme du serveur. COPY FROM sait gérer les lignes terminant par une nouvelle ligne, un retour chariot ou un retour chariot suivi d'une nouvelle ligne. Afin de réduire les risques d'erreurs engendrées par des nouvelles lignes ou des retours chariot non précédés d'antislash, considéré de fait comme des données, COPY FROM émet un avertissement si les fins de lignes ne sont pas toutes identiques.

Format CSV

Ce format est utilisé pour importer et exporter des données au format de fichier CSV (acronyme de *Comma Separated Value*, littéralement valeurs séparées par des virgules). Ce format est utilisé par un grand nombre de programmes, tels les tableurs. À la place des règles d'échappement utilisées par le format texte standard de PostgreSQL, il produit et reconnaît le mécanisme d'échappement habituel de CSV.

Les valeurs de chaque enregistrement sont séparées par le caractère DELIMITER. Si la valeur contient ce caractère, le caractère QUOTE, la chaîne NULL, un retour chariot ou un saut de ligne, la valeur complète est préfixée et suffixée par le caractère QUOTE. De plus, toute occurrence du caractère QUOTE ou du caractère ESCAPE est précédée du caractère d'échappement. FORCE QUOTE peut également être utilisé pour forcer les guillemets lors de l'affichage de valeur non-NULL dans des colonnes spécifiques.

Le format CSV n'a pas de façon standard de distinguer une valeur NULL d'une chaîne vide. La commande COPY de PostgreSQL gère cela avec les guillemets. Un NULL est affiché suivant le paramètre NULL et n'est pas entre guillemets, alors qu'une valeur non NULL correspondant au paramètre NULL est entre guillemets. Par exemple, avec la configuration par défaut, un NULL est écrit avec la chaîne vide sans guillemets alors qu'une chaîne vide est écrit avec des guillemets doubles (""). La lecture des valeurs suit des règles similaires. Vous pouvez utiliser FORCE NOT NULL pour empêcher les comparaisons d'entrée NULL pour des colonnes spécifiques. Vous pouvez aussi utiliser FORCE_NULL pour convertir des valeurs de chaînes NULL entre guillemets en NULL.

L'antislash n'est pas un caractère spécial dans le format CSV. De ce fait, le marqueur de fin de données, \., peut apparaître dans les donnée. Afin d'éviter toute mauvaise interprétation, une valeur \. qui apparaît seule sur une ligne est automatiquement placée entre guillemets en sortie. En entrée, si elle est entre guillemets, elle n'est pas interprétée comme un marqueur de fin de données. Lors du chargement d'un fichier qui ne contient qu'une colonne, dont les valeurs ne sont pas placées entre guillemets, créé par une autre application, qui contient une valeur \., il est nécessaire de placer cette valeur entre guillemets.

> **Note**
>
> Dans le format CSV, tous les caractères sont significatifs. Une valeur entre guillemets entourée d'espaces ou de tout autre caractère différent de DELIMITER inclut ces caractères. Cela peut

être source d'erreurs en cas d'import de données à partir d'un système qui complète les lignes `CSV` avec des espaces fines pour atteindre une longueur fixée. Dans ce cas, il est nécessaire de pré-traiter le fichier `CSV` afin de supprimer les espaces de complètement avant d'insérer les données dans PostgreSQL.

Note

Le format CSV sait reconnaître et produire des fichiers CSV dont les valeurs entre guillemets contiennent des retours chariot et des sauts de ligne. De ce fait, les fichiers ne contiennent pas strictement une ligne par ligne de table comme les fichiers du format texte.

Note

Beaucoup de programmes produisent des fichiers CSV étranges et parfois pervers ; le format de fichier est donc plus une convention qu'un standard. Il est alors possible de rencontrer des fichiers que ce mécanisme ne sait pas importer. De plus, `COPY` peut produire des fichiers inutilisables par d'autres programmes.

Format binaire

Le format `binary` fait que toutes les données sont stockées/lues au format binaire plutôt que texte. Il est un peu plus rapide que les formats texte et `CSV` mais un fichier au format binaire est moins portable suivant les architectures des machines et les versions de PostgreSQL. De plus, le format binaire est très spécifique au type des données ; par exemple, un export de données binaires d'une colonne `smallint` ne pourra pas être importé dans une colonne `integer`, même si cela aurait fonctionné dans le format texte.

Le format de fichier `binary` consiste en un en-tête de fichier, zéro ou plusieurs lignes contenant les données de la ligne et un bas-de-page du fichier. Les en-têtes et les données sont dans l'ordre réseau des octets.

Note

Les versions de PostgreSQL antérieures à la 7.4 utilisaient un format de fichier binaire différent.

Entête du fichier

L'en-tête du fichier est constitutée de 15 octets de champs fixes, suivis par une aire d'extension de l'en-tête de longueur variable. Les champs fixes sont :

Signature

séquence de 11 octets `PGCOPY\n\377\r\n\0` -- l'octet zéro est une partie obligatoire de la signature. La signature est conçue pour permettre une identification aisée des fichiers qui ont été détériorés par un transfert non respectueux des huit bits. Cette signature est modifiée par les filtres de traduction de fin de ligne, la suppression des octets zéro, la suppression des bits de poids forts ou la modification de la parité.

Champs de commutateurs

masque entier de 32 bits décrivant les aspects importants du format de fichier. Les bits sont numérotés de 0 (LSB, ou *Least Significant Bit*, bit de poids faible) à 31 (MSB, ou *Most Significant*

Bit, bit de poids fort). Ce champ est stocké dans l'ordre réseau des octets (l'octet le plus significatif en premier), comme le sont tous les champs entier utilisés dans le format de fichier. Les bits 16 à 31 sont réservés aux problèmes critiques de format de fichier ; tout lecteur devrait annuler l'opération s'il trouve un bit inattendu dans cet ensemble. Les bits 0 à 15 sont réservés pour signaler les problèmes de compatibilité de formats ; un lecteur devrait simplement ignorer les bits inattendus dans cet ensemble. Actuellement, seul un bit est défini, le reste doit être à zéro :

Bit 16

si 1, les OID sont inclus dans la donnée ; si 0, non. Les colonnes système OID ne sont plus supportées dans PostgreSQL mais le format contient toujours l'indicateur.

Longueur de l'aire d'extension de l'en-tête

entier sur 32 bits, longueur en octets du reste de l'en-tête, octets de stockage de la longueur non-compris. À l'heure actuelle ce champ vaut zéro. La première ligne suit immédiatement. De futures modifications du format pourraient permettre la présence de données supplémentaires dans l'en-tête. Tout lecteur devrait ignorer silencieusement toute donnée de l'extension de l'en-tête qu'il ne saurait pas traiter.

L'aire d'extension de l'en-tête est prévue pour contenir une séquence de morceaux s'auto-identifiant. Le champ de commutateurs n'a pas pour but d'indiquer aux lecteurs ce qui se trouve dans l'aire d'extension. La conception spécifique du contenu de l'extension de l'en-tête est pour une prochaine version.

Cette conception permet l'ajout d'en-têtes compatible (ajout de morceaux d'extension d'en-tête, ou initialisation des octets commutateurs de poids faible) et les modifications non compatibles (initialisation des octets commutateurs de poids fort pour signaler de telles modifications, et ajout des données de support dans l'aire d'extension si nécessaire).

Tuples

Chaque tuple débute par un compteur, entier codé sur 16 bits, représentant le nombre de champs du tuple. (Actuellement, tous les tuples d'une table ont le même compteur, mais il est probable que cela ne soit pas toujours le cas.) On trouve ensuite, répété pour chaque champ du tuple, un mot de 32 bits annonçant le nombre d'octets de stockage de la donnée qui suivent. (Ce mot n'inclut pas sa longueur propre et peut donc être nul.) -1, cas spécial, indique une valeur de champ NULL. Dans ce cas, aucun octet de valeur ne suit.

Il n'y a ni complètement d'alignement ni toute autre donnée supplémentaire entre les champs.

Actuellement, toutes les valeurs d'un fichier d'un format binaire sont supposées être dans un format binaire (code de format). Il est probable qu'une extension future ajoute un champ d'en-tête autorisant la spécification de codes de format par colonne.

La consultation du code source de PostgreSQL, et en particulier les fonctions `*send` et `*recv` associées à chaque type de données de la colonne, permet de déterminer le format binaire approprié à la donnée réelle. Ces fonctions se situent dans le répertoire `src/backend/utils/adt/` des sources.

Lorsque les OID sont inclus dans le fichier, le champ OID suit immédiatement le compteur de champ. C'est un champ normal, à ceci près qu'il n'est pas inclus dans le compteur. Notez que les colonnes système OID ne sont pas supportées dans la version courante de PostgreSQL.

Queue du fichier

La fin du fichier consiste en un entier sur 16 bits contenant -1. Cela permet de le distinguer aisément du compteur de champs d'un tuple.

Il est souhaitable que le lecteur rapporte une erreur si le mot compteur de champ ne vaut ni -1 ni le nombre attendu de colonnes. Cela assure une vérification supplémentaire d'une éventuelle désynchronisation d'avec les données.

Exemples

Copier une table vers le client en utilisant la barre verticale (`|`) comme délimiteur de champ :

```
COPY pays TO STDOUT (DELIMITER '|');
```

Copier des données d'un fichier vers la table `pays` :

```
COPY pays FROM '/usr1/proj/bray/sql/pays_donnees';
```

Pour copier dans un fichier les pays dont le nom commence par 'A' :

```
COPY (SELECT * FROM pays WHERE nom_pays LIKE 'A%') TO '/usr1/proj/
bray/sql/une_liste_de_pays.copy';
```

Pour copier dans un fichier compressé, vous pouvez envoyer la sortie à un programme de compression externe :

```
COPY pays TO PROGRAM 'gzip > /usr1/proj/bray/sql/donnees_pays.gz';
```

Exemple de données convenables pour une copie vers une table depuis `STDIN` :

```
AF      AFGHANISTAN
AL      ALBANIE
DZ      ALGERIE
ZM      ZAMBIE
ZW      ZIMBABWE
```

L'espace sur chaque ligne est en fait un caractère de tabulation.

Les mêmes données, extraites au format binaire. Les données sont affichées après filtrage au travers de l'outil Unix `od -c`. La table a trois colonnes ; la première est de type `char(2)`, la deuxième de type `text` et la troisième de type `integer`. Toutes les lignes ont une valeur NULL sur la troisième colonne.

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0
  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013
   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0
 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N
   I
0000100   E 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0
  \0
0000120 007   A   L   G   E   R   I   E 377 377 377 377  \0 003  \0
  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   E 377
 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z
   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

Compatibilité

Il n'existe pas d'instruction COPY dans le standard SQL.

La syntaxe suivante était utilisée avant PostgreSQL 9.0 et est toujours supportée :

```
COPY nomtable [ ( colonne [, ...] ) ]
    FROM { 'nomfichier' | STDIN }
    [ [ WITH ]
        [ BINARY ]
 [ DELIMITER [ AS ] 'caractère_délimiteur' ]
 [ NULL [ AS ] 'chaîne NULL' ]
 [ CSV [ HEADER ]
       [ QUOTE [ AS ] 'caractère_guillemet' ]
       [ ESCAPE [ AS ] 'caractère_échappement' ]
       [ FORCE NOT NULL colonne [, ...] ] ] ]

COPY { nomtable [ ( colonne [, ...] ) ] | ( requête ) }
    TO { 'nomfichier' | STDOUT }
    [ [ WITH ]
        [ BINARY ]
 [ DELIMITER [ AS ] 'caractère_délimiteur' ]
 [ NULL [ AS ] 'chaîne NULL' ]
 [ CSV [ HEADER ]
       [ QUOTE [ AS ] 'caractère_guillemet' ]
       [ ESCAPE [ AS ] 'caractère_échappement' ]
       [ FORCE QUOTE colonne [, ...] | * } ] ] ]
```

Notez que, dans cette syntaxe, BINARY et CSV sont traités comme des mots-clés indépendants, pas comme des arguments à l'option FORMAT.

La syntaxe suivante, utilisée avant PostgreSQL version 7.3, est toujours supportée :

```
COPY [ BINARY ] nom_table
    FROM { 'nom_fichier' | STDIN }
    [ [USING] DELIMITERS 'caractère_délimiteur' ]
    [ WITH NULL AS 'chaîne NULL' ]

COPY [ BINARY ] nom_table
    TO { 'nom_fichier' | STDOUT }
    [ [USING] DELIMITERS 'caractère_délimiteur' ]
    [ WITH NULL AS 'chaîne NULL' ]
```

Voir aussi

Section 28.4.6

CREATE ACCESS METHOD

CREATE ACCESS METHOD — Définir une nouvelle méthode d'accès

Synopsis

```
CREATE ACCESS METHOD nom
    TYPE type_methode_access
    HANDLER fonction_handler
```

Description

`CREATE ACCESS METHOD` crée une nouvelle méthode d'accès.

Le nom de la méthode d'accès doit être unique au sein de la base de données.

Seuls les superutilisateurs peuvent définir de nouvelles méthodes d'accès.

Paramètres

nom

Le nom de la méthode d'accès à créer.

type_methode_access

Cette clause spécifie le type de méthode d'accès à définir. Les types `TABLE` et `INDEX` sont les seuls possibles pour l'instant.

fonction_handler

fonction_handler est le nom d'une fonction existante (potentiellement qualité par le nom du schéma) représentant la méthode d'accès. La fonction gestionnaire doit être déclarée comme prenant un seul argument de type `internal`, et son type de données en retour dépend du type de la méthode d'accès ; pour les méthodes d'accès `TABLE`, cela doit être `table_am_handler` et pour les méthodes d'accès `INDEX`, cela doit être `index_am_handler`. L'API niveau C que la fonction gestionnaire doit implémenter varie suivant le type de méthode d'accès. L'API de la méthode d'accès pour les tables est décrite dans Chapitre 63 et l'API de la méthode d'accès pour les index est décrite dans Chapitre 64.

Exemples

Créer une méthode d'accès d'index `heptree` avec une fonction handler `heptree_handler` :

```
CREATE ACCESS METHOD heptree TYPE INDEX HANDLER heptree_handler;
```

Compatibilité

`CREATE ACCESS METHOD` est une extension PostgreSQL.

Voir aussi

DROP ACCESS METHOD, CREATE OPERATOR CLASS, CREATE OPERATOR FAMILY

CREATE AGGREGATE

CREATE AGGREGATE — Définir une nouvelle fonction d'agrégat

Synopsis

```
CREATE AGGREGATE [ OR REPLACE ] nom ( [ mode_arg ] [ nom_arg
 ] type_donnees_arg [ , ... ] ) (
    SFUNC = sfonc,
    STYPE = type_donnée_état
   [ , SSPACE = taille_donnée_état ]
   [ , FINALFUNC = ffonc ]
   [ , FINALFUNC_EXTRA ]
   [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
   [ , COMBINEFUNC = combinefunc ]
   [ , SERIALFUNC = serialfunc ]
   [ , DESERIALFUNC = deserialfunc ]
   [ , INITCOND = condition_initiale ]
   [ , MSFUNC = msfonc ]
   [ , MINVFUNC = minvfonc ]
   [ , MSTYPE = type_donnée_état_m ]
   [ , MSSPACE = taille_donnée_état_m ]
   [ , MFINALFUNC = mffonc ]
   [ , MFINALFUNC_EXTRA ]
   [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE |
 READ_WRITE } ]
   [ , MINITCOND = condition_initiale_m ]
   [ , SORTOP = operateur_tri ]
   [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
   [ , HYPOTHETICAL ]
 )

CREATE [ OR REPLACE ] AGGREGATE nom ( [ [ mode_arg ] [ nom_arg
 ] type_donnees_arg [ , ... ] ]
                       ORDER BY [ mode_arg ] [ nom_arg
 ] type_donnees_arg [ , ... ] ) (
    SFUNC = sfonc,
    STYPE = type_donnée_état
    [ , SSPACE = taille_donnée_état ]
    [ , FINALFUNC = ffonc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , INITCOND = condition_initiale ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)

ou l'ancienne syntaxe

CREATE [ OR REPLACE ] AGGREGATE nom (
    BASETYPE = type_base,
    SFUNC = sfonc,
    STYPE = type_donnée_état
    [ , SSPACE = taille_donnée_état ]
    [ , FINALFUNC = ffonc ]
    [ , FINALFUNC_EXTRA ]
```

```
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , SERIALTYPE = serialtype ]
    [ , INITCOND = condition_initiale ]
    [ , MSFUNC = sfunc ]
    [ , MINVFUNC = invfunc ]
    [ , MSTYPE = state_data_type ]
    [ , MSSPACE = taille_donnée_état ]
    [ , MFINALFUNC = ffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE |
 READ_WRITE } ]
    [ , MINITCOND = condition_initiale ]
    [ , SORTOP = operateur_tri ]
)
```

Description

`CREATE AGGREGATE` définit une nouvelle fonction d'agrégat. `CREATE OR REPLACE AGGREGATE` définit une nouvelle fonction d'agrégat ou remplace une définition existante. Quelques fonctions d'agrégat basiques et largement utilisées sont fournies dans la distribution standard ; elles sont documentées dans le Section 9.21. `CREATE AGGREGATE` est utilisée pour ajouter des fonctionnalités lors de la définition de nouveaux types ou si une fonction d'agrégat n'est pas fournie.

Lorsqu'une définition existante est remplacée, les types d'arguments, le type de résultat, et le nombre d'arguments directs ne peuvent être changés. La nouvelle définition doit également être du même type (agrégat ordinaire, ou d'ensemble trié, ou d'ensemble hypothétique) que l'ancien.

Si un nom de schéma est donné (par exemple, `CREATE AGGREGATE monschema.monagg ...`), alors la fonction d'agrégat est créée dans le schéma précisé. Sinon, elle est créée dans le schéma courant. Ce comportement est identique à la surcharge de noms de fonctions ordinaires (voir CREATE FUNCTION).

Une fonction d'agrégat simple est identifiée par son nom et son (ou ses) types de données en entrée. Deux agrégats dans le même schéma peuvent avoir le même nom s'ils opèrent sur des types différents en entrée. Le nom et le(s) type(s) de données en entrée d'un agrégat doivent aussi être distincts du nom et du type de données de toutes les fonctions ordinaires du même schéma.

Une fonction d'agrégat est réalisée à partir d'une ou deux fonctions ordinaires : une fonction de transition d'état *`sfonc`*, et une fonction de traitement final optionnelle *`ffonc`*. Elles sont utilisées ainsi :

```
sfonc( état-interne, nouvelle-valeur-données ) ---> prochain-état-
interne
  ffonc( état-interne ) ---> valeur-agrégat
```

PostgreSQL crée une variable temporaire de type *`stype`* pour contenir l'état interne courant de l'agrégat. À chaque ligne en entrée, la valeur de l'argument de l'agrégat est calculée et la fonction de transition d'état est appelé avec la valeur d'état courante et la valeur du nouvel argument pour calculer une nouvelle valeur d'état interne. Une fois que toutes les lignes sont traitées, la fonction finale est appelée une seule fois pour calculer la valeur de retour de l'agrégat. S'il n'existe pas de fonction finale, alors la valeur d'état final est retournée en l'état.

Une fonction d'agrégat peut fournir une condition initiale, c'est-à-dire une valeur initiale pour la valeur de l'état interne. Elle est spécifiée et stockée en base comme une valeur de type `text` mais doit être

une représentation externe valide d'une constante du type de donnée de la valeur d'état. Si elle n'est pas fournie, la valeur d'état est initialement positionnée à NULL.

Si la fonction de transition d'état est déclarée « strict », alors elle ne peut pas être appelée avec des entrées NULL. Avec une telle fonction de transition, l'exécution d'agrégat se comporte comme suit. Les lignes avec une valeur NULL en entrée sont ignorées (la fonction n'est pas appelé et la valeur de l'état précédent est conservé). Si la valeur de l'état initial est NULL, alors, à la première ligne sans valeur NULL, la première valeur de l'argument remplace la valeur de l'état, et la fonction de transition est appelée pour chacune des lignes suivantes avec toutes les valeurs non NULL en entrée. Cela est pratique pour implémenter des agrégats comme `max`. Ce comportement n'est possible que quand *`type_donnée_état`* est identique au premier *`type_donnée_argument`*. Lorsque ces types sont différents, une condition initiale non NULL doit être fournie, ou une fonction de transition non stricte utilisée.

Si la fonction de transition d'état n'est pas stricte, alors elle sera appelée sans condition pour chaque ligne en entrée et devra gérer les entrées NULL et les valeurs de transition NULL. Cela permet à l'auteur de l'agrégat d'avoir le contrôle complet sur la gestion des valeurs NULL par l'agrégat.

Si la fonction finale est déclarée « strict », alors elle ne sera pas appelée quand la valeur d'état finale est NULL ; à la place, un résultat NULL sera retourné automatiquement. C'est le comportement normal de fonctions strictes. Dans tous les cas, la fonction finale peut retourner une valeur NULL. Par exemple, la fonction finale pour `avg` renvoie NULL lorsqu'elle n'a aucune lignes en entrée.

Quelque fois, il est utile de déclarer la fonction finale comme ne retournant pas seulement la valeur d'état, mais des paramètres supplémentaires correspondant aux valeurs en entrée de l'agrégat. La raison principale pour faire ainsi est si la fonction finale est polymorphique et que le type de données de la valeur de l'état serait inadéquate pour trouver le type du résultat. Ces paramètres supplémentaires sont toujours passés en tant que valeurs NULL (et donc la fonction finale ne doit pas être stricte quand l'option `FINALFUNC_EXTRA` est utilisée). Néanmoins, ce sont des paramètres valides. Par exemple, la fonction finale pourrait faire usage de `get_fn_expr_argtype` pour identifier le type d'argument réel dans l'appel actuel.

Un agrégat peut accepter en option un *mode d'agrégat glissant*, comme décrit dans Section 38.12.1. Ceci requiert de spécifier les paramètres `MSFUNC`, `MINVFUNC`, et `MSTYPE` et, en option, les paramètres `MSSPACE`, `MFINALFUNC`, `MFINALFUNC_EXTRA`, `MFINALFUNC_MODIFY`, et `MINITCOND`. En dehors de `MINVFUNC`, ces paramètres fonctionnement comme les paramètres d'agrégat simple sans `M` ; ils définissent une implémentation séparée de l'agrégat qui inclut une fonction de transition inverse.

La syntaxe avec `ORDER BY` dans la liste des paramètres crée un type spécial d'agrégat appelé un *agrégat d'ensemble trié*. Si le mot clé `HYPOTHETICAL` est ajouté, un *agrégat d'ensemble hypothétique* est créé. Ces agrégats opèrent sur des groupes de valeurs triées, donc la spécification d'un ordre de tri en entrée est une partie essentielle d'un appel. De plus, ils peuvent avoir des arguments *directs*, qui sont des arguments évalués une fois seulement par agrégat plutôt qu'une fois par ligne en entrée. Les agrégats d'ensemble hypothétique sont une sous-classe des agrégats d'ensemble trié pour lesquels certains des arguments directs doivent correspondre, en nombre et type de données aux colonnes en argument de l'agrégat. Ceci permet aux valeurs de ces arguments directs d'être ajoutées à la collection de lignes en entrée de l'agrégat comme des lignes supplémentaires « hypothétiques ».

Un agrégat peut supporter en option l'*agrégat partiel*, comme décrit dans Section 38.12.4. Ceci requiert la spécification du paramètre `COMBINEFUNC`. Si le paramètre *`state_data_type`* vaut `internal`, il est généralement approprié de fournir les paramètres `SERIALFUNC` et `DESERIALFUNC` pour qu'un agrégat parallèle soit possible. Notez que l'agrégat doit aussi être marqué `PARALLEL SAFE` pour activer l'agrégation parallélisée.

Les agrégats qui se comportent comme `MIN` ou `MAX` peuvent parfois être optimisés en cherchant un index au lieu de parcourir toutes les lignes en entrée. Si un agrégat peut être optimisé, un *opérateur de tri* est spécifié. Dans ce cas, il est nécessaire que l'agrégat fournisse le premier élément dans l'ordre imposé par l'opérateur ; en d'autres mots :

```
SELECT agg(col) FROM tab;
```

doit être équivalent à :

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

On suppose également que l'agrégat ignore les entrées NULL et qu'il fournit un résultat NULL si et seulement s'il n'y a aucune entrée NULL. D'ordinaire, l'opérateur < d'un type de données est le bon opérateur de tri pour MIN et > celui pour MAX. L'optimisation ne prend jamais effet sauf si l'opérateur spécifié est membre de la stratégie « less than » (NdT : plus petit que) ou « greater than » (NdT : plus grand que) d'une classe d'opérateurs pour un index B-tree.

Pour pouvoir créer une fonction d'agrégat, vous devez avoir le droit USAGE sur le type des arguments, le type de l'état et le type du code retour. Vous devez aussi voir le droit EXECUTE sur les fonction de support.

Paramètres

nom

Le nom de la fonction d'agrégat à créer (éventuellement qualifié du nom du schéma).

mode_arg

Le mode d'un argument : IN ou VARIADIC. (Les fonctions d'agrégat n'acceptent pas les arguments OUT.) Si le mode est omis, la valeur par défaut est IN. Seul le dernier argument peut être marqué comme VARIADIC.

nom_arg

Le nom d'un argument. Ceci est seulement utile pour de la documentation. S'il est omis, l'argument n'a pas de nom.

type_données_arg

Un type de donnée en entrée sur lequel opère la fonction d'agrégat. Pour créer une fonction d'agrégat sans argument, placez * à la place de la liste des types de données en argument. (la fonction count(*) en est un bon exemple.)

type_base

Dans l'ancienne syntaxe de CREATE AGGREGATE, le type de données en entrée est spécifiée par un paramètre type_base plutôt que d'être écrit à la suite du nom de l'agrégat. Notez que cette syntaxe autorise seulement un paramètre en entrée. Pour définir une fonction d'agrégat sans argument avec cette syntaxe, indiquez seulement un paramètre en entrée. Pour définir une fonction d'agrégat sans argument, utilisez "ANY" (et non pas *) pour le type_base. Les agrégats d'ensemble trié ne peuvent pas être définis avec l'ancienne syntaxe.

sfonc

Le nom de la fonction de transition de l'état à appeler pour chaque ligne en entrée. Pour une fonction d'agrégat simple avec *N* arguments, *sfonc* doit prendre *N*+1 arguments, le premier étant de type *type_données_état* et le reste devant correspondre aux types de données en entrée déclarés pour l'agrégat. La fonction doit renvoyer une valeur de type *type_données_état*. Cette fonction prend la valeur actuelle de l'état et les valeurs actuelles des données en entrée. Elle renvoie la prochaine valeur de l'état.

Pour les agrégats d'ensemble trié (incluant les ensembles hypothétiques), la fonction de transition d'état reçoit seulement la valeur de l'état actuel et les arguments agrégés, pas les arguments directs.

type_donnée_état

Le type de donnée pour la valeur d'état de l'agrégat.

taille_données_état

La taille moyenne approximative (en octets) de la valeur d'état de l'agrégat. Si ce paramètre est omis ou s'il vaut zéro, une estimation par défaut est utilisé en se basant sur *type_données_état*. Le planificateur utilise cette valeur pour estimer la mémoire requise pour une requête d'agrégat par groupe.

ffonc

Le nom de la fonction finale à appeler pour traiter le résultat de l'agrégat une fois que toutes les lignes en entrée ont été parcourues. Pour un agrégat normal, la fonction prend un seul argument de type *type_donnée_état*. Le type de retour de l'agrégat de la fonction est défini comme le type de retour de cette fonction. Si *ffonc* n'est pas spécifiée, alors la valeur d'état finale est utilisée comme résultat de l'agrégat et le type de retour est *type_donnée_état*.

Pour les agrégats d'ensemble trié (incluant les ensembles hypothétiques), la fonction finale reçoit non seulement la valeur de l'état final, mais aussi les valeurs de tous les arguments directs.

Si FINALFUNC_EXTRA est indiqué, en plus de la valeur de l'état final et des arguments directs, la fonction finale reçoit des valeurs NULL supplémentaires correspondant aux arguments agrégés standards de l'agrégat. Ceci est principalement utile pour permettre une bonne résolution du type de données pour le résultat agrégé quand un agrégat polymorphique est en cours de définition.

FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }

Cette option spécifie si la fonction finale est une fonction pure qui ne modifie pas ses arguments. READ_ONLY indique qu'il n'y a pas de modification. Les deux autres valeurs indiquent la valeur d'état de transition pourrait changer. Voir Notes ci-dessous pour plus de détail. La valeur par défaut est READ_ONLY, sauf pour les agrégats à ensemble ordonné dont la valeur par défaut est READ_WRITE.

combinefunc

La fonction *combinefunc* peut être indiquée en option pour permettre à la fonction d'agrégat de supporter l'agrégation partielle. Si elle est fournie, la fonction *combinefunc* doit combiner deux valeurs *state_data_type*, chacune contenant le résultat de l'agrégation sur un certain sous-ensemble des valeurs en entrée pour produire un nouveau *state_data_type* qui représente le résultat de l'agrégation sur les différents ensembles en entrée. Cette fonction peut être vue comme un *sfunc*, où, au lieu d'agir sur une ligne individuelle en entrée et de l'ajouter à l'état de l'agrégat en cours, elle ajoute un autre état d'agrégat à l'état en cours.

La fonction *combinefunc* doit être déclarée comme prenant deux arguments de type *state_data_type* et renvoyant une valeur de type *state_data_type*. En option, cette fonction pourrait être « strict ». Dans ce cas, la fonction ne sera pas appelé quand l'un des états en entrée est null ; l'autre état sera utilisé comme résultat.

Pour les fonctions d'agrégat où *state_data_type* vaut internal, la fonction *combinefunc* ne doit pas être stricte. Dans ce cas, la fonction *combinefunc* doit s'assurer que les états null sont gérés correctement et que l'état à renvoyer est correctement enregistré dans le contexte mémoire de l'agrégat.

serialfunc

Une fonction d'agrégat dont *state_data_type* est internal peut participer à une agrégation en parallèle seulement si elle a une fonction *serialfunc*, qui doit sérialiser l'état d'agrégat en une valeur bytea pour sa transmission à un autre processus. Cette fonction doit prendre un seul argument de type internal et renvoyer le bytea. Une fonction *deserialfunc* correspondante est aussi requise.

deserialfunc

Désérialise un état d'agrégat préalablement sérialisé dans son type *state_data_type*. Cette fonction doit prendre deux arguments de type bytea et internal, et produire un résultat de type internal. (Note : le second argument, de type internal, n'est pas utilisé mais est requis pour des raisons de sécurité.)

condition_initiale

La configuration initiale pour la valeur de l'état. Elle doit être une constante de type chaîne de caractères dans la forme acceptée par le type de données *type_donnée_état*. Si non spécifié, la valeur d'état est initialement positionnée à NULL.

msfonc

Le nom de la fonction de transition d'état à appeler pour chaque ligne en entrée dans le mode d'agrégat en déplacement. Elle est identique à la fonction de transition standard, sauf que son premier argument et son résultat sont de type *type_données_état_m*, qui pourrait être différent de *type_données_état*.

minvfonc

Le nom de la fonction de transition d'état inverse à utiliser dans le mode d'agrégat en déplacement. Cette fonction a les mêmes types d'argument et de résultat que *msfonc*, mais il est utilisé pour supprimer une valeur de l'état courant de l'agrégat, plutôt que pour y ajouter une valeur. La fonction de transition inverse doit avoir le même attribut strict que la fonction de transaction d'état.

type_données_état_m

Le type de données pour la valeur d'état de l'agrégat dans le mode d'agrégat en déplacement.

taille_données_état_m

La taille moyenne approximative (en octets) de la valeur d'état de l'agrégat. Ceci fonctionne de la même façon que *taille_données_état*.

mffonc

Le nom de la fonction finale appelée pour calculer le résultat de l'agrégat après que toutes les lignes en entrée aient été traversées, lors de l'utilisation du mode d'agrégat en déplacement. Ceci fonctionne de la même façon que *ffonc*, sauf que le type du premier argument est *type_données_état_m* et des arguments supplémentaires sont indiqués en écrivant MFINALFUNC_EXTRA. Le type en résultat de l'agrégat déterminé par *mffonc* ou *mstate_data_type* doit correspondre à celui déterminé par l'implémentation standard de l'agrégat.

MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }

Cette option est identique à FINALFUNC_MODIFY, mais elle décrit le comportement de la fonction finale pour un agrégat en déplacement.

condition_initiale_m

La configuration initiale de la valeur d'état lors de l'utilisation du mode d'agrégat en déplacement. Ceci fonctionne de la même façon que *condition_initiale*.

sort_operator

L'opérateur de tri associé pour un agrégat de type MIN ou MAX. C'est seulement le nom de l'opérateur (éventuellement qualifié du nom du schéma). L'opérateur est supposé avoir les mêmes types de données en entrée que l'agrégat (qui doit être un agrégat normal à un seul argument).

PARALLEL

La signification de PARALLEL SAFE, PARALLEL RESTRICTED et PARALLEL UNSAFE est la même que pour CREATE FUNCTION. Un agrégat ne sera pas considéré pour la parallélisation s'il est marqué PARALLEL UNSAFE (ce qui est le cas par défaut !) ou PARALLEL RESTRICTED. Notez que le marquage de parallélisation des fonctions de support des agrégats ne sont pas consultés par le planificateur. Ce dernier ne prend en considération que le marquage de l'agrégat lui-même.

HYPOTHETICAL

Pour les agrégats d'ensembles triés seulement, cette option indique que les arguments de l'agrégat sont à traiter suivant les prérequis des agrégats d'ensembles hypothétiques : les derniers arguments directs doivent correspondre aux types de données des arguments agrégés (WITHIN GROUP). L'option HYPOTHETICAL n'a pas d'effet sur le comportement à l'exécution, seulement sur la durée de résolution de l'analyse des types de données et des collationnements des arguments de l'agrégat.

Les paramètres de CREATE AGGREGATE peuvent être écrits dans n'importe quel ordre, pas uniquement dans l'ordre illustré ci-dessus.

Notes

Dans les paramètres qui indiquent les noms de fonction de support, vous pouvez écrire un nom de schéma si nécessaire, par exemple SFUNC = public.sum. N'écrivez pas de types d'argument ici, néanmoins -- les types d'argument des fonctions de support sont déterminés avec d'autres paramètres.

D'ordinaire, les fonctions PostgreSQL sont de vraies fonctions qui ne modifient pas leurs valeurs en entrée. Néanmoins, une fonction transition d'agrégat, *quand elle est utilisée dans le contexte d'un agrégat*, est autorisée à tricher et à modifier son argument d'état de transition. Ceci apporte des améliorations substantielles de performance en comparaison à une copie fraîche de l'état de transition à chaque exécution.

De la même façon, quand une fonction finale d'agrégat ne modifie pas habituellement ses arguments, il n'est parfois pas pratique d'éviter la modification de l'argument d'état de transition. Un tel comportement doit être déclaré en utilisant le paramètre FINALFUNC_MODIFY. La valeur READ_WRITE indique que la fonction finale modifie l'état de transaction de façon non spécifiée. Cette valeur empêche l'utilisation de l'agrégat comme fonction de fenêtrage et il empêche aussi l'assemblage des états de transition pour les appels d'agrégat qui partagent les mêmes valeurs en entrée et de transition. La valeur SHAREABLE indique que la fonction de transition ne peut être appliquée après la fonction finale mais que plusieurs appels de la fonction finale peuvent être réalisés sur la valeur d'état de transition final. Cette valeur empêche l'utilisation de l'agrégat comme fonction de fenêtrage mais permet l'assemblage des états des transition. (l'optimisation intéressante ici n'est pas d'appliquer la même fonction finale de façon répétée mais d'appliquer plusieurs fonctions finales différentes à la même valeur d'état de transition final. Ceci est permis tant qu'aucune des fonctions finales n'est marquées READ_WRITE.)

Si un agrégat accepte le mode d'agrégat par déplacement, cela améliorera l'efficacité du calcul quand l'agrégat est utilisé comme fonction de fenêtrage pour une fenêtre avec un début d'échelle qui se déplace (autrement dit, un mode de début d'échelle autre que UNBOUNDED PRECEDING). Conceptuellement, la fonction de transition ajoute des valeurs en entrée à l'état de l'agrégat quand elles entrent dans la fenêtre à partir du bas, et la fonction de transition inverse les supprime de nouveau quand elles quittent la fenêtre par le haut. Donc, quand les valeurs sont supprimées, elles sont toujours supprimées dans le même ordre qu'elles ont été ajoutées. Quand la fonction de transition inverse est appelée, elle va de ce fait recevoir l'entrée la plus récemment ajoutée, mais pas supprimée. La fonction de transition inverse peut assumer qu'au moins une ligne restera dans l'état courant après avoir supprimé la ligne la plus ancienne. (Quand cela n'est pas le cas, le mécanisme de la fonction de fenêtrage lance une nouvelle agrégation, plutôt que d'utiliser la fonction de transition inverse.)

La fonction de transition pour le mode d'agrégat en déplacement n'est pas autorisée NULL comme nouvelle valeur d'état. Si la fonction de transition inverse renvoie NULL, c'est pris comme une indication que la fonction inverse ne peut pas inverser le calcul d'état pour cette entrée particulière et donc que le calcul d'agrégat sera fait depuis le début à partir du début de l'échelle. Cette convention permet l'utilisation du mode d'agrégat en déplacement dans des situations où il existe certains cas peu courants où il serait difficile d'inverser la valeur d'état courante.

Si aucune implémentation des agrégats en déplacement n'est fournie, l'agrégat peut toujours être utilisé avec des échelles en déplacement mais PostgreSQL devra recalculer l'agrégat complet à partir du débat du déplacement de l'échelle. Notez que si l'agrégat supporte ou non le mode d'agrégat en déplacement, PostgreSQL peut gérer la fin d'une échelle en déplacement sans recalcul ; ceci se fait en continuant d'ajouter de nouvelles valeurs à l'état de l'agrégat. C'est pourquoi l'utilisation d'un agrégat comme fonction de fenêtrage nécessite que la fonction finale soit en lecture seule : elle ne doit pas endommager la valeur d'état de l'agrégat, pour que l'agrégation puisse être continuée même après qu'une valeur de résultat de l'agrégat soit obtenue par un ensemble.

La syntaxe pour des agrégats d'ensemble trié permet d'utiliser `VARIADIC` pour à la fois le dernier paramètre direct et le dernier paramètre agrégé (`WITHIN GROUP`). Néanmoins, l'implémentation actuelle restreint l'utilisation de `VARIADIC` de deux façons. Tout d'abord, les agrégats d'ensemble trié peuvent seulement utiliser `VARIADIC "any"`, et pas les autres types de tableaux variadiques. Ensuite, si le dernier paramètre direct est `VARIADIC "any"`, alors il peut y avoir seulement un paramètre agrégé et il doit aussi être `VARIADIC "any"`. (Dans la représentation utilisée dans les catalogues systèmes, ces deux paramètres sont assemblés en un seul élément `VARIADIC "any"`, car `pg_proc` ne peut pas représenter des fonctions avec plus d'un argument `VARIADIC`.) Si l'agrégat est un agrégat d'ensemble hypothétique, les arguments directs qui correspondent au paramètre `VARIADIC "any"` sont les paramètres hypothétiques. Tous les paramètres précédents représentent des arguments directs supplémentaires qui ne sont pas contraint à correspondre aux arguments agrégés.

Actuellement, les agrégats d'ensemble trié neont pas besoin de supporter le mode d'agrégat en déplacement puisqu'elles ne peuvent pas être utilisées en tant que fonction de fenêtrage.

L'agrégat partiel (y compris parallélisé) n'est pas encore supporté pour les agrégats avec des ensembles de données triés. De plus, il ne sera jamais utilisé pour les appels d'agrégat incluant les clauses `DISTINCT` ou `ORDER BY` car ces sémantiques ne peuvent pas être supportées lors d'un agrégat partiel.

Exemples

Voir Section 38.12.

Compatibilité

`CREATE AGGREGATE` est une extension PostgreSQL. Le standard SQL ne fournit pas de fonctions d'agrégat utilisateur.

Voir aussi

ALTER AGGREGATE, DROP AGGREGATE

CREATE CAST

CREATE CAST — Définir un transtypage

Synopsis

```
CREATE CAST (type_source AS type_cible)
    WITH FUNCTION nom_fonction [ (type_argument [, ...]) ]
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (type_source AS type_cible)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (type_source AS type_cible)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

Description

`CREATE CAST` définit un transtypage. Un transtypage spécifie l'opération de conversion entre deux types de données. Par exemple :

```
SELECT CAST(42 AS float8);
```

convertit la constante entière 42 en `float8` en appelant une fonction précédemment définie, `float8(int4)` dans le cas présent (si aucun transtypage convenable n'a été défini, la conversion échoue).

Deux types peuvent être *coercibles binairement*, ce qui signifie que le transtypage peut être fait « gratuitement » sans invoquer aucune fonction. Ceci impose que les valeurs correspondantes aient la même représentation interne. Par exemple, les types `text` et `varchar` sont coercibles binairement dans les deux sens. La coercibilité binaire n'est pas forcément une relation symétrique. Par exemple, le transtypage du type `xml` au type `text` peut être fait gratuitement dans l'implémentation actuelle, mais l'opération inverse nécessite une fonction qui fasse au moins une validation syntaxique. (Deux types qui sont coercibles binairement dans les deux sens sont aussi appelés binairement compatibles.)

Vous pouvez définir un transtypage comme *transtypage I/O* en utilisant la syntaxe `WITH INOUT`. Un transtype I/O est effectué en appelant la fonction de sortie du type de données source, et en passant la chaîne résultante à la fonction d'entrée du type de données cible. Dans la plupart des cas, cette fonctionnalité évite d'avoir à écrire une fonction de transtypage séparée pour la conversion. Un transtypage I/O agit de la même façon qu'un transtypage standard basé sur une fonction. Seule l'implémentation diffère.

Un transtypage peut être appelé explicitement. Par exemple : `CAST(`*`x`*` AS `*`nomtype`*`)` ou *`x`*`::`*`nomtype`*.

Si le transtypage est marqué `AS ASSIGNMENT` (NDT : à l'affectation), alors son appel peut être implicite lors de l'affectation d'une valeur à une colonne du type de donnée cible. Par exemple, en supposant que `foo.f1` soit une colonne de type `text` :

```
INSERT INTO foo (f1) VALUES (42);
```

est autorisé si la conversion du type `integer` vers le type `text` est indiquée `AS ASSIGNMENT`. Dans le cas contraire, c'est interdit. Le terme de *transtypage d'affectation* est utilisé pour décrire ce type de conversion.

Si la conversion est marquée `AS IMPLICIT`, alors elle peut être appelée implicitement dans tout contexte, soit par une affectation soit en interne dans une expression (nous utilisons généralement le terme *conversion implicite* pour décrire ce type de conversion.) Par exemple, voici une requête :

```
SELECT 2 + 4.0;
```

L'analyseur marque au début les constantes comme étant de type `integer` et `numeric` respectivement. Il n'existe pas d'opérateur `integer + numeric` dans les catalogues systèmes mais il existe un opérateur `numeric + numeric`. La requête sera un succès si une conversion de `integer` vers `numeric` est disponible et marquée `AS IMPLICIT` -- ce qui est le cas. L'analyseur appliquera la conversion implicite et résoudra la requête comme si elle avait été écrite de cette façon :

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

Maintenant, les catalogues fournissent aussi une conversion de `numeric` vers `integer`. Si cette conversion était marquée `AS IMPLICIT` -- mais ce n'est pas le cas -- alors l'analyseur devra choisir entre l'interprétation ci-dessus et son alternative (la conversion de la constante `numeric` en un `integer`) et appliquer l'opérateur `integer + integer`. Comme il n'a aucune information qui lui permettrait de choisir le meilleur moyen, il abandonne et déclare la requête comme étant ambigüe. Le fait qu'une seule des conversions est indiquée comme implicite est le moyen par lequel nous apprenons à l'analyseur de préférer la première solution (c'est-à-dire de transformer une expression `numeric-and-integer` en `numeric`) ; il n'y a pas d'autre moyen.

Il est conseillé d'être conservateur sur le marquage du caractère implicite des transtypages. Une surabondance de transtypages implicites peut conduire PostgreSQL à interpréter étrangement des commandes, voire à se retrouver dans l'incapacité totale de les résoudre parce que plusieurs interprétations s'avèrent envisageables. Une bonne règle est de ne réaliser des transtypages implicites que pour les transformations entre types de la même catégorie générale et qui préservent l'information. Par exemple, la conversion entre `int2` et `int4` peut être raisonnablement implicite mais celle entre `float8` et `int4` est probablement réservée à l'affectation. Les transtypages inter-catégories, tels que de `text` vers `int4`, sont préférablement exécutés dans le seul mode explicite.

> **Note**
>
> Il est parfois nécessaire, pour des raisons de convivialité ou de respect des standards, de fournir plusieurs transtypages implicites sur un ensemble de types de données. Ceux-ci peuvent alors entraîner des ambiguités qui ne peuvent être évitées, comme ci-dessus. L'analyseur possède pour ces cas une heuristique de secours s'appuyant sur les *catégories de types* et les *types préférés*, qui peut aider à fournir le comportement attendu dans ce genre de cas. Voir CREATE TYPE pour plus de détails.

Pour créer un transtypage, il faut être propriétaire du type source ou destination et avoir le droit `USAGE` sur l'autre type. Seul le superutilisateur peut créer un transtypage binairement compatible (une erreur sur un tel transtypage peut aisément engendrer un arrêt brutal du serveur).

Paramètres

`typesource`

Le nom du type de donnée source du transtypage.

typecible

Le nom du type de donnée cible du transtypage.

nom_fonction[(*type_argument* [, ...])]

La fonction utilisée pour effectuer la conversion. Le nom de la fonction peut être qualifié du nom du schéma. Si ce n'est pas le cas, la fonction est recherchée dans le chemin des schémas. Le type de données résultant de la fonction doit correspondre au type cible du transtypage. Ses arguments sont explicités ci-dessous. Si aucune liste d'arguments n'est spécifiée, le nom de la fonction doit être unique dans son schéma.

WITHOUT FUNCTION

Indication d'une compatibilité binaire entre le type source et le type cible pour qu'aucune fonction ne soit requise pour effectuer la conversion.

WITH INOUT

Inique que le transtypage est un transtypage I/O, effectué en appelant la fonction de sortie du type de données source, et en passant la chaîne résultante à la fonction d'entrée du type de données cible.

AS ASSIGNMENT

Lors d'une affectation, l'invocation du transtypage peut être implicite.

AS IMPLICIT

L'invocation du transtypage peut être implicite dans tout contexte.

Les fonctions de transtypage ont un à trois arguments. Le premier argument est du même type que le type source ou doit être compatible avec ce type. Le deuxième argument, si fourni, doit être de type `integer`. Il stocke le modificateur de type associé au type de destination, ou `-1` en l'absence de modificateur. Le troisième argument, si fourni, doit être de type `boolean`. Il vaut `true` si la conversion est explicite, `false` dans le cas contraire. Bizarrement, le standard SQL appelle des comportements différents pour les transtypages explicites et implicites dans certains cas. Ce paramètre est fourni pour les fonctions qui implémentent de tel transtypages. Il n'est pas recommandé de concevoir des types de données utilisateur entrant dans ce cas de figure.

Le type de retour d'une fonction de transtypage doit être identique ou coercible binairement avec le type cible du transtypage.

En général, un transtypage correspond à des types source et destination différents. Cependant, il est permis de déclarer un transtypage entre types source et destination identiques si la fonction de transtypage a plus d'un argument. Cette possibilité est utilisée pour représenter dans le catalogue système des fonctions de transtypage agissant sur la longueur d'un type. La fonction nommée est utilisée pour convertir la valeur d'un type à la valeur du modificateur de type fournie par le second argument.

Quand un transtypage concerne des types source et destination différents et que la fonction a plus d'un argument, le transtypage et la conversion de longueur du type destination sont faites en une seule étape. Quand une telle entrée n'est pas disponible, le transtypage vers un type qui utilise un modificateur de type implique deux étapes, une pour convertir les types de données et la seconde pour appliquer le modificateur.

Le transtypage du ou vers le type d'un domaine n'a actuellement pas d'effet. Transtyper d'un ou vers un domaine utilise le transtypage associé avec son type sous-jacent.

Notes

`DROP CAST` est utilisé pour supprimer les transtypages utilisateur.

Pour convertir les types dans les deux sens, il est obligatoire de déclarer explicitement les deux sens.

Il est n'est pas nécessaire habituellement de créer des conversions entre des types définis par l'utilisateur et des types de chaîne standards (`text`, `varchar` et`char(`*n*`)`, pas plus que pour des types définis par l'utilisateur définis comme entrant dans la catégorie des chaînes). PostgreSQL fournit un transtypage I/O automatique pour cela. Ce transtypage automatique vers des types chaînes est traité comme des transtypages d'affectation, alors que les transtypages automatiques à partir de types chaîne sont de type explicite seulement. Vous pouvez changer ce comportement en déclarant votre propre conversion pour remplacer une conversion automatique. La seule raison usuelle de le faire est de vouloir rendre l'appel de la conversion plus simple que le paramétrage standard (affectation seulement ou explicite seulement). Une autre raison envisageable est de vouloir que la conversion se comporte différement de la fonction I/O du type ; mais c'est suffisamment déroutant pour que vous y pensiez à deux fois avant de le faire. (Un petit nombre de types internes ont en fait des comportements différents pour les conversions, principalement à cause des besoins du standard SQL.)

Bien que cela ne soit pas requis, il est recommandé de suivre l'ancienne convention de nommage des fonctions de transtypage en fonction du type de données de destination. Beaucoup d'utilisateurs sont habitués à convertir des types de données à l'aide d'une notation de style fonction, c'est-à-dire *`nom_type`*(*`x`*). En fait, cette notation n'est ni plus ni moins qu'un appel à la fonction d'implantation du transtypage ; sa gestion n'est pas spécifique à un transtypage. Le non-respect de cette convention peut surprendre certains utilisateurs. Puisque PostgreSQL permet de surcharger un même nom de fonction avec différents types d'argument, il n'y a aucune difficulté à avoir plusieurs fonctions de conversion vers des types différents qui utilisent toutes le même nom de type destination.

> **Note**
>
> En fait, le paragraphe précédent est une sursimplification : il existe deux cas pour lesquels une construction d'appel de fonction sera traitée comme une demande de conversion sans qu'il y ait correspondance avec une fonction réelle. Si un appel de fonction *`nom`*(*`x`*) ne correspond pas exactement à une fonction existante, mais que *`nom`* est le nom d'un type de données et que `pg_cast` fournit une conversion compatible binairement vers ce type à partir du type *`x`*, alors l'appel sera construit à partir de la conversion compatible binairement. Cette exception est faite pour que les conversions compatibles binairement puissent être appelées en utilisant la syntaxe fonctionnelle, même si la fonction manque. De ce fait, s'il n'y pas d'entrée dans `pg_cast` mais que la conversion serait à partir de ou vers un type chaîne, l'appel sera réalisé avec une conversion I/O. Cette exception autorise l'appel de conversion I/O en utilisant la syntaxe fonctionnelle.

> **Note**
>
> Il existe aussi une exception à l'exception : le transtypage I/O convertissant des types composites en types chaîne de caractères ne peut pas être appelé en utilisant la syntaxe fonctionnelle, mais doit être écrite avec la syntaxe de transtypage explicite (soit `CAST` soit `::`). Cette exception a été ajoutée car, après l'introduction du transtypage I/O automatique, il était trop facile de provoquer par erreur une telle conversion alors que l'intention était de référencer une fonction ou une colonne.

Exemples

Création d'un transtypage d'affectation du type `bigint` vers le type `int4` à l'aide de la fonction `int4(bigint)` :

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS
 ASSIGNMENT;
```

(Ce transtypage est déjà prédéfini dans le système.)

Compatibilité

La commande `CREATE CAST` est conforme à SQL à ceci près que SQL ne mentionne pas les types binairement compatibles et les arguments supplémentaires pour les fonctions d'implantation. `AS IMPLICIT` est aussi une extension PostgreSQL.

Voir aussi

CREATE FUNCTION, CREATE TYPE, DROP CAST

CREATE COLLATION

CREATE COLLATION — définit une nouvelle collation

Synopsis

```
CREATE COLLATION [ IF NOT EXISTS ] nom (
    [ LOCALE = locale, ]
    [ LC_COLLATE = lc_collate, ]
    [ LC_CTYPE = lc_ctype, ]
    [ PROVIDER = provider, ]
    [ DETERMINISTIC = boolean, ]
    [ VERSION = version ]
)
CREATE COLLATION [ IF NOT EXISTS ] nom FROM collation_existante
```

Description

`CREATE COLLATION` définit une nouvelle collation utilisant la configuration de locale du système d'exploitation spécifiée ou par copie d'une collation existante.

Pour pouvoir créer une collation, vous devez posséder le privilège `CREATE` sur le schéma de destination.

Paramètres

`IF NOT EXISTS`

Ne renvoie pas d'erreur si une collation du même nom existe déjà. Une note est affichée dans ce cas. Veuillez noter qu'il n'y a aucune garantie que la collation existante ait quelque rapport que ce soit avec la collation qui aurait été créée.

nom

Le nom de la collation. Le nom de la collation peut être qualifié par le schéma. Si ce n'est pas le cas, la collation est définie dans le schéma courant. Le nom de la collation doit être unique au sein de ce schéma. (Le catalogue système peut contenir des collations de même nom pour d'autres encodages, mais ces dernières sont ignorées si l'encodage de la base de données ne correspond pas).

locale

Ceci est un raccourci pour positionner d'un même coup `LC_COLLATE` et `LC_CTYPE`. Si vous spécifiez cela, vous ne pouvez plus spécifier aucun de ces deux paramètres-ci.

lc_collate

Utilise la locale système spécifiée comme catégorie de locale de `LC_COLLATE`.

lc_ctype

Utilise la locale système spécifiée comme catégorie de locale de `LC_CTYPE`.

provider

Spécifie le fournisseur à utiliser pour les services de locale associés à cette collation. Les valeurs possibles sont : `icu`, , `libc`. `libc` est la valeur par défaut. Le choix disponible dépend du système d'exploitation ainsi que des options de compilation.

`DETERMINISTIC`

Indique si la collation doit utiliser des comparaisons déterministes. La valeur par défaut est vraie. Une comparaison déterministe considère que les chaînes de caractères qui ne sont pas équivalentes à l'octet près ne sont pas égales, même si la comparaison logique a pour résultat qu'elles sont égales. PostgreSQL départage les égalités en utilisant une comparaison à l'octet. Une comparaison non déterministe peut faire que la collation est, par exemple, insensible à la casse ou aux accents. Pour cela, vous devez choisir une option `LOCALE` appropriée *et* indiquer ici que la collation est non déterministe.

Les collations non déterministes sont supportées uniquement par le fournisseur ICU.

version

Spécifie le texte de la version à stocker avec la collation. Normalement, ce paramètre devrait être omis, ce qui fera que la version sera calculée en fonction de la version courante de la collation telle que fournie par le système d'exploitation. Cette option est prévue pour être utilisée par `pg_upgrade` pour copier la version depuis une installation existante.

Voir aussi ALTER COLLATION pour savoir comment gérer les incompatibilités de version de collations.

collation_existante

Le nom d'une collation existante à copier. La nouvelle collation aura les mêmes propriétés que celle copiée, mais ce sera un objet indépendant.

Notes

`CREATE COLLATION` acquiert un verrou `SHARE ROW EXCLUSIVE`, qui est exclusif avec lui-même, sur le catalogue système `pg_collation`, de sorte qu'une seule commande `CREATE COLLATION` peut s'exécuter à la fois.

Utilisez `DROP COLLATION` pour supprimer une collation définie par l'utilisateur.

Voir Section 24.2.2.3 pour plus d'informations sur la création de collations.

Lors de l'utilisation du fournisseur de collation `libc`, la locale doit être applicable à l'encodage actuel de la base de données. Voir CREATE DATABASE pour les règles précises.

Exemples

Créer une collation à partir de la locale système `fr_FR.utf8` (en supposant que l'encodage de la base courante est `UTF8`):

```
CREATE COLLATION french (locale = 'fr_FR.utf8');
```

Pour créer une collation en utilisant le fournisseur ICU utilisant l'ordre de tri du carnet de téléphone allemand :

```
CREATE COLLATION german_phonebook (provider = icu, locale = 'de-u-
co-phonebk');
```

Créer une collation à partir d'une collation existante :

```
CREATE COLLATION german FROM "de_DE";
```

Ceci peut être pratique pour pouvoir utiliser dans des applications des noms de collation indépendants du système d'exploitation.

Compatibilité

Dans le standard SQL se trouve un ordre `CREATE COLLATION`, mais il est limité à la copie d'une collation existante. La syntaxe de création d'une nouvelle collation est une extension PostgreSQL.

Voir également

ALTER COLLATION, DROP COLLATION

CREATE CONVERSION

CREATE CONVERSION — Définir une nouvelle conversion d'encodage

Synopsis

```
CREATE [ DEFAULT ] CONVERSION nom
    FOR codage_source TO codage_dest FROM nom_fonction
```

Description

`CREATE CONVERSION` définit une nouvelle conversion entre les encodages de caractères.

Les conversions marquées `DEFAULT` peuvent être utilisées pour automatiser une conversion d'encodage entre le client et le serveur. Pour cela, deux conversions, de l'encodage A vers l'encodage B *et* de l'encodage B vers l'encodage A, doivent être définies.

Pour créer une conversion, il est nécessaire de posséder les droits `EXECUTE` sur la fonction et `CREATE` sur le schéma de destination.

Paramètres

`DEFAULT`

La clause `DEFAULT` indique une conversion par défaut entre l'encodage source et celui de destination. Il ne peut y avoir, dans un schéma, qu'une seule conversion par défaut pour un couple d'encodages.

`nom`

Le nom de la conversion. Il peut être qualifié du nom du schéma. Dans la cas contraire, la conversion est définie dans le schéma courant. Le nom de la conversion est obligatoirement unique dans un schéma.

`codage_source`

Le nom de l'encodage source.

`codage_dest`

Le nom de l'encodage destination.

`nom_fonction`

La fonction utilisée pour réaliser la conversion. Son nom peut être qualifié du nom du schéma. Dans le cas contraire, la fonction est recherchée dans le chemin.

La fonction a la signature suivante :

```
conv_proc(
    integer,  -- ID encodage source
    integer,  -- ID encodage destination
    cstring,  -- chaîne source (chaîne C terminée par un
 caractère nul)
    internal, -- destination (chaîne C terminée par un caractère
 nul)
```

```
    integer,  -- longueur de la chaîne source
    boolean   -- si true, ne renvoie pas une erreur si la
 conversion échoue
) RETURNS integer;
```

La valeur renvoyée est le nombre d'octets de la source qui ont été convertis avec succès. Si le dernier argument vaut false, la fonction doit renvoyer une erreur en cas d'entrée invalide, et la valeur de retour est toujours égale à la longueur de la chaîne source.

Notes

Ni l'encodage source ni l'encodage destination ne peuvent être `SQL_ASCII`, car le comportement du serveur pour les cas impliquant l'encodage `SQL_ASCII` « encoding » est codé en dur.

`DROP CONVERSION` est utilisé pour supprimer une conversion utilisateur.

Il se peut que les droits requis pour créer une conversion soient modifiées dans une version ultérieure.

Exemples

Création d'une conversion de l'encodage `UTF8` vers l'encodage `LATIN1` en utilisant `mafonc` :

```
CREATE CONVERSION maconv FOR 'UTF8' TO 'LATIN1' FROM mafonc;
```

Compatibilité

`CREATE CONVERSION` est une extension PostgreSQL. Il n'existe pas d'instruction `CREATE CONVERSION` dans le standard SQL. Par contre, il existe une instruction `CREATE TRANSLATION` qui est très similaire dans son but et sa syntaxe.

Voir aussi

ALTER CONVERSION, CREATE FUNCTION, DROP CONVERSION

CREATE DATABASE

CREATE DATABASE — Créer une nouvelle base de données

Synopsis

```
CREATE DATABASE nom
    [ WITH ] [ OWNER [=] nom_utilisateur ]
             [ TEMPLATE [=] modèle ]
             [ ENCODING [=] codage ]
             [ STRATEGY [=] stratégie ]
             [ LOCALE [=] locale ]
             [ LC_COLLATE [=] lc_collate ]
             [ LC_CTYPE [=] lc_ctype ]
             [ ICU_LOCALE [=] locale_icu ]
             [ LOCALE_PROVIDER [=] fournisseur_locale ]
             [ COLLATION_VERSION = version_collation ]
             [ TABLESPACE [=] tablespace ]
             [ ALLOW_CONNECTIONS [=] connexion_autorisee ]
             [ CONNECTION LIMIT [=] limite_connexion ]
             [ IS_TEMPLATE [=] est_template ] ]
             [ OID [=] oid ]
```

Description

`CREATE DATABASE` crée une nouvelle base de données.

Pour créer une base de données, il faut être superutilisateur ou avoir le droit spécial `CREATEDB`. Voir à ce sujet CREATE USER.

Par défaut, la nouvelle base de données est créée en clonant la base système standard `template1`. Un modèle différent peut être utilisé en écrivant `TEMPLATE` *`nom`*. En particulier, la clause `TEMPLATE template0` permet de créer une base de données vierge (une base où aucun objet défini par un utilisateur n'existe et où les objets système n'ont pas été modifiés) qui ne contient que les objets standards pré-définis dans la version de PostgreSQL utilisée. C'est utile pour ne pas copier les objets locaux ajoutés à `template1`.

Paramètres

`nom`

Le nom de la base de données à créer.

`nom_utilisateur`

Le nom de l'utilisateur propriétaire de la nouvelle base de données ou `DEFAULT` pour l'option par défaut (c'est-à-dire le nom de l'utilisateur qui exécute la commande). Pour créer une base de données dont le propriétaire est un autre rôle, vous devez être un membre direct ou direct de ce rôle, ou être un superutilisateur.

`modèle`

Le nom du modèle squelette de la nouvelle base de données ou `DEFAULT` pour le modèle par défaut (`template1`).

codage

Le jeu de caractères de la nouvelle base de données. Peut-être une chaîne (par exemple `'SQL_ASCII'`), un nombre de jeu de caractères de type entier ou `DEFAULT` pour le jeu de caractères par défaut (en fait, celui de la base modèle). Les jeux de caractères supportés par le serveur PostgreSQL sont décrits dans Section 24.3.1. Voir ci-dessous pour des restrictions supplémentaires.

stratégie

Stratégie à utiliser pour la création d'un nouvelle base de données. Si la stratégie `WAL_LOG` est utilisée, la base sera copiée bloc par bloc et chaque bloc sera écrit séparément dans les journaux de transactions. C'est la stratégie la plus efficace dans le cas où la base est petit et, de ce fait, c'est la stratégie par défaut. La stratégie plus ancienne, `FILE_COPY`, est aussi disponible. Cette stratégie écrit un petit enregistrement dans le journal pour chaque tablespace utilisé par la base de données cible. Chaque enregistrement représente la copie d'un répertoire entier vers un nouvel emplacement au niveau du système de fichiers. Bien que cela réduise fortement le volume des journaux de transactions, tout spécialement si la base modèle est volumineuse, cela force aussi le système à exécuter un checkpoint avant et après la création de la nouvelle base. Dans certaines situations, ceci pourrait avoir un impact négatif visible sur les performances globales du système.

locale

C'est un raccourci permettant de configurer en une fois `LC_COLLATE` et `LC_CTYPE`.

> **Astuce**
>
> Les autres paramètres de locales, à savoir lc_messages, lc_monetary, lc_numeric et lc_time, ne sont pas fixés par base de données et ne sont donc pas configurés par cette commande. Si vous voulez le faire pour une base spécifique, vous pouvez utiliser `ALTER DATABASE ... SET`.

lc_collate

L'ordre de tri (`LC_COLLATE`) à utiliser dans la nouvelle base. Ceci affecte l'odre de tri appliqué aux chaînes, par exemple dans des requêtes avec ORDER BY, ainsi que l'ordre utilisé dans les index sur les colonnes texte. Le comportement par défaut est de choisir l'ordre de tri de la base de données modèle. Voir ci-dessous pour les restrictions supplémentaires.

lc_ctype

La classification du jeu de caractères (`LC_CTYPE`) à utiliser dans la nouvelle base. Ceci affecte la catégorisation des caractères, par exemple minuscule, majuscule et chiffre. Le comportement par défaut est d'utiliser la classification de la base de données modèle. Voir ci-dessous pour les restrictions supplémentaires.

locale_icu

Précise l'identifiant ICU de la locale si le fournisseur de locale ICU est utilisé.

fournisseur_locale

Indique le fournisseur à utiliser pour la collation par défaut dans cette base. Les valeurs possibles sont : `icu`, `libc`. `libc` est la valeur par défaut. Les choix disponibles dépendent du système d'exploitation et des options de construction.

version_collation

Indique la chaîne de version de la collation à enregistrer dans la base. Normalement, ceci devrait être omis, ce qui fera que la version sera calculée à partir de la version actuelle de la collation

de la base, telle qu'elle était fournie par le système d'exploitation. Cette option a pour but d'être utilisée par `pg_upgrade` en codant la version d'une installation existante.

Voir aussi ALTER DATABASE pour savoir comment gérer les différences de version dans la collation.

`tablespace`

Le nom du tablespace associé à la nouvelle base de données ou `DEFAULT` pour le tablespace de la base de données modèle. Ce tablespace est celui par défaut pour les objets créés dans cette base de données. Voir CREATE TABLESPACE pour plus d'informations.

`allowconn`

À false, personne ne peut se connecter à cette base de données. La valeur par défaut est true, ce qui permet les connexions (sauf restriction par d'autres mécanismes, comme `GRANT`/`REVOKE CONNECT`).

`limite_connexion`

Le nombre de connexions concurrentes à la base de données. -1 (valeur par défaut) signifie qu'il n'y a pas de limite.

`istemplate`

À true, cette base de données peut être clonée par tout utilisateur ayant l'attribut `CREATEDB` ; à false, seuls les superutilisateurs ou le propriétaire de la base de données peuvent la cloner.

`oid`

L'identifiant d'objet à utiliser pour la nouvelle base de données. Si ce paramètre n'est pas spécifié, PostgreSQL choisira automatiquement un OID convenable. Ce paramètre a principalement pour but une utilisation interne par pg_upgrade, et seul pg_upgrade peut indiquer une valeur inférieure à 16384.

L'ordre des paramètres optionnels n'a aucune importance.

Notes

La commande `CREATE DATABASE` ne peut pas être exécutée à l'intérieur d'un bloc de transactions.

Les erreurs sur la ligne « ne peut initialiser le répertoire de la base de données » (« could not initialize database directory » dans la version originale) sont le plus souvent dues à des droits insuffisants sur le répertoire de données, à un disque plein ou à un autre problème relatif au système de fichiers.

L'instruction `DROP DATABASE` est utilisée pour supprimer la base de données.

Le programme createdb est un enrobage de cette commande fourni par commodité.

Les paramètres de configuration au niveau base de données, configurés avec `ALTER DATABASE`) et les droits sur la base (configurés avec `GRANT`) ne sont pas copiés à partir de la base de données modèle.

Bien qu'il soit possible de copier une base de données autre que `template1` en spécifiant son nom comme modèle, cela n'est pas (encore) prévu comme une fonctionnalité « `COPY DATABASE` » d'usage général. La limitation principale est qu'aucune autre session ne peut être connectée à la base modèle pendant sa copie. `CREATE DATABASE` échouera s'il y a une autre connexion au moment de son exécution ; sinon, les nouveaux connexions à la base modèle seront verrouillées jusqu'à la fin de la commande `CREATE DATABASE`. La Section 23.3 fournit plus d'informations à ce sujet.

L'encodage du jeu de caractère spécifié pour la nouvelle base de données doit être compatible avec les paramètre de locale (`LC_COLLATE` et `LC_CTYPE`). Si la locale est `C` (ou de la même façon `POSIX`),

alors tous les encodages sont autorisés. Pour d'autres paramètres de locale, il n'y a qu'un encodage qui fonctionnera correctement. (Néanmoins, sur Windows, l'encodage UTF-8 peut être utilisée avec toute locale.) `CREATE DATABASE` autorisera les superutilisateurs à spécifier l'encodage `SQL_ASCII` quelque soit le paramètre locale mais ce choix devient obsolète et peut occasionner un mauvais comportement des fonctions sur les chaînes si des données dont l'encodage n'est pas compatible avec la locale sont stockées dans la base.

Les paramètres d'encodage et de locale doivent correspondre à ceux de la base modèle, excepté quand la base template0 est utilisée comme modèle. La raison en est que d'autres bases de données pourraient contenir des données qui ne correspondent pas à l'encodage indiqué, ou pourraient contenir des index dont l'ordre de tri est affecté par `LC_COLLATE` et `LC_CTYPE`. Copier ces données peut résulter en une base de données qui est corrompue suivant les nouveaux paramètres. `template0`, par contre, ne contient aucun index pouvant être affecté par ces paramètres.

Il n'existe actuellement aucune option pour utiliser une locale de base avec des comparaisons non déterministes (voir `CREATE COLLATION` pour une explication). Si c'est nécessaire, alors les collations par colonne devront être utilisées.

L'option `CONNECTION LIMIT` n'est qu'approximativement contraignante ; si deux nouvelles sessions commencent sensiblement en même temps alors qu'un seul « connecteur » à la base est disponible, il est possible que les deux échouent. De plus, les superutilisateurs et les processus worker ne sont pas soumis à cette limite.

Exemples

Créer une nouvelle base de données :

```
CREATE DATABASE lusiadas;
```

Créer une base de données `ventes` possédée par l'utilisateur `app_ventes` utilisant le tablespace `espace_ventes` comme espace par défaut :

```
CREATE DATABASE ventes OWNER app_ventes TABLESPACE espace_ventes;
```

Pour créer une base `music` avec une locale différente :

```
CREATE DATABASE music
    LOCALE 'sv_SE.utf8'
    TEMPLATE template0;
```

Dans cet exemple, la clause `TEMPLATE template0` est nécessaire si la locale spécifiée est différente de celle de `template1`. (Sinon, préciser explicitement la locale est redondant.)

Pour créer une base `music2` avec une locale différente et et un jeu de caractère différent :

```
+CREATE DATABASE music2
    LOCALE 'sv_SE.iso885915'
    ENCODING LATIN9
    TEMPLATE template0;
```

La locale et l'encodage spécifiés doivent correspondre, ou une erreur sera levée.

Veuillez noter que les noms de locale sont spécifiques au système d'exploitation, par conséquent la commande précédente pourrait ne pas fonctionner de la même façon partout.

Compatibilité

Il n'existe pas d'instruction `CREATE DATABASE` dans le standard SQL. Les bases de données sont équivalentes aux catalogues, dont la création est définie par l'implantation.

Voir aussi

ALTER DATABASE, DROP DATABASE

CREATE DOMAIN

CREATE DOMAIN — Définir un nouveau domaine

Synopsis

```
CREATE DOMAIN nom [AS] type_donnee
    [ COLLATE collation ]
    [ DEFAULT expression ]
    [ contrainte [ ... ] ]

où contrainte est :

[ CONSTRAINT nom_contrainte ]
{ NOT NULL | NULL | CHECK (expression) }
```

Description

`CREATE DOMAIN` crée un nouveau domaine. Un domaine est essentiellement un type de données avec des contraintes optionnelles (restrictions sur l'ensemble de valeurs autorisées). L'utilisateur qui définit un domaine devient son propriétaire.

Si un nom de schéma est donné (par exemple, `CREATE DOMAIN monschema.mondomaine ...`), alors le domaine est créé dans le schéma spécifié. Sinon, il est créé dans le schéma courant. Le nom du domaine doit être unique parmi les types et domaines existant dans son schéma.

Les domaines permettent d'extraire des contraintes communes à plusieurs tables et de les regrouper en un seul emplacement, ce qui en facilite la maintenance. Par exemple, plusieurs tables pourraient contenir des colonnes d'adresses email, toutes nécessitant la même contrainte de vérification (CHECK) permettant de vérifier que le contenu de la colonne est bien une adresse email. Définissez un domaine plutôt que de configurer la contrainte individuellement sur chaque table.

Pour pouvoir créer un domaine, vous devez avoir le droit `USAGE` sur le type sous-jacent.

Paramètres

nom

Le nom du domaine à créer (éventuellement qualifié du nom du schéma).

type_donnees

Le type de données sous-jacent au domaine. Il peut contenir des spécifications de tableau.

collation

Un collationement optionnel pour le domaine. Si aucun collationnement n'est spécifié, le domaine a le même comportement pour le collationement que son type sous-jacent.

`DEFAULT` *expression*

La clause `DEFAULT` permet de définir une valeur par défaut pour les colonnes d'un type de données du domaine. La valeur est une expression quelconque sans variable (les sous-requêtes ne sont pas autorisées). Le type de données de l'expression par défaut doit correspondre à celui du domaine. Si la valeur par défaut n'est pas indiquée, alors il s'agit de la valeur NULL.

L'expression par défaut est utilisée dans toute opération d'insertion qui ne spécifie pas de valeur pour cette colonne. Si une valeur par défaut est définie sur une colonne particulière, elle surcharge toute valeur par défaut du domaine. De même, la valeur par défaut surcharge toute valeur par défaut associée au type de données sous-jacent.

`CONSTRAINT` *`nom_contrainte`*

Un nom optionnel pour une contrainte. S'il n'est pas spécifié, le système en engendre un.

`NOT NULL`

Les valeurs de ce domaine sont protégées comme les valeurs NULL. Cependant, voir les notes ci-dessous.

`NULL`

Les valeurs de ce domaine peuvent être NULL. C'est la valeur par défaut.

Cette clause a pour seul but la compatibilité avec les bases de données SQL non standard. Son utilisation est découragée dans les applications nouvelles.

`CHECK (`*`expression`*`)`

Les clauses `CHECK` spécifient des contraintes d'intégrité ou des tests que les valeurs du domaine doivent satisfaire. Chaque contrainte doit être une expression produisant un résultat booléen. `VALUE` est obligatoirement utilisé pour se référer à la valeur testée. Les expressions qui renvoient TRUE ou UNKNOWN réussissent. Si l'expression produit le résultat FALSE, une erreur est rapportée et la valeur n'est pas autorisée à être convertie dans le type du domaine.

Actuellement, les expressions `CHECK` ne peuvent ni contenir de sous-requêtes ni se référer à des variables autres que `VALUE`.

Quand un domaine dispose de plusieurs contraintes `CHECK`, elles seront testées dans l'ordre alphabétique de leur nom. (Les versions de PostgreSQL antérieures à la 9.5 n'utilisaient pas un ordre particulier pour la vérification des contraintes `CHECK`.)

Notes

Les contraintes de domaine, tout particulièrement `NOT NULL`, sont vérifiées lors de la conversion d'une valeur vers le type du domaine. Il est possible qu'une colonne du type du domaine soit lue comme un NULL bien qu'il y ait une contrainte spécifiant le contraire. Par exemple, ceci peut arriver dans une requête de jointure externe si la colonne de domaine est du côté de la jointure qui peut être NULL. En voici un exemple :

```
INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE
 false));
```

Le sous-SELECT vide produira une valeur NULL qui est considéré du type du domaine, donc aucune vérification supplémentaire de la contrainte n'est effectuée, et l'insertion réussira.

Il est très difficile d'éviter de tels problèmes car l'hypothèse générale du SQL est qu'une valeur NULL est une valeur valide pour tout type de données. Une bonne pratique est donc de concevoir les contraintes du domaine pour qu'une valeur NULL soit acceptée, puis d'appliquer les contraintes `NOT NULL` aux colonnes du type du domaine quand cela est nécessaire, plutôt que de l'appliquer au type du domaine lui-même.

PostgreSQL suppose que les conditions des contraintes `CHECK` sont immuables, c'est-à-dire qu'elles produisent toujours les mêmes résultats pour les mêmes valeurs d'entrée. Cette supposition justifie que

l'examen des contraintes CHECK est effectué seulement quand une valeur est initialement convertie vers le type domaine, et pas à d'autres moments. (C'est essentiellement le même traitement que les contraintes CHECK s'appliquant aux tables, comme décrit dans Section 5.4.1.)

Un exemple typique contrevenant à cette supposition consiste à faire référence à une fonction définie par l'utilisateur dans l'expression CHECK, puis de modifier le comportement de cette fonction. PostgreSQL n'interdit pas cela, mais il ne pourra pas remarquer qu'il y a des valeurs stockées dans le type du domaine qui seraient en violation de la contrainte CHECK. Cette situation peut ainsi provoquer l'échec du rechargement d'une sauvegarde faite par export. La méthode recommandée pour mener à bien ce type de changement consiste à supprimer la contrainte (en utilisant ALTER DOMAIN), à changer la définition de la fonction, puis à remettre la contrainte, ce qui la testera sur les données stockées.

Exemples

Créer le type de données code_postal_us, et l'utiliser dans la définition d'une table. Un test d'expression rationnelle est utilisé pour vérifier que la valeur ressemble à un code postal US valide :

```
CREATE DOMAIN code_postal_us AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE courrier_us (
  id_adresse SERIAL PRIMARY KEY,
  rue1 TEXT NOT NULL,
  rue2 TEXT,
  rue3 TEXT,
  ville TEXT NOT NULL,
  code_postal code_postal_us NOT NULL
);
```

Compatibilité

La commande CREATE DOMAIN est conforme au standard SQL.

Voir aussi

ALTER DOMAIN, DROP DOMAIN

CREATE EVENT TRIGGER

CREATE EVENT TRIGGER — définir un nouveau trigger sur événement

Synopsis

```
CREATE EVENT TRIGGER nom
  ON evenement
  [ WHEN variable_filtre IN (valeur_filtre [, ... ]) [ AND ... ] ]
  EXECUTE { FUNCTION | PROCEDURE } nom_fonction()
```

Description

`CREATE EVENT TRIGGER` crée un nouveau trigger sur événement. À chaque fois que l'événement désigné intervient et que la condition `WHEN` associée au trigger est satisfaite, la fonction du trigger est exécutée. Pour une introduction générale aux triggers sur événement, voir Chapitre 40. L'utilisateur qui crée un trigger sur événement devient son propriétaire.

Paramètres

nom

Le nom à donner au nouveau trigger. Ce nom doit être unique sur la base de données.

evenement

Le nom de l'événement qui déclenche un appel à la fonction donnée. Voir Section 40.1 pour plus d'informations sur les noms d'événements.

variable_filtre

Le nom d'une variable utilisée pour filtrer les événements. Ceci rend possible de restreindre l'exécution du trigger sur un sous-ensemble des cas dans lesquels ceci est supporté. Actuellement la seule valeur autorisée pour *variable_filtre* est `TAG`.

valeur_filtre

Une liste de valeurs pour la *variable_filtre* associée, pour laquelle le trigger sera déclenché. Pour `TAG`, cela signifie une liste de balises de commande (par exemple `'DROP FUNCTION'`).

nom_fonction

Une fonction fournie par un utilisateur, déclarée ne prendre aucun argument et renvoyant le type de données `event_trigger`.

Dans la syntaxe de `CREATE EVENT TRIGGER`, les mots clés `FUNCTION` et `PROCEDURE` sont équivalents mais la fonction référencée doit dans tous les cas être une fonction, et non pas une procédure. L'utilisation du mot-clé `PROCEDURE` est ici historique et dépréciée.

Notes

Seuls les superutilisateurs peuvent créer des triggers sur événement.

Les triggers sur événement sont désactivées en mode simple utilisateur (voir postgres). Si un trigger sur événement erroné désactive la base de données à tel point que vous ne pouvez même pas supprimer le trigger, redémarrez le serveur en mode simple utilisateur et vous pourrez enfin le faire.

Exemples

Empêche l'exécution de toute commande DDL :

```
CREATE OR REPLACE FUNCTION annule_toute_commande()
  RETURNS event_trigger
 LANGUAGE plpgsql
  AS $$
BEGIN
  RAISE EXCEPTION 'la commande % est désactivée', tg_tag;
END;
$$;

CREATE EVENT TRIGGER annule_ddl ON ddl_command_start
   EXECUTE FUNCTION annule_toute_commande();
```

Compatibilité

Il n'existe pas d'instruction `CREATE EVENT TRIGGER` dans le standard SQL.

Voir aussi

ALTER EVENT TRIGGER, DROP EVENT TRIGGER, CREATE FUNCTION

CREATE EXTENSION

CREATE EXTENSION — installe une nouvelle extension

Synopsis

```
CREATE EXTENSION [ IF NOT EXISTS ] nom_extension
    [ WITH ] [ SCHEMA nom_schema ]
             [ VERSION version ]
             [ CASCADE ]
```

Description

`CREATE EXTENSION` charge une nouvelle extension dans la base de donnée courante. Il ne doit pas y avoir d'extension déjà chargée portant le même nom.

L'utilisateur qui exécute `CREATE EXTENSION` devient le propriétaire de l'extension, ce qui est important pour les vérifications de droit ultérieurement, et devient aussi normalement le propriétaire de tout objet créé par le script de l'extension.

Charger une extension nécessite normalement les même droits que pour créer les objets qui le composent. Pour beaucoup d'extensions, cela signifie que l'attribut `SUPERUSER` est nécessaire. Néanmoins, si l'extension est marquée *de confiance* dans son fichier confiance, alors elle peut être installée par tout utilisateur disposant du droit `CREATE` sur la base de données actuelle. Dans ce cas, l'objet extension sera la propriété de l'utilisateur appelant, mais les objets contenus auront pour propriétaire le superutilisateur (sauf si le script de l'extension ne les affecte lui même à l'utilisateur appelant). Cette configuration donne à l'utilisateur appelant le droit de supprimer l'extension, mais pas de modifier les objets individuels qu'elle contient.

Le chargement d'une extension nécessite les mêmes droits que ceux qui permettent la création de ses objets. La plupart des extensions nécessitent ainsi des droits superutilisateur ou d'être le propriétaire de la base de donnée. L'utilisateur qui lance la commande `CREATE EXTENSION` devient alors le propriétaire de l'extension (une vérification ultérieure des droits permettra de le confirmer) et le propriétaire de chacun des objets créé par le script de l'extension.

Paramètres

`IF NOT EXISTS`

Permet de ne pas retourner d'erreur si une extension de même nom existe déjà. Un simple message d'avertissement est alors rapporté. À noter que l'extension existante n'a potentiellement aucun lien avec l'extension qui aurait pu être créée.

`nom_extension`

Le nom de l'extension à installer. PostgreSQL créera alors l'extension en utilisant les instructions du fichier de contrôle `SHAREDIR/extension/`*`nom_extension`*`.control`.

`nom_schema`

Le nom du schéma dans lequel installer les objets de l'extension, en supposant que l'extension permette de déplacer ses objets dans un autre schéma. Le schéma en question doit exister au

préalable. Si ce nom n'est pas spécifié et que le fichier de contrôle de l'extension ne spécifie pas de schéma, le schéma par défaut en cours sera utilisé.

Si l'extension indique un paramètre `schema` dans son fichier contrôle, alors ce schéma ne peut pas être surchargé avec une clause `SCHEMA`. Habituellement, une erreur est levée si une clause `SCHEMA` est indiquée et qu'elle entre en conflit avec le paramètre `schema` de l'extension. Néanmoins, si la clause `CASCADE` est aussi indiquée, alors *`nom_schema`* est ignoré s'il y a conflit. Le *`nom_schema`* indiqué sera utilisé pour l'installation de toute extension qui ne précise pas `schema` dans son fichier contrôle.

Rappelez-vous que l'extension en soit n'est pas considérée comme étant dans un schéma. Les extensions ont des noms non qualifiés qui doivent être uniques au niveau de la base de données. Par contre, les objets appartenant à l'extension peuvent être dans des schémas.

`version`

La version de l'extension à installer. Il peut s'agir d'un identifiant autant que d'une chaîne de caractère. La version par défaut est celle spécifiée dans le fichier de contrôle de l'extension.

`CASCADE`

Installe automatiquement toute extension non déjà présente dont cette extension dépend. Leurs dépendances sont aussi automatiquement installées, récursivement. La clause `SCHEMA`, si elle est indiquée, s'applique à toutes les extensions installées de cette façon. Les autres options de l'instruction ne sont pas appliquées aux extensions créées automatiquement. En particulier, leurs versions par défaut sont toujours sélectionnées.

Notes

Avant d'utiliser la commande `CREATE EXTENSION` pour charger une extension dans une base de données, il est nécessaire d'installer les fichiers qui l'accompagnent. Les informations de Modules supplémentaires fournis permettent d'installer les extensions fournies avec PostgreSQL.

Les extensions disponibles à l'installation sur le serveur peuvent être identifiées au moyen des vues systèmes `pg_available_extensions` et `pg_available_extension_versions`.

> **Attention**
>
> Installer une extension en tant que superutilisateur nécessite d'avoir confiance dans le fait que l'auteur de l'extension a écrit le script d'installation avec la sécurité en tpete. Il n'est pas particulièrement compliqué pour un utilisateur ayant de mauvaises intentions de créer des objets de type cheval de Troie qui compromèteront une exécution ultérieure d'un script d'extension mal écrit, permettant à l'utilisateur d'acquérir des droits superutilisateur. Néanmoins, les objets chevaux de Troie sont seulement dangereux s'ils se trouvent dans le `search_path` lors de l'exécution du script, signifiant qu'ils sont dans le schéma d'installation de l'extension ou dans le schéma d'une extension dont l'extension installée dépend. De ce fait, une bonne règle lors de la gestion d'extensions dont les scripts n'ont pas été validés est de les installer seulement dans des schémas pour lesquels le droit CREATE n'a pas été et ne sera jamais donné à des utilisateurs qui ne bénéficient pas d'une confiance complète. De même pour toute extension dont elle dépend.
>
> Les extensions fournies avec PostgreSQL sont supposées être sécurisées contre les attaques à l'installation de ce type, sauf quelques unes dépendant d'autres extensions. Comme indiqué dans la documentation pour ces extensions, elles devraient être installées dans des schémas sécurisés ou installés dans les mêmes schémas que celles des extensions dont elles dépendant, ou les deux.

Pour obtenir des informations sur l'écriture de nouvelles extensions, consultez Section 38.17.

Exemples

Installer l'extension hstore dans la base de données courante, en plaçant ses objets dans le schéma `addons` :

```
CREATE EXTENSION hstore SCHEMA addons;
```

Une autre façon d'accomplir la même chose :

```
SET search_path = addons;
CREATE EXTENSION hstore;
```

Compatibilité

La commande `CREATE EXTENSION` est spécifique à PostgreSQL.

Voir aussi

ALTER EXTENSION, DROP EXTENSION

CREATE FOREIGN DATA WRAPPER

CREATE FOREIGN DATA WRAPPER — définit un nouveau wrapper de données distantes

Synopsis

```
CREATE FOREIGN DATA WRAPPER nom
    [ HANDLER fonction_handler | NO HANDLER ]
    [ VALIDATOR fonction_validation | NO VALIDATOR ]
    [ OPTIONS ( option 'valeur' [, ... ] ) ]
```

Description

`CREATE FOREIGN DATA WRAPPER` crée un nouveau wrapper de données distantes. L'utilisateur qui définit un wrapper de données distantes devient son propriétaire.

Le nom du wrapper de données distantes doit être unique dans la base de données.

Seuls les superutilisateurs peuvent créer des wrappers de données distantes.

Paramètres

nom

Le nom du wrapper de données distantes à créer.

`HANDLER` *fonction_handler*

fonction_handler est le nom d'une fonction enregistrée précédemment qui sera appelée pour récupérer les fonctions d'exécution pour les tables distantes. La fonction de gestion ne prend pas d'arguments et son code retour doit être `fdw_handler`.

Il est possible de créer un wrapper de données distantes sans fonction de gestion mais les tables distantes utilisant un tel wrapper peuvent seulement être déclarées mais pas utilisées.

`VALIDATOR` *fonction_validation*

fonction_validation est le nom d'une fonction déjà enregistrée qui sera appelée pour vérifier les options génériques passées au wrapper de données distantes, ainsi que les options fournies au serveur distant, aux correspondances d'utilisateurs (*user mappings*) et aux tables distantes utilisant le wrapper de données distantes. Si aucune fonction de validation n'est spécifiée ou si `NO VALIDATOR` est spécifié, alors les options ne seront pas vérifiées au moment de la création. (Il est possible que les wrappers de données distantes ignorent ou rejettent des spécifications d'options invalides à l'exécution, en fonction de l'implémentation) La fonction de validation doit prendre deux arguments : l'un du type `text[]`, qui contiendra le tableau d'options, tel qu'il est stocké dans les catalogues systèmes, et l'autre de type `oid`, qui sera l'OID du catalogue système contenant les options. Le type de retour est inconnu ; la fonction doit rapporter les options invalides grâce à la fonction `ereport(ERROR)`.

`OPTIONS ( option 'valeur' [, ... ] )`

Cette clause spécifie les options pour le nouveau wrapper de données distantes. Les noms et valeurs d'options autorisés sont spécifiques à chaque wrapper de données distantes. Ils sont validés par la fonction de validation du wrapper de données distantes. Les noms des options doivent être uniques.

Notes

La fonctionnalité de données distantes de PostgreSQL est toujours en développement actif. L'optimisation des requêtes est basique (et plutôt laissé aux bons soins du wrapper). Du coup, il existe certainement beaucoup de possibilités en terme d'amélioration des performances.

Exemples

Créer un wrapper de données distantes `bidon` :

```
CREATE FOREIGN DATA WRAPPER bidon;
```

Créer un wrapper de données distantes `file` avec la fonction de validation `file_fdw_validator` :

```
CREATE FOREIGN DATA WRAPPER postgresql VALIDATOR
 postgresql_fdw_validator;
```

Créer un wrapper de données distantes `monwrapper` avec des options :

```
CREATE FOREIGN DATA WRAPPER monwrapper
    OPTIONS (debug 'true');
```

Compatibilité

`CREATE FOREIGN DATA WRAPPER` est conforme à la norme ISO/IEC 9075-9 (SQL/MED), à l'exception des clauses `HANDLER` et `VALIDATOR` qui sont des extensions, et des clauses `LIBRARY` et `LANGUAGE` qui ne sont pas implémentées dans PostgreSQL.

Notez, cependant, que la fonctionnalité SQL/MED n'est pas encore conforme dans son ensemble.

Voir aussi

ALTER FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE

CREATE FOREIGN TABLE

CREATE FOREIGN TABLE — crée une nouvelle table distante

Synopsis

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] nom_table ( [
  { nom_colonne type_donnee [ OPTIONS ( option
 'valeur' [, ... ] ) ] [ COLLATE collation ] [ contrainte_colonne
 [ ... ] ]
    | contrainte_table }
    [, ... ]
] )
  SERVER nom_serveur
[ OPTIONS ( option 'valeur' [, ... ] ) ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] nom_table
  PARTITION OF table_parente [ (
  { nom_colonne [ WITH OPTIONS ] [ contrainte_colonne [ ... ] ]
    | contrainte_table }
    [, ... ]
) ]
{ FOR VALUES spec_limites_partition | DEFAULT }
  SERVER nom_serveur
[ OPTIONS ( option 'value' [, ... ] ) ]

where contrainte_colonne is:

[ CONSTRAINT nom_contrainte ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT expr_defaut |
  GENERATED ALWAYS AS ( expr_generation ) STORED }

et contrainte_table vaut :

[ CONSTRAINT nom_contrainte ]
CHECK ( expression ) [ NO INHERIT ]

et spec_limites_partition vaut :

IN ( expr_limites_partition [, ...] ) |
FROM ( { expr_limites_partition | MINVALUE | MAXVALUE } [, ...] )
  TO ( { expr_limites_partition | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS literal_numeric, REMAINDER literal_numeric )
```

Description

La commande `CREATE FOREIGN TABLE` crée une nouvelle table distante dans la base de données courante. La table distante appartient à l'utilisateur qui exécute cette commande.

Si un nom de schema est spécifié (par exemple, `CREATE FOREIGN TABLE monschema.matable ...`), alors la table sera créée dans le schéma spécifié. Dans les autres cas,

elle sera créée dans le schéma courant. Le nom de la table distante doit être différent du nom des autres relations (tables, séquences, index, vues, vues matérialisées ou tables distantes) du même schéma.

La commande CREATE FOREIGN TABLE crée aussi automatiquement un type de donnée qui représente le type composite correspondant à une ligne de la table distante. En conséquence, une table distante ne peut pas avoir le même nom qu'un type de donnée existant dans le même schéma.

Si la clause PARTITION OF est spécifiée alors la table est crée comme une partition de table_parente avec les limites spécifiées.

Pour pouvoir créer une table distante, vous devez avoir le droit USAGE sur le serveur distant, ainsi que le droit USAGE sur tous les types de colonne utilisés dans la table.

Paramètres

IF NOT EXISTS

Permet de ne pas retourner d'erreur si une table distante de même nom existe déjà. Une simple notice est alors rapportée. À noter que la table distante existante n'a potentiellement aucun lien avec la table distante qui aurait pu être créée.

nom_table

Le nom de la table distante à créer. Il est aussi possible de spécifier le schéma qui contient cette table.

nom_colonne

Le nom de la colonne à créer dans cette nouvelle table distante.

type_donnee

le type de donnée de la colonne. cela peut inclure des spécificateurs de tableaux. pour plus d'information sur les types de données supportés par postgresql, se référer à Chapitre 8.

COLLATE *collation*

La clause COLLATE affecte un collationnement à la colonne (qui doit être d'un type de données acceptant le collationnement). Si ce n'est pas spécifié, le collationnement par défaut du type de données de la colonne est utilisé.

INHERITS (*table_parent* [, ...])

La clause optionnelle INHERITS indique une liste de tables à partir desquelles la nouvelle table distante hérite automatiquement de toutes les colonnes. Les tables parents sont des tables simples ou des tables distantes. Voir la forme similaire de CREATE TABLE pour plus de détails. Notez que ceci n'est pas accepté pour créer la table distante en tant que partition de la table parent s'il existe des index UNIQUE sur la table parent. (Voir aussi ALTER TABLE ATTACH PARTITION.)

PARTITION OF *table_parent* { FOR VALUES *spec_limites_partition* | DEFAULT }

Cette syntaxe peut être utilisée pour créer la table distante en tant que partition de la table parent indiquée avec les valeurs limites de la partition. Voir la syntaxe similaire de CREATE TABLE pour plus de détails.

CONSTRAINT *nom_contrainte*

Un nom optionnel pour une contrainte de colonne ou de table. Si la contrainte est violée, le nom de la contrainte est présent dans les messages d'erreur, donc des noms de contrainte comme col doit être positif peuvent être utilisés pour communiquer des informations intéressantes sur les contraintes aux applications clientes. (Les guillemets doubles sont nécessaires

pour indiquer les noms de contraintes qui contiennent des espaces.) Si un nom de contrainte n'est pas indiqué, le système en génère un.

`NOT NULL`

Interdit des valeurs NULL dans la colonne.

`NULL`

Les valeurs NULL sont autorisées pour la colonne. il s'agit du comportement par défaut.

Cette clause n'est fournie que pour des raisons de compatibilité avec les bases de données SQL non standard. Son utilisation n'est pas encouragée dans les nouvelles applications.

`CHECK ( expression ) [ NO INHERIT ]`

La clause `CHECK` précise une expression produisant un résultat booléen que chaque ligne de la table distante est attendu satisfaire. Autrement dit, l'expression doit renvoyer TRUE ou UNKNOWN, jamais FALSE, pour toutes les lignes de la table distante. Une contrainte de vérification spécifiée comme contrainte de colonne doit seulement référencer la valeur de la colonne alors qu'une expression apparaissant dans une contrainte de table peut référencer plusieurs colonnes.

Actuellement, les expressions `CHECK` ne peuvent pas contenir de sous-requêtes. Elles ne peuvent pas non plus faire référence à des variables autres que les colonnes de la ligne courante. La colonne système `tableoid` peut être référencée, mais aucune autre colonne système ne peut l'être.

Une contrainte marquée avec `NO INHERIT` ne sera pas propagée aux tables enfants.

`DEFAULT expr_defaut`

La clause `default` affecte une valeur par défaut pour la colonne dont il est l'objet. la valeur est toute expression sans variable (les sous-requêtes et les références croisées à d'autres colonnes de la même table ne sont pas autorisées). le type de données de l'expression doit correspondre au type de données de la colonne.

L'expression par défaut sera utilisée dans toute opération d'insertion qui n'indique pas de valeur pour la colonne. s'il n'y a pas de valeur par défaut pour une colonne, la valeur par défaut implicite est null.

`GENERATED ALWAYS AS ( expr_generation ) STORED`

Cette clause crée la colonne en tant que *colonne générée*. La colonne ne peut pas faire l'objet d'une écriture, et quand elle est lue, c'est le résultat de l'expression spécifiée qui est renvoyé.

Le mot-clé `STORED` est nécessaire pour indiquer que la colonne va être calculée au moment de l'écriture. (La valeur calculée sera présentée au wrapper de données distantes pour stockage et doit être renvoyée en lecture.)

L'expression de génération peut se référer à d'autres colonnes de la table, mais pas à d'autres colonnes générées. Toutes les fonctions et opérateurs qu'elle utilise doivent être immuables. Les références à d'autres tables ne sont pas autorisées.

nom_serveur

Le nom d'un serveur distant existant à utiliser pour la table distante. Pour les détails sur la définition d'un serveur, voir CREATE SERVER.

`OPTIONS ( option 'valeur' [, ...] )`

Options qui peuvent être associés à la nouvelle table distante ou à une de ses colonnes. Les noms des options autorisées et leurs valeurs sont spécifiques à chaque wrapper de données distantes et

sont validées en utilisant la fonction de validation du wrapper de données distantes. L'utilisation répétée de la même option n'est pas autorisée (bien qu'il soit possible qu'une option de table et de colonne ait le même nom).

Notes

Les contraintes sur les tables distantes (comme les clauses `CHECK` ou `NOT NULL`) ne sont pas vérifiées par le système PostgreSQL, et la plupart des wrappers de données distantes ne cherchent pas non plus à les vérifier. La contrainte est supposée être vraie. Il y aurait peu de raisons de la vérifier car elles ne s'appliqueraient qu'aux lignes insérées ou mises à jour via la table distante, et pas aux lignes modifiées d'une autre façon, comme directement sur le serveur distant. À la place, une contrainte attachée à une table distante doit représenter une contrainte vérifiée par le serveur distant.

Certains wrappers de données distantes, dont le but est très spécifique, pourraient être le seul mécanisme d'accès aux données accédées. Dans ce cas, il pourrait être approprié au wrapper de données distantes de s'assurer de la vérification de la contrainte. Mais vous ne devez pas supposer qu'un wrapper le fait, sauf si sa documentation le précise.

Bien que PostgreSQL ne tente pas de vérifier les contraintes sur les tables distantes, il suppose qu'elles sont vérifiées et les utilise pour optimiser les requêtes. S'il y a des lignes visibles dans la table distante qui ne satisfont pas une contrainte déclarée, les requêtes sur la table pourraient produire des erreurs ou des réponses incorrectes. C'est de la responsabilité de l'utilisateur de s'assurer que la définition de la contrainte correspond à la réalité.

> **Attention**
>
> Quand une table distante est utilisée comme partition d'une table partitionnée, il existe une contrainte implicite que son contenu doit satisfaire la règle de partitionnement. Là aussi, c'est de la responsabilité de l'utilisateur que de s'assurer que cela est vrai, ce qui se fait en installer une contrainte correspondante sur le serveur distant.

Dans une table partitionnée contenant des tables distantes comme partitions, une requête `UPDATE` pouvant modifier la valeur de la clé de partitionnement peut causer le déplacement de la ligne d'une partition locale à une partition distante, à condition que le *Foreign Data Wrapper* réalise le routage de la ligne. Néanmoins, il n'est actuellement pas possible de déplacer une ligne d'une partition distante vers une autre partition. Une requête `UPDATE` qui devra le faire échouera à cause de la contrainte de partitionnement, en supposant que c'est correctement assurée par le serveur distant.

Des considérations similaires s'appliquent aux colonnes générées. Les colonnes générées stockées sont calculées au moment de l'insertion et des mises à jour sur le serveur PostgreSQL local, et passés au wrapper de données distantes pour les écrire dans le stockage distant, mais il n'est pas garanti qu'une requête sur la table distante renvoie des valeurs pour les colonnes générées en cohérence avec l'expression de génération. A nouveau, cela peut engendrer des résultats de requête incorrects.

Exemples

Créer une table distante `films` qui sera parcourue via le serveur `serveur_film` :

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
```

```
)
SERVER serveur_films;
```

Créer une table distante `measurement_y2016m07`, qui sera accédée au travers du serveur `server_07`, comme une partition de la table partitionnée par intervalles `measurement` :

```
CREATE FOREIGN TABLE measurement_y2016m07
    PARTITION OF measurement FOR VALUES FROM ('2016-07-01') TO
 ('2016-08-01')
    SERVER server_07;
```

Compatibilité

La commande `CREATE FOREIGN TABLE` est conforme au standard SQL. Toutefois, tout comme la commande `CREATE TABLE`, l'usage de la contrainte `NULL` et des tables distantes sans colonnes sont autorisés. La possibilité de spécifier des valeurs par défaut pour les colonnes est aussi une extension de PostgreSQL. L'héritage de table, dans la forme définie par PostgreSQL, n'est pas standard.

Voir aussi

ALTER FOREIGN TABLE, DROP FOREIGN TABLE, CREATE TABLE, CREATE SERVER, IMPORT FOREIGN SCHEMA

CREATE FUNCTION

CREATE FUNCTION — Définir une nouvelle fonction

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION
    nom ( [ [ modearg ] [ nomarg ] typearg  [ { DEFAULT |
 = } expression_par_defaut ] [, ...] ] ) ] )
    [ RETURNS type_ret
      | RETURNS TABLE ( nom_colonne type_colonne [, ...] ) ]
  { LANGUAGE nom_lang
    | TRANSFORM { FOR TYPE nom_type } [, ... ]
    | WINDOW
    | { IMMUTABLE | STABLE | VOLATILE }
    | [ NOT ] LEAKPROOF
    | { CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT |
 STRICT }
    | { [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY
 DEFINER }
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST cout_execution
    | ROWS nb_lignes_resultat
    | SUPPORT fonction_support
    | SET parametre { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'fichier_obj', 'symbole_lien'
    | corps_sql
  } ...
```

Description

`CREATE FUNCTION` définit une nouvelle fonction. `CREATE OR REPLACE FUNCTION` crée une nouvelle fonction ou la remplace si elle existe déjà. Pour pouvoir créer une fonction, l'utilisateur doit avoir le droit `USAGE` sur le langage associé.

Si un nom de schéma est précisé, la fonction est créée dans le schéma indiqué. Sinon, elle est créée dans le schéma courant. Le nom de la nouvelle fonction ne peut pas correspondre à celui d'une fonction ou procédure existante avec les mêmes types d'arguments en entrée dans le même schéma. Toutefois, les fonctions et procédures de types d'arguments différents peuvent partager le même nom (ceci est appelé *surcharge*).

Pour remplacer la définition actuelle d'une fonction existante, `CREATE OR REPLACE FUNCTION` est utilisé. Il n'est pas possible de changer le nom ou les types d'argument d'une fonction de cette façon (cela crée une nouvelle fonction distincte). De même, `CREATE OR REPLACE FUNCTION` ne permet pas de modifier le type retour d'une fonction existante. Pour cela, il est nécessaire de supprimer et de recréer la fonction. (Lors de l'utilisation de paramètres `OUT`, cela signifie que le type d'un paramètre `OUT` ne peut être modifié que par la suppression de la fonction.)

Quand `CREATE OR REPLACE FUNCTION` est utilisé pour remplacer une fonction existante, le propriétaire et les droits de la fonction ne changent pas. Toutes les autres propriétés de la fonction se voient affectées les valeurs spécifiées dans la commande ou implicites pour les autres. Vous devez être le propriétaire de la fonction pour la remplacer ou être un membre du rôle propriétaire de la fonction.

En cas de suppression et de recréation d'une fonction, la nouvelle fonction n'est pas la même entité que l'ancienne ; il faut supprimer les règles, vues, triggers, etc. qui référencent l'ancienne fonction.

`CREATE OR REPLACE FUNCTION` permet de modifier la définition d'une fonction sans casser les objets qui s'y réfèrent. De plus, `ALTER FUNCTION` peut être utilisé pour modifier la plupart des propriétés supplémentaires d'une fonction existante.

L'utilisateur qui crée la fonction en devient le propriétaire.

Pour pouvoir créer une fonction, vous devez avoir le droit `USAGE` sur les types des arguments et de la valeur de retour.

La lecture de Section 38.3 fournit des informations supplémentaires sur l'écriture de fonctions.

Paramètres

nom

Le nom de la fonction à créer (éventuellement qualifié du nom du schéma).

modearg

Le mode d'un argument : `IN`, `OUT`, `INOUT` ou `VARIADIC`. En cas d'omission, la valeur par défaut est `IN`. Seuls des arguments `OUT` peuvent suivre un argument `VARIADIC`. Par ailleurs, des arguments `OUT` et `INOUT` ne peuvent pas être utilisés en même temps que la notation `RETURNS TABLE`.

nomarg

Le nom d'un argument. Quelques langages (incluant SQL et PL/pgSQL) permettent d'utiliser ce nom dans le corps de la fonction. Pour les autres langages, le nom d'un argument en entrée est purement documentaire en ce qui concerne la fonction elle-même. Mais vous pouvez utiliser les noms d'arguments en entrée lors de l'appel d'une fonction pour améliorer la lisibilité (voir Section 4.3). Dans tous les cas, le nom d'un argument en sortie a une utilité car il définit le nom de la colonne dans la ligne résultat. (En cas d'omission du nom d'un argument en sortie, le système choisit un nom de colonne par défaut.)

argtype

Le(s) type(s) de données des arguments de la fonction (éventuellement qualifié du nom du schéma), s'il y en a. Les types des arguments peuvent être basiques, composites ou de domaines, ou faire référence au type d'une colonne.

En fonction du langage, il est possible d'indiquer des « pseudotypes », tel que `cstring`. Les pseudotypes indiquent que le type d'argument réel est soit non complètement spécifié, soit en dehors de l'ensemble des types de données ordinaires du SQL.

Il est fait référence au type d'une colonne par *`nom_table.nomcolonne`*`%TYPE`. Cette fonctionnalité peut servir à rendre une fonction indépendante des modifications de la définition d'une table.

expression_par_defaut

Une expression à utiliser en tant que valeur par défaut si le paramètre n'est pas spécifié. L'expression doit pouvoir être coercible dans le type d'argument du paramètre. Seuls les paramètres d'entrée (dont les `INOUT`) peuvent avoir une valeur par défaut. Tous les paramètres d'entrée suivant un paramètre avec une valeur par défaut doivent aussi avoir une valeur par défaut.

type_ret

Le type de données en retour (éventuellement qualifié du nom du schéma). Le type de retour peut être un type basique, composite ou de domaine, ou faire référence au type d'une colonne existante.

En fonction du langage, il est possible d'indiquer un « pseudotype », tel que `cstring`. Si la fonction ne doit pas renvoyer de valeur, on indique `void` comme type de retour.

Quand il y a des paramètres `OUT` ou `INOUT`, la clause `RETURNS` peut être omise. Si elle est présente, elle doit correspondre au type de résultat imposé par les paramètres de sortie : `RECORD` s'il y en a plusieurs, ou le type du seul paramètre en sortie.

Le modificateur `SETOF` indique que la fonction retourne un ensemble d'éléments plutôt qu'un seul.

Il est fait référence au type d'une colonne par *`nom_table.nomcolonne`*`%TYPE`.

`nom_colonne`

Le nom d'une colonne de sortie dans la syntaxe `RETURNS TABLE`. C'est une autre façon de déclarer un paramètre OUT nommé, à la différence près que `RETURNS TABLE` implique aussi `RETURNS SETOF`.

`type_colonne`

Le type de données d'une colonne de sortie dans la syntaxe `RETURNS TABLE`.

`nom_lang`

Le nom du langage d'écriture de la fonction. Peut être `SQL`, `C`, `internal` ou le nom d'un langage procédural utilisateur, par exemple `plpgsql`. La valeur par défaut est `sql` si *`corps_sql`* est indiqué. Entourer le nom de guillemets simples est une pratique obsolète et nécessite la bonne casse.

`TRANSFORM { FOR TYPE` *`nom_type`* `} [, ... ] }`

Indique la transformation s'appliquant pour un appel à la fonction. Les transformations convertissent des types de données SQL en des types de données spécifiques au langage. Voir CREATE TRANSFORM. Les implémentations des langages de procédure stockée ont une connaissance codée en dur des types internes, donc ces derniers n'ont pas besoin d'être listés ici. Si l'implémentation d'un langage de procédure ne sait pas gérer un type et qu'aucune transformation n'est fournie, il y a un retour au comportement par défaut pour les conversions des types de données mais ceci dépends de l'implémentation.

`WINDOW`

`WINDOW` indique que la fonction est une *fonction window* plutôt qu'une fonction simple. Ceci n'est à l'heure actuelle utilisable que pour les fonctions écrites en C. L'attribut `WINDOW` ne peut pas être changé lors du remplacement d'une définition de fonction existante.

`IMMUTABLE`
`STABLE`
`VOLATILE`

Ces attributs informent l'optimiseur de requêtes sur le comportement de la fonction. Un seul choix est possible. En son absence, `VOLATILE` est utilisé.

`IMMUTABLE` indique que la fonction ne peut pas modifier la base de données et qu'à arguments constants, la fonction renvoie toujours le même résultat ; c'est-à-dire qu'elle n'effectue pas de recherches dans la base de données, ou alors qu'elle utilise des informations non directement présentes dans la liste d'arguments. Si cette option est précisée, tout appel de la fonction avec des arguments constants peut être immédiatement remplacé par la valeur de la fonction.

`STABLE` indique que la fonction ne peut pas modifier la base de données et qu'à l'intérieur d'un seul parcours de la table, à arguments constants, la fonction retourne le même résultat, mais celui-ci varie en fonction des instructions SQL. Cette option est appropriée pour les fonctions dont les résultats dépendent des recherches en base, des variables de paramètres (tel que la zone

horaire courante), etc. (Ce mode est inapproprié pour les triggers AFTER qui souhaitent voir les lignes modifiées par la commande en cours.) La famille de fonctions current_timestamp est qualifiée de stable car les valeurs de ces fonctions ne changent pas à l'intérieur d'une transaction.

VOLATILE indique que la valeur de la fonction peut changer même au cours d'un seul parcours de table. Aucune optimisation ne peut donc être réalisée. Relativement peu de fonctions de bases de données sont volatiles dans ce sens ; quelques exemples sont random(), currval(), timeofday(). Toute fonction qui a des effets de bord doit être classée volatile, même si son résultat est assez prévisible. Cela afin d'éviter l'optimisation des appels ; setval() en est un exemple.

Pour des détails complémentaires, voir Section 38.7.

LEAKPROOF

LEAKPROOF indique que la fonction n'a pas d'effets de bord. Elle ne fournit aucune information sur ces arguments autrement que par sa valeur de retour. Par exemple, une fonction qui renvoie un message d'erreur pour certaines valeurs d'arguments et pas pour d'autres, ou qui inclut les valeurs des arguments dans des messages d'erreur, ne peut pas utiliser cette clause. Ceci affecte la façon dont le système exécute des requêtes contre les vues créées avec l'option security_barrier ou les tables avec la fonctionnalité RLS activée. Le système force les conditions des politiques de sécurité et les vues avec barrière de sécurité avant toute condition fournie par l'utilisateur sur la requête appelante qui contient des fonctions non sécurisées (non LEAKPROOF), pour empêcher toute exposition involontaire des données. Les fonctions et opérateurs marquées LEAKPROOF sont supposés être sûrs, et peuvent être exécutées avant les conditions des politiques de sécurité et les vues avec barrière de sécurité. De plus, les fonctions qui ne prennent pas d'arguments ou qui ne se voient pas fournies d'arguments par la vue ou la table n'ont pas besoin d'être marquées comme LEAKPROOF pour être exécutées avant les conditions de sécurité. Voir CREATE VIEW et Section 41.5. Cette option peut seulement être utilisée par un superutilisateur.

```
CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT
```

CALLED ON NULL INPUT (la valeur par défaut) indique que la fonction est appelée normalement si certains de ses arguments sont NULL. C'est alors de la responsabilité de l'auteur de la fonction de gérer les valeurs NULL.

RETURNS NULL ON NULL INPUT ou STRICT indiquent que la fonction renvoie toujours NULL si l'un de ses arguments est NULL. Lorsque ce paramètre est utilisé et qu'un des arguments est NULL, la fonction n'est pas exécutée, mais un résultat NULL est automatiquement retourné.

```
[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER
```

SECURITY INVOKER indique que la fonction est exécutée avec les droits de l'utilisateur qui l'appelle. C'est la valeur par défaut. SECURITY DEFINER spécifie que la fonction est exécutée avec les droits de l'utilisateur qui en est le propriétaire.

Le mot clé EXTERNAL est autorisé pour la conformité SQL mais il est optionnel car, contrairement à SQL, cette fonctionnalité s'applique à toutes les fonctions, pas seulement celles externes.

PARALLEL

PARALLEL UNSAFE indique que la fonction ne peut pas être exécutée dans le mode parallèle. La présence d'une fonction de ce type dans une requête SQL force un plan d'exécution en série. C'est la valeur par défaut. PARALLEL RESTRICTED indique que la fonction peut être exécutée en mode parallèle mais l'exécution est restreinte au processus principal d'exécution. PARALLEL SAFE indique que la fonction s'exécute correctement dans le mode parallèle sans restriction.

Les fonctions doivent être marquées comme non parallélisable si elles modifient l'état d'une base ou si elles font des changements sur la transaction telles que l'utilisation de sous-transactions ou si elles accèdent à des séquences ou tentent de faire des modifications persistentes aux configurations (par exemple `setval`). Elles doivent être marquées comme restreintes au parallélisme si elles accèdent aux tables temporaires, à l'état de connexion des clients, aux curseurs, aux requêtes préparées ou à un état local du moteur où le système ne peut pas synchroniser en mode parallèle (par exemple, `setseed` ne peut pas être exécuté autrement que par le processus principal car une modification réalisée par un autre processus ne pourrait pas être reflété dans le processus principal). En général, si une fonction est marquée sûre à la parallélisation alors qu'elle est restreinte ou non parallélisable ou si elle est marquée restreinte quand elle est en fait non parallélisable, elle pourrait renvoyer des erreurs ou fournir de mauvaises réponses lorsqu'elle est utilisée dans une requête parallèle. Les fonctions en langage C peuvent en théorie afficher un comportement indéfini si elles sont marquées de façon erronée car le système ne peut pas se protéger comme du code C arbitraire mais, généralement, le résultat ne sera pas pire que pour toute autre fonction. En cas de doute, les fonctions doivent être marquées comme `UNSAFE`, ce qui correspond à la valeur par défaut.

`COST` *`cout_execution`*

Un nombre positif donnant le coût estimé pour l'exécution de la fonction en unité de cpu_operator_cost. Si la fonction renvoie plusieurs lignes, il s'agit d'un coût par ligne renvoyée. Si le coût n'est pas spécifié, une unité est supposée pour les fonctions en langage C et les fonctions internes. Ce coût est de 100 unités pour les fonctions dans tout autre langage. Des valeurs plus importantes feront que le planificateur tentera d'éviter l'évaluation de la fonction aussi souvent que possible.

`ROWS` *`nb_lignes_resultat`*

Un nombre positif donnant le nombre estimé de lignes que la fonction renvoie, information utile au planificateur. Ceci est seulement autorisé pour les fonctions qui renvoient plusieurs lignes (fonctions SRF). La valeur par défaut est de 1000 lignes.

`SUPPORT` *`fonction_support`*

Le nom (optionnellement qualifié du nom du schéma) d'une *fonction de support de planification* à utiliser pour cette fonction. Voir Section 38.11 pour plus d'informations. Il faut être superutilisateur pour utiliser cette option.

`parametre`
`valeur`

La clause `SET` fait que le paramètre de configuration indiquée est initialisée avec la valeur précisée au lancement de la fonction, puis restaurée à sa valeur d'origine lors de la sortie de la fonction. `SET FROM CURRENT` sauvegarde la valeur actuelle du paramètre quand `ALTER FUNCTION` est exécuté comme valeur à appliquer lors de l'exécution de la fonction.

Si une clause `SET` est attachée à une fonction, alors les effets de la commande `SET LOCAL` exécutée à l'intérieur de la fonction pour la même variable sont restreints à la fonction : la valeur précédente du paramètre de configuration est de nouveau restaurée en sortie de la fonction. Néanmoins, une commande `SET` ordinaire (c'est-à-dire sans `LOCAL`) surcharge la clause `SET`, comme il le ferait pour une précédente commande `SET LOCAL` : les effets d'une telle commande persisteront après la sortie de la fonction sauf si la transaction en cours est annulée.

Voir SET et Chapitre 20 pour plus d'informations sur les paramètres et valeurs autorisés.

`definition`

Une constante de type chaîne définissant la fonction ; la signification dépend du langage. Cela peut être un nom de fonction interne, le chemin vers un fichier objet, une commande SQL ou du texte en langage procédural.

Il est souvent utile d'utiliser les guillemets dollar (voir Section 4.1.2.4) pour écrire le code de la fonction, au lie des la syntaxe habituelle des guillemets. Sans les guillemets dollar, tout guillemet ou antislash dans la définition de la fonction doit être échappé en les doublant.

fichier_obj, symbole_lien

Cette forme de clause AS est utilisée pour les fonctions en langage C chargeables dynamiquement lorsque le nom de la fonction dans le code source C n'est pas le même que celui de la fonction SQL. La chaîne *fichier_obj* est le nom du fichier de la bibliothèque partagée contenant la fonction C compilée et est interprété comme pour une commande LOAD. La chaîne *symbole_lien* est le symbole de lien de la fonction, c'est-à-dire le nom de la fonction dans le code source C. Si ce lien est omis, il est supposé être le même que le nom de la fonction SQL définie. Les noms C de toutes les fonctions doivent être différents, donc vous devez donner aux fonctions C surchargés des noms C différents (par exemple, utilisez les types d'arguments comme partie des noms C).

Lors d'appels répétés à CREATE FUNCTION se référant au même fichier objet, il est chargé seulement une fois par session. Pour décharger et recharger le fichier (par exemple lors du développement de la fonction), démarrez une nouvelle session.

corps_sql

Le corps d'une fonction LANGUAGE SQL. Cela peut être soit une seule requête

```
RETURN expression
```

soit un bloc

```
BEGIN ATOMIC
  statement;
  statement;
  ...
  statement;
END
```

Ceci est similaire à écrire le texte du corps de la fonction sous la forme d'une constante de chaîne (voir *définition* ci-dessus), mais il existe quelques différences : cette forme fonctionne seulement pour LANGUAGE SQL, la forme constante de chaîne fonctionne pour tous les langages. Cette forme est analysée au moment de la définition de la fonction, la forme chaîne de caractères est analysée au moment de l'exécution ; de ce fait, cette forme ne peutpas supporter les types d'argument polymorphiques et toute autre construction qui ne serait pas résolvable au moment de la définition de l'objet. Cette forme trace les dépendances entre la fonction et les objets utilisés dans le corps de la fonction, donc DROP ... CASCADE fonctionnera correctement, alors que la forme utilisant les constantes de chaînes pourraient laisser des fonctions en danger. Enfin, cette forme est plus compatible avec le standard SQL et les autres implémentations SQL.

Overloading

PostgreSQL autorise la *surcharge* des fonctions ; c'est-à-dire que le même nom peut être utilisé pour des fonctions différentes si tant est qu'elles aient des types d'arguments en entrée distincts. Que vous l'utilisiez ou non, cette capacité implique des précautions au niveau de la sécurité lors de l'appel des fonctions dans les bases de données où certains utilisateurs ne font pas confiance à d'autres utilisateurs ; voir Section 10.3.

Deux fonctions sont considérées identiques si elles partagent le même nom et les mêmes types d'argument en *entrée*, sans considération des paramètres OUT. Les déclarations suivantes sont, de fait, en conflit :

```
CREATE FUNCTION truc(int) ...
```

```
CREATE FUNCTION truc(int, out text) ...
```

Des fonctions ayant des listes de types d'arguments différents ne seront pas considérées comme en conflit au moment de leur création, mais si des valeurs par défauts sont fournies, elles peuvent se retrouver en conflit au moment de l'invocation. Considérez par exemple :

```
CREATE FUNCTION truc(int) ...
CREATE FUNCTION truc(int, int default 42) ...
```

Un appel `truc(10)` échouera à cause de l'ambiguïté sur la fonction à appeler.

Notes

La syntaxe SQL complète des types est autorisé pour déclarer les arguments en entrée et la valeur de sortie d'une fonction. Néanmoins, les modificateurs du type de la fonction (par exemple le champ précision pour un `numeric`) sont ignorés par `CREATE FUNCTION`. Du coup, par exemple, `CREATE FUNCTION foo (varchar(10)) ...` est identique à `CREATE FUNCTION foo (varchar) ...`.

Lors du remplacement d'une fonction existante avec `CREATE OR REPLACE FUNCTION`, il existe des restrictions sur le changement des noms de paramètres. Vous ne pouvez pas modifier le nom de paramètre en entrée déjà affecté mais vous pouvez ajouter des noms aux paramètres qui n'en avaient pas. S'il y a plus d'un paramètre en sortie, vous ne pouvez pas changer les noms des paramètres en sortie car cela changera les noms de colonne du type composite anonyme qui décrit le résultat de la fonction. Ces restrictions sont là pour assurer que les appels suivants à la fonction ne s'arrêtent pas de fonctionner lorsqu'elle est remplacée.

Exemples

Ajouter deux entiers en utilisant une fonction SQL :

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

La même fonction écrite dans un style plus conforme avec le standard SQL, utilisant des noms d'argument et un corps pas placé entre guillemets :

```
CREATE FUNCTION add(a integer, b integer) RETURNS integer
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT
    RETURN a + b;
```

Incrémenter un entier, en utilisant le nom de l'argument, dans PL/pgSQL :

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS
 $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

Renvoyer un enregistrement contenant plusieurs paramètres en sortie :

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

La même chose, en plus verbeux, avec un type composite nommé explicitement :

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Une autre façon de renvoyer plusieurs colonnes est d'utiliser une fonction `TABLE` :

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Toutefois, une fonction `TABLE` est différente des exemples précédents parce qu'elle retourne en fait un *ensemble* d'enregistrements, pas juste un enregistrement.

Écrire des fonctions `SECURITY DEFINER` en toute sécurité

Parce qu'une fonction `SECURITY DEFINER` est exécutée avec les droits de l'utilisateur qui en est le propriétaire, une certaine attention est nécessaire pour s'assurer que la fonction ne peut pas être utilisée de façon maline. Pour la sécurité, search_path doit être configuré pour exclure les schémas modifiables par des utilisateurs indignes de confiance. Cela empêche des utilisateurs malveillants de créer des objets (par exemple tables, fonctions et opérateurs) qui masquent les objets utilisés par la fonction. Dans ce sens, le schéma des tables temporaires est particulièrement important car il est le premier schéma parcouru et qu'il est normalement modifiable par tous les utilisateurs. Une solution consiste à forcer le parcours de ce schéma en dernier lieu. Pour cela, on écrit `pg_temp` comme dernière entrée de `search_path`. La fonction suivante illustre une utilisation sûre :

```
CREATE FUNCTION verifie_motdepasse(unom TEXT, motpasse TEXT)
RETURNS BOOLEAN AS $$
DECLARE ok BOOLEAN;
BEGIN
        -- Effectuer le travail sécurisé de la fonction.
        SELECT  (motdepasse = $2) INTO ok
        FROM    motsdepasse
        WHERE   nomutilisateur = $1;

        RETURN ok;
END;
$$  LANGUAGE plpgsql
    SECURITY DEFINER
```

```
    -- Configure un search_path sécurisée : les schémas de
 confiance, puis 'pg_temp'.
    SET search_path = admin, pg_temp;
```

Le but de cette fonction est d'accéder à une table `admin.motsdepasse`. Mais sans la clause `SET` ou avec une clause `SET` mentionnant uniquement `admin`, la fonction pourrait être transformée en créant une table temporaire nommée `motsdepasse`.

Avant PostgreSQL 8.3, la clause `SET` n'était pas disponible, donc les anciennes fonctions pouvaient contenir un code assez complexe pour sauvegarder, initialiser puis restaurer un paramètre comme `search_path`. La clause `SET` est plus simple à utiliser dans ce but.

Un autre point à garder en mémoire est que, par défaut, le droit d'exécution est donné à `PUBLIC` pour les fonctions nouvellement créées (voir Section 5.7 pour plus d'informations). Fréquemment, vous souhaiterez restreindre l'utilisation d'une fonction « security definer » à seulement quelques utilisateurs. Pour cela, vous devez révoquer les droits `PUBLIC` puis donner le droit d'exécution aux utilisateurs sélectionnés. Pour éviter que la nouvelle fonction soit accessible à tous pendant un court moment, créez-la et initialisez les droits dans une même transaction. Par exemple :

```
BEGIN;
CREATE FUNCTION verifie_motdepasse(unom TEXT, motpasse TEXT) ...
 SECURITY DEFINER;
REVOKE ALL ON FUNCTION verifie_motdepasse(unom TEXT, motpasse TEXT)
 FROM PUBLIC;
GRANT EXECUTE ON FUNCTION verifie_motdepasse(unom TEXT, motpasse
 TEXT) TO admins;
COMMIT;
```

Compatibilité

Une commande `CREATE FUNCTION` est définie dans le standard SQL. L'implémentation PostgreSQL peut être utilisée d'une façon compatible mais a de nombreuses extensions. De même, le standard SQL indique un certain nombre de fonctionnalités optionnelles qui ne sont pas implémentées dans PostgreSQL.

Voici les problèmes importants de compatibilité :

- `OR REPLACE` est une extension PostgreSQL.
- Pour la compatibilité avec d'autres systèmes de bases de données, *`mode_arg`* peut être écrit soit avant soit après *`nom_arg`*. Mais seule la première façon est conforme au standard.
- Pour les valeurs par défaut des paramètres, le standard SQL précise seulement la syntaxe avec le mot clé `DEFAULT`. La syntaxe avec = est utilisée dans T-SQL et Firebird.
- La clause `SETOF` est une extension PostgreSQL.
- Seul `SQL` est standrdisé comme langage.
- Tous les autres attributs en dehors de `CALLED ON NULL INPUT` et `RETURNS NULL ON NULL INPUT` ne font pas partie du standard.
- Pour le corps des fonctions `LANGUAGE SQL`, le langage SQL indique seulement la forme de *`corps_sql`*.

Les fonctions simples en `LANGUAGE SQL` peuent être écrites d'une façon qui respecte le standard et reste portable avec les autres implémentations. Les fonctions plus complexes, utilisant des

fonctionnalités avancées, des attributs d'optimisation ou d'autres langages seront nécessairement spécifiques à PostgreSQL d'une façon significative.

Voir aussi

ALTER FUNCTION, DROP FUNCTION, GRANT, LOAD, REVOKE

CREATE GROUP

CREATE GROUP — Définir un nouveau rôle de base de données

Synopsis

```
CREATE GROUP nom [ [ WITH ] option [ ... ] ]

où option peut être :

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT limite_connexion
    | [ ENCRYPTED ] PASSWORD 'mot_de_passe' | PASSWORD NULL
    | VALID UNTIL 'dateheure'
    | IN ROLE nom_role [, ...]
    | IN GROUP nom_role [, ...]
    | ROLE nom_role [, ...]
    | ADMIN nom_role [, ...]
    | USER nom_role [, ...]
    | SYSID uid
```

Description

`CREATE GROUP` est désormais un alias de CREATE ROLE.

Compatibilité

Il n'existe pas d'instruction `CREATE GROUP` dans le standard SQL.

Voir aussi

CREATE ROLE

CREATE INDEX

CREATE INDEX — Définir un nouvel index

Synopsis

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] nom ]
 ON [ ONLY ] nom_table [ USING méthode ]
    ( { nom_colonne | ( expression ) } [ COLLATE collation ]
 [ classeop [ ( parametre_classeop = valeur [, ... ] ) ] ] [ ASC |
 DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ INCLUDE ( nom_colonne [, ...] ) ]
    [ NULLS [ NOT ] DISTINCT ]
    [ WITH ( parametre_stockage = valeur [, ... ] ) ]
    [ TABLESPACE nom_espacelogique ]
    [ WHERE prédicat ]
```

Description

`CREATE INDEX` construit un index sur le (ou les) colonne(s) spécifiée(s) de la relation spécifiée, qui peut être une table ou une vue matérialisée. Les index sont principalement utilisés pour améliorer les performances de la base de données (bien qu'une utilisation inappropriée puisse produire l'effet inverse).

Les champs clé pour l'index sont spécifiés à l'aide de noms des colonnes ou par des expressions écrites entre parenthèses. Plusieurs champs peuvent être spécifiés si la méthode d'indexation supporte les index multi-colonnes.

Un champ d'index peut être une expression calculée à partir des valeurs d'une ou plusieurs colonnes de la ligne de table. Cette fonctionnalité peut être utilisée pour obtenir un accès rapide à des données obtenues par transformation des données basiques. Par exemple, un index calculé sur `upper(col)` autorise la clause `WHERE upper(col) = 'JIM'` à utiliser un index.

PostgreSQL fournit les méthodes d'indexation B-tree (NDT : arbres balancés), hash (NDT : hachage), GiST (NDT : arbres de recherche généralisés), SP-GiST, GIN et BRIN. Il est possible, bien que compliqué, de définir des méthodes d'indexation utilisateur.

Lorsque la clause `WHERE` est présente, un *index partiel* est créé. Un index partiel est un index ne contenant des entrées que pour une portion d'une table, habituellement la portion sur laquelle l'indexation est la plus utile. Par exemple, si une table contient des ordres facturés et d'autres qui ne le sont pas, et que les ordres non facturés n'occupent qu'une petite fraction du total de la table, qui plus est fréquemment utilisée, les performances sont améliorées par la création d'un index sur cette portion. Une autre application possible est l'utilisation de la clause `WHERE` en combinaison avec `UNIQUE` pour assurer l'unicité sur un sous-ensemble d'une table. Voir Section 11.8 pour plus de renseignements.

L'expression utilisée dans la clause `WHERE` peut ne faire référence qu'à des colonnes de la table sous-jacente, mais elle peut utiliser toutes les colonnes, pas uniquement celles indexées. Actuellement, les sous-requêtes et les expressions d'agrégats sont aussi interdites dans la clause `WHERE`. Les mêmes restrictions s'appliquent aux champs d'index qui sont des expressions.

Toutes les fonctions et opérateurs utilisés dans la définition d'index doivent être « immutable » (NDT : immuable), c'est-à-dire que leur résultat ne doit dépendre que de leurs arguments et jamais d'une influence externe (telle que le contenu d'une autre table ou l'heure). Cette restriction permet de s'assurer que le comportement de l'index est strictement défini. Pour utiliser une fonction utilisateur dans une

expression d'index ou dans une clause `WHERE`, cette fonction doit être marquée immutable lors de sa création.

Paramètres

`UNIQUE`

Le système vérifie la présence de valeurs dupliquées dans la table à la création de l'index (si des données existent déjà) et à chaque fois qu'une donnée est ajoutée. Les tentatives d'insertion ou de mises à jour qui résultent en des entrées dupliquées engendrent une erreur.

Des restrictions supplémentaires s'appliquent quand des index uniques sont appliquées aux tables partitionnées. Voir CREATE TABLE.

`CONCURRENTLY`

Quand cette option est utilisée, PostgreSQL construira l'index sans prendre de verrous qui bloquent les insertions, mises à jour, suppression en parallèle sur cette table ; la construction d'un index standard verrouille les écritures (mais pas les lectures) sur la table jusqu'à la fin de la construction. Il est nécessaire d'avoir quelques connaissances avant d'utiliser cette option -- voir Construire des index en parallèle.

Pour les tables temporaires, `CREATE INDEX` est toujours non concurrent car aucune autre session n'y a accès, et la création d'index non concurrent est moins coûteuse.

`IF NOT EXISTS`

Ne renvoie pas une erreur si une relation existe avec le même nom. Un message est renvoyé dans ce cas. Notez qu'il n'existe pas de garantie que l'index existant ressemble à celui qui aurait été créé. Le nom d'index est requis quand `IF NOT EXISTS` est spécifié.

`INCLUDE`

La clause optionnelle `INCLUDE` indique une liste de colonnes qui seront incluses dans l'index comme des colonnes *non clés*. Une colonne non clé ne peut pas être utilisée dans la qualification d'une recherche par parcours d'index, et elle est ignorée pour la contrainte d'unicité ou d'exclusion assurée par l'index. Néanmoins, un parcours d'index couvrant peut renvoyer le contenu des colonnes non clés sans avoir à visiter la table de l'index car il est directement disponible dans l'index. De façon, l'ajout de colonnes non clés autorise l'utilisation de parcours d'index couvrants pour les requêtes qui, autrement, ne les auraient pas utilisés.

Il est conseillé de rester prudent sur l'ajout de colonnes non clés dans un index, tout spécialement pour les colonnes larges. Si un enregistrement d'un index dépasse la taille maximale autorisée pour le type de l'index l'insertion de données échouera. Dans tous les cas, les colonnes non clés dupliquent les données de la table et augmentent la taille de l'index, ralentissant potentiellement les recherches. Furthermore, B-tree deduplication is never used with indexes that have a non-key column.

Les colonnes listées dans la clause `INCLUDE` n'ont pas besoin de classes d'opérateurs appropriées. La clause peut contenir les colonnes dont les types de données n'ont pas de classes d'opérateurs définis pour une méthode d'accès donnée.

Les expressions ne sont pas supportées comme colonnes incluses car elles ne peuvent pas être utilisées dans des parcours d'index couvrants.

Actuellement, seules les méthodes d'accès B-tree, GiST et SP-GiST tirent parti de cette fonctionnalité. Pour ces index, les valeurs des colonnes listées dans la clause `INCLUDE` sont incluses dans les enregistrements feuilles qui correspondent à des enregistrements de lignes de table, mais ne sont pas incluses dans les enregistrements de plus haut niveau.

nom

Le nom de l'index à créer. Aucun nom de schéma ne peut être inclus ici ; l'index est toujours créé dans le même schéma que sa table parent. Le nom de l'index doit être distinct du nom de toute autre relation (table, séquence, index, vue, vue matérialisée, ou table distante) pour ce schéma. Si le nom est omis, PostgreSQL choisit un nom convenable basé sur le nom de la table parent et celui des colonnes indexées.

ONLY

Indique de ne pas faire de récursion pour la création des index sur les partitions si la table est partitionnée. Par défaut, la récursion a lieu.

nom_table

Le nom de la table à indexer (éventuellement qualifié du nom du schéma).

méthode

Le nom de la méthode à utiliser pour l'index. Les choix sont `btree`, `hash`, `gist`, `spgist`, `gin` et `brin`. ou les méthodes d'accès installées par l'utilisateur comme bloom. La méthode par défaut est `btree`.

nom_colonne

Le nom d'une colonne de la table.

expression

Une expression basée sur une ou plusieurs colonnes de la table. L'expression doit habituellement être écrite entre parenthèses, comme la syntaxe le précise. Néanmoins, les parenthèses peuvent être omises si l'expression a la forme d'un appel de fonction.

collation

Le nom du collationnement à utiliser pour l'index. Par défaut, l'index utilise le collationnement déclaré pour la colonne à indexer ou le collationnement résultant de l'expression à indexer. Les index avec des collationnements spécifiques peuvent être utiles pour les requêtes qui impliquent des expressions utilisant des collationnements spécifiques.

classeop

Le nom d'une classe d'opérateurs. Voir plus bas pour les détails.

opclass_parameter

The name of an operator class parameter. See below for details.

ASC

Spécifie un ordre de tri ascendant (valeur par défaut).

DESC

Spécifie un ordre de tri descendant.

NULLS FIRST

Spécifie que les valeurs NULL sont présentées avant les valeurs non NULL. Ceci est la valeur par défaut quand `DESC` est indiqué.

`NULLS LAST`

Spécifie que les valeurs NULL sont présentées après les valeurs non NULL. Ceci est la valeur par défaut quand ASC est indiqué.

`NULLS DISTINCT`
`NULLS NOT DISTINCT`

Spécifique pour un index d'unicité si les valeurs NULL doivent être considérées comme distinctes (différentes). Par défaut, elles sont considérées distinctes, pour qu'un index d'unicité permette de stocker plusieurs valeurs NULL dans la colonne associée.

paramètre_stockage

Le nom d'un paramètre de stockage spécifique à la méthode d'indexage. Voir Paramètres de stockage des index pour les détails.

nom_espacelogique

Le tablespace dans lequel créer l'index. S'il n'est pas précisé, default_tablespace est consulté, sauf si la table est temporaire auquel cas temp_tablespaces est utilisé.

prédicat

L'expression de la contrainte pour un index partiel.

Paramètres de stockage des index

La clause WITH optionnelle spécifie des *paramètres de stockage* pour l'index. Chaque méthode d'indexage peut avoir son propre ensemble de paramètres de stockage. Les méthodes d'index B-tree, hash, GiST et SP-GiST acceptent toutes ce paramètre :

`fillfactor` (`integer`)

Le facteur de remplissage pour un index est un pourcentage qui détermine à quel point les pages d'index seront remplies par la méthode d'indexage. Pour les B-tree, les pages enfants sont remplies jusqu'à ce pourcentage lors de la construction initiale de l'index, et aussi lors de l'extension de l'index sur la droite (ajoutant les valeurs de clé les plus importantes). Si les pages deviennent ensuite totalement remplies, elles seront partagées, amenant une fragmentation de la structure sur disque de l'index. Les arbres B-tree utilisent un facteur de remplissage de 90% par défaut mais toute valeur entière comprise entre 10 et 100 peut être choisie.

Les index B-tree sur des tables où de nombreuses insertions et/ou mises à jour sont prévues peuvent bénéficier d'un facteur de remplissage plus bas lors du CREATE INDEX (suivant le chargement en masse dans la table). Les valeurs dans l'intervalle 50 - 90 peut utilement « diminuer » le *taux* de division de blocs au début de la vie de l'index B-tree (baisser ainsi le facteur de remplissage peut même diminuer le nombre absolu de divisions de blocs, bien que cet effet est fortement dépendant de la charge de travail). La technique de suppression de l'index B-Tree du bas vers le haut décrite dans Section 67.4.2 est dépendent sur la place « supplémentaire » disponible dans les blocs pour des versions « supplémentaires » de lignes, et ainsi peut être affectée par le facteur de remplissage (bien que l'effet n'est habituellement pas significatif).

Dans les autres cas spécifiques, il pourrait être utile d'augmenter le facteur de remplissage à 100 au moment du CREATE INDEX comme moyen de maximiser l'utilisation de l'espace. Vous devez seulement le considérer quand vous êtes complètement sûr que la table est statique (autrement dit qu'elle ne sera jamais affectée par des insertions ou des mises à jour). Une configuration du facteur de remplissage à 100 risque autrement de *baisser* les performances : même un petit nombre de mises à jour ou d'insertions peut causer une soudaine explosion des divisions de blocs.

Les autres méthodes d'index utilisent le facteur de remplissage de façon différente mais grossièrement identique ; le facteur de remplissage par défaut varie suivant les méthodes.

Les index B-tree acceptent aussi ce paramètre :

`deduplicate_items` (`boolean`)

Controls usage of the B-tree deduplication technique described in Section 67.4.3. Set to `ON` or `OFF` to enable or disable the optimization. (Alternative spellings of `ON` and `OFF` are allowed as described in Section 20.1.) The default is `ON`.

> **Note**
>
> Turning `deduplicate_items` off via `ALTER INDEX` prevents future insertions from triggering deduplication, but does not in itself make existing posting list tuples use the standard tuple representation.

Les index GiST acceptent en option ce paramètre :

`buffering` (`enum`)

Détermine si la technique de construction par tampon décrite dans Section 68.4.1 est utilisé pour construire l'index. À `OFF`, cette technique est désactivée. À `ON`, elle est activée. À `AUTO`, elle est initialement désactivée mais peut être activée quand la taille de l'index atteint effective_cache_size. La valeur par défaut est `AUTO`. Notez que si la construction triée est possible, elle sera utilisée à la place de la construction par tampon à moins que `buffering=ON` ne soit spécifié.

Les index GIN acceptent plusieurs paramètres supplémentaires :

`fastupdate` (`boolean`)

Ce paramètre régit l'utilisation de la technique de mise à jour rapide décrite dans Section 70.4.1. C'est un paramètre booléen : `ON` active la mise à jour rapide, `OFF` la désactive. La valeur par défaut est `ON`.

> **Note**
>
> Désactiver `fastupdate` via `ALTER INDEX` empêche les insertions futures d'aller dans la liste d'entrées d'index à traiter, mais ne nettoie pas les entrées précédentes de cette liste. Vous voudrez peut être ensuite exécuter un `VACUUM` sur la table ou exécuter la fonction `gin_clean_pending_list`, afin de garantir que la liste à traiter soit vidée.

`gin_pending_list_limit` (`integer`)

Personnalise le paramètre gin_pending_list_limit. Cette valeur est spécifiée en ko.

Les index BRIN acceptent différents paramètres :

`pages_per_range` (`integer`)

Définit le nombre de blocs de table qui sera résumé en un intervalle de blocs pour chaque entrée dans un index BRIN (voir Section 71.1 pour plus de détails). La valeur par défaut est `128`.

`autosummarize` (`boolean`)

Définit si le lancement d'un calcul de résumé doit être mis en queue pour l'intervalle de blocs précédent chaque fois qu'une insertion est détectée sur l'intervalle suivant. Voir Section 71.1.1 pour plus de détails. La valeur par défaut est `off`.

Les index GiST acceptent en plus ce paramètre :

`buffering`

Détermine si la technique de construction avec tampons décrite dans Section 68.4.1 est utilisée pour construire l'index. À `OFF`, cette technique n'est pas utilisée. À `ON`, elle est utilisée. À `AUTO`, elle est au départ désactivée mais elle est activée une fois que la taille de l'index atteint effective_cache_size. La valeur par défaut est `AUTO`.

Construire des index en parallèle

Créer un index peut interférer avec les opérations normales d'une base de données. Habituellement, PostgreSQL verrouille la table à indexer pour la protéger des écritures et construit l'index complet avec un seul parcours de la table. Les autres transactions peuvent toujours lire la table mais s'ils essaient d'insérer, mettre à jour, supprimer des lignes dans la table, elles seront bloquées jusqu'à la fin de la construction de l'index. Ceci peut avoir un effet sérieux si le système est une base en production. Les très grosses tables peuvent demander plusieurs heures pour être indexées. Même pour les petites tables, une construction d'index peut bloquer les processus qui voudraient écrire dans la table pendant des périodes longues sur un système de production.

PostgreSQL supporte la construction des index sans verrouillage des écritures. Cette méthode est appelée en précisant l'option `CONCURRENTLY` de `CREATE INDEX`. Quand cette option est utilisée, PostgreSQL doit réaliser deux parcours de table et, en plus, il doit attendre que toutes les transactions existantes qui peuvent modifier ou utiliser cet index se terminent. Du coup, cette méthode requiert plus de temps qu'une construction standard de l'index et est bien plus longue à se terminer. Néanmoins, comme cela autorise la poursuite des opérations pendant la construction de l'index, cette méthode est utile pour ajouter de nouveaux index dans un environnement en production. Bien sûr, la charge CPU et I/O supplémentaire imposée par la création de l'index peut ralentir les autres opérations.

Dans la construction en parallèle d'un index, l'index est enregistré sous la forme d'un index « invalide » dans les catalogues systèmes dans une transaction, puis les deux parcours de table interviennent dans deux transactions supplémentaires. Avant chaque parcours de table, la construction de l'index doit attendre la fin des transactions en cours qui ont modifié la table. Après le deuxième parcours, la construction doit attendre la fin de toute transactions ayant une image de base (un snapshot, voir Chapitre 13) datant d'avant le deuxième parcours pour se terminer, ceci incluant les transactions utilisées par toute phase des constructions concurrentes d'index sur les autres tables, si les index impliqués sont partiels ou ont des colonnes qui ne sont pas des références de colonne simple. Ensuite, l'index peut être marqué comme « valide » et utilisable, et la commande `CREATE INDEX` se termine. Néanmoins, même après cela, l'index pourrait ne pas être immédiatement utilisable pour les autres requêtes : dans le pire des cas, il ne peut pas être utilisé tant que des transactions datant d'avant le début de la création de l'index existent.

Si un problème survient lors du parcours de la table, comme un deadlock ou une violation d'unicité dans un index unique, la commande `CREATE INDEX` échouera mais laissera derrière un index « invalide ». Cet index sera ignoré par les requêtes car il pourrait être incomplet ; néanmoins il consommera quand même du temps lors des mises à jour de l'index. La commande `\d` de psql rapportera cet index comme `INVALID` :

```
 postgres=# \d tab
 Table "public.tab"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 col    | integer |           |          |
 Indexes:
 "idx" btree (col) INVALID
```

La méthode de récupération recommandée dans de tels cas est de supprimer l'index et de tenter de nouveau un `CREATE INDEX CONCURRENTLY`. (Une autre possibilité est de reconstruire l'index avec `REINDEX CONCURRENTLY`.)

Lors de la construction d'un index unique en parallèle, la contrainte d'unicité est déjà placée pour les autres transactions quand le deuxième parcours de table commence. Cela signifie que des violations de contraintes pourraient être rapportées dans les autres requêtes avant que l'index ne soit disponible, voire même dans des cas où la construction de l'index va échouer. De plus, si un échec survient dans le deuxième parcours, l'index « invalide » continue à forcer la contrainte d'unicité.

Les constructions en parallèle d'index avec expression et d'index partiels sont supportées. Les erreurs survenant pendant l'évaluation de ces expressions pourraient causer un comportement similaire à celui décrit ci-dessus pour les violations de contraintes d'unicité.

Les constructions d'index standards permettent d'autres constructions d'index en simultanée sur la même table mais seul une construction d'index en parallèle peut survenir sur une table à un même moment. Dans les deux cas, la modification du schéma de la table n'est pas autorisé pendant la construction de l'index. Une autre différence est qu'une commande `CREATE INDEX` normale peut être réalisée à l'intérieur d'un bloc de transactions alors que `CREATE INDEX CONCURRENTLY` ne le peut pas.

Les constructions en parallèle des index sur les tables partitionnées ne sont pas actuellement supportées. Néanmoins, vous pouvez construire l'index en parallèle sur chaque partition individuel, puis créer l'index partitionné sans CONCURRENTLY pour réduire le temps où les écritures seront bloquées sur la table partitionnée. Dans ce cas, construire l'index partitionné est une opération sur les méta-données uniquement.

Notes

Chapitre 11 présente des informations sur le moment où les index peuvent être utilisés, quand ils ne le sont pas et dans quelles situations particulières ils peuvent être utiles.

Actuellement, seules les méthodes d'indexation B-tree, GiST, GIN et BRIN supportent les index multi-colonnes. Qu'il puisse y avoir plusieurs colonnes clés est indépendant du fait que des colonnes `INCLUDE` puissent être ajoutées à l'index. Les index peuvent avoir jusqu'à 32 colonnes, ceci incluant les colonnes `INCLUDE`.(Cette limite peut être modifiée à la compilation de PostgreSQL.) Seul B-tree supporte actuellement les index uniques.

Une *classe d'opérateurs* with optional parameters peut être spécifiée pour chaque colonne d'un index. La classe d'opérateurs identifie les opérateurs à utiliser par l'index pour cette colonne. Par exemple, un index B-tree sur des entiers codés sur quatre octets utilise la classe `int4_ops`, qui contient des fonctions de comparaison pour les entiers sur quatre octets. En pratique, la classe d'opérateurs par défaut pour le type de données de la colonne est généralement suffisant. Les classes d'opérateurs trouvent leur intérêt principal dans l'existence, pour certains types de données, de plusieurs ordonnancements significatifs.

Soit l'exemple d'un type de données « nombre complexe » qui doit être classé par sa valeur absolue ou par sa partie réelle. Cela peut être réalisé par la définition de deux classes d'opérateurs pour le type de données, puis par la sélection de la classe appropriée lors de la création d'un index.

De plus amples informations sur les classes d'opérateurs sont disponibles dans Section 11.10 et dans Section 38.16.

Quand `CREATE INDEX` est appelé sur une table partitionnée, le comportement par défaut est de vérifier que toutes les partitions ont un index correspondant. Chaque partition est tout d'abord vérifiée pour déterminer si un index équivalent existe déjà. Si c'est le cas, cet index sera attaché comme index la partition avec l'index en cours de création, qui deviendra son index parent. Si aucun index correspondant n'existe, un nouvel index sera créé et attaché automatiquement. Le nom du nouvel index dans chaque partition sera déterminé comme si aucun nom d'index n'avait été spécifié dans la commande. Si l'option `ONLY` est indiquée, aucune récursion n'est réalisée et l'index est marqué invalide. (`ALTER INDEX ... ATTACH PARTITION` marque l'index comme valide une fois que toutes les partitions ont acquis l'index correspondant.) Néanmoins, notez que toute partition créée dans le futur en utilisant `CREATE TABLE ... PARTITION OF` contiendra automatiquement l'index correspondant que cette option soit spécifiée ou non.

Pour les méthodes d'indexage qui supportent les parcours ordonnés (actuellement seulement pour les B-tree), les clauses optionnelles ASC, DESC, NULLS FIRST et/ou NULLS LAST peuvent être spécifiées pour modifier l'ordre de tri normal de l'index. Comme un index ordonné peut être parcouru en avant et en arrière, il n'est habituellement pas utile de créer un index DESC sur une colonne -- ce tri est déjà disponible avec un index standard. L'intérêt de ces options se révèle avec les index multi-colonnes. Ils peuvent être créés pour correspondre à un tri particulier demandé par une requête, comme SELECT ... ORDER BY x ASC, y DESC. Les options NULLS sont utiles si vous avez besoin de supporter le comportement « nulls sort low », plutôt que le « nulls sort high » par défaut, dans les requêtes qui dépendent des index pour éviter l'étape du tri.

Le système récupère régulièrement des statistiques sur toutes les colonnes d'une table. Les index nouvellement créés et sans expression peuvent immédiatement utiliser ces statistiques pour déterminer l'utilité d'un index. Pour les nouveaux index à expression, il est nécessaire d'exécuter ANALYZE ou d'attendre que le the processus en tâche de fond autovacuum analyse la table pour générer des statistiques pour ces index.

Pour la plupart des méthodes d'indexation, la vitesse de création d'un index est dépendante du paramètre maintenance_work_mem. Une plus grande valeur réduit le temps nécessaire à la création d'index, tant qu'elle ne dépasse pas la quantité de mémoire vraiment disponible, afin d'éviter que la machine ne doive paginer.

PostgreSQL peut construire des index en utilisant plusieurs CPU pour traiter plus rapidement les lignes de la table. Cette fonctionnalité est connue sous le nom de *construction d'index parallélisée*. Pour les méthodes d'indexage qui supportent la construction d'index en parallèle (actuellement seulement les B-tree), maintenance_work_mem indique la quantité maximale de mémoire pouvant être utilisée pour chaque opération de construction d'index, quelque soit le nombre de processus workers démarrés. Habituellement, un modèle de coût détermine automatiquement le nombre de workers à exécuter.

Les constructions d'index parallélisées pourraient bénéficier d'une augmentation du maintenance_work_mem, là où une construction équivalente mais non parallélisée ne verrait que peu ou pas de bénéfices. Notez que maintenance_work_mem peut influencer le nombre de processus workers demandés car les workers parallélisés doivent avoir au moins 32 Mo provenant du maintenance_work_mem global. Il doit aussi rester 32 Mo pour le processus leader. Augmenter max_parallel_maintenance_workers pourrait permettre l'utilisation d'un plus grand nombre de workers, ce qui réduirait le temps nécessaire pour la création de l'index, à condition que cette création ne soit pas déjà freiné par les disques. Bien sûr, il doit rester suffisamment de CPU qui auraient été autrement inutilisés.

Configurer une valeur pour parallel_workers via ALTER TABLE contrôle directement le nombre de processus workers parallélisés réclamé par un CREATE INDEX sur la table. Ceci contourne complètement le modèle de coût, et empêche maintenance_work_mem d'affcter le nombre demandé de workers parallélisés. Configurer parallel_workers à 0 via ALTER TABLE désactivera les constructions d'index parallélisées sur la table dans tous les cas.

> **Astuce**
>
> Vous pourriez vouloir réinitialiser parallel_workers après l'avoir configuré pour permettre une construction d'index. Ceci évite des changements inattendus dans les plans de requêtes, vu que parallel_workers affecte *tous* les parcours parallélisés de table.

Bien que CREATE INDEX avec l'option CONCURRENTLY accepte les constructions parallélisées sans restrictions particulières, seul le premier parcours de table est réellement exécuté en parallèle.

DROP INDEX est utilisé pour supprimer un index.

Comme pour toute transaction longue, CREATE INDEX sur une table peut affecter les lignes pouvant être supprimées par un VACUUM concurrent sur toute autre table.

Les versions précédentes de PostgreSQL ont aussi une méthode d'index R-tree. Cette méthode a été supprimée car elle n'a pas d'avantages par rapport à la méthode GiST. Si `USING rtree` est indiqué, `CREATE INDEX` l'interprétera comme `USING gist` pour simplifier la conversions des anciennes bases à GiST.

Chaque processus exécutant un `CREATE INDEX` indiquera sa progression dans la vue `pg_stat_progress_create_index`. Voir Section 28.4.2 pour les détails.

Exemples

Créer un index B-tree sur la colonne `titre` dans la table `films` :

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

Pour créer un index B-tree unique sur la colonne `title` avec les colonnes incluses `director` et `rating` de la table `films` :

```
CREATE UNIQUE INDEX title_idx ON films (title) INCLUDE (director,
 rating);
```

To create a B-Tree index with deduplication disabled:

```
CREATE INDEX title_idx ON films (title) WITH (deduplicate_items =
 off);
```

Pour créer un index sur l'expression `lower(titre)`, permettant une recherche efficace quelque soit la casse :

```
CREATE INDEX ON films ((lower(titre)));
```

(dans cet exemple, nous avons choisi d'omettre le nom de l'index, donc le système choisira un nom, typiquement `films_lower_idx`.)

Pour créer un index avec un collationnement spécifique :

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

Pour créer un index avec un ordre de tri des valeurs NULL différent du standard :

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

Pour créer un index avec un facteur de remplissage différent :

```
CREATE UNIQUE INDEX idx_titre ON films (titre) WITH (fillfactor =
 70);
```

Pour créer un index GIN avec les mises à jour rapides désactivées :

```
CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH
 (fastupdate = off);
```

Créer un index sur la colonne `code` de la table `films` et donner à l'index l'emplacement du tablespace `espaceindex` :

```
CREATE INDEX code_idx ON films (code) TABLESPACE espaceindex;
```

Pour créer un index GiST sur un attribut point, de façon à ce que nous puissions utiliser rapidement les opérateurs box sur le résultat de la fonction de conversion :

```
CREATE INDEX pointloc
    ON points USING gist (box(location,location));
SELECT * FROM points
    WHERE box(location,location) && '(0,0),(1,1)'::box;
```

Pour créer un index sans verrouiller les écritures dans la table :

```
CREATE INDEX CONCURRENTLY index_quentite_ventes ON table_ventes
 (quantité);
```

Compatibilité

`CREATE INDEX` est une extension du langage PostgreSQL. Les index n'existent pas dans le standard SQL.

Voir aussi

ALTER INDEX, DROP INDEX, REINDEX, Section 28.4.2

CREATE LANGUAGE

CREATE LANGUAGE — Définir un nouveau langage procédural

Synopsis

```
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE nom
    HANDLER gestionnaire_appel [ VALIDATOR fonction_validation ]
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE nom
```

Description

`CREATE LANGUAGE` enregistre un nouveau langage procédural à une base de données PostgreSQL. En conséquence, les fonctions et procédures peuvent être définies dans ce nouveau langage.

`CREATE LANGUAGE` associe en fait le nom du langage à un ou des fonctions de gestion qui sont responsable de l'exécution des fonctions écrites dans le langage. Chapitre 42 offre de plus amples informations sur les gestionnaires de fonctions.

`CREATE OR REPLACE LANGUAGE` va soit créer un nouveau langage, soit remplacer une définition existante. Si le langage existe déjà, ses paramètres sont mis à jour suivant la commande, mais le propriétaire et les droits ne changent pas, et toutes les fonctions existantes écrite dans ce langage sont supposées être toujours valides.

Un utilisateur doit être un superutilisateur PostgreSQL pour enregistrer un nouveau langage ou modifier les paramètres d'un langage existant. Néanmoins, une fois que le langage est créé, il est possible d'affecter un propriétaire autre qu'un superutilisateur, qui pourrait le supprimer, modifier ses droits, le renommer ou lui affecter un autre propriétaire. (Néanmoins, n'affectez pas un utilisateur standard comme propriétaire d'une fonction C ; cela créerait un moyen d'escalade de droits pour cet utilisateur.

La syntaxe de `CREATE LANGUAGE` qui ne fournit pas de fonction de gestion est obsolète. Pour des raisons de compatibilité ascendante avec les anciens fichiers de sauvegarde, c'est interprété comme un `CREATE EXTENSION`. Ceci fonctionnera si le langage a été préparé sous la forme d'une extension de même nom, ce qui est la façon conventionnelle de configurer les langages de procédures.

Paramètres

`TRUSTED`

`TRUSTED` indique que le langage ne donne pas accès aux données auquel l'utilisateur n'a pas normalement accès. Si ce mot clé est omis à l'enregistrement du langage, seuls les superutilisateurs peuvent utiliser ce langage pour créer de nouvelles fonctions.

`PROCEDURAL`

Sans objet.

`nom`

Le nom du nouveau langage procédural. Il ne peut y avoir deux langages portant le même nom au sein de la base de données.

`HANDLER` *`gestionnaire_appel`*

`gestionnaire_appel` est le nom d'une fonction précédemment enregistrée. C'est elle qui est appelée pour exécuter les fonctions du langage procédural. Le gestionnaire d'appels d'un

langage procédural doit être écrit dans un langage compilé, tel que le C, avec la convention d'appel version 1 et enregistré dans PostgreSQL comme une fonction ne prenant aucun argument et retournant le type `language_handler`, type servant essentiellement à identifier la fonction comme gestionnaire d'appels.

`INLINE` *`gestionnaire_en_ligne`*

`gestionnaire_en_ligne` est le nom d'une fonction déjà enregistrée qui sera appelée pour exécuter un bloc de code anonyme (voir la commande `DO`) dans ce langage. Si aucune fonction *`gestionnaire_en_ligne`* n'est indiquée, le langage ne supporte pas les blocs de code anonymes. La fonction de gestion doit prendre un argument du type `internal`, qui sera la représentation interne de la commande `DO`, et il renverra le type `void`. La valeur de retour du gestionnaire est ignorée.

`VALIDATOR` *`fonction_validation`*

`fonction_validation` est le nom d'une fonction précédemment enregistrée. C'est elle qui est appelée pour valider toute nouvelle fonction écrite dans ce langage. Si aucune fonction de validation n'est spécifiée, alors toute nouvelle fonction n'est pas vérifiée à sa création. La fonction de validation prend obligatoirement un argument de type `oid`, OID de la fonction à créer, et renvoie par convention `void`.

Une fonction de validation contrôle généralement le corps de la fonction pour s'assurer de sa justesse syntaxique mais peut également vérifier d'autres propriétés de la fonction (l'incapacité du langage à gérer certains types d'argument, par exemple). Le signalement d'erreur se fait à l'aide de la fonction `ereport()`. La valeur de retour de la fonction est ignorée.

Notes

Utiliser `DROP LANGUAGE` pour supprimer un langage procédural.

Le catalogue système `pg_language` (voir Section 53.29) contient des informations sur les langages installés. De plus, la commande psql `\dL` liste les langages installées.

Pour créer des fonctions dans un langage procédural, l'utilisateur doit posséder le droit `USAGE` pour ce langage. Par défaut, `USAGE` est donné à `PUBLIC` (c'est-à-dire tout le monde) pour les langages de confiance. Ce droit peut être révoqué si nécessaire.

Les langages procéduraux sont installées par base. Néanmoins, un langage peut être installé dans la base de données `template1`, ce qui le rend automatiquement disponible dans toutes les bases de données créées par la suite.

Exemples

Voici une séquence minimale pour créer un langage de procédures :

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Habituellement, cela devrait être écrit dans le script de création de l'extension, et les utilisateurs feraient ceci pour installer l'extension :

```
CREATE EXTENSION plsample;
```

Compatibilité

`CREATE LANGUAGE` est une extension de PostgreSQL.

Voir aussi

ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, GRANT, REVOKE

CREATE MATERIALIZED VIEW

CREATE MATERIALIZED VIEW — définir une nouvelle vue matérialisée

Synopsis

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] nom_table
    [ (nom_colonne [, ...] ) ]
    [ USING méthode ]
    [ WITH ( paramètre_stockage [= valeur] [, ... ] ) ]
    [ TABLESPACE nom_tablespace ]
    AS requête
    [ WITH [ NO ] DATA ]
```

Description

`CREATE MATERIALIZED VIEW` définit une vue matérialisée à partir d'une requête. La requête est exécutée et utilisée pour peupler la vue à l'exécution de la commande (sauf si la clause `WITH NO DATA` est utilisé) et peut être rafraichi plus tard en utilisant `REFRESH MATERIALIZED VIEW`.

`CREATE MATERIALIZED VIEW` est similaire à `CREATE TABLE AS`, sauf qu'il se rappelle aussi de la requête utilisée pour initialiser la vue pour qu'elle puisse être rafraichie à la demande. Une vue matérialisée a plusieurs propriétés communes avec une table mais il n'y a pas de support pour les vues matérialisées temporaires.

`CREATE MATERIALIZED VIEW` nécessite le droit `CREATE` sur le schéma utilisé par la vue matérialisée.

Paramètres

`IF NOT EXISTS`

Ne renvoie pas une erreur si une vue matérialisée portant le même nom existe déjà. Un message d'avertissement est renvoyé dans ce cas. Notez qu'il n'y a aucune garantie que la vue matérialisée existante ressemble à celle qui aurait dû être créée.

`nom_table`

Le nom de la vue matérialisée (potentiellement qualifié du schéma) à créer. Le nom doit être distinct du nom des autres relations (table, séquence, index, vue, vue matérialisée, ou table externe) du même schéma.

`nom_colonne`

Le nom d'une colonne dans la nouvelle vue matérialisée. Si les noms des colonnes ne sont pas fournis, ils sont pris des noms de colonne en sortie de la requête.

`USING` *`méthode`*

Cette clause optionnelle indique la méthode d'accès à la table à utiliser pour stocker le contenu de la nouvelle vue matérialisée ; la méthode doit être une méthode d'accès de type `TABLE`. Voir Chapitre 63 pour plus d'informations. Si cette option n'est pas spécifiée, la méthode d'accès par défaut à la table est choisie pour la nouvelle vue matérialisée. Voir default_table_access_method pour plus d'informations.

`WITH ( paramètre_stockage [= valeur] [, ... ] )`

Cette clause indique les paramètres de stockage optionnels pour la nouvelle vue matérialisée ; voir Paramètres de stockage dans la documentation de CREATE TABLE pour plus d'informations. Tous les paramètres acceptés par `CREATE TABLE` sont aussi acceptés par `CREATE MATERIALIZED VIEW`. Voir CREATE TABLE pour plus d'informations.

`TABLESPACE nom_tablespace`

`nom_tablespace` est le nom du tablespace dans lequel la nouvelle vue matérialisée sera créée. S'il n'est pas indiqué, default_tablespace est consulté.

`query`

Une commande `SELECT`, `TABLE`, ou `VALUES`. Cette requête sera exécutée dans une opération restreinte au niveau sécurité. En particulier, les appels aux fonctions qui elles-même créent des tables temporaires échoueront.

`WITH [ NO ] DATA`

Cette clause indique si la vue matérialisée doit être peuplée ou non lors de sa création. Si elle ne l'est pas, la vue matérialisée sera marquée comme non parcourable et ne pourra pas être lu jusqu'à ce que `REFRESH MATERIALIZED VIEW` soit utilisé.

Compatibilité

`CREATE MATERIALIZED VIEW` est une extension PostgreSQL.

Voir aussi

ALTER MATERIALIZED VIEW, CREATE TABLE AS, CREATE VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

CREATE OPERATOR

CREATE OPERATOR — Définir un nouvel opérateur

Synopsis

```
CREATE OPERATOR nom (
    {FUNCTION|PROCEDURE} = nom_fonction
    [, LEFTARG = type_gauche ]
    [, RIGHTARG = type_droit ]
    [, COMMUTATOR = op_com ]
    [, NEGATOR = op_neg ]
    [, RESTRICT = proc_res ]
    [, JOIN = proc_join ]
    [, HASHES ] [, MERGES ]
)
```

Description

`CREATE OPERATOR` définit un nouvel opérateur, *`nom`*. L'utilisateur qui définit un opérateur en devient propriétaire. Si un nom de schéma est donné, l'opérateur est créé dans le schéma spécifié. Sinon, il est créé dans le schéma courant.

Le nom de l'opérateur est une séquence d'au plus `NAMEDATALEN`-1 (63 par défaut) caractères parmi la liste suivante :

+ - * / < > = ~ ! @ # % ^ & | ` ?

Il existe quelques restrictions dans le choix du nom :

- `--` et `/*` ne peuvent pas apparaître dans le nom d'un opérateur car ils sont pris pour le début d'un commentaire.
- Un nom d'opérateur multicaractères ne peut pas finir avec `+` ou `-` sauf si le nom contient l'un, au moins, de ces caractères :

  ~ ! @ # % ^ & | ` ?

  Par exemple, `@-` est un nom d'opérateur autorisé mais `*-` n'en est pas un. Cette restriction permet à PostgreSQL d'analyser les commandes compatibles SQL sans nécessiter d'espaces entre les lexèmes.
- Le symbole `=>` est réservé par la grammaire SQL, donc il ne peut pas être utilisé comme nom d'opérateur.

L'opérateur `!=` est remplacé par `<>` à la saisie, ces deux noms sont donc toujours équivalents.

> **Note**
>
> Les opérateurs unaires droits, aussi appelés postfix, sont obsolètes et seront supprimés dans PostgreSQL version 14.

Pour les opérateurs binaires, `LEFTARG` et `RIGHTARG` doivent être définis. Pour les opérateurs préfixes, seul `RIGHTARG` devrait être défini. La fonction *`nom_fonction`* doit avoir été précédemment définie par `CREATE FUNCTION` et doit accepter le bon nombre d'arguments (un ou deux) des types indiqués.

Dans la syntaxe de `CREATE OPERATOR`, les mot-clés `FUNCTION` et `PROCEDURE` sont équivalents mais la fonction référencée doit dans tous les cas être une fonction et non pas une procédure. L'utilisation du mot clé `PROCEDURE` est ici historique et dépréciée.

Les autres clauses spécifient des clauses optionnelles d'optimisation d'opérateur. Leur signification est détaillée dans Section 38.15.

Pour pouvoir créer un opérateur, vous devez avoir le droit `USAGE` sur le type des arguments et sur le type en retour. Vous devez aussi avoir le droit `EXECUTE` sur la fonction sous-jacente. Si un opérateur de commutation ou de négation est spécifié, vous devez être le propriétaire de ces opérateurs.

Paramètres

`nom`

Le nom de l'opérateur à définir. Voir ci-dessus pour les caractères autorisés. Le nom peut être qualifié du nom du schéma, par exemple `CREATE OPERATOR monschema.+ (...)`. Dans le cas contraire, il est créé dans le schéma courant. Deux opérateurs dans le même schéma peuvent avoir le même nom s'ils opérent sur des types de données différents. On parle alors de *surchargement*.

`nom_fonction`

La fonction utilisée pour implanter cet opérateur.

`type_gauche`

Le type de données de l'opérande gauche de l'opérateur, s'il existe. Cette option est omise pour un opérateur préfixe.

`type_droit`

Le type de données de l'opérande droit de l'opérateur.

`op_com`

Le commutateur de cet opérateur.

`op_neg`

La négation de cet opérateur.

`proc_res`

La fonction d'estimation de la sélectivité de restriction pour cet opérateur.

`proc_join`

La fonction d'estimation de la sélectivité de jointure pour cet opérateur.

`HASHES`

L'opérateur peut supporter une jointure de hachage.

`MERGES`

L'opérateur peut supporter une jointure de fusion.

La syntaxe `OPERATOR()` est utilisée pour préciser un nom d'opérateur qualifié d'un schéma dans *`op_com`* ou dans les autres arguments optionnels. Par exemple :

```
COMMUTATOR = OPERATOR(mon_schema.===) ,
```

Notes

Section 38.14 fournit de plus amples informations.

Il n'est pas possible de spécifier la précédence lexicale d'un opérateur dans `CREATE OPERATOR` car le comportement de précédence de l'analyseur n'est pas modifiable. Voir Section 4.1.6 pour des détails sur la gestion de la précédence.

Les options obsolètes, `SORT1`, `SORT2`, `LTCMP` et `GTCMP` étaient utilisées auparavant pour spécifier les noms des opérateurs de tris associés avec un opérateur joignable par fusion (`mergejoinable`). Ceci n'est plus nécessaire car l'information sur les opérateurs associés est disponible en cherchant les familles d'opérateur B-tree. Si une des ces options est fournie, elle est ignorée mais configure implicitement `MERGES` à true.

`DROP OPERATOR` est utilisé pour supprimer les opérateurs utilisateur, `ALTER OPERATOR` pour les modifier.

Exemples

La commande suivante définit un nouvel opérateur, « area-equality », pour le type de données `box` :

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    FUNCTION = area_equal_function,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_function,
    JOIN = area_join_function,
    HASHES, MERGES
);
```

Compatibilité

`CREATE OPERATOR` est une extension PostgreSQL. Il n'existe pas d'opérateurs utilisateur dans le standard SQL.

Voir aussi

ALTER OPERATOR, CREATE OPERATOR CLASS, DROP OPERATOR

CREATE OPERATOR CLASS

CREATE OPERATOR CLASS — Définir une nouvelle classe d'opérateurs

Synopsis

```
CREATE OPERATOR CLASS nom [ DEFAULT ] FOR TYPE type_donnee
  USING methode_indexage [ FAMILY nom_famille ] AS
  {  OPERATOR numero_strategie nom_operateur [ ( type_op, type_op
 ) ] [ FOR SEARCH | FOR ORDER BY nom_famille_tri ]
  | FUNCTION numero_support [ ( type_op [ , type_op
 ] ) ] nom_fonction ( type_argument [, ...] )
   | STORAGE type_stockage
  } [, ... ]
```

Description

`CREATE OPERATOR CLASS` crée une nouvelle classe d'opérateur. Une classe d'opérateurs définit la façon dont un type de données particulier peut être utilisé avec un index. La classe d'opérateurs spécifie le rôle particulier ou la « stratégie » que jouent certains opérateurs pour ce type de données et cette méthode d'indexation. La classe d'opérateurs spécifie aussi les fonctions de support à utiliser par la méthode d'indexation quand la classe d'opérateurs est sélectionnée pour une colonne d'index. Tous les opérateurs et fonctions utilisés par une classe d'opérateurs doivent être définis avant la création de la classe d'opérateurs.

Si un nom de schéma est donné, la classe d'opérateurs est créée dans le schéma spécifié. Sinon, elle est créée dans le schéma courant. Deux classes d'opérateur ne peuvent avoir le même nom que s'ils concernent des méthodes d'indexation différentes.

L'utilisateur qui définit une classe d'opérateurs en devient propriétaire. Actuellement, le créateur doit être superutilisateur. Cette restriction existe parce qu'une définition erronée d'une classe d'opérateurs peut gêner le serveur, voire causer un arrêt brutal de celui-ci.

Actuellement, `CREATE OPERATOR CLASS` ne vérifie pas si la définition de la classe d'opérateurs inclut tous les opérateurs et fonctions requis par la méthode d'indexation. Il ne vérifie pas non plus si les opérateurs et les fonctions forment un ensemble cohérent. Il est de la responsabilité de l'utilisateur de définir une classe d'opérateurs valide.

Les classes d'opérateurs en relation peuvent être groupées dans des *familles d'opérateurs*. Pour ajouter une nouvelle classe d'opérateurs à une famille existante, indiquez l'option `FAMILY` dans `CREATE OPERATOR CLASS`. Sans cette option, la nouvelle classe est placée dans une famille de même nom (créant la famille si elle n'existe pas).

Section 38.16 fournit de plus amples informations.

Paramètres

nom

Le nom (éventuellement qualifié du nom du schéma) de la classe d'opérateurs à créer.

`DEFAULT`

La classe d'opérateurs est celle par défaut pour son type de données. Il ne peut y avoir qu'une classe d'opérateurs par défaut pour un type de données et une méthode d'indexation particuliers.

type_données

Le type de données de la colonne auquel s'applique cette classe d'opérateurs.

méthode_index

Le nom de la méthode d'indexation à laquelle s'applique la classe d'opérateurs.

nom_famille

Le nom d'une famille d'opérateur existante pour lui ajouter cette classe d'opérateur. Si non spécifié, une famille du même nom que l'opérateur est utilisée (la créant si elle n'existe pas déjà).

numéro_stratégie

Le numéro de stratégie de la méthode d'indexation pour un opérateur associé à la classe d'opérateurs.

nom_opérateur

Le nom (éventuellement qualifié du nom du schéma) d'un opérateur associé à la classe d'opérateurs.

op_type

Dans une clause `OPERATOR`, le(s) type(s) de données de l'opérande d'un opérateur ou `NONE` pour signifier un opérateur préfixe. Les types de données de l'opérande peuvent être omis dans le cas où ils sont identiques au type de données de la classe d'opérateurs.

Dans une clause `FUNCTION`, le (ou les) types de données en opérande, supporté par la fonction, si différent du type de données en entrée de la fonction (pour les fonctions de comparaison d'index B-tree et les fonctions des index hash) ou le type de données de la classe (pour les fonctions de support du tri pour les index B-tree, les fonctions d'égalité d'image des B-tree et pour toutes les fonctions des opérateurs de classe des index GiST, SP-GiST, GIN et BRIN). Ces valeurs par défaut sont correctes. Du coup, *op_type* n'a pas besoin d'être précisé dans les clauses `FUNCTION`, sauf dans le cas de la fonction de support du tri pour les index B-tree qui doit supporter les comparaisons inter-types.

nom_famille_tri

Le nom (éventuellement qualifié du nom du schéma) d'une famille d'opérateur `btree` qui décrit l'ordre de tri associé à un opérateur de tri.

Si ni `FOR SEARCH` ni `FOR ORDER BY` ne sont spécifiés, `FOR SEARCH` est la valeur par défaut.

numéro_support

Le numéro de fonction support de la méthode d'indexation pour une fonction associée à la classe d'opérateurs.

nom_fonction

Le nom (éventuellement qualifié du nom du schéma) d'une fonction support pour la méthode d'indexation de la classe d'opérateurs.

types_argument

Le(s) type(s) de données des paramètres de la fonction.

type_stockage

Le type de données réellement stocké dans l'index. C'est normalement le même que le type de données de la colonne mais certaines méthodes d'indexage (GiST, GIN, SP-GiST et BRIN

actuellement) autorisent un type différent. La clause STORAGE doit être omise sauf si la méthode d'indexation autorise un type différent. Si la colonne *type_donnee* est spécifiée comme anyarray, le *type_stockage* peut être déclarré comme anyelement pour indiquer que les entrées dans l'index sont des membres du type d'élément appartenant au type de donnée du tableau courant pour lequel chaque index est créé spécifiquement pour.

L'ordre des clauses OPERATOR, FUNCTION et STORAGE n'a aucune importance.

Notes

Comme toute la partie d'indexage ne vérifie pas les droits d'accès aux fonctions avant de les utiliser, inclure une fonction ou un opérateur dans une classe d'opérateurs est équivalent à donner les droits d'exécution à PUBLIC sur celle-ci. Ce n'est pas un problème habituellement pour les types de fonctions utiles dans une classe d'opérateurs.

Les opérateurs ne doivent pas être définis par des fonctions SQL. Une fonction SQL peut être intégrée dans la requête appelante, ce qui empêche l'optimiseur de faire la correspondance avec un index.

Avant PostgreSQL 8.4, la clause OPERATOR pouvait inclure l'option RECHECK. Cela n'est plus supporté car le fait qu'un index soit « à perte » est maintenant déterminé à l'exécution. Ceci permet une gestion plus efficace des cas où l'opérateur pourrait ou non être à perte.

Exemples

La commande issue de l'exemple suivant définit une classe d'opérateurs d'indexation GiST pour le type de données _int4 (tableau de int4). Voir le module intarray pour l'exemple complet.

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       = (anyarray, anyarray),
        OPERATOR        7       @>,
        OPERATOR        8       <@,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4,
 smallint, oid, internal),
        FUNCTION        2       g_int_union (internal, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal,
 internal),
        FUNCTION        6       g_int_picksplit (internal,
 internal),
        FUNCTION        7       g_int_same (_int4, _int4,
 internal);
```

Compatibilité

CREATE OPERATOR CLASS est une extension PostgreSQL. Il n'existe pas d'instruction CREATE OPERATOR CLASS dans le standard SQL.

Voir aussi

ALTER OPERATOR CLASS, DROP OPERATOR CLASS, CREATE OPERATOR FAMILY, ALTER OPERATOR FAMILY

CREATE OPERATOR FAMILY

CREATE OPERATOR FAMILY — définir une nouvelle famille d'opérateur

Synopsis

```
CREATE OPERATOR FAMILY nom USING methode_indexage
```

Description

`CREATE OPERATOR FAMILY` crée une nouvelle famille d'opérateurs. Une famille d'opérateurs définit une collection de classes d'opérateur en relation et peut-être quelques opérateurs et fonctions de support supplémentaires compatibles avec ces classes d'opérateurs mais non essentiels au bon fonctionnement des index individuels. (Les opérateurs et fonctions essentiels aux index doivent être groupés avec la classe d'opérateur adéquate, plutôt qu'être des membres « lâches » dans la famille d'opérateur. Typiquement, les opérateurs sur un seul type de données peuvent être lâches dans une famille d'opérateur contenant des classes d'opérateurs pour les deux types de données.)

La nouvelle famille d'opérateur est initialement vide. Elle sera remplie en exécutant par la suite des commandes `CREATE OPERATOR CLASS` pour ajouter les classes d'opérateurs contenues et, en option, des commandes `ALTER OPERATOR FAMILY` pour ajouter des opérateurs et leur fonctions de support correspondantes en tant que membres « lâches ».

Si un nom de schéma est précisée, la famille d'opérateur est créée dans le schéma en question. Sinon elle est créée dans le schéma en cours. Deux familles d'opérateurs du même schéma ne peuvent avoir le même nom que s'ils sont des méthodes d'indexage différentes.

L'utilisateur qui définit une famille d'opérateur devient son propriétaire. Actuellement, l'utilisateur qui crée doit être un superutilisateur. (Cette restriction est nécessaire car une définition erronée d'une famille d'opérateur pourrait gêner le serveur, voire même l'arrêter brutalement.)

Voir Section 38.16 pour plus d'informations.

Paramètres

nom

Le nom de la famille d'opérateur (pouvant être qualifié du schéma).

methode_indexage

Le nom de la méthode d'indexage utilisée par cette famille d'opérateur.

Compatibilité

`CREATE OPERATOR FAMILY` est un extension PostgreSQL. Il n'existe pas d'instruction `CREATE OPERATOR FAMILY` dans le standard SQL.

Voir aussi

ALTER OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

CREATE POLICY

CREATE POLICY — définir un niveau de politique de sécurité pour une table

Synopsis

```
CREATE POLICY nom ON nom_table
    [ AS { PERMISSIVE | RESTRICTIVE } ]
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { nom_role | PUBLIC | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER } [, ...] ]
    [ USING ( expression_USING ) ]
    [ WITH CHECK ( expression_CHECK ) ]
```

Description

La commande `CREATE POLICY` définit un nouveau niveau de politique de sécurité pour une table. Notez que le niveau de politique de sécurité doit être actif pour la table. Les politiques de sécurité créées peuvent être appliquées en utilisant la commande suivante : `ALTER TABLE ... ENABLE ROW LEVEL SECURITY`

Une politique (*policy* dans la version originale de la documentation) valide l'autorisation de sélectionner (instruction SELECT), insérer (instruction INSERT), mettre à jour (instruction UPDATE) ou supprimer (instruction DELETE) des lignes qui correspondent à l'expression concordante d'une politique particulière. Une expression spécifiée avec `USING` sera vérifiée par rapport aux lignes existantes dans la table, tandis qu'une expression spécifiée avec `WITH CHECK` sera vérifiée sur les nouvelles lignes crées par `INSERT` ou `UPDATE` Lorsqu'une expression définie dans `USING` renvoie true pour une ligne donnée, alors cette ligne est visible pour l'utilisateur. Dans le cas contraire, cette ligne reste invisible. Généralement, aucune erreur ne survient quand une ligne est invisible, mais voir Tableau 287 pour les exceptions. Lorsqu'une expression définie dans `WITH CHECK` renvoie true pour une ligne, alors cette ligne est insérée. Par contre, si elle renvoie false ou NULL, cela génère une erreur.

Pour les commandes `INSERT`, `UPDATE` et `MERGE`, les expressions définies dans `WITH CHECK` sont appliquées après l'activation du trigger `BEFORE` et avant qu'aucune modification de données n'ait réellement été effectuée. Un trigger `BEFORE ROW` peut éventuellement modifier les données à insérer, influençant ainsi le résultat de la politique de sécurité. Les expressions définies dans `WITH CHECK` sont forcées avant toutes les autres contraintes.

Les noms de politique s'entendent par table. De ce fait, un même nom de politique peut être utilisé pour différentes tables et avoir une définition différente, adaptée à la table en question.

Les politiques peuvent être appliquées pour des commandes ou rôles spécifiques. Par défaut, une nouvelle politique créée sera appliquée à toutes les commandes et pour tous les rôles à moins qu'autre chose ne soit spécifié. Plusieurs politiques peuvent s'appliquer à une seule commande ; voir ci-dessous pour plus de détails. Tableau 287 résume la façon dont s'appliquent les différents types de politique aux commandes spécifiques.

Pour les politiques qui ont simultanément les expressions `USING` et `WITH CHECK` (`ALL` et `UPDATE`), s'il n'y a pas d'expression `WITH CHECK` définie, alors l'expression `USING` sera utilisée pour déterminer les lignes visibles (cas normal d'utilisation de `USING`), et les lignes qui obtiendront l'autorisation d'être ajoutées (cas `WITH CHECK`).

Si un niveau de sécurité est activé pour une table mais qu'aucune politique (policy) n'est applicable, une politique « default deny » est utilisée, plus aucune ligne n'est alors visible ou modifiable.

Paramètres

nom

Nom de la politique à créer. Chaque nom de politique doit être unique au sein d'une table.

nom_table

Le nom (optionnellement qualifié par le schéma) de la table à laquelle s'applique la politique.

PERMISSIVE

Spécifie que la politique doit être crée comme une politique permissive. Toutes les politiques permissives qui s'appliquent à une requête donnée seront combinée ensemble en utilisant l'opérateur booléen « OR ». En créant des politiques permissives, les administrateurs peuvent ajouter des enregistrements à l'ensemble qui sera accédé. Les politiques sont permissives par défaut.

RESTRICTIVE

Spécifie que la politique doit être crée comme une politique restrictive. Toutes les politiques permissives qui s'appliquent à une requête donnée seront combinée ensemble en utilisant l'opérateur booléen « AND ». En créant des politiques restrictives, les administrateurs peuvent retirer des enregistrements de l'ensemble qui sera accédé puisque toutes les politiques restrictives doivent être passées pour chaque enregistrement.

Il est nécessaire d'avoir au moins une politique permissive pour autoriser l'accès aux enregistrements avant que les politiques restrictives ne puissent être utilisées pour réduire cet accès. Si seules des politiques restrictives existent, alors aucun enregistrement ne sera accessible. Quand un mixe de politiques permissives et restrictives est présent, un enregistrement n'est accessible que si au moins une politique permissive passe, en plus de toutes les politiques restrictives.

commande

La commande à laquelle la politique s'applique. Les options valides sont les suivantes : ALL, SELECT, INSERT, UPDATE, et DELETE. ALL est la valeur par défaut. Vous verrez par la suite comment sont appliquées les spécificités de chaque option.

nom_role

Le ou les role(s) auxquels les politiques sont appliquées. Par défaut, c'est le pseudo-rôle PUBLIC, qui applique les politiques à tous les rôles.

expression_USING

Toute expression SQL conditionnelle (autrement dit, renvoyant une donnée de type boolean). L'expression conditionnelle ne peut pas contenir de fonction d'agrégat ou de fenêtrage (window). Si le niveau de politique de sécurité est activé, cette expression sera ajoutée aux requêtes exécutées sur la table. Les lignes pour lesquelles l'expression renvoie true seront visibles. Toute ligne pour laquelle l'expression renvoie false ou NULL sera invisible pour l'utilisateur (avec SELECT) et ne sera pas modifiable (avec UPDATE ou DELETE). Généralement, ces lignes seront supprimées silencieusement ; aucune erreur ne survient (mais voir Tableau 287 pour les exceptions).

expression_CHECK

Toute expression SQL conditionnelle (autrement dit, renvoyant une donnée de type boolean). L'expression conditionnelle ne peut pas contenir de fonction d'agrégat ou de fenêtrage (window). Si le niveau de politique de sécurité est activé, cette expression sera utilisée dans les requêtes contenant INSERT et UPDATE. Seules les lignes pour lesquelles l'expression est évaluée à true seront autorisée à être modifiées. Une erreur sera générée si l'évaluation de la condition de la

commande UPDATE ou INSERT renvoie false ou NULL pour n'importe quel enregistrement parmi l'ensemble des résultats. Notez que *expression_CHECK* est évaluée sur le futur contenu de la ligne, et non pas sur le contenu d'origine.

Politique par commande

ALL

Utiliser ALL pour une politique signifie qu'elle s'appliquera pour toutes les commandes, peu importe le type de commande. Si une politique ALL existe et que des politiques spécifiques supplémentaires existent, alors leur résultat sera appliqué. Pour terminer, les politiques ALL seront appliquées pour la partie extraction et pour la partie modification de la requête, en utilisant l'expression définie dans USING pour les deux cas si seule la partie USING est définie.

Par exemple, si une requête UPDATE est exécutée, alors la politique ALL sera applicable sur les lignes à modifier que la commande UPDATE sera capable de sélectionner (en appliquant l'expression définie dans USING) mais aussi sur le résultat des lignes modifiées, pour vérifier s'il est autorisé de les ajouter à la table (en appliquant l'expression définie dans WITH CHECK si elle est définie, et sinon en appliquant l'expression définie dans USING). Si une commande INSERT ou UPDATE essaie d'ajouter des lignes à une table et est bloquée par l'expression définie dans WITH CHECK de la politique ALL (ou son expression USING si elle n'a pas d'expression WITH CHECK), l'ensemble de la commande est annulé.

SELECT

Utiliser SELECT dans une politique signifie que cette politique s'appliquera à toutes les requêtes SELECT ainsi qu'à toute vérification du droit SELECT nécessaire sur la table pour laquelle la politique est définie. Concernant les requêtes SELECT, le résultat sera composé uniquement des lignes qui auront passé la politique SELECT Pour les requêtes qui demandent des droits, telles que les commandes d'UPDATE, DELETE et MERGE, elles verront uniquement dans le résultat les lignes qui auront été autorisés par la politique SELECT, sauf comme décrit ci-dessous.

Si une requête de modification de données a une clause RETURNING, les droits SELECT sont requis sur la relation, et toute nouvelle ligne insérée ou supprimée pour cette relation doit satisfaire les politiques SELECT de la relation pour être disponible dans la clause RETURNING. Si une ligne nouvellement insérée ou modifiée ne satisfait pas les politiques SELECT de la relation, une erreur sera renvoyée (les lignes insérées ou modifiées à renvoyer ne sont *jamais* ignorées silencieusement).

Si un INSERT a une clause ON CONFLICT DO UPDATE ou une clause ON CONFLICT DO NOTHING avec une spécification d'un index ou d'une contrainte arbitre, les droits SELECT sont requis sur la relation et les lignes proposées en insertion sont vérifiées en utilisant les politiques SELECT de la relation. Si une ligne proposée en insertion ne satisfait pas les politiques SELECT de la relation, une erreur est renvoyée (le INSERT n'est *jamais* silencieusement évité). De plus, si le chemin UPDATE est pris, la ligne à mettre à jour et la nouvelle ligne mise à jour sont vérifiées avec les politiques SELECT de la relation, et une erreur est renvoyée si elles ne sont pas satisfaites (un UPDATE auxiliaire n'est *jamais* évité silencieusement.

Une commande MERGE requiert les droits SELECT sur les relations source et cible, et donc chaque politique SELECT de la relation est appliqué avant de les joindre, et les actions du MERGE verront uniquement les lignes qui sont autorisées par ces politiques. De plus, si un UPDATE est exécuté, les politiques SELECT de la relation cible sont appliquées à la ligne mise à jour, comme pour un UPDATE autonome, sauf qu'une erreur est renvoyée si elles ne sont pas satisfaite. Une politique SELECT ne peut pas avoir une expression définie dans WITH CHECK qui ne s'applique que dans le cas où des enregistrements sont récupérés depuis la table.

INSERT

Utiliser INSERT dans une politique signifie que cette politique s'appliquera à toutes les requêtes INSERT Les lignes à insérer qui ne passent pas la politique renvoient une erreur de violation de

politique, et l'ensemble INSERT de la commande est annulé. Une politique INSERT ne peut pas avoir une expression définie dans USING qui ne s'applique que dans les cas où des enregistrements sont ajoutés à la table.

Notez qu'une commande INSERT avec ON CONFLICT DO UPDATE vérifie les politiques INSERT avec les expressions définies dans WITH CHECK pour toutes les lignes proposées en insertion, qu'elles soient ou non insérées au final.

UPDATE

Utiliser UPDATE dans une politique signifie que cette politique s'appliquera à toutes les requêtes UPDATE, SELECT FOR UPDATE et SELECT FOR SHARE, ainsi qu'aux clauses ON CONFLICT DO UPDATE de la commande INSERT et de la commande MERGE si elle contient une action UPDATE. Puisque la commande UPDATE implique de récupérer un enregistrement existant et le replacer avec un nouvel enregistrement modifié, la politique UPDATE accepte les expressions définies dans USING mais aussi dans WITH CHECK L'expression définie dans USING déterminera sur quelle selection d'enregistrements la commande UPDATE est capable de travailler tandis que l'expression définie dans WITH CHECK déterminera les enregistrements qui pourront être modifiés et réinjectés dans la table.

Si une seule ligne à mettre à jour ne remplit pas les conditions pour être autorisée par l'expression spécifiée dans WITH CHECK, une erreur sera générée, et l'ensemble de la commande est annulé. S'il n'y a que l'expression spécifiée dans USING qui a été définie alors c'est cette expression qui sera utilisée pour vérifier les cas USING et WITH CHECK

Typiquement, une commande UPDATE a aussi besoin de lire les données des colonnes de la relation mise à jour (par exemple dans une clause WHERE ou dans une clause RETURNING ou dans une expression du côté droit de la clause SET). Dans ce cas, les droits SELECT sont aussi requis sur la relation en cours de mise à jour, et les politiques SELECT ou ALL seront appliquées en plus des politiques UPDATE. De ce fait, l'utilisateur doit avoir accès aux lignes en cours de mise à jour via une politique SELECT ou ALL en plus d'avoir le droit de mettre à jour la ligne via une politique UPDATE ou ALL.

Quand une commande INSERT a une clause supplémentaire ON CONFLICT DO UPDATE, si le chemin UPDATE est pris, la ligne à mettre à jour est tout d'abord vérifiée avec les expressions USING de toute politique UPDATE, puis la nouvelle ligne mise à jour est vérifiée avec les expressions WITH CHECK. Néanmoins, notez que, contrairement à une commande UPDATE autonome, si la ligne existante ne passe pas les expressions USING, une erreur sera levée (le chemin UPDATE ne sera *jamais* évité silencieusement). Cela s'applique aussi à une action UPDATE d'une commande MERGE.

DELETE

Utiliser DELETE dans une politique signifie que cette politique s'appliquera à toutes les requêtes DELETE et aux requêtes MERGE contenant des commandes DELETE. Pour une commande DELETE, seules les lignes autorisées par cette politique seront visibles à une commande DELETE Il peut y avoir des lignes visibles retournées par la commande SELECT qui ne sont pas candidates à la suppression si elles ne sont pas validées par l'expression définie dans la clause USING de la politique DELETE Cependant, notez qu'une action DELETE dans une commande MERGE verra des lignes qui sont visibles au travers des politiques SELECT et, si la politique DELETE ne passe pas pour une telle ligne, une erreur sera renvoyée.

Dans la plupart des cas, une commande DELETE doit également lire les données des colonnes qu'elle supprime dans la relation (par exemple, dans une clause WHERE ou une clause RETURNING). Dans ce cas, les droits SELECT sont également requis sur la relation, et les politiques de sécurité SELECT ou ALL seront appliquées en plus des politiques de sécurité DELETE. Ainsi, l'utilisateur doit avoir accès à la ou aux lignes en cours de suppression via une politique de sécurité SELECT ou ALL en plus d'être autorisé à supprimer la ou les lignes via une politique de sécurité DELETE ou ALL.

Une politique DELETE ne peut pas avoir d'expression définie dans WITH CHECK puisque cette politique ne s'applique qu'à des enregistrements qui vont être supprimés de la table. Il n'y a donc pas de nouvelles lignes à vérifier.

Tableau 287 résume comment les différents types de politique s'appliquent aux commandes spécifiques. Dans le tableau, « vérifie » signifie que l'expression de la politique est vérifiée et une erreur est renvoyée si la vérification renvoie false ou null, alors que « filtre » signifie que la ligne est ignorée silencieusement si l'expression de la politique renvoie false ou null.

Tableau 287. Politiques appliquées par type de commande

Command	Politique SELECT/ALL	Politique INSERT/ALL	Politique UPDATE/ALL		Politique DELETE/ALL
	Expression USING	Expression WITH CHECK	Expression USING	Expression WITH CHECK	Expression USING
SELECT / COPY ... TO	Filtre la ligne existante	--	--	--	--
SELECT FOR UPDATE/ SHARE	Filtre la ligne existante	--	Filtre la ligne existante	--	--
INSERT	Vérifie la nouvelle ligne [a]	Vérifie la nouvelle ligne	--	--	--
UPDATE	Filtre la ligne existante [a] & vérifie la nouvelle ligne [a]	--	Filtre la ligne existante	Vérifie la nouvelle ligne	--
DELETE	Filtre la ligne existante [a]	--	--	--	Filtre la ligne existante
INSERT ... ON CONFLICT	Vérifie la nouvelle ligne [bc]	Vérifie la nouvelle ligne [c]	--	--	--
ON CONFLICT DO UPDATE	Vérifie les lignes existantes & nouvelles [d]	--	Vérifie la ligne existante	Vérifie la nouvelle ligne [d]	--
MERGE	Filtre les lignes source & cible	--	--	--	--
MERGE ... THEN INSERT	--	Vérifie la nouvelle ligne	--	--	--
MERGE ... THEN UPDATE	Vérifie la nouvelle ligne	--	Vérifie la ligne existante	Vérifie la nouvelle ligne	--
MERGE ... THEN DELETE	--	--	--	--	Vérifie la ligne existante

[a] Si l'accès en lecture est requis pour la ligne existante ou la nouvelle ligne, une clause WHERE ou RETURNING faisant référence aux colonnes de la relation).

[b] Si un index ou une contrainte arbitre est indiqué.

[c] La ligne proposée en insertion est vérifiée qu'un conflit survienne ou pas.

[d] La nouvelle ligne d'un UPDATE auxiliaire, qui pourrait être différente de la nouvelle ligne d'une commande INSERT originale.

Application de plusieurs politiques

Quand plusieurs politiques de différents types de commande s'appliquent à la même commande (par exemple, des politiques SELECT et UPDATE appliquées à une commande UPDATE), alors l'utilisateur doit avoir les deux types de droits (par exemple, le droit de sélectionner les lignes de la relation ainsi que le droit de les mettre à jour). De ce fait, les expressions pour un type de politique sont combinées pour l'autre type de politique en utilisant l'opérateur AND.

Quand plusieurs politiques du même type de commande s'appliquent à la même commande, alors il doit exister au moins une politique PERMISSIVE donnant accès à la relation, et toutes les politiques RESTRICTIVE doivent passer. De ce fait, toutes les expressions de politique PERMISSIVE sont combinées en utilisant OR, toutes les expressions de politique RESTRICTIVE sont combinées en utilisant AND, et les résultats sont combinés en utilisant AND. S'il n'y a pas de politiques PERMISSIVE, alors l'accès est refusé.

Notez que, pour combiner les différentes politiques, les politiques ALL sont traitées comme ayant le même type que tout autre type de politique appliquée.

Par exemple, dans une commande UPDATE nécessitant des droits pour SELECT et UPDATE, si plusieurs politiques de chaque type sont applicables, elles seront combinées ainsi :

```
expression from RESTRICTIVE SELECT/ALL politique 1
AND
expression from RESTRICTIVE SELECT/ALL politique 2
AND
...
AND
(
  expression from PERMISSIVE SELECT/ALL politique 1
  OR
  expression from PERMISSIVE SELECT/ALL politique 2
  OR
  ...
)
AND
expression from RESTRICTIVE UPDATE/ALL politique 1
AND
expression from RESTRICTIVE UPDATE/ALL politique 2
AND
...
AND
(
  expression from PERMISSIVE UPDATE/ALL politique 1
  OR
  expression from PERMISSIVE UPDATE/ALL politique 2
  OR
  ...
)
```

Notes

Vous devez être le propriétaire de la table pour laquelle vous souhaitez creéer ou modifier des politiques.

Tandis que les politiques sont appliquées pour les requêtes accédant explicitement aux tables de la base de données, elles ne sont pas appliquées lorsque le système réalise des vérifications internes d'intégrité

sur le référentiel ou pour la validation des contraintes. Ce qui signifie qu'il y a des manières indirectes de déterminer si une valeur donnée existe. Par exemple, si vous essayez d'insérer un doublon dans une colonne clé primaire, ou qui possède une contrainte d'unicité. Si l'insertion échoue alors l'utilisateur peut inférer que la valeur existe déjà. (dans cet exemple, il est entendu que l'utilisateur est soumis à une politique de sécurité lui permettant d'insérer des enregistrements qu'il n'est néanmoins pas autorisé à consulter) Un autre exemple, si un utilisateur est autorisé à insérer dans une table qui en référence une autre, une table cachée. Son existence peut être déterminée par l'utilisateur en insérant une valeur dans la table, la réussite indiquerait que la valeur existe dans la table référencée. Ces problèmes peuvent être résolus en vérifiant minutieusement les politiques de façon à ce que les utilisateurs ne puissent pas insérer, supprimer, ou mettre à jour des enregistrements qui pourraient récupérer des valeurs qu'ils ne devraient pas pouvoir consulter, ou en utilisant un générateur de valeur (par exemple clés substituées) à la place de clés à signification externe.

En général le système va appliquer des conditions filtrantes en se servant de politiques de sécurité pour prioriser les conditions apparaissant dans les requêtes utilisateur. Ceci afin d'éviter d'exposer par inadvertance des données protégées à certaines fonctions utilisateurs qui pourraient ne pas être dignes de confiance. Les fonctions et opérateurs, taggués `LEAKPROOF` par le système (ou l'administrateur système) seront évaluées avant les expressions des politiques et seront considérées comme digne de confiance.

Comme les expressions de politique s'appliquent directement à la requête d'un utilisateur, elles seront lancées avec les droits de cet utilisateur pendant toute la durée de la requête. De ce fait, un utilisateur qui utilise une politique donnée doit pouvoir accéder à toutes les tables et fonctions référencées dans l'expression de vérification, sinon il recevra une erreur du type « permission denied » en essayant d'accéder à une référence dont le niveau de sécurité est activé. Cependant, ceci ne modifie pas le fonctionnement des vues. Comme avec les requêtes classiques et leurs vues, les vérifications des autorisations et politiques des tables référencées par la vue utilisent les droits du propriétaire de la vue, ainsi que les politiques qui s'appliquent sur le propriétaire de la la vue, sauf si la vue est définie en utilisant l'option `security_invoker` (voir `CREATE VIEW`).

Aucune politique séparée n'existe pour `MERGE`. À la place, les politiques définies pour `SELECT`, `INSERT`, `UPDATE` et `DELETE` sont appliquées lors de l'exécution de `MERGE`, dépendant des actions réalisées.

Des commentaires supplémentaires et des exemples pratiques peuvent être trouvés ici : Section 5.8.

Compatibilité

`CREATE POLICY` est une extension PostgreSQL.

Voir aussi

ALTER POLICY, DROP POLICY, ALTER TABLE

CREATE PROCEDURE

CREATE PROCEDURE — définit une nouvelle procédure stockée

Synopsis

```
CREATE [ OR REPLACE ] PROCEDURE
    nom ( [ [ mode_argument ] [ nom_argument ] type_argument
 [ { DEFAULT | = } expr_defaut ] [, ...] ] )
  { LANGUAGE nom_langage
    | TRANSFORM { FOR TYPE nom_type } [, ... ]
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | SET parametre_configuration { TO valeur | = valeur | FROM
 CURRENT }
    | AS 'definition'
    | AS 'fichier_objet', 'symbole_lien'
    | corps_sql
  } ...
```

Description

`CREATE PROCEDURE` définit une nouvelle procédure. `CREATE OR REPLACE PROCEDURE` va définir une nouvelle procédure, ou remplacer une définition existante. Pour pouvoir définir une procédure, l'utilisateur doit avoir le privilège `USAGE` sur le langage.

Si le nom du schéma est inclus, alors la procédure est créée dans le schéma spécifié. Sinon elle est créée dans le schéma courant. Le nom de la nouvelle procédure ne doit correspondre à aucune procédure ou fonction existante possédant les mêmes types d'arguments dans le même schéma. Cependant, des procédures et fonctions avec des arguments de types différents peuvent partager le même nom (on appelle cela *surcharge* ou *overloading*).

Pour remplacer la définition en cours d'une procédure existante, utilisez `CREATE OR REPLACE PROCEDURE`. Il n'est pas possible de changer le nom ou les types d'arguments d'une procédure avec cette méthode (si vous le faites, vous créez en fait une nouvelle procédure distincte).

Si `CREATE OR REPLACE PROCEDURE` est utilisé pour remplacer une procédure existante, le propriétaire et les permissions sur la procédure ne changent pas. Toutes les autres propriétés de la procédure se voient assignées les valeurs spécifiées dans la commande. Vous devez être propriétaire de la procédure pour la remplacer (cela fonctionne aussi si vous êtes membre du rôle propriétaire).

L'utilisateur qui crée la procédure devient son propriétaire.

Pour pouvoir créer une procédure, vous devez avoir le privilège `USAGE` sur les types des arguments.

La lecture de Section 38.3 fournit des informations supplémentaires sur l'écriture de procédures.

Paramètres

nom

Le nom (éventuellement qualifié par un schéma) de la procédure à créer.

mode_argument

Le mode d'un argument : `IN`, `OUT`, `INOUT` ou `VARIADIC`. Sans précision, le défaut est `IN`.

nom_argument

Le nom d'un argument.

type_argument

Le(s) type(s) des arguments de la procédure (éventuellement qualifiés par un schéma), s'il y en a. Ils peuvent être les types de base, des types composites, des domaines, ou des références à un type d'une colonne d'une table.

Selon le langage d'implémentation, il peut être permis de spécifier des « pseudo-types » comme `cstring`. Les pseudo-types indiquent le type d'argument que est soit incomplètement spécifié, soit en dehors des types de données ordinaires.

On fait référence au type d'une colonne en écrivant *`table_name.column_name`*`%TYPE`. Cette fonctionnalité permet parfois de rendre une procédure indépendante des changements de définition d'une table.

expr_defaut

Une expression à utiliser comme valeur par défaut si le paramètre n'est pas spécifié. L'expression doit respecter le type d'argument du paramètre. Tous les paramètres en entrée suivant un paramètre avec une valeur par défaut doivent en avoir une également.

nom_langage

Le nom du langage dans lequel la procédure est implémentée. Ce peut être `sql`, `c`, `internal` ou le nom d'un langage procédural défini par l'utilisateur, par exemple `plpgsql`. La valeur par défaut est `sql` si *`corps_sql`* est indiqué. Mettre le nom entre guillemets simples est obsolète et exige une casse identique.

`TRANSFORM { FOR TYPE` *`nom_type`* `} [, ... ] }`

Liste les transformations qu'un appel à la procédure devrait appliquer. Les transformations opèrent des conversions entre les types SQL et les types de données spécifiques au langage ; voir CREATE TRANSFORM. D'habitude les implémentations des langages procéduraux connaissent d'entrée les types internes, ces derniers n'ont donc pas besoin d'être listés ici. Si une implémentation d'un langage procédural ne sait pas traiter un type et qu'aucune transformation n'est fournie, elle se rabattra sur un comportement par défaut pour convertir les données, mais cela dépend de l'implémentation.

```
[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER
```

`SECURITY INVOKER` indique que la procédure doit être exécutée avec les privilèges de l'utilisateur qui l'appelle. C'est le défaut. `SECURITY DEFINER` spécifie que la procédure doit être exécutée avec les privilèges de l'utilisateur qui la possède.

Le mot clé `EXTERNAL` est permis pour la conformité envers le standard SQL, mais il est optionnel puisque, contrairement au SQL, cette fonctionnalité concerne toutes les procédures, et pas seulement les externes.

Une procédure `SECURITY DEFINER` ne peut exécuter des commandes de contrôle de transaction (par exemple `COMMIT` et `ROLLBACK`, selon le langage).

parametre_configuration
valeur

Avec la clause `SET`, le paramètre de configuration indiqué sera positionné à la valeur spécifiée à l'entrée dans la procédure, puis restauré à la valeur précédente à la sortie. `SET FROM CURRENT`

mémorise la valeur du paramètre en cours au moment où CREATE PROCEDURE a été exécuté comme la valeur à appliquer à l'entrée dans la procédure.

Si une clause SET est attachée à une procédure, alors les effets d'une commande SET LOCAL exécutée au sein de la procédure pour la même variable sont restreints à cette procédure : l'ancienne valeur du paramètre est toujours restaurée à la sortie de la procédure. Cependant, une commande SET ordinaire (sans LOCAL) a priorité sur la clause SET, tout comme elle le ferait sur un ordre SET LOCAL précédent : les effets d'une telle commande persisteront après la sortie de la procédure, à moins que la transaction en cours ne soit annulée.

Si une clause SET est attachée à une procédure, alors cette procédure ne peut exécuter d'ordres de contrôle de transaction (comme COMMITet ROLLBACK, selon le langage).

Voir SET et Chapitre 20 pour plus d'informations sur les noms et valeurs de paramètres autorisés.

definition

Une chaîne de caractères constante définissant la procédure ; sa signification dépend du langage. Ce peut être un nom de procédure interne, le chemin d'un fichier objet, un ordre SQL, ou du texte dans un langage procédural.

Le *dollar quoting* (voir Section 4.1.2.4) est souvent utile pour écrire la chaîne de définition de la fonction, plutôt que la syntaxe normale à simple guillemet. Sans *dollar quoting*, le moindre guillemet ou *backslash* dans la définition de la procédure doit être échappé et donc doublé.

fichier_objet, symbole_lien

Cette forme de la clause AS est utilisée pour les procédures en C chargées dynamiquement, quand le nom de la procédure dans le code source en C n'est pas le même que le nom de la procédure SQL. La chaîne *fichier_objet* est le nom de la bibliothèque partagée contenant la procédure C compilée, et est interprétée comme dans la commande LOAD. La chaîne *symbole_lien* est le symbole de lien de la procédure, c'est-à-dire le nom de la procédure dans le code source en C. Si le symbole de lien est absent, on suppose qu'il est le même que le nom de la procédure en train d'être définie.

Quand des commandes CREATE PROCEDURE répétées se réfèrent au même fichier objet, celui-ci n'est chargé d'une fois par session. Pour décharger et recharger le fichier (peut-être pendant le développement), démarrez une nouvelle session.

corps_sql

Le corps d'une procédure LANGUAGE SQL. Cela devrait être un bloc

```
BEGIN ATOMIC
  instruction;
  instruction;
  ...
  instruction;
END
```

Ceci est similaire à écrire le texte du corps de la procédure sous la forme d'une chaîne constante (voir *definition* ci-dessus), mais il existe quelques différences : Cette forme fonctionne seulement pour LANGUAGE SQL, la forme de chaîne constante fonctionne pour tous les langages. Cette forme est analysée au moment de la défintion de la procédure, la forme de chaîne constante est analysée au moment de l'exécution ; de ce fait, cette forme ne supporte pas les arguments de type polymorphique et les autres constructions qui ne sont pas analysables au moment de la définition. Cette forme traque les dépendances entres la procédure et les objets utilisés dans le corps de la procédure, donc un DROP ... CASCADE fonctionnera correctement, alors que la forme utilisant une chaîne pourrait laisser des procédures invalides. Enfin, cette forme est plus compatible avec le standard SQL et les autres implémentations SQL.

Notes

voir CREATE FUNCTION pour plus de détails sur la création de fonctions, qui s'appliquent aussi aux procédures.

Utilisez CALL pour exécuter une procédure.

Exemples

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
AS $$
INSERT INTO tbl VALUES (a);
INSERT INTO tbl VALUES (b);
$$;
```

ou

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
BEGIN ATOMIC
  INSERT INTO tbl VALUES (a);
  INSERT INTO tbl VALUES (b);
END;
```

et l'appeler ainsi :

```
CALL insert_data(1, 2);
```

Compatibilité

Une commande `CREATE PROCEDURE` est définie dans le standard SQL. L'implémentation de PostgreSQL peut être utilisée d'une façon compatible mais a de nombreuses extensions. Pour plus de détails, voir aussi CREATE FUNCTION.

Voir aussi

ALTER PROCEDURE, DROP PROCEDURE, CALL, CREATE FUNCTION

CREATE PUBLICATION

CREATE PUBLICATION — définir une nouvelle publication

Synopsis

```
CREATE PUBLICATION nom
    [ FOR ALL TABLES
      | FOR objet_publication [, ... ] ]
    [ WITH ( paramètre_publication [= valeur] [, ... ] ) ]

où objet_publication est parmi ;:

    TABLE table_et_colonnes [, ... ]
    TABLES IN SCHEMA { nom_schéma | CURRENT_SCHEMA } [, ... ]

et table_et_colonnes vaut :

    [ ONLY ] nom_table [ * ] [ ( nom_colonne [, ... ] ) ] [ WHERE
 ( expression ) ]
```

Description

`CREATE PUBLICATION` ajoute une nouvelle publication dans la base courante. Le nom de la publication doit être différent du nom de toutes les autres publications existante au sein de la base courante.

Une publication est essentiellement un groupe de table dont les changement de données sont destinés à être répliqué grâce à la réplication logique. Voir Section 31.1 pour les détails de comment les publications participent à la mise en place de la réplication logique.

Paramètres

nom

Le nom de la nouvelle publication.

`FOR TABLE`

Spécifie une liste de tables à ajouter à la publication. Si `ONLY` est spécifié avant le nom de la table, seul cette table est ajoutée à la publication. Si `ONLY` n'est pas spécifié, la table ainsi que toutes les tables descendantes (s'il y en a) est ajoutées. De manière facultative, `*` peut être spécifié après le nom de la table pour indiquer explicitement que les tables descendantes doivent être incluses. Néanmoins, ceci ne s'applique pas à une table partitionnée. Les partitions d'une table sont toujours considérées implicitement comme faisant partie de la publication, donc elles ne sont jamais ajoutées explicitement à la publication.

Si la clause `WHERE` optionnelle est indiquée, elle définit une expression de *filtre de lignes*. Les lignes pour lesquelles l'*`expression`* vaut false ou NULL ne seront pas publiées. Notez que les parenthèses sont requises autour de l'expression. Cette clause n'a pas d'effet sur les commandes `TRUNCATE`.

Quand une liste de colonnes est indiquée, seules les colonnes nommées sont répliquées. Si aucune liste de colonnes n'est indiquée, toutes les colonnes de la table sont répliquées au travers de

cette publication, en incluant toutes les colonnes ajoutées après coup. Cela n'a pas d'effet sur `TRUNCATE`. Voir Section 31.4 pour plus de détails sur les listes de colonnes.

Seules les tables persistantes et les tables partitionnées peuvent faire partie d'une publication. Les tables temporaires, tables non journalisées, tables distantes, vues matérialisées et vues standard ne peuvent pas faire partie d'une publication. Pour répliquer une table partitionnée, il faut ajouter chaque partition individuellement à la publication.

Indiquer une liste de colonnes quand la publication publie aussi `FOR TABLES IN SCHEMA` n'est pas autorisé.

Quand une table partitionnée est ajoutée à une publication, toutes les partitions existantes et futures seront considérées implicitement comme faisant partie de la publication. Donc, même les opérations réalisées directement sur une partition seront publiées via les publications dont fait partie la table partitionnée.

`FOR ALL TABLES`

Marque la publication comme publication qui réplique les changement pour toutes les tables de la base, en incluant les tables qui seront crées dans le futur.

`FOR TABLES IN SCHEMA`

Marque la publication comme répliquant les changements de toutes les tables de la liste de schémas indiquée, incluant les tables créées plus tard.

Indiquer un schéma quand la publication publie aussi une table avec une liste de colonnes n'est pas supporté.

Seules les tables persistantes et les tables partitionnées présentes dans le schéma seront inclus dans la publication. Les tables temporaires, les tables non journalisées, les tables distantes, les vues matérialisées et les vues standards du schéma ne feront pas partie de la publication.

Quand une table partitionnée est publiée via la publication au niveau schéma, toutes les partitions existantes et futures sont implicitement considérées comme faisant partie de la publication, qu'elles soient dans le schéma de publication ou pas. Donc, même les opérations réalisées directement sur une partition sont aussi publiées via les publications auxquelles ont fait partie ses ancêtres.

`WITH ( `*`parametre_publication`*` [= `*`valeur`*`] [, ... ] )`

Cette clause spécifique les paramètres facultatifs d'une publication. Les paramètres suivants sont acceptés :

`publish` (`string`)

Ce paramètre détermine quelles opération DML seront publiées par la nouvelle publication aux souscripteurs. Le contenu est une liste d'opération séparé par des virgules. Les opérations autorisées sont `insert`, `update`, `delete` et `truncate`. Par défaut toutes les actions sont publiées, et donc la valeur par défaut pour cette option est `'insert, update, delete, truncate'`.

Ce paramètre affecte seulement les opérations DML. En particulier, la synchronisation initiale des données (voir Section 31.7.1) pour la réplication logique ne prend pas en compte ce paramètre lors de la copie de données existantes de la table.

`publish_via_partition_root` (`boolean`)

Ce paramètre contrôle comment les changements sur une table partitionnée (ou dans ses partitions) sont publiées. Quand configuré à `true`, les changements sont publiés en utilisant

l'identité et le schéma de la table racine. Quand configuré à `false` (valeur par défaut), les changements sont publiés en utilisant l'identité et le schéma des partitions individuelles où les changements ont réellement survenu. Activer cette option permet la réplication des modifications dans une table non partitionnée ou dans une table partitionnée dont la structure de la partition diffère de celle du publieur.

Ce paramètre affecté aussi comment les filtres de lignes et les listes de colonnes sont choisis pour les partitions ; voir ci-dessous pour les détails.

Si ceci est activé, les opérations `TRUNCATE` réalisées directement sur les partitions ne sont pas répliquées.

Notes

Si les clauses `FOR TABLE`, `FOR ALL TABLES` ou `FOR TABLES IN SCHEMA` ne sont pas passées, alors la publication commence avec un ensemble vide de tables. C'est utile si les tables ou schémas sont à ajouter ultérieurement.

La création d'une publication ne démarre pas la réplication. Cela définit uniquement un regroupement ainsi qu'un filtre logique pour les futurs souscripteurs.

Pour créer une publication, l'utilisateur lançant la commande doit avec le droit `CREATE` pour la base de données courante. (Bien entendu, les superutilisateurs contournent cette vérification.)

Pour ajouter une table à une publication, l'utilisateur lançant la commande doit avoir les droits de propriétaire de la table. Les clauses `FOR ALL TABLES` et `FOR TABLES IN SCHEMA` nécessitent d'être superutilisateur pour pouvoir les utiliser.

Les tables ajoutées à une publication qui publie les opérations `UPDATE` et/ou `DELETE` doivent avoir `REPLICA IDENTITY` défini. Autrement ces opérations seront interdites sur ces tables.

Toute liste de colonnes doit inclure les colonnes `REPLICA IDENTITY` dans l'ordre pour la publication d'opérations `UPDATE` ou `DELETE`. Il n'y a pas des restrictions de listes de colonnes si la publication publie seulement des opérations `INSERT`.

Une expression de filtre par lignes (par exemple la clause `WHERE`) doit uniquement contenir les colonnes couvertes par `REPLICA IDENTITY`, pour que les opérations `UPDATE` et `DELETE` soient publiées. Pour la publication des opérations `INSERT`, toute colonne peut être utilisée dans l'expression `WHERE`. Le filtre de lignes autorise des expressions simples qui n'ont pas de fonctions définies par l'utilisateur, d'opérateurs définis par l'utilisateur, de types définis par l'utilisateur, de collations définis par l'utilisateur, de fonctions internes non immutables, ou de références à des colonnes systèmes.

Le filtre de lignes sur une table devient redondant si `FOR TABLES IN SCHEMA` est indiqué et que la table appartient au schéma référencé.

Pour les tables partitionnées publiées, le filtre de lignes pour que partition est pris de la table partitionnée publiée si le paramètre de publication `publish_via_partition_root` vaut true ou à partir de la partition elle-même s'il vaut false (valeur par défaut). Voir Section 31.3 pour les détails sur les filtres de lignes. De façon similaire, pour les tables partitionnées publiées, la liste de colonnes pour chaque partition est prise de la table partitionnée publiée si le paramètre de publication `publish_via_partition_root` vaut true, ou à partir de la partition elle-même si le paramètre vaut false.

Pour une commande `INSERT ... ON CONFLICT`, la publication publiera l'opération qui résulte de la commande. Ainsi, en fonction du résultat, cela pourrait être publiée comme un `INSERT` ou un `UPDATE`, ou cela pourrait ne pas être publié du tout.

Pour une commande `MERGE`, la publication publiera un `INSERT`, `UPDATE` ou `DELETE` pour chaque ligne insérée, mise à jour ou supprimée.

`ATTACH`-er une table dans un arbre de partition dont la racine est publiée en utilisant une publication ayant `publish_via_partition_root` configuré to `true` ne résulte pas en la réplication du contenu existant de la table.

Les commandes `COPY ... FROM` sont publiées comme des opérations `INSERT`.

Les opérations DDL ne sont pas publiées.

L'expression de la clause `WHERE` est exécutée avec le rôle utilisé pour la connexion de réplication.

Exemples

Créer une publication qui publie tous les changements sur deux tables :

```
CREATE PUBLICATION mypublication FOR TABLE users, departments;
```

Créer une publication qui publie tous les changements des départements actifs :

```
CREATE PUBLICATION active_departments FOR TABLE departments WHERE
 (active IS TRUE);
```

Créer une publication qui publie tous les changements sur toutes les tables :

```
CREATE PUBLICATION alltables FOR ALL TABLES;
```

Créer une publication qui ne publie que les opérations d'`INSERT` sur une table :

```
CREATE PUBLICATION insert_only FOR TABLE mydata
    WITH (publish = 'insert');
```

Créer une publication qui publie tous les changements des tables `users`, `departments` et tous les changements pour toutes les tables présentes dans le schéma `production` :

```
CREATE PUBLICATION production_publication FOR TABLE users,
 departments, TABLES IN SCHEMA production;
```

Créer une publication qui publie tous les changements pour toutes les tables présentes dans les schémas `marketing` et `sales` :

```
CREATE PUBLICATION sales_publication FOR TABLES IN SCHEMA
 marketing, sales;
```

Créer une publication qui publie tous les changements pour la table `users`, mais réplique uniquement les colonnes `user_id` et `firstname` :

```
CREATE PUBLICATION users_filtered FOR TABLE users (user_id,
 firstname);
```

Compatibilité

`CREATE PUBLICATION` est une extension PostgreSQL au langage SQL.

Voir aussi

ALTER PUBLICATION, DROP PUBLICATION, CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

CREATE ROLE

CREATE ROLE — Définir un nouveau rôle de base de données

Synopsis

```
CREATE ROLE nom [ [ WITH ] option [ ... ] ]

où option peut être :

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE

    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT limite_connexion
    | [ ENCRYPTED ] PASSWORD 'motdepasse' | PASSWORD NULL
    | VALID UNTIL 'heuredate'
    | IN ROLE nom_role [, ...]
    | IN GROUP nom_role [, ...]
    | ROLE nom_role [, ...]
    | ADMIN nom_role [, ...]
    | USER nom_role [, ...]
    | SYSID uid
```

Description

`CREATE ROLE` ajoute un nouveau rôle dans une instance PostgreSQL. Un rôle est une entité qui peut posséder des objets de la base de données et avoir des droits sur la base et ses objets. Il peut être considéré comme un « utilisateur », un « groupe » ou les deux suivant la façon dont il est utilisé. Chapitre 22 et Chapitre 21 donnent de plus amples informations sur la gestion des utilisateurs et l'authentification. Il est nécessaire de posséder le droit `CREATEROLE` ou d'être superutilisateur pour utiliser cette commande.

Les rôles sont définis au niveau de l'instance, et sont donc disponibles dans toutes les bases de l'instance.

Paramètres

nom

Le nom du nouveau rôle.

`SUPERUSER`
`NOSUPERUSER`

Ces clauses définissent si le nouveau rôle est un « superutilisateur » et peut ainsi outrepasser les droits d'accès à la base de données. Le statut de superutilisateur est dangereux et ne doit être utilisé qu'en cas d'absolue nécessité. Seul un superutilisateur peut créer un superutilisateur. `NOSUPERUSER` est la valeur par défaut.

`CREATEDB`
`NOCREATEDB`

Ces clauses précisent le droit de création de bases de données. Si `CREATEDB` est spécifié, l'autorisation est donnée au rôle. `NOCREATEDB`, valeur par défaut, produit l'effet inverse.

`CREATEROLE`
`NOCREATEROLE`

Ces clauses précisent le droit de création, modification, suppression, ajout de commentaire, modification du label de sécurité, d'ajout et de suppression de membres pour les rôles. Voir création de rôle pour plus de détails sur les possibilités offertes par ce droit. `NOCREATEROLE` est la valeur par défaut.

`INHERIT`
`NOINHERIT`

Ces clauses précisent si un rôle « hérite » des droits d'un rôle dont il est membre. Un rôle qui possède l'attribut `INHERIT` peut automatiquement utiliser tout droit détenu par un rôle dont il est membre direct ou indirect. Sans `INHERIT`, l'appartenance à un autre rôle lui confère uniquement la possibilité d'utiliser `SET ROLE` pour acquérir les droits de l'autre rôle ; ils ne sont disponibles qu'après cela. `INHERIT` est la valeur par défaut.

`LOGIN`
`NOLOGIN`

Ces clauses précisent si un rôle est autorisé à se connecter, c'est-à-dire si le rôle peut être donné comme nom pour l'autorisation initiale de session à la connexion du client. Un rôle ayant l'attribut `LOGIN` peut être vu comme un utilisateur. Les rôles qui ne disposent pas de cet attribut sont utiles pour gérer les droits de la base de données mais ne sont pas des utilisateurs au sens habituel du mot. `NOLOGIN` est la valeur par défaut, sauf lorsque `CREATE ROLE` est appelé à travers la commande `CREATE USER`.

`REPLICATION`
`NOREPLICATION`

Ces clauses déterminent si un rôle est un rôle de réplication. Un rôle doit avoir cet attribut (ou être un superutilisateur) pour être capable de se connecter à un serveur en mode réplication (physique ou logique) et pour être capable de créer ou supprimer des slots de réplication. Vous devez être superutilisateur pour créer un nouveau rôle ayant l'attribut `REPLICATION`.

`BYPASSRLS`
`NOBYPASSRLS`

Ces clauses déterminent si un rôle contourne toute politique de sécurité au niveau ligne (RLS). `NOBYPASSRLS` est la valeur par défaut. Vous devez être superutilisateur pour créer un nouveau rôle ayant l'attribut `BYPASSRLS`.

Notez que l'outil pg_dump configure `row_security` à `OFF` par défaut pour s'assurer que tout le contenu d'une table est sauvegardé. Si l'utilisateur exécutant pg_dump n'a pas les droits appropriés, une erreur est renvoyée. Néanmoins, les superutilisateurs et le propriétaire de la table sauvegardée contournent toujours RLS.

`CONNECTION LIMIT` *`limiteconnexion`*

Le nombre maximum de connexions concurrentes possibles pour le rôle, s'il possède le droit de connexion. -1 (valeur par défaut) signifie qu'il n'y a pas de limite. Il est à noter que seules les connexions normales sont soumises à cette limite. Les transactions préparées et les connexions des processus worker n'y sont pas soumis.

[ENCRYPTED] PASSWORD *motdepasse* | PASSWORD NULL

Le mot de passe du rôle. Il n'est utile que pour les rôles ayant l'attribut LOGIN, mais il est possible d'en définir un pour les rôles qui ne l'ont pas. Cette option peut être omise si l'authentification par mot de passe n'est pas envisagée. Si aucun mot de passe n'est spécifié, le mot de passe est NULL et l'authentification par mot de passe échouera toujours pour cet utilisateur. Un mot de passe NULL peut aussi être indiqué explicitement avec PASSWORD NULL.

> **Note**
>
> Indiquer un mot de passe vide configurera aussi le mot de passe à NULL, ce qui n'était pas le cas avant la version 10 de PostgreSQL. Dans les versions précédentes, une chaîne vide pouvait être utilisée ou non, suivant la méthode d'authentification et la version exacte, alors que libpq refuserait de l'utiliser dans tous les cas. Pour lever l'ambiguité, une chaîne vide doit être évité.

Le mot de passe est toujours stocké chiffré dans les catalogues système. Le mot clé ENCRYPTED n'a aucun effet, mais est accepté pour compatibilité descendante. La méthode de chiffrement est déterminée par le paramètre de configuration password_encryption. Si le texte du mot de passe présenté est chiffré avec un format MD5 ou SCRAM, alors il sera stocké tel quel sans prendre en compte password_encryption (puisque le système ne peut pas déchiffrer la chaîne du mot de passe spécifiée, pour le chiffrer dans un format différent). Cela permet de recharger des mots de passe chiffrés durant une opération de sauvegarde / restauration.

VALID UNTIL '*dateheure*'

Cette clause configure la date et l'heure de fin de validité du mot de passe. Sans précision, le mot de passe est indéfiniment valide.

IN ROLE *nom_role*

Cette clause liste les rôles dont le nouveau rôle est membre. Il n'existe pas d'option pour ajouter le nouveau rôle en tant que superutilisateur ; cela se fait à l'aide d'une commande GRANT séparée.

IN GROUP *nom_role*

IN GROUP est un équivalent obsolète de IN ROLE.

ROLE *nom_role*

Cette clause liste les rôles membres du nouveau rôle. Le nouveau rôle devient ainsi un « groupe ».

ADMIN *nom_role*

Cette clause est équivalente à la clause ROLE, à la différence que les rôles nommés sont ajoutés au nouveau rôle avec l'option WITH ADMIN OPTION. Cela leur confère le droit de promouvoir à d'autres rôles l'appartenance à celui-ci.

USER *nom_role*

USER est un équivalent obsolète de ROLE.

SYSID *uid*

La clause SYSID est ignorée, mais toujours acceptée pour des raisons de compatibilité.

Notes

ALTER ROLE est utilisé pour modifier les attributs d'un rôle, et DROP ROLE pour supprimer un rôle. Tous les attributs positionnés par CREATE ROLE peuvent être modifiés par la suite à l'aide de commandes ALTER ROLE.

Il est préférable d'utiliser GRANT et REVOKE. pour ajouter et supprimer des membres de rôles utilisés comme groupe.

La clause VALID UNTIL définit les date et heure d'expiration du mot de passe uniquement, pas du rôle. En particulier, les date et heure d'expiration ne sont pas vérifiées lors de connexions à l'aide de méthodes d'authentification qui n'utilisent pas les mots de passe.

L'attribut INHERIT gouverne l'héritage des droits conférables (c'est-à-dire les droits d'accès aux objets de la base de données et les appartenances aux rôles). Il ne s'applique pas aux attributs de rôle configurés par CREATE ROLE et ALTER ROLE. Par exemple, être membre d'un rôle disposant du droit CREATEDB ne confère pas automatiquement le droit de création de bases de données, même avec INHERIT positionné ; il est nécessaire d'acquérir ce rôle via SET ROLE avant de créer une base de données.

L'attribut INHERIT est la valeur par défaut pour des raisons de compatibilité descendante : dans les précédentes versions de PostgreSQL, les utilisateurs avaient toujours accès à tous les droits des groupes dont ils étaient membres. Toutefois, NOINHERIT est plus respectueux de la sémantique spécifiée dans le standard SQL.

L'attribut CREATEROLE impose quelques précautions. Il n'y a pas de concept d'héritage des droits pour un tel rôle. Cela signifie qu'un rôle qui ne possède pas un droit spécifique, mais est autorisé à créer d'autres rôles, peut aisément créer un rôle possédant des droits différents des siens (sauf en ce qui concerne la création des rôles superutilisateurs). Par exemple, si le rôle « u1 » a le droit CREATEROLE mais pas le droit CREATEDB, il peut toujours créer un rôle possédant le droit CREATEDB. Il est de ce fait important de considérer les rôles possédant l'attribut CREATEROLE comme des superutilisateurs en puissance.

PostgreSQL inclut un programme, createuser, qui possède les mêmes fonctionnalités que CREATE ROLE (en fait, il appelle cette commande) et peut être lancé à partir du shell.

L'option CONNECTION LIMIT n'est vérifiée qu'approximativement. Si deux nouvelles sessions sont lancées à peu près simultanément alors qu'il ne reste qu'un seule possibilité de connexion pour le rôle, il est possible que les deux échouent. De plus, la limite n'est jamais vérifiée pour les superutilisateurs.

Faites attention lorsque vous donnez un mot de passe non chiffré avec cette commande. Le mot de passe sera transmis en clair au serveur. Ce dernier pourrait être tracé dans l'historique des commandes du client ou dans les traces du serveur. Néanmoins, la commande createuser transmet le mot de passe chiffré. De plus, psql contient une commande \password que vous pouvez utiliser pour modifier en toute sécurité votre mot de passe.

Exemples

Créer un rôle qui peut se connecter mais sans lui donner de mot de passe :

```
CREATE ROLE jonathan LOGIN;
```

Créer un rôle avec un mot de passe :

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

(CREATE USER est identique à CREATE ROLE mais implique l'attribut LOGIN.)

Créer un rôle avec un mot de passe valide jusqu'à fin 2006. Une seconde après le passage à 2007, le mot de passe n'est plus valide.

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL
 '2007-01-01';
```

Créer un rôle qui peut créer des bases de données et gérer des rôles :

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

Compatibilité

L'instruction `CREATE ROLE` est définie dans le standard SQL. Ce dernier n'impose que la syntaxe

```
CREATE ROLE nom [ WITH ADMIN nom_role ]
```

La possibilité d'avoir plusieurs superutilisateurs initiaux et toutes les autres options de `CREATE ROLE` sont des extensions PostgreSQL.

Le standard SQL définit les concepts d'utilisateurs et de rôles mais les considère comme des concepts distincts et laisse la spécification des commandes de définition des utilisateurs à l'implémentation de chaque moteur de bases de données. PostgreSQL a pris le parti d'unifier les utilisateurs et les rôles au sein d'une même entité. Ainsi, les rôles ont plus d'attributs optionnels que dans le standard.

Le comportement spécifié par le standard SQL peut être approché en donnant aux utilisateurs l'attribut `NOINHERIT` et aux rôles l'attribut `INHERIT`.

Voir aussi

SET ROLE, ALTER ROLE, DROP ROLE, GRANT, REVOKE, createuser

CREATE RULE

CREATE RULE — Définir une nouvelle règle de réécriture

Synopsis

```
CREATE [ OR REPLACE ] RULE nom AS ON événement
    TO nom_table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | commande | ( commande
 ; commande ... ) }
```

où *événement* fait partie de :

```
    SELECT | INSERT | UPDATE | DELETE
```

Description

`CREATE RULE` définit une nouvelle règle sur une table ou une vue. `CREATE OR REPLACE RULE` crée une nouvelle règle ou remplace la règle si elle existe déjà.

Le système de règles de PostgreSQL autorise la définition d'actions alternatives sur les insertions, mises à jour ou suppressions dans les tables. Pour résumer, une règle impose des commandes supplémentaires lors de l'exécution d'une instruction sur une table donnée. Une règle `INSTEAD`, au contraire, permet de remplacer une commande par une autre, voire d'empêcher sa réalisation. Ce sont également les règles qui sont utilisées pour implanter les vues.

Une règle est un mécanisme de transformation de commandes, une « macro ». La transformation intervient avant l'exécution de la commande. Pour obtenir une opération qui s'exécute indépendamment pour chaque ligne physique, il faut utiliser des triggers. On trouvera plus d'informations sur le système des règles dans Chapitre 41.

Actuellement, les règles `ON SELECT` peuvent seulement être attachées aux vues. (En attacher une à une table convertit la table en vue.) Une règle de ce type doit être nommée `"_RETURN"`, elle doit être une règle `INSTEAD` inconditionnel, et elle doit avoir une action qui consiste en une simple commande `SELECT`. Cette commande définit le contenu visible de la vue. (La vue elle-même est dans les faits une table simple sans stockage.) Une règle de ce type est un détail d'implémentation. Alors qu'une vue peut être redéfinie via `CREATE OR REPLACE RULE "_RETURN" AS ...`, il serait préférable d'utiliser `CREATE OR REPLACE VIEW`.

On peut donner l'illusion d'une vue actualisable (« updatable view ») par la définition de règles `ON INSERT`, `ON UPDATE` et `ON DELETE` (ou tout sous-ensemble de celles-ci) pour remplacer les actions de mises à jour de la vue par des mises à jours des tables adéquates. Si vous voulez supporter `INSERT RETURNING`, alors assurez-vous de placer une clause `RETURNING` adéquate à chacune de ces règles.

Il y a quelques chausse-trappes à éviter lors de l'utilisation de règles conditionnelles pour la mise à jour de vues complexes : à chaque action autorisée sur la vue *doit* correspondre une règle `INSTEAD` inconditionnelle. Si la règle est conditionnelle ou n'est pas une règle `INSTEAD`, alors le système rejette toute tentative de mise à jour, ceci afin d'éviter toute action sur la table virtuelle de la vue. Pour gérer tous les cas utiles à l'aide de règles conditionnelles, il convient d'ajouter une règle inconditionnelle `DO INSTEAD NOTHING` afin de préciser au système qu'il ne recevra jamais de demande de mise à jour d'une table virtuelle. La clause `INSTEAD` des règles conditionnelles peut alors être supprimée ;dans les cas où ces règles s'appliquent, l'action `INSTEAD NOTHING` est utilisée. (Néanmoins, cette méthode ne fonctionne pas actuellement avec les requêtes `RETURNING`.)

> **Note**
>
> Une vue qui est suffisamment simple pour être modifiable automatiquement (voir CREATE VIEW) ne nécessite pas un règle utilisateur pour être modifiable. Bien que vous puissiez de toute façon créer une règle, la transformation automatique de la mise à jour sera généralement plus performante qu'une règle explicite.
>
> Une autre alternative à considérer est l'utilisateur des triggers `INSTEAD OF` (voir CREATE TRIGGER) à la place des règles.

Paramètres

nom

Le nom de la règle à créer. Elle doit être distincte du nom de toute autre règle sur la même table. Les règles multiples sur la même table et le même type d'événement sont appliquées dans l'ordre alphabétique des noms.

événement

`SELECT`, `INSERT`, `UPDATE` ou `DELETE`. Notez qu'un `INSERT` contenant une clause `ON CONFLICT` nepeut pas être utilisé sur des tables ayant une règle `INSERT` ou `UPDATE`. Utilisez plutôt une vue modifiable automatiquement.

nom_table

Le nom (éventuellement qualifié du nom du schéma) de la table ou de la vue sur laquelle s'applique la règle.

condition

Toute expression SQL conditionnelle (renvoyant un type `boolean`). L'expression de la condition ne peut pas faire référence à une table autre que `NEW` ou `OLD` ni contenir de fonction d'agrégat.

`INSTEAD`

Les commandes sont exécutées *à la place de* la commande originale.

`ALSO`

Les commandes sont exécutées *en plus de* la commande originale.

En l'absence de `ALSO` et de `INSTEAD`, `ALSO` est utilisé par défaut.

commande

Commande(s) réalisant l'action de la règle. Les commandes valides sont `SELECT`, `INSERT`, `UPDATE`, `DELETE` ou `NOTIFY`.

À l'intérieur d'une *`condition`* ou d'une *`commande`*, les noms des tables spéciales `NEW` et `OLD` peuvent être utilisés pour faire référence aux valeurs de la table référencée. `NEW` peut être utilisé dans les règles `ON INSERT` et `ON UPDATE` pour faire référence à la nouvelle ligne lors d'une insertion ou à la nouvelle valeur de la ligne lors d'une mise à jour. `OLD` est utilisé dans les règles `ON UPDATE` et `ON DELETE` pour référencer la ligne existant avant modification ou suppression.

Notes

Vous devez être le propriétaire de la table à créer ou sur laquelle vous ajoutez des règles.

Dans une règle pour l'action INSERT, UPDATE ou DELETE sur une vue, vous pouvez ajouter une clause RETURNING qui émet les colonnes de la vue. Cette clause sera utilisée pour calculer les sorties si la règle est déclenchée respectivement par une commande INSERT RETURNING, UPDATE RETURNINGou DELETE RETURNING. Quand la règle est déclenchée par une commande sans clause RETURNING, la clause RETURNING de la règle est ignorée. L'implémentation actuelle autorise seulement des règles INSTEAD sans condition pour contenir RETURNING ; de plus, il peut y avoir au plus une clause RETURNING parmi toutes les règles pour le même événement. (Ceci nous assure qu'il y a seulement une clause RETURNING candidate utilisée pour calculer les résultats.) Les requêtes RETURNING sur la vue seront rejetées s'il n'existe pas de clause RETURNING dans une des règles disponibles.

Une attention particulière doit être portée aux règles circulaires. Ainsi dans l'exemple suivant, bien que chacune des deux définitions de règles soit acceptée par PostgreSQL, la commande SELECT produira une erreur à cause de l'expansion récursive de la règle :

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
 SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
 SELECT * FROM t1;

SELECT * FROM t1;
```

Actuellement, si l'action d'une règle contient une commande NOTIFY, cette commande est exécutée sans condition, c'est-à-dire que NOTIFY est déclenché même si la règle ne s'applique à aucune ligne. Par exemple, dans :

```
CREATE RULE notify_me AS ON UPDATE TO matable DO ALSO NOTIFY
 matable;

UPDATE matable SET name = 'foo' WHERE id = 42;
```

un événement NOTIFY est lancé durant un UPDATE, qu'il y ait ou non des lignes satisfaisant la condition id = 42. Cette restriction pourrait être corrigée dans les prochaines versions.

Compatibilité

CREATE RULE est une extension PostgreSQL, tout comme l'est le système complet de réécriture de requêtes.

Voir aussi

ALTER RULE, DROP RULE

CREATE SCHEMA

CREATE SCHEMA — Définir un nouveau schéma

Synopsis

```
CREATE SCHEMA nom_schéma [ AUTHORIZATION spécification_rôle ]
 [ élément_schéma [ ... ] ]
CREATE SCHEMA AUTHORIZATION spécification_rôle [ élément_schéma
 [ ... ] ]
CREATE SCHEMA IF NOT EXISTS nom_schéma
 [ AUTHORIZATION spécification_rôle ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION spécification_rôle

où spécification_rôle peut valoir :

    user_name
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
```

Description

`CREATE SCHEMA` crée un nouveau schéma dans la base de données. Le nom du schéma doit être unique au sein de la base de données.

Un schéma est essentiellement un espace de noms : il contient des objets nommés (tables, types de données, fonctions et opérateurs) dont les noms peuvent être identiques à ceux d'objets d'autres schémas. Les objets nommés sont accessibles en préfixant leur nom de celui du schéma (on dit alors que le nom est « qualifié » du nom du schéma), ou par la configuration d'un chemin de recherche incluant le(s) schéma(s) désiré(s). Une commande `CREATE` qui spécifie un objet non qualifié crée l'objet dans le schéma courant (le premier dans le chemin de recherche, obtenu par la fonction `current_schema`).

`CREATE SCHEMA` peut éventuellement inclure des sous-commandes de création d'objets dans le nouveau schéma. Les sous-commandes sont traitées à la façon de commandes séparées lancées après la création du schéma. La différence réside dans l'utilisation de la clause `AUTHORIZATION`. Dans ce cas, l'utilisateur est propriétaire de tous les objets créés.

Paramètres

nom_schéma

Le nom du schéma à créer. S'il est oublié, le paramètre *nomutilisateur* est utilisé comme nom de schéma. Le nom ne peut pas débuter par `pg_`, ces noms étant réservés aux schémas du système.

nom_utilisateur

Le nom de l'utilisateur à qui appartient le schéma. Par défaut, il s'agit de l'utilisateur qui exécute la commande. Pour créer un schéma dont le propriétaire est un autre rôle, vous devez être un membre direct ou indirect de ce rôle, ou être un superutilisateur.

élément_schéma

Une instruction SQL qui définit un objet à créer dans le schéma. À ce jour, seules `CREATE TABLE`, `CREATE VIEW`, `CREATE SEQUENCE`, `CREATE TRIGGER` et `GRANT` peuvent être

utilisées dans la commande `CREATE SCHEMA`. Les autres types d'objets sont créés dans des commandes séparées après la création du schéma.

`IF NOT EXISTS`

Ne rien faire (en dehors de l'envoi d'un message d'avertissement) si un schéma de même nom existe déjà. Les sous-commandes *`élément_schéma`* ne peuvent pas être utilisées quand cette option est indiquée.

Notes

Pour créer un schéma, l'utilisateur doit avoir le droit `CREATE` sur la base de données. (Les superutilisateurs contournent cette vérification.)

Exemples

Créer un schéma :

```
CREATE SCHEMA mon_schema;
```

Créer un schéma pour l'utilisateur `joe`, schéma nommé `joe` :

```
CREATE SCHEMA AUTHORIZATION joe;
```

Créer un schéma nommé `test` dont le propriétaire sera l'utilisateur `joe`, sauf s'il existe déjà un schéma `test` (peu importe si `joe` est le propriétaire du schéma existant).

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

Créer un schéma et lui ajouter une table et une vue :

```
CREATE SCHEMA hollywood
    CREATE TABLE films (titre text, sortie date, recompenses
 text[])
    CREATE VIEW gagnants AS
        SELECT titre, sortie FROM films WHERE recompenses IS NOT
 NULL;
```

Les sous-commandes ne sont pas terminées par un point-virgule.

La même chose, autre écriture :

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (titre text, sortie date, recompenses
 text[]);
CREATE VIEW hollywood.gagnants AS
    SELECT titre, sortie FROM hollywood.films WHERE recompenses IS
 NOT NULL;
```

Compatibilité

Le standard SQL autorise une clause `DEFAULT CHARACTER SET` dans `CREATE SCHEMA`, et des types de sous-commandes en plus grand nombre que ceux supportés actuellement par PostgreSQL.

Le standard SQL n'impose pas d'ordre d'apparition des sous-commandes dans `CREATE SCHEMA`. L'implantation actuelle de PostgreSQL ne gère pas tous les cas de références futures dans les sous-commandes. Il peut s'avérer nécessaire de réordonner les sous-commandes pour éviter ces références.

Dans le standard SQL, le propriétaire d'un schéma est également propriétaire de tous les objets qui s'y trouvent. PostgreSQL permet à un schéma de contenir des objets qui n'appartiennent pas à son propriétaire. Cela n'est possible que si le propriétaire du schéma transmet le privilège `CREATE` sur son schéma ou si un superutilisateur choisir d'y créer des objets.

La clause `IF NOT EXISTS` est une extension PostgreSQL.

Voir aussi

ALTER SCHEMA, DROP SCHEMA

CREATE SEQUENCE

CREATE SEQUENCE — Définir un nouveau générateur de séquence

Synopsis

```
CREATE [ { TEMPORARY | TEMP } | UNLOGGED ] SEQUENCE [ IF NOT
 EXISTS ] name
    [ AS type_donnee ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE valeurmin | NO MINVALUE ]
    [ MAXVALUE valeurmax | NO MAXVALUE ]
    [ START [ WITH ] début ]
    [ CACHE cache ]
    [ [ NO ] CYCLE ]
    [ OWNED BY { nom_table.nom_colonne | NONE } ]
```

Description

`CREATE SEQUENCE` crée un nouveau générateur de séquence de nombres. Cela implique la création et l'initialisation d'une nouvelle table à une seule ligne nommée *nom*. Le générateur appartient à l'utilisateur qui exécute la commande.

Si un nom de schéma est donné, la séquence est créée dans le schéma spécifié. Sinon, elle est créée dans le schéma courant. Les séquences temporaires existent dans un schéma spécial, il n'est donc pas utile de préciser un nom de schéma lors de la création d'une séquence temporaire. Le nom de la séquence doit être distinct du nom de toute autre relation (table, séquence, index, vue, vue matérialisée ou table distante) du même schéma.

Après la création d'une séquence, les fonctions `nextval`, `currval` et `setval` sont utilisées pour agir sur la séquence. Ces fonctions sont documentées dans Section 9.17.

Bien qu'il ne soit pas possible de mettre à jour une séquence en accédant directement à la table, une requête telle que :

```
SELECT * FROM nom;
```

peut être utilisée pour examiner les paramètres et l'état courant d'une séquence. En particulier, le champ `last_value` affiche la dernière valeur allouée par une session. (Cette valeur peut être rendue obsolète à l'affichage par des appels effectifs de `nextval` dans des sessions concurrentes.)

Paramètres

`TEMPORARY` ou `TEMP`

Si ce paramètre est spécifié, l'objet séquence n'est créé que pour la session en cours et est automatiquement supprimé lors de la sortie de session. Les séquences permanentes portant le même nom ne sont pas visibles (dans cette session) tant que la séquence temporaire existe, sauf à être référencées par les noms qualifiés du schéma.

`UNLOGGED`

Si indiqué, la séquence est créée comme une séquence non journalisée. Les modifications d'une séquence non journalisée ne sont pas écrites dans les journaux de transactions. Leur état n'est donc

pas garanti en cas de crash : une séquence non journalisée est automatiquement ré-initialisée à son état initial après un crash ou un arrêt non propre. Les séquences non journalisées ne sont pas répliquées vers des serveurs secondaires.

Contrairement aux tables non journalisées, les séquences non journalisées n'offrent pas d'avantages significatifs pour les performances. Cette option a principalement pour objectif d'être associée aux tables non journalisées via des colonnes d'identité ou des colonnes de type `serial` Dans ces cas, cela n'aurait pas de sens d'avoir la séquence journalisée et répliquée mais pas sa table associée.

`IF NOT EXISTS`

Ne renvoie pas une erreur si une relation de même nom existe déjà. Un message d'avertissement est renvoyé dans ce cas. Notez qu'il n'y a aucune garantie que la relation existante ressemble à la séquence qui aurait été créée. Il est même possible que cela ne soit pas une séquence.

nom

Le nom (éventuellement qualifié du nom du schéma) de la séquence à créer.

type_donnee

La clause facultative `AS` *type_donnee* spécifie le type de donnée de la séquence. Les types valides sont `smallint`, `integer`, et `bigint`. `bigint` est le type par défault. Le type de donnée détermine les valeurs minimales et maximales par défaut pour la séquence.

incrément

La clause optionnelle `INCREMENT BY` *incrément* précise la valeur à ajouter à la valeur courante de la séquence pour créer une nouvelle valeur. Une valeur positive crée une séquence ascendante, une valeur négative une séquence descendante. 1 est la valeur par défaut.

valeurmin
`NO MINVALUE`

La clause optionnelle `MINVALUE` *valeurmin* détermine la valeur minimale de la séquence. Si cette clause n'est pas fournie ou si `NO MINVALUE` est spécifié, alors les valeurs par défaut sont utilisées. La valeur par défaut pour une séquence ascendante est 1. La valeur par défaut pour une séquence descendante est la valeur minimale du type de donnée.

valeurmax
`NO MAXVALUE`

La clause optionnelle `MAXVALUE` *valeurmax* détermine la valeur maximale de la séquence. Si cette clause n'est pas fournie ou si `NO MAXVALUE` est spécifié, alors les valeurs par défaut sont utilisées. La valeur par défaut pour une séquence ascendante est la valeur maximale pour le type de données. La valeur par défaut pour une séquence descendante est -1.

début

La clause optionnelle `START WITH` *début* permet à la séquence de démarrer n'importe où. La valeur de début par défaut est *valeurmin* pour les séquences ascendantes et *valeurmax* pour les séquences descendantes.

cache

La clause optionnelle `CACHE` *cache* spécifie le nombre de numéros de séquence à préallouer et stocker en mémoire pour un accès plus rapide. 1 est la valeur minimale (une seule valeur est engendrée à la fois, soit pas de cache) et la valeur par défaut.

`CYCLE`
`NO CYCLE`

L'option `CYCLE` autorise la séquence à recommencer au début lorsque *`valeurmax`* ou *`valeurmin`* sont atteintes, respectivement, par une séquence ascendante ou descendante. Si la limite est atteinte, le prochain nombre engendré est respectivement *`valeurmin`* ou *`valeurmax`*.

Si `NO CYCLE` est spécifié, tout appel à `nextval` alors que la séquence a atteint la valeur maximale (dans le cas d'une séquence ascendante) ou la valeur minimale (dans l'autre cas) retourne une erreur. En l'absence de précision, `NO CYCLE` est la valeur par défaut.

`OWNED BY` *`nom_table.nom_colonne`*
`OWNED BY NONE`

L'option `OWNED BY` permet d'associer la séquence à une colonne de table spécifique. De cette façon, la séquence sera automatiquement supprimée si la colonne (ou la table entière) est supprimée. La table indiquée doit avoir le même propriétaire et être dans le même schéma que la séquence. `OWNED BY NONE`, valeur par défaut, indique qu'il n'y a pas d'association.

Notes

`DROP SEQUENCE` est utilisé pour supprimer une séquence.

Les séquences sont fondées sur l'arithmétique `bigint`, leur échelle ne peut donc pas excéder l'échelle d'un entier sur huit octets (-9223372036854775808 à 9223372036854775807).

Comme les appels à `nextval` et `setval` ne sont jamais annulés, les objets séquences ne peuvent pas être utilisés si des affectations « sans trous » sont nécessaires. Il est possible de construire une affectation sans trou en utilisant des verrous exclusifs sur une table contenant un compteur. Cependant, cette solution est bien plus coûteuse que les objets séquences, tout spécialement si un grand nombre de transactions ont besoin de numéro de séquence en parallèle.

Des résultats inattendus peuvent être obtenus dans le cas d'un paramétrage de *`cache`* supérieur à un pour une séquence utilisée concurrentiellement par plusieurs sessions. Chaque session alloue et cache des valeurs de séquences successives lors d'un accès à la séquence et augmente en conséquence la valeur de `last_value`. Les *`cache`*-1 appels suivants de `nextval` au cours de la session session retourne simplement les valeurs préallouées sans toucher à la séquence. De ce fait, tout nombre alloué mais non utilisé au cours d'une session est perdu à la fin de la session, créant ainsi des « trous » dans la séquence.

De plus, bien qu'il soit garanti que des sessions différentes engendrent des valeurs de séquence distinctes, si l'on considère toutes les sessions, les valeurs peuvent ne pas être engendrées séquentiellement. Par exemple, avec un paramétrage du *`cache`* à 10, la session A peut réserver les valeurs 1..10 et récupérer `nextval`=1 ; la session B peut alors réserver les valeurs 11..20 et récupérer `nextval`=11 avant que la session A n'ait engendré `nextval`=2. De ce fait, un paramétrage de *`cache`* à un permet d'assumer que les valeurs retournées par `nextval` sont engendrées séquentiellement ; avec un *`cache`* supérieur, on ne peut qu'assumer que les valeurs retournées par `nextval` sont tous distinctes, non qu'elles sont réellement engendrées séquentiellement. De plus, `last_value` reflète la dernière valeur réservée pour toutes les sessions, que `nextval` ait ou non retourné cette valeur.

D'autre part, `setval` exécuté sur une telle séquence n'est pas pris en compte par les autres sessions avant qu'elle n'aient utilisé toutes les valeurs préallouées et cachées.

Exemples

Créer une séquence ascendante appelée `serie`, démarrant à 101 :

```
CREATE SEQUENCE serie START 101;
```

Sélectionner le prochain numéro de cette séquence :

```
SELECT nextval('serie');

 nextval
---------
     101
```

Récupérer le prochain numéro d'une séquence :

```
SELECT nextval('serial');

 nextval
---------
     102
```

Utiliser cette séquence dans une commande `INSERT` :

```
INSERT INTO distributors VALUES (nextval('serie'), 'nothing');
```

Mettre à jour la valeur de la séquence après un `COPY FROM` :

```
BEGIN;
COPY distributeurs FROM 'fichier_entrees';
SELECT setval('serie', max(id)) FROM distributeurs;
END;
```

Compatibilité

`CREATE SEQUENCE` est conforme au standard SQL, exception faites des remarques suivantes :

- Obtenir la prochaine valeur se fait en utilisant la fonction `nextval()` au lieu de l'expression standard `NEXT VALUE FOR`.
- La clause `OWNED BY` est une extension PostgreSQL.

Voir aussi

ALTER SEQUENCE, DROP SEQUENCE

CREATE SERVER

CREATE SERVER — Définir un nouveau serveur distant

Synopsis

```
CREATE SERVER [ IF NOT EXISTS ] nom_serveur [ TYPE 'type_serveur' ]
 [ VERSION 'version_serveur' ]
    FOREIGN DATA WRAPPER nom_fdw
    [ OPTIONS ( option 'valeur' [, ... ] ) ]
```

Description

`CREATE SERVER` définit un nouveau serveur de données distantes. L'utilisateur qui définit le serveur devient son propriétaire.

Un serveur distant englobe typiquement des informations de connexion qu'un wrapper de données distantes utilise pour accéder à une ressource externe de données. Des informations de connexions supplémentaires spécifiques à l'utilisateur pourraient être fournies par l'intermédiaire des correspondances d'utilisateur.

Le nom du serveur doit être unique dans la base de données.

La création d'un serveur nécessite d'avoir le droit `USAGE` sur le wrapper de données distant qui est utilisé.

Paramètres

`IF NOT EXISTS`

Ne renvoie pas d'erreur si un serveur du même nom existe déjà. Une note est affichée dans ce cas. Veuillez noter qu'il n'y a aucune garantie que le serveur existant ait quoi que ce soit à voir avec celui qui aurait été créé.

`nom_serveur`

Nom du serveur de données distant qui sera créé.

`type_serveur`

Type de serveur (optionnel), potentiellement utile pour les wrappers de données distantes.

`version_serveur`

Version du serveur (optionnel), potentiellement utile pour les wrappers de données distantes.

`nom_fdw`

Nom du wrapper de données distantes qui gère le serveur.

`OPTIONS ( option 'valeur' [, ... ] )`

Cette clause spécifie les options pour le serveur. Typiquement, les options définissent les détails de connexion au serveur, mais les noms et valeurs réelles dépendent du wrapper de données distantes du serveur.

Notes

Lors de l'utilisation du module dblink, le nom du serveur distant peut être utilisé comme argument de la fonction dblink_connect pour indiquer les paramètres de connexion. Il est nécessaire de disposer du droit USAGE sur le serveur distant pour être capable de l'utiliser de cette façon.

Si le serveur distant accepte d'envoyer le tri, il est essentiel que l'ordre de tri soit identique au serveur local.

Exemples

Créer un serveur `monserveur` qui utilise le wrapper de données distantes `postgres_fdw` :

```
CREATE SERVER monserver FOREIGN DATA WRAPPER postgres_fdw OPTIONS
 (host 'truc', dbname 'trucdb', port '5432');
```

Voir postgres_fdw pour plus de détails.

Créer un serveur `monserveur` qui utilise le wrapper de données distantes `pgsql` :

```
CREATE SERVER monserveur FOREIGN DATA WRAPPER pgsql OPTIONS (host
 'truc', dbname 'trucdb', port '5432');
```

Compatibilité

`CREATE SERVER` est conforme à ISO/IEC 9075-9 (SQL/MED).

Voir aussi

ALTER SERVER, DROP SERVER, CREATE FOREIGN DATA WRAPPER, CREATE FOREIGN TABLE, CREATE USER MAPPING

CREATE STATISTICS

CREATE STATISTICS — définit des statistiques étendues

Synopsis

```
CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
    ON ( expression )
    FROM table_name

CREATE STATISTICS [ IF NOT EXISTS ] nom_statistiques
    [ ( type_statistique [, ... ] ) ]
    ON nom_colonne, nom_colonne [, ...]
    FROM nom_table
```

Description

`CREATE STATISTICS` créera un nouvel objet de suivi des statistiques étendues sur les données de la table, table distante ou vue matérialisée spécifiée. L'objet statistiques sera créé dans la base de données courante et son propriétaire sera l'utilisateur exécutant la commande.

La commande `CREATE STATISTICS` a deux formes basiques. La première forme autorise des statistiques à une variable pour une simple expression à récupérer, fournissant des bénéfices similaires à un index avec expression sans le surcoût de maintenance de l'index. Cette forme ne permet pas d'indiquer le type de statistiques car les différents types de statistiques sont uniquement pour les statistiques avec au moins deux variables. La seconde forme de la commande permet la récupération de statistiques sur plusieurs colonnes et/ou expressions, en indiquant optionnellement les types de statistiques à inclure. Cette forme causera aussi la récupération de statistiques à une variable sur toute expression incluse dans la liste.

Si un nom de schéma est donné (par exemple, `CREATE STATISTICS monschema.mastat ...`) alors l'objet statistiques est crée dans le schéma spécifié. Autrement, il sera crée dans le schéma courant. Le nom de l'objet statistiques doit être différent du nom de tous les autres objets statistiques dans le même schéma.

Paramètres

`IF NOT EXISTS`

Ne renvoie pas d'erreur si un objet statistiques de même nom existe déjà. Une note est affichée dans ce cas. Veuiller noter que seul le nom de l'objet statistiques est pris en compte ici, et non pas le détail de sa définition.

`nom_statistiques`

Le nom (éventuellement qualifié du nom du schéma) de l'objet statistiques devant être crée ; voir ANALYZE pour une explication sur la gestion de l'héritage et des partitions.

`type_statistique`

Un type de statistique multivarié devant être calculé dans cet objet statistiques. Les types actuellement supportés sont `ndistinct`, qui active des statistiques n-distinct, `dependencies` qui active des statistiques de dépendances fonctionnelles, et `mcv` qui active les listes des valeurs les plus fréquentes. Si cette clause est omise, tous les types statistiques supportés sont inclus dans

l'objet statistique. Les statistiques d'expression sur une variable sont construits automatiquement si la définition des statistiques inclue des expressions complexes plutôt que de simples références de colonnes. Pour plus d'informations, voir Section 14.2.2 et Section 75.2.

nom_colonne

Le nom d'une colonne de la table devant être couverte par les statistiques calculées. Ceci est seulement autorisé pour construire des statistiques sur plusieurs variables. Au moins deux noms de colonne ou expressions doivent être indiqués, mais leur ordre n'est pas significatif.

expression

Une expression couverte par les statistiques calculées. Ceci pourrait être utilisé pour construire des statistiques univariées sur une seule expression, ou faire partie d'une liste de plusieurs noms de colonnes et/ou expressions pour construire des statistiques multivariées. Dans ce dernier cas, les statistiques univariées séparées sont construites automatiquement pour chaque expression de la liste.

nom_table

Le nom (éventuellement qualifié du nom du schéma) de la table contenant le(s) colonne(s) sur lesquelles les statistiques sont calculées ; voir ANALYZE pour une explication de la gestion de l'héritage et des partitions.

Notes

Vous devez être le propriétaire de la table pour créer un objet statistiques lisant ses données. Une fois celui-ci créé, le propriétaire de l'objet statistiques est indépendant de la ou les tables sous-jacentes.

Les statistiques d'expression sont par expression et sont similaires à créer un index sur l'expression, sauf qu'elles évitent le surcoût de maintenance de l'index. Les statistiques d'expression sont construites automatiquement pour chaque expression dans la définition de l'objet statistique.

Les statistiques étendues ne sont actuellement pas utilisées par l'optimiseur pour les estimations de sélectivité réalisées pour les jointures de table. Cette limitation sera probablement supprimée dans une version future de PostgreSQL.

Exemples

Créer une table `t1` avec deux colonnes fonctionnellement dépendantes, c'est-à-dire que la connaissance de la valeur de la première colonne est suffisante pour déterminer la valeur de l'autre colonne. Ensuite des statistiques de dépendances fonctionnelles sont construites sur ces colonnes :

```
CREATE TABLE t1 (
    a   int,
    b   int
);

INSERT INTO t1 SELECT i/100, i/500
                FROM generate_series(1,1000000) s(i);

ANALYZE t1;

-- le nombre de lignes correspondantes sera drastiquement sous-
estimé :
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);

CREATE STATISTICS s1 (dependencies) ON a, b FROM t1;
```

```
ANALYZE t1;

-- à présent le nombre de ligne estimé est plus précis :
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);
```

Sans statistiques fonctionnellement dépendantes, l'optimisateur supposera que les deux conditions `WHERE` sont indépendantes, et multiplierait leurs sélectivités pour arriver à une estimation du nombre de lignes bien trop basse. Avec de telles statistiques, l'optimiseur reconnaît que les conditions `WHERE` sont redondantes et ne sous-estime plus le nombre de lignes.

Créer une table `t2` avec deux colonnes parfaitement corrélées (contenant des données identiques), et une liste MCV sur ces colonnes :

```
CREATE TABLE t2 (
    a   int,
    b   int
);

INSERT INTO t2 SELECT mod(i,100), mod(i,100)
                 FROM generate_series(1,1000000) s(i);

CREATE STATISTICS s2 (mcv) ON a, b FROM t2;

ANALYZE t2;

-- valid combination (found in MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 1);

-- invalid combination (not found in MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 2);
```

La liste MCV donne au planificateur des informations plus détaillées à propos des valeurs spécifiques qui apparaissent le plus fréquemment dans la table, de même qu'une borne supérieure sur les sélectivités des combinaisons de valeurs qui n'apparaissent pas dans la table, lui permettant de générer de meilleures estimations dans les deux cas.

Créer une table `t3` avec une seule colonne de type timestamp, et exécuter des requêtes en utilisant des expressions sur cette colonne. Sans les statistiques étendues, le planificateur n'a pas d'information sur la distribution des données pour les expressions, et utilise les estimations par défaut. Le planificateyr ne réalise pas non plus que la valeur de la date tronquée au mois est complètement déterminée par la valeur de la date tronquée au jour. De ce fait, des statistiques sur l'expression et sur ndistinct sont construites sur ces deux expressions :

```
CREATE TABLE t3 (
    a   timestamp
);

INSERT INTO t3 SELECT i FROM
 generate_series('2020-01-01'::timestamp,

 '2020-12-31'::timestamp,
                                              '1 minute'::interval)
 s(i);
```

```
ANALYZE t3;

-- le nombre de lignes correspondantes va être fortement sous-
estimé :
EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('month', a) = '2020-01-01'::timestamp;

EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('day', a) BETWEEN '2020-01-01'::timestamp
                                 AND '2020-06-30'::timestamp;

EXPLAIN ANALYZE SELECT date_trunc('month', a), date_trunc('day', a)
   FROM t3 GROUP BY 1, 2;

-- construction de statistiques ndistinct sur la paire
 d'expressions (des statistiques
-- par expression sont construites automatiquement)
CREATE STATISTICS s3 (ndistinct) ON date_trunc('month', a),
 date_trunc('day', a) FROM t3;

ANALYZE t3;

-- maintenant les estimations de nombre de lignes sont plus
 précises :
EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('month', a) = '2020-01-01'::timestamp;

EXPLAIN ANALYZE SELECT * FROM t3
  WHERE date_trunc('day', a) BETWEEN '2020-01-01'::timestamp
                                 AND '2020-06-30'::timestamp;

EXPLAIN ANALYZE SELECT date_trunc('month', a), date_trunc('day', a)
   FROM t3 GROUP BY 1, 2;
```

Sans statistiques sur l'expression et le ndistinct, le planificateur n'a pas d'informations sur le nombre de valeurs distinctes pour les expressions, et doit se baser sur les estimations par défaut. Les conditions d'égalité et d'intervalle sont supposées avoir une sélectivité de 0,5%, et le nombre de valeurs distinctes dans l'expression est supposé être le même que pour la colonne (i.e. unique). Ceci résulte en une sous-estimation significative du nombre de lignes pour les deux premières requêtes. De plus, le planificateur ne connaît pas la relation entre les expressions, donc il suppose que les deux conditions du `WHERE` et la condition du `GROUP BY` sont indépendentes, et multiplie leurs sélectivités ensemble pour arriver à une sous-estimation sévère du nombre de groupes dans la requête d'agrégat. Ceci est encore plus exacerbé par le manque de statistiques précises sur les expressions, forçant le planificateur à utiliser une estimation par défaut du ndistinct pour l'expression dérivée du ndistinct pour la colonne. Avec de telles statistiques, le planificateur reconnaît que les conditions sont corrélées, et arrive à des estimations bien plus précises.

Compatibilité

Il n'y a pas de commande `CREATE STATISTICS` dans le standard SQL.

Voir aussi

ALTER STATISTICS, DROP STATISTICS

CREATE SUBSCRIPTION

CREATE SUBSCRIPTION — définir une nouvelle souscription

Synopsis

```
CREATE SUBSCRIPTION nom_souscription
    CONNECTION 'conninfo'
    PUBLICATION nom_publication [, ...]
    [ WITH ( param_souscription [= valeur] [, ... ] ) ]
```

Description

`CREATE SUBSCRIPTION` ajoute une nouvelle souscription de réplication logique. Le nom de la souscription doit être différent du nom de toutes les autres souscriptions existantee dans la base.

Une souscription représente une connexion de réplication vers un serveur publiant des données. De ce fait, en plus d'ajouter les définitions dans les catalogues locaux, cette commence crée normalement un slot de réplication sur le publieur.

Un worker de réplication logique sera démarré pour répliquer les données pour la nouvelle souscription à la validation de la transaction dans laquelle cette commande est lancée, sauf si la souscription est désactivé à sa création.

Des informations supplémentaires sur la souscription et la réplication logique dans son ensemble sont également disponible sur Section 31.2 et Chapitre 31.

Paramètres

`nom_souscription`

Le nom de la nouvelle souscriptions.

`CONNECTION '`*`conninfo`*`'`

La chaîne de connexion libpq définissant la façon de se connecter à la base publieur. Pour plus de détails voir Section 34.1.1.

`PUBLICATION` *`publication_name`* `[, ...]`

Nom des publications sur le serveur publiant les données auxquelles souscrire.

`WITH (` *`param_souscription`* `[=` *`valeur`*`] [, ... ] )`

Cette clause indique les paramètres optionnelles pour une souscription.

Les paramètres suivants contrôlent ce qui arrive lors de la création de la souscription :

`connect` (`boolean`)

Indique si la commande `CREATE SUBSCRIPTION` doit se connecter au publieur. La valeur par défaut est `true`. Configuré à `false`, cela forcera les valeurs de `create_slot`, `enabled` et `copy_data` à `false`. (Vous ne pouvez pas combiner `connect` à `false` avec `create_slot`, `enabled` et/ou `copy_data` à `true`.)

Comme aucune connexion n'a lieu quand cette option vaut `false`, aucune table n'est souscrite et, de ce fait, une fois la souscription activée, rien ne sera répliqué. Vous devrez alors exécuter `ALTER SUBSCRIPTION ... REFRESH PUBLICATION` pour que les tables soient prises en compte.

`create_slot` (`boolean`)

Spécifie si la commande devrait créer le slot de réplication sur le serveur publiant les données. La valeur par défaut est `true`. If set to `false`, you are responsible for creating the publisher's slot in some other way.

`enabled` (`boolean`)

Spécifie si la souscription devrait répliquer activement, ou si elle devrait uniquement configurée mais pas démarrée. La valeur par défaut est `true`.

`slot_name` (`string`)

Nom du slot de réplication à utiliser. Par défaut, le nom de la souscription est utilisé comme nom du slot.

Configurer `slot_name` à `NONE` signifie qu'il n'y aura pas de slot de réplication associé à la souscription. Utiliser cela quand vous créerez le slot de réplication manuellement plus tard. Une telle souscription doit également avoir à la fois `enabled` et `create_slot` positionnés à `false`.

Les paramètres suivants contrôlent le comportement de la réplication pour la souscription après sa création :

`binary` (`boolean`)

Spécifie si la souscription réclamera au publieur d'envoyer les données au format binaire (contrairement au texte). La valeur par défaut est `false`. Même quand cette option est activée, seuls les types de données qui ont les fonctions d'envoi et de réception binaire seront transférés en binaire.

Lors d'une réplication entre versions différentes, le publieur pourrait avoir une fonction d'envoi binaire pour certains types de données mais que le souscripteur n'ait pas de fonction de réception binaire pour ce type. Dans ce cas, le transfert de données échouera et l'option `binary` ne pourra pas être utilisée.

`copy_data` (`boolean`)

Spécifie si les données existantes dans les publications qui sont en train d'être souscrites devraient être copiées une fois la réplication démarrée. La valeur par défaut est `true`.

Si les publications contiennent des clauses `WHERE`, elles affecteront les données copiées. Référez-vous à Notes pour les détails.

`streaming` (`boolean`)

Spécifie l'activation du flux de transactions en cours pour cette souscription. Par défaut, toutes les transactions sont entièrement décodées sur le publieur, puis envoyées entièrement au souscripteur.

`synchronous_commit` (`enum`)

La valeur de ce paramètre surcharge le paramètre synchronous_commit pour les processus workers d'application de cette souscription. La valeur par défaut est `off`.

Il est sans danger d'utiliser `off` pour la réplication logique : si le souscripteur perd des transactions à cause d'une synchronisation manquée, les données seront renvoyées par le serveur publiant les données.

Un paramétrage différent peut être appropriée si l'on utilise une réplication logique synchrone. Les workers de réplication logique rapportent les positions d'écriture et de synchronisation au serveur publiant les données, et, en réplication synchrone, ce dernier attendra que la synchronisation ait lieu. Cela veut dire que laisser `synchronous_commit` à `off` sur une souscription en réplication synchrone peut augmenter la latence des `COMMIT` sur le serveur publiant les données. Dans ce scénario, il peut être avantageux de positionner `synchronous_commit` à `local` ou au-dessus.

`two_phase` (`boolean`)

Spécifie si la validation en deux phases est activée pour cette souscription. La valeur par défaut est `false`.

Quand la validation en deux phases est activée, les transactions préparées sont envoyés au souscripteur au moment du `PREPARE TRANSACTION`, et sont traitées comme des transactions en deux phases, y compris sur le souscripteur. Sinon, les requêtes préparées sont envoyés au souscripteur uniquement quand elles sont validées, et elles sont traitées immédiatement après par le souscripteur.

L'implémentation de la validation en deux phases requiert que la réplication ait terminée avec succès la synchronisation initiale des tables. Donc même si `two_phase` est activé pour une souscription, l'état interne de la validation en deux phases reste en attente temporaire jusqu'à ce que la phase d'initialisation se termine. Voir la colonne `subtwophasestate` de `pg_subscription` pour connaître l'état actuel de two-phase.

`disable_on_error` (`boolean`)

Spécifie si la souscription doit être désactivée automatiquement si des erreurs sont détectées par les workers de la souscription lors de la réplication des données du publieur. La valeur par défaut est `false`.

Notes

Voir Section 31.9 pour plus de détail sur comment configurer le contrôle d'accès entre la souscription et l'instance de publication.

Lors de la création d'un slot de réplication (comportement par défaut), `CREATE SUBSCRIPTION` ne peut pas être exécuté à l'intérieur d'un bloc de transaction.

Créer une souscription qui connecte la même instance (par exemple, pour répliquer entre des bases de données de la même instance ou pour répliquer dans la même base de données) réussira seulement si le slot de réplication n'est pas créé dans la même commande. Sinon, l'appel à `CREATE SUBSCRIPTION` va pauser. Pour le faire fonctionner, créer le slot de réplication séparément (en utilisant la fonction `pg_create_logical_replication_slot` avec le nom de plugin `pgoutput`) et créer la souscription en utilisant le paramètre `create_slot = false`. C'est une restriction d'implémentation qui pourrait être supprimé dans une prochaine version.

Si une table dans la publication a une clause `WHERE`, les lignes pour lesquelles l'*`expression`* s'évalue à false ou null ne seront pas publiées. Si la souscription a plusieurs publications dans lesquelles la même table a été publiée avec des clauses `WHERE` différentes, une ligne sera publiée si une des expressions (référant à cette opération de publication) sera publiée si une des expressions (référant à cette opération de publication) sont satisfaites. Dans le cas où différentes clauses `WHERE`, si une des publications n'a pas de clause `WHERE` (référant à cette opération de publication) ou si la publication est déclarée `FOR ALL TABLES` ou `FOR TABLES IN SCHEMA`, les lignes sont toujours publiées quelque soit la définition des autres expressions. Si le souscripteur est une version PostgreSQL avant la 15, puis toute ligne filtrante est ignorée lors de la phase de synchronisation initiale des données. Pour ce cas, l'utilisateur pourrait vouloir considérer la suppression des données copiées initialement qui serait incompatible avec un filtrage précédent. Comme la synchronisation des données ne prend pas en compte le paramètre de publication `publish` lors de la copie des tables existantes, certaines lignes pourraient être copiées sans être répliquées en utilisant DML. Voir Section 31.2.2 pour des exemples.

Les souscriptions ayant plusieurs publications pour lesquels la même table a été publiée avec des listes de colonnes différentes ne sont pas supportées.

Nous autorisons que des publications inexistantes soient indiquées pour que les utilisateurs puissent les ajouter après coup. Ceci signifie que `pg_subscription` peut avoir des publications inexistantes.

Exemples

Créer une souscription à un serveur distant qui réplique les tables dans la publication `mypublication` et `insert_only` et démarre la réplication immédiatement après le commit :

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo
 dbname=foodb'
        PUBLICATION mypublication, insert_only;
```

Crée une souscription vers un serveur distant qui réplique les tables dans la publication `insert_only` et ne commence pas la réplication jusqu'à ce qu'elle soit activée plus tard.

```
CREATE SUBSCRIPTION mysub
         CONNECTION 'host=192.168.1.50 port=5432 user=foo
 dbname=foodb'
        PUBLICATION insert_only
               WITH (enabled = false);
```

Compatibilité

`CREATE SUBSCRIPTION` est une extension PostgreSQL au standard SQL.

Voir aussi

ALTER SUBSCRIPTION, DROP SUBSCRIPTION, CREATE PUBLICATION, ALTER PUBLICATION

CREATE TABLE

CREATE TABLE — Définir une nouvelle table

Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE
 [ IF NOT EXISTS ] nom_table ( [
  { nom_colonne type_donnees [ COMPRESSION méthode_compression ]
 [ COLLATE collation ] [ contrainte_colonne [ ... ] ]
    | contrainte_table
    | LIKE table_source [ option_like ... ] }
    [, ... ]
] )
[ INHERITS ( table_parent [, ... ] ) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { nom_colonne |
 ( expression ) } [ COLLATE collation ] [ opclass ] [, ... ] ) ]
[ USING methode ]
[ WITH ( parametre_stockage [= valeur] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE nom_tablespace ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ]
 TABLE nom_table
    OF nom_type [ (
  { nom_colonne [ WITH OPTIONS ] [ contrainte_colonne [ ... ] ]
    | contrainte_table }
    [, ... ]
) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { nom_colonne |
 ( expression ) } [ COLLATE collation ] [ opclass ] [, ... ] ) ]
[ USING methode ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE nom_tablespace ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE
 [ IF NOT EXISTS ] table_name
    PARTITION OF parent_table [ (
  { nom_colonne [ WITH OPTIONS ] [ contrainte_colonne [ ... ] ]
    | table_constraint }
    [, ... ]
) ] { FOR VALUES spec_limites_partition | DEFAULT }
[ PARTITION BY { RANGE | LIST } ( { nom_colonne | ( expression ) }
 [ COLLATE collation ] [ opclass ] [, ... ] ) ]
[ USING methode ]
[ WITH ( parametre_stockage [= valeur] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE nom_tablespace ]

où contrainte_colonne
peut être :

[ CONSTRAINT nom_contrainte ]
{ NOT NULL | NULL |
  CHECK ( expression ) [ NO INHERIT ] |
```

```
  DEFAULT expression_par_défaut |
  GENERATED ALWAYS AS ( expr_generation ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
 [ ( options_sequence ) ] |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] paramètres_index |
  PRIMARY KEY parametres_index |
  EXCLUDE [ USING methode_index ] ( élément_exclude WITH opérateur
 [, ... ] ) paramètres_index [ WHERE ( prédicat ) ] |
  REFERENCES table_reference [ ( colonne_reference ) ] [ MATCH FULL
| MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action_referentielle ] [ ON
 UPDATE action_referentielle ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
 IMMEDIATE ]

et option_like peut
valoir :

{ INCLUDING | EXCLUDING } { COMMENTS | COMPRESSION | CONSTRAINTS |
 DEFAULTS | GENERATED | IDENTITY | INDEXES | STATISTICS | STORAGE |
 ALL }

and spec_limites_partition is:

IN ( expr_limite_partition [, ...] ) |
FROM ( { expr_limite_partition | MINVALUE | MAXVALUE } [, ...] )
  TO ( { expr_limite_partition | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS litéral_numérique, REMAINDER litéral_numérique )

et contrainte_table :

[ CONSTRAINT nom_contrainte ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE [ NULLS [ NOT ] DISTINCT ] ( nom_colonne
 [, ... ] ) paramètres_index |
  PRIMARY KEY ( nom_colonne [, ... ] ) parametres_index |
  EXCLUDE [ USING méthode_index ] ( élément_exclude WITH opérateur
 [, ... ] ) paramètres_index [ WHERE ( prédicate ) ] |
  FOREIGN KEY ( nom_colonne [, ...
] ) REFERENCES table_reference [ (
colonne_reference [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON
 DELETE action_referentielle ] [ ON UPDATE action_referentielle ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY
 IMMEDIATE ]

Les paramètres_index dans les
contraintes UNIQUE, PRIMARY KEY et
EXCLUDE sont :

[ INCLUDE ( nom_colonne [, ... ] ) ]
[ WITH ( paramètre_stockage [= valeur] [, ... ] ) ]
[ USING INDEX TABLESPACE nom_tablespace ]

élément_exclude dans une
contrainte EXCLUDE peut valoir :
```

```
{ nom_colonne | ( expression ) } [ COLLATE collation ] [ opclass
 [ ( parametre_opclass = valeur [, ... ] ) ] ] [ ASC | DESC ]
 [ NULLS { FIRST | LAST } ]

referential_actiondans une contrainte FOREIGN KEY/REFERENCES vaut :

{ NO ACTION | RESTRICT | CASCADE | SET NULL [ ( nom_colonne
 [, ... ] ) ] | SET DEFAULT [ ( nom_colonne [, ... ] ) ] }
```

Description

CREATE TABLE crée une nouvelle table initialement vide dans la base de données courante. La table appartient à l'utilisateur qui exécute cette commande.

Si un nom de schéma est donné (par exemple, CREATE TABLE monschema.matable ...), alors la table est créée dans le schéma spécifié. Dans le cas contraire, elle est créée dans le schéma courant. Les tables temporaires existent dans un schéma spécial, il n'est donc pas nécessaire de fournir un nom de schéma lors de la création d'une table temporaire. Le nom de la table doit être distinct du nom des autres relations (table, séquence, index, vue, vue matérialisée ou table distante) dans le même schéma.

CREATE TABLE crée aussi automatiquement un type de données qui représente le type composé correspondant à une ligne de la table. Ainsi, les tables doivent avoir un nom distinct de tout type de données du même schéma.

Les clauses de contrainte optionnelles précisent les contraintes (ou tests) que les nouvelles lignes ou les lignes mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Une contrainte est un objet SQL qui aide à définir l'ensemble des valeurs valides de différentes façons.

Il existe deux façons de définir des contraintes : celles de table et celles de colonnes. Une contrainte de colonne fait partie de la définition de la colonne. Une définition de contrainte de tables n'est pas liée à une colonne particulière et peut englober plusieurs colonnes. Chaque contrainte de colonne peut être écrite comme une contrainte de table ; une contrainte de colonne n'est qu'un outil de notation utilisé lorsque la contrainte n'affecte qu'une colonne.

Pour pouvoir créer une table, vous devez avoir le droit USAGE sur les types de chaque colonne ou sur le type indiqué dans la clause OF.

Paramètres

TEMPORARY ou TEMP

La table est temporaire. Les tables temporaires sont automatiquement supprimées à la fin d'une session ou, optionnellement, à la fin de la transaction en cours (voir ON COMMIT ci-dessous). Le paramètre search_path par défaut inclut tout d'abord le schéma temporaire et donc les tables permanentes existantes nommées de façon identiques ne sont pas choisies pour les nouveaux plans tant que la table temporaire existe sauf s'il y est fait référence par leur nom qualifié du schéma. Tous les index créés sur une table temporaire sont automatiquement temporaires.

Le démon autovacuum ne peut pas accéder et, du coup, ne peut pas exécuter un VACUUM ou un ANALYZE sur les tables temporaires. Pour cette raison, les opérations VACUUM et ANALYZE doivent être traitées via des commandes SQL de session. Par exemple, si une table temporaire doit être utilisée dans des requêtes complexes, il est raisonnable d'exécuter ANALYZE sur la table temporaire après qu'elle ait été peuplée.

On peut éventuellement écrire GLOBAL ou LOCAL avant TEMPORARY ou TEMP. Cela ne fait pas de différence dans PostgreSQL (cf. Cela ne fait actuellement pas de différence dans PostgreSQL et est obsolète ; voir Compatibilité).

`UNLOGGED`

Si spécifié, la table est créée en tant que table non tracée. Les données écrites dans ce type de table ne sont pas écrites dans les journaux de transactions (voir Chapitre 30), ce qui les rend considérablement plus rapides que les tables ordinaires. Néanmoins, elles ne sont pas sûres en cas d'arrêt brutal : une table non tracée est automatiquement vidée après un arrêt brutal. Le contenu d'une table non tracée n'est pas répliqué vers les serveurs en attente. Tout index créé sur une table non tracée est aussi automatiquement non tracé.

Si c'est indiqué, toute séquences créée avec une table non journalisée (pour une colonne d'identité ou pour une colonne de type `serial`) est aussi créée en tant que non journalisée.

`IF NOT EXISTS`

N'affiche pas d'erreur si une relation de même nom existe déjà. Un message de niveau notice est retourné dans ce cas. Notez qu'il n'existe aucune garantie que la relation existante ressemble à celle qui devait être créée..

`nom_table`

Le nom (éventuellement qualifié du nom du schéma) de la table à créer.

`OF` *`nom_type`*

Crée une *table typée*, qui prend sa structure à partir du type composite spécifié (son nom peut être qualifié du schéma). Une table typée est liée à son type ; par exemple, la table sera supprimée si le type est supprimé (avec `DROP TYPE ... CASCADE`).

Quand une table typée est créée, les types de données des colonnes sont déterminés par le type composite sous-jacent et ne sont pas indiqués par la commande `CREATE TABLE`. Mais la commande `CREATE TABLE` peut ajouter des valeurs par défaut et des contraintes à la table. Elle peut aussi indiquer des paramètres de stockage.

`nom_colonne`

Le nom d'une colonne de la nouvelle table.

`type_données`

Le type de données de la colonne. Cela peut inclure des spécificateurs de tableaux. Pour plus d'informations sur les types de données supportés par PostgreSQL, on se référera à Chapitre 8.

`COLLATE` *`collation`*

La clause `COLLATE` affecte un collationnement à une colonne (qui doit être d'un type de données collationnable). Sans information, le collationnement par défaut du type de données de la colonne est utilisé.

`COMPRESSION` *`méthode_compression`*

La clause `COMPRESSION` configure la méthode de compression pour la colonne. La compression est supportée uniquement pour les types de données de taille variable, et est utilisée uniquement quand le mode de stockage de la colonne est `main` ou `extended`.(voir ALTER TABLE pour des informations sur les modes de stockage des colonnes.) Configurer cette propriété sur une table partitionnée n'a pas d'effet direct car de telles tables n'ont pas de stockage propre, mais la valeur configurée sera héritée par les nouvelles partitions. Les méthodes de compression supportées sont `pglz` et `lz4`. (`lz4` est disponible seulement si `--with-lz4` a été utilisé lors de la compilation de PostgreSQL.) De plus, *`méthode_compression`* peut valoir `default` pour indiquer explicitement le comportement par défaut, qui est de consulter la configuration du paramètre default_toast_compression au moment de l'insertion des données pour déterminer la méthode à utiliser.

`INHERITS ( table_parent [, ... ])`

La clause optionnelle `INHERITS` indique une liste de tables dont les colonnes sont automatiquement héritées par la nouvelle table. Les tables parents peuvent être des tables standards ou des tables distantes.

L'utilisation d'`INHERITS` crée une relation persistante entre la nouvelle table enfant et sa table parent. Les modifications de schéma du(des) parent(s) se propagent normalement aux enfants et, par défaut, les données de la table enfant sont incluses dans les parcours de(s) parent(s).

Si un même nom de colonne existe dans plusieurs tables parentes, une erreur est rapportée, à moins que les types de données des colonnes ne correspondent dans toutes les tables parentes. S'il n'y a pas de conflit, alors les colonnes dupliquées sont assemblées pour former une seule colonne dans la nouvelle table. Si la liste des noms de colonnes de la nouvelle table contient un nom de colonne hérité, le type de données doit correspondre à celui des colonnes héritées et les définitions des colonnes sont fusionnées. Si la nouvelle table spécifie explicitement une valeur par défaut pour la colonne, cette valeur surcharge toute valeur par défaut héritée. Dans le cas contraire, les parents qui spécifient une valeur par défaut doivent tous spécifier la même, sans quoi une erreur est rapportée.

Les contraintes `CHECK` sont fusionnées, dans les grandes lignes, de la même façon que les colonnes : si des tables parentes multiples et/ou la nouvelle définition de table contient des contraintes `CHECK` de même nom, ces contraintes doivent toutes avoir la même expression de vérification, ou une erreur sera retournée. Les contraintes qui ont le même nom et la même expression seront fusionnées en une seule. Une contrainte marquée `NO INHERIT` dans une table parent ne sera pas prise en compte. Notez qu'une contrainte `CHECK` non nommée dans la nouvelle table ne sera jamais fusionnée puisqu'un nom unique lui sera toujours affecté.

Les paramètres `STORAGE` de la colonne sont aussi copiés des tables parents.

Si une colonne de la table parente est une colonne d'identité, cette propriété n'est pas héritée. Une colone dans la table enfant peut être déclarée comme colonne d'identité si l'on veut.

`PARTITION BY { RANGE | LIST | HASH } ( { nom_colonne | ( expression ) } [ opclass ] [, ...] )`

La clause facultative `PARTITION BY` spécifie une statégie pour partitionner la table. La table ainsi crée est appelée table *partitionnée*. La liste de colonnes ou d'expressions entre parenthèses forme la *clé de partitionnement* de la table. Quand un partitionnement par intervalle ou hachage est utilisé, la clé de partitionnement peut inclure de multiples colonnes ou expressions (jusqu'à 32, mais cette limite peut être modifiée lors de la compilation de PostgreSQL.), mais pour le partitionnement par liste, la clé de partitionnement doit être constituée d'une seule colonne ou expression.

Les partitionnements par intervalle ou par liste nécessitent une classe d'opérateurs btree, alors que le partitionnement par hachage exige une classe d'opérateurs hash. Si aucune classe d'opérateurs n'est précisée explicitement, la classe d'opérateurs par défaut du type approprié sera utilisée ; si aucune classe d'opérateurs n'existe, une erreur sera levée. Si le partitionnement par hachage est utilisé, la classe d'opérateurs utilisée doit implémenter la fonction de support 2 (voir Section 38.16.3 pour les détails).

Une table partitionnée est divisée en sous tables (appelées partitions), qui sont créées en utilisant des commandes `CREATE TABLE` séparées. La table partitionnée est elle-même vide. Une ligne de données insérée dans la table est redirigée vers une partition en fonction de la valeur des colonnes ou expressions de la clé de partitionnement. S'il n'existe pas de partition correspondant aux valeurs de la nouvelle ligne, une erreur sera levée.

Les tables partitionnées ne supportent pas les contraintes `EXCLUDE` ; cependant vous pouvez définir ces contraintes sur des partitions individuelles.

Voir Section 5.11 pour plus de détails sur le partitionnement des tables.

`PARTITION OF` *`table_parent`* `{ FOR VALUES` *`spec_limites_partition`* `| DEFAULT }`

Crée la table comme une *partition* de la table parente spécifiée. La table peut être créée, soit comme une partition pour des valeurs spécifiques avec `FOR VALUES`, soit comme la partition par défaut avec `DEFAULT`. Tout index, toute contrainte et tout trigger de niveau ligne défini par l'utilisateur existant dans la table parent est clonée sur la nouvelle partition.

Le paramètre *`spec_limites_partition`* doit correspondre à la méthode et à la clé de partitionnement de la table parent, et ne doit pas déborder sur toute partition existante du parent. La forme avec `IN` est utilisée pour le partitionnement de liste, la forme avec `FROM` et `TO` est utilisée pour le partitionnement par intervalles, et la forme avec `WITH` est utilisée pour le partitionnement par hachage.

`expr_limite_partition` peut être n'importe quelle expression sans variable (les sous-requêtes, fonctions de fenêtrage, fonctions d'aggrégation ou toute fonction renvoyant une relation ne sont pas autorisées). Son type de donnée doit être le même que celui de la clé de partitionnement. L'expression étant évaluée une seule fois, au moment de la création de la table, il est possible d'utiliser des expressions volatiles comme `CURRENT_TIMESTAMP` .

Lors de la création d'une partition en liste, `NULL` peut être indiquer pour signifier que la partition permet à la colonne de clé de partitionnement d'être NULL. Néanmoins, il ne peut y avoir plus d'une partition de ce type pour une table parent donnée. `NULL` n'est pas accepté pour les partitions par intervalle.

Lorsqu'une partition de type intervalle est créée, la borne inférieure spécifiée avec `FROM` est une borne inclusive, alors que la borne supérieure spécifiée avec `TO` est une borne exclusive. C'est-à-dire que les valeurs spécifiées dans la liste `FROM` sont des valeurs valides des colonnes correspondantes de la clé de partitionnement pour cette partition, alors que celles dans la liste `TO` ne le sont pas. Notez que ceci doit être compris suivant les règles de la comparaison de lignes (Section 9.24.5). Par exemple, étant donné `PARTITION BY RANGE (x,y)`, une limite de partition `FROM (1, 2) TO (3, 4)` permet `x=1` pour tout `y>=2`, `x=2` avec tout `y` non NULL, et `x=3` avec tout `y<4`.

Les valeurs spéciales `MINVALUE` et `MAXVALUE` peuvent être utilisées lors de la création d'une partition par intervalles pour indiquer qu'il n'y a pas de limite basse ou haute sur la valeur de la colonne. Par exemple, une partition définie comme utilisant `FROM (MINVALUE) TO (10)` accepte toutes les valeurs inférieures à 10, et une partition définie en utilisant `FROM (10) TO (MAXVALUE)` accepte toutes les valeurs supérieures ou égales à 10.

Lors de la création d'une partition par intervalles impliquant plus d'une colonne, il est aussi sensé d'utiliser `MAXVALUE` comme élément de la limite basse et `MINVALUE` comme élément de limite haute. Par exemple, une partition définie en utilisant `FROM (0, MAXVALUE) TO (10, MAXVALUE)` accepte toute ligne où la première colonne de la clé de partitionnement est supérieure à zéro et inférieure ou égale à dix. De la même façon, une partition définie en utilisant `FROM ('a', MINVALUE) TO ('b', MINVALUE)` accepte toute ligne où la première colonne de la clé de partitionnement commence avec la lettre a.

Notez que si `MINVALUE` ou `MAXVALUE` est utilisé pour une colonne d'une limite de partitionnement, la même valeur doit être utilisée pour toutes les colonnes suivantes. Par exemple, `(10, MINVALUE, 0)` n'est pas une limite valide. Vous devriez écrire `(10, MINVALUE, MINVALUE)`.

De plus, notez que certains types d'éléments, tels que `timestamp`, ont une notion d'infinité, qui est simplement une autre valeur qui peut être enregistré. Ceci est différent de `MINVALUE` et `MAXVALUE`, qui ne sont pas de vraies valeurs pouvant être enregistrées, mais plutôt une façon de dire que la valeur est sans limite. `MAXVALUE` peut être vu comme étant supérieur à toute autre valeur, ceci incluant infinity et `MINVALUE` comme étant inférieure à toute autre valeur, ceci

incluant moins infinity. De ce fait, l'intervalle `FROM ('infinity') TO (MAXVALUE)` n'est pas un intervalle vide. Il autorise le stockage d'une seule valeur -- "infinity".

Quand une partition par liste de valeurs est créée, `NULL` peut être spécifié pour dire que la partition autorise la colonne de la clé de partitionnement à être NULL. Cepdnant, il ne peut pas y avoir plus d'une partition par liste de ce type pour une même table parente. `NULL` ne peut pas être utilisé pour les partitions par intervalles.

Si `DEFAULT` est spécifié, la table sera créée comme la partition par défaut de la table parente. Celle option n'est pas disponible pour les tables partitionnées par hachage. Une clé de partition qui ne passe dans aucune autre partition de la table parente sera orientée vers la partition par défaut.

Si une table possède une partition `DEFAULT` et qu'on lui ajoute une nouvelle partition, la partition par défaut doit être parcourue pour vérifier qu'elle ne contient aucune ligne qui appartient normalement à la nouvelle partition. Si la partition par défaut contient un grand nombre de lignes, cela peut être long. Ce parcours peut être évité si la partition par défaut est une table étrangère ou possède une contrainte prouvant qu'elle ne peut contenir des lignes qui devraient appartenir à la nouvelle partition.

À la création d'une partition par hachage, un diviseur et un reste doivent être spécifiés. Le diviseur doit être un entier positif, et le reste un entier non négatif inférieur au diviseur. Typiquement, au début de la mise en place d'un partitionnement par hachage, vous devrez choisir un diviseur égal au nombre de partitions et assigner à chaque table le même diviseur et un reste différent (voir les exemples plus bas). Cependant, il n'est pas obligatoire que chaque partition ait le même diviseur, juste que chaque diviseur apparaissant dans une table partitionnée par hachage soit un facteur du diviseur immédiatement supérieur. Cela permet d'augmenter le nombre de partitions de manière incrémentale sans avoir besoin de déplacer toutes les données d'un coup. Par exemple, supposons que vous ayez une table partitionnée par hachage avec 8 partitions, toutes de diviseur 8, mais que vous trouvez qu'il faille augmenter le nombre de partitions à 16. Vous pouvez détacher une des partitions de diviseur 8, créer deux nouvelles partitions de diviseur 16 couvrant la même partie de l'espace des clés (une avec un reste égal au reste de la partition détachée, l'autre avec un reste de cette valeur plus 8), et les peupler avec les données. Vous pouvez répéter ceci -- peut-être plus tard -- pour chaque partition de diviseur 8 jusqu'à ce qu'il n'y en ait plus. Bien que cela implique de grands mouvements de données à chaque étape, c'est toujours mieux qu'avoir à créer toute une nouvelle table et d'avoir à déplacer toutes les données en une seule fois.

Une partition doit avoir les mêmes noms de colonne et types de données que la table partitionnée à laquelle elle appartient. Toute modification du type ou du nom d'une colonne d'une table partitionnée sera automatiquement propagée à toutes les partitions. Les contraintes de type `CHECK` seront automatiquement héritées pour chaque partition, mais il est possible de définir des contraintes de type `CHECK` supplémentaires sur certaines partitions. Si une contrainte sur une partition porte le même nom qu'une contrainte ajoutée sur la table mère, elle sera fusionnée avec la contrainte de la table parent. Il est possible de définir des valeurs par défaut différentes pour chaque partition. Notez que la valeur par défaut d'une partition ne s'applique pas quand l'insertion de la ligne se fait via la table partitionnée.

Les lignes insérées dans une table partitionnées seront automatiquement redirigées vers la bonne partition. Si aucune des partitions existantes ne convient, une erreur sera levée.

Les opérations telles que `TRUNCATE` qui n'affectent normalement une table ainsi que tous ses enfants hérités seront cascadées sur toutes les partitions, mais peuvent aussi être effectuées sur une partition individuelle.

Notez que créer une partition en utilisant `PARTITION OF` nécessite de prendre un verrou `ACCESS EXCLUSIVE` sur la table partitionnée parente. De même, supprimer une partition avec `DROP TABLE` nécessite de prendre un verrou `ACCESS EXCLUSIVE` sur la table parente. Il est possible d'utiliser `ALTER TABLE ATTACH/DETACH PARTITION` pour réaliser ces opérations avec un verrou plus faible, réduisant ainsi les interférences avec les opérations concurrentes sur la table partitionnée.

`LIKE` *`table_source`* [*`option_like`* ...]

La clause `LIKE` spécifie une table à partir de laquelle la nouvelle table copie automatiquement tous les noms de colonnes, leur types de données et les contraintes non NULL.

Contrairement à `INHERITS`, la nouvelle table et la table originale sont complètement découplées à la fin de la création. Les modifications sur la table originale ne sont pas appliquées à la nouvelle table et les données de la nouvelle table sont pas prises en compte lors du parcours de l'ancienne table.

De plus, contrairement à `INHERITS`, les colonnes et les contraintes copiées par `LIKE` ne sont pas assemblées avec des colonnes et des contraintes nommées de façon similaire. Si le même nom est indiqué explicitement ou dans une autre clause `LIKE`, une erreur est rapportée.

La clause facultative *`like_option`* permet de spécifier quelle propriété supplémentaire de la table originale doit être copiée. Spécifier `INCLUDING` copie la propriété, spécifier `EXCLUDING` ne copie pas la propriété. `EXCLUDING` est la valeur par défaut. Si plusieurs clauses `INCLUDING` et `EXCLUDING` sont indiquées pour le même type d'objet, la dernière clause est utilisée. Les options disponibles sont :

`INCLUDING COMMENTS`

Les commentaires sur les colonnes, contraintes de vérification, index et statistiques étendues seront copiés. Le comportement par défaut exclue les commentaires, ce qui fait que les objets correspondants de la nouvelle table n'ont pas de commentaire.

`INCLUDING COMPRESSION`

La méthode de compression des colonnes sera copiée. Le comportement par défaut est d'exclure les méthodes de compression, résultant en des colonnes ayant la méthode de compression par défaut.

`INCLUDING CONSTRAINTS`

Les contraintes de type `CHECK` seront copiées. Aucune distinction ne sera faite entre les contraintes de colonnes et les contraintes de table. Tout contrainte de non nullité est systématiquement recopiée sur la nouvelle table.

`INCLUDING DEFAULTS`

Les valeurs par défaut sur les colonnes seront recopiées dans la nouvelle table. Si cette clause n'est pas ajoutée, les valeurs par défaut ne sont pas recopiées, ce qui fait que les colonnes de la nouvelle table n'ont pas de valeur par défaut. Il est important de noter que copier une valeur par défaut qui fait appel à une fonction qui peut modifier la base de données, comme la fonction `nextval`, peut créer un lien entre la table originale et les nouvelles tables.

`INCLUDING GENERATED`

Toute définition d'une colonne générée sera copiée. Par défaut, les nouvelles colonnes copiées à partir de colonnes générées sont de simples colonnes normales.

`INCLUDING IDENTITY`

Toute spécification d'une colonne d'identité sera copiée. Une nouvelle séquence est créée pour chaque colonne d'identité de la nouvelle table, séparément des séquences associées à la table d'origine.

`INCLUDING INDEXES`

Les index, les contraintes de type `PRIMARY KEY`, `UNIQUE` et `EXCLUDE` de la table d'origine seront créés sur la nouvelle table. Les noms pour les nouveaux index et les nouvelles

contraintes sont choisis automatiquement suivant des règles fixes, quel que soit leur nom sur la table d'origine. (Ce comportement permet d'éviter d'essayer de créer un index portant un nom déjà utilisé.)

`INCLUDING STATISTICS`

Les statistiques étendues sont copiées sur la nouvelle table.

`INCLUDING STORAGE`

Les spécifications de `STORAGE` sont copiées sur la nouvelle colonne. Le comportement par défaut exclue la clause de `STORAGE`, ce qui fait que, par défaut, les colonnes de la nouvelle table n'ont pas de définition particulière pour le stockage. Pour plus d'informations sur la clause `STORAGE`, voir Section 73.2.

`INCLUDING ALL`

`INCLUDING ALL` permet de sélectionner toutes les options individuelles ci-dessus d'un seul coup. (Il peut être utile d'ajouter des options individuelles `EXCLUDING` après une clause `INCLUDING ALL` pour sélectionner toutes les options sauf certaines.)

La clause `LIKE` peut aussi être utilisée pour copier les définitions de colonne des vues, tables distantes et types composites. Les options inapplicables (comme `INCLUDING INDEXES` à partir d'une vue) sont ignorées.

`CONSTRAINT` *`nom_contrainte`*

Le nom optionnel d'une contrainte de colonne ou de table. Si la contrainte est violée, le nom de la contrainte est présente dans les messages d'erreur. Donc les noms de contraintes comme `col doit être positive` peut être utilisés pour communiquer des informations utiles aux applications clients. (Des doubles guillemets sont nécessaires pour indiquer les noms des contraintes qui contiennent des espaces.) Si un nom de contrainte n'est pas donné, le système en crée un.

`NOT NULL`

Interdiction des valeurs NULL dans la colonne.

`NULL`

Les valeurs NULL sont autorisées pour la colonne. Comportement par défaut.

Cette clause n'est fournie que pour des raisons de compatibilité avec les bases de données SQL non standard. Son utilisation n'est pas encouragée dans les nouvelles applications.

`CHECK (` *`expression`* `) [ NO INHERIT ]`

La clause `CHECK` spécifie une expression de résultat booléen que les nouvelles lignes ou celles mises à jour doivent satisfaire pour qu'une opération d'insertion ou de mise à jour réussisse. Les expressions de résultat TRUE ou UNKNOWN réussissent. Si une des lignes de l'opération d'insertion ou de mise à jour produit un résultat FALSE, une exception est levée et la base de données n'est pas modifiée. Une contrainte de vérification sur une colonne ne fait référence qu'à la valeur de la colonne tandis qu'une contrainte sur la table fait référence à plusieurs colonnes.

Actuellement, les expressions `CHECK` ne peuvent ni contenir des sous-requêtes ni faire référence à des variables autres que les colonnes de la ligne courante (voir Section 5.4.1). La colonne système `tableoid` peut être référencé contrairement aux autres colonnes systèmes.

Une contrainte marquée `NO INHERIT` ne sera pas propagée aux tables filles.

Quand une table a plusieurs contraintes `CHECK`, elles seront testées pour chaque ligne dans l'ordre alphabétique de leur nom, après la vérification des contraintes `NOT NULL`. (Les versions de

PostgreSQL antérieures à la 9.5 ne respectaient pas d'ordre de déclenchement particulier pour les contraintes CHECK.)

DEFAULT *expression_par_défaut*

La clause DEFAULT, apparaissant dans la définition d'une colonne, permet de lui affecter une valeur par défaut. La valeur est une expression libre de variable (en particulier, les références croisées aux autres colonnes de la table courante ne sont pas autorisées). Les sous-requêtes ne sont pas autorisées non plus. Le type de données de l'expression par défaut doit correspondre au type de données de la colonne.

L'expression par défaut est utilisée dans les opérations d'insertion qui ne spécifient pas de valeur pour la colonne. S'il n'y a pas de valeur par défaut pour une colonne, elle est NULL.

GENERATED ALWAYS AS (*generation_expr*) STORED

Cette clause crée la colonne comme une *colonne générée*. La colonne n'est pas accessible en écriture. À la lecture de cette colonne, l'expression spécifiée calculée sera renvoyée.

Le mot-clé STORED est nécessaire pour indiquer que la colonne sera calculée lors de l'écriture et sera stockée sur le disque.

L'expression de génération peut faire référence à d'autres colonnes de la table, mais pas à une autre colonne générée. Toute fonction ou opérateur utilisé dans l'expression de génération devra être IMMUTABLE. Toute référence à d'autres tables est interdite.

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [(*options_sequence*)]

Cette clause crée la colonne comme une *colonne d'identité*. Elle aura une séquence implicite attachée à elle et, dans les lignes nouvellement insérées, la colonne aura automatiquement des valeurs récupérées de la séquence qui lui est assignée. Une telle colonne est implicitement NOT NULL.

Les clauses ALWAYS et BY DEFAULT déterminent comment la valeur de la séquence est prioritaire par rapport à une valeur définie par l'utilisateur dans un ordre INSERT et UPDATE.

Pour un ordre INSERT, si ALWAYS est spécifié, une valeur définie par l'utilisateur ne sera acceptée que si l'ordre INSERT spécifie OVERRIDING SYSTEM VALUE. Si BY DEFAULT est spécifié, alors la valeur spécifiée par l'utilisateur est prioritaire. Voir INSERT pour plus de détails. (Avec une commande COPY, les valeurs spécifiées par l'utilisateur sont toujours utilisées quelque soit ce paramètre).

Pour un ordre UPDATE, si ALWAYS est spécifié, toute mise à jour de la colonne à une valeur autre que DEFAULT sera rejetée. Si BY DEFAULT est spécifié, la colonne peut être mise à jour normalement. (Il n'y a pas de clause OVERRIDING pour les ordres UPDATE).

La clause optionnelle *sequence_options* peut être utilisée pour surcharger les paramètres de la séquence. Les options disponibles incluent celles affichées pour CREATE SEQUENCE, ainsi que SEQUENCE NAME *nom*, LOGGED et UNLOGGED, qui permettent la sélection du nom et du niveau de persistence de la séquence. Sans SEQUENCE NAME, le système choisira un nom inutilisé pour la séquence. Sans LOGGED ou UNLOGGED, la séquence aura le même niveau de persistence que la table.

UNIQUE [NULLS [NOT] DISTINCT] (column constraint)
UNIQUE [NULLS [NOT] DISTINCT] (*nom_colonne* [, ...]) [INCLUDE (*nom_colonne* [, ...])] (contrainte de table)

La contrainte UNIQUE indique qu'un groupe d'une ou plusieurs colonnes d'une table ne peut contenir que des valeurs uniques. Le comportement de la contrainte d'unicité de table est le même que celle de la contrainte d'unicité de colonne, avec la capacité supplémentaire de traiter plusieurs colonnes. Dans ce cas, la contrainte s'assure que tout couple de ligne diffère au moins sur une de ces colonnes.

Pour une contrainte d'unicité, les valeurs NULL ne sont pas considérées comme égales, sauf si la clause `NULLS NOT DISTINCT` est précisée.

Chaque contrainte d'unicité doit nommer un ensemble de colonnes qui est différent de l'ensemble de colonnes nommées par toute autre contrainte d'unicité ou de clé primaire définie pour la table. (Sinon les contraintes d'unicité redondantes seraient ignorées.)

Lors de la mise en place d'une contrainte unique sur une hiérarchie de partitions à plusieurs niveaux, toutes les colonnes de la clé de partitionnement de la table partitionnée cible, ainsi que celles des tables partitionnées filles, doivent être incluses dans la définition de la contrainte.

Ajouter une contrainte d'unicité va automatiquement créer un index btree unique sur la colonne ou le groupe de colonnes utilisée(s) dans la contrainte.

La clause optionnelle `INCLUDE` ajoute à cet index une ou plusieurs colonnes qui sont uniquement une « charge » : l'unicité n'est pas forcée pour elle, et l'index ne peut pas être utilisé dans une recherche sur ces colonnes. Cependant, elles peuvent être récupérées par un parcours d'index seul. Notez cependant que si la contrainte n'est pas appliquée sur ces colonnes incluses, elle en dépend tout de même. En conséquence, certaines opérations sur ces colonnes (par exemple `DROP COLUMN`) peuvent causer une suppression en cascade de la contrainte et de l'index.

`PRIMARY KEY` (contrainte de colonne)
`PRIMARY KEY ( nom_colonne [, ... ] )[ INCLUDE ( nom_colonne [, ...]) ]` (contrainte de table)

La contrainte `PRIMARY KEY` indique qu'une ou plusieurs colonnes d'une table peuvent uniquement contenir des valeurs uniques (pas de valeurs dupliquées) et non NULL. Une table ne peut avoir qu'une seule clé primaire, que ce soit une contrainte au niveau de la colonne ou au niveau de la table.

La contrainte clé primaire doit nommer un ensemble de colonnes différent de l'ensemble de colonnes nommé par toute contrainte unique définie sur la même table. (Sinon, la contrainte unique est redondante et sera ignorée.)

`PRIMARY KEY` force les mêmes contraintes sur les données que la combinaison `UNIQUE` et `NOT NULL`. Néanmoins, identifier un ensemble de colonnes comme une clé primaire fournit aussi des métadonnées sur la conception du schéma car une clé primaire implique que les autres tables peuvent s'appuyer sur cet ensemble de colonnes comme un identifiant unique des lignes de la table.

Lors de leur ajout sur une table partitionnée, les contraintes `PRIMARY KEY` partagent les restrictions des contraintes `UNIQUE` précédemment décrites.

Ajouter une contrainte `PRIMARY KEY` créera automatiquement un index btree d'unicité sur la colonne ou le groupe de colonnes utilisées dans la contrainte.

Ajouter une contrainte `PRIMARY KEY` va automatiquement créer un index btree unique sur la colonne ou le groupe de colonnes utilisée(s) dans la contrainte.

La clause supplémentaire `INCLUDE` ajoute à l'index une ou plusieurs colonnes qui sont une simple « charge »: l'unicité n'est pas contrainte pour ces colonnes, et l'index ne peut pas être utilisé sur la base de ces colonnes. Néanmoins, elles peuvent être récupérées par un parcours d'index seul. Notez que, bien que la contrainte n'est pas imposée sur les colonnes incluses, elle dépend des colonnes. En conséquence, certaines opérations sur ces colonnes (par exemple `DROP COLUMN`) peuvent causer la suppression en cascade de la contrainte et de l'index.

`EXCLUDE [ USING méthode_index ] ( élément_exclusion WITH opérateur [, ... ] ) paramètres_index [ WHERE ( prédicat ) ]`

La clause `EXCLUDE` définit une contrainte d'exclusion qui garantit que si deux lignes sont comparées sur la ou les colonnes spécifiées ou des expressions utilisant le ou les opérateurs

spécifiés, seulement certaines de ces comparaisons, mais pas toutes, renverront TRUE. Si tous les opérateurs spécifiés testent une égalité, ceci est équivalent à une contrainte UNIQUE bien qu'une contrainte unique ordinaire sera plus rapide. Néanmoins, ces contraintes d'exclusion peuvent spécifier des contraintes qui sont plus générales qu'une simple égalité. Par exemple, vous pouvez spécifier qu'il n'y a pas deux lignes dans la table contenant des cercles de surcharge (voir Section 8.8) en utilisant l'opérateur &&. Le(s) opérateur(s) doivent être commutatif(s).

Des contraintes d'exclusion sont implantées en utilisant un index, donc chaque opérateur précisé doit être associé avec une classe d'opérateurs appropriée (voir Section 11.10) pour la méthode d'accès par index, nommée *méthode_index*. Chaque *élément_exclusion* définit une colonne de l'index, donc il peut spécifier en option une collation, une classe d'opérateur, des paramètres pour la classe d'opérateurs, et/ou des options de tri ; toutes ces options sont décrites entièrement dans CREATE INDEX.

La méthode d'accès doit supporter amgettuple (voir Chapitre 64) ; dès à présent, cela signifie que GIN ne peut pas être utilisé. Bien que cela soit autorisé, il existe peu de raison pour utiliser des index B-tree ou hash avec une contrainte d'exclusion parce que cela ne fait rien de mieux que ce que peut faire une contrainte unique ordinaire. Donc, en pratique, la méthode d'accès sera toujours GiST ou SP-GiST.

Le *prédicat* vous permet de spécifier une contrainte d'exclusion sur un sous-ensemble de la table ; en interne, un index partiel est créé. Notez que ces parenthèses sont requis autour du prédicat.

REFERENCES *table_reference* [(*colonne_reference*)] [MATCH *type_correspondance*] [ON DELETE *action_referentielle*] [ON UPDATE *action_referentielle*] (contrainte de colonne)
FOREIGN KEY (*nom_colonne* [, ...]) REFERENCES *table_reference* [(*colonne_reference* [, ...])] [MATCH *matchtype*] [ON DELETE *action*] [ON UPDATE *action*] (contrainte de colonne)

Ces clauses spécifient une contrainte de clé étrangère. Cela signifie qu'un groupe de colonnes de la nouvelle table ne peut contenir que des valeurs correspondant à celles des colonnes de référence de la table de référence. Si la liste *colonne_reference* est omise, la clé primaire de la *table_reference* est utilisée. Sinon, la liste *refcolumn* doit faire référence aux colonnes d'une contrainte d'unicité ou de clé primaire non déferrable d'un index d'unicité non partiel. L'utilisateur doit avoir la permission REFERENCES sur la table référencée (soit toute la table, ou la colonne référencée spécifiquement). L'ajout d'une contrainte de type clé étrangère requiert un verrou SHARE ROW EXCLUSIVE sur la table référencée. Les contraintes de type clé étrangère ne peuvent pas être définies entre des tables temporaires et des tables permanentes.

Une valeur insérée dans les colonnes de la nouvelle table est comparée aux valeurs des colonnes de référence dans la table de référence à l'aide du type de concordance fourni. Il existe trois types de correspondance : MATCH FULL (NDT : correspondance totale), MATCH PARTIAL (NDT : correspondance partielle) et MATCH SIMPLE (NDT : correspondance simple), qui est aussi la valeur par défaut. MATCH FULL n'autorise une colonne d'une clé étrangère composite à être NULL que si l'ensemble des colonnes de la clé étrangère sont NULL. Si elles sont NULL, la ligne n'a pas besoin d'avoir une correspondance dans la table référencée. MATCH SIMPLE permet à n'importe quel colonne d'une clé étrangère d'être NULL ; si l'une d'entre elles est NULL, la ligne n'a pas besoin d'avoir une correspondance dans la table référencée. MATCH PARTIAL n'est pas encore implémentée. Bien sûr, les contraintes NOT NULL peuvent être appliquées sur la (ou les) colonne(s) référençantes pour empêcher ces cas de survenir.

Lorsque les données des colonnes référencées sont modifiées, des actions sont réalisées sur les données de la table référençant. La clause ON DELETE spécifie l'action à réaliser lorsqu'une ligne référencée de la table de référence est supprimée. De la même façon, la clause ON UPDATE spécifie l'action à réaliser lorsqu'une colonne référencée est mise à jour. Si la ligne est mise à jour sans que la valeur de la colonne référencée ne soit modifiée, aucune action n'est réalisée. Les actions référentielles autres que la vérification NO ACTION ne peuvent pas être différées même si la contrainte est déclarée retardable. Les actions suivantes sont possibles pour chaque clause :

NO ACTION

Une erreur est produite pour indiquer que la suppression ou la mise à jour entraîne une violation de la contrainte de clé étrangère. Si la contrainte est différée, cette erreur est produite au moment de la vérification, si toutefois il existe encore des lignes de référence. C'est le comportement par défaut.

RESTRICT

Une erreur est produite pour indiquer que la suppression ou la mise à jour entraîne une violation de la contrainte de clé étrangère. Ce comportement est identique à NO ACTION, si ce n'est que la vérification n'est pas décalable dans le temps.

CASCADE

La mise à jour ou la suppression de la ligne de référence est propagée à l'ensemble des lignes qui la référencent, qui sont, respectivement, mises à jour ou supprimées.

SET NULL [(*nom_colonne* [, ...])]

Initialise toutes les colonnes référençantes ou un sous-ensemble indiqué des colonnes référençantes à NULL. Un sous-ensemble de colonnes peut seulement être référencé pour des actions ON DELETE.

SET DEFAULT [(*nom_colonne* [, ...])]

Initialise toutes les colonnes référençantes ou un sous-ensemble indiqué des colonnes référençantes à leur valeur par défaut. Un sous-ensemble de colonnes peut seulement être indiqué pour les actions ON DELETE. (Il doit exister une ligne dans la table référencée correspondant aux valeurs par défaut, si elles ne sont pas NULL. Dans le cas contraire, l'opération échouera.)

Si les colonnes référencées sont modifiées fréquemment, il est conseillé d'ajouter un index sur les colonnes référençantes pour que les actions associées à la contrainte de clé étrangère soient plus performantes.

DEFERRABLE
NOT DEFERRABLE

Ces clauses contrôlent la possibilité de différer la contrainte. Une contrainte qui n'est pas décalable dans le temps est vérifiée immédiatement après chaque commande. La vérification des contraintes décalables est repoussée à la fin de la transaction (à l'aide de la commande SET CONSTRAINTS). NOT DEFERRABLE est la valeur par défaut. Actuellement, seules les contraintes UNIQUE, PRIMARY KEY, EXCLUDE et REFERENCES (clé étrangère) acceptent cette clause. Les contraintes NOT NULL et CHECK ne sont pas différables. Notez que les contraintes différables ne peuvent pas être utilisées comme arbitres d'un conflit dans une commande INSERT qui inclut une clause ON CONFLICT DO UPDATE.

INITIALLY IMMEDIATE
INITIALLY DEFERRED

Si une contrainte est décalable dans le temps, cette clause précise le moment de la vérification. Si la contrainte est INITIALLY IMMEDIATE, elle est vérifiée après chaque instruction. Si la contrainte est INITIALLY DEFERRED, elle n'est vérifiée qu'à la fin de la transaction. Le moment de vérification de la contrainte peut être modifié avec la commande SET CONSTRAINTS.

USING *method*

Cette clause facultative permet de spécifier la méthode d'accès utilisée pour stocker le contenu de la nouvelle table. La méthode doit être une méthode d'accès de type TABLE. Voir Chapitre 63 pour plus d'informations. Si cette option n'est pas indiquée, la méthode d'accès par défaut est choisie pour la nouvelle table. Voir default_table_access_method pour plus d'informations.

`WITH ( paramètre_stockage [= valeur] [, ... ] )`

Cette clause spécifie les paramètres de stockage optionnels pour une table ou un index ; voir Paramètres de stockage pour plus d'informations. Pour la rétro-compatibilité de la clause `WITH` d'une table, il est possible de spécifier `OIDS=FALSE` pour indiquer que les lignes de la nouvelle table ne doivent pas contenir d'OID (identifiants d'objets). `OIDS=TRUE` n'est plus supporté.

`WITHOUT OIDS`

Il s'agit de la syntaxe rétro-compatible pour déclarer une table sans utiliser d'OID (identifiants d'objets), il n'est plus possible de créer une table avec l'option `WITH OIDS`.

`ON COMMIT`

Le comportement des tables temporaires à la fin d'un bloc de transactions est contrôlé à l'aide de la clause `ON COMMIT`. Les trois options sont :

`PRESERVE ROWS`

Aucune action n'est entreprise à la fin des transactions. Comportement par défaut.

`DELETE ROWS`

Toutes les lignes de la table temporaire sont détruites à la fin de chaque bloc de transactions. En fait, un `TRUNCATE` automatique est réalisé à chaque validation. Lorsque cette clause est utilisée sur une table partitionnée, elle n'est pas exécutée en cascade sur ses partitions.

`DROP`

La table temporaire est supprimée à la fin du bloc de transaction. Lorsque cette clause est utilisée sur une table partitionnée, cette action supprime les partitions et. Quand elle est utilisée sur une table ayant des tables filles, ces dernières sont aussi supprimées.

`TABLESPACE nom_tablespace`

`nom_tablespace` est le nom du tablespace dans lequel est créée la nouvelle table. S'il n'est pas spécifié, default_tablespace est consulté, sauf si la table est temporaire auquel cas temp_tablespaces est utilisé. Pour les tables partitionnées, comme la clause de stockage n'a pas de sens pour la table en elle-même, le tablespace indiqué écrase le `default_tablespace` comme tablespace par défaut à utiliser pour toute nouvelle partition à moins qu'un autre tablespace soit explicitement indiqué.

`USING INDEX TABLESPACE nom_tablespace`

Les index associés à une contrainte `UNIQUE`, `PRIMARY KEY`, ou `EXCLUDE` sont créés dans le tablespace nommé *`nom_tablespace`*. S'il n'est pas précisé, default_tablespace est consulté, sauf si la table est temporaire auquel cas temp_tablespaces est utilisé.

Paramètres de stockage

La clause `WITH` spécifie des *paramètres de stockage* pour les tables ainsi que pour les index associés avec une contrainte `UNIQUE`, `PRIMARY KEY`, ou `EXCLUDE`. Les paramètres de stockage des index sont documentés dans CREATE INDEX. Les paramètres de stockage actuellement disponibles pour les tables sont listés ci-dessous. Pour beaucoup de ces paramètres, comme indiqué, il y a un paramètre additionnel, de même nom mais préfixé par `toast.`, qui contrôle le le comportement de la table TOAST (stockage supplémentaire), si elle existe (voir Section 73.2 pour plus d'informations sur TOAST). Si une valeur de paramètre d'une table est configuré et que le paramètre équivalent `toast.` ne l'est pas, la partie TOAST utilisera la valeur du paramètre de la table. Ces paramètres ne sont pas supportés sur les tables partitionnées mais vous pouvez les indiquer sur des partitions enfants individuelles.

`fillfactor` (`integer`)

Le facteur de remplissage d'une table est un pourcentage entre 10 et 100. 100 (paquet complet) est la valeur par défaut. Quand un facteur de remplissage plus petit est indiqué, les opérations `INSERT` remplissent les pages de table d'au maximum ce pourcentage ; l'espace restant sur chaque page est réservé à la mise à jour des lignes sur cette page. Cela donne à `UPDATE` une chance de placer la copie d'une ligne mise à jour sur la même page que l'original, ce qui est plus efficace que de la placer sur une page différente, et rend les mises à jour heap-only tuple plus probables. Pour une table dont les entrées ne sont jamais mises à jour, la valeur par défaut est le meilleur choix, mais pour des tables mises à jour fréquemment, des facteurs de remplissage plus petits sont mieux appropriés. Ce paramètre n'est pas disponible pour la table TOAST.

`toast_tuple_target` (`integer`)

`toast_tuple_target` spécifie la taille de tuple minimale requise avant de tenter de compresser et/ou déplacer les champs de grande taille vers des tables TOAST, et est aussi la taille cible à laquelle l'on tente de réduire la taille une fois cette opération démarrée. Cela affecte les colonnes marquées External (pour le déplacement), Main (pour la compression) ou Extended (pour les deux) et ne s'applique qu'aux nouveaux enregistrements. Cela n'a pas d'effet sur les lignes existantes. Par défaut ce paramètre est configuré pour permettre au moins 4 lignes par bloc, ce qui donnera 2040 octets avec la taille de bloc par défaut. Les valeurs valides sont entre 128 octets et (taille des blocs - entête), par défaut 8160 octets. Changer cette valeur n'est pas très utile pour les lignes très courtes ou très longues. Notez que la valeur par défaut est souvent proche de la valeur optimale, et qu'il est possible que modifier ce paramètre ait des effets négatifs dans certains cas. Ce paramètre ne peut être positionné pour les tables TOAST.

`parallel_workers` (`integer`)

Ce paramètre configure le nombre de processus pouvant être utilisés pour aider lors d'un parcours parallélisé de cette table. Si ce paramètre n'est pas configuré, le système déterminera une valeur en se basant sur la taille de la relation. Le nombre réel de processus choisis par le planificateur ou par des instructions utilitaires qui utilisent des parcours séquentiels pourrait être moindre, par exemple suite à la configuration de max_worker_processes.

`autovacuum_enabled`, `toast.autovacuum_enabled` (`boolean`)

Active ou désactive le démon autovacuum pour une table particulière. Si elle vaut true, le démon autovacuum réalise des `VACUUM` et/ou `ANALYZE` automatiques sur cette table en suivant les règles discutées dans Section 25.1.6. À false, cette table ne sera pas traitée par le démon autovacuum, sauf s'il y a un risque de réutilisation des identifiants de transaction. Voir Section 25.1.5 pour plus d'informations sur la prévention de ce problème. Notez que le démon autovacuum n'est pas lancé (sauf pour prévenir la réutilisation des identifiants de transaction) si le paramètre autovacuum vaut false ; configurer les paramètres de stockage d'une table ne surcharge pas cela. De ce fait, il y a peu d'intérêt de configurer ce paramètre à `true`.

`vacuum_index_cleanup`, `toast.vacuum_index_cleanup` (`enum`)

Force ou désactive le nettoyage des index quand un `VACUUM` est lancé sur cette table. La valeur par défaut est `AUTO`. À `OFF`, le nettoyage de l'index est désactivé. À `ON`, il est activé. À `AUTO`, une décision est prise dynamiquement, chaque fois que `VACUUM` s'exécute. Le comportement dynamique permet à `VACUUM` d'éviter un parcours inutile des index pour supprimer très peu de lignes mortes. Forcer la désactivation du nettoyage de tous les index peut accélérer très significativement une opération `VACUUM`, mais pourrait résulter en des index sévèrement fragmentés si les modifications de la table sont fréquentes. Le paramètre `INDEX_CLEANUP` de la commande `VACUUM`, si indiqué, surcharge la valeur de cette option.

`vacuum_truncate`, `toast.vacuum_truncate` (`boolean`)

Active ou désactive la suppression par le vacuum des pages vides en fin de table. La valeur par défaut est `true`. Si `true`, la commande `VACUUM` et l'autovacuum suppriment ces pages et

l'espace libéré est redonné au système d'exploitation. La suppression de ces pages vides nécessite un verrou de type ACCESS EXCLUSIVE sur la table. Le paramètre TRUNCATE de la commande VACUUM, si indiqué, écrase cette valeur.

autovacuum_vacuum_threshold, toast.autovacuum_vacuum_threshold (integer)

Valeur spécifique à la table pour le paramètre autovacuum_vacuum_threshold. Ce paramètre peut être configuré pour les tables partitionnées.

autovacuum_vacuum_scale_factor, toast.autovacuum_vacuum_scale_factor (floating point)

Valeur spécifique à la table pour le paramètre autovacuum_vacuum_scale_factor. Ce paramètre peut être configuré pour les tables partitionnées.

autovacuum_vacuum_insert_threshold, toast.autovacuum_vacuum_insert_threshold (integer)

Valeur spécifique à la table pour le paramètre autovacuum_vacuum_insert_threshold. La valeur spéciale de -1 peut être utilisée pour désactiver le vacuum lors des insertions sur la table.

autovacuum_vacuum_insert_scale_factor, toast.autovacuum_vacuum_insert_scale_factor (floating point)

Valeur spécifique à la table pour le paramètre autovacuum_vacuum_insert_scale_factor.

autovacuum_analyze_threshold (integer)

Valeur spécifique à la table pour le paramètre autovacuum_analyze_threshold.

autovacuum_analyze_scale_factor (floating point)

Valeur spécifique à la table pour le paramètre autovacuum_analyze_scale_factor.

autovacuum_vacuum_cost_delay, toast.autovacuum_vacuum_cost_delay (floating point)

Valeur spécifique à la table pour le paramètre autovacuum_vacuum_cost_delay.

autovacuum_vacuum_cost_limit, toast.autovacuum_vacuum_cost_limit (integer)

Valeur spécifique à la table pour le paramètre autovacuum_vacuum_cost_limit.

autovacuum_freeze_min_age, toast.autovacuum_freeze_min_age (integer)

Valeur spécifique à la table pour le paramètre vacuum_freeze_min_age. Notez que l'autovacuum ignorera les paramètres autovacuum_freeze_min_age spécifiques à la table qui sont plus importants que la moitié du paramètre autovacuum_freeze_max_age.

autovacuum_freeze_max_age, toast.autovacuum_freeze_max_age (integer)

Valeur spécifique à la table pour le paramètre autovacuum_freeze_max_age. Notez que l'autovacuum ignorera les paramètres autovacuum_freeze_max_age spécifiques à la table qui sont plus importants que la configuration globale (elle ne peut être que plus petite).

autovacuum_freeze_table_age, toast.autovacuum_freeze_table_age (integer)

Valeur spécifique à la table pour le paramètre vacuum_freeze_table_age.

`autovacuum_multixact_freeze_min_age`, `toast.autovacuum_multixact_freeze_min_age` (`integer`)

Valeur spécifique à la table pour le paramètre vacuum_multixact_freeze_min_age. Notez que l'autovacuum ignorera les paramètres `autovacuum_multixact_freeze_min_age` spécifiques à la table si leur configuration est supérieure à la moitié de la valeur du paramètre global autovacuum_multixact_freeze_max_age.

`autovacuum_multixact_freeze_max_age`, `toast.autovacuum_multixact_freeze_max_age` (`integer`)

Valeur spécifique à la table pour le paramètre autovacuum_multixact_freeze_max_age. Notez que l'autovacuum ignorera les paramètres `autovacuum_multixact_freeze_max_age` spécifiques à la table si leur configuration est supérieure à la valeur du paramètre global (elle peut seulement être inférieure).

`autovacuum_multixact_freeze_table_age`, `toast.autovacuum_multixact_freeze_table_age` (`integer`)

Valeur spécifique à la table pour le paramètre vacuum_multixact_freeze_table_age.

`log_autovacuum_min_duration`, `toast.log_autovacuum_min_duration` (`integer`)

Valeur spécifique à la table pour le paramètre log_autovacuum_min_duration.

`user_catalog_table` (`boolean`)

Déclare la table comme une autre table du catalogue dans le cadre de la réplication logique. Voir Section 49.6.2 pour les détails. Ce paramètre ne peut pas être configuré pour les tables TOAST.

Notes

PostgreSQL crée automatiquement un index pour chaque contrainte d'unicité ou clé primaire afin d'assurer l'unicité. Il n'est donc pas nécessaire de créer un index spécifiquement pour les colonnes de clés primaires. Voir CREATE INDEX pour plus d'informations.

Les contraintes d'unicité et les clés primaires ne sont pas héritées dans l'implantation actuelle. Cela diminue la fonctionnalité des combinaisons d'héritage et de contraintes d'unicité.

Une table ne peut pas avoir plus de 1600 colonnes (en pratique, la limite réelle est habituellement plus basse du fait de contraintes sur la longueur des lignes).

Exemples

Créer une table `films` et une table `distributeurs` :

```
CREATE TABLE films (
    code        char(5) CONSTRAINT premierecle PRIMARY KEY,
    titre       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    genre       varchar(10),
    duree       interval hour to minute
);

CREATE TABLE distributeurs (
     did    integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
     nom    varchar(40) NOT NULL CHECK (nom <> '')
);
```

Créer une table contenant un tableau à deux dimensions :

```
CREATE TABLE array_int (
    vecteur  int[][]
);
```

Définir une contrainte d'unicité pour la table `films`. Les contraintes d'unicité de table peuvent être définies sur une ou plusieurs colonnes de la table :

```
CREATE TABLE films (
    code        char(5),
    titre       varchar(40),
    did         integer,
    date_prod   date,
    genre       varchar(10),
    duree       interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Définir une contrainte de vérification sur une colonne :

```
CREATE TABLE distributeurs (
    did     integer CHECK (did > 100),
    nom    varchar(40)
);
```

Définir une contrainte de vérification sur la table :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40),
    CONSTRAINT con1 CHECK (did > 100 AND nom <> '')
);
```

Définir une contrainte de clé primaire sur la table `films`.

```
CREATE TABLE films (
    code        char(5),
    titre       varchar(40),
    did         integer,
    date_prod   date,
    genre       varchar(10),
    duree       interval hour to minute,
    CONSTRAINT code_titre PRIMARY KEY(code,titre)
);
```

Définir une contrainte de clé primaire pour la table `distributeurs`. Les deux exemples suivants sont équivalents, le premier utilise la syntaxe de contrainte de table, le second la syntaxe de contrainte de colonne :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40),
```

```
    PRIMARY KEY(did)
);

CREATE TABLE distributeurs (
    did     integer PRIMARY KEY,
    nom     varchar(40)
);
```

Affecter une valeur par défaut à la colonne `nom`, une valeur par défaut à la colonne `did`, engendrée à l'aide d'une séquence, et une valeur par défaut à la colonne `modtime`, équivalente au moment où la ligne est insérée :

```
CREATE TABLE distributeurs (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributeurs_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

Définir deux contraintes de colonnes `NOT NULL` sur la table `distributeurs`, dont l'une est explicitement nommée :

```
CREATE TABLE distributeurs (
    did     integer CONSTRAINT no_null NOT NULL,
    nom     varchar(40) NOT NULL
);
```

Définir une contrainte d'unicité sur la colonne `nom` :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40) UNIQUE
);
```

La même chose en utilisant une contrainte de table :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40),
    UNIQUE(nom)
);
```

Créer la même table en spécifiant un facteur de remplissage de 70% pour la table et les index uniques :

```
CREATE TABLE distributeurs (
    did     integer,
    nom     varchar(40),
    UNIQUE(nom) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

Créer une table `cercles` avec une contrainte d'exclusion qui empêche le croisement de deux cercles :

```
CREATE TABLE cercles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

Créer une table `cinemas` dans le tablespace `diskvol1` :

```
CREATE TABLE cinemas (
    id serial,
    nom text,
    emplacement text
) TABLESPACE diskvol1;
```

Créer un type composite et une table typée :

```
CREATE TYPE type_employe AS (nom text, salaire numeric);

CREATE TABLE employes OF type_employe (
    PRIMARY KEY (nom),
    salaire WITH OPTIONS DEFAULT 1000
);
```

Créer une table partitionée par intervalles :

```
CREATE TABLE measurement (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);
```

Créer une table partitionnée par intervalles avec plusieurs colonnes dans la clé de partitionnement :

```
CREATE TABLE measurement_year_month (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (EXTRACT(YEAR FROM logdate), EXTRACT(MONTH
 FROM logdate));
```

Créer une table partitionnée par liste de valeurs :

```
CREATE TABLE cities (
    city_id      bigserial not null,
    name         text not null,
    population   bigint
) PARTITION BY LIST (left(lower(name), 1));
```

Créer une table partitionnée par hachage :

```
CREATE TABLE orders (
    order_id     bigint not null,
    cust_id      bigint not null,
    status       text
) PARTITION BY HASH (order_id);
```

Créer une partition d'une table partitionnée par intervalles :

```
CREATE TABLE measurement_y2016m07
    PARTITION OF measurement (
    unitsales DEFAULT 0
) FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
```

Créer quelques partitions d'une table partitionnée par intervalles avec plusieurs colonnes dans la clé de partitionnement :

```
CREATE TABLE measurement_ym_older
    PARTITION OF measurement_year_month
    FOR VALUES FROM (MINVALUE, MINVALUE) TO (2016, 11);

CREATE TABLE measurement_ym_y2016m11
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 11) TO (2016, 12);

CREATE TABLE measurement_ym_y2016m12
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2016, 12) TO (2017, 01);

CREATE TABLE measurement_ym_y2017m01
    PARTITION OF measurement_year_month
    FOR VALUES FROM (2017, 01) TO (2017, 02);
```

Créer une partition d'une table partitionnée par liste de valeur :

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b');
```

Créer une partition d'une table partitionnée par liste e valeur qui est elle-même partitionnée, puis y ajouter une partition :

```
CREATE TABLE cities_ab
    PARTITION OF cities (
    CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b') PARTITION BY RANGE (population);

CREATE TABLE cities_ab_10000_to_100000
    PARTITION OF cities_ab FOR VALUES FROM (10000) TO (100000);
```

Créer des partitions d'une table partitionnée par hachage :

```
CREATE TABLE orders_p1 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE orders_p2 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE orders_p3 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE orders_p4 PARTITION OF orders
    FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

Créer une partition par défaut :

```
CREATE TABLE cities_partdef
    PARTITION OF cities DEFAULT;
```

Compatibilité

La commande `CREATE TABLE` est conforme au standard SQL, aux exceptions indiquées ci-dessous.

Tables temporaires

Bien que la syntaxe de `CREATE TEMPORARY TABLE` ressemble à celle du SQL standard, l'effet n'est pas le même. Dans le standard, les tables temporaires sont définies une seule fois et existent automatiquement (vide de tout contenu au démarrage) dans toute session les utilisant. PostgreSQL, au contraire, impose à chaque session de lancer une commande `CREATE TEMPORARY TABLE` pour chaque table temporaire utilisée. Cela permet à des sessions différentes d'utiliser le même nom de table temporaire dans des buts différents (le standard contraint toutes les instances d'une table temporaire donnée à pointer sur la même structure de table).

Le comportement des tables temporaires tel que défini par le standard est largement ignorée. Le comportement de PostgreSQL sur ce point est similaire à celui de nombreuses autres bases de données SQL.

Le standard SQL distingue aussi les tables temporaires globales et locales. Une table temporaire local a un contenu séparé pour chaque module SQL à l'intérieur de chaque session bien que sa définition est toujours partagée entre les sessions. Comme PostgreSQL ne supporte pas les modules SQL, la distinction n'a pas de raison d'être avec PostgreSQL.

Pour des raisons de compatibilité, PostgreSQL accepte néanmoins les mots-clés `GLOBAL` et `LOCAL` dans la définition d'une table temporaire, mais ils n'ont actuellement aucun effet. L'utilisation de ces mots clés n'est pas conseillée car les versions futures de PostgreSQL pourrait adopter une interprétation plus standard de leur signification.

La clause `ON COMMIT` sur les tables temporaires diffère quelque peu du standard SQL. Si la clause `ON COMMIT` est omise, SQL spécifie `ON COMMIT DELETE ROWS` comme comportement par défaut. PostgreSQL utilise `ON COMMIT PRESERVE ROWS` par défaut. De plus, l'option `ON COMMIT DROP` n'existe pas en SQL.

Contraintes d'unicité non déferrées

Quand une contrainte `UNIQUE` ou `PRIMARY KEY` est non déferrable, PostgreSQL vérifie l'unicité immédiatement après qu'une ligne soit insérée ou modifiée. Le standard SQL indique que l'unicité doit être forcée seulement à la fin de l'instruction ; ceci fait une différence quand, par exemple, une seule commande met à jour plusieurs valeurs de clés. Pour obtenir un comportement compatible au standard, déclarez la contrainte comme `DEFERRABLE` mais non déferrée (c'est-à-dire que `INITIALLY`

IMMEDIATE). Faites attention que cela peut être beaucoup plus lent qu'une vérification d'unicité immédiate.

Contraintes de vérification de colonnes

Dans le standard, les contraintes de vérification CHECK de colonne ne peuvent faire référence qu'à la colonne à laquelle elles s'appliquent ; seules les contraintes CHECK de table peuvent faire référence à plusieurs colonnes. PostgreSQL n'impose pas cette restriction ; les contraintes de vérifications de colonnes et de table ont un traitement identique.

EXCLUDE Constraint

Le type de contrainte EXCLUDE est une extension PostgreSQL.

Contraintes de clé étrangère

La possibilité d'indiquer des listes de colonne dans les actions SET DEFAULT et SET NULL des clés étrangères est une extension PostgreSQL.

PostgreSQL étend la contrainte de clé étrangère en permettant de référencer les colonnes d'un index d'unicité au lieu des colonnes d'une clé primaire ou d'une contrainte d'unicité.

Contrainte NULL

La « contrainte » NULL (en fait, une non-contrainte) est une extension PostgreSQL au standard SQL, incluse pour des raisons de compatibilité avec d'autres systèmes de bases de données (et par symétrie avec la contrainte NOT NULL). Comme c'est la valeur par défaut de toute colonne, sa présence est un simple bruit.

Nommage de contrainte

Le standard SQL stipule que les contraintes de table et de domaine doivent avoir des noms uniques sur le schéma contenant la table ou le domaine. PostgreSQL est laxiste : il requiert seulement que le nom des contraintes soit unique parmi les contraintes attachées à une table ou un domaine particulier. Néanmoins, cette liberté supplémentaire n'existe pas pour les contraintes basées sur des index (contraintes UNIQUE, PRIMARY KEY et EXCLUDE) parce que l'index associé est nommé de la même façon que la contrainte, et les noms d'index doivent être uniques parmi toutes les relations du même schéma.

Actuellement, PostgreSQL n'enregistre pas de noms pour les contraintes NOT NULL, donc elles ne sont pas sujettes aux restrictions d'unicité. Ceci pourrait changer dans une prochaine version.

Héritage

L'héritage multiple via la clause INHERITS est une extension du langage PostgreSQL. SQL:1999 et les versions ultérieures définissent un héritage simple en utilisant une syntaxe et des sémantiques différentes. L'héritage style SQL:1999 n'est pas encore supporté par PostgreSQL.

Tables sans colonne

PostgreSQL autorise la création de tables sans colonne (par exemple, CREATE TABLE foo();). C'est une extension du standard SQL, qui ne le permet pas. Les tables sans colonne ne sont pas très utiles mais les interdire conduit à un comportement étrange de ALTER TABLE DROP COLUMN. Il est donc plus sage d'ignorer simplement cette restriction.

Colonnes d'identités multiples

PostgreSQL autorise une table à avoir plus d'une colonne d'identité. le standard spécifie qu'une table peut avoir au plus une colonne d'identité. Cette règle est assouplie principalement pour donner

plus de flexibilité pour effectuer des changements de schéma ou des migrations. Veuillez noter que la commande `INSERT` supporte uniquement une seule clause de surcharge qui s'appliquent à la commande entière, et donc avoir de multiples colonnes d'identités avec des comportements différents n'est pas bien supporté.

Generated Columns

L'option `STORED` n'est pas standard mais est également utilisée dans d'autres implémentations SQL. le standard SQL ne spécifie pas de clause de stockage pour les colonnes générées.

Clause `LIKE`

Alors qu'une clause `LIKE` existe dans le standard SQL, beaucoup des options acceptées par PostgreSQL ne sont pas dans le standard, et certaines options du standard ne sont pas implémentées dans PostgreSQL.

Clause `WITH`

La clause `WITH` est une extension PostgreSQL ; les paramètres de stockage ne sont pas dans le standard.

Tablespaces

Le concept PostgreSQL de tablespace n'est pas celui du standard. De ce fait, les clauses `TABLESPACE` et `USING INDEX TABLESPACE` sont des extensions.

Tables typées

Les tables typées implémentent un sous-ensemble du standard SQL. Suivant le standard, une table typée a des colonnes correspondant au type composite ainsi qu'une autre colonne qui est la « colonne auto-référente ». PostgreSQL ne supporte pas ces colonnes auto-référentes explicitement.

Clause `PARTITION BY`

La clause`PARTITION BY` est une extension PostgreSQL de la norme SQL.

Clause `PARTITION OF`

La clause `PARTITION OF` est une extension PostgreSQL de la norme SQL.

Voir aussi

ALTER TABLE, DROP TABLE, CREATE TABLE AS, CREATE TABLESPACE

CREATE TABLE AS

CREATE TABLE AS — Définir une nouvelle table à partir des résultats d'une requête

Synopsis

```
+CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ]
 TABLE [ IF NOT EXISTS ] nom_table
    [ (nom_colonne [, ...] ) ]
    [ USING method ]
    [ WITH ( parametre_stockage [= valeur] [, ... ] ) | WITHOUT
 OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE nom_espace_logique ]
    AS requête
    [ WITH [ NO ] DATA ]
```

Description

`CREATE TABLE AS` crée une table et y insère les données récupérées par une commande `SELECT`. Les colonnes de la table ont les noms et les types de données associés aux colonnes en sortie du `SELECT` (les noms des colonnes peuvent toutefois être surchargés).

`CREATE TABLE AS` semble posséder des similitudes avec la création d'une vue mais est, en fait, assez différente : elle crée une nouvelle table et n'évalue la requête qu'une seule fois, pour le chargement initial de la nouvelle table. Les modifications ultérieures de la table source ne sont pas prises en compte. Au contraire, une vue réévalue l'instruction `SELECT` de définition à chaque appel.

`CREATE TABLE AS` nécessite le droit `CREATE` sur le schéma utilisé par la table.

Paramètres

`GLOBAL` ou `LOCAL`

Ignoré. Ces mots clés sont obsolètes, ils ne sont conservés que pour la compatibilité (cf. CREATE TABLE).

`TEMPORARY` ou `TEMP`

Si spécifié, la table est temporaire (cf. CREATE TABLE).

`UNLOGGED`

Si spécifié, la table est créée comme une table non tracée dans les journaux de transactions. Voir CREATE TABLE pour plus de détails.

`IF NOT EXISTS`

Ne renvoie pas une erreur si une relation de même nom existe déjà. Un message d'avertissement est renvoyé dans ce cas et la table n'est pas modifiée.

`nom_table`

Le nom de la table à créer (éventuellement qualifié du nom du schéma).

nom_colonne

Le nom d'une colonne dans la nouvelle table. Si les noms de colonnes ne sont pas précisés, ils sont issus des noms des colonnes en sortie de la requête.

USING *méthode*

Cette clause optionnelle indique la méthode d'accès à la table à utiliser pour stocker le contenu de la nouvelle table ; la méthode doit être une méthode d'accès de type TABLE. Voir Chapitre 63 pour plus d'informations. Si cette option n'est pas spécifiée, la méthode d'accès par défaut à la table est choisie pour la nouvelle table. Voir default_table_access_method pour plus d'informations.

WITH (*paramètre_stockage* [= *valeur*] [, ...])

Cette clause indique les paramètres de stockage optionnels pour la nouvelle table ; voir Paramètres de stockage pour plus d'informations. Par souci de rétrocompatibilité, la clause WITH pour une table peut aussi inclure OIDS=FALSE pour spécifier que les lignes de la table ne doivent pas comporter d'OID (object identifiers), OIDS=TRUE n'étant plus supporté.

WITHOUT OIDS

Cette clause est une syntaxe rétrocompatible pour déclarer une table WITHOUT OIDS. La création d'une table WITH OIDS n'est plus supportée.

ON COMMIT

Le comportement des tables temporaires à la fin d'un bloc de transaction est contrôlable en utilisant ON COMMIT. Voici les trois options :

PRESERVE ROWS

Aucune action spéciale n'est effectuée à la fin de la transaction. C'est le comportement par défaut.

DELETE ROWS

Toutes les lignes de la table temporaire seront supprimées à la fin de chaque bloc de transaction. Habituellement, un TRUNCATE automatique est effectué à chaque COMMIT.

DROP

La table temporaire sera supprimée à la fin du bloc de transaction en cours.

TABLESPACE *nom_espace_logique*

L'*nom_espace_logique* est le nom du tablespace dans lequel est créée la nouvelle table. S'il n'est pas indiqué, default_tablespace est consulté, sauf si la table est temporaire auquel cas temp_tablespaces est utilisé.

requête

Une commande SELECT, TABLE, ou VALUES, voire un EXECUTE qui exécute un SELECT préparé, TABLE ou une requête VALUES.

WITH [NO] DATA

Cette clause indique si les données produites par la requêtes doivent être copiées dans la nouvelle table. Si non, seule la structure de la table est copiée. La valeur par défaut est de copier les données.

Notes

Cette commande est fonctionnellement équivalente à SELECT INTO. Elle lui est cependant préférée car elle présente moins de risques de confusion avec les autres utilisations de la syntaxe SELECT INTO. De plus, CREATE TABLE AS offre plus de fonctionnalités que SELECT INTO.

Exemples

Créer une table `films_recent` contenant les entrées récentes de la table `films` :

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2006-01-01';
```

Pour copier une table complètement, la forme courte utilisant la clause `TABLE` peut aussi être utilisée :

```
CREATE TABLE films2 AS
  TABLE films;
```

Créer une nouvelle table temporaire `films_recents` consistant des seules entrées récentes provenant de la table `films` en utilisant une instruction préparée. La nouvelle table sera supprimée à la validation (COMMIT) :

```
PREPARE films_recents(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recents ON COMMIT DROP AS
  EXECUTE films_recents('2002-01-01');
```

Compatibilité

`CREATE TABLE AS` est conforme au standard SQL. Voici les extensions non standard :

- Le standard requiert des parenthèses autour de la clause de la sous-requête ; elles sont optionnelles dans PostgreSQL.
- Dans le standard, la clause `WITH [ NO ] DATA` est requise alors que PostgreSQL la rend optionnelle.
- PostgreSQL gère les tables temporaires d'une façon bien différente de celle du standard ; voir CREATE TABLE pour les détails.
- La clause `WITH` est une extension PostgreSQL ; les paramètres de stockage ne sont dans le standard.
- Le concept PostgreSQL des tablespaces ne fait pas partie du standard. Du coup, la clause `TABLESPACE` est une extension.

Voir aussi

CREATE MATERIALIZED VIEW, CREATE TABLE, EXECUTE, SELECT, SELECT INTO, VALUES

CREATE TABLESPACE

CREATE TABLESPACE — Définir un nouvel tablespace

Synopsis

```
+CREATE TABLESPACE nom_tablespace
    [ OWNER { nouveau_propriétaire | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER } ]
    LOCATION 'répertoire'
    [ WITH ( option_tablespace = valeur [, ... ] ) ]
```

Description

`CREATE TABLESPACE` enregistre un nouveau tablespace pour la grappe de bases de données. Le nom du tablespace doit être distinct du nom de tout autre tablespace de la grappe.

Un tablespace permet aux superutilisateurs de définir un nouvel emplacement sur le système de fichiers pour le stockage des fichiers de données contenant des objets de la base (comme les tables et les index).

Un utilisateur disposant des droits appropriés peut passer *`nom_tablespace`* comme paramètre de `CREATE DATABASE`, `CREATE TABLE`, `CREATE INDEX` ou `ADD CONSTRAINT` pour que les fichiers de données de ces objets soient stockés à l'intérieur du tablespace spécifié.

> **Avertissement**
>
> Un tablespace ne peut pas être utilisé indépendamment de l'instance dans laquelle il a été défini ; voir Section 23.6.

Paramètres

`nom_tablespace`

Le nom du tablespace à créer. Le nom ne peut pas commencer par `pg_`, de tels noms sont réservés pour les tablespaces système.

`nom_utilisateur`

Le nom de l'utilisateur, propriétaire du tablespace. En cas d'omission, il s'agit de l'utilisateur ayant exécuté la commande. Seuls les superutilisateurs peuvent créer des tablespaces mais ils peuvent en donner la propriété à des utilisateurs standard.

`répertoire`

Le répertoire qui sera utilisé pour le tablespace. Le répertoire doit exister (`CREATE TABLESPACE` ne le créera pas), doit être vide et doit appartenir à l'utilisateur système PostgreSQL. Le répertoire doit être spécifié par un chemin absolu.

`option_tablespace`

Un paramètre à configurer ou réinitialiser pour un tablespace. Actuellement, les seuls paramètres disponibles sont `seq_page_cost` et `random_page_cost`, `effective_io_concurrency` et `maintenance_io_concurrency`. Configurer ces

valeurs pour un tablespace particulier surchargera l'estimation habituelle de l'optimiseur pour le coût de lecture des pages provenant de tables stockées dans ce tablespace, ainsi que le comportement de lecture en avance de l'exécuteur, telle qu'établie par les paramètres de configuration du même nom (voir seq_page_cost, random_page_cost, effective_io_concurrency, maintenance_io_concurrency). Ceci peut se révéler utile si un des tablespaces est situé sur un disque plus rapide ou plus lent que le reste du système d'entrées/sorties.

Notes

Les tablespaces ne sont supportés que sur les systèmes gérant les liens symboliques.

`CREATE TABLESPACE` ne peut pas être exécuté à l'intérieur d'un bloc de transactions.

Exemples

Pour créer un tablespace `dbspace` à l'emplacement `/data/dbs` du système de fichiers, créer d'abord le répertoire en utilisant les commandes du système d'exploitation et lui donner le propriétaire adéquat :

```
mkdir /data/dbs
chown postgres:postgres /data/dbs
```

Puis passer la commande de création de tablespace dans PostgreSQL:

```
CREATE TABLESPACE espace_base LOCATION '/data/dbs';
```

Pour créer un tablespace appartenant à un autre utilisateur de base de données, utiliser une commande comme ceci :

```
CREATE TABLESPACE espace_index OWNER genevieve LOCATION '/data/
indexes';
```

Compatibilité

`CREATE TABLESPACE` est une extension PostgreSQL.

Voir aussi

CREATE DATABASE, CREATE TABLE, CREATE INDEX, DROP TABLESPACE, ALTER TABLESPACE

CREATE TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH CONFIGURATION — définir une nouvelle configuration de recherche plein texte

Synopsis

```
CREATE TEXT SEARCH CONFIGURATION nom (
    PARSER = nom_analyseur |
    COPY = config_source
)
```

Description

`CREATE TEXT SEARCH CONFIGURATION` crée une nouvelle configuration de recherche plein texte. Une configuration indique l'analyseur qui peut diviser une chaîne en jetons, ainsi que les dictionnaires pouvant être utilisés pour déterminer les jetons intéressants à rechercher.

Si seul l'analyseur est indiqué, la nouvelle configuration de recherche plein texte n'a initialement aucune relation entre les types de jeton et les dictionnaires et, du coup, ignorera tous les mots. De nouveaux appels aux commandes `ALTER TEXT SEARCH CONFIGURATION` doivent être utilisés pour créer les correspondances et rendre la configuration réellement utile. Autrement, une configuration de recherche plein texte peut être copiée.

Si un nom de schéma est précisé, alors le modèle de recherche plein texte est créé dans le schéma indiqué. Sinon il est créé dans le schéma en cours.

L'utilisateur qui définit une configuration de recherche plein texte en devient son propriétaire.

Voir Chapitre 12 pour plus d'informations.

Paramètres

nom

Le nom de la configuration de recherche plein texte (pouvant être qualifié du schéma).

parser_name

Le nom de l'analyseur de recherche plein texte à utiliser pour cette configuration.

source_config

Le nom d'une configuration existante de recherche plein texte à copier.

Notes

Les options `PARSER` et `COPY` sont mutuellement exclusives car, quand une configuration existante est copiée, sa sélection de son analyseur est aussi copiée.

Compatibilité

Il n'existe pas d'instruction `CREATE TEXT SEARCH CONFIGURATION` dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH DICTIONARY — définir un dictionnaire de recherche plein texte

Synopsis

```
CREATE TEXT SEARCH DICTIONARY nom (
    TEMPLATE = modele
    [, option = valeur [, ... ]]
)
```

Description

`CREATE TEXT SEARCH DICTIONARY` crée un nouveau dictionnaire de recherche plein texte. Un dictionnaire de recherche plein texte indique une façon de distinguer les mots intéressants à rechercher des mots inintéressants. Un dictionnaire dépend d'un modèle de recherche plein texte qui spécifie les fonctions qui font réellement le travail. Typiquement, le dictionnaire fournit quelques options qui contrôlent le comportement détaillé des fonctions du modèle.

Si un nom de schéma est précisé, alors le dictionnaire de recherche plein texte est créé dans le schéma indiqué. Sinon il est créé dans le schéma en cours.

L'utilisateur qui définit un dictionnaire de recherche plein texte en devient son propriétaire.

Voir Chapitre 12 pour plus d'informations.

Paramètres

nom

Le nom du dictionnaire de recherche plein texte (pouvant être qualifié du schéma).

modele

Le nom du modèle de recherche plein texte qui définira le comportement basique de ce dictionnaire.

option

Le nom d'une option, spécifique au modèle, à configurer pour ce dictionnaire.

valeur

La valeur à utiliser pour une option spécifique au modèle. Si la valeur n'est pas un simple identifiant ou un nombre, elle doit être entre guillemets simples (mais vous pouvez toujours le faire si vous le souhaitez).

Les options peuvent apparaître dans n'importe quel ordre.

Exemples

La commande exemple suivante crée un dictionnaire basé sur Snowball avec une liste spécifique de mots d'arrêt.

```
CREATE TEXT SEARCH DICTIONARY mon_dico_russe (
    template = snowball,
    language = russian,
    stopwords = myrussian
);
```

Compatibilité

Il n'existe pas d'instructions `CREATE TEXT SEARCH DICTIONARY` dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH PARSER

CREATE TEXT SEARCH PARSER — définir un nouvel analyseur de recherche plein texte

Synopsis

```
CREATE TEXT SEARCH PARSER nom (
    START = fonction_debut ,
    GETTOKEN = function_gettoken ,
    END = fonction_fin ,
    LEXTYPES = fonction_lextypes
    [, HEADLINE = fonction_headline ]
)
```

Description

`CREATE TEXT SEARCH PARSER` crée un nouvel analyseur de recherche plein texte. Un analyseur de recherche plein texte définit une méthode pour diviser une chaîne en plusieurs jetons et pour assigner des types (catégories) aux jetons. Un analyseur n'est pas particulièrement utile en lui-même mais doit être limité dans une configuration de recherche plein texte avec certains dictionnaires de recherche plein texte à utiliser pour la recherche.

Si un nom de schéma est précisé, alors le dictionnaire de recherche plein texte est créé dans le schéma indiqué. Sinon il est créé dans le schéma en cours.

Vous devez être un superutilisateur pour utiliser `CREATE TEXT SEARCH PARSER`. (Cette restriction est faite parce que la définition d'un analyseur de recherche plein texte peut gêner, voire arrêter brutalement, le serveur.)

Voir Chapitre 12 pour plus d'informations.

Paramètres

name

Le nom d'un analyseur de recherche plein texte (pouvant être qualifié du schéma).

fonction_debut

Le nom d'une fonction de démarrage pour l'analyseur.

fonction_gettoken

Le nom d'une fonction pour l'obtention du prochain jeton (get-next-token) pour l'analyseur.

fonction_fin

Le nom de la fonction d'arrêt de l'analyseur.

fonction_lextypes

Le nom de la fonction lextypes pour l'analyseur (une fonction qui renvoie de l'information sur l'ensemble de types de jeton qu'il produit).

fonction_headline

Le nom de la fonction headline pour l'analyseur (une fonction qui résume un ensemble de jetons).

Les noms des fonctions peuvent se voir qualifier du nom du schéma si nécessaire. Le type des arguments n'est pas indiqué car la liste d'argument pour chaque type de fonction est prédéterminé. Toutes les fonctions sont obligatoires sauf headline.

Les options peuvent apparaître dans n'importe quel ordre, pas seulement celui indiqué ci-dessus.

Compatibilité

Il n'existe pas d'instruction `CREATE TEXT SEARCH PARSER` dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

CREATE TEXT SEARCH TEMPLATE

CREATE TEXT SEARCH TEMPLATE — définir un nouveau modèle de recherche plein texte

Synopsis

```
CREATE TEXT SEARCH TEMPLATE nom (
    [ INIT = fonction_init , ]
    LEXIZE = fonction_lexize
)
```

Description

`CREATE TEXT SEARCH TEMPLATE` crée un nouveau modèle de recherche plein texte. Les modèles de recherche plein texte définissent les fonctions qui implémentent les dictionnaires de recherche plein texte. Un modèle n'est pas utile en lui-même mais doit être instancié par un dictionnaire pour être utilisé. Le dictionnaire spécifie typiquement les paramètres à donner aux fonctions modèle.

Si un nom de schéma est précisé, alors le modèle de recherche plein texte est créé dans le schéma indiqué. Sinon il est créé dans le schéma en cours.

Vous devez être un superutilisateur pour utiliser `CREATE TEXT SEARCH TEMPLATE`. Cette restriction est faite parce que la définition d'un modèle de recherche plein texte peut gêner, voire arrêter brutalement le serveur. La raison de la séparation des modèles et des dictionnaires est qu'un modèle encapsule les aspects « non sûrs » de la définition d'un dictionnaire. Les paramètres qui peuvent être définis lors de la mise en place d'un dictionnaire sont suffisamment sûrs pour être utilisé par des utilisateurs sans droits. Du coup, la création d'un dictionnaire ne demande pas de droits particuliers.

Voir Chapitre 12 pour plus d'informations.

Paramètres

nom

Le nom du modèle de recherche plein texte (pouvant être qualifié du schéma).

fonction_init

Le nom de la fonction d'initialisation du modèle.

fonction_lexize

Le nom de la fonction lexize du modèle.

Les noms des fonctions peuvent se voir qualifier du nom du schéma si nécessaire. Le type des arguments n'est pas indiqué car la liste d'argument pour chaque type de fonction est prédéterminé. La fonction lexize est obligatoire mais la fonction init est optionnelle.

Les arguments peuvent apparaître dans n'importe quel ordre, pas seulement dans celui indiqué ci-dessus.

Compatibilité

Il n'existe pas d'instruction `CREATE TEXT SEARCH TEMPLATE` dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

CREATE TRANSFORM

CREATE TRANSFORM — définir une nouvelle transformation

Synopsis

```
CREATE [ OR REPLACE ] TRANSFORM FOR nom_type LANGUAGE nom_lang (
    FROM SQL WITH FUNCTION nom_fonction_from_sql_ [ (type_argument
 [, ...]) ],
    TO SQL WITH FUNCTION nom_fonction_to_sql_ [ (type_argument
 [, ...]) ]
);
```

Description

`CREATE TRANSFORM` définit une nouvelle transformation. `CREATE OR REPLACE TRANSFORM` va soit créer une nouvelle transformation, soit en remplacer une déjà existante.

Une transformation définit comment adapter un type de données à un langage procédural. Par exemple, pour une fonction en PL/Python écrite avec le type `hstore`, PL/Python n'a pas les informations permettant de déterminer comment il doit présenter ces valeurs dans un environnement Python. Par défaut, les implémentations d'un langage vont utiliser la représentation typée text, mais c'est loin d'être optimal lorsque la représentation devrait être typée en tableau associatif ou liste.

Une transformation spécifie 2 fonctions :

- Une fonction « from SQL » qui convertit le type depuis l'environnement SQL vers le langage. Cette fonction sera appelée pour un argument d'une fonction écrite dans ce langage.
- Une fonction « to SQL » qui convertit le type depuis le langage vers l'environnement SQL. Cette fonction sera appelée sur la valeur retournée par une fonction écrite dans ce langage.

Il n'est pas nécessaire de définir l'ensemble de ces fonctions. Si l'une d'entre elle n'est pas spécifiée et au besoin le comportement par défaut du langage sera appliqué. (Pour éviter qu'une transformation soit effectuée dans un sens, vous pouvez aussi écrire une fonction de transformation qui renvoie systématiquement une erreur.)

Pour pouvoir créer une transformation, vous devez être le propriétaire du type et avoir le droit `USAGE` sur le type et le droit `USAGE` sur le langage, ainsi qu'être le propriétaire et avoir le droit `EXECUTE` sur les fonctions from-SQL et to-SQL si spécifié.

Paramètres

nom_type

Le nom du type de données de la transformation.

nom_lang

Le nom du langage de la transformation.

nom_fonction_from_sql[(*type_argument* [, ...])]

Nom de la fonction qui va convertir le type depuis l'environnement SQL vers le langage. Il doit prendre un argument type `internal` et renvoyer un type `internal` . L'argument présent

sera du type de la transformation, et la fonction devrait être codée en tant que tel. (Mais il n'est pas autorisé de déclarer une fonction de niveau SQL qui retournerait un type `internal` sans avoir au moins un argument de type `internal`.) La valeur retournée sera spécifique à ce qui est implémenté dans le langage. Si aucune liste n'est spécifiée en argument, le nom de la fonction doit être unique dans son schéma.

nom_fonction_to_sql[(*type_argument* [, ...])]

Nom de la fonction qui va convertir le type depuis le langage vers l'environnement SQL. Il doit prendre un argument de type `internal` et renvoyer un type qui est le type de la transformation. Cet argument sera spécifique à ce qui est implémenté dans le langage. Si aucune liste n'est spécifiée en argument, le nom de la fonction doit être unique dans son schéma.

Notes

Utiliser `DROP TRANSFORM` pour supprimer des transformations.

Exemples

Pour créer une transformation pour le type `hstore` et le langage `plpython3u`, il faut d'abord définir le type et le langage :

```
CREATE TYPE hstore ...;

CREATE EXTENSION plpython3u;
```

Puis créer les fonctions idoines :

```
CREATE FUNCTION hstore_to_plpython(val internal) RETURNS internal
LANGUAGE C STRICT IMMUTABLE
AS ...;

CREATE FUNCTION plpython_to_hstore(val internal) RETURNS hstore
LANGUAGE C STRICT IMMUTABLE
AS ...;
```

Et enfin, créer la transformation pour les lier ensemble :

```
CREATE TRANSFORM FOR hstore LANGUAGE plpython3u (
    FROM SQL WITH FUNCTION hstore_to_plpython(internal),
    TO SQL WITH FUNCTION plpython_to_hstore(internal)
);
```

En pratique, cette commande est encapsulée dans les extensions.

La section `contrib` contient un certain nombre d'extensions fournissant des transformations, qui peuvent être utilisés comme des exemples concrets.

Compatibilité

Cette forme de `CREATE TRANSFORM` est une extension PostgreSQL. Il existe une commande `CREATE TRANSFORM` dans le standard SQL, mais elle est utilisée pour adapter les types de données aux langages clients. Cette utilisation n'est pas supportée par PostgreSQL.

Voir aussi

CREATE FUNCTION, CREATE LANGUAGE, CREATE TYPE, DROP TRANSFORM

CREATE TRIGGER

CREATE TRIGGER — Définir un nouveau trigger

Synopsis

```
CREATE [ OR REPLACE ] [ CONSTRAINT ] TRIGGER nom { BEFORE | AFTER |
 INSTEAD OF } { événement [ OR ... ] }
    ON nom_table
    [ FROM nom_table_referencee ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE |
 INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE
 [ AS ] nom_relation_transition } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE { FUNCTION | PROCEDURE } nom_fonction ( arguments )

où événement fait partie de :

    INSERT
    UPDATE [ OF nom_colonne [, ... ] ]
    DELETE
    TRUNCATE
```

Description

`CREATE TRIGGER` crée un nouveau trigger. `CREATE OR REPLACE TRIGGER` va soit créer un nouveau trigger, soit remplacer un trigger existant. Le trigger est associé à la table, à la vue ou à la table distante spécifiée et exécute la fonction *`nom_fonction`* lorsque certaines opérations sont réalisées sur cette table.

Pour remplacer la définition actuelle d'un trigger existant, utilisez `CREATE OR REPLACE TRIGGER`, en indiquant le nom du trigger existant et la table parent. Toutes les autres propriétés sont remplacés.

L'appel du trigger peut avoir lieu avant que l'opération ne soit tentée sur une ligne (avant la vérification des contraintes et la tentative d'`INSERT`, `UPDATE` ou `DELETE`) ou une fois que l'opération est terminée (après la vérification des contraintes et la fin de la commande `INSERT`, `UPDATE` ou `DELETE`) ; ou bien en remplacement de l'opération (dans le cas d'opérations INSERT, UPDATE ou DELETE sur une vue). Si le trigger est lancé avant l'événement ou en remplacement de l'événement, le trigger peut ignorer l'opération sur la ligne courante ou modifier la ligne en cours d'insertion (uniquement pour les opérations `INSERT` et `UPDATE`). Si le trigger est activé après l'événement, toute modification, dont celles effectuées par les autres triggers, est « visible » par le trigger.

Un trigger marqué `FOR EACH ROW` est appelé pour chaque ligne que l'opération modifie. Par exemple, un `DELETE` affectant dix lignes entraîne dix appels distincts de tout trigger `ON DELETE` sur la relation cible, une fois par ligne supprimée. Au contraire, un trigger marqué `FOR EACH STATEMENT` ne s'exécute qu'une fois pour une opération donnée, quelque soit le nombre de lignes modifiées (en particulier, une opération qui ne modifie aucune ligne résulte toujours en l'exécution des triggers `FOR EACH STATEMENT` applicables).

Les triggers définis en remplacement (`INSTEAD OF`) doivent obligatoirement être marqués `FOR EACH ROW`, et ne peuvent être définis que sur des vues. Les triggers `BEFORE` et `AFTER` portant sur des vues devront quant à eux être marqués `FOR EACH STATEMENT`.

Les triggers peuvent également être définis pour l'événement TRUNCATE, mais ne pourront, dans ce cas, qu'être marqués FOR EACH STATEMENT.

Le tableau suivant récapitule quels types de triggers peuvent être utilisés sur les tables, les vues et les tables distantes :

Déclenchement	Événement	Niveau ligne	Niveau instruction
BEFORE	INSERT/ UPDATE/DELETE	Tables et tables distantes	Tables, vues et tables distantes
	TRUNCATE	--	Tables
AFTER	INSERT/ UPDATE/DELETE	Tables et tables distantes	Tables, vues et tables distantes
	TRUNCATE	--	Tables
INSTEAD OF	INSERT/ UPDATE/DELETE	Vues	--
	TRUNCATE	--	--

De plus, les triggers peuvent être définis pour être déclenchés suite à l'exécution d'un TRUNCATE, mais seulement dans le cas d'un trigger FOR EACH STATEMENT.

En outre, la définition d'un trigger peut spécifier une condition WHEN qui sera testée pour vérifier si le trigger doit réellement être déclenché. Dans les triggers au niveau ligne, la condition WHEN peut examiner l'ancienne et/ou la nouvelle valeurs des colonnes de la ligne. Les triggers au niveau instruction peuvent aussi avoir des conditions WHEN, bien que la fonctionnalité n'est pas aussi utile pour elles car la condition ne peut pas faire référence aux valeurs de la table.

Si plusieurs triggers du même genre sont définis pour le même événement, ils sont déclenchés suivant l'ordre alphabétique de leur nom.

Lorsque l'option CONSTRAINT est spécifiée, cette commande crée un *trigger contrainte*. Ce nouvel objet est identique aux triggers normaux excepté le fait que le moment de déclenchement peut alors être ajusté via l'utilisation de SET CONSTRAINTS. Les triggers contraintes ne peuvent être que de type AFTER ROW sur des tables standards (pas des tables distantes). Ils peuvent être déclenchés soit à la fin de l'instruction causant l'événement, soit à la fin de la transaction ayant contenu l'instruction de déclenchement ; dans ce dernier cas, ils sont alors définis comme *différés*. L'exécution d'un trigger différé peut également être forcée en utilisant l'option SET CONSTRAINTS. Le comportement attendu des triggers contraintes est de générer une exception en cas de violation de la contrainte qu'ils implémentent.

L'option REFERENCING active la récupération des *relations de transition*, qui sont des ensembles de lignes incluant toutes les lignes insérées, supprimées ou modifiées par l'instruction SQL en cours. Cette fonctionnalité donne au trigger une vue globale de ce qu'a réalisé l'instruction, et non pas une vue ligne par ligne. Cette option est seulement autorisée pour un trigger AFTER sur une table standard (pas une table distante). Ce trigger ne doit pas être un trigger de contrainte. De plus, si le trigger est un trigger UPDATE, il ne doit pas indiquer une liste de *nom_colonne* lors de l'utilisation de cette option. OLD TABLE peut seulement être indiqué une fois, et seulement pour un trigger qui est déclenché par un UPDATE ou un DELETE ; il crée une relation de transition contenant les *images-avant* de toutes les lignes mises à jour ou supprimées par l'instruction. De la même façon, NEW TABLE ne peut être indiqué qu'une seule fois, et seulement pour un trigger déclenché par un UPDATE ou un INSERT ; il crée une relation de transition contenant les *images-après* de toutes les lignes mises à jour ou insérées par l'instruction.

SELECT ne modifie aucune ligne ; la création de triggers sur SELECT n'est donc pas possible. Les règles et vues peuvent fournir des solutions fonctionnelles aux problèmes qui nécessitent des triggers sur SELECT.

Chapitre 39 présente de plus amples informations sur les triggers.

Paramètres

nom

Le nom du nouveau trigger. Il doit être distinct du nom de tout autre trigger sur la table. Le nom ne peut pas être qualifié d'un nom de schéma, le trigger héritant du schéma de sa table. Pour un trigger contrainte, c'est également le nom à utiliser lorsqu'il s'agira de modifier son comportement via la commande SET CONSTRAINTS.

BEFORE
AFTER
INSTEAD OF

Détermine si la fonction est appelée avant, après ou en remplacement de l'événement. Un trigger contrainte ne peut être spécifié qu'AFTER.

événement

Peut-être INSERT, UPDATE ou DELETE ou TRUNCATE ; précise l'événement qui active le trigger. Plusieurs événements peuvent être précisés en les séparant par OR, sauf quand les tables de transitions sont demandées.

Pour les triggers se déclenchant suite à un UPDATE, il est possible de spécifier une liste de colonnes utilisant cette syntaxe :

```
UPDATE OF nom_colonne_1 [, nom_colonne_2 ... ]
```

Le trigger se déclenchera seulement si au moins une des colonnes listées est mentionnée comme cible de la commande UPDATE ou si une des colonnes listées est une colonne générée qui dépend d'une colonne cible d'une commande UPDATE.

Les événements INSTEAD OF UPDATE n'acceptent pas de listes de colonnes. Une liste de colonnes ne peut pas être indiquée lorsque les tables de transition sont nécessaires.

nom_table

Le nom (éventuellement qualifié du nom du schéma) de la table, de la vue ou de la table distante à laquelle est rattaché le trigger.

nom_table_referencee

Le nom d'une autre table (possiblement qualifiée par un nom de schéma) référencée par la contrainte. Cette option est à utiliser pour les contraintes de clés étrangères et n'est pas recommandée pour d'autres types d'utilisation. Elle ne peut être spécifiée que pour les triggers contraintes.

DEFERRABLE
NOT DEFERRABLE
INITIALLY IMMEDIATE
INITIALLY DEFERRED

La spécification du moment de déclenchement par défaut. Voir la partie CREATE TABLE pour plus de détails sur cette option. Elle ne peut être spécifiée que pour les triggers contraintes.

REFERENCING

Ce mot-clé précède immédiatement la déclaration d'une ou deux noms de table fournissant l'accès aux relations de transition de l'instruction trigger.

`OLD TABLE`
`NEW TABLE`

Cette clause indique si le nom de la relation suivante est pour la relation de transition précédente ou suivante.

`nom_relation_transition`

Le nom (non qualifié) à utiliser au sein du trigger pour cette relation de transition.

`FOR EACH ROW`
`FOR EACH STATEMENT`

Précise si la fonction trigger doit être lancée pour chaque ligne affectée par l'événement ou simplement pour chaque instruction SQL. `FOR EACH STATEMENT` est la valeur par défaut. Les triggers de contrainte ne peuvent être spécifiés que pour `FOR EACH ROW`.

`condition`

Une expression booléenne qui détermine si la fonction trigger sera réellement exécutée. Si `WHEN` est indiqué, la fonction sera seulement appelée si la *`condition`* renvoie `true`. Pour les triggers `FOR EACH ROW`, la condition `WHEN` peut faire référence aux valeurs des colonnes des ancienne et nouvelle lignes en utilisant la notation `OLD.`*`nom_colonne`* ou `NEW.`*`nom_colonne`*, respectivement. Bien sûr, le triggers sur `INSERT` ne peuvent pas faire référence à `OLD` et ceux sur `DELETE` ne peuvent pas faire référence à `NEW`.

Les triggers `INSTEAD OF` ne supportent pas de condition `WHEN`.

Actuellement, les expressions `WHEN` ne peuvent pas contenir de sous-requêtes.

À noter que pour les triggers contraintes, l'évaluation de la clause `WHEN` n'est pas différée mais intervient immédiatement après que l'opération de mise à jour de la ligne soit effectuée. Si la condition n'est pas évaluée à vrai, alors le trigger n'est pas placé dans la file d'attente des exécutions différées.

`nom_fonction`

Une fonction utilisateur, déclarée sans argument et renvoyant le type `trigger`, exécutée à l'activation du trigger.

Dans la syntaxe de `CREATE TRIGGER`, les mots-clés `FUNCTION` et `PROCEDURE` sont équivalents mais la fonction référencée doit dans tous les cas être une fonction, et non pas une procédure. L'utilisation du mot-clé `PROCEDURE` est ici historique et dépréciée.

`arguments`

Une liste optionnelle d'arguments séparés par des virgules à fournir à la fonction lors de l'activation du trigger. Les arguments sont des chaînes littérales constantes. Il est possible d'écrire ici de simples noms et des constantes numériques mais ils sont tous convertis en chaîne. L'accès aux arguments du trigger depuis la fonction peut différer de l'accès aux arguments d'une fonction standard ; la consultation des caractéristiques d'implantation du langage de la fonction peut alors s'avérer utile.

Notes

Pour créer ou remplacer un trigger sur une table, l'utilisateur doit posséder le droit `TRIGGER` sur la table. L'utilisateur doit aussi avoir le droit `EXECUTE` sur la fonction trigger.

Utiliser `DROP TRIGGER` pour supprimer un trigger.

Créer un trigger niveau ligne sur une table partitionnée causera la création d'un trigger « clone » identique sur chacune des partitions existantes ; et toute partition créée ou attachée après aura aussi un trigger identique. S'il existe déjà un trigger de même nom sur une partition, une erreur est renvoyée sauf si `CREATE OR REPLACE TRIGGER` est utilisé, auquel cas ce trigger est remplacé avec un trigger clone. Quand une partition est détachée de son parent, ses triggers clone sont supprimés.

Un trigger sur colonne spécifique (définie en utilisant la syntaxe `UPDATE OF` *`nom_colonne`*) se déclenchera quand une des colonnes indiquées est listée comme cible de la liste `SET` pour la commande `UPDATE`. Il est possible qu'une valeur de colonne change même si le trigger n'est pas déclenché parce que les modifications au contenu de la ligne par les triggers `BEFORE UPDATE` ne sont pas pris en compte. De même, une commande comme `UPDATE ... SET x = x ...` déclenchera le trigger sur la colonne `x`, bien que la valeur de cette colonne ne change pas.

Dans un trigger `BEFORE`, la condition `WHEN` est évaluée juste avant l'exécution de la fonction, donc utiliser `WHEN` n'est pas matériellement différent de tester la même condition au début de la fonction trigger. Notez en particulier que la ligne `NEW` vu par la condition est sa valeur courante et possiblement modifiée par des triggers précédents. De plus, la condition `WHEN` d'un trigger `BEFORE` n'est pas autorisé à examiner les colonnes système de la ligne `NEW` (comme le `ctid`), car elles n'auront pas encore été initialisées.

Dans un trigger `AFTER`, la condition `WHEN` est évaluée juste après la mise à jour de la ligne et elle détermine si un événement doit déclencher le trigger à la fin de l'instruction. Donc, quand la condition `WHEN` d'un trigger `AFTER` ne renvoie pas true, il n'est pas nécessaire de préparer un événement ou de relire la ligne à la fin de l'instruction. Cela peut apporter une amélioration significative des performances dans les instructions qui modifient de nombreuses lignes, si le trigger a besoin d'être déclencher pour quelques lignes.

Dans certains cas, il est possible pour une seule commande SQL de déclencher plus d'un type de trigger. Par exemple, un `INSERT` avec une clause `ON CONFLICT DO UPDATE` peut être la cause du déclenchement d'opérations d'insertion et de mise à jour, donc il déclenchera l'exécution des deux types de trigger. Les relations de transition fournies par les triggers sont spécifique au type de l'événement. Donc un trigger `INSERT` ne verra que les lignes insérées, alors qu'un `UPDATE` ne verra que les lignes mises à jour.

Les mises à jour et suppressions de lignes causées par des actions dûes aux clés étrangères, comme un `ON UPDATE CASCADE` ou un `ON DELETE SET NULL`, sont traitées comme faisant partie de la commande SQL qui les a causé (notez que ces actions ne sont jamais différées). Les triggers adéquats sur la table impactée seront déclenchées, donc cela fournit un autre moyen avec lequel une commande SQL pourrait déclencher des triggers ne correspondant pas directement à son type. Dans les cas simples, les triggers demandant les relations de transition verront tous les changements causés dans leur table par une commande SQL simple comme une relation de transition unique. Néanmoins, il existe des cas où la présence d'un trigger `AFTER ROW` réclament les relations de transition causera que les actions des clés étrangères déclenchées par une commande SQL simple soient séparées en plusieurs étapes, chacune avec ses propres relations de transition. Dans de tels cas, tout trigger de niveau instruction présent se déclenchera une fois par ensemble de relation de transition créé, s'assurant ainsi que les triggers voient bien chaque ligne affectée dans une seul relation de transition.

Modifier une table partitionnée ou une table avec des enfants héritées déclenche les triggers au niveau requête attachés à cette table spécifiquement nommée, mais pas les triggers au niveau requête de ses partitions ou tables filles. Par contre, les triggers au niveau ligne sont déclenchés pour pour toutes les partitions et tables enfants affectées. Si un trigger au niveau requête a été défini avec des relations de transactions nommées par une clause `REFERENCING`, alors les images avant et après des lignes sont visibles pour toutes les partitions affectées et pour toutes les tables filles. Dans le cas de l'héritage, les images de ligne incluent seulement les colonnes présentes dans la table où le trigger est attaché.

Actuellement, les triggers niveau ligne avec des relations de transition ne peuvent pas être définis sur des partitions ou des tables d'héritage. De plus, les triggers sur des tables partitionnées ne peuvent pas être `INSTEAD OF`.

Actuellement, l'option `OR REPLACE` n'est pas supportée par les triggers de contrainte.

Remplacer un trigger existant dans une transaction qui a déjà réalisée des actions de mises à jour sur la table du trigger n'est pas recommandé. Les décisions de déclenchement du trigger, ou des portions de ces décisions, qui ont déjà été réalisées ne seront pas reconsidérées, donc les effets pourraient être surprenant.

Il existe quelques fonctions triggers natives pouvant être utilisées pour résoudre des problèmes communs sans avoir à écrire son propre code trigger ; voir Section 9.28.

Les triggers de niveau instruction sur une vue sont déclenchés uniquement si l'action sur la vue est géré par un trigger niveau ligne `INSTEAD OF`. Si l'action est gérée par une règle `INSTEAD`, alors toute instruction émise par la règle est exécutée à la place de l'instruction originale nommant la vue, pour que les triggers qui seront déclenchés soient ceux des tables nommées dans les instructions de remplacement. De façon similaire, si la vue est en mise à jour automatique, alors l'action est gérée en réécrivant automatiquement l'instruction en une action sur la table de base de la vue pour que les triggers niveau instruction de la table de base soient déclenchés.

Exemples

Exécutez la fonction `check_account_update` quand une ligne de la table `accounts` est sur le point d'être mise à jour :

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

Modifiez la définition du trigger pour seulement exécuter la fonction si la colonne `balance` est indiquée comme cible dans la commande `UPDATE` :

```
CREATE OR REPLACE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE FUNCTION check_account_update();
```

Cette forme exécute la fonction seulement si la colonne `balance` a réellement changé de valeur :

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE FUNCTION check_account_update();
```

Appelle une fonction pour tracer les mises à jour de la table `accounts`, mais seulement si quelque chose a changé :

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE FUNCTION log_account_update();
```

Éxecute la fonction `view_insert_row` pour chacune des lignes à insérer dans la table sous-jacente à la vue my_view :

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
    EXECUTE FUNCTION view_insert_row();
```

Exécute la fonction `check_transfer_balances_to_zero` pour chaque commande pour confirmer que les lignes de `transfert` engendrent un net de zéro :

```
CREATE TRIGGER transfer_insert
    AFTER INSERT ON transfer
    REFERENCING NEW TABLE AS inserted
    FOR EACH STATEMENT
    EXECUTE FUNCTION check_transfer_balances_to_zero();
```

Exécute la fonction `check_matching_pairs` pour chaque ligne pour confirmer que les changement sont fait sur des pairs correspondantes au même moment (par la même commande) :

```
CREATE TRIGGER paired_items_update
    AFTER UPDATE ON paired_items
    REFERENCING NEW TABLE AS newtab OLD TABLE AS oldtab
    FOR EACH ROW
    EXECUTE PROCEDURE check_matching_pairs();
```

Section 39.4 contient un exemple complet d'une fonction trigger écrit en C.

Compatibilité

L'instruction `CREATE TRIGGER` de PostgreSQL implante un sous-ensemble du standard SQL. Les fonctionnalités manquantes sont :

- Bien que les tables de transition pour les triggers `AFTER` triggers sont spécifiés en utilisant la clause `REFERENCING` de la manière standard, les variables de lignes utilisées dans les triggers `FOR EACH ROW` peuvent ne pas être spécifiées dans la clause `REFERENCING`. Ils sont disponibles d'une façon qui dépend du langage dans lequel la fonction trigger est écrite, mais est fixe sur un langage. Certains langages se comportent effectivement comme s'il y avait une clause `REFERENCING` contenant `OLD ROW AS OLD NEW ROW AS NEW`.

- Le standard autorise l'utilisation de tables de transition avec les triggers `UPDATE` spécifique à une colonne mais dans ce cas, l'ensemble des lignes qui doit être visible dans les tables de transition dépend de la liste de colonnes du trigger. Ceci n'est pas encore implémenté dans PostgreSQL.

- PostgreSQL n'autorise comme action déclenchée que l'exécution d'une fonction utilisateur. Le standard SQL, en revanche, autorise l'exécution d'autres commandes SQL, telles que `CREATE TABLE`. Cette limitation de PostgreSQL peut être facilement contournée par la création d'une fonction utilisateur qui exécute les commandes désirées.

Le standard SQL définit l'ordre de création comme ordre de lancement des triggers multiples. PostgreSQL utilise l'ordre alphabétique de leur nom, jugé plus pratique.

Le standard SQL précise que les triggers `BEFORE DELETE` sur des suppressions en cascade se déclenchent *après* la fin du `DELETE` en cascade. PostgreSQL définit que `BEFORE DELETE` se

déclenche toujours avant l'action de suppression, même lors d'une action en cascade. Cela semble plus cohérent. Il existe aussi un comportement non standard quand les triggers `BEFORE` modifient les lignes ou empêchent les mises à jour causées par une action référente. Ceci peut amener à des violations de contraintes ou au stockage de données qui n'honorent pas la contrainte référentielle.

La capacité à préciser plusieurs actions pour un seul trigger avec `OR` est une extension PostgreSQL.

La possibilité d'exécuter un trigger suite à une commande `TRUNCATE` est une extension PostgreSQL du standard SQL, tout comme la possibilité de définir des triggers de niveau instruction sur des vues.

`CREATE CONSTRAINT TRIGGER` est une extension spécifique à PostgreSQL du standard SQL. L'option `OR REPLACE` est aussi une extension.

Voir aussi

ALTER TRIGGER, DROP TRIGGER, SET CONSTRAINTS, CREATE FUNCTION

CREATE TYPE

CREATE TYPE — Définir un nouveau type de données

Synopsis

```
CREATE TYPE nom AS
    ( nom_attribut type_donnée [ COLLATE collation ] [, ... ] )

CREATE TYPE nom AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
    SUBTYPE = sous_type
    [ , SUBTYPE_OPCLASS = classe_operateur_sous_type ]
    [ , COLLATION = collationnement ]
    [ , CANONICAL = fonction_canonique ]
    [ , SUBTYPE_DIFF = fonction_diff_sous_type ]
    [ , MULTIRANGE_TYPE_NAME = nom_type_multirange ]
)

CREATE TYPE nom (
    INPUT = fonction_entrée,
    OUTPUT = fonction_sortie
    [ , RECEIVE = fonction_réception ]
    [ , SEND = fonction_envoi ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = fonction_analyse ]
    [ , SUBSCRIPT = fonction_indice ]
    [ , INTERNALLENGTH = { longueurinterne | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignement ]
    [ , STORAGE = stockage ]
    [ , LIKE = type_like ]
    [ , CATEGORY = catégorie ]
    [ , PREFERRED = préféré ]
    [ , DEFAULT = défaut ]
    [ , ELEMENT = élément ]
    [ , DELIMITER = délimiteur ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE nom
```

Description

`CREATE TYPE` enregistre un nouveau type de données utilisable dans la base courante. L'utilisateur qui définit un type en devient le propriétaire.

Si un nom de schéma est précisé, le type est créé dans ce schéma. Sinon, il est créé dans le schéma courant. Le nom du type doit être distinct du nom de tout type ou domaine existant dans le même schéma. Les tables possèdent des types de données associés. Il est donc nécessaire que le nom du type soit également distinct du nom de toute table existant dans le même schéma.

Il existe cinq formes de CREATE TYPE, comme indiqué dans la syntaxe ci-dessus. Elles créent respectivement un *type composite*, un *type enum*, un *type range (intervalle)*, un *type de base* ou un *type shell*. Les autre premiers sont discutés dans l'ordre ci-dessous. Un type shell est un simple conteneur de type qui sera défini ultérieurement. Il est créé en lançant CREATE TYPE sans paramètre en dehors de son nom. Les types shell sont nécessaires comme référence lors de la création de types intervalles et de types de base, comme indiqué dans ces sections.

Types composites

La première forme de CREATE TYPE crée un type composite. Le type composite est défini par une liste de noms d'attributs et de types de données. Un collationnement d'attribute peut aussi être spécifié si son type de données est collationnable. Un type composite est essentiellement le même que le type ligne (NDT : *row type* en anglais) d'une table, mais l'utilisation de CREATE TYPE permet d'éviter la création d'une table réelle quand seule la définition d'un type est voulue. Un type composite autonome est utile, par exemple, comme type d'argument ou de retour d'une fonction.

Pour pouvoir créer un type composite, vous devez avoir le droit USAGE sur les types de tous les attributs.

Types énumérés

La seconde forme de CREATE TYPE crée un type énuméré (enum), comme décrit dans Section 8.7. Les types enum prennent une liste de plusieurs labels entre guillemets, chacun devant faire moins de NAMEDATALEN octets (64 octets dans une installation PostgreSQL standard). (Il est possible de créer un type énuméré avec zéro label, mais un tel type ne peut pas être utilisé pour contenir des valeurs avant qu'au moins un label soit ajouté en utilisant ALTER TYPE.)

Types intervalles

La troisième forme de CREATE TYPE crée un type intervalle, comme décrit dans Section 8.17.

Le *sous-type* du type intervalle peut être de tout type qui soit associé avec une classe d'opérateurs B-tree (pour déterminer l'ordre des valeurs pour le type intervalle). Habituellement, la classe d'opérateurs par défaut du sous-type est utilisée pour déterminer l'ordre. Pour utiliser un opérateur de classe autre que celle par défaut, indiquez son nom avec *classe_operateur_sous_type*. Si le sous-type est collationnable et que vous voulez utiliser un collationnement autre que celui par défaut dans l'ordre de l'intervalle, indiquez le collationnement souhaité avec l'option *collationnement*.

La fonction optionnelle *canonique* prend un argument du type intervalle défini, et renvoie une valeur du même type. C'est utilisé pour convertir les valeurs intervalles en leur forme canonique, lorsque c'est applicable. Voir Section 8.17.8 pour plus d'informations. Créer une fonction *canonique* peut être un peu compliqué car il doit être défini avant que le type intervalle ne soit défini. Pour cela, vous devez tout d'abord créer un type shell, qui est un coquille vide qui n'a aucune propriété en dehors de son nom et propriétaire. Cela se crée en exécutant la commande CREATE TYPE *nom*, sans paramètre supplémentaire. Ensuite, la fonction peut être déclarée en utilisant le type shell comme argument et résultat. Enfin, le type intervalle peut être déclaré en utilisant le même nom. Ceci remplace automatiquement l'entrée du type shell avec un type intervalle valide.

La fonction optionnelle *diff_sous_type* doit prendre deux valeurs du type *sous-type* comme arguments, et renvoie une valeur de type double precision représentant la différence entre les deux valeurs données. Bien que cela soit optionnel, la fournir autorise une plus grande efficacité des index GiST sur les colonnes du type intervalle. Voir Section 8.17.8 pour plus d'informations.

Le paramètre optionnel *nom_type_multirange* indique le nom du type multirange correspondant. Si non specifié, ce nom est choisi automatiquement comme suit. Si le type de type range contient la sous-chaîne range, alors le nom du type multirange est formé par le remplacement de la sous-chaîne avec multirange dans le nom du type range. Sinon le nom du type multirange est formé en ajoutant le suffixe _multirange au nom du type range.

Pour être capable de créer un type range, vous devez avoir le droit USAGE sur le sous-type.

Types de base

La quatrième forme de CREATE TYPE crée un nouveau type de base (type scalaire). Pour créer un nouveau type de base, il faut être superutilisateur. (Cette restriction est imposée parce qu'une définition de type erronée pourrait embrouiller voire arrêter brutalement le serveur.)

L'ordre des paramètres, dont la plupart sont optionnels, n'a aucune d'importance. Avant de définir le type, il est nécessaire de définir au moins deux fonctions (à l'aide de la commande CREATE FUNCTION). Les fonctions de support *fonction_entrée* et *fonction_sortie* sont obligatoires. Les fonctions *fonction_réception*, *fonction_envoi*, *type_modifier_input_function*, *type_modifier_output_function*, *fonction_analyse* et *fonction_indice* sont optionnelles. Généralement, ces fonctions sont codées en C ou dans un autre langage de bas niveau.

La *fonction_entrée* convertit la représentation textuelle externe du type en représentation interne utilisée par les opérateurs et fonctions définis pour le type. La *fonction_sortie* réalise la transformation inverse. La fonction entrée peut être déclarée avec un argument de type cstring ou trois arguments de types cstring, oid, integer. Le premier argument est le texte en entrée sous la forme d'une chaîne C, le second argument est l'OID du type (sauf dans le cas des types tableau où il s'agit de l'OID du type de l'élément) et le troisième est le typmod de la colonne destination, s'il est connu (-1 sinon). La fonction entrée doit renvoyer une valeur du nouveau type de données. Habituellement, une fonction d'entrée devrait être déclarée comme STRICT si ce n'est pas le cas, elle sera appelée avec un premier paramètre NULL à la lecture d'une valeur NULL en entrée. La fonction doit toujours envoyer NULL dans ce cas, sauf si une erreur est rapportée. (Ce cas a pour but de supporter les fonctions d'entrée des domaines qui ont besoin de rejeter les entrées NULL.) La fonction sortie doit prendre un argument du nouveau type de données, et retourner le type cstring. Les fonctions sortie ne sont pas appelées pour des valeurs NULL.

La *fonction_réception*, optionnelle, convertit la représentation binaire externe du type en représentation interne. Si cette fonction n'est pas fournie, le type n'accepte pas d'entrée binaire. La représentation binaire est choisie de telle sorte que sa conversion en forme interne soit peu coûteuse, tout en restant portable. (Par exemple, les types de données standard entiers utilisent l'ordre réseau des octets comme représentation binaire externe alors que la représentation interne est dans l'ordre natif des octets de la machine.) La fonction de réception réalise les vérifications adéquates pour s'assurer que la valeur est valide. Elle peut être déclarée avec un argument de type internal ou trois arguments de types internal, integer et oid. Le premier argument est un pointeur vers un tampon StringInfo qui contient la chaîne d'octets reçue ; les arguments optionnels sont les mêmes que pour la fonction entrée de type texte. La fonction de réception retourne une valeur du type de données. Habituellement, une fonction de réception devrait être déclarée comme STRICT si ce n'est pas le cas, elle sera appelée avec un premier paramètre NULL à la lecture d'une valeur NULL en entrée. La fonction doit toujours envoyer NULL dans ce cas, sauf si une erreur est rapportée. (Ce cas a pour but de supporter les fonctions de réception des domaines qui ont besoin de rejeter les entrées NULL.) De façon similaire, la *fonction_envoi*, optionnelle, convertit la représentation interne en représentation binaire externe. Si cette fonction n'est pas fournie, le type n'accepte pas de sortie binaire. La fonction d'envoi doit être déclarée avec un argument du nouveau type de données et retourner le type bytea. Les fonctions réception ne sont pas appelées pour des valeurs NULL.

À ce moment-là, vous pouvez vous demander comment les fonctions d'entrée et de sortie peuvent être déclarées avoir un résultat ou un argument du nouveau type alors qu'elles sont à créer avant que le nouveau type ne soit créé. La réponse est que le type sera tout d'abord défini en tant que *type squelette* (*shell type*), une ébauche de type sans propriété à part un nom et un propriétaire. Ceci se fait en exécutant la commande CREATE TYPE *nom* sans paramètres supplémentaires. Ensuite, les fonctions d'entrée/sortie C peuvent être définies en référençant le squelette. Enfin, le CREATE TYPE avec une définition complète remplace le squelette avec une définition complète et valide du type, après quoi le nouveau type peut être utilisé normalement.

Les fonctions optionnelles *type_modifier_input_function* et *type_modifier_output_function* sont nécessaires si le type supporte des modificateurs, c'est-à-dire des contraintes optionnelles attachées à une déclaration de type comme char(5) ou

`numeric(30,2)`. PostgreSQL autorise les types définis par l'utilisateur à prendre une ou plusieurs constantes ou identifiants comme modifieurs ; néanmoins, cette information doit être capable d'être englobée dans une seule valeur entière positive pour son stockage dans les catalogues système. *`type_modifier_input_function`* se voit fournir le modifieur déclaré de la forme d'un tableau de `cstring`. Il doit vérifier la validité des valeurs et renvoyer une erreur si elles sont invalides. Dans le cas contraire, il renvoie une valeur entière positive qui sera stockée dans la colonne « typmod ». Les modifieurs de type seront rejetés si le type n'a pas de *`type_modifier_input_function`*. *`type_modifier_output_function`* convertit la valeur typmod integer en une forme correcte pour l'affichage. Il doit renvoyer une valeur de type `cstring` qui est la chaîne exacte à ajouter au nom du type ; par exemple la fonction de `numeric` pourrait renvoyer `(30,2)`. Il est permis d'omettre le *`type_modifier_output_function`*, auquel cas le format d'affichage par défaut est simplement la valeur typmod stockée entre parenthèses.

La *`fonction_analyse`*, optionnelle, calcule des statistiques spécifiques au type de données pour les colonnes de ce type. Par défaut, ANALYZE tente de récupérer des statistiques à l'aide des opérateurs d'« égalité » et d'« infériorité » du type, s'il existe une classe d'opérateurs B-tree par défaut pour le type. Ce comportement est inadapté aux types non-scalaires ; il peut être surchargé à l'aide d'une fonction d'analyse personnalisée. La fonction d'analyse doit être déclarée avec un seul argument de type `internal` et un résultat de type `boolean`. L'API détaillée des fonctions d'analyses est présentée dans `src/include/commands/vacuum.h`.

La fonction optionnelle *`fonction_indice`* autorise l'utilisation d'indice pour le type de données dans les commandes SQL. Indiquer cette fonction ne fait pas en sorte que le type soit considéré comme un « vrai » type tableau ; par exemple, il ne sera pas un candidat pour le type résultat de constructions ARRAY `[]`. Mais si indicer une valeur du type est une notation naturelle pour en extraire des données, alors une *`fonction_indice`* peut être écrite pour définir la signification. La fonction d'indice doit être déclarée comme prenant un seul argument de type `internal`, et renvoyer un résultat de type `internal`, qui est un pointeur vers une structure de méthodes (fonctions) qui implémentent l'indiçage. L'API détaillée pour les fonctions d'indice apparaît dans in `src/include/nodes/subscripting.h`. Il pourrait aussi être utile de lire l'implémentation des tableaux dans `src/backend/utils/adt/arraysubs.c`, ou le code simple de `contrib/hstore/hstore_subs.c`. Des informations supplémentaires apparaissent dans Types tableau, ci-dessous.

Alors que les détails de la représentation interne du nouveau type ne sont connus que des fonctions d'entrées/sorties et des fonctions utilisateurs d'interaction avec le type, plusieurs propriétés de la représentation interne doivent être déclarées à PostgreSQL. La première est *`longueurinterne`*. Les types de données basiques peuvent être de longueur fixe (dans ce cas, *`longueurinterne`* est un entier positif) ou de longueur variable (indiquée par le positionnement de *`longueurinterne`* à VARIABLE ; en interne, cela est représenté en initialisant `typlen` à -1). La représentation interne de tous les types de longueur variable doit commencer par un entier de quatre octets indiquant la longueur totale de cette valeur. (Notez que le champ length est souvent encodé, comme décrit dans Section 73.2 ; il n'est pas conseillé d'y accéder directement.)

Le drapeau optionnel PASSEDBYVALUE indique que les valeurs de ce type de données sont passées par valeur plutôt que par référence. Les types dont la représentation interne est plus grande que la taille du type `Datum` (quatre octets sur la plupart des machines, huit sur quelques-unes) ne doivent pas être passés par valeur. Les types passés par valeur doivent avoir une longueur fixe et leur représentation interne ne peut pas être plus large que la taille du type `Datum` (4 octets sur certaines machines, 8 octets sur d'autres).

Le paramètre *`alignement`* spécifie l'alignement de stockage requis pour le type de données. Les valeurs permises sont des alignements sur 1, 2, 4 ou 8 octets. Les types de longueurs variables ont un alignement d'au moins quatre octets car leur premier composant est nécessairement un `int4`.

Le paramètre *`stockage`* permet de choisir une stratégie de stockage pour les types de données de longueur variable. (Seul `plain` est autorisé pour les types de longueur fixe.) `plain` indique des données stockées en ligne et non compressées. Dans le cas d'`extended` le système essaie tout d'abord de compresser une valeur longue et déplace la valeur hors de la ligne de la table principale si elle est

toujours trop longue. `external` permet à la valeur d'être déplacée hors de la table principale mais le système ne tente pas de la compresser. `main` autorise la compression mais ne déplace la valeur hors de la table principale qu'en dernier recours. (Ils seront déplacés s'il n'est pas possible de placer la ligne dans la table principale, mais sont préférentiellement conservés dans la table principale, contrairement aux éléments `extended` et `external`.)

Toutes les valeurs *`storage`* autres que `plain` impliquent que les fonctions du type de données peuvent gérer les valeurs placées en *TOAST*, comme décrit dans Section 73.2 et Section 38.13.1. L'aure valeur spécifique détermine la stratégie de stockage par défaut pour les colonnes d'un type de données externalisable ; les utilisateurs peuvent sélectionner les autres stratégies pour les colonnes individuelles en utilisant `ALTER TABLE SET STORAGE`.

Le paramètre *`type_like`* fournit une méthode alternative pour spécifier les propriétés de représentation de base d'un type de données : les copier depuis un type existant. Les valeurs de *`longueurinterne`*, *`passedbyvalue`*, *`alignement`* et *`stockage`* sont copiées du type indiqué. (C'est possible, mais habituellement non souhaité, d'écraser certaines de ces valeurs en les spécifiant en même temps que la clause `LIKE`.) Spécifier la représentation de cette façon est particulièrement pratique quand l'implémentation de bas niveau du nouveau type emprunte celle d'un type existant d'une façon ou d'une autre.

Les paramètres *`catégorie`* et *`préféré`* peuvent être utilisés pour aider à contrôler la conversion implicite appliquée en cas d'ambiguïté. Chaque type de données appartient à une catégorie identifiée par un seul caractère ASCII, et chaque type est « préféré » ou pas de sa catégorie. L'analyseur préfèrera convertir vers des types préférés (mais seulement à partir d'autres types dans la même catégorie) quand cette règle peut servir à résoudre des fonctions ou opérateurs surchargés. Pour plus de détails, voir Chapitre 10. Pour les types qui n'ont pas de conversion implicite de ou vers d'autres types, on peut se contenter de laisser ces paramètres aux valeurs par défaut. Par contre, pour un groupe de types liés entre eux qui ont des conversions implicites, il est souvent pratique de les marquer tous comme faisant partie d'une même catégorie, et de choisir un ou deux des types les « plus généraux » comme étant les types préférés de la catégorie. Le paramètre *`catégorie`* est particulièrement utile quand on ajoute un type défini par l'utilisateur à un type interne, comme un type numérique ou chaîne. Toutefois, c'est aussi tout à fait possible de créer des catégories de types entièrement nouvelles. Choisissez un caractère ASCII autre qu'une lettre en majuscule pour donner un nom à une catégorie de ce genre.

Une valeur par défaut peut être spécifiée dans le cas où l'utilisateur souhaite que cette valeur soit différente de NULL pour les colonnes de ce type. La valeur par défaut est précisée à l'aide du mot clé `DEFAULT`. (Une telle valeur par défaut peut être surchargée par une clause `DEFAULT` explicite attachée à une colonne particulière.)

Pour indiquer qu'un type est un tableau de taille fixe, le type des éléments du tableau est précisé par le mot clé `ELEMENT`. Par exemple, pour définir un tableau d'entiers de quatre octets (`int4`), `ELEMENT = int4` est utilisé. Pour plus de détails, voir Types tableau ci-dessous.

Pour préciser le délimiteur de valeurs utilisé dans la représentation externe des tableaux de ce type, *`délimiteur`* peut être positionné à un caractère particulier. Le délimiteur par défaut est la virgule (`,`). Le délimiteur est associé avec le type élément de tableau, pas avec le type tableau.

Si le paramètre booléen optionnel *`collatable`* vaut true, les définitions et expressions de colonnes du type peuvent embarquer une information de collationnement via la clause `COLLATE`. C'est aux implémentations des fonctions du type de faire bon usage de cette information. Cela n'arrive pas automatiquement en marquant le type collationnable.

Types tableau

À chaque fois qu'un type défini par un utilisateur est créé, PostgreSQL crée automatiquement un type tableau associé dont le nom est composé à partir du type de base préfixé d'un tiret bas et tronqué si nécessaire pour que le nom généré fasse moins de `NAMEDATALEN` octets. (Si le nom généré est en conflit avec un autre nom, le traitement est répété jusqu'à ce qu'un nom sans conflit soit trouvé.) Ce type tableau créé implicitement est de longueur variable et utilise les fonctions d'entrée et sortie

`array_in` et `array_out`. De plus, ce type est ce que le système utilise pour les constructions telles que `ARRAY[]` sur le type défini par l'utilisateur. Le type tableau trace tout changement dans du type de base pour le propriétaire et le schéma. Il est aussi supprimé quand le type de base l'est.

Pourquoi existe-t-il une option `ELEMENT` si le système fabrique automatiquement le bon type tableau ? L'utilité principale d'`ELEMENT` est la création d'un type de longueur fixe représenté en interne par un tableau d'éléments identiques auxquels on souhaite accéder directement par leurs indices (en plus de toute autre opération effectuée sur le type dans sa globalité). Par exemple, le type `point` est représenté par deux nombres à virgule flottante, qui sont accessibles par `point[0]` et `point[1]`. Cette fonctionnalité n'est possible qu'avec les types de longueur fixe dont la forme interne est strictement une séquence de champs de longueur fixée. Pour des raisons historiques (c'est-à-dire pour de mauvaises raisons, mais il est trop tard pour changer) les indices des tableaux de types de longueur fixe commencent à zéro et non à un comme c'est le cas pour les tableaux de longueur variable.

Ajouter l'option `SUBSCRIPT` permet au type de données d'utiliser des indices, même si le système ne le gère pas comme un type tableau. Le comportement tout juste décrit pour les tableaux à longueur fixe est en fait implémenté par la fonction gestionnaire `SUBSCRIPT` appelée `raw_array_subscript_handler`, qui est utilisée automatiquement si vous indiquez `ELEMENT` pour un type de longueur fixe sans écrire aussi `SUBSCRIPT`.

Lors de l'indication explicite d'une fonction `SUBSCRIPT`, il n'est pas nécessaire d'indiquer `ELEMENT` sauf si la fonction gestionnaire `SUBSCRIPT` a besoin de consulter `typelem` pour savoir ce qu'elle renvoie. Faites attentent qu'indiquer `ELEMENT` cause le système à assumer que le nouveau type contient ou est quelque part physiquement dépendant, du type élément ; de ce fait, changer par exemple les propriétés du type élément n'est pas autorisé s'il y a des colonnes du type dépendant.

Paramètres

nom

Le nom (éventuellement qualifié du nom du schéma) du type à créer.

nom_attribut

Le nom d'un attribut (colonne) du type composite.

type_données

Le nom d'un type de données existant utilisé comme colonne du type composite.

collationnement

Le nom d'un collationnement existant à associer avec une colonne d'un type composite ou avec un type intervalle.

label

Une chaîne représentant le label associé à une valeur du type enum.

sous_type

Le nom du type élément dont le type intervalle va représenter des intervalles.

classe_operateur_sous_type

Le nom d'une classe d'opérateurs B-tree pour le sous-type.

fonction_canonique

Le nom de la fonction canonique pour le type intervalle.

fonction_diff_sous_type

Le nom de la fonction de différence pour le sous-type.

multirange_type_name

Le nom du type multirange correspondant.

fonction_entrée

Le nom d'une fonction de conversion des données de la forme textuelle externe du type en forme interne.

fonction_sortie

Le nom d'une fonction de conversion des données de la forme interne du type en forme textuelle externe.

fonction_réception

Le nom d'une fonction de conversion des données de la forme binaire externe du type en forme interne.

fonction_envoi

Le nom d'une fonction de conversion des données de la forme interne du type en forme binaire externe.

type_modifier_input_function

Le nom d'une fonction qui convertit un tableau de modifieurs pour le type vers sa forme interne.

type_modifier_output_function

Le nom d'une fonction qui convertit la forme interne des modifieurs du type vers leur forme textuelle externe.

fonction_analyze

Le nom d'une fonction d'analyses statistiques pour le type de données.

fonction_indice

Le nom d'une fonction qui définit ce qu'indicer une valeur de ce type de données signifie.

longueur_interne

Une constante numérique qui précise la longueur en octets de la représentation interne du nouveau type. Supposée variable par défaut.

alignement

La spécification d'alignement du stockage du type de données. Peut être `char`, `int2`, `int4` ou `double` ; `int4` par défaut.

stockage

La stratégie de stockage du type de données. Peut être `plain`, `external`, `extended` ou `main` ; `plain` par défaut.

type_like

Le nom d'un type de données existant dont le nouveau type partagera la représentation. Les valeurs de *longueurinterne*, *passedbyvalue*, *alignement* et *stockage* sont recopiées à

partir de ce type, sauf si elles sont écrasées explicitement ailleurs dans la même commande CREATE TYPE.

catégorie

Le code de catégorie (un unique caractère ASCII) pour ce type. La valeur par défaut est U pour « user-defined type » (type défini par l'utilisateur). Les autres codes standard de catégorie peuvent être trouvés dans Tableau 53.65. Vous pouvez aussi choisir d'autres caractères ASCII pour créer vos propres catégories personnalisées.

préféré

True si ce type est un type préféré dans sa catégorie de types, sinon false. La valeur par défaut est false. Faites très attention en créant un nouveau type préféré à l'intérieur d'une catégorie existante car cela pourrait créer des modifications surprenantes de comportement.

défaut

La valeur par défaut du type de données. Omise, elle est NULL.

élément

Type des éléments du type tableau créé.

délimiteur

Le caractère délimiteur des valeurs des tableaux de ce type.

collatable

Vrai si les opérations de ce type peuvent utiliser les informations de collationnement. Par défaut, à faux.

Notes

Comme il n'y a pas de restrictions à l'utilisation d'un type de données une fois qu'il a été créé, créer un type de base ou un type range est équivalent à donner les droits d'exécution sur les fonctions mentionnées dans la définition du type. Ce n'est pas un problème habituellement pour le genre de fonctions utiles dans la définition d'un type mais réfléchissez bien avant de concevoir un type d'une façon qui nécessiterait que des informations « secrètes » soient utilisées lors de sa convertion vers ou à partir d'une forme externe.

Avant PostgreSQL version 8.3, le nom d'un type tableau généré était toujours exactement le nom du type élément avec un caractère tiret bas (_) en préfixe. (Les noms des types étaient du coup limités en longueur à un caractère de moins que les autres noms.) Bien que cela soit toujours le cas, le nom d'un type tableau peut varier entre ceci dans le cas des noms de taille maximum et les collisions avec des noms de type utilisateur qui commencent avec un tiret bas. Écrire du code qui dépend de cette convention est du coup obsolète. À la place, utilisez pg_type.typarray pour situer le type tableau associé avec un type donné.

Il est conseillé d'éviter d'utiliser des noms de table et de type qui commencent avec un tiret bas. Alors que le serveur changera les noms des types tableau générés pour éviter les collisions avec les noms donnés par un utilisateur, il reste toujours un risque de confusion, particulièrement avec les anciens logiciels clients qui pourraient supposer que les noms de type commençant avec un tiret bas représentent toujours des tableaux.

Avant PostgreSQL version 8.2, la syntaxe de création d'un type shell CREATE TYPE *nom* n'existait pas. La façon de créer un nouveau type de base était de créer en premier les fonctions paramètres. Dans cette optique, PostgreSQL verra tout d'abord le nom d'un nouveau type de données comme type de retour de la fonction en entrée. Le type shell est créé implicitement dans ce cas et il est ensuite référencé

dans le reste des fonctions d'entrée/sortie. Cette approche fonctionne toujours mais est obsolète et pourrait être interdite dans une version future. De plus, pour éviter de faire grossir les catalogues de façon accidentelle avec des squelettes de type erronés, un squelette sera seulement créé quand la fonction en entrée est écrit en C.

Exemples

Créer un type composite utilisé dans la définition d'une fonction :

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

Cet exemple crée un type énuméré et l'utilise dans la création d'une table :

```
CREATE TYPE statut_bogue AS ENUM ('nouveau', 'ouvert', 'fermé');

CREATE TABLE bogue (
    id serial,
    description text,
    status statut_bogue
);
```

Cet exemple crée un type intervalle :

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff =
 float8mi);
```

Créer le type de données basique `box` utilisé dans la définition d'une table :

```
CREATE TYPE box;

CREATE FUNCTION ma_fonction_entree_box(cstring) RETURNS box
 AS ... ;
CREATE FUNCTION ma_fonction_sortie_box(box) RETURNS cstring
 AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = ma_fonction_entree_box,
    OUTPUT = ma_fonction_sortie_box
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

Si la structure interne de `box` est un tableau de quatre éléments `float4`, on peut écrire :

```
CREATE TYPE box (
```

```
    INTERNALLENGTH = 16,
    INPUT = ma_fonction_entree_box,
    OUTPUT = ma_fonction_sortie_box,
    ELEMENT = float4
);
```

ce qui permet d'accéder aux nombres composant la valeur d'une boîte par les indices. Le comportement du type n'est pas modifié.

Créer un objet large utilisé dans la définition d'une table :

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

D'autres exemples, intégrant des fonctions utiles d'entrée et de sortie, peuvent être consultés dans Section 38.13.

Compatibilité

La première forme de la commande `CREATE TYPE`, qui crée un type composite, est conforme au standard SQL. Les autres formes sont des extensions de PostgreSQL. L'instruction `CREATE TYPE` du standard SQL définit aussi d'autres formes qui ne sont pas implémentées dans PostgreSQL.

La possibilité de créer un type composite sans attributs est une différence spécifique de PostgreSQL que le standard ne propose pas (de façon analogue au `CREATE TABLE`).

Voir aussi

ALTER TYPE, CREATE DOMAIN, CREATE FUNCTION, DROP TYPE

CREATE USER

CREATE USER — Définir un nouveau rôle de base de données

Synopsis

```
CREATE USER nom [ [ WITH ] option [ ... ] ]

où option peut être :

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT limite_connexion
    | [ ENCRYPTED ] PASSWORD 'motdepasse' | PASSWORD NULL
    | VALID UNTIL 'dateheure'
    | IN ROLE nom_role [, ...]
    | IN GROUP nom_role [, ...]
    | ROLE nom_role [, ...]
    | ADMIN nom_role [, ...]
    | USER nom_role [, ...]
    | SYSID uid
```

Description

`CREATE USER` est dorénavant un alias de `CREATE ROLE`. Il y a toutefois une petite différence entre les deux commandes. Lorsque la commande `CREATE USER` est exécutée, `LOGIN` est le comportement par défaut. Au contraire, quand `CREATE ROLE` est exécutée, `NOLOGIN` est utilisé.

Compatibilité

L'instruction `CREATE USER` est une extension PostgreSQL. Le standard SQL laisse la définition des utilisateurs à l'implantation.

Voir aussi

CREATE ROLE

CREATE USER MAPPING

CREATE USER MAPPING — Définir une nouvelle correspondance d'utilisateur (*user mapping*) pour un serveur distant

Synopsis

```
CREATE USER MAPPING [ IF NOT EXISTS ] FOR { nom_utilisateur | USER
 | CURRENT_ROLE | CURRENT_USER | PUBLIC }
    SERVER nom_serveur
    [ OPTIONS ( option 'valeur' [ , ... ] ) ]
```

Description

`CREATE USER MAPPING` définit une nouvelle correspondance d'utilisateur (*user mapping*) pour un serveur distant. Une correspondance d'utilisateur englobe typiquement les informations de connexion qu'un wrapper de données distantes utilise avec l'information d'un serveur distant pour accéder à des ressources externes de données.

Le propriétaire d'un serveur distant peut créer des correspondances d'utilisateur pour ce serveur pour n'importe quel utilisateur. Par ailleurs, un utilisateur peut créer une correspondance d'utilisateur pour son propre nom d'utilisateur si le droit `USAGE` a été donné sur le serveur à son utilisateur.

Paramètres

`IF NOT EXISTS`

Ne remonte pas d'erreur si une correspondance pour l'utilisateur donné pour le serveur distant donné existe déjà. Une note est affichée dans ce cas. Veuillez noter qu'il n'y a aucune garantie que la correspondante d'utilisateur existante ait quoi que ce soit à voir avec celle qui aurait été créée.

`nom_utilisateur`

Le nom d'un utilisateur existant qui est mis en correspondance sur un serveur distant. `CURRENT_ROLE`, `CURRENT_USER` et `USER` correspondent au nom de l'utilisateur courant. Quand `PUBLIC` est ajoutée, une correspondance appelée publique est créée pour être utilisée quand aucune correspondance d'utilisateur spécifique n'est applicable.

`nom_serveur`

Le nom d'un serveur existant pour lequel la correspondance d'utilisateur sera créée.

`OPTIONS ( option 'valeur' [, ... ] )`

Cette clause définit les options pour la correspondance d'utilisateurs. Les options définissent typiquement le nom et le mot de passe réels de la correspondance. Les nom d'options doivent être uniques. Les noms et valeurs d'options autorisés sont propres au wrapper de données étrangère du serveur.

Exemples

Créer une correspondance d'utilisateur pour l'utilisateur `bob`, sur le serveur `truc` :

```
CREATE USER MAPPING FOR bob SERVER truc OPTIONS (user 'bob',
 password 'secret');
```

Compatibilité

`CREATE USER MAPPING` est conforme à la norme ISO/IEC 9075-9 (SQL/MED).

Voir aussi

ALTER USER MAPPING, DROP USER MAPPING, CREATE FOREIGN DATA WRAPPER, CREATE SERVER

CREATE VIEW

CREATE VIEW — Définir une vue

Synopsis

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW nom
 [ ( nom_colonne [, ...] ) ]
    [ WITH ( nom_option_vue [= valeur_option_vue] [, ... ] ) ]
    AS requête
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Description

`CREATE VIEW` définit une vue d'après une requête. La vue n'est pas matérialisée physiquement. Au lieu de cela, la requête est lancée chaque fois qu'une vue est utilisée dans une requête.

`CREATE OR REPLACE VIEW` a la même finalité, mais si une vue du même nom existe déjà, elle est remplacée. La nouvelle requête doit générer les mêmes colonnes que celles de l'ancienne requête (c-est-à-dire les mêmes noms de colonnes dans le même ordre avec les mêmes types de données). Par contre, elle peut ajouter des colonnes supplémentaires en fin de liste. Les traitements qui donnent les colonnes en sortie pourraient être complètement différents.

Si un nom de schéma est donné (par exemple `CREATE VIEW monschema.mavue ...`), alors la vue est créée dans ce schéma. Dans le cas contraire, elle est créée dans le schéma courant. Les vues temporaires existent dans un schéma spécial. Il n'est donc pas nécessaire de fournir de schéma pour les vues temporaires. Le nom de la vue doit être différent du nom de toute autre relation (table, séquence, index, vue, vue matérialisée ou table distante) du même schéma.

Paramètres

`TEMPORARY` ou `TEMP`

La vue est temporaire. Les vues temporaires sont automatiquement supprimées en fin de session. Les relations permanentes qui portent le même nom ne sont plus visibles pour la session tant que la vue temporaire existe, sauf s'il y est fait référence avec le nom du schéma.

Si l'une des tables référencées par la vue est temporaire, la vue est alors elle-aussi temporaire (que `TEMPORARY` soit spécifié ou non).

`RECURSIVE`

Crée une vue récursive. La syntaxe

```
CREATE RECURSIVE VIEW [ schéma . ] nom (colonnes) AS SELECT ...;
```

est équivalente à

```
CREATE VIEW [ schéma . ] nom AS WITH RECURSIVE nom (colonnes) AS
 (SELECT ...) SELECT colonne FROM nom;
```

Une liste de noms de colonne doit être spécifiée pour la vue récursive.

nom

Le nom de la vue à créer (éventuellement qualifié du nom du schéma).

nom de colonne

Une liste optionnelle de noms à utiliser pour les colonnes de la vue. Si elle n'est pas donnée, le nom des colonnes est déduit de la requête.

WITH (*nom de l'option de vue* [= *valeur de l'option*] [, ...])

Cette clause spécifie des paramètres optionnels pour une vue. Les paramètres supportés sont les suivants :

check_option (enum)

Ce paramètre peut avoir soit local soit cascaded, et est l'équivalent de spécifier WITH [CASCADED | LOCAL] CHECK OPTION (voir ci-dessous).

security_barrier (boolean)

Ceci doit être utilisé si la vue a pour but de fournir une sécurité au niveau ligne. Voir Section 41.5 pour plus de détails.

security_invoker (boolean)

Cette option fait que les relations de base sous-jacentes soient vérifiées par rapport aux droits de l'utilisateur de la base, plutôt que par rapport à son créateur. Voir les notes ci-dessous pour plus de détails.

Toutes les options ci-dessous peuvent être modifiées sur les vues existantes en utilisant ALTER VIEW.

requête

Une commande SELECT ou VALUES qui fournira les colonnes et lignes de la vue.

WITH [CASCADED | LOCAL] CHECK OPTION

Cette option contrôle le comportement des vues automatiquement modifiables. Quand cette option est spécifiée, les commandes INSERT et UPDATE sur la vue seront vérifiées pour s'assurer que les nouvelles lignes satisfont la condition définie dans la vue (autrement dit, les nouvelles lignes sont vérifiées pour s'assurer qu'elles sont visibles par la vue). Dans le cas contraire, la mise à jour est rejetée. Si l'option CHECK OPTION n'est pas indiquée, les commandes INSERT et UPDATE sur la vue sont autorisées à créer des lignes qui ne sont pas visibles avec la vue. Les options de vérification suivantes sont supportées :

LOCAL

Les nouvelles lignes sont seulement vérifiées avec les conditions définies directement dans la vue. Toute condition définie dans les relations sous-jacentes ne sont pas vérifiées (sauf si elles disposent elles-même de l'option CHECK OPTION).

CASCADED

Les nouvelles lignes sont vérifiées avec les conditions de la vue et de toutes les relations sous-jacentes. Si l'option CHECK OPTION est précisée, et que ni LOCAL ni CASCADED ne le sont, alors CASCADED est supposé.

L'option CHECK OPTION ne peut pas être utilisé dans les vues RECURSIVE.

Il faut noter que l'option CHECK OPTION est seulement acceptée sur les vues qui sont automatiquement modifiables, et n'ont pas de triggers INSTEAD OF ou de règles INSTEAD.

Si une vue modifiable automatiquement est définie au-dessus d'une vue de base qui dispose de triggers `INSTEAD OF`, alors l'option `LOCAL CHECK OPTION` peut être utilisé pour vérifier les conditions de la vue automatiquement modifiable mais les conditions de la vue de base comprenant des triggers `INSTEAD OF` ne seront pas vérifiées (une option de vérification en cascade ne continuera pas après vue avec trigger et toute option de vérification définie directement sur une vue automatiquement modifiable sera ignorée). Si la vue ou une des relations sous-jacentes a une règle `INSTEAD` qui cause la réécriture des commandes `INSERT` ou `UPDATE`, alors toutes les options de vérification seront ignorées dans la requête réécrite, ainsi que toutes les vérifications provenant de vues automatiquement modifiables définies au niveau haut d'une relation avec la règle `INSTEAD`.

Notes

L'instruction `DROP VIEW` est utilisée pour supprimer les vues.

Il est important de s'assurer que le nom et le type des colonnes de la vue correspondent à ce qui est souhaité. Ainsi :

```
CREATE VIEW vista AS SELECT 'Hello World';
```

est une mauvaise façon de procéder car le nom de la colonne vaudra par défaut`?column?`; de plus, le type de donnée de la colonne vaudra par défaut `text`, ce qui pourrait ne pas être ce que vous voulez. Un meilleur style pour une chaîne litérale dans le résultat d'une vue est quelque chose comme :

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

Par défaut, l'accès aux relations de base sous-jacentes est déterminée par les droits du propriétaire de la vue. Dans certains cas, ceci peut être utilisé pour fournir des accès sécurisé et restreint aux tables sous-jacentes. Néanmoins, les vues ne sont pas toutes sécurisées contre des accès inattendus ; voir Section 41.5 pour plus de détails.

Si la propriété `security_invoker` de la vue est configurée à `true`, l'accès aux relations de base sous-jacentes est déterminée par les droits de l'utilisateur exécutant la requête, plutôt que par ceux de son propriétaire. De ce fait, l'utilisateur d'une vue *security invoker* doit avoir les droits adéquats sur la vue et sur les relations de base sous-jacentes.

Si une des relations de base sous-jacentes est une vue *security invoker*, elle sera traitée comme si elle avait été accédée directement depuis la requête originale. De ce fait, une vue *security invoker* vérifiera toujours ses relations de base sous-jacentes en utilisant les droits de l'utilisateur courant, même si elle est accédée à partir d'une vue sans propriété `security_invoker`.

Si une des relations de base sous-jacentes dispose de l'activation de la sécurité niveau ligne, alors, par défaut, les politiques de sécurité niveau ligne du propriétaire de la vue sont appliquées, et l'accès à toute relation supplémentaire auquel ces politiques font références est déterminé par les droits du propriétaire de la vue. Néanmoins, si l'option `security_invoker` de la vue est configurée à `true`, alors les politiques et les droits de l'utilisateur invoqué sont utilisés à la place, comme si les relations de base avaient été référencées directement à partir de la requête utilisant la vue.

Les fonctions appelées dans la vue sont traitées de la même façon que si elles avaient été appelées directement à partir de la requête utilisant la vue. De ce fait, l'utilisateur d'une vue doit avoir les droits pour appeler toutes les fonctions utilisées par la vue. Les fonctions dans la vue sont exécutées avec les droits de l'utilisateur exécutant la requête ou le propriétaire de la fonction, suivant si les fonctions sont définies comme `SECURITY INVOKER` ou `SECURITY DEFINER`. Par exemple, appeler `CURRENT_USER` directement dans une vue renverra toujours l'utilisateur appelant, et non pas le propriétaire de la vue. Ceci n'est pas affecté par la configuration `security_invoker`, et donc une vue pour laquelle `security_invoker` est configurée à `false` n'est *pas* équivalent à une fonction `SECURITY DEFINER` et ces concepts ne doivent pas être confondus.

L'utilisateur créant et remplaçant une vue doit avoir les droits `USAGE` sur tout schéma référencé dans la requête de la vue, pour rechercher les objets référencés dans ces schémas. Néanmoins, notez que cette

recherche survient seulement quand la vue est créée ou remplacée. De ce fait, l'utilisateur de la vue a uniquement besoin du droit USAGE sur le schéma contenant la vue, pas sur les schémas référencés dans la requête de la vue, y compris pour une vue *security invoker*.

Quand CREATE OR REPLACE VIEW est utilisé sur une vue existante, seuls la règle SELECT définissant la vue, plus tout paramètre WITH (...) et ses CHECK OPTION sont modifiés. Les autres propriétés, comme les droits, le propriétaire et les règles autres que le SELECT, ne sont pas modifiées. Vous devez être le propriétaire de la vue pour la remplacer (ceci incluant aussi les membres du rôle propriétaire).

Vues modifiables

Les vues simples sont automatiquement modifiables : le système autorise l'utilisation des commandes INSERT, UPDATE et DELETE sur les vues comme sur les tables. Une vue est modifiable automatiquement si elle satisfait les conditions suivantes :

- La vue doit avoir exactement une entrée (une table ou une autre vue modifiable) dans la liste FROM.
- La définition de la vue ne doit pas contenir de clauses WITH, DISTINCT, GROUP BY, HAVING, LIMIT ou OFFSET au niveau le plus haut.
- La définition de la vue ne doit pas contenir d'opérations sur des ensembles (UNION, INTERSECT ou EXCEPT) au niveau le plus haut.
- La liste de sélection de la vue ne doit pas contenir d'agrégats, de fonctions de fenêtrage ou de fonctions renvoyant des ensembles de lignes.

Une vue à mise à jour automatique peut contenir un mélange de colonnes modifiables et non modifiables. Une colonne est modifiable si elle est une référence simple à une colonne modifiable de la relation sous-jacente. Dans le cas contraire, la colonne est en lecture seule et une erreur sera levée si une instruction INSERT ou UPDATE tente d'assigner une valeur à cette colonne.

Si la vue est modifiable automatiquement, le système convertira automatiquement toute commande INSERT, UPDATE ou DELETE sur la vue dans la commande correspondante sur la relation sous-jacente. Les requêtes INSERT qui ont une clause ON CONFLICT UPDATE sont supportées.

Si une vue modifiable automatiquement contient une condition WHERE, la condition restreint les lignes modifiables dans la relation de base par une commande UPDATE ou DELETE. Néanmoins, un UPDATE peut modifier une ligne qui ne satisfait plus la condition WHERE, et du coup qui n'est plus visible par la vue. De la même façon, une commande INSERT peut insérer des lignes dans la relation de base qui ne satisfont par la condition WHERE et qui, du coup, ne sont pas visibles via la vue (ON CONFLICT UPDATE pourrait aussi impacter une ligne non visible au travers de la vue). La clause CHECK OPTION peut être utilisée pour empêcher que les commandes INSERT et UPDATE créent de telles lignes qui ne sont pas visibles au travers de la vue.

Si une vue modifiable automatiquement est marquée avec la propriété security_barrier, alors toutes les conditions de la clause WHERE (et toutes les conditions utilisant des opérateurs marqués LEAKPROOF) seront toujours évaluées avant les conditions ajoutées par l'utilisateur de la vue. Voir Section 41.5 pour les détails complets. Notez qu'à cause de ce comportement, les lignes qui ne sont pas renvoyées (parce qu'elles ne satisfont pas les conditions de la clause WHERE de l'utilisateur) pourraient quand même se trouver bloquées. EXPLAIN peut être utilisé pour voir les conditions appliquées au niveau de la relation (pas de verrou des lignes dans ce cas) et celles qui ne le sont pas.

Une vue plus complexe qui ne satisfait par toutes les conditions ci-dessus est par défaut en lecture seule : le système ne permettra ni insertion, ni mise à jour, ni suppression sur la vue. Vous pouvez obtenir le même effet qu'une vue modifiable en créant des triggers INSTEAD OF sur la vue. Ces triggers doivent convertir l'insertion, ... tentée sur la vue par l'action appropriée sur les autres tables. Pour plus d'informations, voir CREATE TRIGGER. Une autre possibilité revient à créer des règles (voir CREATE RULE). Cependant, en pratique, les triggers sont plus simples à comprendre et à utiliser correctement.

Notez que l'utilisateur réalisant l'insertion, la mise à jour ou la suppression sur la vue doit avoir les droits correspondants sur la vue. De plus, par défaut, le propriétaire de la vue doit avoir les droits correspondants sur les relations sous-jacentes mais l'utilisateur réalisant la mise à jour n'a pas besoin de droits sur les relations sous-jacentes (voir Section 41.5). Néanmoins, si l'option `security_invoker` de la vue est configurée à `true`, l'utilisateur réalisant la mise à jour, plutôt que celui qui a créé la vue, doit avoir les droits adéquats sur les relations de base sous-jacentes.

Exemples

Créer une vue composée des comédies :

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE genre = 'Comédie';
```

Cette requête crée une vue contenant les colonnes de la table `film` au moment de la création de la vue. Bien que l'étoile (`*`) soit utilisée pour créer la vue, les colonnes ajoutées par la suite à la table `film` ne feront pas partie de la vue.

Créer une vue avec l'option `LOCAL CHECK OPTION` :

```
CREATE VIEW comedies_universelles AS
    SELECT *
    FROM comedies
    WHERE classification = 'U'
    WITH LOCAL CHECK OPTION;
```

Ceci créera une vue basée sur la vue `comedies`, ne montrant que les films pour lesquels `kind = 'Comedy'` et `classification = 'U'`. Toute tentative d'`INSERT` ou d'`UPDATE` d'une ligne dans la vue sera rejeté si la nouvelle ligne ne correspond pas à `classification = 'U'`, mais le type du film (colonne `genre`) ne sera pas vérifié.

Créer une vue avec `CASCADED CHECK OPTION` :

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

Ceci créera une vue qui vérifie les colonnes `kind` et `classification` de chaque nouvelle ligne.

Créer une vue avec un ensemble de colonnes modifiables et non modifiables :

```
CREATE VIEW comedies AS
    SELECT f.*,
           code_pays_a_nom(f.code_pays) AS pays,
           (SELECT avg(r.score)
            FROM utilisateurs_score r
            WHERE r.film_id = f.id) AS score_moyen
    FROM films f
    WHERE f.genre = 'Comedy';
```

Cette vue supportera les commandes `INSERT`, `UPDATE` et `DELETE`. Toutes les colonnes de la table `films` seront modifiables, alors que les colonnes calculées, `pays` et `score_moyen` seront en lecture seule.

Créer une vue récursive consistant en des nombres 1 à 100 :

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
    VALUES (1)
UNION ALL
    SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

Notez que, bien que le nom de la vue récursive est qualifié du schéma dans cette commande `CREATE`, sa propre référence interne n'est pas qualifiée du schéma. Ceci est dû au fait que le nom, implicitement créé, de la CTE ne peut pas être qualifié d'un schéma.

Compatibilité

Le standard SQL spécifie quelques possibilités supplémentaires pour l'instruction `CREATE VIEW` :

```
CREATE VIEW nom [ ( nom_colonne [, ...] ) ]
    AS requête
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

`CREATE OR REPLACE VIEW` est une extension PostgreSQL, tout comme le concept de vue temporaire. La clause `WITH ( ... )` est aussi une extension, comme le sont les vues avec barrière de sécurité et les vues *security invoker*.

Voir aussi

ALTER VIEW, DROP VIEW, CREATE MATERIALIZED VIEW

DEALLOCATE

DEALLOCATE — Désaffecter (libérer) une instruction préparée

Synopsis

```
DEALLOCATE [ PREPARE ] { nom | ALL }
```

Description

`DEALLOCATE` est utilisé pour désaffecter une instruction SQL préparée précédemment. Une instruction préparée qui n'est pas explicitement libérée l'est automatiquement en fin de session.

Pour plus d'informations sur les instructions préparées, voir PREPARE.

Paramètres

`PREPARE`

Mot clé ignoré.

`nom`

Le nom de l'instruction préparée à désaffecter.

`ALL`

Désaffecte toutes les instructions préparées.

Compatibilité

Le standard SQL inclut une instruction `DEALLOCATE` qui n'est utilisée que pour le SQL imbriqué.

Voir aussi

EXECUTE, PREPARE

DECLARE

DECLARE — Définir un curseur

Synopsis

```
DECLARE nom [ BINARY ] [ ASENSITIVE | INSENSITIVE ] [ [ NO ]
 SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR requête
```

Description

`DECLARE` permet à un utilisateur de créer des curseurs. Ils peuvent être utilisés pour récupérer un petit nombre de lignes à la fois à partir d'une requête plus importante. Après la création du curseur, les lignes sont récupérées en utilisant `FETCH`.

> **Note**
>
> Cette page décrit l'utilisation des curseurs au niveau de la commande SQL. Si vous voulez utiliser des curseurs dans une fonction PL/pgSQL, les règles sont différentes -- voir Section 43.7.

Paramètres

nom

Le nom du curseur à créer.

`BINARY`

Le curseur retourne les données au format binaire.

`ASENSITIVE`
`INSENSITIVE`

La sensibilité du curseur détermine si les changements de données sous le curseur, réalisés pendant la même transaction, une fois que le curseur a été déclaré, sont visibles dans le curseur. `INSENSITIVE` signifie qu'elles ne sont pas visibles, `ASENSITIVE` signifie que le comportement est dépendant de l'implémentation. Un troisième comportement, `SENSITIVE`, signifiant que les changements sont visibles dans le curseur, n'est pas disponible avec PostgreSQL. Dans PostgreSQL, tous les curseurs sont insensibles ; donc ces mots clés n'ont pas d'effet et sont seulement acceptés pour la compatibilité avec le standard SQL.

Indiquer `INSENSITIVE` avec `FOR UPDATE` ou `FOR SHARE` renvoie une erreur.

`SCROLL`
`NO SCROLL`

`SCROLL` indique une utilisation possible du curseur pour récupérer des lignes de façon non séquentielle (c'est-à-dire en remontant la liste). En fonction de la complexité du plan d'exécution de la requête, `SCROLL` peut induire des pertes de performance sur le temps d'exécution de la requête. `NO SCROLL` indique que le curseur ne peut pas être utilisé pour récupérer des lignes de

façon non séquentielle. La valeur par défaut autorise la non-séquentialité du curseur dans certains cas ; ce n'est pas la même chose que de spécifier SCROLL. Voir Notes pour les détails.

WITH HOLD
WITHOUT HOLD

WITH HOLD (NDT : persistant) indique une utilisation possible du curseur après la validation de la transaction qui l'a créé. WITHOUT HOLD (NDT : volatil) interdit l'utilisation du curseur en dehors de la transaction qui l'a créé. WITHOUT HOLD est la valeur par défaut.

requête

Une commande SELECT ou VALUES qui fournira les lignes à renvoyer par le curseur.

Les mots clés ASENSITIVE, BINARY, INSENSITIVE et SCROLL peuvent apparaître dans n'importe quel ordre.

Notes

Les curseurs normaux renvoient les données au format texte, le même que produirait un SELECT. L'option BINARY spécifie que le curseur doit renvoyer les données au format binaire. Ceci réduit les efforts de conversion pour le serveur et le client, au coût d'un effort particulier de développement pour la gestion des formats de données binaires dépendants des plateformes. Comme exemple, si une requête renvoie une valeur de un dans une colonne de type integer, vous obtiendrez une chaîne 1 avec un curseur par défaut. Avec un curseur binaire, vous obtiendrez un champ sur quatre octet contenant la représentation interne de la valeur (dans l'ordre big-endian).

Les curseurs binaires doivent être utilisés en faisant très attention. Beaucoup d'applications, incluant psql, ne sont pas préparées à gérer des curseurs binaires et s'attendent à ce que les données reviennent dans le format texte.

> **Note**
>
> Quand l'application cliente utilise le protocole des « requêtes étendues » pour exécuter la commande FETCH, le message Bind du protocole spécifie si les données sont à récupérer au format texte ou binaire. Ce choix surcharge la façon dont le curseur est défini. Le concept de curseur binaire est donc obsolète lors de l'utilisation du protocole des requêtes étendues -- tout curseur peut être traité soit en texte soit en binaire.

Si la clause WITH HOLD n'est pas précisée, le curseur créé par cette commande ne peut être utilisé qu'à l'intérieur d'une transaction. Ainsi, DECLARE sans WITH HOLD est inutile à l'extérieur d'un bloc de transaction : le curseur survivrait seulement jusqu'à la fin de l'instruction. PostgreSQL rapporte donc une erreur si cette commande est utilisée en dehors d'un bloc de transactions. On utilise BEGIN et COMMIT (ou ROLLBACK) pour définir un bloc de transaction.

Si la clause WITH HOLD est précisée, et que la transaction qui a créé le curseur est validée, ce dernier reste accessible par les transactions ultérieures de la session. Au contraire, si la transaction initiale est annulée, le curseur est supprimé. Un curseur créé avec la clause WITH HOLD est fermé soit par un appel explicite à la commande CLOSE, soit par la fin de la session. Dans l'implantation actuelle, les lignes représentées par un curseur persistant (WITH HOLD) sont copiées dans un fichier temporaire ou en mémoire afin de garantir leur disponibilité pour les transactions suivantes.

WITH HOLD n'est pas utilisable quand la requête contient déjà FOR UPDATE ou FOR SHARE.

L'option SCROLL est nécessaire à la définition de curseurs utilisés en récupération remontante (retour dans la liste des résultats, backward fetch), comme précisé par le standard SQL. Néanmoins, pour des raisons de compatibilité avec les versions antérieures, PostgreSQL autorise les récupérations remontantes sans que l'option SCROLL ne soit précisé, sous réserve que le plan d'exécution du curseur

soit suffisamment simple pour être géré sans surcharge. Toutefois, il est fortement conseillé aux développeurs d'application ne pas utiliser les récupérations remontantes avec des curseurs qui n'ont pas été créés avec l'option SCROLL. Si NO SCROLL est spécifié, les récupérations remontantes sont toujours dévalidées.

Les parcours inverses sont aussi interdits lorsque la requête inclut les clauses FOR UPDATE et FOR SHARE ; donc SCROLL peut ne pas être indiqué dans ce cas.

> **Attention**
>
> Les curseurs scrollables pourraient donner des résultats inattendues s'ils font appel à des fonctions volatiles (voir Section 38.7). Quand une ligne précédemment récupérée est de nouveau récupérée, la fonction pourrait être ré-exécutée, amenant peut-être des résultats différentes de la première exécution. Il est préférable d'indiquer NO SCROLL pour une requête impliquant des fonctions volatiles. Si ce n'est pas pratique, un contournement est de déclarer le curseur SCROLL WITH HOLD et de valider la transaction avant de lire toute ligne de ce curseur. Cela forcera la sortie entière du cuseur à être matérialisée dans un stockage temporaire, pour que les fonctions volatiles soient exécutées exactement une fois pour chaque ligne.

Si la requête du curseur inclut les clauses FOR UPDATE ou FOR SHARE, alors les lignes renvoyées sont verrouillées au moment où elles sont récupérées, de la même façon qu'une commande SELECT standard avec ces options. De plus, les lignes renvoyées seront les versions les plus à jour.

> **Attention**
>
> Il est généralement recommandé d'utiliser FOR UPDATE si le curseur doit être utilisé avec UPDATE ... WHERE CURRENT OF ou DELETE ... WHERE CURRENT OF. Utiliser FOR UPDATE empêche les autres sessions de modifier les lignes entre le moment où elles sont récupérées et celui où elles sont modifiées. Sans FOR UPDATE, une commande WHERE CURRENT OF suivante n'aura pas d'effet si la ligne a été modifiée depuis la création du curseur.
>
> Une autre raison d'utiliser FOR UPDATE est que, sans ce dernier, un appel suivant à WHERE CURRENT OF pourrait échouer si la requête curseur ne répond pas aux règles du standard SQL d'être « mise à jour simplement » (en particulier, le curseur doit référencer une seule table et ne pas utiliser de regroupement ou de tri comme ORDER BY). Les curseurs qui ne peuvent pas être mis à jour pourraient fonctionner, ou pas, suivant les détails du plan choisi ; dans le pire des cas, une application pourrait fonctionner lors des tests puis échouer en production. Si FOR UPDATE est indiqué, le curseur est garanti être modifiable.
>
> La principale raison de ne pas utiliser FOR UPDATE avec WHERE CURRENT OF est si vous avez besoin que le curseur soit déplaçable ou qu'il soit isolé des mises à jour suivantes (c'est-à-dire qu'il continue à afficher les anciennes données). Si c'est un prérequis, faites très attention aux problèmes expliqués ci-dessus.

Le standard SQL ne mentionne les curseurs que pour le SQL embarqué. PostgreSQL n'implante pas l'instruction OPEN pour les curseurs ; un curseur est considéré ouvert à sa déclaration. Néanmoins, ECPG, le préprocesseur de SQL embarqué pour PostgreSQL, supporte les conventions du standard SQL relatives aux curseurs, dont celles utilisant les instructions DECLARE et OPEN.

Vous pouvez voir tous les curseurs disponibles en exécutant une requête sur la vue système pg_cursors.

Exemples

Déclarer un curseur :

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

Voir FETCH pour plus d'exemples sur l'utilisation des curseurs.

Compatibilité

Le standard SQL n'autorise les curseurs que dans le SQL embarqué et dans les modules. PostgreSQL permet une utilisation interactive des curseurs.

Le standard SQL autorise les curseurs à mettre à jour les données d'une table. Tous les curseurs PostgreSQL sont en lecture seule.

D'après le standard SQL, les modifications faites à des curseurs insensibles par les instructions `UPDATE ... WHERE CURRENT OF` et `DELETE ... WHERE CURRENT OF` sont visibles dans ce même curseur. PostgreSQL traite ces instructions comme toutes les autres instructions de changement de données, dans le sens où elles ne sont pas visibles dans les curseurs insensibles.

Les curseurs binaires sont une extension de PostgreSQL.

Voir aussi

CLOSE, FETCH, MOVE

DELETE

DELETE — Supprimer des lignes d'une table

Synopsis

```
[ WITH [ RECURSIVE ] requête_with [, ...] ]
DELETE FROM [ ONLY ] nom_table [ * ] [ [ AS ] alias ]
    [ USING element_from ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
    [ RETURNING { * | expression_sortie [ [ AS ] nom_sortie ] }
 [, ...] ]
```

Description

`DELETE` supprime de la table spécifiée les lignes qui satisfont la clause `WHERE`. Si la clause `WHERE` est absente, toutes les lignes de la table sont supprimées. Le résultat est une table valide, mais vide.

> **Astuce**
>
> `TRUNCATE` fournit un mécanisme plus rapide de suppression de l'ensemble des lignes d'une table.

Il existe deux façons de supprimer des lignes d'une table en utilisant les informations d'autres tables de la base de données : les sous-sélections ou la spécification de tables supplémentaires dans la clause `USING`. La technique la plus appropriée dépend des circonstances.

La clause `RETURNING` optionnelle fait que `DELETE` calcule et renvoie le(s) valeur(s) basée(s) sur chaque ligne en cours de suppression. Toute expression utilisant les colonnes de la table et/ou les colonnes de toutes les tables mentionnées dans `USING` peut être calculée. La syntaxe de la liste `RETURNING` est identique à celle de la commande `SELECT`.

Il est nécessaire de posséder le droit `DELETE` sur la table pour en supprimer des lignes, et le droit `SELECT` sur toute table de la clause `USING` et sur toute table dont les valeurs sont lues dans la *`condition`*.

Paramètres

`requête_with`

La clause `WITH` vous permet de spécifier une ou plusieurs sous-requêtes qui peuvent être référencées par nom dans la requête`DELETE`. Voir Section 7.8 et SELECT pour les détails.

`nom_table`

Le nom (éventuellement qualifié du nom du schéma) de la table dans laquelle il faut supprimer des lignes. Si `ONLY` est indiqué avant le nom de la table, les lignes supprimées ne concernent que la table nommée. Si `ONLY` n'est pas indiquée, les lignes supprimées font partie de la table nommée et de ses tables filles. En option, `*` peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles doivent être inclues.

alias

Un nom de substitution pour la table cible. Quand un alias est fourni, il cache complètement le nom réel de la table. Par exemple, avec `DELETE FROM foo AS f`, le reste de l'instruction `DELETE` doit référencer la table avec `f` et non plus `foo`.

element_from

Une expression de table, qui permet de faire apparaître des colonnes d'autres tables dans la condition `WHERE`. Cela utilise la même syntaxe que la clause Clause FROM d'une instruction `SELECT` ; par exemple, un alias peut être spécifié pour un nom de table. Ne pas répéter la table cible comme un *element_from* sauf si vous souhaitez configurer une jointure avec elle-même (auquel cas elle doit apparaître avec un alias dans *element_from*).

condition

Une expression retournant une valeur de type `boolean`. Seules les lignes pour lesquelles cette expression renvoie `true` seront supprimées.

nom_curseur

Le nom du curseur à utiliser dans une condition `WHERE CURRENT OF`. La ligne à supprimer est la dernière ligne récupérée avec ce curseur. Le curseur doit être une requête sans regroupement sur la table cible du `DELETE`. Notez que `WHERE CURRENT OF` ne peut pas se voir ajouter de condition booléenne. Voir DECLARE pour plus d'informations sur l'utilisation des curseurs avec `WHERE CURRENT OF`.

expression_sortie

Une expression à calculer et renvoyée par la commande `DELETE` après chaque suppression de ligne. L'expression peut utiliser tout nom de colonne de la table nommée *nom_table* ou des tables listées dans la clause `USING`. Indiquez `*` pour que toutes les colonnes soient renvoyées.

nom_sortie

Un nom à utiliser pour une colonne renvoyée.

Sorties

En cas de succès, une commande `DELETE` renvoie une information de la forme

`DELETE` *nombre*

Le *nombre* correspond au nombre de lignes supprimées. Notez que ce nombre peut être inférieur au nombre de lignes qui satisfont la *condition* lorsque les lignes ont été supprimées via un trigger `BEFORE DELETE`. Si *nombre* vaut 0, aucune ligne n'a été supprimée par cette requête (ce qui n'est pas considéré comme une erreur).

Si la commande `DELETE` contient une clause `RETURNING`, le résultat sera similaire à celui d'une instruction `SELECT` contenant les colonnes et les valeurs définies dans la liste `RETURNING`, à partir de la liste des lignes supprimées par la commande.

Notes

PostgreSQL autorise les références à des colonnes d'autres tables dans la condition `WHERE` par la spécification des autres tables dans la clause `USING`. Par exemple, pour supprimer tous les films produits par un producteur donné :

```
DELETE FROM films USING producteurs
```

```
  WHERE id_producteur = producteurs.id AND producteurs.nom = 'foo';
```

Pour l'essentiel, une jointure est établie entre `films` et `producteurs` avec toutes les lignes jointes marquées pour suppression. Cette syntaxe n'est pas standard. Une façon plus standard de procéder consiste à utiliser une sous-selection :

```
DELETE FROM films
  WHERE id_producteur IN (SELECT id FROM producteur WHERE nom =
 'foo');
```

Dans certains cas, la jointure est plus facile à écrire ou plus rapide à exécuter que la sous-sélection.

Exemples

Supprimer tous les films qui ne sont pas des films musicaux :

```
DELETE FROM films WHERE genre <> 'Comédie musicale';
```

Effacer toutes les lignes de la table `films` :

```
DELETE FROM films;
```

Supprimer les tâches terminées tout en renvoyant le détail complet des lignes supprimées :

```
DELETE FROM taches WHERE statut = 'DONE' RETURNING *;
```

Supprimer la ligne de `taches` sur lequel est positionné le curseur `c_taches` :

```
DELETE FROM taches WHERE CURRENT OF c_taches;
```

Compatibilité

Cette commande est conforme au standard SQL, à l'exception des clauses `USING` et `RETURNING`, qui sont des extensions de PostgreSQL, comme la possibilité d'utiliser la clause `WITH` avec `DELETE`.

Voir aussi

TRUNCATE

DISCARD

DISCARD — Annuler l'état de la session

Synopsis

```
DISCARD { ALL | PLANS | SEQUENCES | TEMPORARY | TEMP }
```

Description

`DISCARD` libère les ressources internes associées avec une session de la base de données. Ces ressources sont normalement libérées à la fin de la session. Cette commande est intéressante pour réinitialiser l'état de la session partiellement ou complètement. Il existe plusieurs sous-commandes pour relâcher différents types de ressources. La variante `DISCARD ALL` comprend toutes les autres, et réinitialise aussi un état supplémentaire.

Paramètres

`PLANS`

Supprime tous les plans de requête en cache, forçant une nouvelle planification la prochaine fois que la requête préparée est utilisée.

`SEQUENCES`

Supprime du cache l'état des séquences, ceci incluant les informations `currval()`/`lastval()` ainsi que toutes les valeurs préallouées des séquences, qui n'ont pas encore été renvoyées par `nextval()`. (Voir CREATE SEQUENCE pour une description des valeurs préallouées des séquences.)

`TEMPORARY` or `TEMP`

Supprime toutes les tables temporaires créées dans la session actuelle.

`ALL`

Libère les ressources temporaires associées à cette session et réinitialise une session à son état d'origine. Actuellement, ceci a le même effet que la séquence d'instructions suivantes :

```
CLOSE ALL;
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD TEMP;
DISCARD SEQUENCES;
```

Notes

`DISCARD ALL` ne peut pas être utilisé dans un bloc de transaction.

Compatibilité

`DISCARD` est une extension PostgreSQL.

DO

DO — exécute un bloc de code anonyme

Synopsis

```
DO [ LANGUAGE nom_langage ] code
```

Description

`DO` exécute un bloc de code anonyme, autrement dit une fonction temporaire dans le langage de procédure indiqué.

Le bloc de code est traité comme le corps d'une fonction sans paramètre et renvoyant `void`. Il est analysé et exécuté une seule fois.

La clause `LANGUAGE` optionnelle est utilisable avant ou après le bloc de code.

Paramètres

code

Le code à exécuter. Il doit être spécifié comme une chaîne litérale, tout comme une fonction `CREATE FUNCTION`. L'utilisation de la syntaxe des guillemets dollar est recommandée.

nom_langage

Le nom du langage utilisé par le code. Par défaut à `plpgsql`.

Notes

Le langage de procédure utilisé doit déjà être installé dans la base de données avec l'instruction `CREATE LANGUAGE`. `plpgsql` est installé par défaut contrairement aux autres langages.

L'utilisateur doit avoir le droit `USAGE` sur le langage de procédures ou être un superutilisateur s'il ne s'agit pas d'un langage de confiance. Il s'agit des mêmes prérequis que pour la création d'une fonction dans ce langage.

Si `DO` est exécuté dans un bloc de transaction, alors le code de la procédure ne peut pas exécuter des instructions de contrôle de la transaction. Ce type d'instruction n'est autorisé que si `DO` est exécuté dans sa propre transaction.

Exemples

Donner les droits sur toutes les vues du schéma `public` au rôle `webuser` :

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM
 information_schema.tables
             WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
```

```
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) ||
 '.' || quote_ident(r.table_name) || ' TO webuser';
    END LOOP;
END$$;
```

Compatibilité

Il n'existe pas d'instruction `DO` dans le standard SQL.

Voir aussi

CREATE LANGUAGE

DROP ACCESS METHOD

DROP ACCESS METHOD — Supprimer une méthode d'accès

Synopsis

```
DROP ACCESS METHOD [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

`DROP ACCESS METHOD` supprime une méthode d'accès existante. Les superutilisateurs sont les seuls à pouvoir supprimer une méthode d'accès.

Paramètres

`IF EXISTS`

Dans le cas où la méthode d'accès n'existe pas, ce paramètre indique de renvoyer un message plutôt qu'une erreur.

name

Le nom de la méthode d'accès à supprimer.

`CASCADE`

Supprimer automatiquement les objets qui dépendent de la méthode d'accès (tels que classes d'opérateurs, familles d'opérateurs, index), ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Empêche la suppression de la méthode d'accès s'il existe des objets qui en dépendent. C'est le comportement par défaut.

Exemples

Supprimer la méthode d'accès `heptree` :

```
DROP ACCESS METHOD heptree;
```

Compatibilité

`DROP ACCESS METHOD` est une extension PostgreSQL.

Voir aussi

CREATE ACCESS METHOD

DROP AGGREGATE

DROP AGGREGATE — Supprimer une fonction d'agrégat

Synopsis

```
DROP AGGREGATE [ IF EXISTS ] nom ( signature_agrégat ) [, ...]
 [ CASCADE | RESTRICT ]

where signature_agrégat is:

* |
[ mode_arg ] [ nom_arg ] type_arg [ , ... ] |
[ [ mode_arg ] [ nom_arg ] type_arg [ , ... ] ] ORDER BY [ mode_arg
 ] [ nom_arg ] type_arg [ , ... ]
```

Description

`DROP AGGREGATE` supprime une fonction d'agrégat. Pour exécuter cette commande, l'utilisateur courant doit être le propriétaire de la fonction.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`

Le nom (éventuellement qualifié du nom de schéma) d'une fonction d'agrégat.

`mode_arg`

Le mode d'un argument : `IN` ou `VARIADIC`. Si non précisé, la valeur par défaut est `IN`.

`nom_arg`

Le nom d'un argument. Notez que `DROP AGGREGATE` ne se préoccupe pas du nom de l'argument, puisque seul le type de donnée de l'argument est nécessaire pour déterminer l'identité de la fonction d'aggrégat.

`type_arg`

Un type de données en entrée avec lequel la fonction d'agrégat opère. Pour référencer une fonction d'agrégat sans arguments, écrivez `*` à la place de la liste des spécifications d'argument. Pour référencer une fonction d'agrégat d'ensemble trié, écrivez `ORDER BY` entre les spécifications des arguments directs et des arguments agrégés.

`CASCADE`

Les objets qui dépendent de la fonction d'agrégat sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

La fonction d'agrégat n'est pas supprimée si un objet en dépend. Comportement par défaut.

Notes

Les syntaxes alternatives pour référencer des agrégats de tri d'ensemble sont décrits sur ALTER AGGREGATE.

Exemples

Supprimer la fonction d'agrégat `mamoyenne` pour le type `integer` :

```
DROP AGGREGATE mamoyenne(integer);
```

Pour supprimer la fonction d'agrégat d'ensemble hypothétique `monrang`, qui prend une liste arbitraire de colonnes pour le tri et une liste de comparaison des arguments directs :

```
DROP AGGREGATE monrang(VARIADIC "any" ORDER BY VARIADIC "any");
```

Pour supprimer plusieurs fonctions d'agrégat en une seule commande :

```
DROP AGGREGATE myavg(integer), myavg(bigint);
```

Compatibilité

Il n'existe pas d'instruction `DROP AGGREGATE` dans le standard SQL.

Voir aussi

ALTER AGGREGATE, CREATE AGGREGATE

DROP CAST

DROP CAST — Supprimer un transtypage

Synopsis

```
DROP CAST [ IF EXISTS ] (type_source AS type_cible) [ CASCADE |
 RESTRICT ]
```

Description

`DROP CAST` supprime un transtypage (conversion entre deux types de données) précédemment défini.

Seul le propriétaire du type de données source ou cible peut supprimer un transtypage. Les mêmes droits sont requis que pour la création d'un transtypage.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`type_source`

Le nom du type de données source du transtypage.

`type_cible`

Le nom du type de données cible du transtypage.

`CASCADE`
`RESTRICT`

Ces mots clés n'ont pas d'effet car il n'y aucune dépendance dans les transtypages.

Exemples

Supprimer le transtypage du type `text` en type `int` :

```
DROP CAST (text AS int);
```

Compatibilité

La commande `DROP CAST` est conforme au standard SQL.

Voir aussi

CREATE CAST

DROP COLLATION

DROP COLLATION — supprime une collation

Synopsis

```
DROP COLLATION [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

`DROP COLLATION` supprime une collation que vous avez défini auparavant. Pour pouvoir supprimer une collation, vous devez en être propriétaire.

Paramètres

`IF EXISTS`

Ne génère pas d'erreur si la collation n'existe pas. Dans ce cas, un avertissement est généré.

nom

Le nom de la collation. Le nom de la collation peut être préfixé par le schéma.

`CASCADE`

Supprime automatiquement les objets qui sont dépendants de la collation, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Refuse de supprimer la collection si un quelconque objet en dépend. C'est l'option par défaut.

Exemples

Pour supprimer la collation nommée `allemand`:

```
DROP COLLATION allemand;
```

Compatibilité

La commande `DROP COLLATION` est conforme au standard SQL , sauf pour l'option `IF EXISTS` , qui est une extension PostgreSQL.

Voir également

ALTER COLLATION, CREATE COLLATION

DROP CONVERSION

DROP CONVERSION — Supprimer une conversion

Synopsis

```
DROP CONVERSION [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

`DROP CONVERSION` supprime une conversion précédemment définie. Seul son propriétaire peut supprimer une conversion.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom

Le nom de la conversion (éventuellement qualifié du nom de schéma).

`CASCADE`
`RESTRICT`

Ces mots clés n'ont pas d'effet car il n'existe pas de dépendances sur les conversions.

Exemples

Supprimer la conversion nommée `mon_nom` :

```
DROP CONVERSION mon_nom;
```

Compatibilité

Il n'existe pas d'instruction `DROP CONVERSION` dans le standard SQL. Par contre, une instruction `DROP TRANSLATION` est disponible. Elle va de paire avec l'instruction `CREATE TRANSLATION` qui est similaire à l'instruction `CREATE CONVERSION` de PostgreSQL.

Voir aussi

ALTER CONVERSION, CREATE CONVERSION

DROP DATABASE

DROP DATABASE — Supprimer une base de données

Synopsis

```
DROP DATABASE [ IF EXISTS ] nom [ [ WITH ] ( option [, ...] ) ]

où option fait partie
de :

    FORCE
```

Description

La commande `DROP DATABASE` détruit une base de données. Elle supprime les entrées du catalogue pour la base et le répertoire contenant les données. Elle ne peut être exécutée que par le propriétaire de la base de données ou le superutilisateur. De plus, elle ne peut être exécutée si vous êtes connecté sur la base de données cible, y compris l'utilisateur effectuant la demande de suppression. (On peut se connecter à `postgres` ou à toute autre base de données pour lancer cette commande.) De plus, si quelqu'un d'autre est connecté à la base de données cible, cette commande échouera sauf si l'option `FORCE`, décrite ci-dessous, est utilisée.

`DROP DATABASE` ne peut pas être annulée. Il convient donc de l'utiliser avec précaution !

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

name

Le nom de la base de données à supprimer.

`FORCE`

Tente d'arrêter toutes les connexions existantes à la base de données cible. Il ne le fait pas si des transactions préparées, des slots actifs de réplication logique ou des souscriptions sont présentes dans la base de données cible.

Ceci arrête les connexions des *background worker* et les connexions que l'utilisateur courant a le droit d'arrêter avec la fonction `pg_terminate_backend`, décrite dans Section 9.27.2. Si des connexions sont toujours présentes, cette commande échouera.

Notes

`DROP DATABASE` ne peut pas être exécutée à l'intérieur d'un bloc de transactions.

Cette commande ne peut pas être exécutée en cas de connexion à la base de données cible. Il peut paraître plus facile d'utiliser le programme dropdb à la place, qui est un enrobage de cette commande.

Compatibilité

Il n'existe pas d'instruction `DROP DATABASE` dans le standard SQL.

Voir aussi

CREATE DATABASE, Variables d'environnement (Section 34.15)

DROP DOMAIN

DROP DOMAIN — Supprimer un domaine

Synopsis

```
DROP DOMAIN [ IF EXISTS ] nom [, ...]  [ CASCADE | RESTRICT ]
```

Description

`DROP DOMAIN` supprime un domaine. Seul le propriétaire d'un domaine peut le supprimer.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom

Le nom (éventuellement qualifié du nom du schéma) d'un domaine.

`CASCADE`

Les objets qui dépendent du domaine (les colonnes de table, par exemple) sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Le domaine n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Supprimer le domaine `boite` :

```
DROP DOMAIN boite;
```

Compatibilité

Cette commande est conforme au standard SQL, à l'exception de l'option `IF EXISTS` qui est une extension PostgreSQL.

Voir aussi

CREATE DOMAIN, ALTER DOMAIN

DROP EVENT TRIGGER

DROP EVENT TRIGGER — supprimer un trigger sur événement

Synopsis

```
DROP EVENT TRIGGER [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

`DROP EVENT TRIGGER` supprime un trigger sur événement existant. Pour exécuter cette commande, l'utilisateur courant doit être le propriétaire du trigger sur événement.

Paramètres

`IF EXISTS`

Ne renvoie pas d'erreur si le trigger sur événement n'existe pas. Un message d'avertissement est renvoyé dans ce cas.

name

Le nom d'un trigger sur événement à supprimer.

`CASCADE`

Supprime automatiquement les objets qui dépendent de ce trigger, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Refuse de supprimer le trigger si des objets dependent de lui. C'est le comportement par défaut.

Exemples

Supprimer le trigger `balance` :

```
DROP EVENT TRIGGER balance;
```

Compatibilité

Il n'existe pas de commande `DROP EVENT TRIGGER` dans le standard SQL.

Voir aussi

CREATE EVENT TRIGGER, ALTER EVENT TRIGGER

DROP EXTENSION

DROP EXTENSION — Supprime une extension

Synopsis

```
DROP EXTENSION [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP EXTENSION` supprime les extensions de la base de données. La suppression d'une extension entraine la suppression des objets membres de l'extension et des autres routines dépendantes explicitement (voir ALTER ROUTINE, pour les dépendances des extensions).

Vous devez être propriétaire de l'extension pour utiliser `DROP EXTENSION`.

Paramètres

`IF EXISTS`

Permet de ne pas retourner d'erreur si l'extension n'existe pas. Une simple notice est alors rapportée.

nom

Le nom d'une extension préalablement installée.

`CASCADE`

Supprime automatiquement les objets dont dépend cette extension, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Cette option empêche les extensions indiquées d'être supprimées si d'autres objets en dépendent, en dehors de ces extensions, de leurs membres, et des routines qui en dépendent explicitement. C'est le fonctionnement par défaut.

Exemples

Pour supprimer l'extension `hstore` de la base de données en cours:

```
DROP EXTENSION hstore;
```

Cette commande va échouer si parmi les objets de `hstore` certains sont en cours d'utilisation sur la base de données. Par exemple, si des tables ont des colonnes du type `hstore`. Dans ce cas de figure, ajoutez l'option cascade `CASCADE` pour forcer la suppression de ces objets.

Compatibilité

`DROP EXTENSION` est une extension PostgreSQL.

Voir aussi

CREATE EXTENSION, ALTER EXTENSION

DROP FOREIGN DATA WRAPPER

DROP FOREIGN DATA WRAPPER — Supprimer un wrapper de données distantes

Synopsis

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] nom [, ...] [ CASCADE |
 RESTRICT ]
```

Description

`DROP FOREIGN DATA WRAPPER` supprime un wrapper de données distantes existant. Pour exécuter cette commande, l'utilisateur courant doit être le propriétaire du wrapper de données distantes.

Paramètres

`IF EXISTS`

Ne génère pas d'erreur si le wrapper de données distantes n'existe pas. Un avertissement est émis dans ce cas.

nom

Le nom d'un wrapper de données distantes existant.

`CASCADE`

Supprime automatiquement les objets dépendant du wrapper de données distantes (tels que les serveurs), ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Refuse de supprimer le wrapper de données distantes si un objet dépend de celui-ci. C'est le cas par défaut.

Exemples

Supprimer le wrapper de données distantes `dbi` :

```
DROP FOREIGN DATA WRAPPER dbi;
```

Compatibilité

`DROP FOREIGN DATA WRAPPER` est conforme à la norme ISO/IEC 9075-9 (SQL/MED). La clause `IF EXISTS` est une extension PostgreSQL .

Voir aussi

CREATE FOREIGN DATA WRAPPER, ALTER FOREIGN DATA WRAPPER

DROP FOREIGN TABLE

DROP FOREIGN TABLE — Supprime une table distante

Synopsis

```
DROP FOREIGN TABLE [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP FOREIGN TABLE` supprime une table distante.

Vous devez être propriétaire de la table distante pour utiliser `DROP FOREIGN TABLE`.

Paramètres

`IF EXISTS`

Permet de ne pas retourner d'erreur si la table distante n'existe pas. Une simple notice est alors rapportée.

nom

Le nom de la table distante à supprimer. Il est aussi possible de spécifier le schéma qui contient cette table.

`CASCADE`

Supprime automatiquement les objets qui dépendent de cette table distante (comme les vues par exemple), ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Permet de spécifier que la table distante ne sera pas supprimée si des objets en dépendent. Il s'agit du comportement par défaut.

Exemples

Pour supprimer deux tables distantes, `films` et `distributeurs`:

```
DROP FOREIGN TABLE films, distributeurs;
```

Cette commande va échouer s'il existe des objets qui dépendent de `films` ou `distributeurs`. Par exemple, si des contraintes sont liées à des colonnes de `films`. Dans ce cas de figure, ajoutez l'option cascade `CASCADE` pour forcer la suppression de ces objets.

Compatibilité

Cette commande est conforme avec le standard ISO/IEC 9075-9 (SQL/MED), aux exceptions prêtes que ce standard n'accepte la suppression que d'une table distante par commande, et de l'option `IF EXISTS`, qui est une spécificité de PostgreSQL.

Voir aussi

ALTER FOREIGN TABLE, CREATE FOREIGN TABLE

DROP FUNCTION

DROP FUNCTION — Supprimer une fonction

Synopsis

```
DROP FUNCTION [ IF EXISTS ] nom [ ( [ [ modearg ] [ nomarg
 ] typearg [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

Description

`DROP FUNCTION` supprime la définition d'une fonction. Seul le propriétaire de la fonction peut exécuter cette commande. Les types d'argument de la fonction doivent être précisés car plusieurs fonctions peuvent exister avec le même nom et des listes différentes d'arguments.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom

Le nom (éventuellement qualifié du nom du schéma) de la fonction. Si aucune liste d'argument n'est spécifiée, le nom doit être unique dans son schéma.

modearg

Le mode d'un argument : `IN`, `OUT`, `INOUT` ou `VARIADIC`. Sans précision, la valeur par défaut est `IN`. `DROP FUNCTION` ne s'intéresse pas aux arguments `OUT` car seuls ceux en entrée déterminent l'identité de la fonction. Il est ainsi suffisant de lister les arguments `IN`, `INOUT` et `VARIADIC`.

nomarg

Le nom d'un argument. `DROP FUNCTION` ne tient pas compte des noms des arguments car seuls les types de données sont nécessaires pour déterminer l'identité de la fonction.

typearg

Le(s) type(s) de données des arguments de la fonction (éventuellement qualifié(s) du nom du schéma).

`CASCADE`

Les objets qui dépendent de la fonction (opérateurs ou triggers) sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

La fonction n'est pas supprimée si un objet en dépend. Comportement par défaut.

Exemples

Supprimer la fonction de calcul d'une racine carrée :

```
DROP FUNCTION sqrt(integer);
```

Supprimer plusieurs fonctions en une commande :

```
DROP FUNCTION sqrt(integer), sqrt(bigint);
```

Si le nom de fonction est unique dans son schéma, il peut être utilisé sans liste d'argument :

```
DROP FUNCTION update_employee_salaries;
```

Veuillez noter que c'est différent de :

```
DROP FUNCTION update_employee_salaries();
```

Qui se réfère à une fonction avec zéro argument, alors que la première variante peut se référer à une fonction ayant n'importe quel nombre d'arguments, y compris zéro, du moment que le nom est unique.

Compatibilité

Cette commande est conforme avec le standard SQL, avec ces extensions PostgreSQL :

- Le standard n'autorise qu'une seule fonction à être supprimée par commande.
- L'option `IF EXISTS`
- La possibilité de spécifier les modes et noms d'argument.

Voir aussi

CREATE FUNCTION, ALTER FUNCTION

DROP GROUP

DROP GROUP — Supprimer un rôle de base de données

Synopsis

```
DROP GROUP [ IF EXISTS ] nom [, ... ]
```

Description

`DROP GROUP` est désormais un alias de `DROP ROLE`.

Compatibilité

Il n'existe pas d'instruction `DROP GROUP` dans le standard SQL.

Voir aussi

DROP ROLE

DROP INDEX

DROP INDEX — Supprimer un index

Synopsis

```
DROP INDEX [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] nom [, ...] [ CASCADE |
 RESTRICT ]
```

Description

`DROP INDEX` supprime un index. Seul le propriétaire de l'index peut exécuter cette commande.

Paramètres

`CONCURRENTLY`

Supprime l'index sans verrouiller les lectures et les modifications (insertions, modifications, suppressions) sur la table de l'index. Un `DROP INDEX` standard acquiert un verrou de type `ACCESS EXCLUSIVE` sur la table, bloquant tous les autres accès jusqu'à ce que la suppression de l'index soit terminée. Avec cette option, la commande attends que toute transaction en conflit soit terminée.

Cette option pose quelques soucis. Un seul index peut être indiqué, et l'option `CASCADE` n'est pas autorisée. (Du coup, un index qui renforce une contrainte `UNIQUE` ou `PRIMARY KEY` ne peut pas être supprimée ainsi.) De plus, les commandes `DROP INDEX` standards sont exécutées dans un bloc de transaction, mais `DROP INDEX CONCURRENTLY` ne le peut pas. Enfin, les index sur des tables partitionnées ne peuvent pas être supprimées en utilisant cette option.

Pour les tables temporaires, `DROP INDEX` est toujours non concurrent car aucune autre session n'y a accès, et la suppression d'index non concurrent est moins coûteuse.

`IF EXISTS`

Ne pas renvoyer d'erreur si l'index n'existe pas. Un message d'information est envoyé dans ce cas.

nom

Le nom (éventuellement qualifié du nom du schéma) de l'index à supprimer.

`CASCADE`

Les objets qui dépendent de l'index sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

L'index n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Supprimer l'index `title_idx` :

```
DROP INDEX title_idx;
```

Compatibilité

`DROP INDEX` est une extension PostgreSQL. Il n'est pas fait mention des index dans le standard SQL.

Voir aussi

CREATE INDEX

DROP LANGUAGE

DROP LANGUAGE — Supprimer un langage procédural

Synopsis

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] nom [ CASCADE |
 RESTRICT ]
```

Description

`DROP LANGUAGE` supprime la définition d'un langage procédural enregistré précédemment. Vous devez être un superutilisateur ou le propriétaire du langage pour utiliser `DROP LANGUAGE`.

> **Note**
>
> À partir de PostgreSQL 9.1, la plupart des langages procéduraux sont devenus des « extensions » et doivent du coup être supprimés avec la commande `DROP EXTENSION`, et non pas avec `DROP LANGUAGE`.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si le langage n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom

Le nom du langage procédural à supprimer.

`CASCADE`

Les objets qui dépendent du langage (fonctions, par exemple) sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Le langage n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Supprimer le langage procédural `plexemple` :

```
DROP LANGUAGE plexemple;
```

Compatibilité

Il n'existe pas d'instruction `DROP LANGUAGE` dans le standard SQL.

Voir aussi

ALTER LANGUAGE, CREATE LANGUAGE

DROP MATERIALIZED VIEW

DROP MATERIALIZED VIEW — supprimer une vue matérialisée

Synopsis

```
DROP MATERIALIZED VIEW [ IF EXISTS ] nom [, ...] [ CASCADE |
 RESTRICT ]
```

Description

`DROP MATERIALIZED VIEW` supprime une vue matérialisée existante. Pour exécuter cette commande, vous devez être le propriétaire de la vue matérialisée.

Paramètres

`IF EXISTS`

Ne renvoie pas d'erreur si la vue matérialisée n'existe pas. Un message d'avertissement est renvoyé dans ce cas.

nom

Le nom de la vue matérialisée (potentiellement qualifié du schéma) à supprimer.

`CASCADE`

Supprime automatiquement les objets dépendant de la vue matérialisée (comme d'autres vues matérialisées ou des vues standards), ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Refuse de supprimer la vue matérialisée si des objets dependent de lui. C'est le comportement par défaut.

Exemples

Cette commande supprimera la vue matérialisée appelée `resume_commandes` :

```
DROP MATERIALIZED VIEW resume_commandes;
```

Compatibilité

`DROP MATERIALIZED VIEW` est une extension PostgreSQL.

Voir aussi

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

DROP OPERATOR

DROP OPERATOR — Supprimer un opérateur

Synopsis

```
DROP OPERATOR [ IF EXISTS ] nom ( { type_gauche |
 NONE } , type_droit ) [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP OPERATOR` supprime un opérateur. Seul le propriétaire de l'opérateur peut exécuter cette commande.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`

Le nom de l'opérateur (éventuellement qualifié du nom du schéma) à supprimer.

`type_gauche`

Le type de données de l'opérande gauche de l'opérateur ; `NONE` est utilisé si l'opérateur n'en a pas.

`type_droit`

Le type de données de l'opérande droit de l'opérateur.

`CASCADE`

Les objets qui dépendent de l'opérateur sont automatiquement supprimés (tels que les vues les utilisant), ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

L'opérateur n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Supprimer l'opérateur puissance `a^b` sur le type `integer` :

```
DROP OPERATOR ^ (integer, integer);
```

Supprimer l'opérateur de complément binaire `~b` sur le type `bit` :

```
DROP OPERATOR ~ (none, bit);
```

Supprimer de multiples opérateurs en une commande :

```
DROP OPERATOR ~ (none, bit), ^ (integer, integer);
```

Compatibilité

Il n'existe pas d'instruction `DROP OPERATOR` dans le standard SQL.

Voir aussi

CREATE OPERATOR, ALTER OPERATOR

DROP OPERATOR CLASS

DROP OPERATOR CLASS — Supprimer une classe d'opérateurs

Synopsis

```
DROP OPERATOR CLASS [ IF EXISTS ] nom USING méthode_index [ CASCADE
 | RESTRICT ]
```

Description

`DROP OPERATOR CLASS` supprime une classe d'opérateurs. Seul le propriétaire de la classe peut la supprimer.

`DROP OPERATOR CLASS` ne supprime aucun des opérateurs et aucune des fonctions référencés par la classe. Si un index dépend de la classe d'opérateurs, vous devez indiquer `CASCADE` pour que la suppression se fasse réellement.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`

Le nom (éventuellement qualifié du nom du schéma) d'une classe d'opérateurs.

`méthode_index`

Le nom de la méthode d'accès aux index pour laquelle l'opérateur est défini.

`CASCADE`

Les objets qui dépendent de cette classe sont automatiquement supprimés (tels que les index), ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

La classe d'opérateurs n'est pas supprimée si un objet en dépend. Comportement par défaut.

Notes

`DROP OPERATOR CLASS` ne supprimera pas la famille d'opérateur contenant la classe, même si la famille en devient vide (en particulier, dans le cas où la famille a été implicitement créée par `CREATE OPERATOR CLASS`). Avoir une famille d'opérateur vide est sans risque. Pour plus de clareté, il est préférable de supprimer la famille avec `DROP OPERATOR FAMILY` ; ou encore mieux, utilisez `DROP OPERATOR FAMILY` dès le début.

Exemples

Supprimer la classe d'opérateurs `widget_ops` des index de type arbre-balancé (B-tree) :

```
DROP OPERATOR CLASS widget_ops USING btree;
```

La commande échoue si un index utilise la classe d'opérateur. `CASCADE` permet de supprimer ces index simultanément.

Compatibilité

Il n'existe pas d'instruction `DROP OPERATOR CLASS` dans le standard SQL.

Voir aussi

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR FAMILY

DROP OPERATOR FAMILY

DROP OPERATOR FAMILY — Supprimer une famille d'opérateur

Synopsis

```
DROP OPERATOR FAMILY [ IF EXISTS ] nom USING methode_indexage
 [ CASCADE | RESTRICT ]
```

Description

`DROP OPERATOR FAMILY` supprime une famille d'opérateur existante. Pour exécuter cette commande, vous devez être le propriétaire de la famille d'opérateur.

`DROP OPERATOR FAMILY` inclut la suppression de toutes classes d'opérateurs contenues dans la famille, mais elle ne supprime pas les opérateurs et fonctions référencées par la famille. Si des index dépendent des classes d'opérateurs de la famille, vous devez ajouter `CASCADE` pour que la suppression réussisse.

Paramètres

`IF EXISTS`

Ne renvoie pas une erreur si la famille d'opérateur n'existe pas. Un message de niveau « NOTICE » est enregistré dans ce cas.

`nom`

Le nom de la famille d'opérateur (quelque fois qualifié du schéma).

`methode_indexage`

Le nom de la méthode d'accès à l'index associée à la famille d'opérateur.

`CASCADE`

Supprime automatiquement les objets dépendant de cette famille d'opérateur, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Refuse la suppression de la famille d'opérateur si des objets en dépendent. C'est la valeur par défaut.

Exemples

Supprimer la famille d'opérateur B-tree `float_ops` :

```
DROP OPERATOR FAMILY float_ops USING btree;
```

Cette commande échouera car il existe des index qui utilisent les classes d'opérateur de cette famille. Ajoutez `CASCADE` pour supprimer les index avec la famille d'opérateurs.

Compatibilité

Il n'existe pas d'instruction `DROP OPERATOR FAMILY` dans le standard SQL.

Voir aussi

ALTER OPERATOR FAMILY, CREATE OPERATOR FAMILY, ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR CLASS

DROP OWNED

DROP OWNED — Supprimer les objets de la base possédés par un rôle

Synopsis

```
DROP OWNED BY { nom | CURRENT_ROLE | CURRENT_USER | SESSION_USER }
 [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP OWNED` supprime tous les objets de la base qui ont pour propriétaire un des rôles spécifiés. Tout droit donné à un des rôles sur ces objets ainsi qu'aux objets partagés (bases de données, tablespaces, paramètres de configuration) sera aussi supprimé.

Paramètres

nom

Le nom d'un rôle dont les objets seront supprimés et dont les droits seront révoqués.

`CASCADE`

Supprime automatiquement les objets qui dépendent des objets affectés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Refuse de supprimer les objets possédés par un rôle si un autre objet de la base dépend de ces objets. C'est la valeur par défaut.

Notes

`DROP OWNED` est souvent utilisé pour préparer la suppression d'un ou plusieurs rôles. Comme `DROP OWNED` affecte seulement les objets de la base en cours, il est généralement nécessaire d'exécuter cette commande dans chaque base contenant des objets appartenant au rôle à supprimer.

Utiliser l'option `CASCADE` pourrait demander la suppression d'objets appartenant à d'autres utilisateurs.

La commande `REASSIGN OWNED` est une alternative qui ré-affecte la propriété de tous les objets de la base possédés par un ou plusieurs rôles. Néanmoins, `REASSIGN OWNED` ne gère pas les droits des autre objets.

Les bases de données et les tablespaces appartenant au(x) rôle(s) ne seront pas supprimés.

Voir Section 22.4 pour plus d'informations.

Compatibilité

La commande `DROP OWNED` est une extension PostgreSQL.

Voir aussi

REASSIGN OWNED, DROP ROLE

DROP POLICY

DROP POLICY — supprimer une politique de sécurité définie pour une table

Synopsis

```
DROP POLICY [ IF EXISTS ] nom ON nom_table [ CASCADE | RESTRICT ]
```

Description

`DROP POLICY` supprime la politique de sécurité de la table spécifiée. Notez bien, si la dernière politique est supprimée pour une table et que le niveau de sécurité est toujours activé, alors la politique par défaut (tout empêcher) est appliquée : plus aucune ligne n'est accessible ou modifiable. La commande `ALTER TABLE ... DISABLE ROW LEVEL SECURITY` peut être utilisée pour désactiver la politique de sécurité pour une table, indépendamment du fait que des politiques existent ou pas pour cette table.

Paramètres

`IF EXISTS`

Permet de ne pas générer d'erreur si la politique n'existe pas alors que l'on tente de la supprimer. Une notification est simplement renvoyée dans ce cas.

`nom`

Nom de la politique à supprimer.

`nom_table`

Nom de la table (éventuellement qualifiée par le schéma) de la table pour laquelle la politique est définie.

`CASCADE`
`RESTRICT`

Ces mots clés n'ont pas d'effet car il n'y a pas de dépendances sur les politiques de sécurité.

Exemples

Suppression d'une politique nommée `p1` d'une table nommée `ma_table` :

```
DROP POLICY p1 ON ma_table;
```

Compatibilité

`DROP POLICY` est une extension PostgreSQL.

Voir aussi

CREATE POLICY, ALTER POLICY

DROP PROCEDURE

DROP PROCEDURE — supprimer une procédure

Synopsis

```
DROP PROCEDURE [ IF EXISTS ] nom [ ( [ [ mode_argument ]
 [ nom_argument ] type_argument [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

Description

`DROP PROCEDURE` supprime la définition d'une procédure existante ou de plusieurs. Pour exécuter cette commande, l'utilisateur doit être le propriétaire des procédures. Les types des arguments des procédures doivent être spécifiés, car plusieurs procédures différentes peuvent coexister avec le même nom et des listes d'arguments différentes.

Paramètres

`IF EXISTS`

Ne génère pas d'erreur si la procédure n'existe pas. Une notification est fournie dans ce cas.

`nom`

Le nom d'une procédure existante (éventuellement qualifié par le schéma).

`mode_argument`

Le mode d'un argument : `IN`, `OUT`, `INOUT` ou `VARIADIC`. Si non précisé, le défaut est `IN` (mais voir ci-dessous).

`nom_arg`

Le nom d'un argument. Notez que `DROP PROCEDURE` ne fait pas vraiment attention aux noms des arguments, puisqu'il n'a besoin que des types des arguments pour déterminer la procédure.

`type_argument`

Les types de données des arguments de la procédure (éventuellement qualifiés par le schéma), s'il y en a. Voir ci-dessous pour les détails.

`CASCADE`

Supprime automatiquement les objets qui dépendent de la procédure, puis à leur tour tous les objets qui dépendent de ces objets. (voir Section 5.14).

`RESTRICT`

Refuse de supprimer une procédure si un objet en dépend. C'est le comportement par défaut.

Notes

S'il n'existe qu'une seule procédure de ce nom, la liste d'arguments est omise. Omettez aussi les parenthèses dans ce cas.

Dans PostgreSQL, il est suffisant de lister les arguments enentrée(en incluant arguments `INOUT`) parce qu'il n'est pas autorisé que deux routines qui partagent le même nom aient la même liste d'arguments en entrée. De plus, la commande `DROP` ne vérifiera pas que vous avez écrit les arguments `OUT` correctement ; donc tout argument explicitement marqué `OUT` n'est que du bruit. Les indiquer est recommandé par cohérence avec la commande `CREATE`.

Par compatibilité avec le standard SQL, il est aussi autorisé d'écrire tous les types de données des arguments (en incluant les arguments `OUT`) sans marqueurs *`argmode`*. Quand c'est fait, les types des arguments `OUT` de la procédure *seront* vérifiés avec lacommande. Cette provision crée une ambiguité, dans le fait que, quand la liste d'arguments ne contient aucun marqueurs *`argmode`*, il n'est pas clair de savoir quel règle suivre. La commande `DROP` tentera une recherche dans les deux sens et renverra une erreur si deux procédures différentes sont trouvées. Pour éviter le risque d'une telle ambiguité, il est recommandé d'écrire des marqueurs `IN` explicitement plutôt que de les laisser par défaut, forçant l'utilisation de l'interprétation traditionnelle de PostgreSQL.

Les règles de recherche sont aussi utilisées par d'autres commandes qui agissent sur les procédures existantes, par exemple `ALTER PROCEDURE` et `COMMENT ON PROCEDURE`.

Exemples

S'il n'existe qu'une seule procédure `do_db_maintenance`, cette commande est suffisante pour la supprimer :

```
DROP PROCEDURE do_db_maintenance;
```

Étant donné cette définition de procédure :

```
CREATE PROCEDURE do_db_maintenance(IN target_schema text, OUT
 results text) ...
```

n'importe laquelle de ces commandes pourrait la supprimer :

```
DROP PROCEDURE do_db_maintenance();
DROP PROCEDURE do_db_maintenance(IN target_schema text, OUT results
 text);
DROP PROCEDURE do_db_maintenance(IN text, OUT text);
DROP PROCEDURE do_db_maintenance(IN text);
DROP PROCEDURE do_db_maintenance(text);
DROP PROCEDURE do_db_maintenance(text, text);  -- potentially
 ambiguous
```

Néanmoins, le dernier exemple serait ambigu s'il existait aussi, disons :

```
CREATE PROCEDURE do_db_maintenance(IN target_schema text, IN
 options text) ...
```

Compatibilité

Cette commande se conforme au standard SQL, avec les extensions PostgreSQL suivantes :

- Le standard n'autorise qu'une seule procédure supprimée par commande.
- L'option `IF EXISTS` est une extension.
- La capacité de spécifier les modes et noms des arguments est une extension, et les règles de recherche diffèrent lorsque les modes sont fournis.

Voir aussi

CREATE PROCEDURE, ALTER PROCEDURE, DROP FUNCTION, DROP ROUTINE

DROP PUBLICATION

DROP PUBLICATION — supprime une: publication

Synopsis

```
DROP PUBLICATION [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP PUBLICATION` supprime une publication existante de la base.

Une publication ne peut être supprimée que par son propriétaire ou un superutilisateur.

Paramètres

`IF EXISTS`

Ne remonte pas d'erreur si la publication n'existe pas. Une note est affichée dans ce cas.

nom

Le nom d'une publication existante.

`CASCADE`
`RESTRICT`

Ces mots clés n'ont aucune effet, puisqu'il n'y a pas de dépendances sur les publications.

Exemples

Supprime une publication :

```
DROP PUBLICATION mypublication;
```

Compatibilité

`DROP PUBLICATION` est une extension PostgreSQL.

Voir aussi

CREATE PUBLICATION, ALTER PUBLICATION

DROP ROLE

DROP ROLE — Supprimer un rôle de base de données

Synopsis

```
DROP ROLE [ IF EXISTS ] nom [, ...]
```

Description

`DROP ROLE` supprime le(s) rôle(s) spécifié(s). Seul un superutilisateur peut supprimer un rôle de superutilisateur. Le droit `CREATEROLE` est nécessaire pour supprimer les autres rôles.

Un rôle ne peut pas être supprimé s'il est toujours référencé dans une base de données du groupe. Dans ce cas, toute tentative aboutit à l'affichage d'une erreur. Avant de supprimer un rôle, il est nécessaire de supprimer au préalable tous les objets qu'il possède (ou de modifier leur appartenance) et de supprimer tous les droits définis par ce rôle sur d'autres objets. Les commandes `REASSIGN OWNED` et `DROP OWNED` peuvent être utiles pour cela. Voir Section 22.4 pour plus de discussions sur ce sujet.

Néanmoins, il n'est pas nécessaire de supprimer toutes les appartenances de rôle impliquant ce rôle ; `DROP ROLE` supprime automatiquement toute appartenance du rôle cible dans les autres rôles et des autres rôles dans le rôle cible. Les autres rôles ne sont pas supprimés ou affectés.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom

Le nom du rôle à supprimer.

Notes

PostgreSQL inclut un programme dropuser qui a la même fonctionnalité que cette commande (en fait, il appelle cette commande) mais qui est lancé à partir du shell.

Exemples

Supprimer un rôle :

```
DROP ROLE jonathan;
```

Compatibilité

Le standard SQL définit `DROP ROLE` mais il ne permet la suppression que d'un seul rôle à la fois et il spécifie d'autres droits obligatoires que ceux utilisés par PostgreSQL.

Voir aussi

CREATE ROLE, ALTER ROLE, SET ROLE

DROP ROUTINE

DROP ROUTINE — Supprimer une routine

Synopsis

```
DROP ROUTINE [ IF EXISTS ] nom [ ( [ [ mode_arg ] [ nom_arg
 ] type_arg [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

Description

`DROP ROUTINE` supprime la définition d'une routine existante ou de plusieurs. Le terme « routine » inclut les fonctions d'agrégat, les fonctions normales et les procédures. Voir DROP AGGREGATE, DROP FUNCTION, et DROP PROCEDURE pour la description des paramètres, plus d'exemples, et de détails.

Notes

Les règles de recherche utilisées par `DROP ROUTINE` sont fondamentalement les mêmes que pour `DROP PROCEDURE` ; en particulier, `DROP ROUTINE` considère lui-aussi une liste d'arguments qui n'a pas de marqueurs *`argmode`* comme utilisant probablement la définition du standard SQL pour laquelle des arguments `OUT` sont inclus dans la liste. (`DROP AGGREGATE` et `DROP FUNCTION` ne le fait pas.)

Dans certains cas où le même nom est partagé par des routines de type différent, il est possible que `DROP ROUTINE` échoue avec une erreur d'ambiguïté quand une commande plus spécifique (`DROP FUNCTION`, etc.) fonctionnerait. Indiquer précisément la liste des types d'arguments peut aussi résoudre de tels problèmes.

Ces règles de recherche sont aussi utilisées par d'autres commandes qui agissent sur les routines existantes, telles que `ALTER ROUTINE` et `COMMENT ON ROUTINE`.

Exemples

Pour supprimer la routine `foo` pour le type `integer` :

```
DROP ROUTINE foo(integer);
```

Cette commande fonctionnera indépendamment du type de `foo` (fonction d'agrégat, fonction, procédure).

Compatibilité

Cette commande se conforme au standard SQL, avec ces extensions PostgreSQL :

- Le standard autorise aussi la suppression d'une routine par commande.
- L'option `IF EXISTS` est une extension.
- La possibilité de spécifier les modes et noms des arguments est une extension, et les règles de recherche diffèrent quand les modes sont indiqués.

- Les fonctions d'agrégat définies par les utilisateurs sont une extension.

Voir aussi

DROP AGGREGATE, DROP FUNCTION, DROP PROCEDURE, ALTER ROUTINE

Notez qu'il n'existe pas de commande `CREATE ROUTINE`.

DROP RULE

DROP RULE — Supprimer une règle de réécriture

Synopsis

```
DROP RULE [ IF EXISTS ] nom ON nom_table [ CASCADE | RESTRICT ]
```

Description

`DROP RULE` supprime une règle de réécriture.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`

Le nom de la règle à supprimer.

`nom_table`

Le nom (éventuellement qualifié du nom du schéma) de la table ou vue sur laquelle s'applique la règle.

`CASCADE`

Les objets qui dépendent de la règle sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

La règle n'est pas supprimée si un objet en dépend. Comportement par défaut.

Exemples

Suppression de la règle de réécriture `nouvellerègle` :

```
DROP RULE nouvellerègle ON matable;
```

Compatibilité

`DROP RULE` est une extension du langage par PostgreSQL comme tout le système de réécriture des requêtes.

Voir aussi

CREATE RULE, ALTER RULE

DROP SCHEMA

DROP SCHEMA — Supprimer un schéma

Synopsis

```
DROP SCHEMA [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP SCHEMA` supprime des schémas de la base de données.

Un schéma ne peut être supprimé que par son propriétaire ou par un superutilisateur. Son propriétaire peut supprimer un schéma et tous les objets qu'il contient quand bien même il ne possède pas tous les objets contenus dans ce schéma.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom

Le nom du schéma.

`CASCADE`

Les objets (tables, fonctions...) contenus dans le schéma sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Le schéma n'est pas supprimé s'il contient des objets. Comportement par défaut.

Notes

Utiliser l'option `CASCADE` pourrait causer la suppression d'objets dans d'autres schémas que celui indiqué.

Exemples

Supprimer le schéma `mes_affaires` et son contenu :

```
DROP SCHEMA mes_affaires CASCADE;
```

Compatibilité

`DROP SCHEMA` est totalement compatible avec le standard SQL. Le standard n'autorise cependant pas la suppression de plusieurs schémas en une seule commande. L'option `IF EXISTS` est aussi une extension de PostgreSQL.

Voir aussi

ALTER SCHEMA, CREATE SCHEMA

DROP SEQUENCE

DROP SEQUENCE — Supprimer une séquence

Synopsis

```
DROP SEQUENCE [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP SEQUENCE` permet de supprimer les générateurs de nombres séquentiels. Une séquence peut seulement être supprimée par son propriétaire ou par un superutilisateur.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom

Le nom de la séquence (éventuellement qualifié du nom du schéma).

`CASCADE`

Les objets qui dépendent de la séquence sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

La séquence n'est pas supprimée si un objet en dépend. Comportement par défaut.

Exemples

Supprimer la séquence `serie` :

```
DROP SEQUENCE serie;
```

Compatibilité

`DROP SEQUENCE` est conforme au standard SQL. Cependant, le standard n'autorise pas la suppression de plusieurs séquences en une seule commande. De plus, l'option `IF EXISTS` est une extension de PostgreSQL.

Voir aussi

CREATE SEQUENCE, ALTER SEQUENCE

DROP SERVER

DROP SERVER — Supprimer un descripteur de serveur distant

Synopsis

```
DROP SERVER [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP SERVER` supprime un descripteur de serveur distant existant. Pour exécuter cette commande, l'utilisateur courant doit être le propriétaire du serveur.

Paramètres

`IF EXISTS`

Ne génère pas d'erreur si le serveur n'existe pas. Un avertissement est émis dans ce cas.

nom

Nom d'un serveur existant.

`CASCADE`

Supprime automatiquement les objets dépendant du serveur (tels que les correspondances d'utilisateur), ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Refuse de supprimer le serveur si des objets en dépendent. C'est le cas par défaut.

Exemples

Supprimer un serveur `truc` s'il existe :

```
DROP SERVER IF EXISTS truc;
```

Compatibilité

`DROP SERVER` est conforme à la norme ISO/IEC 9075-9 (SQL/MED). La clause `IF EXISTS` est une extension PostgreSQL .

Voir aussi

CREATE SERVER, ALTER SERVER

DROP STATISTICS

DROP STATISTICS — supprime une statistique étendue

Synopsis

```
DROP STATISTICS [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP STATISTICS` supprime le ou les objets statistiques étendues d'une base. Seul le propriétaire de l'objet statistiques, le propriétaire du schéma ou un superutilisateur pour supprimer un objet statistique.

Paramètres

`IF EXISTS`

Ne remonte pas d'erreur si l'objet statistiques n'existe pas. Une note est affichée dans ce cas.

nom

Le nom (éventuellement qualifié du nom du schéma) de l'objet statistiques à supprimer.

`CASCADE`
`RESTRICT`

Ces mots clés n'ont pas d'effet car il n'existe pas de dépendances pour les statistiques.

Exemples

Pour supprimer deux objets statistiques dans des schémas différents, sans échouer s'ils n'existent pas :

```
DROP STATISTICS IF EXISTS
    accounting.users_uid_creation,
    public.grants_user_role;
```

Compatibilité

Il n'y a pas de commande `DROP STATISTICS` dans le standard SQL.

Voir aussi

ALTER STATISTICS, CREATE STATISTICS

DROP SUBSCRIPTION

DROP SUBSCRIPTION — supprimer une souscription

Synopsis

```
DROP SUBSCRIPTION [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

`DROP SUBSCRIPTION` supprime une souscription de l'instance de bases de données.

Une souscription peut seulement être supprimée par un superutilisateur.

`DROP SUBSCRIPTION` ne peut pas être exécutée dans un bloc de transaction si la souscription est associée à un slot de réplication. (Vous pouvez utiliser `ALTER SUBSCRIPTION` pour désinitialiser le slot.)

Paramètres

`IF EXISTS`

Ne renvoie pas une erreur si la souscription n'existe pas. Un message d'information est renvoyé dans ce cas.

nom

Le nom d'une souscription à supprimer.

`CASCADE`
`RESTRICT`

Ces mots-clés n'ont pas d'effet car il n'y a pas de dépendances sur les souscriptions.

Notes

Lors de la suppression d'une souscription associée à un slot de réplication sur l'hôte distant (l'état normal), `DROP SUBSCRIPTION` se connectera à l'hôte distant et tentera de supprimer le slot de réplication (et tout slot de synchronisation de table restant). Ceci est nécessaire pour que les ressources allouées pour la souscription sur l'hôte distant soient supprimées. Si cela échoue, soit parce que l'hôte distant n'est pas atteignable soit parce que le slot de réplication distant ne peut être supprimé ou n'existe pas ou n'a jamais existé, la commande `DROP SUBSCRIPTION` échouera. Pour continuer avec cette situation, tout d'abord désactivez la souscription en exécutant `ALTER SUBSCRIPTION ... DISABLE`, puis dissociez la souscription du slot de réplication en exécutant la commande `ALTER SUBSCRIPTION ... SET (slot_name = NONE)`. Après cela, `DROP SUBSCRIPTION` ne tentera pas de supprimer le propre slot de réplication de l'abonné. Il pourrait toujours se connecter au publieur pour supprimer les slots de synchronisation des tables créées en interne si certaines synchronisations des tables sont laissées non terminées ; si le publieur est inatteignable, ces slots (et le slot principal s'il existe toujours) doivent être supprimés manuellement. Sinon, ils continueront de réserver des WAL pour éventuellement remplir le disque. Voir aussi Section 31.2.1.

Si une souscription est associée avec un slot de réplication, `DROP SUBSCRIPTION` ne peut pas être exécutée à l'intérieur d'un bloc de transaction.

Exemples

Supprimer une souscription :

```
DROP SUBSCRIPTION mysub;
```

Compatibilité

`DROP SUBSCRIPTION` est une extension PostgreSQL.

Voir aussi

CREATE SUBSCRIPTION, ALTER SUBSCRIPTION

DROP TABLE

DROP TABLE — Supprimer une table

Synopsis

```
DROP TABLE [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP TABLE` supprime des tables de la base de données. Seuls le propriétaire de la table, le propriétaire du schéma et un superutilisateur peuvent détruire une table. `DELETE` et `TRUNCATE` sont utilisées pour supprimer les lignes d'une table sans détruire la table.

`DROP TABLE` supprime tout index, règle, trigger ou contrainte qui existe sur la table cible. Néanmoins, pour supprimer une table référencée par une vue ou par une contrainte de clé étrangère d'une autre table, `CASCADE` doit être ajouté. (`CASCADE` supprime complètement une vue dépendante mais dans le cas de la clé étrangère, il ne supprime que la contrainte, pas l'autre table.)

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom

Le nom de la table à supprimer (éventuellement qualifié du nom du schéma).

`CASCADE`

Les objets qui dépendent de la table (vues, par exemple) sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

La table n'est pas supprimée si un objet en dépend. Comportement par défaut.

Exemples

Supprimer les deux tables `films` et `distributeurs` :

```
DROP TABLE films, distributeurs;
```

Compatibilité

Cette commande est conforme au standard SQL. Cependant, le standard n'autorise pas la suppression de plusieurs tables en une seule commande. De plus, l'option `IF EXISTS` est une extension de PostgreSQL.

Voir aussi

ALTER TABLE, CREATE TABLE

DROP TABLESPACE

DROP TABLESPACE — Supprimer un tablespace

Synopsis

```
DROP TABLESPACE [ IF EXISTS ] nom
```

Description

`DROP TABLESPACE` supprime un tablespace du système.

Un tablespace ne peut être supprimé que par son propriétaire ou par un superutilisateur. Le tablespace doit être vide de tout objet de base de données avant sa suppression. Même si le tablespace ne contient plus d'objets de la base de données courante, il est possible que des objets d'autres bases de données l'utilisent. De plus, si le tablespace se trouve parmi les tablespaces du paramètre temp_tablespaces d'une session active, la commande `DROP` pourrait échouer à cause de fichiers temporaires stockés dans le tablespace.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom

Le nom du tablespace.

Notes

`DROP TABLESPACE` ne peut pas être exécuté à l'intérieur d'un bloc de transactions.

Exemples

Supprimer le tablespace `mes_affaires` :

```
DROP TABLESPACE mes_affaires;
```

Compatibilité

`DROP TABLESPACE` est une extension PostgreSQL.

Voir aussi

CREATE TABLESPACE, ALTER TABLESPACE

DROP TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH CONFIGURATION — Supprimer une configuration de recherche plein texte

Synopsis

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] nom [ CASCADE |
 RESTRICT ]
```

Description

`DROP TEXT SEARCH CONFIGURATION` supprime une configuration existante de la recherche plein texte. Pour exécuter cette commande, vous devez être le propriétaire de la configuration.

Paramètres

`IF EXISTS`

Ne renvoie pas une erreur si la configuration de recherche plein texte n'existe pas. Un message de niveau « NOTICE » est enregistré dans ce cas.

name

Le nom de la configuration de recherche plein texte (quelque fois qualifié du schéma).

`CASCADE`

Supprime automatiquement les objets dépendant de cette configuration de recherche plein texte, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Refuse la suppression de la configuration de recherche plein texte si des objets en dépendent. C'est la valeur par défaut.

Exemples

Supprimer la configuration de recherche plein texte `my_english` :

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

Cette commande échouera s'il existe des index qui référencent la configuration dans des appels `to_tsvector`. Ajoutez `CASCADE` pour supprimer ces index avec la configuration de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction `DROP TEXT SEARCH CONFIGURATION` dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH CONFIGURATION, CREATE TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH DICTIONARY

DROP TEXT SEARCH DICTIONARY — Supprimer un dictionnaire de recherche plein texte

Synopsis

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] nom [ CASCADE |
 RESTRICT ]
```

Description

`DROP TEXT SEARCH DICTIONARY` supprime un dictionnaire existant de la recherche plein texte. Pour exécuter cette commande, vous devez être le propriétaire du dictionnaire.

Paramètres

`IF EXISTS`

Ne renvoie pas une erreur si le dictionnaire de recherche plein texte n'existe pas. Un message de niveau « NOTICE » est enregistré dans ce cas.

name

Le nom du dictionnaire de recherche plein texte (quelque fois qualifié du schéma).

`CASCADE`

Supprime automatiquement les objets dépendant de ce dictionnaire de recherche plein texte, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Refuse la suppression du dictionnaire de recherche plein texte si des objets en dépendent. C'est la valeur par défaut.

Exemples

Supprimer le dictionnaire de recherche plein texte `english` :

```
DROP TEXT SEARCH DICTIONARY english;
```

Cette commande échouera s'il existe des configurations qui utilisent ce dictionnaire. Ajoutez `CASCADE` pour supprimer ces configurations avec le dictionnaire de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction `DROP TEXT SEARCH DICTIONARY` dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH DICTIONARY

DROP TEXT SEARCH PARSER

DROP TEXT SEARCH PARSER — Supprimer un analyseur de recherche plein texte

Synopsis

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

`DROP TEXT SEARCH PARSER` supprime un analyseur existant de la recherche plein texte. Pour exécuter cette commande, vous devez être superutilisateur.

Paramètres

`IF EXISTS`

Ne renvoie pas une erreur si l'analyseur de recherche plein texte n'existe pas. Un message de niveau « NOTICE » est enregistré dans ce cas.

name

Le nom de l'analyseur de recherche plein texte (quelque fois qualifié du schéma).

`CASCADE`

Supprime automatiquement les objets dépendant de l'analyseur de recherche plein texte, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Refuse la suppression de l'analyseur de recherche plein texte si des objets en dépendent. C'est la valeur par défaut.

Exemples

Supprimer l'analyseur de recherche plein texte `mon_analyseur` :

```
DROP TEXT SEARCH PARSER mon_analyseur;
```

Cette commande échouera s'il existe des configurations qui utilisent ce dictionnaire. Ajoutez `CASCADE` pour supprimer ces configurations avec l'analyseur de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction `DROP TEXT SEARCH PARSER` dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH PARSER, CREATE TEXT SEARCH PARSER

DROP TEXT SEARCH TEMPLATE

DROP TEXT SEARCH TEMPLATE — Supprimer un modèle de recherche plein texte

Synopsis

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] nom [ CASCADE | RESTRICT ]
```

Description

`DROP TEXT SEARCH TEMPLATE` supprime un modèle existant de la recherche plein texte. Pour exécuter cette commande, vous devez superutilisateur.

Paramètres

`IF EXISTS`

Ne renvoie pas une erreur si le modèle de recherche plein texte n'existe pas. Un message de niveau « NOTICE » est enregistré dans ce cas.

name

Le nom du modèle de recherche plein texte (quelque fois qualifié du schéma).

`CASCADE`

Supprime automatiquement les objets dépendant de ce modèle de recherche plein texte, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Refuse la suppression du modèle de recherche plein texte si des objets en dépendent. C'est la valeur par défaut.

Exemples

Supprimer le modèle de recherche plein texte `thesaurus` :

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

Cette commande échouera s'il existe des dictionnaires qui utilisent ce modèles. Ajoutez `CASCADE` pour supprimer ces dictionnaires avec le modèle de recherche plein texte.

Compatibilité

Il n'existe pas d'instruction `DROP TEXT SEARCH TEMPLATE` dans le standard SQL.

Voir aussi

ALTER TEXT SEARCH TEMPLATE, CREATE TEXT SEARCH TEMPLATE

DROP TRANSFORM

DROP TRANSFORM — supprime une transformation

Synopsis

```
DROP TRANSFORM [ IF EXISTS ] FOR nom_type
LANGUAGE nom_lang [ CASCADE | RESTRICT ]
```

Description

`DROP TRANSFORM` supprime une transformation définie précédement.

Pour pouvoir supprimer une transformation, vous devez être propriétaire du type et du langage. Ce sont les mêmes droits nécessaires lors de la création d'une transformation.

Paramètres

`IF EXISTS`

Permet de ne pas générer d'erreur si la transformation n'existe pas alors qu'on tente de la supprimer. Une notification est simplement renvoyée dans ce cas.

`nom_type`

Le nom du type de données de la transformation.

`nom_lang`

Le nom du langage de la transformation.

`CASCADE`

Supprime automatiquement tous les objets dépendants de la transformation, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Si des objets sont dépendants de la transformation, elle ne pourra pas être supprimée. Ce comportement est celui par défaut.

Exemples

Pour supprimer une relation du type `hstore` et du langage `plpython3u` :

```
DROP TRANSFORM FOR hstore LANGUAGE plpython3u;
```

Compatibilité

Cette forme de `DROP TRANSFORM` est une extension PostgreSQL. Voir CREATE TRANSFORM pour plus de détails.

Voir aussi

CREATE TRANSFORM

DROP TRIGGER

DROP TRIGGER — Supprimer un trigger

Synopsis

```
DROP TRIGGER [ IF EXISTS ] nom ON nom_table [ CASCADE | RESTRICT ]
```

Description

`DROP TRIGGER` supprime la définition d'un trigger. Seul le propriétaire de la table sur laquelle le trigger est défini peut exécuter cette commande.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

`nom`

Le nom du trigger à supprimer.

`nom_table`

Le nom de la table (éventuellement qualifié du nom du schéma) sur laquelle le trigger est défini.

`CASCADE`

Les objets qui dépendent du trigger sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Le trigger n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Destruction du trigger `si_dist_existe` de la table `films` :

```
DROP TRIGGER si_dist_existe ON films;
```

Compatibilité

L'instruction `DROP TRIGGER` de PostgreSQL est incompatible avec le standard SQL. Dans le standard, les noms de triggers ne se définissent pas par rapport aux tables. La commande est donc simplement `DROP TRIGGER` *`nom`*.

Voir aussi

CREATE TRIGGER

DROP TYPE

DROP TYPE — Supprimer un type de données

Synopsis

```
DROP TYPE [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP TYPE` supprime un type de données utilisateur. Seul son propriétaire peut le supprimer.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom

Le nom du type de données (éventuellement qualifié du nom de schéma) à supprimer.

`CASCADE`

Les objets qui dépendent du type (colonnes de table, fonctions, opérateurs...) sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

Le type n'est pas supprimé si un objet en dépend. Comportement par défaut.

Exemples

Supprimer le type de données `boite` :

```
DROP TYPE boite;
```

Compatibilité

Cette commande est similaire à celle du standard SQL en dehors de l'option `IF EXISTS` qui est une extension PostgreSQL. La majorité de la commande `CREATE TYPE` et les mécanismes d'extension de type de données de PostgreSQL diffèrent du standard.

Voir aussi

ALTER TYPE, CREATE TYPE

DROP USER

DROP USER — Supprimer un rôle de base de données

Synopsis

```
DROP USER [ IF EXISTS ] nom [, ...]
```

Description

`DROP USER` est une autre façon de faire un `DROP ROLE`.

Compatibilité

L'instruction `DROP USER` est une extension PostgreSQL. Le standard SQL laisse la définition des utilisateurs à l'implantation.

Voir aussi

DROP ROLE

DROP USER MAPPING

DROP USER MAPPING — Supprimer une correspondance d'utilisateur pour un serveur distant

Synopsis

```
DROP USER MAPPING [ IF EXISTS ] FOR { nom_utilisateur | USER |
 CURRENT_ROLE | CURRENT_USER | PUBLIC } SERVER nom_serveur
```

Description

`DROP USER MAPPING` supprime une correspondance d'utilisateur existant pour un serveur distant.

Le propriétaire d'un serveur distant peut supprimer les correspondances d'utilisateur pour ce serveur pour n'importe quel utilisateur. Par ailleurs, un utilisateur peut supprimer une correspondance d'utilisateur pour son propre nom d'utilisateur s'il a reçu le droit `USAGE` sur le serveur.

Paramètres

`IF EXISTS`

Ne génère pas d'erreur si la correspondance d'utilisateur n'existe pas. Un avertissement est émis dans ce cas.

`nom_utilisateur`

Nom d'utilisateur de la correspondance. `CURRENT_ROLE`, `CURRENT_USER` et `USER` correspondent au nom de l'utilisateur courant. `PUBLIC` est utilisé pour correspondre à tous les noms d'utilisateurs présents et futurs du système.

`nom_serveur`

Nom du serveur de la correspondance d'utilisateur.

Exemples

Supprimer une correspondance d'utilisateur `bob`, sur le serveur `truc` si elle existe :

```
DROP USER MAPPING IF EXISTS FOR bob SERVER truc;
```

Compatibilité

`DROP USER MAPPING` est conforme à la norme ISO/IEC 9075-9 (SQL/MED). La clause `IF EXISTS` est une extension PostgreSQL.

Voir aussi

CREATE USER MAPPING, ALTER USER MAPPING

DROP VIEW

DROP VIEW — Supprimer une vue

Synopsis

```
DROP VIEW [ IF EXISTS ] nom [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP VIEW` supprime une vue existante. Seul le propriétaire de la vue peut exécuter cette commande.

Paramètres

`IF EXISTS`

Ne pas renvoyer une erreur si l'agrégat n'existe pas. Un message d'avertissement est affiché dans ce cas.

nom

Le nom de la vue (éventuellement qualifié du nom de schéma) à supprimer.

`CASCADE`

Les objets qui dépendent de la vue (d'autres vues, par exemple) sont automatiquement supprimés, ainsi que tous les objets dépendants de ces objets (voir Section 5.14).

`RESTRICT`

La vue n'est pas supprimée si un objet en dépend. Comportement par défaut.

Exemples

Supprimer la vue `genre` :

```
DROP VIEW genre;
```

Compatibilité

Cette commande est conforme au standard SQL. Cependant, le standard n'autorise pas la suppression de plusieurs vues en une seule commande. De plus, l'option `IF EXISTS` est une extension de PostgreSQL.

Voir aussi

ALTER VIEW, CREATE VIEW

END

END — Valider la transaction en cours

Synopsis

```
END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Description

`END` valide la transaction en cours. Toutes les modifications réalisées lors de la transaction deviennent visibles pour les autres utilisateurs et il est garanti que les données ne seront pas perdues si un arrêt brutal survient. Cette commande est une extension PostgreSQL équivalente à `COMMIT`.

Paramètres

`WORK`
`TRANSACTION`

Mots clés optionnels. Ils n'ont pas d'effet.

`AND CHAIN`

Si `AND CHAIN` est précisé, une nouvelle transaction est immédiatement commencée avec les mêmes caractéristiques (voir SET TRANSACTION) que celle qui vient de se terminer. Sinon, aucune nouvelle transaction n'est démarrée.

Notes

`ROLLBACK` est utilisé pour annuler une transaction.

Lancer `END` à l'extérieur d'une transaction n'a aucun effet mais provoque un message d'avertissement.

Exemples

Valider la transaction en cours et rendre toutes les modifications persistantes :

```
END;
```

Compatibilité

`END` est une extension PostgreSQL fournissant une fonctionnalité équivalente à `COMMIT`, spécifiée dans le standard SQL.

Voir aussi

BEGIN, COMMIT, ROLLBACK

EXECUTE

EXECUTE — Exécuter une instruction préparée

Synopsis

```
EXECUTE nom [ (paramètre [, ...] ) ]
```

Description

EXECUTE est utilisé pour exécuter une instruction préparée au préalable. Comme les instructions préparées existent seulement pour la durée d'une session, l'instruction préparée doit avoir été créée par une instruction PREPARE exécutée plus tôt dans la session en cours.

Si l'instruction PREPARE qui crée l'instruction est appelée avec des paramètres, un ensemble compatible de paramètres doit être passé à l'instruction EXECUTE, sinon une erreur est levée. Contrairement aux fonctions, les instructions préparées ne sont pas surchargées en fonction de leur type ou du nombre de leurs paramètres ; le nom d'une instruction préparée doit être unique au sein d'une session.

Pour plus d'informations sur la création et sur l'utilisation des instructions préparées, voir PREPARE.

Paramètres

nom

Le nom de l'instruction préparée à exécuter.

paramètre

La valeur réelle du paramètre d'une instruction préparée. Ce paramètre doit être une expression ramenant une valeur dont le type est compatible avec celui spécifié pour ce paramètre positionnel dans la commande PREPARE qui a créé l'instruction préparée.

Sorties

La sortie renvoyée par la commande EXECUTE est celle de l'instruction préparée, et non celle de la commande EXECUTE.

Exemples

Des exemples sont donnés dans la section Exemples de la documentation de PREPARE.

Compatibilité

Le standard SQL inclut une instruction EXECUTE qui n'est utilisée que dans le SQL embarqué. La syntaxe utilisée par cette version de l'instruction EXECUTE diffère quelque peu.

Voir aussi

DEALLOCATE, PREPARE

EXPLAIN

EXPLAIN — Afficher le plan d'exécution d'une instruction

Synopsis

```
EXPLAIN [ ( option [, ...] ) ] instruction
EXPLAIN [ ANALYZE ] [ VERBOSE ] instruction

où option est :

    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    SETTINGS [ boolean ]
    BUFFERS [ boolean ]
    WAL [ boolean ]
    TIMING [ boolean ]
    SUMMARY [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

Description

Cette commande affiche le plan d'exécution que l'optimiseur de PostgreSQL engendre pour l'instruction fournie. Le plan d'exécution décrit le parcours de la (des) table(s) utilisée(s) dans la requête -- parcours séquentiel, parcours d'index, etc. -- . Si plusieurs tables sont référencées, il présente également les algorithmes de jointures utilisés pour rassembler les lignes issues des différentes tables.

La partie la plus importante de l'affichage concerne l'affichage des coûts estimés d'exécution. Ils représentent l'estimation faite par le planificateur des temps d'exécution de la requête (mesuré en une unité de coût arbitraire bien que conventionnellement ce sont des lectures de page disque). Deux nombres sont affichés : le coût de démarrage, écoulé avant que la première ligne soit renvoyée, et le coût d'exécution total, nécessaire au renvoi de toutes les lignes. Pour la plupart des requêtes, le coût qui importe est celui d'exécution totale. Mais dans certains cas, tel que pour une sous-requête dans la clause `EXISTS`, le planificateur choisira le coût de démarrage le plus court, et non celui d'exécution totale (car, de toute façon, l'exécuteur s'arrête après la récupération d'une ligne). De même, lors de la limitation des résultats à retourner par une clause `LIMIT`, la planificateur effectue une interpolation entre les deux coûts limites pour choisir le plan réellement le moins coûteux.

L'option `ANALYZE` impose l'exécution de la requête en plus de sa planification. De ce fait, les statistiques d'exécution réelle sont ajoutées à l'affichage, en incluant le temps total écoulé à chaque nœud du plan (en millisecondes) et le nombre total de lignes renvoyées. C'est utile pour vérifier la véracité des informations fournies par le planificateur.

> **Important**
>
> Il ne faut pas oublier que l'instruction est réellement exécutée avec l'option `ANALYZE`. Bien qu'`EXPLAIN` inhibe l'affichage des retours d'une commande `SELECT`, les autres effets de l'instruction sont présents. Si `EXPLAIN ANALYZE` doit être utilisé sur une instruction `INSERT`, `UPDATE`, `DELETE`, `MERGE`, `CREATE TABLE AS` ou `EXECUTE` sans que la commande n'affecte les données, l'approche suivante peut être envisagée :
>
> ```
> BEGIN;
> ```

```
EXPLAIN ANALYZE ...;
ROLLBACK;
```

Seules les options ANALYZE et VERBOSE peuvent être utilisées et dans cet ordre seulement si la liste d'options entre parenthèses n'est pas utilisé. Avant PostgreSQL 9.0, la seule syntaxe supportée était celle sans parenthèses. Les nouvelles options ne seront supportées que par la nouvelle syntaxe, celle avec les parenthèses.

Paramètres

ANALYZE

Exécute la commande et affiche les temps d'exécution réels et d'autres statistiques. Ce paramètre est par défaut à FALSE.

VERBOSE

Affiche des informations supplémentaires sur le plan. Cela inclut la liste des colonnes en sortie pour chaque nœud du plan, les noms des tables et fonctions avec le nom du schéma, les labels des variables dans les expressions avec des alias de tables et le nom de chaque trigger pour lesquels les statistiques sont affichées. L'identifiant de requête sera aussi affiché si une requête a été évaluée, voir compute_query_id pour plus de détails. Ce paramètre est par défaut à FALSE.

COSTS

Inclut des informations sur le coût estimé au démarrage et au total de chaque nœud du plan, ainsi que le nombre estimé de lignes et la largeur estimée de chaque ligne. Ce paramètre est par défaut à TRUE.

BUFFERS

Inclut des informations sur l'utilisation des tampons. Spécifiquement, inclut le nombre de blocs partagés lus dans la cache, lus en dehors du cache, modifiés et écrits, le nombre de blocs locaux lus dans le cache, lus en dehors du cache, modifiés, et écrits, et le nombre de blocs temporaires lus et écrits, et le temps passé à lire et écrire des blocs dans les fichiers de données et dans les fichiers temporaires (en millisecondes) si le paramètre track_io_timing est activé. Le terme *hit* signifie que la lecture a été évitée car le bloc se trouvait déjà dans le cache. Les blocs partagés contiennent les données de tables et index standards ; les blocs locaux contiennent les tables et index temporaires ; les blocs temporaires contiennent les données de travail à court terme, comme les tris, les hachages, les nœuds Materialize, et des cas similaires. Le nombre de blocs modifiés (*dirtied*) indique le nombre de blocs précédemment propres qui ont été modifiés par cette requête ; le nombre de blocs écrits (*written*) indique le nombre de blocs déjà modifiés qui a été enlevé du cache pour être écrit sur disque lors de l'exécution de cette requête. Le nombre de blocs affichés pour un nœud de niveau supérieur inclut ceux utilisés par tous ses enfants. Dans le format texte, seules les valeurs différentes de zéro sont affichées. Sa valeur par défaut est FALSE.

SETTINGS

Inclut des informations sur les paramètres de configuration. Plus précisément, inclut les options affectant l'optimisation des requêtes et qui ont des valeurs différentes de leur valeur par défaut. Sa valeur par défaut est FALSE.

WAL

Inclut des informations sur les génération des enregistrements dans les WAL. Spécifiquement, inclut le nombre d'enregistrements, le nombre d'images complète de bloc (FPI), et la quantité d'octets générés dans les WAL. Dans le format texte, seules les valeurs différentes de zéro seront

affichées. Ce paramètre peut seulement aussi être utilisé quand ANALYZE est aussi activé. Il vaut par défaut FALSE.

TIMING

Inclut le temps réel de démarrage et le temps réel passé dans le nœud en sortie. La surcharge de la lecture répétée de l'horloge système peut ralentir la requête de façon significative sur certains systèmes, et donc il est utile de pouvoir configurer ce paramètre à FALSE quand seuls le décompte réel des lignes est nécessaire. La durée d'exécution complète de la commande est toujours mesurée, même si le chronométrage des nœuds est désactivé avec cette option. Ce paramètre peut seulement être utilisé quand l'option ANALYZE est aussi activée. La valeur par défaut est TRUE.

SUMMARY

Inclut des informations résumées (par exemple : information de temps total) après le plan de la requête. Les informations résumées sont inclues par défaut quand ANALYZE est utilisé mais sinon ne sont pas inclues par défaut, mais peuvent être activées avec cette option. Le temps de planification dans EXPLAIN EXECUTE inclut le temps nécessaire pour récupérer le plan du cache ainsi que le temps nécessaire pour le replanifier, si nécessaire.

FORMAT

Indique le format de sortie. Il peut valoir TEXT, XML, JSON ou YAML. Toutes les sorties contiennent les mêmes informations, mais les programmes pourront plus facilement traiter les sorties autres que TEXT. Ce paramètre est par défaut à TEXT.

boolean

Spécifie si l'option sélectionnée doit être activée ou désactivée. Vous pouvez écrire TRUE, ON ou 1 pour activer l'option, et FALSE, OFF ou 0 pour la désactiver. La valeur de type *boolean* peut aussi être omise, auquel cas la valeur sera TRUE.

instruction

Toute instruction SELECT, INSERT, UPDATE, DELETE, MERGE, VALUES, EXECUTE, DECLARE, CREATE TABLE AS ou CREATE MATERIALIZED VIEW AS dont le plan d'exécution est souhaité.

Sorties

La sortie de la commande est une description textuelle du plan sélectionné pour la *requête*, annotée en option des statistiques d'exécution. Section 14.1 décrit les informations fournies.

Notes

Pour permettre au planificateur de requêtes de PostgreSQL de prendre des décisions en étant raisonnablement informé pour l'optimisation des requêtes, les données du catalogue pg_statistic doivent être à jour pour toutes les tables utilisées dans la requête. Habituellement, le démon autovacuum s'en chargera automatiquement. Mais si une table a eu récemment des changements importants dans son contenu, vous pourriez avoir besoin de lancer un ANALYZE manuel plutôt que d'attendre que l'autovacuum s'occupe des modifications.

Pour mesurer le coût d'exécution de chaque nœud dans le plan d'exécution, l'implémentation actuelle de la commande EXPLAIN ANALYZE ajoute une surcharge de profilage à l'exécution de la requête. En résultat, exécuter EXPLAIN ANALYZE sur une requête peut parfois prendre un temps significativement plus long que l'exécution de la requête. La durée supplémentaire dépend de la nature de la requête ainsi que de la plateforme utilisée. Le pire des cas survient pour les nœuds du plan nécessitant en eux-même peu de durée d'exécution par exécution et sur les machines disposant d'appels systèmes relativement lents pour obtenir l'heure du jour.

Exemples

Afficher le plan d'une requête simple sur une table d'une seule colonne de type `integer` et 10000 lignes :

```
EXPLAIN SELECT * FROM foo;

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

Voici le même plan, mais formaté avec JSON :

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
           QUERY PLAN
--------------------------------
 [                             +
   {                           +
     "Plan": {                 +
       "Node Type": "Seq Scan",+
       "Relation Name": "foo", +
       "Alias": "foo",         +
       "Startup Cost": 0.00,   +
       "Total Cost": 155.00,   +
       "Plan Rows": 10000,     +
       "Plan Width": 4         +
     }                         +
   }                           +
 ]
(1 row)
```

S'il existe un index et que la requête contient une condition `WHERE` indexable, `EXPLAIN` peut afficher un plan différent :

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                          QUERY PLAN
--------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

Voici le même plan, mais formaté avec YAML :

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
          QUERY PLAN
-------------------------------
 - Plan:                      +
     Node Type: "Index Scan"  +
     Scan Direction: "Forward"+
     Index Name: "fi"         +
     Relation Name: "foo"     +
     Alias: "foo"             +
     Startup Cost: 0.00       +
```

```
     Total Cost: 5.98         +
     Plan Rows: 1             +
     Plan Width: 4            +
     Index Cond: "(i = 4)"
(1 row)
```

L'obtention du format XML est laissé en exercice au lecteur.

Voici le même plan avec les coûts supprimés :

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;

        QUERY PLAN
---------------------------
 Index Scan using fi on foo
   Index Cond: (i = 4)
(2 rows)
```

Exemple de plan de requête pour une requête utilisant une fonction d'agrégat :

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                            QUERY PLAN
-------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6
 width=4)
         Index Cond: (i < 10)
(3 rows)
```

Exemple d'utilisation de `EXPLAIN EXECUTE` pour afficher le plan d'exécution d'une requête préparée :

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);

                                                              QUERY PLAN
-----------------------------------------------------------------------------------------
 HashAggregate  (cost=9.54..9.54 rows=1 width=8) (actual
 time=0.156..0.161 rows=11 loops=1)
   Group Key: foo
   ->  Index Scan using test_pkey on test  (cost=0.29..9.29 rows=50
 width=8) (actual time=0.039..0.091 rows=99 loops=1)
         Index Cond: ((id > $1) AND (id < $2))
 Planning time: 0.197 ms
 Execution time: 0.225 ms
(6 rows)
```

Il est évident que les nombres présentés ici dépendent du contenu effectif des tables impliquées. De plus, les nombres, et la stratégie sélectionnée elle-même, peuvent différer en fonction de la version de PostgreSQL du fait des améliorations apportées au planificateur. Il faut également savoir que la

commande ANALYZE calcule les statistiques des données à partir d'extraits aléatoires ; il est de ce fait possible que les coûts estimés soient modifiés après l'exécution de cette commande, alors même la distribution réelle des données dans la table n'a pas changé.

Compatibilité

L'instruction EXPLAIN n'est pas définie dans le standard SQL.

Voir aussi

ANALYZE

FETCH

FETCH — Récupérer les lignes d'une requête à l'aide d'un curseur

Synopsis

```
FETCH [ direction ] [ FROM | IN ] nom_curseur

où direction fait partie
de :

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE nombre
    RELATIVE nombre
    nombre
    ALL
    FORWARD
    FORWARD nombre
    FORWARD ALL
    BACKWARD
    BACKWARD nombre
    BACKWARD ALL
```

Description

`FETCH` récupère des lignes en utilisant un curseur précédemment ouvert.

À un curseur est associée une position associée utilisée par `FETCH`. Le curseur peut être positionné avant la première ligne du résultat de la requête, sur une ligne particulière du résultat ou après la dernière ligne du résultat. À sa création, le curseur est positionné avant la première ligne. Après récupération de lignes, le curseur est positionné sur la ligne la plus récemment récupérée. Si `FETCH` atteint la fin des lignes disponibles, il est positionné après la dernière ligne ou avant la première ligne dans le cas d'une récupération remontante. `FETCH ALL` ou `FETCH BACKWARD ALL` positionne toujours le curseur après la dernière ligne ou avant la première ligne.

Les formes `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE`, `RELATIVE` récupèrent une seule ligne après déplacement approprié du curseur. Si cette ligne n'existe pas, un résultat vide est renvoyé et le curseur est positionné avant la première ligne ou après la dernière ligne, en fonction du sens de la progression.

Les formes utilisant `FORWARD` et `BACKWARD` récupèrent le nombre de lignes indiqué en se déplaçant en avant ou en arrière, laissant le curseur positionné sur la dernière ligne renvoyée (ou après/avant toutes les lignes si *`nombre`* dépasse le nombre de lignes disponibles).

`RELATIVE 0`, `FORWARD 0` et `BACKWARD 0` récupèrent tous la ligne actuelle sans déplacer le curseur, c'est-à-dire qu'ils effectuent une nouvelle récupération de la ligne dernièrement récupérée. La commande réussit sauf si le curseur est positionné avant la première ligne ou après la dernière ligne ; dans ce cas, aucune ligne n'est renvoyée.

> **Note**
>
> Cette page décrit l'utilisation des curseurs au niveau de la commande SQL. Si vous voulez utiliser des curseurs dans une fonction PL/pgSQL, les règles sont différentes -- voir Section 43.7.3.

Paramètres

direction

La direction et le nombre de lignes à récupérer. Ce paramètre peut prendre les valeurs suivantes :

`NEXT`

La ligne suivante est récupérée. C'est le comportement par défaut si *direction* est omis.

`PRIOR`

La ligne précédente est récupérée.

`FIRST`

La première ligne de la requête est récupérée. C'est identique à `ABSOLUTE 1`.

`LAST`

La dernière ligne de la requête est récupérée. C'est identique à `ABSOLUTE -1`.

`ABSOLUTE` *nombre*

La *nombre*-ième ligne de la requête est récupérée, ou la `abs(`*nombre*`)`-ième ligne à partir de la fin si *nombre* est négatif. Le curseur est positionné avant la première ligne ou après la dernière si *nombre* est en dehors des bornes ; en particulier, `ABSOLUTE 0` le positionne avant la première ligne.

`RELATIVE` *nombre*

La *nombre*-ième ligne suivante est récupérée, ou la `abs(`*nombre*`)`-ième ligne précédente si *nombre* est négatif. `RELATIVE 0` récupère de nouveau la ligne courante, si elle existe.

nombre

Les *nombre* lignes suivantes sont récupérées. C'est identique à `FORWARD` *nombre*.

`ALL`

Toutes les lignes restantes sont récupérées. C'est identique à `FORWARD ALL`).

`FORWARD`

La ligne suivante est récupérée. C'est identique à `NEXT`.

`FORWARD` *nombre*

Les *nombre* lignes suivantes sont récupérées. `FORWARD 0` récupère de nouveau la ligne courante.

`FORWARD ALL`

Toutes les lignes restantes sont récupérées.

BACKWARD

La ligne précédente est récupérée. C'est identique à PRIOR.

BACKWARD *nombre*

Les *nombre* lignes précédentes sont récupérées (parcours inverse). BACKWARD 0 récupère de nouveau la ligne courante.

BACKWARD ALL

Toutes les lignes précédentes sont récupérées (parcours inverse).

nombre

Constante de type entier éventuellement signé, qui précise l'emplacement ou le nombre de lignes à récupérer. Dans le cas de FORWARD et BACKWARD, préciser une valeur négative pour *nombre* est équivalent à modifier le sens de FORWARD et BACKWARD.

nom_curseur

Le nom d'un curseur ouvert.

Sorties

En cas de succès, une commande FETCH renvoie une balise de commande de la forme

FETCH *nombre*

Le *nombre* est le nombre de lignes récupérées (éventuellement zéro). Dans psql, la balise de commande n'est pas réellement affichée car psql affiche à la place les lignes récupérées.

Notes

Le curseur doit être déclaré avec l'option SCROLL si les variantes de FETCH autres que FETCH NEXT ou FETCH FORWARD avec un nombre positif sont utilisées. Pour les requêtes simples, PostgreSQL autorise les parcours inverses à partir de curseurs non déclarés avec SCROLL. il est toutefois préférable de ne pas se fonder sur ce comportement. Si le curseur est déclaré avec NO SCROLL, aucun parcours inverse n'est autorisé.

Les récupérations ABSOLUTE ne sont pas plus rapides que la navigation vers la ligne désirée par déplacement relatif : de toute façon, l'implantation sous-jacente doit parcourir toutes les lignes intermédiaires. Les récupérations absolues négatives font même pis : la requête doit être lue jusqu'à la fin pour trouver la dernière ligne, puis relue en sens inverse à partir de là. Néanmoins, remonter vers le début de la requête (comme avec FETCH ABSOLUTE 0) est rapide.

DECLARE est utilisé pour définir un curseur. MOVE est utilisé pour modifier la position du curseur sans récupérer les données.

Exemples

Parcourir une table à l'aide d'un curseur :

```
BEGIN WORK;

-- Initialiser le curseur :
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;
```

```
-- Récupèrer les 5 premières lignes du curseur liahona :
FETCH FORWARD 5 FROM liahona;

 code  |          titre          | did | date_prod  |   genre  |
 longueur
-------+-------------------------+-----+------------+----------
+-----------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    |
 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic |
 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic |
 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   |
 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    |
 02:28

-- Récupèrer la ligne précédente :
FETCH PRIOR FROM liahona;

 code  |  titre  | did | date_prod  |  genre |  longueur
-------+---------+-----+------------+--------+-----------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- Fermer le curseur et terminer la transaction:
CLOSE liahona;
COMMIT WORK;
```

Compatibilité

Le standard SQL ne définit FETCH que pour une utilisation en SQL embarqué. La variante de FETCH décrite ici renvoie les données comme s'il s'agissait du résultat d'un SELECT plutôt que de le placer dans des variables hôtes. À part cela, FETCH est totalement compatible avec le standard SQL.

Les formes de FETCH qui impliquent FORWARD et BACKWARD, ainsi que les formes FETCH *nombre* et FETCH ALL, dans lesquelles FORWARD est implicite, sont des extensions PostgreSQL.

Le standard SQL n'autorise que FROM devant le nom du curseur ; la possibilité d'utiliser IN, ou de les laisser, est une extension.

Voir aussi

CLOSE, DECLARE, MOVE

GRANT

GRANT — Définir les droits d'accès

Synopsis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES
 | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] nom_table [, ...]
         | ALL TABLES IN SCHEMA nom_schéma [, ...] }
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( nom_colonne
 [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( nom_colonne [, ...] ) }
    ON [ TABLE ] nom_table [, ...]
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE nom_séquence [, ...]
         | ALL SEQUENCES IN SCHEMA nom_schéma [, ...] }
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL
 [ PRIVILEGES ] }
    ON DATABASE nom_base [, ...]
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN nom_domaine [, ...]
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER nom_fdw [, ...]
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER nom_serveur [, ...]
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } nom_routine
 [ ( [ [ mode_arg ] [ nom_arg ] type_arg [, ...] ] ) ] [, ...]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN
 SCHEMA nom_schéma [, ...] }
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]
```

```
GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE nom_lang [, ...]
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { { SET | ALTER SYSTEM } [, ... ] | ALL [ PRIVILEGES ] }
    ON PARAMETER paramètre_configuration [, ...]
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA nom_schéma [, ...]
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE nom_type [, ...]
    TO spécification_rôle [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY spécification_rôle ]

GRANT nom_role [, ...] TO spécification_rôle [, ...]
    [ WITH ADMIN OPTION ]
    [ GRANTED BY spécification_rôle ]

où spécification_rôle peut valoir :

    [ GROUP ] nom_rôle
  | PUBLIC
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
```

Description

La commande GRANT a deux variantes basiques : la première donne des droits sur un objet de la base de données (table, colonne, vue, table distante, séquence, base de données, wrapper de données distantes, serveur distant, fonction, procédure, langage de procédure, « Large Object », paramètre de configuration, schéma, tablespace ou type), la seconde gère les appartenances à un rôle. Ces variantes sont assez similaires mais somme toute assez différentes pour être décrites séparément.

GRANT sur les objets de la base de données

Cette variante de la commande GRANT donne des droits spécifiques sur un objet de la base de données a un ou plusieurs rôles. Ces droits sont ajoutés à ceux déjà possédés, s'il y en a.

Le mot clé PUBLIC indique que les droits sont donnés à tous les rôles, y compris ceux créés ultérieurement. PUBLIC peut être vu comme un groupe implicitement défini qui inclut en permanence

tous les rôles. Un rôle particulier dispose de la somme des droits qui lui sont acquis en propre, des droits de tout rôle dont il est membre et des droits donnés à PUBLIC.

Si WITH GRANT OPTION est précisé, celui qui reçoit le droit peut le transmettre à son tour (NDT : par la suite on parlera d'« option de transmission de droit », là où en anglais il est fait mention de « grant options »). Sans l'option GRANT, l'utilisateur ne peut pas le faire. Cette option ne peut pas être donnée à PUBLIC.

Si GRANTED BY est précisé, le donneur indiqué doit être l'utilisateur courant. Cette clause est actuellement présente dans cette forme pour la compatibilité SQL.

Il n'est pas nécessaire d'accorder des droits au propriétaire d'un objet (habituellement l'utilisateur qui l'a créé) car, par défaut, le propriétaire possède tous les droits. (Le propriétaire peut toutefois choisir de révoquer certains de ses propres droits.)

Le droit de supprimer un objet ou de modifier sa définition n'est pas configurable avec cette commande. Il est spécifique au propriétaire de l'objet. Ce droit ne peut ni être donné ni supprimé. Néanmoins, il est possible d'avoir le même effet en rendant un utilisateur membre du rôle qui possède cet object ou en le supprimant de ce rôle. Le propriétaire a aussi implicitement les options de transmission de droits pour l'objet.

Les droits possibles sont :

SELECT
INSERT
UPDATE
DELETE
TRUNCATE
REFERENCES
TRIGGER
CREATE
CONNECT
TEMPORARY
EXECUTE
USAGE
SET
ALTER SYSTEM

Types spécifiques de droits, comme définis dans Section 5.7.

TEMP

Autre écriture de TEMPORARY.

ALL PRIVILEGES

Donner tous les droits disponibles pour ce type d'objet. Le mot-clé PRIVILEGES est optionnel dans PostgreSQL, bien qu'il soit requis en SQL.

La syntaxe FUNCTION fonctionne pour les fonctions simples, les fonctions d'agrégat, et les fonctions de fenêtrage, mais pas pour les procédures ; utilisez PROCEDURE pour ces dernières. Vous pouvez aussi utiliser ROUTINE pour faire référence à une fonction simple, une fonction d'agrégat, une fonction de fenêtrage ou une procédure.

Il existe aussi une option pour donner les droits sur tous les objets de même type dans un ou plusieurs schémas. Cette fonctionnalité est actuellement supportée par les tables, séquences, fonctions et procédures. ALL TABLES affecte aussi les vues et les tables externes, tout comme la commande GRANT de cet objet. ALL FUNCTIONS affecte aussi les fonctions d'agrégat et les fonctions de fenêtrage, mais pas les procédures, encore une fois tout comme la commande GRANT spécifique à l'objet. Utilisez ALL ROUTINES pour inclure les procédures.

GRANT sur les rôles

Cette variante de la commande GRANT définit l'appartenance d'un (ou plusieurs) rôle(s) à un autre. L'appartenance à un rôle est importante car elle offre tous les droits accordés à un rôle à l'ensemble de ses membres.

Si WITH ADMIN OPTION est spécifié, le membre peut à la fois en octroyer l'appartenance à d'autres rôles, et la révoquer. Sans cette option, les utilisateurs ordinaires ne peuvent pas le faire. Un rôle ne dispose pas de l'option WITH ADMIN OPTION lui-même Les superutilisateurs de la base peuvent donner ou révoquer l'appartenance à un rôle pour tout personne. Les rôles ayant l'attribut CREATEROLE peuvent donner ou révoquer l'appartenance à un rôle pour toute personne qui n'est pas un superutilisateur.

Si GRANTED BY est utilisé, l'ajout du droit est enregistré comme étant fait avec le rôle indiqué. Seuls les superutilisateurs peuvent utiliser cette option, sauf quand le nom indiqué est le même que celui qui exécute la commande.

Contrairement au cas avec les droits, l'appartenance à un rôle ne peut pas être donné à PUBLIC. Notez aussi que ce format de la commande n'autorise pas le mot GROUP dans *spécification_rôle*.

Notes

La commande REVOKE est utilisée pour retirer les droits d'accès.

Depuis PostgreSQL 8.1, le concept des utilisateurs et des groupes a été unifié en un seul type d'entité appelé rôle. Il n'est donc plus nécessaire d'utiliser le mot clé GROUP pour indiquer si le bénéficiaire est un utilisateur ou un groupe. GROUP est toujours autorisé dans cette commande mais est ignoré.

Un utilisateur peut exécuter des SELECT, INSERT, etc. sur une colonne si il a le privilège soit sur cette colonne spécifique, soit sur la table entière. Donner un privilège de table puis le révoquer pour une colonne ne fera pas ce que vous pourriez espérer : l'autorisation au niveau de la table n'est pas affectée par une opération au niveau de la colonne.

Quand un utilisateur, non propriétaire d'un objet, essaie d'octroyer des droits sur cet objet, la commande échoue si l'utilisateur n'a aucun droit sur l'objet. Tant que des privilèges existent, la commande s'exécute, mais n'octroie que les droits pour lesquels l'utilisateur dispose de l'option de transmission. Les formes GRANT ALL PRIVILEGES engendrent un message d'avertissement si aucune option de transmission de droit n'est détenue, tandis que les autres formes n'engendrent un message que lorsque les options de transmission du privilège concerné par la commande ne sont pas détenues. (Cela s'applique aussi au propriétaire de l'objet, mais comme on considère toujours que ce dernier détient toutes les options de transmission, le problème ne se pose jamais.)

Les superutilisateurs de la base de données peuvent accéder à tous les objets sans tenir compte des droits qui les régissent. Cela est comparable aux droits de root sur un système Unix. Comme avec root, il est déconseillé d'opérer en tant que superutilisateur, sauf en cas d'impérieuse nécessité.

Si un superutilisateur lance une commande GRANT ou REVOKE, tout se passe comme si la commande était exécutée par le propriétaire de l'objet concerné. Les droits octroyés par cette commande semblent ainsi l'avoir été par le propriétaire de l'objet. (L'appartenance à rôle, elle, semble être donnée par le rôle conteneur.)

GRANT et REVOKE peuvent aussi être exécutées par un rôle qui n'est pas le propriétaire de l'objet considéré, mais est membre du rôle propriétaire de l'objet, ou membre du rôle titulaire du privilège WITH GRANT OPTION sur cet objet. Dans ce cas, les droits sont enregistrés comme donnés par le rôle propriétaire de l'objet ou titulaire du privilège WITH GRANT OPTION. Par exemple, si la table t1 appartient au rôle g1, dont le rôle u1 est membre, alors u1 peut donner les droits sur t1 à u2, mais ces droits apparaissent octroyés directement par g1. Tout autre membre du rôle g1 peut les révoquer par la suite.

Si le rôle qui exécute GRANT détient, de manière indirecte, les droits souhaités à travers plus d'un niveau d'appartenance, il est difficile de prévoir le rôle reconnu comme fournisseur du privilège. Dans de tels cas, le meilleur moyen d'utiliser SET ROLE est de devenir le rôle qui doit octroyer les droits.

Donner un droit sur une table n'étend pas automatiquement les droits sur les séquences utilisées par cette table, ceci incluant les séquences liées par des colonnes de type SERIAL. Les droits sur les séquences doivent être donnés séparément.

Voir Section 5.7 pour plus d'informations sur les types de droit spécifiques, ainsi que sur la façon d'inspecter les droits sur les objets.

Exemples

Donner le droit d'insertion à tous les utilisateurs sur la table films :

```
GRANT INSERT ON films TO PUBLIC;
```

Donner tous les droits possibles à l'utilisateur manuel sur la vue genres :

```
GRANT ALL PRIVILEGES ON genres TO manuel;
```

Bien que la commande ci-dessus donne tous les droits lorsqu'elle est exécutée par un superutilisateur ou par le propriétaire de genres, exécutée par quelqu'un d'autre, elle n'accorde que les droits pour lesquels cet utilisateur possède l'option de transmission.

Rendre joe membre de admins :

```
GRANT admins TO joe;
```

Compatibilité

Conformément au standard SQL, le mot clé PRIVILEGES est requis dans ALL PRIVILEGES. Le standard SQL n'autorise pas l'initialisation des droits sur plus d'un objet par commande.

PostgreSQL autorise un propriétaire d'objet à révoquer ses propres droits ordinaires : par exemple, le propriétaire d'un objet peut le placer en lecture seule pour lui-même en révoquant ses propres droits INSERT, UPDATE, DELETE et TRUNCATE. Le standard SQL ne l'autorise pas. La raison en est que PostgreSQL traite les droits du propriétaire comme ayant été donnés par le propriétaire ; il peut, de ce fait, aussi les révoquer. Dans le standard SQL, les droits du propriétaire sont donnés par une entité « _SYSTEM ». N'étant pas « _SYSTEM », le propriétaire ne peut pas révoquer ces droits.

D'après le standard SQL, les options de cette commande peuvent être données à PUBLIC ; PostgreSQL supporte seulement l'ajout des options de droits aux rôles.

Le standard SQL autorise l'utilisation de l'option GRANTED BY pour indiquer seulement CURRENT_USER ou CURRENT_ROLE. Les autres variants sont des extensions de PostgreSQL.

Le standard SQL fournit un droit USAGE sur d'autres types d'objet : jeux de caractères, collations, conversions.

Dans le standard SQL, seules les séquences ont un droit USAGE qui contrôle l'utilisation de l'expression NEXT VALUE FOR, un équivalent de la fonction nextval dans PostgreSQL. Les droits SELECT et UPDATE des séquences sont une extension de PostgreSQL. L'application du droit USAGE de la séquence à la fonction currval est aussi une extension PostgreSQL (comme l'est la fonction elle-même).

Les droits sur les bases de données, tablespaces, schémas, langages et paramètres de configuration sont des extensions PostgreSQL.

Voir aussi

REVOKE, ALTER DEFAULT PRIVILEGES

IMPORT FOREIGN SCHEMA

IMPORT FOREIGN SCHEMA — importe les définitions d'une table d'une instance différente

Synopsis

```
IMPORT FOREIGN SCHEMA schema_distant
    [ { LIMIT TO | EXCEPT } ( nom_table [, ...] ) ]
    FROM SERVER nom_serveur
    INTO schema_local
    [ OPTIONS ( option 'valeur' [, ... ] ) ]
```

Description

`IMPORT FOREIGN SCHEMA` crée une table externe qui représente une table existant dans une autre instance. L'utilisateur qui lance la commande sera propriétaire de la nouvelle table externe. La table sera créée avec des définition de colonnes et options en cohérence avec ce qui est définit pour l'instance distante.

Par défaut, toutes les tables et vues, existantes dans un schéma particulier de l'instance distante, sont importées. Il est possible de limiter la liste des tables à un sous ensemble, ou d'exclure des tables spécifiques. Les nouvelles tables externes sont toutes créées dans le schéma cible, qui doit déjà exister.

Pour utiliser `IMPORT FOREIGN SCHEMA`, l'utilisateur doit avoir le droit `USAGE` sur l'instance distante, ainsi que le droit `CREATE` sur le schéma cible.

Paramètres

schema_distant

C'est le schéma distant depuis lequel on réalise l'import. La signification spécifique d'un schéma distant dépend du wrapper de données distantes (foreign data wrapper) en cours d'utilisation.

`LIMIT TO ( nom_table [, ...] )`

Importe seulement les tables distantes qui ont été spécifiées. Toutes les autres tables du schéma distant seront ignorées.

`EXCEPT ( nom_table [, ...] )`

Exclut toutes les tables distantes qui ont été spécifiées. Toutes les tables du schéma distant seront importées sauf celles définies dans cette liste.

nom_serveur

Le serveur distant depuis lequel on importe.

schéma_local

Le schéma dans lequel sont créées les tables externes pour y importer les données distantes.

`OPTIONS ( option 'valeur' [, ...] )`

Options à utiliser lors de l'import. Les noms et valeurs autorisés d'options sont spécifiques à chaque wrapper de données distantes.

Exemples

On importe la définition des tables depuis un schéma distant `films_distants` du serveur `serveur_film`, en créant une table étrangère dans le schéma local `films` :

```
IMPORT FOREIGN SCHEMA films_distants
    FROM SERVER serveur_film INTO films;
```

Comme précédemment mais en important seulement les deux tables `acteurs` et `réalisateurs` (s'ils existent) :

```
IMPORT FOREIGN SCHEMA films_distants LIMIT TO (acteurs,
 réalisateurs)
    FROM SERVER serveur_film INTO films;
```

Compatibilité

La commande `IMPORT FOREIGN SCHEMA` se conforme au standard SQL standard, sauf sur la clause `OPTIONS` qui est une extension PostgreSQL.

Voir aussi

CREATE FOREIGN TABLE, CREATE SERVER

INSERT

INSERT — Insérer de nouvelles lignes dans une table

Synopsis

```
[ WITH [ RECURSIVE ] requête_with [, ...] ]
INSERT INTO nom_table [ AS alias ] [ ( nom_colonne [, ...] ) ]
    [ OVERRIDING { SYSTEM | USER } VALUE ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] )
 [, ...] | requête }
    [ ON CONFLICT [ cible_conflit ] action_conflit ]
    [ RETURNING { * | expression_sortie [ [ AS ] nom_sortie ] }
 [, ...] ]

où cible_conflit peut valoir :

    ( { nom_colonne_index | ( expression_index ) }
 [ COLLATE collation ] [ classe_operateur ] [, ...] )
 [ WHERE predicat_index ]
    ON CONSTRAINT nom_contrainte

et action_conflit peut valoir :

    DO NOTHING
    DO UPDATE SET { nom_colonne = { expression | DEFAULT } |
                    ( nom_colonne [, ...] ) = [ ROW ]
 ( { expression | DEFAULT } [, ...] ) |
                    ( nom_colonne [, ...] ) = ( sous-SELECT )
                  } [, ...]
              [ WHERE condition ]
```

Description

`INSERT` insère de nouvelles lignes dans une table. Vous pouvez insérer une ou plusieurs lignes spécifiées par les expressions de valeur, ou zéro ou plusieurs lignes provenant d'une requête.

L'ordre des noms des colonnes n'a pas d'importance. Si aucune liste de noms de colonnes n'est donnée, toutes les colonnes de la table sont utilisée dans l'ordre de leur déclaration (les *`N`* premiers noms de colonnes si seules *`N`* valeurs de colonnes sont fournies dans la clause `VALUES` ou dans la *`requête`*). Les valeurs fournies par la clause `VALUES` ou par la *`requête`* sont associées à la liste explicite ou implicite des colonnes de gauche à droite.

Chaque colonne absente de la liste, implicite ou explicite, des colonnes se voit attribuer sa valeur par défaut, s'il y en a une, ou NULL dans le cas contraire.

Un transtypage automatique est entrepris lorsque l'expression d'une colonne ne correspond pas au type de donnée déclaré.

Des `INSERT` dans des tables pour lesquelles il manque des index d'unicité ne seront pas bloqués par des activités concurrentes. Les tables avec des index d'unicité pourraient bloquer si des sessions concurrentes réalisent des actions qui verrouillent ou modifient des lignes correspondant aux valeurs en cours d'insertion dans l'index ; les détails sont disponibles dans Section 64.5. `ON CONFLICT` peut être utilisé pour indiquer une action alternative lorsqu'une erreur sur une contrainte unique ou une contrainte d'exclusion est levée (voir Clause ON CONFLICT ci-dessous).

La clause RETURNING optionnelle fait que INSERT calcule et renvoie le(s) valeur(s) basée(s) sur chaque ligne en cours d'insertion (ou mises à jour si une clause ON CONFLICT DO UPDATE a été utilisée). C'est principalement utile pour obtenir les valeurs qui ont été fournies par défaut, comme un numéro de séquence. Néanmoins, toute expression utilisant les colonnes de la table est autorisée. La syntaxe de la liste RETURNING est identique à celle de la commande SELECT. Seules les lignes qui ont été insérées ou mises à jour avec succès sont retournées. Par exemple, si une ligne a été verrouillée mais non mise à jour parce que la *condition* de la clause ON CONFLICT DO UPDATE ... WHERE n'a pas été satisfaite, la ligne ne sera pas renvoyée.

Vous devez avoir le droit INSERT sur une table pour insérer des données dedans. Si ON CONFLICT DO UPDATE est indiqué, le droit UPDATE est aussi requis.

Si une liste de colonnes est indiquée, vous avez seulement besoin d'avoir le droit INSERT sur les colonnes spécifiées. De la même manière, lorsque ON CONFLICT DO UPDATE est indiqué, vous avez seulement besoin d'avoir le droit UPDATE sur les colonnes qui sont listées comme à mettre à jour. Cependant, toutes les formes de ON CONFLICT exigent également le droit SELECT sur toutes les colonnes dont les valeurs sont lues. Ceci inclut toute colonne mentionnée dans *cible_conflit* (ainsi que les colonnes référencées par la contrainte arbitre), et toute colonne mentionnée dans une clause ON CONFLICT DO UPDATE *expression*, ou dans une clause WHERE clause *condition*.

L'utilisation de la clause RETURNING requiert le droit SELECT sur toutes les colonnes mentionnées dans RETURNING. Si vous utilisez la clause *requête* pour insérer des lignes à partir d'une requête, vous avez bien sûr besoin d'avoir le droit SELECT sur toutes les tables ou colonnes référencées dans la requête.

Paramètres

Insertion

Cette section concerne les paramètres qui peuvent être utilisés lors de l'insertion de nouvelles lignes. Les paramètres *exclusivement* utilisés avec la clause ON CONFLICT sont décrits séparément.

requête_with

La clause WITH vous permet de spécifier une ou plusieurs sous-requêtes qui peuvent être référencées par leur nom dans la commande INSERT. Voir Section 7.8 et SELECT pour les détails.

Il est possible que la *requête* (commande SELECT) contienne également une clause WITH. Dans un tel cas, les deux ensembles de *requête_with* peuvent être référencés à l'intérieur de *requête*, mais le second prime dans la mesure où il est plus proche.

nom_table

Le nom (éventuellement préfixé du schéma) d'une table existante.

alias

Un nom de substitution pour *nom_table*. Lorsqu'un alias est indiqué, il masque complètement le nom actuel de la table. Ceci est particulièrement utile lorsque ON CONFLICT DO UPDATE fait référence à une table nommée excluded, puisque sinon ce nom serait utilisé pour le nom de la table spéciale représentant la ligne proposée à l'insertion.

nom_colonne

Le nom d'une colonne dans la table nommée par *nom_table*. Le nom de la colonne peut être qualifié avec un nom de sous-champ ou un indice de tableau, si besoin. (L'insertion uniquement dans certains champs d'une colonne composite positionne les autres champs à NULL.) Lorsque vous référencez une colonne avec ON CONFLICT DO UPDATE, n'incluez pas le nom de la table dans la spécification de la colonne. Par exemple, INSERT INTO nom_table ... ON

`CONFLICT DO UPDATE tab SET nom_table.col = 1` est invalide (ceci est conforme au comportement général pour la commande `UPDATE`).

`OVERRIDING SYSTEM VALUE`

Si cette clause est indiquée, alors toutes les valeurs fournies par les colonnes d'identité surchargeront les valeurs par défaut générées par les séquences.

Pour une colonne d'identité définie comme `GENERATED ALWAYS`, insérer une valeur explicite (autre que `DEFAULT`) sans préciser soit `OVERRIDING SYSTEM VALUE` soit `OVERRIDING USER VALUE` est considéré comme une erreur. (Pour une colonne d'identité définie comme `GENERATED BY DEFAULT`, `OVERRIDING SYSTEM VALUE` est le comportement normal et le préciser ne change rien mais PostgreSQL l'autorise comme extension.)

`OVERRIDING USER VALUE`

Si cette clause est spécifiée, alors toute valeur fournir pour les colonnes d'identité sont ignorées et les valeurs par défaut générée par la séquence sont appliquées.

Cette clause est utile par exemple lors de la copie de valeur entre des tables. Écrire `INSERT INTO tbl2 OVERRIDING USER VALUE SELECT * FROM tbl1` copiera de `tbl1` toutes les colonnes de `tbl2` qui ne sont pas des colonnes d'identité dans `tbl2` alors que des valeurs pour les colonnes d'identité dans `tbl2` seront générées par les séquences associées avec `tbl2`.

`DEFAULT VALUES`

Toutes les colonnes seront remplies avec leur valeur par défaut, comme si `DEFAULT` était indiqué explicitement pour chaque colonne. (Une clause `OVERRIDING` n'est pas permise dans cette forme.)

`expression`

Une expression ou valeur à assigner à la colonne correspondante.

`DEFAULT`

La colonne correspondante sera remplie avec sa valeur par défaut. Une colonne d'identité sera remplie avec une nouvelle valeur générée par la séquence associée. Pour une colonne générée, l'indiquer est permis mais ne fait que préciser le comportement normal du calcul de la colonne à partir de son expression de génération.

`requête`

Une requête (commande `SELECT`) qui fournit les lignes à insérer. Référez-vous à la commande SELECT pour une description de la syntaxe.

`expression_sortie`

Une expression à calculer et à retourner par la commande `INSERT` après que chaque ligne soit insérée ou mise à jour. L'expression peut utiliser n'importe quel nom de colonnes de la table nommée *`nom_table`*. Écrivez `*` pour renvoyer toutes les colonnes de(s) ligne(s) insérée(s) ou mise(s) à jour.

`nom_sortie`

Un nom à utiliser pour une colonne renvoyée.

Clause `ON CONFLICT`

La clause optionelle `ON CONFLICT` indique une action alternative lors d'une erreur de violation d'une contrainte unique ou d'exclusion. Pour chaque ligne individuelle proposée pour l'insertion, soit l'insertion est effectuée, soit si une contrainte *arbitrale* ou un index indiqué par *`cible_conflit`*

est violé, l'action alternative *cible_conflit* est effectuée. ON CONFLICT DO NOTHING évite simplement d'insérer une ligne comme action alternative. Comme action alternative, ON CONFLICT DO UPDATE met à jour la ligne existante en conflit avec la ligne proposée pour l'insertion.

cible_conflit peut effectuer une *inférence d'un index unique*. L'inférence consiste à indiquer un ou plusieurs *nom_colonne_index* et/ou *expression_index*. Tous les index uniques de *nom_table* qui, indépendamment de l'ordre, contiennent exactement les colonnes/ expressions *cible_conflit* spécifiées sont inférés (choisis) comme index arbitraux. Si un *predicat_index* est indiqué, il doit, comme une condition supplémentaire pour l'inférence, satisfaire les index arbitraux. Notez que cela signifie qu'un index unique non partiel (un index unique sans prédicat) sera inféré (et donc utilisé par ON CONFLICT) si un tel index remplissant l'ensemble des autres critères est disponible. Si une tentative d'inférence est impossible, une erreur est levée.

ON CONFLICT DO UPDATE garantit un traitement atomique de INSERT ou de UPDATE ; dans la mesure où il n'y a pas d'erreur indépendante, l'un de ces deux traitements est garanti, y compris en cas d'accès concurrents. Ceci est aussi connu sous le nom d'*UPSERT* (« UPDATE ou INSERT »).

cible_conflit

Indique les conflits ON CONFLICT entrainant l'action alternative en choisissant les *index arbitraux*. Soit effectue l'*inférence d'un index unique*, soit nomme une contrainte explicitement. Pour ON CONFLICT DO NOTHING, l'indication de *cible_conflit* est facultatif ; s'il est omis, les conflits avec toutes les contraintes utilisables (et index uniques) sont retenus. Pour ON CONFLICT DO UPDATE, *cible_conflit* *doit* être indiqué.

action_conflit

action_conflit indique une action alternative à ON CONFLICT. Elle peut être soit une clause DO NOTHING, soit une clause DO UPDATE indiquant le détail exact de l'action UPDATE à effectuer en cas de conflit. Les clauses SET et UPDATE dans ON CONFLICT DO UPDATE ont accès à la ligne existante en utilisant le nom de la table (ou un alias), et à la ligne proposée à l'insertion en utilisant la table spéciale de nom excluded. Le droit SELECT est requis sur l'ensemble des colonnes de la table cible où les colonnes correspondantes de excluded sont lues.

Notez que les effets de tous les trigegrs par ligne BEFORE INSERT sont reflétés dans les valeurs de excluded, dans la mesure où ces effets peuvent avoir contribués à la ligne exclue de l'insertion.

nom_colonne_index

Le nom d'une colonne de *nom_table*. Utilisé pour inférer les index arbitraux. Suit le format de CREATE INDEX. Le droit SELECT sur *nom_colonne_index* est nécessaire.

expression_index

Similaire à *nom_colonne_index*, mais utilisé pour inférer les expressions sur les colonnes de *nom_table* apparaissant dans les définitions de l'index (pas de simples colonnes). Suit le format de CREATE INDEX. Le droit SELECT sur toutes les colonnes apparaissant dans *expression_index* est nécessaire.

collation

Lorsque mentionné, indique que la colonne *nom_colonne_index* correspondante ou *expression_index* utilise une collation particulière pour être mis en correspondance durant l'inférence. Typiquement, ceci est omis, dans la mesure où les collations n'ont généralement pas d'incidence sur la survenu ou non d'une violation de contrainte. Suit le format de CREATE INDEX.

classe_operateur

Lorsque mentionné, elle indique que la colonne *nom_colonne_index* correspondante ou *expression_index* utilise une classe d'opérateurs en particulier pour être mis en

correspondance durant l'inférence. Typiquement, ceci est omis, dans la mesure où les sémantiques d'*égalité* sont souvent équivalentes entre les différents types de classes d'opérateurs, ou parce qu'il est suffisant de s'appuyer sur le fait que les définitions d'index uniques ont une définition pertinente de l'égalité. Suit le format de `CREATE INDEX`.

`predicat_index`

Utilisé pour permettre l'inférence d'index uniques partiels. Tous les index qui satisfont le prédicat (qui ne sont pas nécessairement des index partiels) peuvent être inférés. Suit le format de `CREATE INDEX`. Le droit `SELECT` sur toutes les colonnes apparaissant dans *`predicat_index`* est nécessaire.

`nom_contrainte`

Spécifie explicitement une *contrainte* arbitrale par nom, plutôt que d'inférer une contrainte par nom ou index.

`condition`

Une expression qui renvoie une valeur de type `boolean`. Seules les lignes pour lesquelles cette expression renvoie `true` seront mises à jour, bien que toutes les lignes seront verrouillées lorsque l'action `ON CONFLICT DO UPDATE` est prise. Notez que *`condition`* est évaluée en dernier, après qu'un conflit ait été identifié comme un candidat à la mise à jour.

Notez que les contraintes d'exclusion ne sont pas supportées comme arbitres avec `ON CONFLICT DO UPDATE`. Dans tous les cas, seules les contraintes `NOT DEFERRABLE` et les index uniques sont supportés comme arbitres.

La commande `INSERT` avec une clause `ON CONFLICT DO UPDATE` est une instruction déterministe. Ceci signifie que la commande ne sera pas autorisée à modifier n'importe quelle ligne individuelle plus d'une fois ; une erreur de violation de cardinalité sera levée si cette situation arrive. Les lignes proposées à l'insertion ne devraient pas avoir de duplication les unes par rapport aux autres relativement aux attributs contraints par un index arbitral ou une contrainte.

Notez qu'il n'y a pas de support d'une clause `ON CONFLICT DO UPDATE` d'un `INSERT` appliquée à une table partitionnée pour mettre à jour la clé de partitionnement d'une ligne en conflit qui causerait le déplacement de la ligne dans une nouvelle partition.

Astuce

Il est souvent préférable d'utiliser l'inférence d'un index unique plutôt que de nommer une contrainte directement en utilisant `ON CONFLICT ON CONSTRAINT` *`nom_contrainte`*. L'inférence continuera de fonctionner correctement lorsque l'index sous-jacent est remplacé par un autre plus ou moins équivalent de manière recouvrante, par exemple en utilisant `CREATE UNIQUE INDEX ... CONCURRENTLY` avant de supprimer l'index remplacé.

Avertissement

Alors que `CREATE INDEX CONCURRENTLY` ou `REINDEX CONCURRENTLY` s'exécute sur un index d'unicité, les instructions `INSERT ... ON CONFLICT` sur la même table peuvent échouer de façon inattendue sur une violation d'unicité.

Sorties

En cas de succès, la commande `INSERT` renvoie un code de la forme

```
INSERT oid nombre
```

nombre correspond au nombre de lignes insérées ou mises à jour. *oid* vaut toujours 0 (il s'agissait de l'OID affecté à la ligne insérée si *count* valait exactement 1 et que la table cible était déclarée WITH OIDS et 0 dans les autres cas, mais créer une table WITH OIDS n'est plus supporté).

Si la commande INSERT contient une clause RETURNING, le résultat sera similaire à celui d'une instruction SELECT contenant les colonnes et les valeurs définies dans la liste RETURNING, à partir de la liste des lignes insérées ou mises à jour par la commande.

Notes

Si la table spécifiée est une table partitionnée, chaque ligne est redirigée vers la partition appropriée et insérée dedans. Si la table spécifiée est une partition, une erreur sera remontée si une des lignes en entrée viole la contrainte de partition.

Vous pouvez aussi considérer l'utilisation de MERGE, vu qu'elle mixe des commandes INSERT, UPDATE et DELETE en une seule instruction. Voir MERGE.

Exemples

Insérer une ligne dans la table films :

```
INSERT INTO films
 VALUES ('UA502', 'Bananas', 105, '1971-07-13', 'Comédie', '82
 minutes');
```

Dans l'exemple suivant, la colonne longueur est omise et prend donc sa valeur par défaut :

```
INSERT INTO films (code, titre, did, date_prod, genre)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drame');
```

L'exemple suivant utilise la clause DEFAULT pour les colonnes date plutôt qu'une valeur précise :

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comédie', '82 minutes');
INSERT INTO films (code, titre, did, date_prod, genre)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drame');
```

Insérer une ligne constituée uniquement de valeurs par défaut :

```
INSERT INTO films DEFAULT VALUES;
```

Pour insérer plusieurs lignes en utilisant la syntaxe multi-lignes VALUES :

```
INSERT INTO films (code, titre, did, date_prod, genre) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

Insérer dans la table films des lignes extraites de la table tmp_films (la disposition des colonnes est identique dans les deux tables) :

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod <
 '2004-05-07';
```

Insérer dans des colonnes de type tableau :

```
-- Créer un jeu de 3 cases sur 3
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- Les indices de l'exemple ci-dessus ne sont pas vraiment
 nécessaires
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

Insérer une ligne simple dans la table `distributeurs`, en renvoyant le numéro de séquence généré par la clause `DEFAULT` :

```
INSERT INTO distributeurs (did, dnom) VALUES (DEFAULT, 'XYZ
 Widgets')
   RETURNING did;
```

Augmenter le nombre de ventes du vendeur qui gère le compte Acme Corporation, et enregistrer la ligne complètement mise à jour avec l'heure courante dans une table de traçage :

```
WITH upd AS (
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme
 Corporation')
    RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

Insérer ou mettre à jour de nouveaux distributeurs comme approprié. Suppose qu'un index unique a été défini qui contraint les valeurs apparaissant dans la colonne `did`. Notez que la table spéciale `excluded` est utilisée pour référencer les valeurs proposées à l'origine pour l'insertion :

```
INSERT INTO distributeurs (did, dnom)
    VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing,
 Inc')
    ON CONFLICT (did) DO UPDATE SET dnom = EXCLUDED.dnom;
```

Insérer un distributeur, ou ne fait rien pour les lignes proposées à l'insertion lorsqu'une ligne existante, exclue (une ligne avec une contrainte correspondante sur une ou plusieurs colonnes après que les triggers après ou avant se soient déclenchés) existe. L'exemple suppose qu'un index unique a été défini qui contraint les valeurs apparaissant dans la colonne `did` :

```
INSERT INTO distributeurs (did, dnom) VALUES (7, 'Redline GmbH')
    ON CONFLICT (did) DO NOTHING;
```

Insérer ou mettre à jour de nouveaux distributeurs comme approprié. L'exemple suppose qu'un index unique a été défini qui contraint les valeurs apparaissant dans la colonne `did`. La clause `WHERE` est utilisée pour limiter les lignes mises à jour (toutes les lignes existantes non mises à jour seront tout de même verrouillées) :

```
-- Ne pas mettre à jour les distributeurs existants avec un certain
 code postal
INSERT INTO distributeurs AS d (did, dnom) VALUES (8, 'Anvil
 Distribution')
    ON CONFLICT (did) DO UPDATE
    SET dnom = EXCLUDED.dnom || ' (précédemment ' || d.dnom || ')'
    WHERE d.code_postal <> '21201';

-- Nomme une contrainte directement dans l'instruction (utilise
-- l'index associé pour décider de prendre l'action DO NOTHING)
INSERT INTO distributeurs (did, dnom) VALUES (9, 'Antwerp Design')
    ON CONFLICT ON CONSTRAINT distributeurs_pkey DO NOTHING;
```

Insérer un nouveau distributeur si possible ; sinon `DO NOTHING`. L'exemple suppose qu'un index unique a été défini qui contraint les valeurs apparaissant dans la colonne `did` à un sous-ensemble des lignes où la colonne booléenne `est_actif` est évaluée à `true` :

```
-- Cette instruction pourrait inférer un index unique partiel sur
 "did"
-- avec un prédicat de type "WHERE est_actif", mais il pourrait
 aussi
-- juste utiliser une contrainte unique régulière sur "did"
INSERT INTO distributeurs (did, dnom) VALUES (10, 'Conrad
 International')
    ON CONFLICT (did) WHERE est_actif DO NOTHING;
```

Compatibilité

`INSERT` est conforme au standard SQL, sauf la clause `RETURNING` qui est une extension PostgreSQL, comme la possibilité d'utiliser la clause `WITH` avec l'instruction `INSERT`, et de spécifier une action alternative avec `ON CONFLICT`. Le standard n'autorise toutefois pas l'omission de la liste des noms de colonnes alors qu'une valeur n'est pas affectée à chaque colonne, que ce soit à l'aide de la clause `VALUES` ou à partir de la *`requête`*. Si vous préférez une instruction plus conforme au standard SQL que `ON CONFLICT`, voir MERGE.

Le standard SQL spécifie que `OVERRIDING SYSTEM VALUE` ne peut être spécifié que si une colonne d'identité qui est toujours générée existe. PostgreSQL autorise cette clause dans tous les cas et l'ignore si elle ne s'applique pas.

Les limitations possibles de la clause *`requête`* sont documentées sous SELECT.

LISTEN

LISTEN — Attendre une notification

Synopsis

```
LISTEN canal
```

Description

`LISTEN` enregistre la session courante comme listener du canal de notification *`canal`*. Si la session courante est déjà enregistrée comme listener de ce canal de notification, il ne se passe rien de plus.

À chaque appel de la commande `NOTIFY` *`canal`*, que ce soit par cette session ou par une autre connectée à la même base de données, toutes les sessions attendant sur ce canal en sont avisées et chacune en avise en retour son client. Voir `NOTIFY` pour plus d'informations.

La commande `UNLISTEN` permet d'annuler l'enregistrement d'une session comme listener d'un canal de notification. Les enregistrements d'écoute d'une session sont automatiquement effacés lorsque la session se termine.

La méthode utilisé par un client pour détecter les événements de notification dépend de l'interface de programmation PostgreSQL qu'il utilise. Avec la bibliothèque libpq, l'application exécute `LISTEN` comme une commande SQL ordinaire, puis appelle périodiquement la fonction `PQnotifies` pour savoir si un événement de notification est reçu. Les autres interfaces, telle libpgtcl, fournissent des méthodes de plus haut niveau pour gérer les événements de notification ; en fait, avec libpgtcl, le développeur de l'application n'a même pas à lancer `LISTEN` ou `UNLISTEN` directement. Tous les détails se trouvent dans la documentation de l'interface utilisée.

Paramètres

`canal`

Le nom d'un canal de notification (tout identifiant).

Notes

`LISTEN` prend effet à la validation de la transaction. Si `LISTEN` ou `UNLISTEN` est exécuté dans une transaction qui sera ensuite annulée, l'ensemble des canaux de notification écoutés sera inchangé.

Une transaction qui a exécuté `LISTEN` ne peut pas être préparée pour la validation en deux phases.

Il existe une fenêtre de vulnérabilité lors de la mise en place d'une session d'écoute : si des transactions validant en concurrence envoient des notifications, quels sont celles que la nouvelle session en écoute va recevoir ? La réponse est que la session recevra tous les événements validés après un instant lors de l'étape de validation de la transaction. Mais c'est légèrement plus tard que n'importe quel état de base de données qu'aurait pu observer une transaction avec des requêtes. Cela conduit à la règle suivante pour l'utilisation de `LISTEN` : exécutez d'abord (et validez !) cette commande, puis dans une nouvelle transaction, inspectez l'état de la base de données selon les besoins de la logique de l'application. Ensuite, utilisez les notifications pour connaître les modifications ultérieures de l'état de la base de données. Les premières notifications reçues peuvent faire référence à des mises à jour déjà constatées lors de l'inspection initiale de la base de données, mais cela est généralement sans conséquence.

NOTIFY contient une discussion plus étendue sur l'utilisation de `LISTEN` et `NOTIFY`.

Exemples

Configurer et exécuter une séquence listen/notify à partir de psql :

```
LISTEN virtual;
NOTIFY virtual;
Notification asynchrone "virtual" reçue en provenance du processus
 serveur de PID 8448.
```

Compatibilité

Il n'existe pas d'instruction `LISTEN` dans le standard SQL.

Voir aussi

NOTIFY, UNLISTEN

LOAD

LOAD — Charger une bibliothèque partagée

Synopsis

```
LOAD 'fichier'
```

Description

Cette commande charge une bibliothèque partagée dans l'espace d'adressage de PostgreSQL. Si le fichier a déjà été chargé, la commande ne fait rien. Les fichiers des bibliothèques partagées contenant des fonctions C sont automatiquement chargés à chaque fois qu'une de leur fonctions est appelée. Du coup, un appel explicite à `LOAD` est habituellement seulement nécessaire pour charger une bibliothèque qui modifie le comportement du serveur via des « points d'accroche » plutôt qu'en fournissant un ensemble de fonctions.

Le nom du fichier de la bibliothèque est typiquement donné sous la forme d'un simple nom de fichier, qui est cherché dans le chemin de recherches des bibliothèques du serveur (configuré avec dynamic_library_path). Il peut aussi être donné sous la forme d'un nom complet. Quelque soit le cas, l'extension du nom de fichier pour les bibliothèques partagées de la plateforme peut être omise. Voir Section 38.10.1 pour plus d'informations sur ce sujet.

Les utilisateurs normaux peuvent seulement utiliser `LOAD` avec des bibliothèques situées dans `$libdir/plugins/` -- le *`nom_fichier`* indiqué doit commencer avec cette chaîne exacte. (Il est de la responsabilité de l'administrateur de bases de données de s'assurer que seules des bibliothèques « sûres » y sont installées.)

Compatibilité

`LOAD` est une extension PostgreSQL.

Voir aussi

CREATE FUNCTION

LOCK

LOCK — verrouiller une table

Synopsis

```
LOCK [ TABLE ] [ ONLY ] nom [ * ] [, ...] [ IN mode_verrou MODE ]
 [ NOWAIT ]

où mode_verrou peut être :

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE
 EXCLUSIVE | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS
 EXCLUSIVE
```

Description

`LOCK TABLE` prend un verrou de niveau table, attendant si nécessaire que tout verrou conflictuel soit relâché. Si `NOWAIT` est spécifié, `LOCK TABLE` n'attend pas l'acquisition du verrou désiré : s'il ne peut pas être obtenu immédiatement, la commande est annulée et une erreur est émise. Une fois obtenu, le verrou est conservé jusqu'à la fin de la transaction en cours. (Il n'y a pas de commande `UNLOCK TABLE` ; les verrous sont systématiquement relâchés à la fin de la transaction.)

Quand une vue est verrouillée, toutes les relations apparaissant dans la requête de définition de la vue sont aussi verrouillée récursivement avec le même mode de verrou.

Lors de l'acquisition automatique de verrous pour les commandes qui référencent des tables, PostgreSQL utilise toujours le mode de verrou le moins restrictif possible. `LOCK TABLE` est utilisable lorsqu'il est nécessaire d'obtenir des verrous plus restrictifs.

Soit, par exemple, une application qui exécute une transaction de niveau d'isolation `READ COMMITTED`. Pour s'assurer que les données de la table sont immuables pendant toute la durée de la transaction, un verrou `SHARE` de niveau table peut être obtenu avant d'effectuer la requête. Cela empêche toute modification concurrente des données. Cela assure également que toute lecture intervenant ensuite sur la table accède à la même vue des données validées. En effet, un verrou `SHARE` entre en conflit avec le verrou `ROW EXCLUSIVE` pris par les modificateurs et l'instruction `LOCK TABLE` *`nom`* `IN SHARE MODE` attend que tout détenteur concurrent de verrous de mode `ROW EXCLUSIVE` valide ou annule. De ce fait, une fois le verrou obtenu, il ne reste aucune écriture non validée en attente ; de plus, aucune ne peut commencer tant que le verrou acquis n'est pas relâché.

Pour obtenir un effet similaire lors de l'exécution d'une transaction de niveau d'isolation `REPEATABLE READ` ou `SERIALIZABLE`, il est nécessaire d'exécuter l'instruction `LOCK TABLE` avant toute instruction `SELECT` ou de modification de données. La vue des données utilisée par une transaction `REPEATABLE READ` or `SERIALIZABLE` est figée au moment où débute la première instruction `SELECT` ou de modification des données. Un `LOCK TABLE` ultérieur empêche encore les écritures concurrentes -- mais il n'assure pas que la transaction lit les dernières données validées.

Si une telle transaction modifie les données de la table, elle doit utiliser le mode de verrou `SHARE ROW EXCLUSIVE` au lieu du mode `SHARE`. Cela assure l'exécution d'une seule transaction de ce type à la fois. Sans cela, une situation de verrou mort est possible : deux transactions peuvent acquérir le mode `SHARE` et être ensuite incapables d'acquérir aussi le mode `ROW EXCLUSIVE` pour réellement effectuer leurs mises à jour. (Les verrous d'une transaction ne sont jamais en conflit. Une transaction peut de ce fait acquérir le mode `ROW EXCLUSIVE` alors qu'elle détient le mode `SHARE` -- mais pas si une autre transaction détient le mode `SHARE`.) Pour éviter les verrous bloquants, il est préférable que toutes les transactions qui acquièrent des verrous sur les mêmes objets le fassent dans le même ordre.

De plus si de multiples modes de verrous sont impliqués pour un même objet, le verrou de mode le plus restrictif doit être acquis le premier.

Plus d'informations sur les modes de verrou et les stratégies de verrouillage sont disponibles dans Section 13.3.

Paramètres

nom

Le nom d'une table à verrouiller (éventuellement qualifié du nom du schéma). Si `ONLY` est précisé avant le nom de la table, seule cette table est verrouillée. Dans le cas contraire, la table et toutes ses tables filles (si elle en a) sont verrouillées. En option, `*` peut être placé après le nom de la table pour indiquer explicitement que les tables filles sont inclues.

La commande `LOCK a, b;` est équivalente à `LOCK a; LOCK b;`. Les tables sont verrouillées une par une dans l'ordre précisé par la commande `LOCK TABLE`.

modeverrou

Le mode de verrou précise les verrous avec lesquels ce verrou entre en conflit. Les modes de verrou sont décrits dans Section 13.3.

Si aucun mode de verrou n'est précisé, `ACCESS EXCLUSIVE`, mode le plus restrictif, est utilisé.

`NOWAIT`

`LOCK TABLE` n'attend pas que les verrous conflictuels soient relâchés : si le verrou indiqué ne peut être acquis immédiatement sans attente, la transaction est annulée.

Notes

`LOCK TABLE ... IN ACCESS SHARE MODE` requiert les droits `SELECT` sur la table cible. `LOCK TABLE ... IN ROW EXCLUSIVE MODE` requiert des droits `INSERT`, `UPDATE`, `DELETE`, ou `TRUNCATE` sur la table cible. Toutes les autres formes de `LOCK` requièrent au moins un des droits `UPDATE`, `DELETE` et `TRUNCATE` au niveau table.

L'utilisateur réalisant un verrou sur la vue doit avoir le droit correspondant sur la vue. De plus, par défaut, le propriétaire de la vue doit avoir les droits correspondants sur les relations de base sous-jacentes mais l'utilisateur réalisant le verrou n'a pas besoin de ces droits. Néanmoins, si la vue a l'option `security_invoker` configurée à `true` (voir `CREATE VIEW`), l'utilisateur réalisant le verrou, plutôt que le propriétaire de la vue, doit avoir les droits adéquats pour les relations sous-jacentes.

`LOCK TABLE` est inutile à l'extérieur d'un bloc de transaction : le verrou est détenu jusqu'à la fin de l'instruction. Du coup, PostgreSQL renvoie une erreur si `LOCK` est utilisé en dehors d'un bloc de transaction. Utilisez `BEGIN` et `COMMIT` (ou `ROLLBACK`) pour définir un bloc de transaction.

`LOCK TABLE` ne concernent que les verrous de niveau table. Les noms de mode contenant `ROW` sont donc tous mal nommés. Ces noms de modes doivent généralement être compris comme indiquant l'intention de l'utilisateur d'acquérir des verrous de niveau ligne à l'intérieur de la table verrouillée. Le mode `ROW EXCLUSIVE` est également un verrou de table partageable. Tous les modes de verrou ont des sémantiques identiques en ce qui concerne `LOCK TABLE` ; ils ne diffèrent que dans les règles de conflit entre les modes. Pour des informations sur la façon d'acquérir un vrai verrou de niveau ligne, voir Section 13.3.2 et Clause de verrouillage dans la documentation de référence de SELECT.

Exemples

Obtenir un verrou `SHARE` sur une table avec clé primaire avant de réaliser des insertions dans une table disposant de la clé étrangère :

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE nom = 'Star Wars : Episode I - La menace fantôme';
-- Effectuer un ROLLBACK si aucun enregistrement n'est retourné
INSERT INTO commentaires_films VALUES
    (_id_, 'SUPER ! Je l''attendais depuis si longtemps !');
COMMIT WORK;
```

Prendre un verrou `SHARE ROW EXCLUSIVE` sur une table avec clé primaire lors du début des opérations de suppression :

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM commentaires_films WHERE id IN
    (SELECT id FROM films WHERE score < 5);
DELETE FROM films WHERE score < 5;
COMMIT WORK;
```

Compatibilité

`LOCK TABLE` n'existe pas dans le standard SQL. À la place, il utilise `SET TRANSACTION` pour spécifier les niveaux de concurrence entre transactions. PostgreSQL en dispose également ; voir SET TRANSACTION pour les détails.

À l'exception des modes de verrous `ACCESS SHARE`, `ACCESS EXCLUSIVE` et `SHARE UPDATE EXCLUSIVE`, les modes de verrou PostgreSQL et la syntaxe `LOCK TABLE` sont compatibles avec ceux présents dans Oracle.

MERGE

MERGE — Insère, met à jour ou efface des lignes d'une table selon une condition

Synopsis

```
[ WITH clause_with [, ...] ]
MERGE INTO [ ONLY ] nom_table_cible [ * ] [ [ AS ] alias_cible ]
    USING source_des_données ON condition_de_jointure
    clause_when [...]

où source_des_données est :

    { [ ONLY ] nom_table_source [ * ] | ( requête_source ) }
 [ [ AS ] alias_source ]

et clause_when est :

    { WHEN MATCHED [ AND condition ] THEN { merge_update
 | merge_delete | DO NOTHING } |
      WHEN NOT MATCHED [ AND condition ] THEN { merge_insert | DO
 NOTHING } }

et merge_insert est :

    INSERT [( nom_colonne [, ...] )]
        [ OVERRIDING { SYSTEM | USER } VALUE ]
        { VALUES ( { expression | DEFAULT } [, ...] ) | DEFAULT
 VALUES }

et merge_update est :

    UPDATE SET { nom_de_colonne = { expression | DEFAULT } |
                 ( nom_de_colonne [, ...] ) = [ ROW ]
 ( { expression | DEFAULT } [, ...] ) |
                 ( nom_de_colonne [, ...] ) = ( sous-SELECT )
               } [, ...]

et merge_delete est :

    DELETE
```

Description

`MERGE` exécute des actions qui modifient des lignes dans la table cible identifiée par *`nom_table_cible`*, à partir d'une *`source_des_données`*. `MERGE` est un ordre SQL qui, à lui seul et selon la condition, peut opérer un `INSERT`, un `UPDATE` ou un `DELETE` sur des lignes, une tâche qui nécessiterait sinon de multiples appels en langage procédural.

Tout d'abord, la commande `MERGE` fait une jointure entre *`source_des_données`* et la table cible, ce qui donne zéro ou plus ligne(s) candidates au changement. Pour chacune, on définit le statut `MATCHED` ou `NOT MATCHED` une fois pour toutes ; puis les clauses `WHEN` sont évaluées dans l'ordre spécifié. Pour chaque ligne candidate à un changement, la première clause évaluée à true est exécutée. Au plus une clause `WHEN` est exécutée pour chaque ligne candidate au changement.

Les opérations de MERGE ont le même effet que les commandes UPDATE, INSERT, ou DELETE de même nom. La syntaxe de ces commandes diffère, notamment sur l'absence de clause WHERE et de nom de table. Toutes les actions opèrent sur la table cible, bien que des modifications sur d'autres tables puissent être faites en passant par des triggers.

Si DO NOTHING est précisé, la ligne source est passée. Comme les actions sont évaluées dans l'ordre spécifié, il peut être pratique de sauter les lignes source sans intérêt avant une gestion plus fine.

Il n'y a pas de droit MERGE dédié. Si vous spécifiez une mise à jour, vous devez posséder le droit UPDATE sur les colonnes de la table cible mentionnées dans la clause SET. Pour une insertion, vous devez posséder le droit INSERT sur la table cible. Et pour une suppression, vous devez posséder le droit DELETE sur la table cible. Si vous ajoutez une action DO NOTHING, vous devrez avoir le droit SELECT sur au moins une colonne de la table cible. Vous aurez aussi besoin du droit SELECT sur toute colonne de *source_des_données* et sur toute colonne de la table cible référencée dans toute condition (incluant condition_jointure) ou expression. Les droits sont testés et vérifiés une fois au démarrage de l'ordre, que les clauses WHEN concernées soient exécutées ou pas.

MERGE n'est pas supporté si la table cible est une vue matérialisée, une table étrangère, ou si des règles y ont été posées.

Paramètres

requête_with

La clause WITH vous permet d'ajouter une ou plusieurs sous-requêtes pouvant être référencées dans la requête MERGE. Voir Section 7.8 et SELECT pour des détails. Notez que WITH RECURSIVE n'est pas supporté par MERGE.

nom_table_cible

Nom (éventuellement qualifié du schéma) de la table cible où fusionner des informations. Si ONLY est indiqué avant le nom de la table, les lignes correspondantes sont mises à jour ou supprimées uniquement dans la table nommée. Si ONLY n'est pas indiquée, les lignes correspondantes sont aussi mises à jour ou supprimées des tables héritant de la table nommée. * peut être indiquée en option après le nom de la table pour indiquer explicitement que les tables enfants sont inclus. Le mot clé ONLY et l'option * n'affectent pas les actions d'insertion, qui insèrent toujours uniquement dans la table nommée.

alias_cible

Un nom à substituer à la table cible. Quand un alias est fourni, il cache complètement le nom réel de la table. Par exemple, avec MERGE INTO foo AS f, le reste de l'ordre MERGE devra appeler la table avec f et non foo.

nom_table_source

Le nom (éventuellement qualifié du schéma) de la table, vue ou table de transition source. Si ONLY est précisé avant le nom de la table, les lignes correspondantes sont inclus uniquement à partir de la table nommée. Si ONLY n'est pas indiqué, les lignes correspondantes sont aussi incluses à partir des tables héritant de la table nommée. En option, * peut être ajouté après le nom de la table pour indiquer explicitement que les tables enfants sont incluses.

requête_source

Une requête (commande SELECT ou VALUES) fournissant les lignes à fusionner dans la table cible. Se référer à la documentation de SELECT ou VALUES pour une description des syntaxes.

alias_source

Un nom substitut de la source de données. Quand un alias est fourni, il cache complètement le nom réel de la table ou le fait qu'une requête ait été fournie.

condition_de_jointure

condition_de_jointure est une expression dont le résultat est un `boolean` (similaire à une clause `WHERE`), lequel indique quelles lignes de *source_des_données* correspondent à des lignes dans la table cible.

> **Avertissement**
>
> Seules les colonnes de la table cible que l'on tente de rapprocher de *source_des_données* devraient apparaître dans *condition_de_jointure*. Les sous-expressions de la condition qui ne référencent que la table cible peuvent affecter le choix de l'action, souvent de manière surprenante.

clause_when

Au moins une clause `WHEN` est nécessaire.

Si la clause `WHEN` indique `WHEN MATCHED` et que la ligne candidate correspond à une ligne de la table cible, et si *condition* est absente ou évaluée à `true`, alors la clause `WHEN` est exécutée.

À l'inverse, si la clause `WHEN` indique `WHEN NOT MATCHED`, et que la ligne candidate n'a pas de ligne correspondante dans la table cible, et si *condition* est absente ou évaluée à `true`, alors la clause `WHEN` est exécutée.

condition

Une expression dont le résultat est un `boolean`. Si cette expression renvoie `true`, alors la clause `WHEN` qu'elle accompagne est exécutée pour la ligne en cours.

Une condition d'une clause `WHEN MATCHED` peut se référer à des colonnes de la relation source comme de la cible. Une condition d'une clause `WHEN NOT MATCHED` ne peut se référer qu'à des colonnes de la relation source, puisque, par définition, il n'y a pas de ligne correspondante dans la cible. Seuls les attributs système de la table cible sont accessibles.

merge_insert

Spécification de la clause `INSERT` qui insère une ou plusieurs lignes dans la table cible. Les noms des colonnes cibles peuvent être listés dans n'importe quel ordre. Si aucune liste n'est donnée, le défaut est l'ensemble des colonnes de la table dans leur ordre de déclaration.

Chaque colonne absente de la liste des champs, explicite ou implicite, sera remplie par une valeur par défaut, soit sa valeur par défaut déclarée, soit NULL.

Si la table cible est une table partitionnée, chaque ligne est routée vers la partition appropriée et insérée dedans. Si la table cible est une partition, une erreur se produira si une ligne quelconque viole la contrainte de partitionnement.

Les noms de colonnes ne peuvent être spécifiés qu'une fois. Les commandes `INSERT` ne peuvent pas contenir des sous-SELECT.

Une seule clause `VALUES` peut être spécifiée. Elle ne peut se référer qu'à des colonnes de la table source, puisque par définition il n'y a pas de ligne cible correspondante.

merge_update

Spécification de la commande `UPDATE` qui met à jour la ligne courante de la table cible. Les noms de colonnes ne peuvent être utilisés plus d'une fois.

Ne sont permis ni un nom de table, ni une clause `WHERE`.

merge_delete

Spécifie une action DELETE qui efface la ligne courante de la table cible. N'incluez ni le nom de la table ni aucune autre clause, comme vous le feriez pour une commande DELETE.

nom_de_colonne

Le nom d'une colonne dans la table cible. Au besoin, il peut être complété avec un nom de sous-champ ou un indice de tableau. (N'insérer que dans certains champs d'une colonne composée laisse les autres champs à NULL). N'incluez pas le nom de la table dans la définition d'une colonne cible.

OVERRIDING SYSTEM VALUE

Sans cette clause, une erreur sera levée si une valeur explicite (autre que DEFAULT) est spécifiée pour une colonne identité définie comme GENERATED ALWAYS. Cette clause passe outre à la restriction.

OVERRIDING USER VALUE

Avec cette clause, toutes les valeurs fournies pour des colonnes identité définies comme GENERATED BY DEFAULT sont ignorées, et les valeurs générées par séquence sont appliquées.

DEFAULT VALUES

Toutes les colonnes seront remplies avec leur valeur par défaut. (Une clause OVERRIDING n'est alors pas permise.)

expression

Une expression à assigner à une colonne. Au sein d'une clause WHEN MATCHED, elle peut utiliser des valeurs de la ligne originale dans la table cible, et des valeurs de la ligne issue de *source_des_données*. Dans une clause WHEN NOT MATCHED, l'expression peut utiliser des valeurs de *source_des_données*.

DEFAULT

Positionne une colonne à sa valeur par défaut (qui sera NULL si aucune n'a été définie).

sous-SELECT

Une sous-requête SELECT qui produit autant de colonnes en sortie que celles listées dans la liste de colonnes entre parenthèses la précédent. La sous-requête ne doit pas renvoyer plus d'une ligne quand elle est exécutée. Si elle renvoie une ligne, la valeur de ses colonnes est affectée aux colonnes cibles ; si elle ne renvoie aucune ligne, des valeurs NULL sont affectées aux colonnes cibles. La sous-requête peut faire référence aux valeurs de la ligne originale dans la table cible, et aux valeurs provenant de la ligne *source_de_donnees*.

Résultats

Si elle termine avec succès, la commande MERGE renvoie un résultat de la forme :

MERGE *nombre_total*

nombre_total est le nombre total des lignes changées (qu'elles aient été insérées, mises à jour ou effacées). Si *nombre_total* vaut 0, aucune ligne n'a été changée de quelque manière que ce soit.

Notes

Les étapes suivantes se déroulent pendant l'exécution de MERGE.

1. Exécute tous les triggers BEFORE STATEMENT de chaque action spécifiée, que leur clause WHEN renvoie quelque chose ou non.

2. Fait une jointure entre la source et la table cible. La requête résultante sera optimisée comme à l'habitude, et produira un ensemble de lignes candidates au changement. Pour chacune :

   a. Évalue pour chaque ligne si elle a une correspondance (MATCHED) ou pas (NOT MATCHED).

   b. Teste chaque condition WHEN dans l'ordre spécifié jusqu'à ce qu'une renvoie true.

   c. Quand une condition renvoie true, exécute les actions suivantes :

      i. exécute tous les triggers BEFORE ROW qui se déclenchent sur ce type d'action ;

      ii. exécute l'action spécifiée, en vérifiant toutes les contraintes sur la table cible ;

      iii. exécute tous les triggers AFTER ROW qui se déclenchent sur ce type d'action.

3. Exécute tous les triggers AFTER STATEMENT pour les actions spécifiées, qu'elles se soient ou non réellement produites. C'est équivalent à un ordre UPDATE qui ne modifie aucune ligne.

En résumé, les triggers de niveau instruction (par exemple INSERT) seront déclenchés à chaque *spécification* d'une action de leur genre. Par contre, les triggers de niveau ligne se déclencheront uniquement pour les types d'événements *exécutés*. Ainsi un ordre MERGE peut déclencher à la fois des triggers sur instruction UPDATE et sur INSERT, même si seuls des triggers de ligne sur UPDATE sont déclenchés.

Vous devez vous assurer que la jointure produit au plus une ligne candidate pour chaque ligne de la source. En d'autres termes, une ligne cible ne devrait pas être jointe à plus d'une ligne de données source. Si cela devait arriver, une seule ligne parmi les candidates sera utilisée pour modifier la ligne cible ; une erreur sera levée lors des tentatives suivantes de modification de la ligne. Cela peut arriver si des triggers sur ligne modifient la table cible et que les lignes modifiées sont par la suite aussi modifiées par MERGE. Si l'action répétée est un INSERT, cela provoquera une violation d'unicité, alors que des ordres UPDATE ou DELETE répétés causeront une violation de cardinalité ; ce dernier comportement est exigé par le standard SQL. Cela diffère du comportement historique de PostgreSQL dans les jointures au sein d'ordres UPDATE et DELETE, où le deuxième essai de modification d'une même ligne est simplement ignoré, tout comme les suivants.

Si une clause WHEN omet une sous-clause AND, elle devient la clause finale atteinte pour ce type (MATCHED ou NOT MATCHED). Si une autre clause WHEN est spécifiée pour le même type, elle serait probablement inatteignable et une erreur est levée. Si aucune clause finale n'est spécifiée pour un type, il est possible qu'aucune action ne soit exécutée pour la ligne candidate.

L'ordre dans lequel les lignes seront générées depuis la source des données est par défaut indéterminé. Au besoin, une *requête_source* peut être utilisée pour définir un ordre cohérent, qui pourrait être nécessaire pour éviter des verrous mutuels entre transactions simultanées.

Il n'y a pas de clause RETURNING à une commande MERGE. Les action INSERT, UPDATE et DELETE ne peuvent contenir de clauses RETURNING ou WITH.

Quand MERGE est lancé en parallèle à d'autres commandes qui modifient la table cible, les règles usuelles d'isolation des transactions s'appliquent ; voir Section 13.2 pour une explication du comportement de chaque niveau d'isolation. Comme alternative permettant de lancer un ordre UPDATE en même temps que se déroule un INSERT, vous pouvez considérer INSERT ... ON CONFLICT. Il y a beaucoup de différences entre les deux ordres, et leurs limites diffèrent, si bien qu'ils ne sont pas interchangeables.

Exemples

Effectue une maintenance sur customer_accounts à partir de nouvelles recent_transactions.

```
MERGE INTO customer_account ca
USING recent_transactions t
ON t.customer_id = ca.customer_id
WHEN MATCHED THEN
  UPDATE SET balance = balance + transaction_value
WHEN NOT MATCHED THEN
  INSERT (customer_id, balance)
  VALUES (t.customer_id, t.transaction_value);
```

Notez que c'est exactement la même chose que l'ordre suivant, car le résultat du `MATCHED` ne change pas pendant l'exécution.

```
MERGE INTO customer_account ca
USING (SELECT customer_id, transaction_value FROM
 recent_transactions) AS t
ON t.customer_id = ca.customer_id
WHEN MATCHED THEN
  UPDATE SET balance = balance + transaction_value
WHEN NOT MATCHED THEN
  INSERT (customer_id, balance)
  VALUES (t.customer_id, t.transaction_value);
```

Tente d'insérer un nouvel article dans un stock avec sa quantité. Si l'article existe déjà, met plutôt à jour son stock existant. Interdit les entrées avec un stock à zéro.

```
MERGE INTO wines w
USING wine_stock_changes s
ON s.winename = w.winename
WHEN NOT MATCHED AND s.stock_delta > 0 THEN
  INSERT VALUES(s.winename, s.stock_delta)
WHEN MATCHED AND w.stock + s.stock_delta > 0 THEN
  UPDATE SET stock = w.stock + s.stock_delta
WHEN MATCHED THEN
  DELETE;
```

La table `wine_stock_changes` peut être, par exemple, une table temporaire récemment chargée dans la base.

Compatibilité

Cette commande se conforme au standard SQL.

La clause `WITH` et l'action `DO NOTHING` sont des extensions au standard SQL.

MOVE

MOVE — positionner un curseur

Synopsis

```
MOVE [ direction ] [ FROM | IN ] nom_curseur

où direction fait partie
de :

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE nombre
    RELATIVE nombre
    nombre
    ALL
    FORWARD
    FORWARD nombre
    FORWARD ALL
    BACKWARD
    BACKWARD nombre
    BACKWARD ALL
```

Description

`MOVE` repositionne un curseur sans retourner de donnée. `MOVE` fonctionne exactement comme la commande `FETCH` à la différence que `MOVE` ne fait que positionner le curseur et ne retourne aucune ligne.

Les paramètres de la commande `MOVE` sont identiques à ceux de la commande `FETCH`. FETCH contient les détails de syntaxe et d'utilisation.

Sortie

En cas de réussite, une commande `MOVE` retourne une balise de commande de la forme

```
MOVE compteur
```

`compteur` est le nombre de lignes qu'une commande `FETCH` avec les mêmes paramètres aurait renvoyée (éventuellement zéro).

Exemples

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- Saute les 5 premières lignes :
MOVE FORWARD 5 IN liahona;
MOVE 5
```

```
-- Récupère la 6ème ligne à partir du curseur liahona :
FETCH 1 FROM liahona;
 code  | titre  | did | date_prod  | genre  |  longueur
-------+--------+-----+------------+--------+-----------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- Ferme le curseur liahona et termine la transaction :
CLOSE liahona;
COMMIT WORK;
```

Compatibilité

Il n'existe pas d'instruction `MOVE` dans le standard SQL.

Voir aussi

CLOSE, DECLARE, FETCH

NOTIFY

NOTIFY — engendrer une notification

Synopsis

```
NOTIFY canal [ , charge ]
```

Description

La commande NOTIFY envoie une notification avec une chaîne de « charge » supplémentaire à chaque application cliente qui a exécuté précédemment la commande LISTEN *canal* dans la base de données courante pour le nom du canal indiqué. Les notifications sont visibles par tous les utilisateurs.

NOTIFY fournit un mécanisme simple de communication interprocessus pour tout ensemble de processus accédant à la même base de données PostgreSQL. Une chaîne de charge peut être envoyée avec la notification, et des mécanismes de plus haut niveau permettant de passer des données structurées peuvent être construits en utilisant les tables de la base de données.

L'information passée au client pour une notification inclut le nom de la notification et le PID du processus serveur de la session le notifiant.

C'est au concepteur de la base de données de définir les noms de notification utilisés dans une base de données précise et la signification de chacun. Habituellement, le nom du canal correspond au nom d'une table dans la base de données. L'événement notify signifie essentiellement « J'ai modifié cette table, jetez-y un œil pour vérifier ce qu'il y a de nouveau ». Mais cette association n'est pas contrôlée par les commandes NOTIFY et LISTEN. Un concepteur de bases de données peut, par exemple, utiliser plusieurs noms de canal différents pour signaler différentes sortes de modifications au sein d'une même table. Sinon, la chaîne de charge peut être utilisée pour différencier plusieurs cas.

Lorsque NOTIFY est utilisé pour signaler des modifications sur une table particulière, une technique de programmation utile est de placer le NOTIFY dans un trigger sur instruction déclenchée par les mises à jour de la table. De cette façon, la notification est automatique lors d'une modification de la table et le programmeur de l'application ne peut accidentellement oublier de le faire.

NOTIFY interagit fortement avec les transactions SQL. Primo, si un NOTIFY est exécuté à l'intérieur d'une transaction, les événements notify ne sont pas délivrés avant que la transaction ne soit validée, et à cette condition uniquement. En effet, si la transaction est annulée, les commandes qu'elle contient n'ont aucun effet, y compris NOTIFY. Cela peut toutefois s'avérer déconcertant pour quiconque s'attend à une délivrance immédiate des notifications.

Secondo, si une session à l'écoute reçoit un signal de notification alors qu'une transaction y est active, la notification n'est pas délivrée au client connecté avant la fin de cette transaction (par validation ou annulation). Là encore, si une notification est délivrée à l'intérieur d'une transaction finalement annulée, on pourrait espérer annuler cette notification par quelque moyen -- mais le serveur ne peut pas « reprendre » une notification déjà envoyée au client. C'est pourquoi les notifications ne sont délivrés qu'entre les transactions. Il est, de ce fait, important que les applications qui utilisent NOTIFY pour l'envoi de signaux en temps réel conservent des transactions courtes.

Si le même nom de canal est signalé plusieurs fois à partir de la même transaction avec des chaînes de charge identiques, seule une instance de la notification est délivrée aux processus en écoute. Par contre, les notifications avec des chaînes de charges distinctes seront toujours délivrées par des notifications distinctes. De façon similaire, les notifications provenant de différentes transactions ne seront jamais regroupées en une seule notification. Sauf pour supprimer des instances ultérieures de notifications dupliquées, la commande NOTIFY garantie que les notifications de la même transaction

seront délivrées dans l'ordre où elles ont été envoyées. Il est aussi garantie que les messages de transactions différentes seront délivrées dans l'ordre dans lequel les transactions ont été validées.

Il est courant qu'un client qui exécute `NOTIFY` écoute lui-même des notifications de même canal. Dans ce cas, il récupère une notification, comme toutes les autres sessions en écoute. Suivant la logique de l'application, cela peut engendre un travail inutile, par exemple lire une table de la base de données pour trouver les mises à jour que cette session a elle-même écrites. Il est possible d'éviter ce travail supplémentaire en vérifiant si le PID du processus serveur de la session notifiante (fourni dans le message d'événement de la notification) est le même que le PID de la session courante (disponible à partir de libpq). S'ils sont identiques, la notification est le retour du travail actuel et peut être ignorée.

Paramètres

canal

Nom du canal à signaler (identifiant quelconque).

charge

La chaîne de « charge » à communiquer avec la notification. Elle doit être spécifiée comme une chaîne litérale. Dans la configuration par défaut, elle doit avoir une taille inférieure à 8000 octets. (Si des données binaires ou de tailles plus importantes doivent être communiquées, il est mieux de les placer dans une table de la base et d'envoyer la clé correspondant à l'enregistrement.)

Notes

Il existe une queue qui récupère les notifications qui ont été envoyées mais pas encore traitées par les sessions en écoute. Si la queue est remplie, les transactions appelant `NOTIFY` échoueront à la validation. La queue est assez large (8 Go dans une installation standard) et devrait être suffisamment bien taillée dans la majorité des cas. Néanmoins, aucun nettoyage ne peut se faire si une session exécute `LISTEN` puis entre en transaction pendant une longue période. Une fois qu'une queue est à moitié pleine, des messages d'avertissements seront envoyés dans les traces indiquant la session qui empêche le nettoyage. Dans ce cas, il faut s'assurer que la session termine sa transaction en cours pour que le nettoyage puisse se faire.

La fonction `pg_notification_queue_usage` renvoie la fraction de queue actuellement occupée par des notifications en attente. Voir Section 9.26 pour plus d'informations.

Une transaction qui a exécuté `NOTIFY` ne peut pas être préparée pour une validation en deux phases.

pg_notify

Pour envoyer une notification, vous pouvez aussi utiliser la fonction `pg_notify(text, text)`. La fonction prend en premier argument le nom du canal et en second la charge. La fonction est bien plus simple à utiliser que la commande `NOTIFY` si vous avez besoin de travailler avec des noms de canaux et des charges non constants.

Exemples

Configurer et exécuter une séquence listen/notify à partir de psql :

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process
 with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the
 payload" received from server process with PID 8448.
```

```
LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received
 from server process with PID 14728.
```

Compatibilité

Il n'y a pas d'instruction `NOTIFY` dans le standard SQL.

Voir aussi

LISTEN, UNLISTEN

PREPARE

PREPARE — prépare une instruction pour exécution

Synopsis

```
PREPARE nom [ (type_données [, ...] ) ] AS instruction
```

Description

PREPARE crée une instruction préparée. Une instruction préparée est un objet côté serveur qui peut être utilisé pour optimiser les performances. Quand l'instruction PREPARE est exécutée, l'instruction spécifiée est lue, analysée et réécrite. Quand une commande EXECUTE est lancée par la suite, l'instruction préparée est planifiée et exécutée. Cette division du travail évite une analyse répétitive tout en permettant au plan d'exécution de dépendre des valeurs spécifiques du paramètre.

Les instructions préparées peuvent prendre des paramètres : les valeurs sont substituées dans l'instruction lorsqu'elle est exécutée. Lors de la création de l'instruction préparée, faites référence aux paramètres suivant leur position, $1, $2, etc. Une liste correspondante des types de données des paramètres peut être spécifiée si vous le souhaitez. Quand le type de donnée d'un paramètre n'est pas indiqué ou est déclaré comme inconnu (unknown), le type est inféré à partir du contexte dans lequel le paramètre est référencé en premier (si possible). Lors de l'exécution de l'instruction, indiquez les valeurs réelles de ces paramètres dans l'instruction EXECUTE. Référez-vous à EXECUTE pour plus d'informations à ce sujet.

Les instructions préparées sont seulement stockées pour la durée de la session en cours. Lorsque la session se termine, l'instruction préparée est oubliée et, du coup, elle doit être recréée avant d'être utilisée de nouveau. Ceci signifie aussi qu'une seule instruction préparée ne peut pas être utilisée par plusieurs clients de bases de données simultanément ; néanmoins, chaque client peut créer sa propre instruction préparée à utiliser. Les instructions préparées peuvent être supprimées manuellement en utilisant la commande DEALLOCATE.

Les instructions préparées sont principalement intéressantes quand une seule session est utilisée pour exécuter un grand nombre d'instructions similaires. La différence de performances est potentiellement significative si les instructions sont complexes à planifier ou à réécrire, par exemple, si la requête implique une jointure de plusieurs tables ou requiert l'application de différentes règles. Si l'instruction est relativement simple à planifier ou à réécrire mais assez coûteuse à exécuter, l'avantage de performance des instructions préparées est moins net.

Paramètres

nom

Un nom quelconque donné à cette instruction préparée particulière. Il doit être unique dans une session et est utilisé par la suite pour exécuter ou désallouer cette instruction préparée.

type_données

Le type de données d'un paramètre de l'instruction préparée. Si le type de données d'un paramètre particulier n'est pas spécifié ou est spécifié comme étant inconnu (unknown), il sera inferré à partir du contexte dans lequel le paramètre est référencé en premier. Pour référencer les paramètres de l'instruction préparée, utilisez $1, $2, etc.

instruction

Toute instruction SELECT, INSERT, UPDATE, DELETE, MERGE ou VALUES.

Notes

Une requête préparée peut être exécutée soit avec un *plan générique* soit avec un *plan personnalisé*. Un plan générique est le même pour toutes les exécutions alors qu'un plan personnalisé est généré pour une exécution spécifique en utilisant les valeurs des paramètres données lors de son appel. L'utilisation d'un plan générique évite une charge supplémentaire pour la planification mais, dans certaines situations, un plan personnalisé sera bien plus efficace à exécuter parce que le planificateur pourra utiliser sa connaissance des valeurs des paramètres. (Bien sûr, si la requête préparée n'a pas de paramètres, ceci n'a pas d'intérêt et un plan générique sera toujours utilisé.)

Par défaut (autrement dit, quand plan_cache_mode est configuré à `auto`), le serveur choisira automatiquement l'utilisation d'un plan générique ou personnalisé pour une requête préparée utilisant des paramètres. La règle actuelle pour ceci est que les cinq premières exécutions sont réalisées avec des plans personnalisés et le coût moyen estimé pour ces plans est calculé. Un plan générique est créé et son estimation de coût est comparé au coût moyens des cinq plans personnalisés. Les exécutions suivantes utilisent le plan générique si son coût n'est pas tellement plus élevé que le coût moyen des plans personnalisés pour éviter de perdre du temps sur la planification.

Cette heuristique peut être surchargée, en forçant le serveur à utiliser soit des plans génériques soit des plans personnalisés, en configurant `plan_cache_mode` à, respectivement, `force_generic_plan` ou `force_custom_plan`. Ce paramètre est principalement utile si l'estimation de coût du plan générique est très mauvais, autorisant son choix même si le coût réel est bien plus important que celui d'un plan personnalisé.

Pour examiner le plan de requête que PostgreSQL utilise pour une instruction préparée, utilisez `EXPLAIN`, par exemple :

```
EXPLAIN EXECUTE nom(valeurs_parametres);
```

Si un plan générique est utilisé, il contiendra des symboles $*n*, alors qu'un plan personnalisé contiendra les valeurs fournies pour les paramètres.

Pour plus d'informations sur la planification de la requête et les statistiques récupérées par PostgreSQL dans ce but, voir la documentation de ANALYZE.

Bien que le but principal d'une requête préparée est déviter une analyse et une planification répétée, PostgreSQL forcera une nouvelle analyse et une nouvelle planification de la requête à chaque fois que les objets de la base utilisés dans la requête auront vus leur définition modifiée (requête DDL) or their planner statistics have been updated depuis la dernière utilisation de la requête préparée. De plus, si la valeur de search_path change d'une exécution à l'autre, la requête sera de nouveau analysée d'après la nouvelle valeur du paramètre `search_path`. (Ce dernier comportement est nouveau depuis PostgreSQL 9.3.) Ces règles font d'une requête préparée l'équivalent sémantique de la soumission sans fin de la même requête, avec de meilleures performances si aucun objet n'est modifié, tout spécialement si le meilleur plan reste le même au travers des utilisations. Un exemple d'un cas où l'équivalence sémantique n'est pas parfaite est que, si la requête fait référence à une table dont le nom n'est pas qualifié du nom du schéma et qu'une nouvelle table de même nom est créée dans un schéma apparaissant avant dans le paramètre `search_path`, aucune nouvelle analyse n'intervient vu qu'aucun objet de la requête n'a été modifié. Néanmoins, si une autre modification force une nouvelle analyse, la nouvelle table sera référencée dans les utilisations suivantes.

Vous pouvez voir toutes les instructions préparées disponibles dans la session en exécutant une requête sur la vue système `pg_prepared_statements`.

Exemples

Crée une instruction préparée pour une instruction `INSERT`, puis l'exécute :

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Crée une instruction préparée pour une instruction SELECT, puis l'exécute :

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND
 u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

Dans cet exemple, notez que le type de données du deuxième paramètre n'est pas indiqué, donc il est déduit du contexte dans lequel $2 est utilisé.

Compatibilité

Le standard SQL inclut une instruction PREPARE mais il est seulement utilisé en SQL embarqué. Cette version de l'instruction PREPARE utilise aussi une syntaxe quelque peu différente.

Voir aussi

DEALLOCATE, EXECUTE

PREPARE TRANSACTION

PREPARE TRANSACTION — prépare la transaction en cours pour une validation en deux phases

Synopsis

```
PREPARE TRANSACTION id_transaction
```

Description

`PREPARE TRANSACTION` prépare la transaction courante en vue d'une validation en deux phases. À la suite de cette commande, la transaction n'est plus associée à la session courante ; au lieu de cela, son état est entièrement stocké sur disque. La probabilité est donc forte qu'elle puisse être validée avec succès, y compris en cas d'arrêt brutal de la base de données avant la demande de validation.

Une fois préparée, une transaction peut être validée ou annulée ultérieurement par, respectivement, `COMMIT PREPARED` ou `ROLLBACK PREPARED`. Ces commandes peuvent être exécutées à partir d'une session quelconque. Il n'est pas nécessaire de le faire depuis celle qui a exécuté la transaction initiale.

Du point de vue de la session l'initiant, `PREPARE TRANSACTION` diffère peu de la commande `ROLLBACK` : après son exécution, il n'y a plus de transaction active et les effets de la transaction préparée ne sont plus visibles. (Les effets redeviendront visibles si la transaction est validée.)

Si la commande `PREPARE TRANSACTION` échoue, quelqu'en soit la raison, elle devient une commande `ROLLBACK` : la transaction courante est annulée.

Paramètres

id_transaction

Un identifiant arbitraire de la transaction pour les commandes `COMMIT PREPARED` et `ROLLBACK PREPARED`. L'identifiant, obligatoirement de type chaîne littérale, doit être d'une longueur inférieure à 200 octets. Il ne peut être identique à un autre identifiant de transaction préparée.

Notes

`PREPARE TRANSACTION` n'a pas pour but d'être utilisé dans des applications ou des sessions interactives. Son but est de permettre à un gestionnaire de transactions externe pour réaliser des transactions globales atomiques au travers de plusieurs bases de données ou de ressources transactionnelles. Sauf si vous écrivez un gestionnaire de transactions, vous ne devriez probablement pas utiliser `PREPARE TRANSACTION`.

Cette commande doit être utilisée dans un bloc de transaction, initié par `BEGIN`.

Il n'est actuellement pas possible de préparer (`PREPARE`) une transaction qui a exécuté des opérations impliquant des tables temporaires ou le schéma temporaire de la session, ou qui a créé des curseurs `WITH HOLD`, ou qui a exécuté `LISTEN`, `UNLISTEN` ou `NOTIFY`. Ces fonctionnalités sont trop intégrées à la session en cours pour avoir la moindre utilité dans une transaction préparée.

Si la transaction a modifié des paramètres en exécution à l'aide de la commande `SET` (sans l'option `LOCAL`), ces effets persistent au-delà du `PREPARE TRANSACTION` et ne seront pas affectés par les commandes `COMMIT PREPARED` et `ROLLBACK PREPARED`. Du coup, dans ce cas, `PREPARE TRANSACTION` agit plus comme `COMMIT` que comme `ROLLBACK`.

Toutes les transactions préparées disponibles sont listées dans la vue système `pg_prepared_xacts`.

> **Attention**
>
> Il est préférable de ne pas conserver trop longtemps des transactions préparées dans cet état ; cela compromet, par exemple, les possibilités de récupération de l'espace par `VACUUM`, et dans certains cas extrêmes peut causer l'arrêt de la base de données pour empêcher une réutilisation d'identifiants de transactions (voir Section 25.1.5). Il ne faut pas oublier non plus qu'une telle transaction maintient les verrous qu'elle a posé. L'usage principal de cette fonctionnalité consiste à valider ou annuler une transaction préparée dès lors qu'un gestionnaire de transactions externe a pu s'assurer que les autres bases de données sont préparées à la validation.
>
> Si vous n'avez pas configuré un gestionnaire de transactions externe pour gérer les transactions préparées et vous assurer qu'elles sont fermées rapidement, il est préférable de désactiver la fonctionnalité des transactions préparées en configurant max_prepared_transactions à zéro. Ceci empêchera toute création accidentelle de transactions préparées qui pourraient alors être oubliées, ce qui finira par causer des problèmes.

Exemples

Préparer la transaction en cours pour une validation en deux phases en utilisant `foobar` comme identifiant de transaction :

```
PREPARE TRANSACTION 'foobar';
```

Compatibilité

`PREPARE TRANSACTION` est une extension PostgreSQL. Elle est conçue pour être utilisée par des systèmes extérieurs de gestion des transactions. Certains de ceux-là sont couverts par des standards (tels que X/Open XA), mais la partie SQL de ces systèmes n'est pas standardisée.

Voir aussi

COMMIT PREPARED, ROLLBACK PREPARED

REASSIGN OWNED

REASSIGN OWNED — Modifier le propriétaire de tous les objets de la base appartenant à un rôle spécifique

Synopsis

```
REASSIGN OWNED BY { ancien_rôle | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER } [, ...]
               TO { nouveau_rôle | CURRENT_ROLE | CURRENT_USER |
 SESSION_USER }
```

Description

`REASSIGN OWNED` demande au système de changer le propriétaire certains objets de la base. Les objets appartenant à l'un des *`old_role`* auront ensuite comme propriétaire *`new_role`*.

Paramètres

`ancien_rôle`

Le nom d'un rôle. Tous les objets de la base à l'intérieur de la base de connexion et tous les objets partagés (bases de données, tablespaces), dont le rôle est propriétaire, seront la propriété de *`nouveau_rôle`*.

`nouveau_rôle`

Le nom du rôle qui sera le nouveau propriétaire des objets affectés.

Notes

`REASSIGN OWNED` est souvent utilisé pour préparer à la suppression de un ou plusieurs rôles. Comme `REASSIGN OWNED` n'affecte pas les objets des autres bases, il est généralement nécessaire d'exécuter cette commande pour chaque base contenant des objets dont le rôle à supprimer est propriétaire.

`REASSIGN OWNED` nécessite des droits sur le rôle source et sur le rôle cible.

La commande `DROP OWNED` est une alternative qui supprime tous les objets de la base possédés par un ou plusieurs rôles.

La commande `REASSIGN OWNED` ne modifie pas les droits donnés aux *`ancien_rôle`* pour les objets dont il n'est pas propriétaire. De même, elle ne modifie pas les droits par défaut ajoutés avec `ALTER DEFAULT PRIVILEGES`. Utilisez `DROP OWNED` pour supprimer ces droits.

Voir Section 22.4 pour plus de détails.

Compatibilité

L'instruction `REASSIGN OWNED` est une extension PostgreSQL.

Voir aussi

DROP OWNED, DROP ROLE, ALTER DATABASE

REFRESH MATERIALIZED VIEW

REFRESH MATERIALIZED VIEW — remplacer le contenu d'une vue matérialisée

Synopsis

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] nom
    [ WITH [ NO ] DATA ]
```

Description

`REFRESH MATERIALIZED VIEW` remplace le contenu entier d'une vue matérialisée. Pour exécuter cette commande, vous devez être le propriétaire de la vue matérialisée. L'ancien contenu est supprimé. Si `WITH DATA` est ajouté, la requête de la vue est exécutée pour fournir les nouvelles données et la vue matérialisée est laissé dans un état parcourable. Si `WITH NO DATA` est indiqué, aucune nouvelle donnée n'est générée et la vue matérialisée est laissée dans un état non parcourable.

`CONCURRENTLY` et `WITH NO DATA` ne peuvent pas être utilisées ensemble.

Paramètres

`CONCURRENTLY`

Rafraichit les données de la vue matérialisée sans bloquer les lectures de la vue matérialisée. Sans cette option, un rafraichissement des données qui affecte un grand nombre de lignes aura tendance à utiliser moins de ressources et à se terminer plus rapidement, mais pourrait bloquer les autres connexions qui essaieraient de lire la vue matérialisée. Cette option pourrait être plus rapide dans les cas où le nombre de lignes mises à jour est plus petit.

Cette option est seulement autorisée s'il existe au moins un index `UNIQUE` sur la vue matérialisée utilisant uniquement les noms de colonnes et incluant toutes les lignes ; autrement dit, cela ne peut pas être un index fonctionnel ou partiel (incluant une clause `WHERE`).

Cette option ne peut pas être utilisée dans la vue matérialisée n'est pas déjà peuplée.

Même avec cette option, seul un `REFRESH` peut être exécuté à un instant t sur une vue matérialisé.

nom

Ne renvoie pas d'erreur si la vue matérialisée n'existe pas. Un message d'avertissement est renvoyé dans ce cas.

Notes

S'il existe une clause `ORDER BY` dans la requête de définition de la vue, le contenu original de la vue matérialisée sera trié de cette façon. Cependant, `REFRESH MATERIALIZED VIEW` ne garantit pas de préserver cet ordre.

Exemples

Cette commande remplacera le contenu de la vue matérialisée `resume_commandes` en utilisant la requête indiquée dans la définition de la vue matérialisée et en la laissant dans un état parcourable :

```
REFRESH MATERIALIZED VIEW resume_commandes;
```

Cette commande libèrera le stockage associé avec la vue matérialisée `stats_base_annuel` et la laissera dans un état non parcourable :

```
REFRESH MATERIALIZED VIEW stats_base_annuel WITH NO DATA;
```

Compatibilité

`REFRESH MATERIALIZED VIEW` est une extension PostgreSQL.

Voir aussi

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, DROP MATERIALIZED VIEW

REINDEX

REINDEX — reconstruit les index

Synopsis

```
REINDEX [ ( option [, ...] ) ] { INDEX | TABLE | SCHEMA | DATABASE
 | SYSTEM } [ CONCURRENTLY ] nom

où option peut faire
partie de :

    CONCURRENTLY [ boolean ]
    TABLESPACE new_tablespace
    VERBOSE [ boolean ]
```

Description

`REINDEX` reconstruit un index en utilisant les données stockées dans la table, remplaçant l'ancienne copie de l'index. Il y a plusieurs raisons pour utiliser `REINDEX` :

- Un index a été corrompu et ne contient plus de données valides. Bien qu'en théorie, ceci ne devrait jamais arriver, en pratique, les index peuvent se corrompre à cause de bogues dans le logiciel ou d'échecs matériels. `REINDEX` fournit une méthode de récupération.
- L'index en question a « explosé », c'est-à-dire qu'il contient beaucoup de pages d'index mortes ou presque mortes. Ceci peut arriver avec des index B-tree dans PostgreSQL sous certains modèles d'accès inhabituels. `REINDEX` fournit un moyen de réduire la consommation d'espace de l'index en écrivant une nouvelle version de l'index sans les pages mortes. Voir Section 25.2 pour plus d'informations.
- Vous avez modifié un paramètre de stockage (par exemple, fillfactor) pour un index et vous souhaitez vous assurer que la modification a été prise en compte.
- Si la création d'un index échoue avec l'option `CONCURRENTLY`, cet index est laissé comme « invalide ». De tels index sont inutilisables mais il peut être pratique d'utiliser alors `REINDEX` pour les reconstruire. Il est à noter que seul `REINDEX INDEX` peut reconstruire un index invalide en parallèle.

Paramètres

`INDEX`

Recrée l'index spécifié. Cette forme de `REINDEX` ne peut pas être exécutée à l'intérieur d'un bloc de transaction quand elle est utilisée avec un index partitionné.

`TABLE`

Recrée tous les index de la table spécifiée. Si la table a une seconde table « TOAST », elle est aussi ré-indexée. Cette forme de `REINDEX` ne peut pas être exécutée à l'intérieur d'un bloc de transaction quand elle est utilisée avec un index partitionné.

`SCHEMA`

Recrée tous les index du schéma spécifié. Si une table de ce schéma a une table secondaire (« TOAST »), elle est aussi ré-indexée. Les index sur les catalogues systèmes partagés sont aussi traités. Cette forme de `REINDEX` ne peut pas être exécutée dans un bloc de transaction.

DATABASE

Recrée tous les index de la base de données en cours. Les index sur les catalogues système partagés sont aussi traités. Cette forme de REINDEX ne peut pas être exécutée à l'intérieur d'un bloc de transaction.

SYSTEM

Recrée tous les index des catalogues système à l'intérieur de la base de données en cours. Les index sur les catalogues système partagés sont aussi inclus. Les index des tables utilisateur ne sont pas traités. Cette forme de REINDEX ne peut pas être exécutée à l'intérieur d'un bloc de transaction.

nom

Le nom de l'index, de la table ou de la base de données spécifique à ré-indexer. Les noms de table et d'index peuvent être qualifiés du nom du schéma. Actuellement, REINDEX DATABASE et REINDEX SYSTEM ne peuvent ré-indexer que la base de données en cours, donc ce paramètre doit correspondre au nom de la base de données en cours.

CONCURRENTLY

Lorsque cette option est utilisée, PostgreSQL va reconstruire l'index sans prendre de verrou pouvant empêcher les insertions, mises à jour et suppressions en parallèle sur la table ; alors qu'une réindexation standard empêche les écritures (mais pas les lectures) sur la table jusqu'à ce qu'elle soit terminée. Certaines mises en garde sont à prendre en compte lors de l'utilisation de cette option -- voir Reconstruire des index en parallèle.

Pour les tables temporaires, REINDEX est toujours non concurrent car aucune autre session ne peut y accéder, et qu'une réindexation non concurrente est moins coûteuse.

TABLESPACE

Spécifie le nouveau tablespace des index une fois reconstruits.

VERBOSE

Affiche un message de progression à chaque index traité.

boolean

Indique si l'option sélectionnée doit être activée ou désactivée. Vous pouvez écrire TRUE, ON, ou 1 pour activer l'option, et FALSE, OFF ou 0 pour la désactiver. La valeur *boolean* peut aussi être omise, auquel cas TRUE esy supposée.

nouveau_tablespace

Le tablespace où les index seront reconstruits.

Notes

Si vous suspectez la corruption d'un index sur une table utilisateur, vous pouvez simplement reconstruire cet index, ou tous les index de la table, en utilisant REINDEX INDEX ou REINDEX TABLE.

Les choses sont plus difficiles si vous avez besoin de récupérer la corruption d'un index sur une table système. Dans ce cas, il est important pour le système de ne pas avoir utilisé lui-même un des index suspects. (En fait, dans ce type de scénario, vous pourriez constater que les processus serveur s'arrêtent brutalement au lancement du service, mettant en cause l'utilisation des index corrompus.) Pour récupérer proprement, le serveur doit être lancé avec l'option -P, qui inhibe l'utilisation des index pour les recherches dans les catalogues système.

Une autre façon est d'arrêter le serveur et de relancer le serveur PostgreSQL en mode simple utilisateur avec l'option `-P` placée sur la ligne de commande. Ensuite, `REINDEX DATABASE`, `REINDEX SYSTEM`, `REINDEX TABLE` ou `REINDEX INDEX` peuvent être lancés suivant ce que vous souhaitez reconstruire. En cas de doute, utilisez la commande `REINDEX SYSTEM` pour activer la reconstruction de tous les index système de la base de données. Enfin, quittez la session simple utilisateur du serveur et relancez le serveur en mode normal. Voir la page de référence de postgres pour plus d'informations sur l'interaction avec l'interface du serveur en mode simple utilisateur.

Une session standard du serveur peut aussi être lancée avec `-P` dans les options de la ligne de commande. La méthode pour ce faire varie entre les clients mais dans tous les clients basés sur libpq, il est possible de configurer la variable d'environnement `PGOPTIONS` à `-P` avant de lancer le client. Notez que, bien que cette méthode ne verrouille pas les autres clients, il est conseillé d'empêcher les autres utilisateurs de se connecter à la base de données endommagée jusqu'à la fin des réparations.

`REINDEX` est similaire à une suppression et à une nouvelle création de l'index. Dans les faits, le contenu de l'index est complètement recréé. Néanmoins, les considérations de verrouillage sont assez différentes. `REINDEX` verrouille les écritures mais pas les lectures de la table mère de l'index. Il positionne également un verrou de type `ACCESS EXCLUSIVE` sur l'index en cours de traitement, ce qui bloque les lectures qui tentent de l'utiliser. En particulier, l'optimiseur de requêtes essaie de prendre un verrou `ACCESS SHARE` sur chaque index de la table, quelque soit la requête, et de ce fait, `REINDEX` bloque virtuellement toute requête sauf certaines requêtes préparées dont le plan a été mis en cache et qui n'utilise pas cet index. Au contraire, `DROP INDEX` prends temporairement un verrou de type `ACCESS EXCLUSIVE` sur la table parent, bloquant ainsi écritures et lectures. Le `CREATE INDEX` qui suit verrouille les écritures mais pas les lectures ; comme l'index n'existe pas, aucune lecture ne peut être tentée, signifiant qu'il n'y a aucun blocage et que les lectures sont probablement forcées de réaliser des parcours séquentiels complets.

Ré-indexer un seul index ou une seule table requiert d'être le propriétaire de cet index ou de cette table. Ré-indexer un schéma ou une base de données requiert d'être le propriétaire du schéma ou de la base de données. Notez que, du coup, il est parfois possible pour des utilisateurs standards de reconstruire les index de tables dont ils ne sont pas propriétaires Néanmoins, il existe une exception spéciale, quand la commande `REINDEX DATABASE`, `REINDEX SCHEMA` ou `REINDEX SYSTEM` est exécutée par un utilisateur standard, les index sur les catalogues partagés seront ignorés sauf si l'utilisateur possède le catalogue (ce qui ne sera généralement pas le cas). Bien sûr, les superutilisateurs peuvent toujours tout ré-indexer.

Réindexer des index partitionnés ou des tables partitionnées est supporté avec les commandes respectives `REINDEX INDEX` et `REINDEX TABLE`. Chaque partition de la relation partitionnée est réindexée dans une transaction séparée. Ces commandes ne peuvent pas être utilisées à l'intérieur d'un bloc de transaction lors d'un travail sur une table partitionnée ou sur un index partitionnée.

Lors de l'utilisation de la clause `TABLESPACE` avec `REINDEX` sur une table ou un index partitionné, seules les références du tablespace des partitions feuilles sont mises à jour. Comme les index partitionnées ne sont pas mis à jour, il est recommandé d'utiliser séparément `ALTER TABLE ONLY` sur celles-ci pour que toute nouvelle partition attachée hérite du nouveau tablespace. En cas d'échec, il pourrait ne pas avoir déplacer tous les index vers le nouveau tablespace. Ré-exécuter la commande reconstruira les partitions feuilles et déplacera les index non traités précédemment vers le nouveau tablespace.

Si `SCHEMA`, `DATABASE` ou `SYSTEM` sont utilisés avec `TABLESPACE`, les relations systèmes sont ignorés et un simple message `WARNING` sera généré. Les index sur les tables TOAST sont reconstruits, mais pas déplacés dans le nouveau tablespace.

Reconstruire des index en parallèle

Recréer un index peut interférer avec les opérations normales d'une base de données. Habituellement, PostgreSQL verrouille la table dont dépend l'index à reconstruire pour la protéger des écritures et reconstruit l'index complet avec un seul parcours de la table. Les autres transactions peuvent toujours lire la table mais si elles essaient d'insérer, mettre à jour ou supprimer des lignes dans la table, elles

seront bloquées jusqu'à la fin de la reconstruction de l'index. Ceci peut avoir un effet sérieux si le système est une base en production. Les très grosses tables peuvent demander plusieurs heures pour être indexées. Même pour les petites tables, une reconstruction d'index peut bloquer les processus qui voudraient écrire dans la table pendant des périodes longues inacceptables sur un système de production.

PostgreSQL supporte la reconstruction des index sans verrouillage des écritures. Cette méthode est appelée en précisant l'option `CONCURRENTLY` de `REINDEX`. Quand cette option est utilisée, PostgreSQL doit réaliser deux parcours de table pour chaque index à reconstruire et doit attendre la fin de toutes les transactions existantes qui peuvent utiliser cet index. Du coup, cette méthode requiert plus de travail qu'une reconstruction standard de l'index et est bien plus longue à se terminer puisqu'elle doit également attendre la fin des transactions en cours qui pourraient modifier cet index. Néanmoins, comme cela autorise la poursuite des opérations pendant la reconstruction de l'index, cette méthode est utile pour reconstruire des index dans un environnement en production. Bien sûr, la charge CPU et I/O supplémentaire imposée par la recréation de l'index peut ralentir les autres opérations.

Les étapes suivantes interviennent dans la ré-indexation en parallèle. Chaque étape se déroule dans une transaction séparée. Si plusieurs index doivent être reconstruits, alors chaque étape itérera sur l'ensemble des index avant de passer à l'étape suivante.

1. Une nouvelle définition transitoire de l'index est ajoutée au catalogue `pg_index`. Cette définition sera utilisée pour remplacer celle de l'ancien index. Un verrou `SHARE UPDATE EXCLUSIVE` est posé au niveau de la session sur les index reconstruits ainsi que sur les tables qui leurs sont associées pour éviter toute modification de structure pendant le processus.

2. Une première itération de construction de l'index est réalisée pour chaque nouvel index. Dès que l'index est construit, son marqueur `pg_index.indisready` est positionné à « true » pour le rendre disponible aux insertions et visible aux autres transactions dès que la session ayant procédé à la construction est terminée. Cette étape est réalisée dans une transaction séparée pour chaque index.

3. Ensuite, une seconde itération est réalisée pour ajouter les enregistrements qui ont été créés pendant que la première itération était exécutée. Cette étape est également réalisée dans une transaction séparée pour chaque index.

4. Toutes les contraintes qui font référence à l'index reconstruit sont changées pour pointer sur la nouvelle définition d'index, et le nom des index sont également changés. À ce stade, `pg_index.indisvalid` est positionné à « true » pour le nouvel index et à « false » pour l'ancien. Une invalidation de cache est alors demandée entraînant l'annulation de toutes les sessions référençant l'ancien index.

5. Les anciens index voient leurs marqueurs `pg_index.indisready` positionnés à « false » pour y empêcher tout nouvel ajout d'enregistrement, après avoir attendu que les requêtes en cours qui pourraient faire référence à ces index se terminent.

6. Les anciens index sont supprimés. Les verrous de session `SHARE UPDATE EXCLUSIVE` pour les index et leurs tables sont relâchés.

Si un problème survient pendant la recréation des index, comme une violation d'unicité dans un index unique par exemple, la commande `REINDEX` va échouer mais laisser le nouvel index « invalide » en plus de celui qui existait déjà. L'index sera ignoré par les requêtes car il pourrait être incomplet ; cependant, il entraînera encore un surcoût lors des mises à jour. La commande psql `\d` rapportera un tel index comme `INVALID` :

```
postgres=# \d tab
       Table "public.tab"
 Column |  Type   | Modifiers
--------+---------+-----------
 col    | integer |
```

```
Indexes:
    "idx" btree (col)
    "idx_ccnew" btree (col) INVALID
```

Si l'index marqué `INVALID` a pour suffixe `_ccnew`, alors il correspond à l'index transitoire créé pendant l'opération concurrente, et la méthode de récupération recommandée est de le supprimer en utilisant `DROP INDEX`, puis de tenter de nouveau un `REINDEX CONCURRENTLY`. Si l'index invalide a pour suffixe `_ccold`, il correspond à l'index original qui n'a pas pu être supprimé ; la méthode de récupération recommandé est de seulement supprimer l'index car la reconstruction a réussi. Un nombre différent de zéro peut être ajouté au suffixe des noms d'index invalides pour qu'ils restent uniques, par exemple `_ccnew1`, `_ccold2`, etc.

La construction normale d'index permet d'autres constructions d'index sur la même table simultanément, mais uniquement une seule création d'index en parallèle peut être réalisée sur une table à la fois. Dans les deux cas, il n'est pas permis de procéder à une modification de structure de la table pendant l'opération. Une autre différence est que les commandes `REINDEX TABLE` ou `REINDEX INDEX` classiques peuvent être exécutées à l'intérieur d'un bloc de transaction, alors que ce n'est pas le cas avec `REINDEX CONCURRENTLY`.

Comme toute transaction longue, `REINDEX` sur une table peut affecter les lignes supprimables par un `VACUUM` concurrent sur toute autre table.

`REINDEX SYSTEM` ne supporte pas `CONCURRENTLY` puisque les catalogues systèmes ne peuvent pas être re-indexés en parallèle.

En outre, les index des contraintes d'exclusion ne peuvent pas être reconstruits en parallèle. Si un tel index est appelé directement dans cette commande, une erreur est soulevée. Si une table ou base de données avec des index de contraintes d'exclusion est ré-indexée en parallèle, ces index seront ignorés. (Il est toutefois possible de reconstruire de tels index sans l'option `CONCURRENTLY`.)

Chaque processus exécutant `REINDEX` indiquera sa progression dans la vue `pg_stat_progress_create_index`. Voir Section 28.4.2 pour les détails.

Exemples

Reconstruit un index simple :

```
REINDEX INDEX my_index;
```

Recrée les index sur la table `ma_table` :

```
REINDEX TABLE ma_table;
```

Reconstruit tous les index d'une base de données particulière sans faire confiance à la validité des index système :

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

Reconstruit les index d'une table tout en ne bloquant pas les opérations de lectures et d'écriture sur les relations impliquées lors de la réindexation :

```
REINDEX TABLE CONCURRENTLY my_broken_table;
```

Compatibilité

Il n'existe pas de commande `REINDEX` dans le standard SQL.

Voir aussi

CREATE INDEX, DROP INDEX, reindexdb, Section 28.4.2

RELEASE SAVEPOINT

RELEASE SAVEPOINT — détruit un point de sauvegarde précédemment défini

Synopsis

```
RELEASE [ SAVEPOINT ] nom_pointsauvegarde
```

Description

`RELEASE SAVEPOINT` détruit un point de sauvegarde défini précédemment dans la transaction courante.

La destruction d'un point de sauvegarde le rend indisponible comme point de retour. C'est, pour l'utilisateur, le seul comportement visible. Elle ne défait pas les commandes exécutées après l'établissement du point de sauvegarde (pour cela, voir ROLLBACK TO SAVEPOINT). Détruire un point de sauvegarde quand il n'est plus nécessaire peut permettre au système de récupérer certaines ressources sans attendre la fin de la transaction.

`RELEASE SAVEPOINT` détruit aussi tous les points de sauvegarde créés ultérieurement au point de sauvegarde indiqué.

Paramètres

`nom_pointsauvegarde`

Le nom du point de sauvegarde à détruire.

Notes

Spécifier un nom de point de sauvegarde qui n'a pas été défini est une erreur.

Il n'est pas possible de libérer un point de sauvegarde lorsque la transaction est dans un état d'annulation.

Si plusieurs points de transaction ont le même nom, seul le plus récemment défini et non libéré est libéré. Des commandes répétées libéreront progressivement les anciens points de transaction.

Exemples

Pour établir puis détruire un point de sauvegarde :

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT mon_pointsauvegarde;
COMMIT;
```

La transaction ci-dessus insère à la fois 3 et 4.

Compatibilité

Cette commande est conforme au standard SQL. Le standard impose le mot clé `SAVEPOINT` mais PostgreSQL autorise son omission.

Voir aussi

BEGIN, COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT, SAVEPOINT

RESET

RESET — réinitialise un paramètre d'exécution à sa valeur par défaut

Synopsis

```
RESET paramètre_configuration
RESET ALL
```

Description

`RESET` réinitialise les paramètres d'exécution à leur valeur par défaut. `RESET` est une alternative à

```
SET paramètre_configuration TO DEFAULT
```

On pourra se référer à SET pour plus de détails.

La valeur par défaut est définie comme la valeur qu'aurait la variable si aucune commande `SET` n'avait modifié sa valeur pour la session en cours. La source effective de cette valeur peut être dans les valeurs par défaut compilées, le fichier de configuration, les options de la ligne de commande ou les paramétrages spécifiques à la base de données ou à l'utilisateur. Ceci est subtilement différent de le définir comme « la valeur qu'a le paramètre au lancement de la session » parce que, si la valeur provenait du fichier de configuration, elle sera annulée par ce qui est spécifié maintenant dans le fichier de configuration. Voir Chapitre 20 pour les détails.

Le comportement transactionnel de `RESET` est identique à celui de la commande `SET` : son effet sera annulée par une annulation de la transaction.

Paramètres

`paramètre_configuration`

Nom d'un paramètre configurable. Les paramètres disponibles sont documentés dans Chapitre 20 et sur la page de référence SET.

`ALL`

Réinitialise tous les paramètres configurables à l'exécution.

Exemples

Pour réinitialiser `timezone` :

```
RESET timezone;
```

Compatibilité

`RESET` est une extension de PostgreSQL.

Voir aussi

SET, SHOW

REVOKE

REVOKE — supprime les droits d'accès

Synopsis

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES |
 TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] nom_table [, ...]
         | ALL TABLES IN SCHEMA nom_schéma [, ...] }
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( nom_colonne
 [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( nom_colonne [, ...] ) }
    ON [ TABLE ] nom_table [, ...]
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE nom_séquence [, ...]
         | ALL SEQUENCES IN SCHEMA nom_schéma [, ...] }
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL
 [ PRIVILEGES ] }
    ON DATABASE nom_base [, ...]
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN nom_domaine [, ...]
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER nom_fdw [, ...]
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER nom_serveur [, ...]
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } nom_fonction
 [ ( [ [ mode_arg ] [ nom_arg ] type_arg [, ...] ] ) ] [, ...]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN
 SCHEMA nom_schéma [, ...] }
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE nom_lang [, ...]
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SET | ALTER SYSTEM } [, ...] | ALL [ PRIVILEGES ] }
    ON PARAMETER paramètre_configuration [, ...]
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA nom_schéma [, ...]
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE nom_tablespace [, ...]
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE nom_type [, ...]
    FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ ADMIN OPTION FOR ]
    nom_role [, ...] FROM spécification_rôle [, ...]
    [ GRANTED BY spécification_rôle ]
    [ CASCADE | RESTRICT ]

où spécification_rôle fait partie de :

    [ GROUP ] role_name
  | PUBLIC
  | CURRENT_ROLE
  | CURRENT_USER
  | SESSION_USER
```

Description

La commande REVOKE retire des droits précédemment attribués à un ou plusieurs rôles. Le mot clé PUBLIC fait référence au groupe implicitement défini de tous les rôles.

Voir la description de la commande GRANT pour connaître la signification des types de droits.

Notez qu'un rôle possède la somme des droits qui lui ont été donnés directement, des droits qui ont été donnés à un rôle dont il est membre et des droits donnés à PUBLIC. Ainsi, par exemple, retirer les droits de SELECT à PUBLIC ne veut pas nécessairement dire que plus aucun rôle n'a le droit de faire de SELECT sur l'objet : ceux qui en avaient obtenu le droit directement ou via un autre rôle l'ont toujours. De même, révoquer SELECT d'un utilisateur ne l'empêchera peut-être pas d'utiliser SELECT si PUBLIC ou un autre de ses rôle a toujours les droits SELECT.

Si GRANT OPTION FOR est précisé, seul l'option de transmission de droit (grant option) est supprimée, pas le droit lui même. Sinon, le droit et l'option de transmission de droits sont révoqués.

Si un utilisateur détient un privilège avec le droit de le transmettre, et qu'il l'a transmis à d'autres utilisateurs, alors les droits de ceux-ci sont appelés des droits dépendants. Si les droits ou le droit de transmettre du premier utilisateur sont supprimés, et que des droits dépendants existent, alors ces droits dépendants sont aussi supprimés si l'option CASCADE est utilisée. Dans le cas contraire, la suppression de droits est refusée. Cette révocation récursive n'affecte que les droits qui avaient été attribués à travers une chaîne d'utilisateurs traçable jusqu'à l'utilisateur qui subit la commande REVOKE. Du coup, les utilisateurs affectés peuvent finalement garder le droit s'il avait aussi été attribué via d'autres utilisateurs.

En cas de révocation des droits sur une table, les droits sur les colonnes correspondantes (s'il y en a) sont automatiquement révoqués pour toutes les colonnes de la table en même temps. D'un autre côté, si un rôle a des droits sur une table, supprimer les mêmes droits pour des colonnes individuelles n'aura aucun effet.

Lors de la révocation de l'appartenance d'un rôle, GRANT OPTION est appelé ADMIN OPTION mais le comportement est similaire. Cette syntaxe de la commande autorise aussi une option GRANTED BY mais cette option est actuellement ignorée (sauf pour vérifier l'existence du rôle nommé). Notez aussi que cette forme de la commande ne permet pas le mot GROUP. dans *role_specification*.

Notes

Un utilisateur ne peut révoquer que les droits qu'il a donnés directement. Si, par exemple, un utilisateur A a donné un droit et la possibilité de le transmettre à un utilisateur B, et que B à son tour l'a donné à C, alors A ne peut pas retirer directement le droit de C. À la place, il peut supprimer le droit de transmettre à B et utiliser l'option CASCADE pour que le droit soit automatiquement supprimé à C. Autre exemple, si A et B ont donné le même droit à C, A peut révoquer son propre don de droit mais pas celui de B, donc C dispose toujours de ce droit.

Lorsqu'un utilisateur, non propriétaire de l'objet, essaie de révoquer (REVOKE) des droits sur l'objet, la commande échoue si l'utilisateur n'a aucun droit sur l'objet. Tant que certains droits sont disponibles, la commande s'exécute mais ne sont supprimés que les droits dont l'utilisateur a l'option de transmission. La forme REVOKE ALL PRIVILEGES affiche un message d'avertissement si les options de transmissions pour un des droits nommés spécifiquement dans la commande ne sont pas possédés. (En principe, ces instructions s'appliquent aussi au propriétaire de l'objet mais comme le propriétaire est toujours traité comme celui détenant toutes les options de transmission, ces cas n'arrivent jamais.)

Si un superutilisateur choisit d'exécuter une commande GRANT ou REVOKE, la commande est exécutée comme si elle était lancée par le propriétaire de l'objet affecté. Comme tous les droits proviennent du propriétaire d'un objet (directement ou via une chaîne de transmissions de droits), un superutilisateur peut supprimer tous les droits sur un objet mais cela peut nécessiter l'utilisation de CASCADE comme expliqué précédemment.

REVOKE peut aussi être effectué par un rôle qui n'est pas le propriétaire de l'objet affecté mais qui est un membre du rôle qui possède l'objet ou qui est un membre d'un rôle qui détient les droits WITH GRANT OPTION sur cet objet. Dans ce cas, la commande est exécutée comme si elle avait été exécutée par le rôle qui possède réellement l'objet ou détient les droits WITH GRANT OPTION. Par exemple, si la table t1 est possédée par le rôle g1, dont le rôle u1 est membre, alors u1 peut supprimer des droits sur t1 qui sont enregistrés comme donnés par g1. Ceci incluera les dons de droits effectués par u1 ainsi que ceux effectués par les autres membres du rôle g1.

Si le rôle exécutant REVOKE détient les droits indirectement via plus d'un chemin d'appartenance, le rôle indiqué comme ayant effectué la commande est non déterminable à l'avance. Dans de tels cas, il est préférable d'utiliser SET ROLE pour devenir le rôle que vous souhaitez voir exécuter la commande REVOKE. Ne pas faire cela peut avoir comme résultat de supprimer des droits autres que ceux que vous vouliez, voire même de ne rien supprimer du tout.

Voir Section 5.7 pour plus d'informations sur les types de droit spécifiques, ainsi que sur la façon pour inspecter les droits des objets.

Exemples

Enlève au groupe public le droit d'insérer des lignes dans la table films :

```
REVOKE INSERT ON films FROM PUBLIC;
```

Supprime tous les droits de l'utilisateur manuel sur la vue genres :

```
REVOKE ALL PRIVILEGES ON genres FROM manuel;
```

Notez que ceci signifie en fait « révoque tous les droits que j'ai donné ».

Supprime l'appartenance de l'utilisateur joe au rôle admins :

```
REVOKE admins FROM joe;
```

Compatibilité

La note de compatibilité de la commande GRANT s'applique par analogie à REVOKE. Les mots clés RESTRICT ou CASCADE sont requis d'après le standard, mais PostgreSQL utilise RESTRICT par défaut.

Voir aussi

GRANT, ALTER DEFAULT PRIVILEGES

ROLLBACK

ROLLBACK — annule la transaction en cours

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

Description

`ROLLBACK` annule la transaction en cours et toutes les modifications effectuées lors de cette transaction.

Paramètres

`WORK`
`TRANSACTION`

Mots clés optionnels. Ils sont sans effet.

`AND CHAIN`

Si `AND CHAIN` est précisé, une nouvelle transaction est immédiatement commencée avec les mêmes caractéristiques (voir SET TRANSACTION) que celle qui vient de se terminer. Sinon, aucune nouvelle transaction n'est démarrée.

Notes

L'utilisation de la commande `COMMIT` permet de terminer une transaction avec succès.

Exécuter `ROLLBACK` en dehors d'un bloc de transaction cause l'émission d'un message d'avertissement mais n'a pas d'autres effets. `ROLLBACK AND CHAIN` renvoie une erreur quand il est exécuté en dehors d'un bloc de transaction.

Exemples

Pour annuler toutes les modifications :

```
ROLLBACK;
```

Compatibilité

La commande `ROLLBACK` est conforme au standard SQL. La syntaxe `ROLLBACK TRANSACTION` est une extension PostgreSQL.

Voir aussi

BEGIN, COMMIT, ROLLBACK TO SAVEPOINT

ROLLBACK PREPARED

ROLLBACK PREPARED — annule une transaction précédemment préparée en vue d'une validation en deux phases

Synopsis

```
ROLLBACK PREPARED id_transaction
```

Description

`ROLLBACK PREPARED` annule une transaction préparée.

Paramètres

`id_transaction`

L'identifiant de la transaction à annuler.

Notes

Pour annuler une transaction préparée, il est impératif d'être soit l'utilisateur qui a initié la transaction, soit un superutilisateur. Il n'est, en revanche, pas nécessaire d'être dans la session qui a initié la transaction.

Cette commande ne peut pas être exécutée à l'intérieur d'un bloc de transaction. La transaction préparée est annulée immédiatement.

Toutes les transactions préparées disponibles sont listées dans la vue système `pg_prepared_xacts`.

Exemples

Annuler la transaction identifiée par `foobar` :

```
ROLLBACK PREPARED 'foobar';
```

Compatibilité

L'instruction `ROLLBACK PREPARED` est une extension PostgreSQL. Elle est destinée à être utilisée par des systèmes tiers de gestion des transactions, dont le fonctionnement est parfois standardisé (comme X/Open XA), mais la portion SQL de ces systèmes ne respecte pas le standard.

Voir aussi

PREPARE TRANSACTION, COMMIT PREPARED

ROLLBACK TO SAVEPOINT

ROLLBACK TO SAVEPOINT — annule les instructions jusqu'au point de sauvegarde

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ] TO
 [ SAVEPOINT ] nom_pointsauvegarde
```

Description

Annule toutes les commandes qui ont été exécutées après l'établissement du point de sauvegarde. Le point de sauvegarde reste valide. Il est possible d'y d'y revenir encore si cela s'avérait nécessaire.

`ROLLBACK TO SAVEPOINT` détruit implicitement tous les points de sauvegarde établis après le point de sauvegarde indiqué.

Paramètres

nom_pointsauvegarde

Le point de sauvegarde où retourner.

Notes

`RELEASE SAVEPOINT` est utilisé pour détruire un point de sauvegarde sans annuler les effets de commandes exécutées après son établissement.

Spécifier un nom de point de sauvegarde inexistant est une erreur.

Les curseurs ont un comportement quelque peu non transactionnel en ce qui concerne les points de sauvegarde. Tout curseur ouvert à l'intérieur d'un point de sauvegarde est fermé lorsque le point de sauvegarde est rejoint. Si un curseur précédemment ouvert est affecté par une commande `FETCH` ou `MOVE` à l'intérieur d'un point de sauvegarde rejoint par la suite, la position du curseur reste celle obtenue par `FETCH` (c'est-à-dire que le déplacement du curseur dû au `FETCH` n'est pas annulé). La fermeture d'un curseur n'est pas non plus remise en cause par une annulation. Néanmoins, certains effets de bord causés par la requête du curseur (comme les effets de bord des fonctions volatiles appelées par la requête) *sont* annulés s'ils surviennent lors d'un point de sauvegarde qui est annulé plus tard. Un curseur dont l'exécution provoque l'annulation d'une transaction est placé dans un état non exécutable. De ce fait, alors même que la transaction peut être restaurée par `ROLLBACK TO SAVEPOINT`, le curseur ne peut plus être utilisé.

Exemples

Pour annuler les effets des commandes exécutées après l'établissement de `mon_pointsauvegarde` :

```
ROLLBACK TO SAVEPOINT mon_pointsauvegarde;
```

La position d'un curseur n'est pas affectée par l'annulation des points de sauvegarde :

```
BEGIN;
```

```
DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        1

ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

Compatibilité

Le standard SQL spécifie que le mot clé `SAVEPOINT` est obligatoire mais PostgreSQL et Oracle autorisent son omission. SQL n'autorise que `WORK`, pas `TRANSACTION`, après `ROLLBACK`. De plus, SQL dispose d'une clause optionnelle `AND [ NO ] CHAIN` qui n'est actuellement pas supportée par PostgreSQL. Pour le reste, cette commande est conforme au standard SQL.

Voir aussi

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, SAVEPOINT

SAVEPOINT

SAVEPOINT — définit un nouveau point de sauvegarde à l'intérieur de la transaction en cours

Synopsis

```
SAVEPOINT nom_pointsauvegarde
```

Description

`SAVEPOINT` établit un nouveau point de sauvegarde à l'intérieur de la transaction en cours.

Un point de sauvegarde est une marque spéciale à l'intérieur d'une transaction qui autorise l'annulation de toutes les commandes exécutées après son établissement, restaurant la transaction dans l'état où elle était au moment de l'établissement du point de sauvegarde.

Paramètres

nom_pointsauvegarde

Le nom du nouveau point de sauvegarde. Si des points de sauvegarde de même nom existent déjà, ils deviendront inaccessibles jusqu'à ce que les points de sauvegarde de même nom mais plus récents ne soient libérés.

Notes

Utilisez `ROLLBACK TO` pour annuler un point de sauvegarde. Utilisez `RELEASE SAVEPOINT` pour détruire un point de sauvegarde, conservant l'effet des commandes exécutées après son établissement.

Les points de sauvegarde peuvent seulement être établis à l'intérieur d'un bloc de transaction. Plusieurs points de sauvegarde peuvent être définis dans une transaction.

Exemples

Pour établir un point de sauvegarde et annuler plus tard les effets des commandes exécutées après son établissement :

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

La transaction ci-dessus insère les valeurs 1 et 3, mais pas 2.

Pour établir puis détruire un point de sauvegarde :

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT mon_pointsauvegarde;
    INSERT INTO table1 VALUES (4);
```

```
    RELEASE SAVEPOINT mon_pointsauvegarde;
COMMIT;
```

La transaction ci-dessus insère à la fois les valeurs 3 et 4.

Pour utiliser un seul point de transaction :

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);

    -- rollback jusqu'au deuxième point de sauvegarde
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;               -- affiche les lignes 1 et
 2

    -- libère le deuxième point de sauvegarde
    RELEASE SAVEPOINT my_savepoint;

    -- annule jusqu'au premier point de sauvegarde
    ROLLBACK TO SAVEPOINT my_savepoint;
    SELECT * FROM table1;               -- affiche seulement la
 ligne 1
COMMIT;
```

La transaction ci-dessus montre que la ligne 3 est annulée en premier, puis c'est au tour de la ligne 2.

Compatibilité

SQL requiert la destruction automatique d'un point de sauvegarde quand un autre point de sauvegarde du même nom est créé. Avec PostgreSQL, l'ancien point de sauvegarde est conservé, mais seul le plus récent est utilisé pour une annulation ou une libération. (Libérer avec `RELEASE SAVEPOINT` le point de sauvegarde le plus récent fait que l'ancien est de nouveau accessible aux commandes `ROLLBACK TO SAVEPOINT` et `RELEASE SAVEPOINT`.) Sinon, `SAVEPOINT` est totalement conforme à SQL.

Voir aussi

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, ROLLBACK TO SAVEPOINT

SECURITY LABEL

SECURITY LABEL — Définir ou modifier un label de sécurité appliqué à un objet

Synopsis

```
SECURITY LABEL [ FOR fournisseur ] ON
{
  TABLE nom_objet |
  COLUMN nom_table.nom_colonne |
  AGGREGATE nom_agrégat ( signature_agrégat ) |
  DATABASE nom_objet |
  DOMAIN nom_objet |
  EVENT TRIGGER nom_objet |
  FOREIGN TABLE nom_objet |
  FUNCTION nom_fonction [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ] |
  LARGE OBJECT oid_large_object |
  MATERIALIZED VIEW nom_objet |
  [ PROCEDURAL ] LANGUAGE nom_objet |
  PROCEDURE nom_procédure [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ] |
  PUBLICATION nom_objet |
  ROLE nom_objet |
  ROUTINE nom_routine [ ( [ [ mode_arg ] [ nom_arg ] type_arg
 [, ...] ] ) ] |
  SCHEMA nom_objet |
  SEQUENCE nom_objet |
  SUBSCRIPTION nom_objet |
  TABLESPACE nom_objet |
  TYPE nom_objet |
  VIEW nom_objet
} IS { texte | NULL }

où signature_agrégat est :

* |
[ mode_arg ] [ nom_arg ] type_arg [ , ... ] |
[ [ mode_arg ] [ nom_arg ] type_arg [ , ... ] ] ORDER BY [ mode_arg
 ] [ nom_arg ] type_arg [ , ... ]
```

Description

`SECURITY LABEL` applique un label de sécurité à un objet de la base de données. Un nombre arbitraire de labels de sécurité, un par fournisseur d'labels, peut être associé à un objet donné de la base. Les fournisseurs de labels sont des modules dynamiques qui s'enregistrent eux-mêmes en utilisant la fonction `register_label_provider`.

> **Note**
>
> `register_label_provider` n'est pas une fonction SQL ; elle ne peut être appelée que depuis du code C chargé et exécuté au sein du serveur.

Le fournisseur de labels détermine si un label donné est valide, et dans quelle mesure il est permis de l'appliquer à un objet donné. Le sens des labels est également laissé à la discrétion du fournisseur d'labels. PostgreSQL n'impose aucune restriction quant à l'interprétation que peut faire un fournisseur d'un label donné, se contentant simplement d'offrir un mécanisme de stockage de ces labels. En pratique, il s'agit de permettre l'intégration de systèmes de contrôles d'accès obligatoires (en anglais, *mandatory access control* ou MAC) tels que SELinux. De tels systèmes fondent leurs autorisations d'accès sur des labels appliqués aux objets, contrairement aux systèmes traditionnels d'autorisations d'accès discrétionnaires (en anglais, *discretionary access control* ou DAC) généralement basés sur des concepts tels que les utilisateurs et les groupes.

Vous devez être le propriétaire des objets visés par une commande `SECURITY LABEL`.

Paramètres

`nom_objet`
`nom_table.nom_colonne`
`nom_agrégat`
`nom_fonction`
`nom_procédure`
`nom_routine`

Le nom de l'objet à labelliser. Les noms d'objets résidant dans des schémas (tables, fonctions, etc.) peuvent être qualifiés du nom de schéma.

`fournisseur`

Le nom du fournisseur auquel le label est associé. Le fournisseur désigné doit être chargé et accepter l'opération qui lui est proposée. Si un seul et unique fournisseur est chargé, le nom du fournisseur peut être omis par soucis de concision.

`mode_arg`

Le mode d'un argument de fonction, de procédure ou d'agrégat : `IN`, `OUT`, `INOUT` ou `VARIADIC`. Si le mode est omis, le mode par défaut `IN` est alors appliqué. À noter que `SECURITY LABEL` ne porte actuellement pas sur les arguments de mode `OUT` dans la mesure où seuls les arguments fournis en entrée sont nécessaires à l'identification d'une fonction. Il suffit donc de lister les arguments `IN`, `INOUT`, et `VARIADIC`.

`nom_arg`

Le nom d'un argument de fonction, de procédure ou d'agrégat. À noter que `SECURITY LABEL ON FUNCTION` ne porte actuellement pas sur les nom des arguments fournis aux fonctions dans la mesure où seul le type des arguments est nécessaire à l'identification d'une fonction.

`type_arg`

Le type de données d'un argument de fonction, de procédure ou d'agrégat.

`oid_large_objet`

L'OID de l'objet large.

`PROCEDURAL`

Qualificatif optionnel du langage, peut être omis.

`label`

La nouvelle configuration du label de sécurité, fourni sous la forme d'une chaine littérale.

`NULL`

Écrire `NULL` pour supprimer le label de sécurité.

Exemples

L'exemple suivant montre comment le label de sécurité d'une table pourrait être configuré ou modifié.

```
SECURITY LABEL FOR selinux ON TABLE matable IS
 'system_u:object_r:sepgsql_table_t:s0';
```

Pour supprimer le label :

```
SECURITY LABEL FOR selinux ON TABLE matable IS NULL;
```

Compatibilité

La commande `SECURITY LABEL` n'existe pas dans le standard SQL.

Voir aussi

sepgsql, `src/test/modules/dummy_seclabel`

SELECT

SELECT, TABLE, WITH — récupère des lignes d'une table ou d'une vue

Synopsis

```
[ WITH [ RECURSIVE ] requête_with [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ { * | expression [ [ AS ] nom_sortie ] } [, ...] ]
    [ FROM éléments_from [, ...] ]
    [ WHERE condition ]
    [ GROUP BY [ ALL | DISTINCT ] element_regroupement [, ...] ]
    [ HAVING condition ]
    [ WINDOW nom_window AS ( définition_window ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING opérateur ] [ NULLS
 { FIRST | LAST } ] [, ...] ]
    [ LIMIT { nombre | ALL } ]
    [ OFFSET début ] [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ total ] { ROW | ROWS } { ONLY | WITH
 TIES } ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE }
 [ OF nom_table [, ...] ] [ NOWAIT | SKIP LOCKED ] [...] ]

avec éléments_from qui peut être :

    [ ONLY ] nom_table [ * ] [ [ AS ] alias [ ( alias_colonne
 [, ...] ) ] ]
                [ TABLESAMPLE methode_echantillonnage ( argument
 [, ...] ) [ REPEATABLE ( pourcentage_echantillon ) ] ]
    [ LATERAL ] ( select ) [ AS ] alias [ ( alias_colonne
 [, ...] ) ]
    nom_requête_with [ [ AS ] alias [ ( alias_colonne [, ...] ) ] ]
    [ LATERAL ] nom_fonction ( [ argument [, ...] ] )
                [ WITH ORDINALITY ] [ [ AS ] alias
 [ ( alias_colonne [, ...] ) ] ]
    [ LATERAL ] nom_fonction ( [ argument [, ...] ] ) [ AS ] alias
 ( définition_colonne [, ...] )
    [ LATERAL ] nom_fonction ( [ argument [, ...] ] ) AS
 ( définition_colonne [, ...] )
    [ LATERAL ] ROWS FROM( nom_fonction ( [ argument [, ...] ] )
 [ AS ( définition_colonne [, ...] ) ] [, ...] )
                [ WITH ORDINALITY ] [ [ AS ] alias
 [ ( alias_colonne [, ...] ) ] ]
    éléments_from type_jointure élément_from { ON type_jointure |
 USING ( colonne_jointure [, ...] ) [ AS jointure_utilisant_alias
 ] }
    élément_from NATURAL type_jointure élément_from
    élément_from CROSS JOIN élément_from

et element_regroupement peut valoir :

    ( )
    expression
    ( expression [, ...] )
```

```
    ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
    CUBE ( { expression | ( expression [, ...] ) } [, ...] )
    GROUPING SETS ( element_regroupement [, ...] )

et requête_with est :

    nom_requête_with [ ( nom_colonne [, ...] ) ] AS [ [ NOT ]
 MATERIALIZED ] ( select | valeurs | insert | update | delete )
        [ SEARCH { BREADTH | DEPTH } FIRST BY nom_colonne [, ...]
 SET search_seq_col_name ]
        [ CYCLE nom_colonne [, ...] SET cycle_mark_col_name
 [ TO cycle_mark_value DEFAULT cycle_mark_default ]
 USING cycle_path_col_name ]

TABLE [ ONLY ] nom_table [ * ]
```

Description

`SELECT` récupère des lignes de zéro ou plusieurs tables. Le traitement général de `SELECT` est le suivant :

1. Toutes les requêtes dans la liste `WITH` sont évaluées. Elles jouent le rôle de tables temporaires qui peuvent être référencées dans la liste `FROM`. Une requête `WITH` qui est référencée plus d'une fois dans `FROM` n'est calculée qu'une fois, unless specified otherwise with `NOT MATERIALIZED` (voir Clause WITH ci-dessous).

2. Tous les éléments de la liste `FROM` sont calculés. (Chaque élément dans la liste `FROM` est une table réelle ou virtuelle.) Si plus d'un élément sont spécifiés dans la liste `FROM`, ils font l'objet d'une jointure croisée (cross-join). (Voir Clause FROM ci-dessous.)

3. Si la clause `WHERE` est spécifiée, toutes les lignes qui ne satisfont pas les conditions sont éliminées de l'affichage. (Voir Clause WHERE ci-dessous.)

4. Si la clause `GROUP BY` est spécifiée or if there are aggregate function calls, l'affichage est divisé en groupes de lignes qui correspondent à une ou plusieurs valeurs, et aux résultats des fonctions d'agrégat calculés. Si la clause `HAVING` est présente, elle élimine les groupes qui ne satisfont pas la condition donnée. (Voir Clause GROUP BY et Clause HAVING ci-dessous.) Bien que les colonnes en sortie d'une requête sont calculées nominalement à la prochaine étape, elles peuvent aussi être référencées (par nom ou numéro) dans la clause `GROUP BY`.

5. Les lignes retournées sont traitées en utilisant les expressions de sortie de `SELECT` pour chaque ligne ou groupe de ligne sélectionné. (Voir Liste SELECT ci-dessous.)

6. `SELECT DISTINCT` élimine du résultat les lignes en double. `SELECT DISTINCT ON` élimine les lignes qui correspondent sur toute l'expression spécifiée. `SELECT ALL` (l'option par défaut) retourne toutes les lignes, y compris les doublons. (cf. Clause DISTINCT ci-dessous.)

7. En utilisant les opérateurs `UNION`, `INTERSECT` et `EXCEPT`, l'affichage de plusieurs instructions `SELECT` peut être combiné pour former un ensemble unique de résultats. L'opérateur `UNION` renvoie toutes les lignes qui appartiennent, au moins, à l'un des ensembles de résultats. L'opérateur `INTERSECT` renvoie toutes les lignes qui sont dans tous les ensembles de résultats. L'opérateur `EXCEPT` renvoie les lignes qui sont présentes dans le premier ensemble de résultats mais pas dans le deuxième. Dans les trois cas, les lignes dupliquées sont éliminées sauf si `ALL` est spécifié. Le mot-clé supplémentaire `DISTINCT` peut être ajouté pour signifier explicitement que les lignes en doublon sont éliminées. Notez bien que `DISTINCT` est là le comportement par défaut, bien que `ALL` soit le défaut pour la commande `SELECT`. (Voir Clause UNION, Clause INTERSECT et Clause EXCEPT ci-dessous.)

8. Si la clause ORDER BY est spécifiée, les lignes renvoyées sont triées dans l'ordre spécifié. Si ORDER BY n'est pas indiqué, les lignes sont retournées dans l'ordre qui permet la réponse la plus rapide du système. (Voir Clause ORDER BY ci-dessous.)

9. Si les clauses LIMIT (ou FETCH FIRST) ou OFFSET sont spécifiées, l'instruction SELECT ne renvoie qu'un sous-ensemble de lignes de résultats. (Voir Clause LIMIT ci-dessous.)

10. Si la clause FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE ou FOR KEY SHARE est spécifiée, l'instruction SELECT verrouille les lignes sélectionnées contre les mises à jour concurrentes. (Voir Clause de verrouillage ci-dessous.)

Le droit SELECT sur chaque colonne utilisée dans une commande SELECT est nécessaire pour lire ses valeurs. L'utilisation de FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE ou FOR KEY SHARE requiert en plus le droit UPDATE (pour au moins une colonne de chaque table sélectionnée).

Paramètres

Clause WITH

La clause WITH vous permet de spécifier une ou plusieurs sous-requêtes qui peuvent être utilisées par leur nom dans la requête principale. Les sous-requêtes se comportent comme des tables temporaires ou des vues pendant la durée d'exécution de la requête principale. Chaque sous-requête peut être un ordre SELECT, TABLE, VALUES, INSERT, UPDATE ou bien DELETE. Lorsque vous écrivez un ordre de modification de données (INSERT, UPDATE ou DELETE) dans une clause WITH, il est habituel d'inclure une clause RETURNING. C'est la sortie de cette clause RETURNING, *et non pas* la table sous-jacente que l'ordre modifie, qui donne lieu à la table temporaire lue par la requête principale. Si la clause RETURNING est omise, l'ordre est tout de même exécuté, mais il ne produit pas de sortie ; il ne peut donc pas être référencé comme une table par la requête principale.

Un nom (sans qualification de schéma) doit être spécifié pour chaque requête WITH. En option, une liste de noms de colonnes peut être spécifié ; si elle est omise, les noms de colonnes sont déduites de la sous-requête.

Si RECURSIVE est spécifié, la sous-requête SELECT peut se référencer elle même. Une sous-requête de ce type doit avoir la forme

```
terme_non_récursif UNION [ ALL | DISTINCT ] terme_récursif
```

où l'auto-référence récursive doit apparaître dans la partie droite de l'UNION. Seule une auto-référence récursive est autorisée par requête. Les ordres de modification récursifs ne sont pas supportés, mais vous pouvez utiliser le résultat d'une commande SELECT récursive dans un ordre de modification. Voir Section 7.8 pour un exemple.

Un autre effet de RECURSIVE est que les requêtes WITH n'ont pas besoin d'être ordonnées : une requête peut en référencer une autre qui se trouve plus loin dans la liste (toutefois, les références circulaires, ou récursion mutuelle, ne sont pas implémentées). Sans RECURSIVE, les requêtes WITH ne peuvent référencer d'autres requêtes WITH soœurs que si elles sont déclarées avant dans la liste WITH.

La clause optionnelle SEARCH calcule une *colonne de séquence de recherche* qui peut être utilisé pour trier les résultats d'une requête récursive soit en *breadth-first* ou des *depth-first*. La liste de noms de colonnes fournie indique la clé de ligne qui est utilisée pour garder trace des lignes visitées. Une colonne nommée *search_seq_col_name* sera ajoutée à la liste des colonnes résultats de la requête WITH. Cette colonne peut être triée dans la requête externe pour réaliser le tri demandé. Voir Section 7.8.2.1 pour des exemples.

La clause optionnelle CYCLE est utilisé pour détecter les cycles dans les requêtes récursives. La liste des noms de colonnes fournie indiqué la clé de ligne qui est à utiliser pour garder trace des lignes

visitées. Une colonne nommée *cycle_mark_col_name* sera ajoutée à la liste de colonnes en résultat de la requête WITH. Cette colonne est initialisée à *cycle_mark_value* quand un cycle est détecté, et sinon à *cycle_mark_default*. De plus, le traitement de l'union récursif s'arrêtera quand un cycle aura été détecté. *cycle_mark_value* et *cycle_mark_default* doivent être des constantes et elles sont convertibles automatiquement vers un type de données commun. Le type de données doit avoir un opérateur d'inégalité. (Le standard SQL requiert qu'elles doivent être de type booléen ou chaîne de caractères, mais PostgreSQL ne l'oblige pas.) Par défaut, TRUE et FALSE (de type boolean) sont utilisés. De plus, une colonne nommée *cycle_path_col_name* sera ajoutée à la liste de colonnes en résultat de la requête WITH. Cette colonne est utilisée en interne pour traquer les lignes visitées. Voir Section 7.8.2.2 pour des exemples.

Les clauses SEARCH et CYCLE sont seulement valides pour les requêtes WITH récursives. *with_query* doit être une UNION(ou UNION ALL) de deux commandes SELECT (ou équivalent) commands (pas de UNIONimbriqués). Si les deux clauses sont utilisées, la colonne ajoutée par la clause SEARCH apparaît avant les colonnes ajoutées par la clause CYCLE.

La requête principale et les requêtes WITH sont toutes exécutées en même temps. Ceci implique que les effets d'une requête modifiant des données dans la clause WITH ne peuvent pas être vus des autres parties de la requête, autrement qu'en lisant son retour avec la clause RETURNING. Si des telles instructions de modification de données essaient de modifier la même ligne, les résultats sont inconnus.

Quand il y a plusieurs requêtes dans la clause WITH, RECURSIVE ne devra être écrit qu'une seule fois, immédiatement après WITH. Cela s'applique à toutes les requêtes de la clause WITH, bien que cela n'a pas d'effet sur les requêtes qui n'utilisent pas de récursion de référence en avant (*forward references*).

Une propriété clé des requêtes WITH est qu'elles ne sont normalement évaluées qu'une seule fois par exécution de la requête principale, même si la 1requête principale les utilise plus d'une fois. En particulier, vous avez la garantie que les traitements de modification de données sont exécutés une seule et unique fois, que la requête principale lise tout ou partie de leur sortie.

Néanmoins, une requête WITH peut être marquée NOT MATERIALIZED pour supprimer cette garantie. Dans ce cas, la requête WITH peut être intégrée dans la requête principale comme s'il s'agissait d'un simple sous-SELECT dans la clause FROM de la requête principale. Ceci résulte en des calculs dupliquées sur la requête principale fait référence à la requête WITH plus d'une fois ; mais si chaque utilisation requiert seulement quelques lignes de la sortie complète de la requête WITH, la clause NOT MATERIALIZED peut apporter un gain net en autorisant les requêtes à être optimisées globalement. NOT MATERIALIZED est ignoré s'il est attaché à une requête WITH récursive ou qui n'est pas sans effet de bord (autrement dit, pas un simple SELECT contenant aucune fonction volatile).

Par défaut, une requête WITH sans effet de bord est intégrée dans la requête principale si elle est utilisée exactement une fois dans la clause FROM de la requête. Ceci permet une optimisation de la jointure des deux requêtes dans des situations où cela serait sémantiquement invisible. Néanmoins, cette intégration peut être empêchée en marquant la requête WITH avec le mot-clé MATERIALIZED. Ceci peut être utile si la requête WITH est utilisée comme barrière d'optimisation pour empêcher le planificateur de choisir un mauvais plan. Les versions de PostgreSQL antérieures à la 12 ne faisaient jamais ce type d'intégration, donc les requêtes écrites pour les versions précédentes pourraient se fier sur WITH comme barrières d'optimisation.

Voir Section 7.8 pour plus d'informations.

Clause FROM

La clause FROM spécifie une ou plusieurs tables source pour le SELECT. Si plusieurs sources sont spécifiées, le résultat est un produit cartésien (jointure croisée) de toutes les sources. Mais habituellement, des conditions de qualification (via WHERE) sont ajoutées pour restreindre les lignes renvoyées à un petit sous-ensemble du produit cartésien.

La clause FROM peut contenir les éléments suivants :

nom_table

Le nom (éventuellement qualifié par le nom du schéma) d'une table ou vue existante. Si ONLY est spécifié avant le nom de la table, seule cette table est parcourue. Dans le cas contraire, la table et toutes ses tables filles (s'il y en a) sont parcourues. En option, * peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles sont inclues.

alias

Un nom de substitution pour l'élément FROM contenant l' alias. Un alias est utilisé par brièveté ou pour lever toute ambiguïté lors d'auto-jointures (la même table est parcourue plusieurs fois). Quand un alias est fourni, il cache complètement le nom réel de la table ou fonction ; par exemple, avec FROM truc AS, le reste du SELECT doit faire référence à cet élément de FROM par f et non pas par truc. Si un alias est donné, une liste d' alias de colonnes peut aussi être saisi comme noms de substitution pour différentes colonnes de la table.

TABLESAMPLE *methode_echantillonnage* (*argument* [, ...]) [REPEATABLE (*pourcentage_echantillon*)]

Une clause TABLESAMPLE après un *nom_table* indique que la *methode_echantillonnage* indiquée doit être utilisé pour récupérer un sous-ensemble des lignes de cette table. Cet échantillonnage précède l'application de tout autre filtre tel que la clause WHERE. La distribution standard de PostgreSQL inclut deux méthodes d'échantillonnage, BERNOULLIet SYSTEM mais d'autres méthodes d'échantillonnage peuvent être installées via des extensions.

Les méthodes d'échantillonnage BERNOULLI et SYSTEM acceptent chacune un seul *argument* correspondant à la fraction à échantillonner pour la table, exprimée sous la forme d'un pourcentage entre 0 et 100. Cet argument peut être une expression renvoyant un flottant (real). (D'autres méthodes d'échantillonnage pourraient accepter plus d'arguments ou des arguments différents.) Ces deux méthodes retournent chacune un sous-ensemble choisi au hasard de la table qui contiendra approximativement le pourcentage indiqué de lignes pour cette table. La méthode BERNOULLI parcourt la table complète et sélectionne ou ignore des lignes individuelles indépendemment avec la probabilité sélectionnée. La méthode SYSTEM fait un échantillonnage au niveau des blocs, chaque bloc ayant la chance indiquée d'être sélectionnée ; toutes les lignes de chaque bloc sélectionné sont renvoyées. La méthode SYSTEM est bien plus rapide que la méthode BERNOULLI quand un petit pourcentage est indiqué pour l'échantillonnage mais elle peut renvoyer un échantillon moins aléatoire de la table, dû aux effets de l'ordre des lignes.

La clause optionnelle REPEATABLE indique un nombre *seed* ou une expression à utiliser pour générer des nombres aléatoires pour la méthode d'échantillonnage. La valeur peut être toute valeur flottante non NULL. Deux requêtes précisant la même valeur seed et les mêmes valeurs en *argument* sélectionneront le même échantillon de la table si celle-ci n'a pas changé entre temps. Mais différentes valeurs seed produiront généralement des échantillons différents. Si REPEATABLE n'est pas indiqué, alors un nouvel échantillon est choisi au hasard pour chaque requête, basé sur une graine générée par le système. Notez que certaines méthodes d'échantillonnage supplémentaires pourraient ne pas accepter la clausse REPEATABLE, et toujours produire de nouveau échantillon à chaque utilisation.

select

Un sous-SELECT peut apparaître dans la clause FROM. Il agit comme si sa sortie était transformée en table temporaire pour la durée de cette seule commande SELECT. Le sous-SELECT doit être entouré de parenthèses et un alias *doit* lui être fourni. Une commande VALUES peut aussi être utilisée ici.

requête_with

Une requête WITH est référencée par l'écriture de son nom, exactement comme si le nom de la requête était un nom de table (en fait, la requête WITH cache toutes les tables qui auraient le même

nom dans la requête principale. Si nécessaire, vous pouvez accéder à une table réelle du même nom en précisant le schéma du nom de la table). Un alias peut être indiqué de la même façon que pour une table.

nom_fonction

Des appels de fonctions peuvent apparaître dans la clause FROM. (Cela est particulièrement utile pour les fonctions renvoyant des ensembles de résultats, mais n'importe quelle fonction peut être utilisée.) Un appel de fonction agit comme si la sortie de la fonction était créée comme une table temporaire pour la durée de cette seule commande SELECT. Si le type du résultat de la fonction est composite (en incluant le cas d'une fonction avec plusieurs paramètres OUT), chaque attribut devient une colonne séparée dans la table implicite.

Quand la clause optionnelle WITH ORDINALITY est ajoutée à l'appel de fonction, une colonne supplémentaire de type bigint sera ajouté aux colonnes du résultat de la fonction. Cette colonne numérote les lignes de l'ensemble de résultats de la fonction, en commençant à 1. Par défaut, cette colonne est nommée ordinality.

Un alias peut être fourni de la même façon pour une table. Si un alias de table est donné, une liste d'alias de colonnes peut aussi être écrite pour fournir des noms de substitution pour un ou plusieurs attributs du type composite en retour de la fonction, ceci incluant la colonne numérotée automatiquement.

Plusieurs appels de fonction peuvent être combinés en un seul élément dans la clause FROM en les entourant de ROWS FROM(...). La sortie d'un tel élément est la concaténation de la première ligne de chaque fonction, puis la deuxième ligne de chaque fonction, etc. Si certaines fonctions produisent moins de lignes que d'autres, des NULL sont ajoutées pour les données manquantes, ce qui permet d'avoir comme nombre de lignes celui de la fonction qui en renvoie le plus.

Si la fonction a été définie comme renvoyant le type de données record, un alias ou le mot clé AS doivent être présents, suivi par une liste de définition de colonnes de la forme (*nom_colonne type_donnée* [, ...]). La liste de définition des colonnes doit correspondre au nombre réel et aux types réels des colonnes renvoyées par la fonction.

Lors de l'utilisation de la syntaxe ROWS FROM(...), si une des fonctions nécessite une liste de définition des colonnes, il est préférable de placer la liste de définition des colonnes après l'appel de la fonction dans ROWS FROM(...). Une liste ded définition des colonnes peut être placé après la construction ROWS FROM(...) seulement s'il n'y a qu'une seule fonction et pas de clause WITH ORDINALITY.

Pour utiliser ORDINALITY avec une liste de définition de colonnes, vous devez utiliser la syntaxe ROWS FROM(...) et placer la liste de définition de colonnes dans ROWS FROM(...).

type_jointure

Un des éléments

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN

Pour les types de jointures INNER et OUTER, une condition de jointure doit être spécifiée, à choisir parmi ON *condition_jointure* ou USING (*colonne_jointure* [, ...]), ou NATURAL. Voir ci-dessous pour la signification.

Une clause JOIN combine deux éléments FROM, que nous allons appelons « tables » par simplicité bien qu'ils puissent être n'importe quel élément utilisable dans une clause FROM. Les parenthèses peuvent être utilisées pour déterminer l'ordre d'imbrication. En l'absence de parenthèses, les JOIN

sont imbriqués de gauche à droite. Dans tous les cas, JOIN est plus prioritaire que les virgules séparant les éléments FROM. Toutes les options de JOIN sont une facilité d'écriture car elles ne font rien que vous ne pourriez faire avec des clauses FROM et WHERE basiques.

LEFT OUTER JOIN renvoie toutes les lignes du produit cartésien qualifié (c'est-à-dire toutes les lignes combinées qui satisfont la condition de jointure), plus une copie de chaque ligne de la table de gauche pour laquelle il n'y a pas de ligne à droite qui satisfasse la condition de jointure. La ligne de gauche est étendue à la largeur complète de la table jointe par insertion de valeurs NULL pour les colonnes de droite. Seule la condition de la clause JOIN est utilisée pour décider des lignes qui correspondent. Les conditions externes sont appliquées après coup.

À l'inverse, RIGHT OUTER JOIN renvoie toutes les lignes jointes plus une ligne pour chaque ligne de droite sans correspondance (complétée par des NULL pour le côté gauche). C'est une simple aide à la notation car il est aisément convertible en LEFT en inversant les tables gauche et droite.

FULL OUTER JOIN renvoie toutes les lignes jointes, plus chaque ligne gauche sans correspondance (étendue par des NULL à droite), plus chaque ligne droite sans correspondance (étendue par des NULL à gauche).

ON *condition_jointure*

condition_jointure est une expression qui retourne une valeur de type boolean (comme une clause WHERE) qui spécifie les lignes d'une jointure devant correspondre.

USING (*colonne_jointure* [, ...]) [AS *jointure_utilisant_alias*]

Une clause de la forme USING (a, b, ...) est un raccourci pour ON table_gauche.a = table_droite.a AND table_gauche.b = table_droite.b .... De plus, USING implique l'affichage d'une seule paire des colonnes correspondantes dans la sortie de la jointure.

Si un nom *jointure_utilisant_alias* est indiqué, il fournit un alias de table pour les colonnes de la jointure. Seules les colonnes de la jointure listées dans la clause USING sont adressables par ce nom. Contrairement à l'habituel *alias*, cela ne cache pas les noms des tables jointes du reste de la requête. De plus, contrairement à un *alias* standard, vous ne pouvez pas écrire une liste d'alias de colonnes -- les noms en sortie des colonnes de jointure sont identiques à comment elles apparaissent en utilisant la liste USING.

NATURAL

NATURAL est un raccourci pour une liste USING qui mentionne toutes les colonnes de même nom dans les deux tables. USING qui mentionne toutes les colonnes de même nom dans les deux tables. S'il n'y a pas de noms de colonnes communs, NATURAL est équivalent à ON TRUE.

CROSS JOIN

CROSS JOIN est équivalent à INNER JOIN ON (TRUE), autrement dit, aucune ligne n'est supprimé par la qualification. Ils réalisent un produit cartésien, vous obtenez le même résultat en listant les deux tables au plus haut niveau de FROM mais restreinte par la condition de jointure s'il y en a une.

LATERAL

Le mot clé LATERAL peut précéder un élément sous-SELECT de la clause FROM. Ceci permet au sous-SELECT de faire référence aux colonnes des éléments du FROM qui apparaissent avant lui dans la liste FROM. (Sans LATERAL, chaque sous-SELECT est évalué indépendamment et donc ne peut pas faire référence à tout autre élément de la clause FROM.)

LATERAL peut aussi précéder un élément fonction dans la clause FROM mais dans ce cas, ce n'est pas requis car l'expression de la fonction peut faire référence aux éléments du FROM dans tous les cas.

Un élément LATERAL peut apparaître au niveau haut dans la liste FROM ou à l'intérieur d'un arbre JOIN. Dans ce dernier cas, il peut aussi faire référence à tout élément qui se trouvent à la gauche d'un JOIN qui est à sa droite.

Quand un élément du FROM des références LATERAL, l'évaluation se fait ainsi : pour chaque ligne d'un élément FROM fournissant une colonne référencée ou un ensemble de lignes provenant de plusieurs éléments FROM fournissant les colonnes, l'élément LATERAL est évaluée en utilisant la valeur des colonnes de cette (ou ces) ligne(s). Les lignes résultantes sont jointes comme d'habitude avec les lignes pour lesquelles elles ont été calculées. Ceci est répété pour chaque ligne ou chaque ensemble de lignes provenant de la table contenant les colonnes référencées.

Le(s) table(s) contenant les colonnes référencées doivent être jointes avec INNER ou LEFT à l'élément LATERAL. Sinon il n'y aurait pas un ensemble bien défini de lignes à partir duquel on pourrait construire chaque ensemble de lignes pour l'élément LATERAL. Du coup, bien qu'une construction comme *X* RIGHT JOIN LATERAL *Y* est valide syntaxiquement, il n'est pas permis à *Y* de référencer *X*.

Clause WHERE

La clause WHERE optionnelle a la forme générale

```
WHERE condition
```

où *condition* est une expression dont le résultat est de type boolean. Toute ligne qui ne satisfait pas cette condition est éliminée de la sortie. Une ligne satisfait la condition si elle retourne vrai quand les valeurs réelles de la ligne sont substituées à toute référence de variable.

Clause GROUP BY

La clause GROUP BY optionnelle a la forme générale

```
GROUP BY [ ALL | DISTINCT ] element_regroupement [, ...]
```

GROUP BY condensera en une seule ligne toutes les lignes sélectionnées partageant les mêmes valeurs pour les expressions regroupées. Une *expression* utilisée à l'intérieur d'un *element_regroupement* peut être un nom de colonne en entrée, ou le nom ou le numéro d'une colonne en sortie (élément de la liste SELECT), ou une expression arbitraire formée à partir des valeurs ou colonnes en entrée. En cas d'ambiguité, un nom GROUP BY sera interprété comme un nom de colonne en entrée plutôt qu'en tant que nom de colonne en sortie.

Si une clause parmi GROUPING SETS, ROLLUP ou CUBE est présente comme élément de regroupement, alors la clause GROUP BY dans sa globalité définit un certain nombre d'*ensembles de regroupement* indépendants. L'effet de ceci est l'équivalent de la construction d'un UNION ALL des sous-requêtes pour chaque ensemble de regroupement individuel avec leur propre clause GROUP BY. La clause optionnelle DISTINCT supprime les ensembles dupliqués avant le traitement ; elle ne transforme *pas* le UNION ALL en un UNION DISTINCT. Pour plus de détails sur la gestion des ensembles de regroupement, voir Section 7.2.4.

Les fonctions d'agrégat, si utilisées, sont calculées pour toutes les lignes composant un groupe, produisant une valeur séparée pour chaque groupe. (S'il y a des fonctions d'agrégat mais pas de clause GROUP BY, la requête est traitée comme ayant un seul groupe contenant toutes les lignes sélectionnées.) L'ensemble de lignes envoyées à la fonction d'agrégat peut être en plus filtré en ajoutant une clause FILTER lors de l'appel à la fonction d'agrégat ; voir Section 4.2.7 pour plus d'informations. Quand une clause FILTER est présente, seules les lignes correspondant au filtre sont incluses en entrée de cette fonction d'agrégat.

Quand GROUP BY est présent ou que des fonctions d'agrégat sont présentes, les expressions du SELECT ne peuvent faire référence qu'à des colonnes groupées, sauf à l'intérieur de fonctions

d'agrégat, ou bien si la colonne non groupée dépend fonctionnellement des colonnes groupées. En effet, s'il en était autrement, il y aurait plus d'une valeur possible pour la colonne non groupée. Une dépendance fonctionnelle existe si les colonnes groupées (ou un sous-ensemble de ces dernières) sont la clé primaire de la table contenant les colonnes non groupées.

Rappelez-vous que toutes les fonctions d'agrégat sont évaluées avant l'évaluation des expressions « scalaires » dans la clause HAVING ou la liste SELECT. Ceci signifie que, par exemple, une expression CASE ne peut pas être utilisée pour ignorer l'évaluation de la fonction d'agrégat ; voir Section 4.2.14.

Actuellement, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE et FOR KEY SHARE ne peuvent pas être spécifiées avec GROUP BY.

Clause HAVING

La clause optionnelle HAVING a la forme générale

```
HAVING condition
```

où *condition* est identique à celle spécifiée pour la clause WHERE.

HAVING élimine les lignes groupées qui ne satisfont pas à la condition. HAVING est différent de WHERE : WHERE filtre les lignes individuelles avant l'application de GROUP BY alors que HAVING filtre les lignes groupées créées par GROUP BY. Chaque colonne référencée dans *condition* doit faire référence sans ambiguïté à une colonne groupée, sauf si la référence apparaît dans une fonction d'agrégat ou que les colonnes non groupées sont fonctionnement dépendantes des colonnes groupées.

Même en l'absence de clause GROUP BY, la présence de HAVING transforme une requête en requête groupée. Cela correspond au comportement d'une requête contenant des fonctions d'agrégats mais pas de clause GROUP BY. Les lignes sélectionnées ne forment qu'un groupe, la liste du SELECT et la clause HAVING ne peuvent donc faire référence qu'à des colonnes à l'intérieur de fonctions d'agrégats. Une telle requête ne produira qu'une seule ligne si la condition HAVING est réalisée, aucune dans le cas contraire.

Actuellement, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE et FOR KEY SHARE ne peuvent pas être spécifiées avec GROUP BY.

Clause WINDOW

La clause optionnelle WINDOW a la forme générale

```
WINDOW nom_window AS ( définition_window ) [, ...]
```

où *nom_window* est un nom qui peut être référencé par des clauses OVER ou par des définitions Window, et *définition_window* est

```
[ nom_window_existante ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operateur ] [ NULLS
 { FIRST | LAST } ] [, ...] ]
[ clause_frame ]
```

Si un *nom_window_existante* est spécifié, il doit se référer à une entrée précédente dans la liste WINDOW ; la nouvelle Window copie sa clause de partitionnement de cette entrée, ainsi que sa clause

de tri s'il y en a. Dans ce cas, la nouvelle Window ne peut pas spécifier sa propre clause PARTITION BY, et ne peut spécifier de ORDER BY que si la Window copiée n'en a pas. La nouvelle Window utilise toujours sa propre clause frame ; la Window copiée ne doit pas posséder de clause frame.

Les éléments de la liste PARTITION BY sont interprétés à peu près de la même façon que des éléments d'une clause GROUP BY, sauf qu'ils sont toujours des expressions simples et jamais le nom ou le numéro d'une colonne en sortie. Une autre différence est que ces expressions peuvent contenir des appels à des fonctions d' agrégat, ce qui n'est pas autorisé dans une clause GROUP BY classique. Ceci est autorisé ici parce que le windowing se produit après le regroupement et l' agrégation.

De façon similaire, les éléments de la liste ORDER BY sont interprétés à peu près de la même façon que les éléments d'un ORDER BY, sauf que les expressions sont toujours prises comme de simples expressions et jamais comme le nom ou le numéro d'une colonne en sortie.

La clause *clause_frame* optionnelle définit la *frame window* pour les fonctions window qui dépendent de la frame (ce n'est pas le cas de toutes). La frame window est un ensemble de lignes liées à chaque ligne de la requête (appelée la *ligne courante*). La *clause_frame* peut être une des clauses suivantes :

```
{ RANGE | ROWS | GROUPS } début_portée [ exclusion_portée ]
{ RANGE | ROWS | GROUPS } BETWEEN début_portée AND fin_portée
 [ exclusion_portée ]
```

où *début_frame* et *fin_frame* peuvent valoir

```
UNBOUNDED PRECEDING
décalage PRECEDING
CURRENT ROW
décalage FOLLOWING
UNBOUNDED FOLLOWING
```

et *exclusion_portée* peut valoir

```
EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS
```

Si *fin_frame* n'est pas précisé, il vaut par défaut CURRENT ROW. Les restrictions sont les suivantes : *début_frame* ne peut pas valoir UNBOUNDED FOLLOWING, *fin_frame* ne peut pas valoir UNBOUNDED PRECEDING, et le choix *fin_frame* ne peut apparaître avant les options *frame_start* et *frame_end* que le choix *début_frame* -- par exemple RANGE BETWEEN CURRENT ROW AND *décalage* PRECEDING n'est pas permis.

L'option de portée par défaut est RANGE UNBOUNDED PRECEDING, qui est identique à RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW ; cela configure la portée à toutes les lignes du début de la partition jusqu'au dernier *peer* de la ligne courant (une ligne que la clause ORDER BY de fenêtrage considère équivalente à la ligne courante ; toutes les lignes sont dans ce cas s'il n'y a pas d'ORDER BY). En général, UNBOUNDED PRECEDING signifie que la portée comment avec la première ligne de la partition et, de façon similaire, UNBOUNDED FOLLOWING signifie que la portée se termine avec la dernière ligne de la partition, quelque soit le mot (RANGE, ROWS or GROUPS). Dans le mode ROWS, CURRENT ROW signifie que la portée commence ou se termine avec la ligne actuelle ; mais dans les modes RANGE et GROUPS, il signifie que la portée commence ou se termine avec le premier ou le dernier équivalent de la ligne courante d'après le tri ORDER BY. Les options *offset*

PRECEDING et *offset* FOLLOWING varient en signification suivant le mode de portée. Dans le mode ROWS, *offset* est un entier indiquant que la portée commence ou se termine par ce nombre de lignes avant ou après la ligne actuelle. Dans le mode GROUPS, *offset* est un entier indiquant que la portée commence ou se termine par ce nombre de groupes d'équivalents avant ou après le groupe d'équivalents de la ligne courante, où un *groupe d'équivalents* est un groupe de lignes équivalentes suivant la clause ORDER BY de fenêtrage. Dans le mode RANGE, l'utilisation de l'option *offset* requiert qu'il y ait exactement une colonne ORDER BY dans la définition de la fenêtre. Ensuite, la portée contient ces lignes dont la valeur de la colonne de tri n'est pas inférieur de *offset* (pour PRECEDING) ou supérieur (pour FOLLOWING) à la valeur de la colonne de tri de la ligne courante. Dans ces cas, le type de données de l'expression *offset* dépend du typ de données de la colonne de tri. Pour les colonnes numériques, il s'agit typiquement du même type que la colonne de tri. Pour les colonnes date/heure, il s'agit typiquement d'un `interval`. Dans tous les cas, la valeur de *offset* doit être non NULL et non négative. De plus, alors que *offset* n'a pas besoin d'être une simple constante, elle ne peut pas contenir des variables, des fonctions d'agrégat et des fonctions de fenêtrage.

L'option *exclusion_portée* autorise les lignes autour de la ligne courante d'être exclues de la portée, même si elles devraient être includes d'après les options de début et de fin de portée. EXCLUDE CURRENT ROW exclut la ligne courante de la portée. EXCLUDE GROUP exclut la ligne courante et ses équivalents de tri à partir de la portée. EXCLUDE TIES exclut tout équivalent de la ligne courante à partir de la portée, mais pas la ligne courante elle-même. EXCLUDE NO OTHERS indique seulement explicitement le comportement par défaut qui est de ne pas exclure la ligne courante et ses équivalents.

Notez que le mode ROWS peut produire des résultats inattendus si la clause ORDER BY ne trie pas les lignes de façon unique. Les modes RANGE et GROUPS sont conçus pour s'assurer que les lignes équivalents d'après le tri ORDER BY sont traitées de la même façon : toutes les lignes d'un groupe d'équivalent sera inclus dans la portée ou en sera exclus.

L'utilité d'une clause WINDOW est de spécifier le comportement des *fonctions window* apparaissant dans la clause SELECT ou la clause ORDER BY de la requête. Ces fonctions peuvent référencer les entrées de clauses WINDOW par nom dans leurs clauses OVER. Toutefois, il n'est pas obligatoire qu'une entrée de clause WINDOW soit référencée quelque part ; si elle n'est pas utilisée dans la requête, elle est simplement ignorée. Il est possible d'utiliser des fonctions window sans aucune clause WINDOW puisqu'une fonction window peut spécifier sa propre définition de window directement dans sa clause OVER. Toutefois, la clause WINDOW économise de la saisie quand la même définition window est utilisée pour plus d'une fonction window.

Actuellement, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE et FOR KEY SHARE ne peuvent pas être spécifiées avec GROUP BY.

Les fonctions window sont décrites en détail dans Section 3.5, Section 4.2.8 et Section 7.2.5.

Liste SELECT

La liste SELECT (entre les mots clés SELECT et FROM) spécifie les expressions qui forment les lignes en sortie de l'instruction SELECT. Il se peut que les expressions fassent référence aux colonnes traitées dans la clause FROM. En fait, en général, elles le font.

Comme pour une table, chaque colonne de sortie d'un SELECT a un nom. Dans un SELECT simple, ce nom est juste utilisé pour donner un titre à la colonne pour l'affichage, mais quand le SELECT est une sous-requête d'une requête plus grande, le nom est vu par la grande requête comme le nom de colonne de la table virtuelle produite par la sous-requête. Pour indiquer le nom à utiliser pour une colonne de sortie, écrivez AS *nom_de_sortie* après l'expression de la colonne. (Vous pouvez omettre AS seulement si le nom de colonne souhaité n'est pas un mot clé réservé par PostgreSQL (voir Annexe C). Pour vous protéger contre l'ajout futur d'un mot clé, il est recommandé que vous écriviez toujours AS ou que vous mettiez le nom de sortie entre guillemets. Si vous n'indiquez pas de nom de colonne, un nom est choisi automatiquement par PostgreSQL. Si l'expression de la colonne est une simple référence à une colonne alors le nom choisi est le même que le nom de la colonne. Dans les cas plus complexes, un nom de fonction ou de type peut être utilisé, ou le système peut opter pour un nom généré automatiquement tel que `?column?`.

Un nom de colonne de sortie peut être utilisé pour se référer à la valeur de la colonne dans les clauses ORDER BY et GROUP BY, mais pas dans la clauseWHERE ou HAVING ; à cet endroit, vous devez écrire l'expression.

* peut être utilisé, à la place d'une expression, dans la liste de sortie comme raccourci pour toutes les colonnes des lignes sélectionnées. De plus, *nom_table*.* peut être écrit comme raccourci pour toutes les colonnes de cette table. Dans ces cas, il est impossible de spécifier de nouveaux noms avec AS ; les noms des colonnes de sorties seront les même que ceux de la table.

Suivant le standard SQL, les expressions dans la liste en sortie doivent être calculées avant d'appliquer les clauses DISTINCT, ORDER BY et LIMIT. Ceci est évidemment nécessaire lors de l'utilisation de DISTINCT car, dans le cas contraire, il est difficile de distinguer les valeurs. Néanmoins, dans de nombreux cas, il est plus intéressant que les expressions en sortie soient calculées après les clauses ORDER BY et LIMIT, tout particulièrement si la liste en sortie contient des fonctions volatiles ou coûteuses. Avec ce comportement, l'ordre d'évaluation des fonctions est plus intuitive et il n'y aurait pas d'évaluations correspondant aux lignes n'apparaissant pas en sortie. PostgreSQL évaluera réellement les expressions en sortie après le tri et la limite, si tant est que ces expressions ne sont pas référencées dans les clauses DISTINCT, ORDER BY et GROUP BY. (En contre-exemple, SELECT f(x) FROM tab ORDER BY 1 doit forcément évaluer f(x) avant de réaliser le tri.) Les expressions en sortie contenant des fonctions renvoyant plusieurs lignes sont réellement évaluées après le tri et avant l'application de la limite, pour que LIMIT permette d'éviter l'exécution inutile de la fonction.

> **Note**
>
> Les versions de PostgreSQL antérieures à la 9.6 ne fournissaient pas de garantie sur la durée de l'évaluation des expressions en sortie par rapport aux tris et aux limites. Cela dépendait de la forme du plan d'exécution sélectionné.

Clause DISTINCT

Si SELECT DISTINCT est spécifié, toutes les lignes en double sont supprimées de l'ensemble de résultats (une ligne est conservée pour chaque groupe de doublons). SELECT ALL spécifie le contraire : toutes les lignes sont conservées. C'est l'option par défaut.

SELECT DISTINCT ON (*expression* [, ...]) conserve seulement la première ligne de chaque ensemble de lignes pour lesquelles le résultat de l'expression est identique. Les expressions DISTINCT ON expressions sont interprétées avec les mêmes règles que pour ORDER BY (voir ci-dessous). Notez que la « première ligne » de chaque ensemble est imprévisible, à moins que la clause ORDER BY ne soit utilisée, assurant ainsi que la ligne souhaitée apparaisse en premier. Par exemple :

```
SELECT DISTINCT ON (lieu) lieu, heure, rapport
    FROM rapport_météo
    ORDER BY lieu, heure DESC;
```

renvoie le rapport météo le plus récent de chaque endroit. Mais si nous n'avions pas utilisé ORDER BY afin de forcer le tri du temps dans le sens descendant des temps pour chaque endroit, nous aurions récupéré, pour chaque lieu, n'importe quel bulletin de ce lieu.

La (ou les) expression(s) DISTINCT ON doivent correspondre à l'expression (ou aux expressions) ORDER BY la(les) plus à gauche. La clause ORDER BY contient habituellement des expressions supplémentaires qui déterminent l'ordre des lignes au sein de chaque groupe DISTINCT ON.

Actuellement, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE et FOR KEY SHARE ne peuvent pas être spécifiées avec DISTINCT.

Clause UNION

La clause UNION a la forme générale :

instruction_select UNION [ALL | DISTINCT] *instruction_select*

instruction_select est une instruction SELECT sans clause ORDER BY, LIMIT, FOR SHARE ou FOR UPDATE. (ORDER BY et LIMIT peuvent être attachés à une sous-expression si elle est entourée de parenthèses. Sans parenthèses, ces clauses s'appliquent au résultat de l'UNION, non à l'expression à sa droite.)

L'opérateur UNION calcule l'union ensembliste des lignes renvoyées par les instructions SELECT impliquées. Une ligne est dans l'union de deux ensembles de résultats si elle apparaît dans au moins un des ensembles. Les deux instructions SELECT qui représentent les opérandes directes de l'UNION doivent produire le même nombre de colonnes et les colonnes correspondantes doivent être d'un type de données compatible.

Sauf lorsque l'option ALL est spécifiée, il n'y a pas de doublons dans le résultat de UNION. ALL empêche l'élimination des lignes dupliquées. UNION ALL est donc significativement plus rapide qu'UNION, et sera préféré. DISTINCT peut éventuellement être ajouté pour préciser explicitement le comportement par défaut : l'élimination des lignes en double.

Si une instruction SELECT contient plusieurs opérateurs UNION, ils sont évalués de gauche à droite, sauf si l'utilisation de parenthèses impose un comportement différent.

Actuellement, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE et FOR KEY SHARE ne peuvent pas être spécifiés pour un résultat d'UNION ou pour toute entrée d'un UNION.

Clause INTERSECT

La clause INTERSECT a la forme générale :

instruction_select INTERSECT [ALL | DISTINCT] *instruction_select*

instruction_select est une instruction SELECT sans clause ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE ou FOR KEY SHARE.

L'opérateur INTERSECT calcule l'intersection des lignes renvoyées par les instructions SELECT impliquées. Une ligne est dans l'intersection des deux ensembles de résultats si elle apparaît dans chacun des deux ensembles.

Le résultat d'INTERSECT ne contient aucune ligne dupliquée sauf si l'option ALL est spécifiée. Dans ce cas, une ligne dupliquée *m* fois dans la table gauche et *n* fois dans la table droite apparaît min(*m*,*n*) fois dans l'ensemble de résultats. DISTINCT peut éventuellement être ajouté pour préciser explicitement le comportement par défaut : l'élimination des lignes en double.

Si une instruction SELECT contient plusieurs opérateurs INTERSECT, ils sont évalués de gauche à droite, sauf si l'utilisation de parenthèses impose un comportement différent. INTERSECT a une priorité supérieur à celle d'UNION. C'est-à-dire que A UNION B INTERSECT C est lu comme A UNION (B INTERSECT C).

Actuellement, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE et FOR KEY SHARE ne peuvent pas être spécifiés pour un résultat d'INTERSECT ou pour une entrée d'INTERSECT.

Clause EXCEPT

La clause EXCEPT a la forme générale :

instruction_select EXCEPT [ALL | DISTINCT] *instruction_select*

instruction_select est une instruction SELECT sans clause ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE ou FOR KEY SHARE.

L'opérateur EXCEPT calcule l'ensemble de lignes qui appartiennent au résultat de l'instruction SELECT de gauche mais pas à celui de droite.

Le résultat d'EXCEPT ne contient aucune ligne dupliquée sauf si l'option ALL est spécifiée. Dans ce cas, une ligne dupliquée *m* fois dans la table gauche et *n* fois dans la table droite apparaît max(*m*-*n*,0) fois dans l'ensemble de résultats. DISTINCT peut éventuellement être ajouté pour préciser explicitement le comportement par défaut : l'élimination des lignes en double.

Si une instruction SELECT contient plusieurs opérateurs EXCEPT, ils sont évalués de gauche à droite, sauf si l'utilisation de parenthèses impose un comportement différent. EXCEPT a la même priorité qu'UNION.

Actuellement, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE et FOR KEY SHARE ne peuvent pas être spécifiés dans un résultat EXCEPT ou pour une entrée d'un EXCEPT.

Clause ORDER BY

La clause optionnelle ORDER BY a la forme générale :

```
ORDER BY expression [ ASC | DESC | USING opérateur ] [ NULLS
 { FIRST | LAST } ] [, ...]
```

La clause ORDER BY impose le tri des lignes de résultat suivant les expressions spécifiées. Si deux lignes sont identiques suivant l'expression la plus à gauche, elles sont comparées avec l'expression suivante et ainsi de suite. Si elles sont identiques pour toutes les expressions de tri, elles sont renvoyées dans un ordre dépendant de l'implantation.

Chaque *expression* peut être le nom ou le numéro ordinal d'une colonne en sortie (élément de la liste SELECT). Elle peut aussi être une expression arbitraire formée à partir de valeurs des colonnes.

Le numéro ordinal fait référence à la position ordinale (de gauche à droite) de la colonne de résultat. Cette fonctionnalité permet de définir un ordre sur la base d'une colonne dont le nom n'est pas unique. Ce n'est pas particulièrement nécessaire parce qu'il est toujours possible d'affecter un nom à une colonne de résultat avec la clause AS.

Il est aussi possible d'utiliser des expressions quelconques dans la clause ORDER BY, ce qui inclut des colonnes qui n'apparaissent pas dans la liste résultat du SELECT. Ainsi, l'instruction suivante est valide :

```
SELECT nom FROM distributeurs ORDER BY code;
```

Il y a toutefois une limitation à cette fonctionnalité. La clause ORDER BY qui s'applique au résultat d'une clause UNION, INTERSECT ou EXCEPT ne peut spécifier qu'un nom ou numéro de colonne en sortie, pas une expression.

Si une expression ORDER BY est un nom qui correspond à la fois à celui d'une colonne résultat et à celui d'une colonne en entrée, ORDER BY l'interprète comme le nom de la colonne résultat. Ce comportement est à l'opposé de celui de GROUP BY dans la même situation. Cette incohérence est imposée par la compatibilité avec le standard SQL.

Un mot clé ASC (ascendant) ou DESC (descendant) peut être ajouté après toute expression de la clause ORDER BY. ASC est la valeur utilisée par défaut. Un nom d'opérateur d'ordre spécifique peut également être fourni dans la clause USING. Un opérateur de tri doit être un membre plus-petit-que ou plus-grand-que de certaines familles d'opérateur B-tree. ASC est habituellement équivalent à USING

< et DESC à USING >. Le créateur d'un type de données utilisateur peut définir à sa guise le tri par défaut qui peut alors correspondre à des opérateurs de nom différent.

Si NULLS LAST est indiqué, les valeurs NULL sont listées après toutes les valeurs non NULL si NULLS FIRST est indiqué, les valeurs NULL apparaissent avant toutes les valeurs non NULL. Si aucune des deux n'est présente, le comportement par défaut est NULLS LAST quand ASC est utilisé (de façon explicite ou non) et NULLS FIRST quand DESC est utilisé (donc la valeur par défaut est d'agir comme si les NULL étaient plus grands que les non NULL). Quand USING est indiqué, le tri des NULL par défaut dépend du fait que l'opérateur est un plus-petit-que ou un plus-grand-que.

Notez que les options de tri s'appliquent seulement à l'expression qu'elles suivent. Par exemple, ORDER BY x, y DESC ne signifie pas la même chose que ORDER BY x DESC, y DESC.

Les chaînes de caractères sont triées suivant le collationnement qui s'applique à la colonne triée. Ce collationnement est surchargeable si nécessaire en ajoutant une clause COLLATE dans l'*expression*, par exemple ORDER BY mycolumn COLLATE "en_US". Pour plus d'informations, voir Section 4.2.10 et Section 24.2.

Clause LIMIT

La clause LIMIT est constituée de deux sous-clauses indépendantes :

```
LIMIT { nombre | ALL }
OFFSET début
```

nombre spécifie le nombre maximum de lignes à renvoyer alors que *début* spécifie le nombre de lignes à passer avant de commencer à renvoyer des lignes. Lorsque les deux clauses sont spécifiées, *début* lignes sont passées avant de commencer à compter les *nombre* lignes à renvoyer.

Si l'expression de *compte* est évaluée à NULL, il est traité comme LIMIT ALL, c'est-à-dire sans limite. Si *début* est évalué à NULL, il est traité comme OFFSET 0.

SQL:2008 a introduit une sytaxe différente pour obtenir le même résultat. PostgreSQL supporte aussi cette syntaxe.

```
OFFSET début { ROW | ROWS }
FETCH { FIRST | NEXT } [ compte ] { ROW | ROWS } { ONLY | WITH
 TIES }
```

Avec cette syntaxe, le standard SQL exige que la valeur de *start* ou *count* soit une constante litérale, un paramètre ou un nom de variable. PostgreSQL propose en extension l'utilisation d'autres expressions. Ces dernières devront généralement être entre parenthèses pour éviter toute ambiguïté. Si *compte* est omis dans une clause FETCH, il vaut 1 par défaut. L'option WITH TIES est utilisée pour renvoyer toute ligne supplémentaire qui se lie pour la dernière place dans le jeu de résultats selon la clause ORDER BY ; ORDER BY est requis dans ce cas, et SKIP LOCKED n'est pas autorisé. ROW et ROWS ainsi que FIRST et NEXT sont des mots qui n'influencent pas les effets de ces clauses. D'après le standard, la clause OFFSET doit venir avant la clause FETCH si les deux sont présentes ; PostgreSQL est plus laxiste et autorise un ordre différent.

Avec LIMIT, utiliser la clause ORDER BY permet de contraindre l'ordre des lignes de résultat. Dans le cas contraire, le sous-ensemble obtenu n'est pas prévisible -- rien ne permet de savoir à quel ordre correspondent les lignes retournées. Celui-ci ne sera pas connu tant qu'ORDER BY n'aura pas été précisé.

Lors de la génération d'un plan de requête, le planificateur tient compte de LIMIT. Le risque est donc grand d'obtenir des plans qui diffèrent (ordres des lignes différents) suivant les valeurs utilisées pour LIMIT et OFFSET. Ainsi, sélectionner des sous-ensembles différents d'un résultat à partir de valeurs différentes de LIMIT/OFFSET *aboutit à des résultats incohérents* à moins d'avoir figé l'ordre des

lignes à l'aide de la clause ORDER BY. Ce n'est pas un bogue, mais une conséquence du fait que SQL n'assure pas l'ordre de présentation des résultats sans utilisation d'une clause ORDER BY.

Il est même possible pour des exécutions répétées de la même requête LIMIT de renvoyer différents sous-ensembles des lignes d'une table s'il n'y a pas de clause ORDER BY pour forcer la sélection d'un sous-ensemble déterministe. Encore une fois, ce n'est pas un bogue ; le déterminisme des résultats n'est tout simplement pas garanti dans un tel cas.

Clause de verrouillage

FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE et FOR KEY SHARE sont des *clauses de verrouillage*. Elles affectent la façon dont SELECT verrouille les lignes au moment de leur obtention sur la table.

La clause de verrouillage a la forme suivante :

```
FOR force_verrou [ OF nom_table [, ...] ] [ NOWAIT | SKIP LOCKED ]
```

où *force_verrou* fait partie de :

```
UPDATE
NO KEY UPDATE
SHARE
KEY SHARE
```

Pour plus d'informations sur chaque mode de verrouillage au niveau ligne, voir Section 13.3.2.

Pour éviter que l'opération attende la validation d'autres transactions, utilisez soit l'option NOWAIT soit l'option SKIP LOCKED. Avec NOWAIT, l'instruction renvoie une erreur, plutôt que de rester en attente, si une ligne sélectionnée ne peut pas être immédiatement verrouillée. Avec SKIP LOCKED, toute ligne sélectionnée qui ne peut pas être immédiatement verrouillée est ignorée. Ignorer les lignes verrouillées fournit une vue incohérente des données, donc ce n'est pas acceptable dans un cadre général, mais ça peut être utilisé pour éviter les contentions de verrou lorsque plusieurs consommateurs cherchent à accéder à une table de style queue. Notez que NOWAIT et SKIP LOCKED s'appliquent seulement au(x) verrou(x) niveau ligne -- le verrou niveau table ROW SHARE est toujours pris de façon ordinaire (voir Chapitre 13). L'option NOWAIT de LOCK peut toujours être utilisée pour acquérir le verrou niveau table sans attendre.

Si des tables particulières sont nommées dans une clause de verrouillage, alors seules les lignes provenant de ces tables sont verrouillées ; toute autre table utilisée dans le SELECT est simplement lue. Une clause de verrouillage sans liste de tables affecte toutes les tables utilisées dans l'instruction. Si une clause de verrouillage est appliquée à une vue ou à une sous-requête, cela affecte toutes les tables utilisées dans la vue ou la sous-requête. Néanmoins, ces clauses ne s'appliquent pas aux requêtes WITH référencées par la clé primaire. Si vous voulez qu'un verrouillage de lignes intervienne dans une requête WITH, spécifiez une clause de verrouillage à l'intérieur de la requête WITH.

Plusieurs clauses de verrouillage peuvent être données si il est nécessaire de spécifier différents comportements de verrouillage pour différentes tables. Si la même table est mentionné (ou affectée implicitement) par plus d'une clause de verrouillage, alors elle est traitée comme la clause la plus forte. De façon similaire, une table est traitée avec NOWAIT si c'est spécifiée sur au moins une des clauses qui l'affectent. Sinon, il est traité comme SKIP LOCKED si c'est indiqué dans une des clauses qui l'affectent.

Les clauses de verrouillage nécessitent que chaque ligne retournée soit clairement identifiable par une ligne individuelle d'une table ; ces options ne peuvent, par exemple, pas être utilisées avec des fonctions d'agrégats.

Quand une clause de verrouillage apparaissent au niveau le plus élevé d'une requête SELECT, les lignes verrouillées sont exactement celles qui sont renvoyées par la requête ; dans le cas d'une requête avec jointure, les lignes verrouillées sont celles qui contribuent aux lignes jointes renvoyées. De plus, les lignes qui ont satisfait aux conditions de la requête au moment de la prise de son instantané sont verrouillées, bien qu'elles ne seront pas retournées si elles ont été modifiées après la prise du snapshot et ne satisfont plus les conditions de la requête. Si LIMIT est utilisé, le verrouillage cesse une fois que suffisamment de lignes ont été renvoyées pour satisfaire la limite (mais notez que les lignes ignorées à cause de la clause OFFSET seront verrouillées). De la même manière, si une clause de verrouillage est utilisé pour la requête d'un curseur, seules les lignes réellement récupérées ou parcourues par le curseur seront verrouillées.

Si une clause de verrouillage apparait dans un sous-SELECT, les lignes verrouillées sont celles renvoyées par la sous-requête à la requête externe. Cela peut concerner moins de lignes que l'étude de la sous-requête seule pourrait faire penser, parce que les conditions de la requête externe peuvent être utilisées pour optimiser l'exécution de la sous-requête. Par exemple,

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

verrouillera uniquement le lignes pour lesquelles `col1 = 5`, même si cette condition n'est pas écrite dans la sous-requête.

Les anciennes versions échouaient à préserver un verrou qui est mis à jour par un point de sauvegarde ultérieur. Par exemple, ce code :

```
BEGIN;
SELECT * FROM ma_table WHERE cle = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE ma_table SET ... WHERE cle = 1;
ROLLBACK TO s;
```

va échouer à conserver le verrou FOR UPDATE après la commande ROLLBACK TO. Ceci a été corrigé en 9.3.

Attention

Il est possible qu'une commande SELECT exécutée au niveau d'isolation READ COMMITTED et utilisant ORDER BY et une clause de verrouillage renvoie les lignes dans le désordre. C'est possible car l' ORDER BY est appliqué en premier. La commande trie le résultat, mais peut alors être bloquée le temps d'obtenir un verrou sur une ou plusieurs des lignes. Une fois que le SELECT est débloqué, des valeurs sur la colonne qui sert à ordonner peuvent avoir été modifiées, ce qui entraîne ces lignes apparaissant dans le désordre (bien qu'elles soient dans l'ordre par rapport aux valeurs d'origine de ces colonnes). Ceci peut être contourné si besoin en plaçant la clause FOR UPDATE/SHARE dans une sous-requête, par exemple

```
SELECT * FROM (SELECT * FROM matable FOR UPDATE) ss ORDER BY
 column1;
```

Notez que cela entraîne le verrouillage de toutes les lignes de matable, alors que FOR UPDATE au niveau supérieur verrouillerait seulement les lignes réellement renvoyées. Cela peut causer une différence de performance significative, en particulier si l' ORDER BY est combiné avec LIMIT ou d'autres restrictions. Cette technique est donc recommandée uniquement si vous vous attendez à des mises à jour concurrentes sur les colonnes servant à l'ordonnancement et qu'un résultat strictement ordonné est requis.

> Au niveau d'isolation de transactions REPEATABLE READ et SERIALIZABLE, cela causera une erreur de sérialisation (avec un SQLSTATE valant '40001'), donc il n'est pas possible de recevoir des lignes non triées avec ces niveaux d'isolation.

Commande TABLE

La commande

```
TABLE nom
```

est équivalente à

```
SELECT * FROM nom
```

Elle peut être utilisée comme commande principale d'une requête, ou bien comme une variante syntaxique permettant de gagner de la place dans des parties de requêtes complexes. Seuls les clauses de verrou de WITH, UNION, INTERSECT, EXCEPT, ORDER BY, LIMIT, OFFSET, FETCH et FOR peuvent être utilisées avec TABLE ; la clause WHERE et toute forme d'agrégation ne peuvent pas être utilisées.

Exemples

Joindre la table films avec la table distributeurs :

```
SELECT f.titre, f.did, d.nom, f.date_prod, f.genre
    FROM distributeurs d JOIN films f USING (did);
```

```
       titre       | did |     nom      | date_prod  |   genre
-------------------+-----+--------------+------------+------------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drame
 The African Queen | 101 | British Lion | 1951-08-11 | Romantique
 ...
```

Additionner la colonne longueur de tous les films, grouper les résultats par genre :

```
SELECT genre, sum(longueur) AS total FROM films GROUP BY genre;
```

```
   genre    | total
------------+-------
 Action     | 07:34
 Comédie    | 02:58
 Drame      | 14:28
 Musical    | 06:42
 Romantique | 04:38
```

Additionner la colonne longueur de tous les films, grouper les résultats par genre et afficher les groupes dont les totaux font moins de cinq heures :

```
SELECT genre, sum(longueur) AS total
    FROM films
    GROUP BY genre
```

```
    HAVING sum(longueur) < interval '5 hours';

   genre    | total
------------+-------
 Comedie    | 02:58
 Romantique | 04:38
```

Les deux exemples suivants représentent des façons identiques de trier les résultats individuels en fonction du contenu de la deuxième colonne (`nom`) :

```
SELECT * FROM distributeurs ORDER BY nom;
SELECT * FROM distributeurs ORDER BY 2;

 did |       nom
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

L'exemple suivant présente l'union des tables `distributeurs` et `acteurs`, restreignant les résultats à ceux de chaque table dont la première lettre est un W. Le mot clé `ALL` est omis, ce qui permet de n'afficher que les lignes distinctes.

```
distributeurs:               acteurs:
 did |     nom              id |     nom
-----+--------------        ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                          ...

SELECT distributeurs.nom
    FROM distributeurs
    WHERE distributeurs.nom LIKE 'W%'
UNION
SELECT actors.nom
    FROM acteurs
    WHERE acteurs.nom LIKE 'W%';

      nom
---------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

L'exemple suivant présente l'utilisation d'une fonction dans la clause `FROM`, avec et sans liste de définition de colonnes :

```
CREATE FUNCTION distributeurs(int) RETURNS SETOF distributeurs AS $
$
    SELECT * FROM distributeurs WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributeurs(111);
 did |    name
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributeurs_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributeurs WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributeurs_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

Voici un exemple d'une fonction avec la colonne ordinality :

```
SELECT * FROM unnest(ARRAY['a','b','c','d','e','f']) WITH
 ORDINALITY;
 unnest | ordinality
--------+----------
 a      |        1
 b      |        2
 c      |        3
 d      |        4
 e      |        5
 f      |        6
(6 rows)
```

Cet exemple montre comment utiliser une clause `WITH` simple:

```
WITH t AS (
    SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM t
UNION ALL
SELECT * FROM t;

         x
--------------------
  0.534150459803641
  0.520092216785997
 0.0735620250925422
  0.534150459803641
  0.520092216785997
 0.0735620250925422
```

Notez que la requête `WITH` n'a été évaluée qu'une seule fois, ce qui fait qu'on a deux jeux contenant les mêmes trois valeurs.

Cet exemple utilise `WITH RECURSIVE` pour trouver tous les subordonnés (directs ou indirects) de l'employée Marie, et leur niveau de subordination, à partir d'une table qui ne donne que les subordonnés directs :

```
WITH RECURSIVE recursion_employes(distance, nom_employe,
 nom_manager) AS (
    SELECT 1, nom_employe, nom_manager
    FROM employe
    WHERE nom_manager = 'Marie'
  UNION ALL
    SELECT er.distance + 1, e.nom_employe, e.nom_manager
    FROM recursion_employes er, employe e
    WHERE er.nom_employe = e.nom_manager
  )
SELECT distance, nom_employe FROM recursion_employes;
```

Notez la forme typique des requêtes récursives : une condition initiale, suivie par `UNION`, suivis par la partie récursive de la requête. Assurez-vous que la partie récursive de la requête finira par ne plus retourner d'enregistrement, sinon la requête bouclera indéfiniment (Voir Section 7.8 pour plus d'exemples).

Cet exemple utilise `LATERAL` pour appliquer une fonction renvoyant des lignes, `recupere_nom_produits()`, pour chaque ligne de la table `manufacturiers` :

```
SELECT m.nom AS mnom, pnom
FROM manufacturiers m, LATERAL recupere_nom_produits(m.id) pnom;
```

Les manufacturiers qui n'ont pas encore de produits n'apparaîtront pas dans le résultat car la jointure est interne. Si vous voulons inclure les noms de ces manufacturiers, la requête doit être écrite ainsi :

```
SELECT m.name AS mnom, pnom
FROM manufacturiers m LEFT JOIN LATERAL recupere_nom_produits(m.id)
 pnom ON true;
```

Compatibilité

L'instruction `SELECT` est évidemment compatible avec le standard SQL. Mais il y a des extensions et quelques fonctionnalités manquantes.

Clauses `FROM` omises

PostgreSQL autorise l'omission de la clause `FROM`. Cela permet par exemple de calculer le résultat d'expressions simples :

```
SELECT 2+2;

 ?column?
----------
```

```
4
```

D'autres bases de données SQL interdisent ce comportement, sauf à introduire une table virtuelle d'une seule ligne sur laquelle exécuter la commande `SELECT`.

Listes SELECT vides

La liste des expressions en sortie après `SELECT` peut être vide, produisant ainsi une table de résultats à zéro colonne. Ceci n'est pas une syntaxe valide suivant le standard SQL. PostgreSQL l'autorise pour être cohérent avec le fait qu'il accepte des tables à zéro colonne. Néanmoins, une liste vide n'est pas autorisé quand un `DISTINCT` est utilisé.

Omettre le mot clé AS

Dans le standard SQL, le mot clé `AS` peut être omis devant une colonne de sortie à partir du moment où le nouveau nom de colonne est un nom valide de colonne (c'est-à-dire, différent d'un mot clé réservé). PostgreSQL est légèrement plus restrictif : `AS` est nécessaire si le nouveau nom de colonne est un mot clé quel qu'il soit, réservé ou non. Il est recommandé d'utiliser `AS` ou des colonnes de sortie entourées de guillemets, pour éviter tout risque de conflit en cas d'ajout futur de mot clé.

Dans les éléments de `FROM`, le standard et PostgreSQL permettent que `AS` soit omis avant un alias qui n'est pas un mot clé réservé. Mais c'est peu pratique pour les noms de colonnes, à causes d'ambiguïtés syntaxiques.

ONLY et l'héritage

Le standard SQL impose des parenthèses autour du nom de table après la clause `ONLY`, comme dans `SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE ...`. PostgreSQL considère les parenthèses comme étant optionnelles.

PostgreSQL autorise une `*` en fin pour indiquer explicitement le comportement opposé de la clause `ONLY` (donc inclure les tables filles). Le standard ne le permet pas.

(Ces points s'appliquent de la même façon à toutes les commandes SQL supportant l'option `ONLY`.)

Restrictions de la clause TABLESAMPLE

La clause `TABLESAMPLE` est actuellement seulement acceptée pour les tables standards et les vues matérialisées. D'après le standard SQL, il devrait être possible de l'appliquer à tout élément faisant partie de la clause `FROM`.

Appels de fonction dans la clause FROM

PostgreSQL autorise un appel de fonction dans la liste `FROM`. Pour le standard SQL, il serait nécessaire de placer cet appel de fonction dans un sous-`SELECT` ; autrement dit, la syntaxe `FROM` *`fonc`*`(...)` *`alias`* est à peu près équivalente à `FROM LATERAL (SELECT` *`fonc`*`(...))` *`alias`*. Notez que `LATERAL` est considéré comme étant implicite ; ceci est dû au fait que le standard réclame la sémantique de `LATERAL` pour un élément `UNNEST()` dans la clause `FROM`. PostgreSQL traite `UNNEST()` de la même façon que les autres fonctions renvoyant des lignes.

Espace logique disponible pour GROUP BY et ORDER BY

Dans le standard SQL-92, une clause `ORDER BY` ne peut utiliser que les noms ou numéros des colonnes en sortie, une clause `GROUP BY` que des expressions fondées sur les noms de colonnes en entrée. PostgreSQL va plus loin, puisqu'il autorise chacune de ces clauses à utiliser également l'autre possibilité. En cas d'ambiguïté, c'est l'interprétation du standard qui prévaut. PostgreSQL autorise aussi l'utilisation d'expressions quelconques dans les deux clauses. Les noms apparaissant dans ces

expressions sont toujours considérés comme nom de colonne en entrée, pas en tant que nom de colonne du résultat.

SQL:1999 et suivant utilisent une définition légèrement différente, pas totalement compatible avec le SQL-92. Néanmoins, dans la plupart des cas, PostgreSQL interprète une expression `ORDER BY` ou `GROUP BY` en suivant la norme SQL:1999.

Dépendances fonctionnelles

PostgreSQL reconnaît les dépendances fonctionnelles (qui permettent que les nom des colonnes ne soient pas dans le `GROUP BY`) seulement lorsqu'une clé primaire est présente dans la liste du `GROUP BY`. Le standard SQL spécifie des configurations supplémentaires qui doivent être reconnues.

`LIMIT` et `OFFSET`

Les clauses `LIMIT` et `OFFSET` sont une syntaxe spécifique à PostgreSQL, aussi utilisée dans MySQL. La norme SQL:2008 a introduit les clauses `OFFSET ... FETCH {FIRST|NEXT}...` pour la même fonctionnalité, comme montré plus haut dans Clause LIMIT. Cette syntaxe est aussi utilisée par IBM DB2. (Les applications écrites pour Oracle contournent fréquemment le problème par l'utilisation de la colonne auto-générée `rownum` pour obtenir les effets de ces clauses, qui n'est pas disponible sous PostgreSQL).

`FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE`, `FOR KEY SHARE`

Bien que `FOR UPDATE` soit présent dans le standard SQL, le standard ne l'autorise que comme une option de `DECLARE CURSOR`. PostgreSQL l'autorise dans toute requête `SELECT` et dans toute sous-requête `SELECT`, mais c'est une extension. Les variantes `FOR NO KEY UPDATE`, `FOR SHARE` et `FOR KEY SHARE`, ainsi que `NOWAIT` et `SKIP LOCKED`, n'apparaissent pas dans le standard.

Ordre de modification de données dans un `WITH`

PostgreSQL permet que les clauses `INSERT`, `UPDATE`, et `DELETE` soient utilisées comme requêtes `WITH`. Ceci n'est pas présent dans le standard SQL.

Clauses non standard

La clause `DISTINCT ON` est une extension du standard SQL.

`ROWS FROM( ... )` est une extension du standard SQL.

Les options `MATERIALIZED` et `NOT MATERIALIZED` de la clause `WITH` sont des extensions au standard SQL.

SELECT INTO

SELECT INTO — définit une nouvelle table à partir des résultats d'une requête

Synopsis

```
[ WITH [ RECURSIVE ] requête_with [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ { * | expression [ [ AS ] nom_sortie ] } [, ...] ]
    INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] nouvelle_table
    [ FROM élément_from [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition ]
    [ WINDOW nom_window AS ( définition_window ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING opérateur ]
 [, ...] ]
    [ LIMIT { nombre | ALL } ]
    [ OFFSET début [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ nombre ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF nomtable [, ...] ] [ NOWAIT ]
 [...] ]
```

Description

`SELECT INTO` crée une nouvelle table en la remplissant avec des données récupérées par une requête. Les données ne sont pas renvoyées au client comme le fait habituellement l'instruction `SELECT`. Les nouvelles colonnes de la table ont les noms et les types de données associés avec les colonnes en sortie du `SELECT`.

Paramètres

`TEMPORARY` ou `TEMP`

Si spécifié, la table est créée comme une table temporaire. Référez-vous à CREATE TABLE pour plus de détails.

`UNLOGGED`

Si spécifié, la table est créée comme une table non tracée dans les journaux de transactions. Voir CREATE TABLE pour plus de détails.

`new_table`

Le nom de la table à créer (pouvant être qualifié par le nom du schéma).

Tous les autres paramètres sont décrits en détail dans SELECT.

Notes

`CREATE TABLE AS` est fonctionnellement équivalent à `SELECT INTO`. `CREATE TABLE AS` est la syntaxe recommandée car cette forme de `SELECT INTO` n'est pas disponible dans ECPG ou PL/pgSQL. En effet, ils interprètent la clause `INTO` différemment. De plus, `CREATE TABLE AS` offre un ensemble de fonctionnalités plus important que celui de `SELECT INTO`.

En contraste à `CREATE TABLE AS`, `SELECT INTO` ne permet pas de spécifier les propriétés telles que la méthode d'accès à la table avec `USING` *`method`* ou le tablespace de la table avec `TABLESPACE` *`nom_tablespace`*. Utilisez `CREATE TABLE AS` si nécessaire. De ce fait, la méthode d'accès par défaut de la table est choisie pour la nouvelle table. Voir default_table_access_method pour plus d'informations.

Exemples

Crée une nouvelle table `films_recent` ne contenant que les entrées récentes de la table `films` :

```
SELECT * INTO films_recent FROM films WHERE date_prod >=
 '2002-01-01';
```

Compatibilité

Le standard SQL utilise `SELECT INTO` pour représenter la sélection de valeurs dans des variables scalaires d'un programme hôte plutôt que la création d'une nouvelle table. Ceci est en fait l'utilisation trouvée dans ECPG (voir Chapitre 36) et dans PL/pgSQL (voir Chapitre 43). L'usage de PostgreSQL de `SELECT INTO` pour représenter une création de table est historique. Certaines autres implémentations SQL utilisent aussi `SELECT INTO` de cette façon (mais la plupart des implémentations SQL acceptent `CREATE TABLE AS` à la place). En dehors de ces considérations de compatibilité, il est préférable d'utiliser `CREATE TABLE AS` dans un nouveau programme.

Voir aussi

CREATE TABLE AS

SET

SET — change un paramètre d'exécution

Synopsis

```
SET [ SESSION | LOCAL ] paramètre_configuration { TO | = } { valeur
 | 'valeur' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { valeur | 'valeur' | LOCAL |
 DEFAULT }
```

Description

La commande SET permet de modifier les paramètres d'exécution. Un grand nombre de paramètres d'exécution, listés dans Chapitre 20, peuvent être modifiés à la volée avec la commande SET. (Certains paramètres peuvent être modifiés uniquement par les superutilisateurs et les utilisateurs qui bénéficient du droit SET sur ce paramètre. Il existe aussi des paramètres qui ne peuvent pas être modifiés après le démarrage du serveur ou de la session.) SET ne modifie que les paramètres utilisés par la session courante.

Certains paramètres ne peuvent être modifiés que par le superutilisateur, d'autres ne peuvent plus être changés après le démarrage du serveur ou de la session.

Si SET ou SET SESSION sont utilisés dans une transaction abandonnée par la suite, les effets de la commande SET disparaissent dès l'annulation de la transaction. Lorsque la transaction englobant la commande est validée, les effets de la commande persistent jusqu'à la fin de la session, à moins qu'ils ne soient annulés par une autre commande SET.

Les effets de SET LOCAL ne durent que jusqu'à la fin de la transaction en cours, qu'elle soit validée ou non. Dans le cas particulier d'une commande SET suivie par SET LOCAL dans une même transaction, la valeur de SET LOCAL est utilisée jusqu'à la fin de la transaction, et celle de SET prend effet ensuite (si la transaction est validée).

Les effets de SET et SET LOCAL sont aussi annulés par le retour à un point de sauvegarde précédant la commande.

Si SET LOCAL est utilisé à l'intérieur d'une fonction qui comprend l'option SET pour la même variable (voir CREATE FUNCTION), les effets de la commande SET LOCAL disparaîtront à la sortie de la fonction ; en fait, la valeur disponible lors de l'appel de la fonction est restaurée de toute façon. Ceci permet l'utilisation de SET LOCAL pour des modifications dynamiques et répétées d'un paramètre à l'intérieur d'une fonction, avec l'intérêt d'utiliser l'option SET pour sauvegarder et restaurer la valeur de l'appelant. Néanmoins, une commande SET standard surcharge toute option SET de la fonction ; son effet persistera sauf en cas d'annulation.

> **Note**
>
> De PostgreSQL version 8.0 à 8.2, les effets de SET LOCAL sont annulés suite au relachement d'un point de sauvegarde précédent, ou par une sortie avec succès d'un bloc d'exception PL/pgSQL. Ce comportement a été modifié car il n'était pas du tout intuitif.

Paramètres

`SESSION`

Indique que la commande prend effet pour la session courante. C'est la valeur par défaut lorsque `SESSION` et `LOCAL` sont omis.

`LOCAL`

Indique que la commande n'est effective que pour la transaction courante. Utiliser cette option en dehors d'une transaction émet un avertissement et n'a aucun autre effet.

`paramètre_configuration`

Nom d'un paramètre ajustable pendant l'exécution. La liste des paramètres disponibles est documentée dans Chapitre 20 et ci-dessous.

`valeur`

Nouvelle valeur du paramètre. Les valeurs peuvent être indiquées sous forme de constantes de chaîne, d'identifiants, de nombres ou de listes de ceux-ci, séparées par des virgules, de façon approprié pour ce paramètre. `DEFAULT` peut être utilisé pour repositionner le paramètre à sa valeur par défaut (c'est-à-dire quelque soit la valeur qu'il aurait eu si aucun `SET` n'avait été exécuté lors de cette session).

En plus des paramètres de configuration documentés dans Chapitre 20, il y en a quelques autres qui ne peuvent être initialisés qu'avec la commande `SET` ou ont une syntaxe spéciale.

`SCHEMA`

`SET SCHEMA '`*`valeur`*`'` est un alias pour `SET search_path TO` *`valeur`*. Seul un schéma peut être précisé en utilisant cette syntaxe.

`NAMES`

`SET NAMES` *`valeur`* est un équivalent de `SET client_encoding TO` *`valeur`*.

`SEED`

Précise la valeur interne du générateur de nombres aléatoires (la fonction `random`). Les valeurs autorisées sont des nombres à virgule flottante entre -1 et 1.

Le générateur de nombres aléatoires peut aussi être initialisé en appelant la fonction `setseed` :

```
SELECT setseed(valeur);
```

`TIME ZONE`

`SET TIME ZONE '`*`valeur`*`'` est équivalent à `SET timezone TO '`*`valeur`*`'`. La syntaxe `SET TIME ZONE` permet d'utiliser une syntaxe spéciale pour indiquer le fuseau horaire. Quelques exemples de valeurs valides :

`'America/Los_Angeles'`

Le fuseau horaire de Berkeley, Californie.

`'Europe/Rome'`

Le fuseau horaire de l'Italie.

`-7`

Le fuseau horaire situé 7 heures à l'ouest de l'UTC (équivalent à PDT). Les valeurs positives sont à l'est de l'UTC.

`INTERVAL '-08:00' HOUR TO MINUTE`

Le fuseau horaire situé 8 heures à l'ouest de l'UTC (équivalent à PST).

`LOCAL`
`DEFAULT`

Utilise le fuseau horaire local (c'est-à-dire la valeur `timezone` par défaut du serveur).

Les réglages du fuseau horaire fournis en nombre ou intervalles sont convertis en interne en syntaxe de fuseau horaire POSIX. Par exemple, après avoir effectué `SET TIME ZONE -7`, `SHOW TIME ZONE` afficherait `<-07>+07`.

Les abréviations de fuseaux horaires ne sont pas acceptées par `SET` ; voir Section 8.5.3 pour de plus amples informations sur les fuseaux horaires.

Notes

La fonction `set_config` propose des fonctionnalités équivalentes. Voir Section 9.27.1. De plus, il est possible de mettre à jour (via UPDATE) la vue système `pg_settings` pour réaliser l'équivalent de `SET`.

Exemples

Mettre à jour le chemin de recherche :

```
SET search_path TO my_schema, public;
```

Utiliser le style de date traditionnel POSTGRES avec comme convention de saisie « les jours avant les mois » :

```
SET datestyle TO postgres, dmy;
```

Utiliser le fuseau horaire de Berkeley, Californie :

```
SET TIME ZONE 'America/Los_Angeles';
```

Utiliser le fuseau horaire de l'Italie :

```
SET TIME ZONE 'Europe/Rome';
```

Compatibilité

`SET TIME ZONE` étend la syntaxe définie dans le standard SQL. Le standard ne permet que des fuseaux horaires numériques alors que PostgreSQL est plus souple dans les syntaxes acceptées. Toutes les autres fonctionnalités de `SET` sont des extensions de PostgreSQL.

Voir aussi

RESET, SHOW

SET CONSTRAINTS

SET CONSTRAINTS — initialise le moment de vérification de contrainte de la transaction en cours

Synopsis

```
SET CONSTRAINTS { ALL | nom [, ...] } { DEFERRED | IMMEDIATE }
```

Description

`SET CONSTRAINTS` initialise le comportement de la vérification des contraintes dans la transaction en cours. Les contraintes `IMMEDIATE` sont vérifiées à la fin de chaque instruction. Les contraintes `DEFERRED` ne sont vérifiées qu'à la validation de la transaction. Chaque contrainte a son propre mode `IMMEDIATE` ou `DEFERRED`.

À la création, une contrainte se voit donner une des trois caractéristiques : `DEFERRABLE INITIALLY DEFERRED`, `DEFERRABLE INITIALLY IMMEDIATE` ou `NOT DEFERRABLE`. La troisième forme est toujours `IMMEDIATE` et n'est pas affectée par la commande `SET CONSTRAINTS`. Les deux premières classes commencent chaque transaction dans le mode indiqué mais leur comportement peut changer à l'intérieur d'une transaction par `SET CONSTRAINTS`.

`SET CONSTRAINTS` avec une liste de noms de contraintes modifie le mode de ces contraintes (qui doivent toutes être différables). Chaque nom de contrainte peut être qualifié d'un schéma. Le chemin de recherche des schémas est utilisé pour trouver le premier nom correspondant si aucun nom de schéma n'a été indiqué. `SET CONSTRAINTS ALL` modifie le mode de toutes les contraintes déférables.

Lorsque `SET CONSTRAINTS` modifie le mode d'une contrainte de `DEFERRED` à `IMMEDIATE`, le nouveau mode prend effet rétroactivement : toute modification de données qui aurait été vérifiée à la fin de la transaction est en fait vérifiée lors de l'exécution de la commande `SET CONSTRAINTS`. Si une contrainte est violée, la commande `SET CONSTRAINTS` échoue (et ne change pas le mode de contrainte). Du coup, `SET CONSTRAINTS` peut être utilisée pour forcer la vérification de contraintes à un point spécifique d'une transaction.

Actuellement, seules les contraintes `UNIQUE`, `PRIMARY KEY`, `REFERENCES` (clé étrangère) et `EXCLUDE` sont affectées par ce paramètre. Les contraintes `NOT NULL` et `CHECK` sont toujours vérifiées immédiatement quand une ligne est insérée ou modifiée (*pas* à la fin de l'instruction). Les contraintes uniques et d'exclusion qui n'ont pas été déclarées `DEFERRABLE` sont aussi vérifiées immédiatement.

Le déclenchement des triggers qui sont déclarés comme des « triggers de contraintes » est aussi contrôlé par ce paramètre -- ils se déclenchent au même moment que la contrainte associée devait être vérifiée.

Notes

Comme PostgreSQL ne nécessite pas les noms de contraintes d'être uniques à l'intérieur d'un schéma (mais seulement par tables), il est possible qu'il y ait plus d'une correspondance pour un nom de contrainte spécifié. Dans ce cas, `SET CONSTRAINTS` agira sur toutes les correspondances. Pour un nom sans qualification de schéma, une fois qu'une ou plusieurs correspondances ont été trouvées dans les schémas du chemin de recherche, les autres schémas du chemin ne sont pas testés.

Cette commande altère seulement le comportement des contraintes à l'intérieur de la transaction en cours. Exécuter cette commande en dehors d'un bloc de transaction cause l'émission d'un message d'avertissement mais n'a pas d'autres effets.

Compatibilité

Cette commande est compatible avec le comportement défini par le standard SQL en dehors du fait que, dans PostgreSQL, il ne s'applique pas aux contraintes `NOT NULL` et `CHECK`. De plus, PostgreSQL vérifie les contraintes uniques non déferrables immédiatement, pas à la fin de l'instruction comme le standard le suggère.

SET ROLE

SET ROLE — initialise l'identifiant utilisateur courant de la session en cours

Synopsis

```
SET [ SESSION | LOCAL ] ROLE nom_rôle
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

Description

Cette commande positionne l'identifiant utilisateur courant suivant la session SQL en cours à *nom_rôle*. Le nom du rôle peut être un identifiant ou une chaîne littérale. Après SET ROLE, la vérification des droits sur les commandes SQL est identique à ce qu'elle serait si le rôle nommé s'était lui-même connecté.

Il est obligatoire que l'utilisateur de la session courante soit membre du rôle *nom_rôle* (si l'utilisateur de la session est superutilisateur, tous les rôles sont utilisables).

Les modificateurs SESSION et LOCAL agissent de la même façon que pour la commande SET.

SET ROLE NONE initialise l'identifiant utilisateur courant avec l'identifiant de la session en cours, tel qu'il est renvoyé par session_user. RESET ROLE initialise l'identifiant de l'utilisateur courant avec le paramètre à la connexion indiqué par les options en ligne de commande, ALTER ROLE ou ALTER DATABASE, si de tels paramètres existent. Sinon, RESET ROLE initialise l'identifiant utilisateur courant avec l'identifiant utilisateur courant. Ces formats peuvent être exécutés par tout utilisateur.

Notes

L'utilisation de cette commande permet d'étendre ou de restreindre les privilèges d'un utilisateur. Si le rôle de l'utilisateur de la session comprend l'attribut INHERIT, alors il acquiert automatiquement les droits de chaque rôle qu'il peut prendre par la commande SET ROLE ; dans ce cas, SET ROLE supprime tous les droits affectés directement à l'utilisateur de la session et les autres droits des rôles dont il est membre, ne lui laissant que les droits disponibles sur le rôle nommé. A l'opposé, si le rôle session de l'utilisateur dispose de l'attribut NOINHERIT, SET ROLE supprime les droits affectés directement à l'utilisateur session et les remplace par les privilèges du rôle nommé.

En particulier, quand un utilisateur choisit un rôle autre que superutilisateur via SET ROLE, il perd les droits superutilisateur.

SET ROLE a des effets comparables à SET SESSION AUTHORIZATION mais la vérification des droits diffère. De plus, SET SESSION AUTHORIZATION détermine les rôles autorisés dans les commandes SET ROLE ultérieures alors que SET ROLE ne modifie pas les rôles accessibles par un futur SET ROLE.

SET ROLE ne traite pas les variables de session indiqué par les paramètres du rôle (et configurés avec ALTER ROLE ; cela ne survient qu'à la connexion.

SET ROLE ne peut pas être utilisé dans une fonction SECURITY DEFINER.

Exemples

```
SELECT SESSION_USER, CURRENT_USER;
```

```
 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | paul
```

Compatibilité

PostgreSQL autorise la syntaxe identifiant ("*nom_role*") alors que le SQL standard impose une chaîne littérale pour le nom du rôle. SQL n'autorise pas cette commande lors d'une transaction ; PostgreSQL n'est pas aussi restrictif, rien ne justifie cette interdiction. Les modificateurs `SESSION` et `LOCAL` sont des extensions PostgreSQL tout comme la syntaxe `RESET`.

Voir aussi

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION — Initialise l'identifiant de session de l'utilisateur et l'identifiant de l'utilisateur actuel de la session en cours

Synopsis

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION nom_utilisateur
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

Description

Cette commande positionne l'identifiant de session de l'utilisateur et celui de l'utilisateur courant pour la session SQL en cours à *nom_utilisateur*. Le nom de l'utilisateur peut être un identifiant ou une chaîne littérale. En utilisant cette commande, il est possible, par exemple, de devenir temporairement un utilisateur non privilégié et de redevenir plus tard superutilisateur.

L'identifiant de session de l'utilisateur est initialement positionné au nom de l'utilisateur (éventuellement authentifié) fourni par le client. L'identifiant de l'utilisateur courant est habituellement identique à l'identifiant de session de l'utilisateur mais il peut être temporairement modifié par le contexte de fonctions SECURITY DEFINER ou de mécanismes similaires ; il peut aussi être changé par SET ROLE. L'identifiant de l'utilisateur courant est essentiel à la vérification des permissions.

L'identifiant de session de l'utilisateur ne peut être changé que si l'utilisateur de session initial (l'*utilisateur authentifié*) dispose des privilèges superutilisateur. Dans le cas contraire, la commande n'est acceptée que si elle fournit le nom de l'utilisateur authentifié.

Les modificateurs SESSION et LOCAL agissent de la même façon que la commande standard SET.

Les formes DEFAULT et RESET réinitialisent les identifiants courant et de session de l'utilisateur à ceux de l'utilisateur originellement authentifié. Tout utilisateur peut les exécuter.

Notes

SET SESSION AUTHORIZATION ne peut pas être utilisé dans une fonction SECURITY DEFINER.

Exemples

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

Compatibilité

Le standard SQL autorise l'apparition de quelques autres expressions à la place de *`nom_utilisateur`*. Dans la pratique, ces expressions ne sont pas importantes. PostgreSQL autorise la syntaxe de l'identifiant (`"`*`nom_utilisateur`*`"`) alors que SQL ne le permet pas. SQL n'autorise pas l'exécution de cette commande au cours d'une transaction ; PostgreSQL n'impose pas cette restriction parce qu'il n'y a pas lieu de le faire. Les modificateurs `SESSION` et `LOCAL` sont des extensions PostgreSQL tout comme la syntaxe `RESET`.

Le standard laisse la définition des droits nécessaires à l'exécution de cette commande à l'implantation.

Voir aussi

SET ROLE

SET TRANSACTION

SET TRANSACTION — initialise les caractéristiques de la transaction actuelle

Synopsis

```
SET TRANSACTION mode_transaction [, ...]
SET TRANSACTION SNAPSHOT id_snapshot
SET SESSION CHARACTERISTICS AS TRANSACTION mode_transaction [, ...]

où mode_transaction fait
partie de :

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ
 COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Description

La commande `SET TRANSACTION` initialise les caractéristiques de la transaction courante. Elle est sans effet sur les transactions suivantes. `SET SESSION CHARACTERISTICS` positionne les caractéristiques par défaut pour toutes les transactions à venir d'une session. Ces valeurs peuvent ensuite être surchargées par `SET TRANSACTION` pour une transaction particulière.

Les caractéristiques de transaction disponibles sont le niveau d'isolation, le mode d'accès de la transaction (lecture/écriture ou lecture seule) et le mode différable. De plus, un snapshot peut être sélectionné, bien que pour la transaction en cours, et non pas pour la session.

Le niveau d'isolation détermine les données que la transaction peut voir quand d'autres transactions fonctionnent concurrentiellement :

`READ COMMITTED`

Une instruction ne peut voir que les lignes validées avant qu'elle ne commence. C'est la valeur par défaut.

`REPEATABLE READ`

Toute instruction de la transaction en cours ne peut voir que les lignes validées avant que la première requête ou instruction de modification de données soit exécutée dans cette transaction.

`SERIALIZABLE`

Toutes les requêtes de la transaction en cours peuvent seulement voir les lignes validées avant l'exécution de la première requête ou instruction de modification de données de cette transaction. Si un ensemble de lectures et écritures parmi les transactions sérialisables concurrentes créait une situation impossible à obtenir avec une exécution en série (une à la fois) de ces transactions, l'une d'entre elles sera annulée avec une erreur `serialization_failure`.

Le standard SQL définit un niveau supplémentaire, `READ UNCOMMITTED`. Dans PostgreSQL, `READ UNCOMMITTED` est traité comme `READ COMMITTED`.

Le niveau d'isolation de la transaction ne peut plus être modifié après l'exécution de la première requête ou instruction de modification de données (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `MERGE`, `FETCH` ou `COPY`) d'une transaction. Voir Chapitre 13 pour plus d'informations sur l'isolation et le contrôle de concurrence.

La méthode d'accès de la transaction détermine si elle est en lecture/écriture ou en lecture seule. Lecture/écriture est la valeur par défaut. Quand une transaction est en lecture seule, les commandes SQL suivantes sont interdites : INSERT, UPDATE, DELETE, MERGE et COPY FROM si la table modifiée n'est pas temporaire ; toutes les commandes CREATE, ALTER et DROP ; COMMENT, GRANT, REVOKE, TRUNCATE ; EXPLAIN ANALYZE et EXECUTE si la commande exécutée figure parmi celles listées plus haut. C'est une notion de haut niveau de lecture seule qui n'interdit pas toutes les écritures sur disque.

La propriété DEFERRABLE d'une transaction n'a pas d'effet tant que la transaction est aussi SERIALIZABLE et READ ONLY. Quand ces trois propriétés sont sélectionnées pour une transaction, la transaction pourrait bloquer lors de la première acquisition de son image de la base, après quoi il est possible de fonctionner sans la surcharge normale d'une transaction SERIALIZABLE et sans risque de contribuer ou d'être annulé par un échec de sérialisation. Ce mode convient bien à l'exécution de longs rapports ou à la création de sauvegardes.

La commande SET TRANSACTION SNAPSHOT permet à une nouvelle transaction de s'exécuter avec le même *snapshot* que celle d'une transaction existante. La transaction pré-existante doit avoir exportée son snapshot avec la fonction pg_export_snapshot (voir Section 9.27.5). Cette fonction renvoie un identifiant de snapshot, qui doit être fourni à SET TRANSACTION SNAPSHOT pour indiquer le snapshot à importer. L'identifiant doit être écrit sous la forme d'une chaîne litérale dans cette commande, par exemple '00000003-0000001B-1'. SET TRANSACTION SNAPSHOT peut seulement être exécuté au début d'une transaction, avant la première requête ou la première instruction de modification de données (SELECT, INSERT, DELETE, UPDATE, MERGE, FETCH ou COPY) de la transaction. De plus, la transaction doit déjà être configurée au niveau d'isolation SERIALIZABLE ou REPEATABLE READ (sinon le snapshot sera immédiatement annulé car le mode READ COMMITTED prend un nouveau snapshot pour chaque commande). Si la transaction d'import utilise le niveau d'isolation SERIALIZABLE, la transaction qui a exporté le snapshot doit aussi utiliser ce niveau d'isolation. De plus, une transaction sérialisable en lecture/écriture ne peut pas importer un snapshot à partir d'une transaction en lecture seule.

Notes

Si SET TRANSACTION est exécuté sans START TRANSACTION ou BEGIN préalable, il n'a aucun effet et un avertissement est renvoyé.

Il est possible de se dispenser de SET TRANSACTION en spécifiant le *mode_transaction* désiré dans BEGIN ou START TRANSACTION. Mais cette option n'est pas disponible pour SET TRANSACTION SNAPSHOT.

Les modes de transaction par défaut d'une session peuvent aussi être configurés ou examinés en initialisant les paramètres de configuration default_transaction_isolation, default_transaction_read_only et default_transaction_deferrable. (En fait, SET SESSION CHARACTERISTICS est un équivalent verbeux de la configuration de ces variables avec SET.) Les valeurs par défaut peuvent ainsi être initialisées dans le fichier de configuration, via ALTER DATABASE, etc. Chapitre 20 fournit de plus amples informations.

Les modes de la transaction en cours peuvent similairement être configurés ou examinés via les paramètres de configuration transaction_isolation, transaction_read_only et transaction_deferrable. Configurer un de ces paramètres agit de la même façon que l'option SET TRANSACTION correspondante, avec les mêmes restrictions quand cela peut se faire. Néanmoins, ces paramètres ne peuvent pas être configurés dans le fichier de configuration ou dans tout autre source que du SQL direct.

Exemples

Pour commencer une nouvelle transaction avec le même snapshot qu'une autre transaction en cours d'exécution, commencez par exporter le snapshot de la transaction existante. Cela renvoie un identifiant de snapshot, par exemple :

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
 pg_export_snapshot
--------------------
 00000003-0000001B-1
(1 row)
```

Ensuite, donnez l'identifiant de snapshot dans une commande `SET TRANSACTION SNAPSHOT` au début de la nouvelle transaction :

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

Compatibilité

Ces commandes sont définies dans le standard SQL, sauf en ce qui concerne le mode de transaction `DEFERRABLE` et la forme `SET TRANSACTION SNAPSHOT`, qui sont des extensions de PostgreSQL.

`SERIALIZABLE` est le niveau d'isolation par défaut dans le standard. Dans PostgreSQL, le niveau par défaut est d'habitude `READ COMMITTED` mais il est possible de le modifier comme indiqué ci-dessus.

Dans le standard SQL, il existe une autre caractéristique de transaction pouvant être configurée avec ces commandes : la taille de l'aire de diagnostic. Ce concept est spécifique au SQL embarqué et, du coup, n'est pas implémenté dans PostgreSQL.

Le standard SQL requiert des virgules entre des *`transaction_modes`* successifs mais, pour des raisons historiques, PostgreSQL autorise de ne pas mettre de virgules.

SHOW

SHOW — affiche la valeur d'un paramètre d'exécution

Synopsis

```
SHOW nom
SHOW ALL
```

Description

`SHOW` affiche la configuration courante des paramètres d'exécution. Ces variables peuvent être initialisées à l'aide de l'instruction `SET`, par le fichier de configuration `postgresql.conf`, par la variable d'environnement `PGOPTIONS` (lors de l'utilisation de libpq ou d'une application fondée sur libpq), ou à l'aide d'options en ligne de commande lors du démarrage de `postgres`. Voir Chapitre 20 pour plus de détails.

Paramètres

nom

Le nom d'un paramètre d'exécution. Les paramètres disponibles sont documentés dans Chapitre 20 et sur la page de référence SET. De plus, il existe quelques paramètres qui peuvent être affichés mais ne sont pas initialisables :

`SERVER_VERSION`

Affiche le numéro de version du serveur.

`SERVER_ENCODING`

Affiche l'encodage des caractères côté serveur. À ce jour, ce paramètre peut être affiché mais pas initialisé parce que l'encodage est déterminé au moment de la création de la base de données.

`LC_COLLATE`

Affiche la locale de la base de données pour le tri de texte. À ce jour, ce paramètre est affichable mais pas initialisé parce que la configuration est déterminée lors de la création de la base de données.

`LC_CTYPE`

Affiche la locale de la base de données pour la classification des caractères. À ce jour, ce paramètre peut être affiché mais pas initialisé parce que la configuration est déterminée lors de la création de la base de données.

`IS_SUPERUSER`

Vrai si le rôle courant a des droits de superutilisateur.

`ALL`

Affiche les valeurs de tous les paramètres de configuration avec leur description.

Notes

La fonction `current_setting` affiche les mêmes informations. Voir Section 9.27.1. De plus, la vue système `pg_settings` propose la même information.

Exemples

Affiche la configuration courante du paramètre `datestyle` :

```
SHOW datestyle;
 datestyle
-----------
 ISO, MDY
(1 row)
```

Affiche la configuration courante du paramètre `geqo` :

```
SHOW geqo;
 geqo
------
 on
(1 row)
```

Affiche tous les paramètres :

```
            name         | setting |                description
-------------------------+---------
+-----------------------------------------------
 allow_system_table_mods | off     | Allows modifications of the
 structure of ...
    .
    .
    .
 xmloption               | content | Sets whether XML data in
 implicit parsing ...
 zero_damaged_pages      | off     | Continues processing past
 damaged page headers.
(196 rows)
```

Compatibilité

La commande `SHOW` est une extension PostgreSQL.

Voir aussi

SET, RESET

START TRANSACTION

START TRANSACTION — débute un bloc de transaction

Synopsis

```
START TRANSACTION [ mode_transaction [, ...] ]

où mode_transaction fait
partie de :

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ
 COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Description

Cette commande débute un nouveau bloc de transaction. Si le niveau d'isolation, le mode lecture/écriture ou le mode différable est spécifié, la nouvelle transaction adopte ces caractéristiques, comme si `SET TRANSACTION` avait été exécuté. Cette commande est identique à la commande `BEGIN`.

Paramètres

Pour obtenir la signification des paramètres de cette instruction, on pourra se référer à SET TRANSACTION.

Compatibilité

Le standard SQL n'impose pas de lancer `START TRANSACTION` pour commencer un bloc de transaction : toute commande SQL débute implicitement un bloc. On peut considérer que PostgreSQL exécute implicitement un `COMMIT` après chaque commande non précédée de `START TRANSACTION` (ou `BEGIN`). Ce comportement est d'ailleurs souvent appelé « autocommit ». D'autres systèmes de bases de données relationnelles offrent une fonctionnalité de validation automatique.

L'option `DEFERRABLE` de *`transaction_mode`* est une extension de PostgreSQL.

Le standard SQL impose des virgules entre les *`modes_transaction`* successifs mais, pour des raisons historiques, PostgreSQL autorise l'omission des virgules.

Voir aussi la section de compatibilité de SET TRANSACTION.

Voir aussi

BEGIN, COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION

TRUNCATE

TRUNCATE — vide une table ou un ensemble de tables

Synopsis

```
TRUNCATE [ TABLE ] [ ONLY ] nom [ * ] [, ... ]
    [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

Description

La commande TRUNCATE supprime rapidement toutes les lignes d'un ensemble de tables. Elle a le même effet qu'un DELETE non qualifié sur chaque table, mais comme elle ne parcourt par la table, elle est plus rapide. De plus, elle récupère immédiatement l'espace disque, évitant ainsi une opération VACUUM. Cette commande est particulièrement utile pour les tables volumineuses.

Paramètres

nom

Le nom d'une table à vider (pouvant être qualifié par le schéma). Si la clause ONLY est précisée avant le nom de la table, seule cette table est tronquée. Dans le cas contraire, la table et toutes ses tables filles (si elle en a) sont tronquées. En option, * peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles sont inclues.

RESTART IDENTITY

Redémarre les séquences intégrées aux colonnes des tables tronquées.

CONTINUE IDENTITY

Ne change pas la valeur des séquences. C'est la valeur par défaut.

CASCADE

Vide toutes les tables qui ont des références de clés étrangères sur une des tables nommées et sur toute table ajoutée au groupe à cause du CASCADE.

RESTRICT

Refuse le vidage si une des tables a des références de clés étrangères sur une table qui ne sont pas listées dans la commande. Cette option est active par défaut.

Notes

Vous devez avoir le droit TRUNCATE sur la table que vous voulez tronquer.

TRUNCATE nécessite un verrou d'accès exclusif (ACCESS EXCLUSIVE) sur chaque table qu'il traite, ce qui bloque toutes les autres opérations en parallèle sur cette table. Quand RESTART IDENTITY est spécifié, toutes les séquences qui doivent être réinitialisées ont un verrou exclusif. Si un accès concurrent est nécessaire, alors la commande DELETE doit être utilisée.

TRUNCATE ne peut pas être utilisé sur une table référencée par d'autres tables au travers de clés étrangères, sauf si ces tables sont aussi comprises dans la commande. Dans le cas contraire, la vérification nécessiterait des parcours complets de tables, ce qui n'est pas le but de la commande

TRUNCATE. L'option CASCADE est utilisable pour inclure automatiquement toutes les tables dépendantes -- faites attention lorsque vous utilisez cette option parce que vous pourriez perdre des données que vous auriez souhaitez conserver ! Notez en particulier que, quand la table à tronquer est une partition, les partitions filles ne sont pas touchées mais la cascade survient pour toutes les tables référencées et toutes leurs partitions sans distinction.

TRUNCATE ne déclenchera aucun trigger ON DELETE qui pourrait exister sur les tables. Par contre, il déclenchera les triggers ON TRUNCATE. Si des triggers ON TRUNCATE sont définis sur certaines des tables, alors tous les triggers BEFORE TRUNCATE sont déclenchés avant que le troncage n'intervienne, et tous les triggers AFTER TRUNCATE sont déclenchés après la réalisation du dernier troncage et toutes les séquences sont réinitialisées. Les triggers se déclencheront dans l'ordre de traitement des tables (tout d'abord celles listées dans la commande, puis celles ajoutées à cause des cascades).

TRUNCATE n'est pas sûre au niveau MVCC. Après la troncature, la table apparaîtra vide aux transactions concurrentes si elles utilisent une image prise avant la troncature. Voir Section 13.6 pour plus de détails.

TRUNCATE est compatible avec le système des transactions. Les données seront toujours disponibles si la transaction est annulée.

Quand RESTART IDENTITY est spécifié, les opérations ALTER SEQUENCE RESTART impliquées sont aussi réalisées de façon transactionnelles. Autrement dit, elles seront annulées si la transaction n'est pas validée. Faites attention au fait que si des opérations supplémentaires sur les séquences impliquées est faite avant l'annulation de la transaction, les effets de ces opérations sur les séquences seront aussi annulés mais pas les effets sur currval() ; autrement dit, après la transaction, currval() continuera à refléter la dernière valeur de la séquence obtenue au sein de la transaction échouée, même si la séquence elle-même pourrait ne plus être en adéquation avec cela. C'est similaire au comportement habituel de currval() après une transaction échouée.

TRUNCATE peut être utilisé pour les tables distantes si c'est supporté par le « Foreign Data Wrapper ». Voir, par exemple, postgres_fdw.

Exemples

Vider les tables grossetable et grandetable :

```
TRUNCATE grossetable, grandetable;
```

La même chose, en réinitialisant les générateurs des séquences associées :

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

Vide la table uneautretable, et cascade cela à toutes les tables qui référencent uneautretable via des contraintes de clés étrangères :

```
TRUNCATE uneautretable CASCADE;
```

Compatibilité

Le standard SQL:2008 inclut une commande TRUNCATE avec la syntaxe TRUNCATE TABLE *nom_table*. Les clauses CONTINUE IDENTITY/RESTART IDENTITY font aussi partie du standard mais ont une signification légèrement différente, quoique en rapport. Certains des comportements de concurrence de cette commande sont laissés au choix de l'implémentation par le

standard, donc les notes ci-dessus doivent être comprises et comparées avec les autres implémentations si nécessaire.

Voir également

DELETE

UNLISTEN

UNLISTEN — arrête l'écoute d'une notification

Synopsis

```
UNLISTEN { canal | * }
```

Description

`UNLISTEN` est utilisé pour supprimer un abonnement aux événements `NOTIFY`. `UNLISTEN` annule tout abonnement pour la session PostgreSQL en cours sur le canal de notification nommé *`canal`*. Le caractère générique `*` annule tous les abonnements de la session en cours.

NOTIFY contient une discussion plus complète de l'utilisation de `LISTEN` et de `NOTIFY`.

Paramètres

`canal`

Le nom d'un canal de notification (un identificateur quelconque).

`*`

Tous les abonnements de cette session sont annulés.

Notes

Il est possible de se désabonner de quelque chose pour lequel il n'y a pas d'abonnement ; aucun message d'avertissement ou d'erreur n'est alors retourné.

À la fin de chaque session, `UNLISTEN *` est exécuté automatiquement.

Une transaction qui a exécuté `UNLISTEN` ne peut pas être préparée pour une validation en deux phases.

Exemples

Pour s'abonner :

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process
 with PID 8448.
```

Une fois que `UNLISTEN` a été exécuté, les messages `NOTIFY` suivants sont ignorés :

```
UNLISTEN virtual;
NOTIFY virtual;
-- aucun événement NOTIFY n'est reçu
```

Compatibilité

Il n'y a pas de commande `UNLISTEN` dans le standard SQL.

Voir aussi

LISTEN, NOTIFY

UPDATE

UPDATE — mettre à jour les lignes d'une table

Synopsis

```
[ WITH [ RECURSIVE ] requête_with [, ...] ]
UPDATE [ ONLY ] nom_table [ * ] [ [ AS ] alias ]
    SET { nom_colonne = { expression | DEFAULT } |
          ( nom_colonne [, ...] ) = [ ROW ] ( { expression |
 DEFAULT } [, ...] ) |
          ( nom_colonne [, ...] ) = ( sous-SELECT )
        } [, ...]
    [ FROM element_from [, ...] ]
    [ WHERE condition | WHERE CURRENT OF nom_curseur ]
    [ RETURNING { * | expression_sortie [ [ AS ] nom_sortie ] }
 [, ...] ]
```

Description

`UPDATE` modifie les valeurs des colonnes spécifiées pour toutes les lignes qui satisfont la condition. Seules les colonnes à modifier doivent être mentionnées dans la clause `SET` ; les autres colonnes conservent leur valeur.

Il existe deux façons de modifier le contenu d'une table à partir d'informations contenues dans d'autres tables de la base de données : à l'aide de sous-requêtes ou en spécifiant des tables supplémentaires dans la clause `FROM`. Le contexte permet de décider de la technique la plus appropriée.

La clause `RETURNING` optionnelle fait que `UPDATE` calcule et renvoie le(s) valeur(s) basée(s) sur chaque ligne en cours de mise à jour. Toute expression utilisant les colonnes de la table et/ou les colonnes d'autres tables mentionnées dans `FROM` peut être calculée. La syntaxe de la liste `RETURNING` est identique à celle de la commande `SELECT`.

L'utilisateur doit posséder le droit `UPDATE` sur la table, ou au moins sur les colonnes listées pour la mise à jour. Vous devez aussi avoir le droit `SELECT` sur toutes les colonnes dont les valeurs sont lues dans les *`expression`*s ou *`condition`*.

Paramètres

`requête_with`

La clause `WITH` vous permet de spécifier une ou plusieurs sous-requêtes qui peuvent être référencées par nom dans la requête`UPDATE`. Voir Section 7.8 et SELECT pour les détails.

`nom_table`

Le nom de la table à mettre à jour (éventuellement qualifié du nom du schéma). Si `ONLY` est indiqué avant le nom de la table, les lignes modifiées ne concernent que la table nommée. Si `ONLY` n'est pas indiquée, les lignes modifiées font partie de la table nommée et de ses tables filles. En option, `*` peut être ajouté après le nom de la table pour indiquer explicitement que les tables filles doivent être inclues.

`alias`

Un nom de substitution pour la table cible. Quand un alias est fourni, il cache complètement le nom réel de la table. Par exemple, avec `UPDATE foo AS f`, le reste de l'instruction `UPDATE` doit référencer la table avec `f` et non plus `foo`.

nom_colonne

Le nom d'une colonne dans *nom_table*. Le nom de la colonne peut être qualifié avec un nom de sous-champ ou un indice de tableau, si nécessaire. Ne pas inclure le nom de la table dans la spécification d'une colonne cible -- par exemple, `UPDATE nom_table SET nom_table.col = 1` est invalide.

expression

Une expression à affecter à la colonne. L'expression peut utiliser les anciennes valeurs de cette colonne et d'autres colonnes de la table.

`DEFAULT`

Réinitialise la colonne à sa valeur par défaut (qui vaut NULL si aucune expression par défaut ne lui a été affectée). Une colonne d'identité sera configurée à une nouvelle valeur générée par la séquence associée. Pour une colonne générée, spécifier ceci est permis mais ne fait que surligner le comportement normal du calcul de la colonne à partir de l'expression.

sous-SELECT

Une sous-requête `SELECT` qui produit autant de colonnes en sortie que de colonnes comprises dans la liste entre parenthèses la précédant. La sous-requête doit ne renvoyer qu'une seule ligne lors de son exécution. Si elle renvoie une seule ligne, les valeurs des colonnes du résultat sont affectées aux colonnes cibles. Si elle ne renvoie aucune ligne, des valeurs NULL sont affectées aux colonnes cibles. La sous-requête peut faire référence aux anciennes valeurs de la ligne en cours de mise à jour.

eelment_from

Une expression de table, qui permet aux colonnes des autres tables d'apparaître dans la condition `WHERE` et dans les expressions de mise à jour. Cela utilise la même syntaxe que le `FROM` d'une instruction `SELECT` ; par exemple, un alias peut être indiqué pour le nom de la table. Ne répétez pas la table cible dans un *element_from*, sauf si vous souhaitez faire un jointure sur elle-même (auquel cas elle doit apparaitre avec un alias dans *element_from*).

condition

Une expression qui renvoie une valeur de type `boolean`. Seules les lignes pour lesquelles cette expression renvoie `true` sont mises à jour.

nom_curseur

Le nom du curseur à utiliser dans une condition `WHERE CURRENT OF`. La ligne à mettre à jour est la dernière récupérée à partir de ce curseur. Le curseur doit être une requête sans regroupement sur la table cible de l'`UPDATE`. Notez que `WHERE CURRENT OF` ne peut pas être spécifié avec une condition booléenne. Voir DECLARE pour plus d'informations sur l'utilisation des curseurs avec `WHERE CURRENT OF`.

expression_sortie

Une expression à calculer et renvoyée par la commande `UPDATE` après chaque mise à jour de ligne. L'expression peut utiliser tout nom de colonne de la table nommée *nom_table* ou des tables listées dans le `FROM`. Indiquez `*` pour que toutes les colonnes soient renvoyées.

nom_sortie

Un nom à utiliser pour une colonne renvoyée.

Sorties

En cas de succès, une commande `UPDATE` renvoie un message de la forme

```
UPDATE total
```

total est le nombre de lignes mises à jour, en incluant les lignes qui correspondent au filtre mais dont la valeur des colonnes ne change pas. Notez que le nombre peut être inférieur au nombre de lignes filtrées par la *condition* quand certaines mises à jour sont supprimées par un trigger BEFORE UPDATE. S'il vaut 0, aucune ligne n'a été mise à jour par cette requête (ce qui n'est pas considéré comme une erreur).

Notes

Lorsqu'une clause FROM est précisée, la table cible est jointe aux tables mentionnées dans *element_from*, et chaque ligne en sortie de la jointure représente une opération de mise à jour pour la table cible. Lors de l'utilisation de FROM, il faut s'assurer que la jointure produit au plus une ligne en sortie par ligne à modifier. En d'autres termes, une ligne cible ne doit pas être jointe à plus d'une ligne des autres tables. Le cas échéant, seule une ligne de jointure est utilisée pour mettre à jour la ligne cible, mais il n'est pas possible de prédire laquelle.

À cause de ce manque de déterminisme, il est plus sûr de ne référencer les autres tables qu'à l'intérieur de sous-requêtes. Même si c'est plus difficile à lire et souvent plus lent que l'utilisation d'une jointure.

Dans le cas d'une table partitionnée, mettre à jour une ligne pourrait faire qu'elle ne satisfait plus la contrainte de partitionnement de la partition contenante. Dans ce cas, s'il existe une autre partition dans l'arbre de partition pour laquelle cette ligne satisfait sa contrainte de partitionnement, alors la ligne est déplacée dans cette partition. Si une telle partition n'existe pas, une erreur sera levée. Dans les faits, un mouvement de ligne est en fait une opération DELETE suivi d'un INSERT.

Il est possible qu'une commande UPDATE ou DELETE en concurrence sur la ligne en déplacement obtienne une erreur d'échec de sérialisation. Supposons que la session 1 réalise un UPDATE sur une clé de partitionnement alors que la session 2, pour laquelle cette ligne est visible, réalise un UPDATE ou un DELETE sur cette ligne. Dans ce cas, l'opération UPDATE ou DELETE de la session 2 détectera le déplacement de ligne et renverra une erreur d'échec de sérialisation (qui renvoie toujours le code SQLSTATE '40001'). Les applications pourraient souhaiter tenter de nouveau la transaction si cela arrive. Dans le cas inhabituel où la table n'est pas partitionné ou qu'il n'y a pas de mouvement de ligne, la session 2 aura identifié la ligne nouvellement mise à jour et continué l'opération UPDATE/DELETE sur cette nouvelle version de ligne.

Notez que, bien que les lignes puissent être déplacées des partitions locales vers une partition distante (fournie par le foreign data wrapper qui supporte le déplacement de lignes), elles ne peuvent pas être déplacées d'une partition distante vers une autre partition.

Une tentative de déplacer une ligne d'une partition à une autre échouera si une clé étrangère référence directement un ancêtre de la partition source qui n'est pas le même que l'ancêtre mentionné dans la requête UPDATE.

Si la commande UPDATE contient une clause RETURNING, le résultat sera similaire à celui d'une instruction SELECT contenant les colonnes et les valeurs définies dans la liste RETURNING, à partir de la liste des lignes mises à jour par la commande, comme la possibilité d'utiliser la clause WITH avec la commande UPDATE.

Exemples

Changer le mot Drame en Dramatique dans la colonne genre de la table films :

```
UPDATE films SET genre = 'Dramatique' WHERE genre = 'Drame';
```

Ajuster les entrées de température et réinitialiser la précipitation à sa valeur par défaut dans une ligne de la table temps :

```
UPDATE temps SET temp_basse = temp_basse+1, temp_haute = temp_basse
+15, prcp = DEFAULT
  WHERE ville = 'San Francisco' AND date = '2005-07-03';
```

Réaliser la même opération et renvoyer les lignes mises à jour :

```
UPDATE temps SET temp_basse = temp_basse+1, temp_haute = temp_basse
+15, prcp = DEFAULT
  WHERE ville = 'San Francisco' AND date = '2003-07-03'
  RETURNING temp_basse, temp_haute, prcp;
```

Utiliser une autre syntaxe pour faire la même mise à jour :

```
UPDATE temps SET (temp_basse, temp_haute, prcp) = (temp_basse+1,
 temp_basse+15, DEFAULT)
  WHERE ville = 'San Francisco' AND date = '2003-07-03';
```

Incrémenter le total des ventes de la personne qui gère le compte d'Acme Corporation, à l'aide de la clause `FROM` :

```
UPDATE employes SET total_ventes = total_ventes + 1 FROM comptes
  WHERE compte.nom = 'Acme Corporation'
  AND employes.id = compte.vendeur;
```

Réaliser la même opération en utilisant une sous-requête dans la clause `WHERE` :

```
UPDATE employes SET total_ventes = total_ventes + 1 WHERE id =
  (SELECT vendeur FROM comptes WHERE nom = 'Acme Corporation');
```

Mettre à jour les noms du contat dans la table comptes pour correspondre au vendeur actuellement affecté :

```
UPDATE comptes SET (prenom_compte, nom_compte) =
    (SELECT prenom, nom FROM vendeurs
     WHERE vendeurs.id = comptes.id_vendeur);
```

Un résultat similaire peut être obtenu avec une jointure :

```
UPDATE comptes SET prenom_contact = prenom,
                   nom_contact = nom
  FROM vendeurs WHERE vendeurs.id = comptes.id_vendeur;
```

Néanmoins, la deuxième requête pourrait donner des résultats inattendus si `vendeurs.id` n'est pas une clé unique alors que la première requête garantie la levée d'une erreur si plusieurs `id` correspondent. De plus, s'il n'y a pas de correspondance pour un certain `comptes.id_vendeur`, la première requête configurera les champs correspondants à NULL alors que la deuxième requête ne mettra pas du tout la ligne à jour.

Mettre à jour les statistiques dans une table de résumé pour correspondre aux données actuelles :

```
UPDATE resumes s SET (somme_x, somme_y, moyenne_x, moyenne_y) =
```

```
    (SELECT sum(x), sum(y), avg(x), avg(y) FROM donnees d
     WHERE d.id_groupe = s.id_groupe);
```

Tenter d'insérer un nouvel élément dans le stock avec sa quantité. Si l'élément existe déjà, mettre à jour le total du stock de l'élément. Les points de sauvegarde sont utilisés pour ne pas avoir à annuler l'intégralité de la transaction en cas d'erreur :

```
BEGIN;
-- autres opérations
SAVEPOINT sp1;
INSERT INTO vins VALUES('Chateau Lafite 2003', '24');
-- A supposer que l'instruction ci-dessus échoue du fait d'une
 violation de clé
-- unique, les commandes suivantes sont exécutées :
ROLLBACK TO sp1;
UPDATE vins SET stock = stock + 24 WHERE nomvin = 'Chateau Lafite
 2003';
-- continuer avec les autres opérations, et finir
COMMIT;
```

Modifier la colonne `genre` de la table `films` dans la ligne où le curseur `c_films` est actuellement positionné :

```
UPDATE films SET genre = 'Dramatic' WHERE CURRENT OF c_films;
```

Compatibilité

Cette commande est conforme au standard SQL, à l'exception des clauses `FROM` et `RETURNING` qui sont des extensions PostgreSQL.

D'autres systèmes de bases de données offrent l'option `FROM` dans laquelle la table cible est supposée être à nouveau indiquée dans le `FROM`. PostgreSQL n'interprète pas la clause `FROM` ainsi. Il est important d'en tenir compte lors du portage d'applications qui utilisent cette extension.

D'après le standard, la valeur source pour une sous-liste de noms de colonnes peut être toute expression de ligne renvoyant le bon nombre de colonnes. PostgreSQL autorise seulement la valeur source à être un constructeur de ligne ou un sous-`SELECT`. Une valeur mise à jour pour une colonne individuelle peut être spécifiée en tant que `DEFAULT` dans le cas d'une liste d'expressions, mais pas à l'intérieur d'un sous-`SELECT`.

VACUUM

VACUUM — récupère l'espace inutilisé et, optionnellement, analyse une base

Synopsis

```
VACUUM [ ( option [, ...] ) ] [ table_et_colonnes [, ...] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ ANALYZE ]
 [ table_et_colonnes [, ...] ]

où option fait partie de :

    FULL [ boolean ]
    FREEZE [ boolean ]
    VERBOSE [ boolean ]
    ANALYZE [ boolean ]
    DISABLE_PAGE_SKIPPING [ boolean ]
    SKIP_LOCKED [ boolean ]
    INDEX_CLEANUP [ AUTO | ON | OFF ]
    PROCESS_TOAST [ boolean ]
    TRUNCATE [ boolean ]
    PARALLEL integer

et table_et_colonnes est :

    nom_table [ ( nom_colonne [, ...] ) ]
```

Description

VACUUM récupère l'espace de stockage occupé par des lignes mortes. Lors des opérations normales de PostgreSQL, les lignes supprimées ou rendues obsolètes par une mise à jour ne sont pas physiquement supprimées de leur table. Elles restent présentes jusqu'à ce qu'un VACUUM soit lancé. C'est pourquoi, il est nécessaire de faire un VACUUM régulièrement, spécialement sur les tables fréquemment mises à jour.

Sans une liste *table_et_colonnes*, VACUUM traite chaque table et vue matérialisée que l'utilisateur actuel a le droit de traiter ainsi. Avec une liste, VACUUM ne traite que ces tables.

VACUUM ANALYZE fait un VACUUM, puis un ANALYZE sur chaque table sélectionnée. C'est une combinaison pratique pour les scripts de maintenance de routine. Voir la description d'ANALYZE pour avoir plus de détails sur ce qu'il traite.

Le VACUUM standard (sans FULL) récupère simplement l'espace et le rend disponible pour une réutilisation. Cette forme de la commande peut opérer en parallèle avec les opérations normales de lecture et d'écriture de la table, car elle n'utilise pas de verrou exclusif. Néanmoins, l'espace récupéré n'est pas renvoyé au système de fichiers dans la plupart des cas ; il est conservé pour être réutilisé dans la même table. Elle nous permet aussi d'exploiter plusieurs CPU pour traiter les index. Cette fonctionnalité est connue sous le nom *vacuum parallélisé*. Pour désactiver cette fonctionnalité, l'option PARALLEL peut être utilisée en spécifiant le nombre de workers à zéro. VACUUM FULL réécrit le contenu entier d'une table dans un nouveau fichier sur disque sans espace supplémentaire, permettant ainsi à l'espace non utilisé d'être retourné au système d'exploitation. Cette forme est bien plus lente et requiert un verrou de type ACCESS EXCLUSIVE sur chaque table pendant qu'elle s'exécute.

Quand la liste d'options est entourée de parenthèses, les options peuvent être écrites dans n'importe quel ordre. Sans parenthèses, les options doivent être écrit dans l'ordre exact décrit ci-dessus. La

syntaxe avec parenthèse a été ajoutée dès la version 9.0 de PostgreSQL ; la syntaxe sans parenthèse est maintenant considérée comme obsolète.

Paramètres

`FULL`

Choisit un vacuum « full », qui récupère plus d'espace, mais est beaucoup plus long et prend un verrou exclusif sur la table. Cette méthode requiert aussi un espace disque supplémentaire car il écrit une nouvelle copie de la table et ne supprime l'ancienne copie qu'à la fin de l'opération. Habituellement, cela doit seulement être utilisé quand une quantité importante d'espace doit être récupérée de la table.

`FREEZE`

Choisit un « gel » agressif des lignes. Indiquer `FREEZE` est équivalent à réaliser un `VACUUM` avec les paramètres vacuum_freeze_min_age et vacuum_freeze_table_age configurés à zéro. Un gel agressif est toujours effectué quand la table est réécrite, cette option est donc redondante quand `FULL` est spécifié.

`VERBOSE`

Affiche un rapport détaillé de l'activité de vacuum sur chaque table.

`ANALYZE`

Met à jour les statistiques utilisées par l'optimiseur pour déterminer la méthode la plus efficace pour exécuter une requête.

`DISABLE_PAGE_SKIPPING`

Habituellement, `VACUUM` ignorera certains blocs en se basant sur la carte de visibilité. Les blocs connues pour être entièrement gelés peuvent toujours être ignorés, et ceux où toutes les lignes sont connues pour être visibles par toutes les transactions peuvent être ignorées sauf lors de l'exécution d'un vacuum agressif. De plus, en dehors d'un vacuum agressif, certains blocs peuvent être ignorés pour éviter d'attendre la fin de leur utilisation par d'autres sessions. Cette option désactive entièrement ce comportement permettant d'ignorer certains blocs, et a pour but d'être utilisé uniquement quand le contenu de la carte de visibilité semble suspect, ce qui peut arrive seulement s'il y a un problème matériel ou logiciel causant une corruption de la base de données.

`SKIP_LOCKED`

Indique que `VACUUM` ne doit pas attendre la disponibilité d'un verrou en conflit sur une table : si une table ne peut pas être verrouillée immédiatement sans attente, la table est ignorée. Notez que, même avec cette option, `VACUUM` pourrait toujours être bloqué lors de l'ouverture des index de la table. De plus, `VACUUM ANALYZE` pourrait toujours être bloqué lors de l'accès aux lignes de l'échantillon pour les partitions, les enfants dans le cadre d'un héritage de tables, et certains types de tables distantes. De plus, alors que `VACUUM` traite de façon standard toutes les partitions des tables partitionnées spécifiées, cette option fera en sorte que `VACUUM` ignorera toutes les partitions s'il y a un verrou en conflit sur la table partitionnée.

`INDEX_CLEANUP`

Habituellement, `VACUUM` ignorera le traitement des index quand il y a très peu de lignes mortes dans la table. Quand cela arrive, le coût de traitement de tous les index de la table devrait dépasser fortement l'intérêt de supprimer les lignes mortes de l'index. Cette option peut être utilisée pour forcer un `VACUUM` à traiter les index quand il y a plus de zéro ligne morte. La valeur par défaut est `AUTO`, ce qui permet au `VACUUM` d'ignorer le traitement des index quand il estime que c'est approprié. Si `INDEX_CLEANUP` est configuré à `ON`, `VACUUM` supprimera de façon conservatrice

toutes les lignes mortes des index. Ceci pourrait être utile pour une compatibilité descendante avec les anciennes versions de PostgreSQL pour lesquelles ceci était le comportement par défaut.

INDEX_CLEANUP peut aussi être configuré à OFF pour forcer VACUUM à *toujours* ignorer le traitement des index, même s'il y a un grand nombre de lignes mortes dans la table. Ceci pourrait être utile quand il est nécessaire que VACUUM termine aussi rapidement que possible pour éviter une réutilisation des identifiants de transaction(voir Section 25.1.5). Néanmoins, le mécanisme de sécurité du wraparound contrôlé par vacuum_failsafe_age se déclenchera automatiquement pour éviter un échec de la réutilisation des identifiants. Si le nettoyage des index n'est pas réalisé régulièrement, les performances pourraient en souffrir car, comme la table est modifiée, les index accumuleraient des lignes mortes et la table elle-même accumulerait des pointeurs des pointeurs de lignes mortes qui ne peuvent être supprimés jusqu'au nettoyage de l'index.

Cette option n'a pas d'effet pour les tables qui n'ont pasd'index et est ignoré si l'option FULL est utilisée. Elle n'a aussi aucun effet sur le mécanisme de sécurité sur la réutilisation ds identifiants de transactions. Lorsqu'elle est déclenchée, le traitement des indexsera ignoré même si INDEX_CLEANUP est configuré à ON.

PROCESS_TOAST

Indique que VACUUM doit tenter de traiter la table TOAST correspondante pour chaque relation, si cette table existe. Ceci est normalement le comportement désiré et c'est de ce fait le comportement par défaut. Configurer cette option à false peut être utile quand il est seulement nécessaire de traiter la table principale. Cette option est requise quand l'option FULL est utilisée.

TRUNCATE

Indique que VACUUM doit tenter de tronquer toute page vide en fin de table et permettre que l'espace disque des pages tronquées soit rendu au système d'exploitation. Ceci est le comportement désiré habituellement et est le comportement par défaut sauf si l'option vacuum_truncate a été désactivée pour la table en cours de traitement. Configurer cette option à false peut être utile pour éviter un verrou ACCESS EXCLUSIVE sur la table que le troncage requiert. Cette option est ignorée si l'option FULL est utilisée.

PARALLEL

Effectue les phases de vacuum et de nettoyage d'index lors d'un VACUUM en parallèle en utilisant *integer* processus en tâche de fond (pour les détails de chaque phase de vacuum, référez-vous au Tableau 28.41). Le nombre de workers utilisés pour réaliser l'opération est égal au nombre d'index sur la relation qui supporte le vacuum parallélisé, qui est limité par le nombre de workers spécifiés avec la clause PARALLEL qui est elle-même limitée par le paramètre max_parallel_maintenance_workers. Un index peut participer à un vacuum parallélisé si et seulement si la taille de l'index est supérieur à la valeur du paramètre min_parallel_index_scan_size. Notez qu'il n'est pas garanti que le nombre de workers parallélisés indiqué dans *integer* sera utilisé lors de l'exécution. Il est possible qu'un vacuum s'exécute avec moins de workers que spécifié, voire sans workers du tout. Seul un worker peut être utilisé par index. Donc les workers de parallélisation sont lancés seulement quand il y a au moins 2 index dans la table. Les workers pour le vacuum sont lancés avant le début de chaque phase et se terminent à la fin de la phase. Ces comportements pourraient changer dans une version future. Cette option ne peut pas être utilisée avec l'option FULL.

boolean

Indique si l'option sélectionnée doit être activée ou désactivée. Vous pouvez écrire TRUE, ON ou 1 pour activer l'option, et FALSE, OFF ou 0 pour la désactiver. La valeur *boolean* peut aussi être omise, auquel cas TRUE est supposé.

integer

Spécifie une valeur entière non négative passée à l'option sélectionnée.

nom_table

Le nom (optionnellement qualifié par le nom d'un schéma) d'une table ou d'une vue matérialisée à traiter par vacuum. Si la table spécifiée est partitionnée, toutes les partitions enfants seront traitées.

nom_colonne

Le nom d'une colonne spécifique à analyser. Par défaut, toutes les colonnes. Si une liste de colonnes est spécifiée, ANALYZE en est déduit.

Sorties

Lorsque VERBOSE est précisé, VACUUM indique sa progression par des messages indiquant la table en cours de traitement. Différentes statistiques sur les tables sont aussi affichées.

Notes

Pour exécuter un VACUUM sur une table, vous devez habituellement être le propriétaire de la table ou un superutilisateur. Néanmoins, les propriétaires de la base de données sont autorisés à exécuter VACUUM sur toutes les tables de leurs bases de données, sauf sur les catalogues partagés. Cette restriction signifie qu'un vrai VACUUM sur une base complète ne peut se faire que par un superutilisateur.) VACUUM ignorera toutes les tables pour lesquelles l'utilisateur n'a pas le droit d'exécuter un VACUUM.

VACUUM ne peut pas être exécuté à l'intérieur d'un bloc de transactions.

Pour les tables ayant des index GIN, VACUUM (sous n'importe quelle forme) termine aussi toutes les insertions d'index en attente, en déplaçant les entrées d'index aux bons endroits dans la structure d'index GIN principale. Voir la Section 70.4.1 pour les détails.

Nous recommandons que toutes les bases de données soient traitées par vacuum fréquemment pour supprimer les lignes mortes. PostgreSQL inclut un outil appelé « autovacuum » qui peut automatiser une maintenance de routine avec VACUUM. Pour plus d'informations sur ce traitement manuel et automatique, voir Section 25.1.

L'option FULL n'est pas recommandée en usage normal, mais elle peut être utile dans certains cas. Par exemple, si vous avez supprimé ou mis à jour l'essentiel des lignes d'une table et si vous voulez que la table diminue physiquement sur le disque pour n'occuper que l'espace réellement nécessaire et pour que les parcours de table soient plus rapides. Généralement, VACUUM FULL réduit plus la table qu'un simple VACUUM.

L'option PARALLEL est utilisée seulement pour le vacuum. Si cette option est spécifiée avec l'option ANALYZE, elle n'affecte pas ANALYZE.

VACUUM peut engendrer une augmentation substantielle du trafic en entrées/sorties pouvant causer des performances diminuées pour les autres sessions actives. Du coup, il est quelque fois conseillé d'utiliser la fonctionnalité du délai du vacuum basé sur le coût. Pour un vacuum en parallèle, chaque processus dort proportionnellement au travail effectué par ce processus. Voir la Section 20.4.4 pour des informations supplémentaires.

Chaque processus exécutant VACUUM sans l'option FULL indiquera sa progression dans la vue pg_stat_progress_vacuum. Les processus exécutant VACUUM FULL indiqueront leur progression dans la vue pg_stat_progress_cluster. Voir Section 28.4.3 et Section 28.4.4 pour les détails.

Exemples

Pour nettoyer une seule table onek, l'analyser pour l'optimiseur et afficher un rapport détaillé de l'activité du VACUUM :

```
VACUUM (VERBOSE, ANALYZE) onek;
```

Compatibilité

Il n'y a pas de commande VACUUM dans le standard SQL.

Voir aussi

vacuumdb, Section 20.4.4, Section 25.1.6, Section 28.4.3, Section 28.4.4

VALUES

VALUES — calcule un ensemble de lignes

Synopsis

```
VALUES ( expression [, ...] ) [, ...]
    [ ORDER BY expression_de_tri [ ASC | DESC | USING operateur ]
 [, ...] ]
    [ LIMIT { nombre | ALL } ]
    [ OFFSET debut ] [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ nombre ] { ROW | ROWS } ONLY ]
```

Description

`VALUES` calcule une valeur de ligne ou un ensemble de valeurs de lignes spécifiées par des expressions. C'est généralement utilisé pour générer une « table statique » à l'intérieur d'une commande plus large mais elle peut aussi être utilisée séparément.

Quand plus d'une ligne est indiquée, toutes les lignes doivent avoir le même nombre d'éléments. Les types de données des colonnes de la table résultante sont déterminés en combinant les types explicites et les types inférés des expressions apparaissant dans cette colonne, en utilisant les mêmes règles que pour l'`UNION` (voir Section 10.5).

À l'intérieur de grosses commandes, `VALUES` est autorisé au niveau de la syntaxe partout où la commande `SELECT` l'est. Comme la grammaire traite cette commande comme un `SELECT`, il est possible d'utiliser les clauses `ORDER BY`, `LIMIT` (ou de façon équivalente `FETCH FIRST`) et `OFFSET` avec une commande `VALUES`.

Paramètres

`expression`

Une constante ou une expression à calculer et à insérer à l'emplacement indiqué dans la table résultante (ensemble de lignes). Dans une liste `VALUES` apparaissant en haut d'une commande `INSERT`, une *`expression`* peut être remplacée par `DEFAULT` pour demander l'insertion de la valeur par défaut de la colonne de destination. `DEFAULT` ne peut pas être utilisé quand `VALUES` apparaît dans d'autres contextes.

`expression_de_tri`

Une expression ou un entier indiquant comment trier les lignes de résultat. Cette expression peut faire référence aux colonnes de `VALUES` en tant que `column1`, `column2`, etc. Pour plus de détails, voir Clause ORDER BY dans la documentation SELECT.

`operateur`

Un opérateur de tri. Pour plus de détails, voir Clause ORDER BY dans la documentation SELECT.

`nombre`

Le nombre maximum de lignes à renvoyer. Pour plus de détails, voir Clause LIMIT dans la documentation SELECT.

debut

Le nombre de lignes à échapper avant de commencer à renvoyer des lignes. Pour plus de détails, Clause LIMIT dans la documentation SELECT.

Notes

Évitez les listes VALUES comprenant un très grand nombre de lignes car vous pourriez rencontrer des problèmes comme un manque de mémoire et/ou des performances pauvres. Un VALUES apparaissant dans un INSERT est un cas spécial (parce que le type des colonnes est trouvé à partir de la table cible du INSERT et n'a donc pas besoin d'être deviné en parcourant la liste VALUES), du coup il peut gérer des listes plus importantes que dans d'autres contextes.

Exemples

Une simple commande VALUES :

```
VALUES (1, 'un'), (2, 'deux'), (3, 'trois');
```

Ceci renverra une table statique comprenant deux colonnes et trois lignes. En fait, c'est équivalent à :

```
SELECT 1 AS column1, 'un' AS column2
UNION ALL
SELECT 2, 'deux'
UNION ALL
SELECT 3, 'trois';
```

Plus généralement, VALUES est utilisé dans une commande SQL plus importante. L'utilisation la plus fréquente est dans un INSERT :

```
INSERT INTO films (code, titee, did, date_prod, genre)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drame');
```

Dans le contexte de la commande INSERT, les entrées d'une liste VALUES peuvent être DEFAULT pour indiquer que la valeur par défaut de la colonne ciblée doit être utilisée :

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comédie', '82 minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drame', DEFAULT);
```

VALUES peut aussi être utilisé là où un sous-SELECT peut être écrit, par exemple dans une clause FROM :

```
SELECT f.*
  FROM films f, (VALUES('MGM', 'Horreur'), ('UA', 'Sci-Fi')) AS t
 (studio, genre)
  WHERE f.studio = t.studio AND f.genre = t.genre;

UPDATE employes SET salaire = salaire * v.augmentation
  FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (no_dep,
 cible, augmentation)
```

```
 WHERE employees.no_dep = v.no_dep AND employees.ventes >=
v.cible;
```

Notez qu'une clause AS est requise quand VALUES est utilisé dans une clause FROM, par exemple dans un SELECT. Il n'est pas nécessaire de spécifier les noms de toutes les colonnes dans une clause AS c'est une bonne pratique (les noms des colonnes par défaut pour VALUES sont column1, column2, etc dans PostgreSQL mais ces noms pourraient être différents dans d'autres SGBD).

Quand VALUES est utilisé dans INSERT, les valeurs sont toutes automatiquement converties dans le type de données de la colonne destination correspondante. Quand elle est utilisée dans d'autres contextes, il pourrait être nécessaire de spécifier le bon type de données. Si les entrées sont toutes des constantes litérales entre guillemets, convertir la première est suffisante pour déterminer le type de toutes :

```
SELECT * FROM machines
WHERE adresse_ip IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'),
 ('192.168.1.43'));
```

Astuce

Pour de simples tests IN, il est préférable de se baser sur des listes de valeurs pour IN que d'écrire une requête VALUES comme indiquée ci-dessus. La méthode des listes de valeurs simples requiert moins d'écriture et est souvent plus efficace.

Compatibilité

VALUES est conforme au standard SQL. Les clauses LIMIT et OFFSET sont des extensions PostgreSQL ; voir aussi SELECT.

Voir aussi

INSERT, SELECT

Applications client de PostgreSQL

Cette partie contient les informations de référence concernant les applications client et les outils de PostgreSQL. Ces commandes ne sont pas toutes destinées à l'ensemble des utilisateurs. Certaines nécessitent des privilèges spécifiques. La caractéristique commune à toutes ces applications est leur fonctionnement sur toute machine, indépendamment du serveur sur lequel se trouve le serveur de base de données.

Lorsqu'ils sont spécifiés en ligne de commande, la casse du nom d'utilisateur et du nom de la base est respectée -- si un nom contient un espace ou des caractères spéciaux alors il faut l'encadrer par des guillemets. La casse des noms des tables et des autres identifiants n'est pas conservée, sauf indication contraire dans la documentation. Pour conserver cette casse il faut utiliser des guillemets.

Table des matières

clusterdb 2125
createdb 2128
createuser 2132
dropdb 2137
dropuser 2140
ecpg 2143
pg_amcheck 2146
pg_basebackup 2152
pgbench 2162
pg_config 2188
pg_dump 2191
pg_dumpall 2206
pg_isready 2214
pg_receivewal 2216
pg_recvlogical 2221
pg_restore 2225
pg_verifybackup 2235
psql 2238
reindexdb 2285
vacuumdb 2289

clusterdb

clusterdb — Grouper une base de données PostgreSQL

Synopsis

`clusterdb` [*connection-option*...] [`--verbose` | `-v`] [`--table` | `-t` *table*] ... [*nom_base*]

`clusterdb` [*connection-option*...] [`--verbose` | `-v`] `--all` | `-a`

Description

clusterdb est un outil de tri de tables au sein d'une base de données PostgreSQL. Il trouve les tables précédemment triées et les trie à nouveau sur l'index utilisé lors du tri initial. Les tables qui n'ont jamais été triées ne sont pas affectées.

clusterdb est un enrobage de la commande SQL CLUSTER. Il n'y a pas de différence réelle entre le tri d'une base par cet outil ou par d'autres méthodes d'accès au serveur.

Options

clusterdb accepte les arguments suivants en ligne de commande :

`-a`
`--all`

Grouper toutes les bases de données.

`[-d] nom_base`
`[--dbname=]nom_base`

Le nom de la base de données à trier quand `-a`/`--all` n'est pas utilisé. Si cette option n'est pas indiquée, le nom de la base de données est lu à partir de la variable d'environnement PGDATABASE. Si cette dernière n'est pas initialisée, le nom de l'utilisateur spécifié pour la connexion est utilisé. Le paramètre *nom_base* peut être une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargent toutes les options en ligne de commande conflictuelles.

`-e`
`--echo`

Les commandes engendrées par clusterdb et envoyées au serveur sont affichées.

`-q`
`--quiet`

Aucun message de progression n'est affiché.

`-t table`
`--table=table`

Seule la table *table* est triée. Plusieurs tables peuvent être traitées en même temps en utilisant plusieurs fois l'option `-t`.

`-v`
`--verbose`

Affiche des informations détaillées lors du traitement.

`-V`
`--version`

Affiche la version de clusterdb puis quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de clusterdb, puis quitte

clusterdb accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *`hôte`*
`--host` *`hôte`*

Le nom de la machine hôte sur laquelle le serveur fonctionne. Si la valeur commence par une barre oblique (slash), elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *`port`*
`--port=`*`port`*

Le port TCP ou l'extension du fichier du socket local de domaine Unix sur lequel le serveur attend les connexions.

`-U` *`nomutilisateur`*
`--username=`*`nomutilisateur`*

Le nom de l'utilisateur utilisé pour la connexion.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force clusterdb à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car clusterdb demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, clusterdb perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`--maintenance-db=`*`nom_base_maintenance`*

Quand l'option `-a`/`--all` est utilisée, il se connecte à cette base pour récupérer la liste des bases à traiter. Si cette option n'est pas ajoutée, la base `postgres` sera utilisée. Si cette base n'existe pas, la base `template1` sera utilisée. Le nom de la base peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront les options en ligne de commande conflictuelles. De plus, les paramètres de la chaîne de connexion autres que le nom de la base lui-même seront réutilisés lors de la connexion aux autres bases.

Environnement

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Paramètres de connexion par défaut.

`PG_COLOR`

Spécifie s'il faut utiliser de la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 34.15).

Diagnostics

En cas de difficulté, voir CLUSTER et psql qui présentent les problèmes et messages d'erreur éventuels. Le serveur de bases de données doit fonctionner sur l'hôte cible. De plus, toutes les configurations de connexion par défaut et variables d'environnement utilisées par la bibliothèque client libpq s'appliquent.

Exemples

Trier toutes les tables de la base de données `test` :

```
$ clusterdb test
```

Trier la seule table `foo` de la base de données nommée `xyzzy` :

```
$ clusterdb --table=foo xyzzy
```

Voir aussi

CLUSTER

createdb

createdb — Créer une nouvelle base de données PostgreSQL

Synopsis

`createdb` [*option_connexion*...] [*option*...] [*nom_base*] [*description*]

Description

createdb crée une nouvelle base de données.

Normalement, l'utilisateur de la base de données qui exécute cette commande devient le propriétaire de la nouvelle base de données. Néanmoins, un propriétaire différent peut être spécifié via l'option `-O`, sous réserve que l'utilisateur qui lance la commande ait les droits appropriés.

createdb est un enrobage de la commande SQL `CREATE DATABASE`. Il n'y a pas de réelle différence entre la création de bases de données par cet outil ou à l'aide d'autres méthodes d'accès au serveur.

Options

createdb accepte les arguments suivants en ligne de commande :

nom_base

Le nom de la base de données à créer. Le nom doit être unique parmi toutes les bases de données PostgreSQL de ce groupe. La valeur par défaut est le nom de l'utilisateur courant.

description

Le commentaire à associer à la base de données créée.

`-D` *tablespace*
`--tablespace=`*tablespace*

Le tablespace par défaut de la base de données (la syntaxe prise en compte est la même que celle d'un identifiant qui accepte les guillemets doubles).

`-e`
`--echo`

Les commandes engendrées par createdb et envoyées au serveur sont affichées.

`-E` *locale*
`--encoding=`*locale*

L'encodage des caractères à utiliser dans la base de données. Les jeux de caractères supportés par le serveur PostgreSQL sont décrits dans Section 24.3.1.

`-l` *locale*
`--locale=`*locale*

Indique la locale à utiliser dans cette base de données. C'est équivalent à préciser à la fois `--lc-collate` et `--lc-ctype`.

`--lc-collate=`*locale*

Indique le paramètre LC_COLLATE utilisé pour cette base de données.

`--lc-ctype=`*`locale`*

Indique le paramètre LC_CTYPE utilisé pour cette base de données.

`--icu-locale=`*`locale`*

Indique l'identifiant de la locale ICU à utiliser pour cette base de données si le fournisseur de locale ICU est sélectionnée.

`--locale-provider={libc|icu}`

Indique le fournisseur de locale pour la collation par défaut de la base de données.

`-O` *`propriétaire`*
`--owner=`*`propriétaire`*

Le propriétaire de la base de données. (la syntaxe prise en compte est la même que celle d'un identifiant qui accepte les guillemets doubles).

`-S` *`strategy`*
`--strategy=`*`strategy`*

Indique la stratégie de création de la base. Voir CREATE DATABASE STRATEGY pour plus de détails.

`-T` *`modèle`*
`--template=`*`modèle`*

La base de données modèle. (la syntaxe prise en compte est la même que celle d'un identifiant qui accepte les guillemets doubles)

`-V`
`--version`

Affiche la version de createdb puis quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de createdb, puis quitte.

Les options `-D`, `-l`, `-E`, `-O` et `-T` correspondent aux options de la commande SQL sous-jacente `CREATE DATABASE`, à consulter pour plus d'informations sur ces options.

createdb accepte aussi les arguments suivants en ligne de commande, pour les paramètres de connexion :

`-h` *`hôte`*
`--host=`*`hôte`*

Le nom de l'hôte sur lequel le serveur est en cours d'exécution. Si la valeur commence avec un slash (NDT : barre oblique, /), elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *`port`*
`--port=`*`port`*

Le port TCP ou l'extension du fichier socket de domaine Unix local sur lequel le serveur attend les connexions.

`-U` *`nomutilisateur`*
`--username=`*`nomutilisateur`*

Le nom de l'utilisateur utilisé pour la connexion.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force createdb à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car createdb demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, createdb perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`--maintenance-db=`*`nom_base_maintenance`*

Spécifie le nom de la base de donnée à laquelle se connecter pour créer la nouvelle base de donnée. Si elle n'est pas spécifiée, la base de données `postgres` est utilisée ; si elle n'existe pas (ou si il s'agit du nom de la nouvelle base à créer), la base `template1` sera utilisée. Ce nom de base peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront toutes les options en ligne de commande conflictuelles.

Environnement

PGDATABASE

S'il est configuré, précise le nom de la base de données à créer. Peut-être surchargé sur la ligne de commande.

PGHOST
PGPORT
PGUSER

Paramètres de connexion par défaut. PGUSER détermine aussi le nom de la base de données à créer si ce dernier n'est pas spécifié sur la ligne de commande ou par PGDATABASE.

PG_COLOR

Indique s'il faut utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 34.15).

Diagnostics

En cas de difficulté, on peut se référer à CREATE DATABASE et psql qui présentent les problèmes éventuels et les messages d'erreurs. Le serveur de bases de données doit être en cours d'exécution sur l'hôte cible. De plus, tous les paramètres de connexion et variables d'environnement par défaut utilisés par la bibliothèque d'interface libpq s'appliquent.

Exemples

Créer la base de données `demo` sur le serveur de bases de données par défaut :

```
$ createdb demo
```

Créer la base de données `demo` sur le serveur hébergé sur l'hôte `eden`, port 5000, en utilisant la base de données modèle `template0`, voici la commande en ligne de commande ainsi que la commande SQL sous-jacente :

```
   $ createdb -p 5000 -h eden -T template0 -e demo
CREATE DATABASE demo TEMPLATE template0;
```

Voir aussi

dropdb, CREATE DATABASE

createuser

createuser — Définir un nouveau compte utilisateur PostgreSQL

Synopsis

createuser [*option_connexion*...] [*option*...] [*nom_utilisateur*]

Description

createuser crée un nouvel utilisateur PostgreSQL (ou, plus précisément, un rôle). Seuls les superutilisateurs et les utilisateurs disposant du droit CREATEROLE peuvent créer de nouveaux utilisateurs. createuser ne peut de ce fait être invoqué que par un utilisateur pouvant se connecter en superutilisateur ou en utilisateur ayant le droit CREATEROLE.

Pour créer un rôle disposant de l'attribut SUPERUSER, REPLICATION ou BYPASSRLS, il est impératif de se connecter en superutilisateur ; la connexion avec un rôle ne disposant que du droit CREATEROLE n'est pas suffisante. Être superutilisateur implique la capacité d'outrepasser toutes les vérifications de droits d'accès à la base de données ; l'attribut SUPERUSER ne doit pas être accordé à la légère. CREATEROLE donne aussi accès à des droits très étendus.

createuser est un enrobage de la commande SQL CREATE ROLE. Il n'y a pas de différence réelle entre la création d'utilisateurs par cet outil ou au travers d'autres méthodes d'accès au serveur.

Options

createuser accepte les arguments suivant en ligne de commande

nom_utilisateur

Le nom de l'utilisateur à créer. Ce nom doit être différent de tout rôle de l'instance courante de PostgreSQL.

-c *numéro*
--connection-limit=*numéro*

Configure le nombre maximum de connexions simultanées pour le nouvel utilisateur. Par défaut, il n'y a pas de limite.

-d
--createdb

Le nouvel utilisateur est autorisé à créer des bases de données.

-D
--no-createdb

Le nouvel utilisateur n'est pas autorisé à créer des bases de données. Cela correspond au comportement par défaut.

-e
--echo

Les commandes engendrées par createuser et envoyées au serveur sont affichées.

-E
--encrypted

Cette option est obsolète mais est toujours acceptée pour raison de compatibilité descendante.

`-g` *`role`*
`--role=`*`role`*

Indique le rôle auquel ce rôle sera automatiquement ajouté comme nouveau membre. Plusieurs rôles auquels ce rôle sera ajouté comme membre peuvent être spécifiés en utilisant plusieurs fois l'option `-g`.

`-i`
`--inherit`

Le nouveau rôle hérite automatiquement des droits des rôles dont il est membre. Comportement par défaut.

`-I`
`--no-inherit`

Le nouveau rôle n'hérite pas automatiquement des droits des rôles dont il est membre.

`--interactive`

Demande le nom de l'utilisateur si aucun n'a été fourni sur la ligne de commande, et demande aussi les attributs équivalents aux options `-d`/`-D`, `-r`/`-R`, `-s`/`-S` si les options en ligne de commande n'ont pas été explicitement indiquées. (Cela correspond au comportement par défaut de PostgreSQL 9.1.)

`-l`
`--login`

Le nouvel utilisateur est autorisé à se connecter (son nom peut être utilisé comme identifiant initial de session). Comportement par défaut.

`-L`
`--no-login`

Le nouvel utilisateur n'est pas autorisé à se connecter. (Un rôle sans droit de connexion est toujours utile pour gérer les droits de la base de données.)

`-P`
`--pwprompt`

L'utilisation de cette option impose à createuser d'afficher une invite pour la saisie du mot de passe du nouvel utilisateur. Cela n'a pas d'utilité si l'authentification par mot de passe n'est pas envisagée.

`-r`
`--createrole`

Le nouvel utilisateur sera autorisé à créer, modifier, supprimer, ajouter un commentaire, modifier le label de sécurité, et donner ou supprimer les membres dans d'autres rôles. Autrement dit, cet utilisateur aura l'attribut `CREATEROLE`. Voir création de rôle pour plus de détails sur les possibilités offertes par ce droit.

`-R`
`--no-createrole`

Le nouvel utilisateur n'est pas autorisé à créer de nouveaux rôles. Cela correspond au comportement par défaut.

`-s`
`--superuser`

Le nouvel utilisateur a les privilèges superutilisateur.

`-S`
`--no-superuser`

Le nouvel utilisateur n'a pas les privilèges superutilisateur. Cela correspond au comportement par défaut.

`-V`
`--version`

Affiche la version de createuser, puis quitte.

`--replication`

Le nouvel utilisateur a l'attribut `REPLICATION`, décrit plus en détails dans la documentation pour CREATE ROLE.

`--no-replication`

Le nouvel utilisateur n'a pas l'attribut `REPLICATION`, décrit plus en détails dans la documentation pour CREATE ROLE.

createuser accepte aussi les arguments suivant en ligne de commande pour les paramètres de connexion :

`-h` *`hôte`*
`--host=`*`hôte`*

Le nom de l'hôte sur lequel le serveur est en cours d'exécution. Si la valeur commence avec un slash (/), elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *`port`*
`--port=`*`port`*

Le port TCP ou l'extension du fichier socket de domaine Unix sur lequel le serveur attend des connexions.

`-U` *`nomutilisateur`*
`--username=`*`nomutilisateur`*

Nom de l'utilisateur utilisé pour la connexion (pas celui à créer).

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force createuser à demander un mot de passe (pour la connexion au serveur, pas pour le mot de passe du nouvel utilisateur).

Cette option n'est jamais obligatoire car createuser demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, createuser perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Environnement

PGHOST
PGPORT
PGUSER

Paramètres de connexion par défaut

PG_COLOR

Indique s'il faut utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 34.15).

Diagnostics

En cas de problèmes, on peut consulter CREATE ROLE et psql qui fournissent des informations sur les problèmes potentiels et les messages d'erreur. Le serveur de la base de données doit être en cours d'exécution sur l'hôte cible. De plus, tout paramétrage de connexion par défaut et toute variable d'environnement utilisée par le client de la bibliothèque libpq s'applique.

Exemples

Créer un utilisateur `joe` sur le serveur de bases de données par défaut :

```
$ createuser joe
```

Pour créer un utilisateur `joe` sur le serveur de base de données avec le mode interactif :

```
$ createuser --interactive joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

Créer le même utilisateur, `joe`, sur le serveur `eden`, port 5000, sans interaction, avec affichage de la commande sous-jacente :

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

Créer l'utilisateur `joe`, superutilisateur, et lui affecter immédiatement un mot de passe :

```
$ createuser -P -s -e joe
Enter password for new role: xyzzy
Enter it again: xyzzy
+CREATE ROLE joe PASSWORD 'SCRAM-SHA-256$4096:44560wPMLfjqiAzyPDZ/
eQ==$4CA054rZlSFEq8Z3FEhToBTa2X6KnWFxFkPwIbKoDe0=:L/
nbSZRCjp6RhOhKK56GoR1zibCCSePKshCJ9lnl3yw=' SUPERUSER CREATEDB
 CREATEROLE INHERIT LOGIN;
```

Dans l'exemple ci-dessus, le nouveau mot de passe n'est pas affiché lorsqu'il est saisi. Il ne l'est ici que pour plus de clareté. Comme vous le voyez, le mot de passe est chiffré avant d'être envoyé au client.

Voir aussi

dropuser, CREATE ROLE

dropdb

dropdb — Supprimer une base de données PostgreSQL

Synopsis

`dropdb` [*option_connexion*...] [*option*...] *nom_base*

Description

dropdb supprime une base de données PostgreSQL. L'utilisateur qui exécute cette commande doit être superutilisateur ou le propriétaire de la base de données.

dropdb est un enrobage de la commande SQL `DROP DATABASE`. Il n'y a aucune différence réelle entre la suppression de bases de données avec cet outil et celles qui utilisent d'autres méthodes d'accès au serveur.

Options

dropdb accepte les arguments suivants en ligne de commande :

nom_base

Le nom de la base de données à supprimer.

`-e`
`--echo`

Les commandes engendrées et envoyées au serveur par dropdb sont affichées.

`-f`
`--force`

Tente de fermer toutes les connexions existantes à la base de données cible avant de la supprimer. Voir DROP DATABASE pour plus d'informations sur cette option.

`-i`
`--interactive`

Une confirmation préalable à toute destruction est exigée.

`-V`
`--version`

Affiche la version de dropdb puis quitte.

`--if-exists`

Permet de ne pas déclencher d'erreur si la base de données n'existe pas. Un simple message d'avertissement est retourné dans ce cas.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de dropdb, puis quitte.

dropdb accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *`hôte`*
`--host=`*`hôte`*

Le nom d'hôte de la machine sur laquelle le serveur fonctionne. Si la valeur débute par une barre oblique (/ ou slash), elle est utilisée comme répertoire de la socket de domaine Unix.

`-p` *`port`*
`--port=`*`port`*

Le port TCP ou l'extension du fichier de la socket locale de domaine Unix sur laquelle le serveur attend les connexions.

`-U` *`nomutilisateur`*
`--username=`*`nomutilisateur`*

Le nom de l'utilisateur utilisé pour la connexion.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force dropdb à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car dropdb demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, dropdb perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`--maintenance-db=`*`nom_base_maintenance`*

Spécifie le nom de la base de données à laquelle se connecter pour supprimer la base de donnée spécifiée. Si elle n'est pas spécifiée, la base de donnée `postgres` est utilisée ; si elle n'existe pas (ou si il s'agit du nom de la base à supprimer), la base `template1` est utilisée. Ce nom de base peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront toutes les options en ligne de commande conflictuelles.

Environnement

`PGHOST`
`PGPORT`
`PGUSER`

Paramètres de connexion par défaut

`PG_COLOR`

Indique s'il faut utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 34.15).

Diagnostics

En cas de difficultés, il peut être utile de consulter DROP DATABASE et psql, sections présentant les problèmes éventuels et les messages d'erreur.

Le serveur de base de données doit fonctionner sur le serveur cible. Les paramètres de connexion éventuels et les variables d'environnement utilisés par la bibliothèque cliente libpq s'appliquent.

Exemples

Détruire la base de données `demo` sur le serveur de bases de données par défaut :

```
$ dropdb demo
```

Détruire la base de données `demo` en utilisant le serveur hébergé sur l'hôte `eden`, qui écoute sur le port 5000, avec demande de confirmation et affichage de la commande sous-jacente :

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE demo;
```

Voir aussi

createdb, DROP DATABASE

dropuser

dropuser — Supprimer un compte utilisateur PostgreSQL

Synopsis

`dropuser` [*option_connexion*...] [*option*...] [*nomutilisateur*]

Description

dropuser supprime un utilisateur. Seuls les superutilisateurs et les utilisateurs disposant du droit `CREATEROLE` peuvent supprimer des utilisateurs (seul un superutilisateur peut supprimer un superutilisateur).

dropuser est un enrobage de la commande SQL `DROP ROLE`. Il n'y a pas de différence réelle entre la suppression des utilisateurs à l'aide de cet outil ou à l'aide d'autres méthodes d'accès au serveur.

Options

dropuser accepte les arguments suivants en ligne de commande :

nomutilisateur

Le nom de l'utilisateur PostgreSQL à supprimer. Un nom est demandé s'il n'est pas fourni sur la ligne de commande et que l'option `-i` / `--interactive` est utilisé.

`-e`
`--echo`

Les commandes engendrées et envoyées au serveur par dropuser sont affichées.

`-i`
`--interactive`

Une confirmation est demandée avant la suppression effective de l'utilisateur. La commande demande aussi le nom de l'utilisateur si aucun nom n'a été fourni sur la ligne de commande.

`-V`
`--version`

Affiche la version de dropuser puis quitte.

`--if-exists`

Ne renvoie pas d'erreur si l'utilisateur n'existe pas. Un message d'avertissement est envoyé dans ce cas.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de dropuser, puis quitte

dropuser accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *hôte*
`--host=`*hôte*

Le nom d'hôte de la machine sur lequel le serveur fonctionne. Si la valeur commence par une barre oblique (/ ou slash), elle est utilisée comme répertoire du socket de domaine Unix.

`-p` *`port`*
`--port=`*`port`*

Le port TCP ou l'extension du fichier du socket local de domaine Unix sur lequel le serveur attend les connexions.

`-U` *`nomutilisateur`*
`--username=`*`nomutilisateur`*

Le nom de l'utilisateur utilisé pour la connexion.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force dropuser à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car dropuser demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, dropuser perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Environnement

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Paramètres de connexion par défaut.

`PG_COLOR`

Indique s'il faut utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 34.15).

Diagnostics

En cas de difficultés, il peut être utile de consulter DROP ROLE et psql, sections présentant les problèmes éventuels et les messages d'erreur.

Le serveur de base de données doit fonctionner sur le serveur cible. Les paramètres de connexion éventuels et les variables d'environnement utilisés par la bibliothèque cliente libpq s'appliquent.

Exemples

Supprimer l'utilisateur `joe` de la base de données par défaut :

```
$ dropuser joe
```

Supprimer l'utilisateur `joe` sur le serveur hébergé sur l'hôte `eden`, qui écoute sur le port 5000, avec demande de confirmation et affichage de la commande sous-jacente :

```
$ dropuser -p 5000 -h eden -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE joe;
```

Voir aussi

createuser, DROP ROLE

ecpg

ecpg — Préprocesseur C pour le SQL embarqué

Synopsis

ecpg [*option*...] *fichier*...

Description

ecpg est le préprocesseur du SQL embarqué pour les programmes écrits en C. Il convertit des programmes écrits en C contenant des instructions SQL embarqué en code C normal. Pour se faire, les appels au SQL sont remplacés par des appels spéciaux de fonctions. Les fichiers en sortie peuvent être traités par n'importe quel compilateur C.

ecpg convertit chaque fichier en entrée, donné sur la ligne de commande, en un fichier C correspondant. Si le nom d'un fichier en entrée n'a pas d'extension, .pgc est supposé. L'extension du fichier sera remplacée par .c pour construire le nom du fichier en sortie. Mais ce nom peut aussi être surchargé en utilisant l'option -o.

Si un nom de fichier en entrée est simplement -, ecpg lit le source du programme à partir de l'entrée standard (et écrit sur la sortie standard, sauf si ce comportement est surchargé avec l'option -o).

Cette page de référence ne décrit pas le langage SQL embarqué. Voir Chapitre 36 pour plus d'informations sur ce thème.

Options

ecpg accepte les arguments suivants en ligne de commande :

-c

Engendre automatiquement du code C à partir de code SQL. Actuellement, cela fonctionne pour EXEC SQL TYPE.

-C *mode*

Initialise un mode de compatibilité. *mode* peut être INFORMIX, INFORMIX_SE ou ORACLE.

-D *symbole*[=*valeur*]

Définit un symbole du préprocesseur, tout comme la directive EXEC SQL DEFINE directive. Si aucune *valeur* n'a été indiqué, le symbole est défini avec la valeur 1.

-h

Traite les fichiers d'en-tête. Quand cette option est utilisée, l'extension du fichier en sortie devient .h, et non pas .c, et l'extension par défaut du fichier en entrée est .pgh, et non pas .pgc. De plus, l'option -c est forcée.

-i

Les fichiers d'en-tête du système sont également analysés.

-I *répertoire*

Spécifie un chemin d'inclusion supplémentaire, utilisé pour trouver les fichiers inclus via EXEC SQL INCLUDE. Par défaut, il s'agit de . (répertoire courant), /usr/local/include, du

répertoire de fichiers entêtes de PostgreSQL défini à la compilation (par défaut : `/usr/local/pgsql/include`), puis de `/usr/include`, dans cet ordre.

`-o` *`nom_fichier`*

Indique le nom du fichier de sortie, *`nom_fichier`*, utilisé par `ecpg`. Écrire `-o -` pour envoyer toute la sortie sur la sortie standard.

`-r` *`option`*

Sélectionne un comportement en exécution. *`option`* peut avoir une des valeurs suivantes :

`no_indicator`

Ne pas utiliser d'indicateurs mais utiliser à la place des valeurs spéciales pour représenter les valeurs NULL. Historiquement, certaines bases de données utilisent cette approche.

`prepare`

Préparer toutes les instructions avant de les utiliser. Libecpg conservera un cache d'instructions préparées et réutilisera une instruction si elle est de nouveau exécutée. Si le cache est plein, libecpg libérera l'instruction la moins utilisée.

`questionmarks`

Autoriser les points d'interrogation comme marqueur pour des raisons de compatibilité. C'était la valeur par défaut il y a longtemps.

`-t`

Active la validation automatique (autocommit) des transactions. Dans ce mode, chaque commande SQL est validée automatiquement, sauf si elle est à l'intérieur d'un bloc de transaction explicite. Dans le mode par défaut, les commandes ne sont validées qu'à l'exécution de `EXEC SQL COMMIT`.

`-v`

Affiche des informations supplémentaires dont la version et le chemin des entêtes.

`--version`

Affiche la version de ecpg et quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de ecpg et quitte.

Notes

Lors de la compilation de fichiers C prétraités, le compilateur a besoin de trouver les fichiers d'en-tête ECPG dans le répertoire des entêtes de PostgreSQL. De ce fait, il faut généralement utiliser l'option `-I` lors de l'appel du compilateur (c'est-à-dire `-I/usr/local/pgsql/include`).

Les programmes C qui utilisent du SQL embarqué doivent être liés avec la bibliothèque `libecpg`. Cela peut peut être effectué, par exemple, en utilisant les options de l'éditeur de liens `-L/usr/local/pgsql/lib -lecpg`.

La valeur réelle des répertoires, fonction de l'installation, peut être obtenue par l'utilisation de la commande pg_config.

Exemples

Soit un fichier source C contenant du SQL embarqué nommé `prog1.pgc`. Il peut être transformé en programme exécutable à l'aide des commandes suivantes :

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_amcheck

pg_amcheck — vérifie la corruption d'une ou plusieurs bases de données PostgreSQL

Synopsis

`pg_amcheck` [*option*...] [*nom_base*]

Description

pg_amcheck exécute les fonctions de vérification de corruption fournies par amcheck sur une ou plusieurs bases de données, avec des options pour sélectionner schémas, tables et index à vérifier, et s'il faut réaliser les tests en parallèle et, si oui, le nombre de connexions parallélisées à établir et utiliser.

Seuls les relations de tables ordinaires et de tables TOAST, les vues matérialisées, les séquences et les index BTree sont actuellement acceptés. Les autres types de relation sont silencieusement ignorés.

Si `nom_base` est précisé, elle devra correspondre au nom d'une base à vérifier, et aucune autre option de sélection de base ne devra être présente. Sinon, toutes les bases correspondantes seront vérifiées. Si aucune de ces options n'est présente, la base de données par défaut sera vérifiée. Les options de sélection de bases incluent `--all`, `--database` et `--exclude-database`. Elles incluent aussi `--relation`, `--exclude-relation`, `--table`, `--exclude-table`, `--index` et `--exclude-index`, mais seulement quand de telles options sont utilisées avec un motif en trois parties(par exemple, `mabase*.monschema*.marelation*`). Enfin, elles incluent `--schema` et `--exclude-schema` quand ces options sont utilisées avec un motif en deux parties(par exemple, `mabase*.monschema*`).

`nom_base` peut aussi être remplacé par une chaîne de connexion.

Options

Les lignes de commande suivante contrôlent ce qui doit être vérifié :

`-a`
`--all`

Vérifie toutes les bases de données, sauf celles exclues avec `--exclude-database`.

`-d` *`motif`*
`--database=`*`motif`*

Vérifie les bases de données correspondant au *`motif`*, indiqué en dehors de celles exclues par l'option `--exclude-database`. Cette option peut être utilisée plus d'une fois.

`-D` *`motif`*
`--exclude-database=`*`motif`*

Exclut les bases de données correspondant au *`motif`* indiqué. Cette option peut être utilisée plus d'une fois.

`-i` *`motif`*
`--index=`*`motif`*

Vérifie les index correspondant au *`motif`* indiqué, sauf s'ils sont exclus d'une autre façon. Cette option peut être utilisée plus d'une fois.

Ceci est similaire à l'option `--relation`, sauf qu'il s'applique seulement aux index, et non pas aux autres types de relation.

`-I` *`motif`*
`--exclude-index=`*`motif`*

Exclut les index correspondant au *`motif`* indiqué. Cette option peut être utilisée plus d'une fois.

Ceci est similaire à l'option `--exclude-relation`, sauf qu'il s'applique seulement aux index, et non pas aux autres types de relation.

`-r` *`motif`*
`--relation=`*`motif`*

Vérifie les relations correspondant au *`motif`* indiqué, sauf si elles sont exclues autrement. Cette option peut être utilisée plus d'une fois.

Les motifs peuvent être sans qualification de schéma, par exemple `marelation*`, ou ils peuvent avoir une qualification comme `monschema*.marelation*` ou une qualification de base et de schéma, par exemple `mabase*.monschema*.marelation*`. Un motif avec une qualification de base ajoutera la correspondance de bases à la liste des bases de données à vérifier.

`-R` *`motif`*
`--exclude-relation=`*`motif`*

Exclut les relations correspondant au *`motif`* indiqué. Cette option peut être utilisée plus d'une fois.

Comme avec `--relation`, le *`motif`* peut être sans qualification, qualité du schéma, qualifié du schéma et de la base.

`-s` *`motif`*
`--schema=`*`motif`*

Vérifie les tables et indexes des schémas correspondant au *`motif`* indiqué, sauf s'ils sont exclus autrement. Cette option peut être utilisée plus d'une fois.

Pour sélectionner seulement les tables dans les schémas correspondant à un motif particulier, considérez l'utilisation de quelque chose comme `--table=MOTIFSCHEMA.* --no-dependent-indexes`. Pour sélectionner seulement les index, considérez l'utilisation de quelque chose comme `--index=MOTIFSCHEMA.*`.

Un motif de schémas pourrait être qualifié du nom de base. Par exemple, vous pourriez écrire `--schema=mabase*.monschema*` pour sélectionner les schémas correspondant à `monschema*` dans les bases de données correspondant à `mabase*`.

`-S` *`motif`*
`--exclude-schema=`*`motif`*

Exclut les tables et index dans les schémas correspondant au *`motif`* indiqué. Cette option peut être utilisée plus d'une fois.

Comme avec `--schema`, le motif peut être qualifié du nom de la base.

`-t` *`motif`*
`--table=`*`motif`*

Vérifie les tables correspondant au *`motif`* indiqué, sauf si elles sont exclues autrement. Cette option peut être utilisée plus d'une fois.

Ceci est similaire à l'option `--relation`, sauf qu'elle s'applique seulement aux tables, vues matérialisées et séquences, et non pas aux index.

`-T` *`motif`*
`--exclude-table=`*`motif`*

Exclut les tables correspondant au *`motif`* indiqué. Cette option peut être utilisée plus d'une fois.

Ceci est similaire à l'option `--exclude-relation`, sauf qu'elle s'applique seulement aux tables, vues matérialisées et séquences, et non pas aux index.

`--no-dependent-indexes`

Par défaut, si une table est vérifiée, tout index btree de cette table est lui-aussi vérifié, même s'ils ne sont pas explicitement sélectionnés par une option telle que `--index` ou `--relation`. Cette option désactive ce comportement.

`--no-dependent-toast`

Par défaut, si une table est vérifiée, sa table TOAST sera aussi vérifiée, même si elle n'est pas explicitement sélectionnée par une option telle que `--table` ou `--relation`. Cette option désactive ce comportement.

`--no-strict-names`

Par défaut, si un argument de `--database`, `--table`, `--index` ou `--relation` ne correspond à aucun objet, ceci est traité comme une erreur fatale. Cette option diminue le niveau de l'erreur à un simple message d'avertissement.

Les options en ligne de commande suivantes contrôlent la vérification des tables :

`--exclude-toast-pointers`

Par défaut, quand un pointeur TOAST est rencontré sur une table, une recherche est réalisée pour s'assurer qu'elle référence toujours des entrés apparemment valides dans la table TOAST. Ces vérifications peuvent être assez lentes, et cette option peut être utilisée pour ne pas les faire.

`--on-error-stop`

Après avoir affiché toutes les corruptions du premier bloc d'une table om une corruption a été trouvé, arrête le traitement de cette table et continue avec la prochaine table ou le prochain index.

Notez que cette vérification des index s'arrête toujours après le premier bloc corrompu. Cette option a seulement les tables pour cible.

`--skip=`*`option`*

Si `all-frozen` est indiqué, les vérifications de corruption de table ignoreront les blocs des tables marqués comme gelés entièrement.

Si `all-visible` est indiqué, les vérifications de corruption de table ignoreront les blocs des tables marqués comme visibles entièrement.

Par défaut, aucun bloc n'est ignoré. Ceci peut être indiqué avec `none` mais comme il s'agit de la valeur par défaut, il n'est pas nécessaire de le mentionner.

`--startblock=`*`bloc`*

Commence la vérification au numéro de bloc mentionné. Une erreur surviendra si la table en cours de vérification a moins de blocs que ce numéro. Cette option ne s'applique pas aux index et est probablement uniquement utile lors de la vérification d'une seule table. Voir `--endblock` pour d'autres mises en garde.

`--endblock=`*`bloc`*

Termine la vérification au numéro de bloc mentionné. Une erreur surviendra si la table en cours de vérification a moins de blocs que ce numéro. Cette option ne s'applique pas aux index et est

probablement uniquement utile lors de la vérification d'une seule table. Si une table normale et une table TOAST sont vérifiées, cette option s'appliquera aux deux mais des blocs supérieurs du TOAST pourraient être accédés pour valider les pointeurs des valeurs TOAST, sauf si cela est désactivé avec l'option `--exclude-toast-pointers`.

Les options en ligne de commande suivantes contrôlent la vérification des index B-tree :

`--heapallindexed`

Pour chaque index vérifié, vérifie la présence de toutes les lignes de la table comme enregistrements dans l'index en utilisant l'option `heapallindexed` d'amcheck.

`--parent-check`

Pour chaque index btree vérifié, utilisez la fonction `bt_index_parent_check` d' amcheck, qui réalise des vérifications supplémentaires des relations parent/enfant lors de la vérification de l'index.

Le comportement par défaut est d'utiliser la fonction `bt_index_check` d'amcheck, mais notez que l'utilisation de l'option `--rootdescend` sélectionne implicitement `bt_index_parent_check`.

`--rootdescend`

Pour chaque index vérifié, retrouve les lignes au niveau feuille en réalisant une nouvelle recherche du bloc racine pour chaque ligne en utilisant l'option `rootdescend` d'amcheck.

Utiliser cette option implicitement sélectionne aussi l'option `--parent-check`.

Cette forme de vérification a été écrit à l'origine pour aider au développement des fonctionnalités de l'index btree. Elle sera d'utilisation limitée, voire inutile, pour aider à la détection de corruptions survenant réellement. Elle pourrait aussi faire que la vérification de corruption prenne beaucoup plus de temps et consomme beaucoup plus de ressources sur le serveur.

> **Avertissement**
>
> Les vérifications supplémentaires réalisées avec les index B-tree quand l'option `--parent-check` ou l'option `--rootdescend` sont spécifiées nécessitent des verrous relativement fort sur les relations. Ce sont les seules vérifications qui bloquent les modifications concurrentes de données par les commandes `INSERT`, `UPDATE` et `DELETE`.

Les options en ligne de commande suivantes contrôlent la connexion au serveur :

`-h` *`nom_hôte`*
`--host=`*`nom_hôte`*

Indique le nom d'hôte de la machine qui héberge le serveur de bases de données. Si la valeur commence par une barre oblique (/), elle est utilisée comme répertoire pour la socket de domaine Unix.

`-p` *`port`*
`--port=`*`port`*

Indique le port TCP ou l'extension du fichier local de la socket de domaine Unix sur lequel le serveur attend les connexions.

`-U`
`--username=`*`nom_rôle`*

Nom d'utilisateur pour la connexion.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force pg_amcheck à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car pg_amcheck demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, pg_amcheck perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`--maintenance-db=`*`nom_base`*

Indique le nom de la base ou une chaîne de connexion à utiliser pour récupérer la liste des bases à vérifier. Si ni `--all` ni d'autres options incluant un motif de nom de bases de données n'est utilisé, cette connexion n'est pas nécessaire et cette option ne fera rien. Sinon, tous les paramètres de la chaîne de connexion autre que le nom de le base, qui sont inclus dans la valeur pour cette option, seront aussi utilisés pour se connecter aux bases de données à vérifier. Si cette option est omise, la valeur par défaut est `postgres` ou, en cas d'échec, `template1`.

D'autres options sont aussi disponibles :

`-e`
`--echo`

Afficher sur la sortie standard toutes les requêtes SQL envoyées au serveur.

`-j` *`nombre`*
`--jobs=`*`nombre`*

Utilise *`nombre`* connexions concurrentes au serveur ou une par objet à vérifier, suivant ce qui fera le moins de connexions.

Par défaut, utilise une seule connexion.

`-P`
`--progress`

Affiche des informations de progression. Ces dernières inclient le nombre de relations pour lesquelles la vérification est terminée, et la taille totale de ces relations. Elles incluent aussi le nombre total de relations qui devront être vérifiées et la taille estimée de ces relations.

`-v`
`--verbose`

Affiche plus de messages. En particulier, cela affichera un message pour chaque relation en cours de vérification, et cela accroitera le niveau de détails utilisé pour les erreurs du serveur.

`-V`
`--version`

Affiche la version de pg_amcheck, puis quitte.

`--install-missing`
`--install-missing=`*`schema`*

Installe toute extension manquante requise pour vérifier les bases de données. S'ils ne sont pas encore installés, les objets de chaque extension seront installés dans le schéma *`schema`*. S'il n'est pas spécifié, ils iront dans le schéma `pg_catalog`.

Actuellement, la seule extension requise est amcheck.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_amcheck, puis quitte.

Notes

pg_amcheck est conçu pour fonctionner avec PostgreSQL 14.0 et ultérieur.

Voir aussi

amcheck

pg_basebackup

pg_basebackup — réalise une sauvegarde de base d'une instance PostgreSQL

Synopsis

`pg_basebackup` [*option*...]

Description

pg_basebackup est utilisé pour effectuer une sauvegarde de base d'une instance PostgreSQL en cours d'exécution. Elle se fait sans affecter les autres clients du serveur de bases de données, et peut être utilisée pour une restauration à un certain point dans le temps (PITR, voir Section 26.3) ou comme point de départ pour un serveur secondaire, par envoi des journaux de transactions ou par le flux de réplication (voir Section 27.2).

pg_basebackup fait une copie exacte des fichiers de l'instance, en s'assurant que le système entre et sort du mode sauvegarde automatiquement. Les sauvegardes sont toujours faites sur l'ensemble de l'instance ; il n'est donc pas possible de sauvegarder une base individuelle ou des objets d'une base. Pour de telles sauvegardes, utiliser un outil comme pg_dump.

La sauvegarde se fait via une connexion PostgreSQL standard qui utilise le protocole de réplication. La connexion doit se faire avec un rôle doté de l'attribut `REPLICATION` ou `SUPERUSER` (voir Section 22.2), et `pg_hba.conf` doit explicitement permettre la connexion de réplication. Le serveur doit aussi être configuré avec un max_wal_senders suffisamment élevé pour fournir au moins une connexion disponible pour la sauvegarde, et une autre pour le transfert par flux des journaux de transactions (si utilisé).

Plusieurs commandes `pg_basebackup` peuvent tourner en même temps, mais, du point de vue des performances, il est généralement préférable de n'en faire qu'une seule, puis de faire une copie du résultat.

pg_basebackup peut effectuer une sauvegarde non seulement à partir du serveur primaire, mais aussi d'un serveur secondaire. Pour cela, paramétrez le secondaire pour accepter les connexions pour réplication (c'est-à-dire configurez les paramètres `max_wal_senders` et hot_standby, et configurez le fichier `pg_hba.conf` de façon approprié). Il sera aussi nécessaire d'activer full_page_writes sur le serveur primaire.

Il est à noter qu'il existe des limites à la sauvegarde à chaud depuis un serveur secondaire :

- Le fichier d'historique de la sauvegarde n'est pas créé dans l'instance sauvegardée.
- pg_basebackup ne peut forcer le serveur standby à basculer vers un nouveau fichier WAL à la fin de la sauvegarde. Quand vous utilisez `-X none`, si l'activité en écriture sur le primaire est basse, pg_basebackup peut avoir besoin d'attendre un long moment pour que le dernier fichier WAL requis par la sauvegarde soit archivé. Dans ce cas, il pourrait être utile d'exécuter `pg_switch_wal` ur le primaire pour causer un changement immédiat de fichier WAL.
- Si le serveur secondaire est promu en tant que primaire durant la sauvegarde à chaud, celle-ci échouera.
- Toutes les entrées WAL nécessaires à la sauvegarde doivent disposer de suffisamment de pages complètes, ce qui nécessite d'activer `full_page_writes` sur le primaire.

La vue `pg_stat_progress_basebackup` du serveur sauvegardé permet de suivre la progression d'une sauvegarde de base effectuée avec pg_basebackup. Voir Section 28.4.5 pour les détails.

Options

Les options de ligne de commande suivantes contrôlent l'emplacement et le format du résultat.

`-D` *répertoire*
`--pgdata=`*répertoire*

Configure le répertoire cible de la sauvegarde. pg_basebackup créera le répertoire (et les répertoires parents) s'il n'existe pas. S'il existe déjà, il doit être vide.

Quand la sauvegarde est en mode tar, le répertoire doit être spécifié sous la forme d'un tiret (`-`), causant l'écriture du fichier tar sur `stdout`.

Cette option est obligatoire.

`-F` *format*
`--format=`*format*

Sélectionne le format de sortie. *format* peut valoir :

`p`
`plain`

Écrit des fichiers standards, avec l'agencement d'origine du répertoire des données et des tablespaces du serveur source. Si l'instance n'a pas de tablespace supplémentaire, toute l'instance sera placée dans le répertoire cible. Si l'instance contient des tablespaces supplémentaires, le répertoire principal des données sera placé dans le répertoire cible mais les autres tablespaces seront placés dans le même chemin absolu que celui d'origine. (Voir `--tablespace-mapping` pour changer cela.)

C'est le format par défaut.

`t`
`tar`

Écrit des fichiers tar dans le répertoire cible. Le contenu du répertoire principal de données sera écrit dans un fichier nommé `base.tar`, et tous les autres tablespaces seront écrit dans un fichier tar séparé nommé d'après l'OID du tablespace.

Si le répertoire cible est indiqué avec un tiret `-`, le contenu du fichier tar sera écrit sur la sortie standard, ce qui est permet de l'envoyer par un tube vers gzip, par exemple. Ceci n'est autorisé que si l'instance n'a pas de tablespaces supplémentaires, ou que le transfert des WAL par streaming n'est pas utilisé.

`-R`
`--write-recovery-conf`

Crée un fichier `standby.signal` et ajoute les paramètres de connexion dans le fichier `postgresql.auto.conf` du répertoire cible (ou dans le fichier d'archive du répertoire principal des données lors de l'utilisation du format tar). Ceci facilite la configuration d'un serveur secondaire en utilisant les résultats de la sauvegarde.

Le fichier `postgresql.auto.conf` enregistrera les paramètres de connexion et, si indiqué, le slot de réplication utilisé par pg_basebackup, pour que la réplication en streaming utilise plus tard la même configuration.

`-t` *cible*
`--target=`*cible*

Indique au serveur où placer la sauvegarde de base. La cible par défaut est `client`, qui indique que la sauvegarde devrait être envoyée à la machine où pg_basebackup est exécuté. Si la cible est

configuré à `server:/un/chemin`, la sauvegarde sera restaurée sur la machine où le serveur est exécuté dans le répertoire `/un/chemin`. Enregistrer une sauvegarde nécessite des droits de superutilisateur ou d'avoir les droits du rôle `pg_write_server_files`. Si la cible est configuré à `blackhole`, le contenu est ignoré et n'est pas sauvegardé. Ceci devrait seulement être utilisé pour des tests. Cela ne devrait pas concerné une sauvegarde réelle.

Comme le flux de journaux de transactions est implémenté par pg_basebackup plutôt que par le serveur, cette option ne peut pas être utilisée avec `-Xstream`. Comme il s'agit de la valeur par défaut, quand cette option est soumse, vous debez également spécifier soit `-Xfetch` soit `-Xnone`.

`-T` *`ancien_repertoire`*`=`*`nouveau_repertoire`*
`--tablespace-mapping=`*`ancien_repertoire`*`=`*`nouveau_repertoire`*

Déplace le tablespace du répertoire *`ancien_repertoire`* vers le répertoire *`nouveau_repertoire`* pendant la sauvegarde. Pour bien fonctionner, *`ancien_repertoire`* doit correspondre exactement à la spécification du tablespace tel qu'il est actuellement défini sur le serveur source. (Mais il n'y a pas d'erreur s'il n'y a aucun tablespace dans *`ancien_repertoire`* sur le serveur source.) Pendant ce temps, *`nouveau_repertoire`* est un répertoire dans le système de fichiers de l'hôte cible. Comme avec le répertoire principal cible *`nouveau_repertoire`* n'a pas besoin d'exister déjà, mais s'il existe, il doit être vide. *`ancien_repertoire`* et *`nouveau_repertoire`* doivent tous deux être des chemins absolus. S'il se trouve qu'un chemin contient un signe `=`, échappez-le avec un anti-slash (\). Cette option peut être spécifiée plusieurs fois pour différents tablespaces.

Si un tablespace est déplacé de cette façon, les liens symboliques à l'intérieur du répertoire de données principal sont mis à jour pour pointer vers le nouvel emplacement. Du coup, le nouveau répertoire de données est prêt à être utilisé sur la nouvelle instance avec les nouveaux chemins.

Actuellement, cette option ne fonctionne qu'avec le format de sortie plain. Elle est ignorée sur le format tar est sélectionné.

`--waldir=`*`rep_wal`*

Indique l'emplacement du répertoire des journaux de transactions. Par défaut, les journaux de transactions seront placés dans le sous-répertoire `pg_wal` du répertoire cible, mais cette option peut être utilisée pour les placer ailleurs. *`rep_wal`* doit être un chemin absolu. Comme avec le répertoire principal des données, *`waldir`* peut ne pas exister mais s'il existe déjà, il doit être vide. Cette option ne peut être utilisée que quand la sauvegarde est au format plain.

`-X` *`methode`*
`--wal-method=`*`methode`*

Inclut les journaux de transactions requis (fichiers WAL) dans la sauvegarde. Cela inclura toutes les transactions intervenues pendant la sauvegarde. À moins que la méthode `none` ne soit spécifiée, il est possible de démarrer un postmaster sur le répertoire cible sans avoir besoin de consulter les archives des journaux, ce qui rend la sauvegarde complètement autonome.

Les méthodes suivantes sont supportées pour récupérer les journaux de transactions :

`n`
`none`

N'inclut pas les journaux de transactions dans la sauvegarde.

`f`
`fetch`

Les journaux de transactions sont récupérés à la fin de la sauvegarde. Il est donc nécessaire de définir le paramètre wal_keep_size du serveur source suffisamment haut pour que les données WAL requises ne soient pas recyclées avant la fin de la sauvegarde. Si les données

WAL ont été recyclées avant qu'il n'ait été possible de les transférer, la sauvegarde échouera et sera inutilisable.

Quand le format tar est utilisé, les journaux de transactions seront écrits dans le fichier `base.tar`.

`s`
`stream`

Envoie les données des journaux de transactions lors de la sauvegarde. Cette option ouvre une seconde connexion sur le serveur et entame l'envoi du journal de transactions en parallèle de la sauvegarde. À cet effet, ce mécanisme consommera deux connexions, et non pas juste une. Ce mode permet de ne pas avoir à sauvegarder des journaux de transactions additionnels sur le serveur primaire, aussi longtemps que le client pourra suivre le flux des journaux de transactions.

Quand le format tar est utilisé, les journaux de transactions seront écrits dans un fichier séparé nommé `pg_wal.tar` (si le serveur est d'une version antérieure à la version 10, le fichier sera nommé `pg_xlog.tar`).

Cette valeur est la valeur par défaut.

`-z`
`--gzip`

Active la compression gzip de l'archive tar en sortie, avec le niveau de compression par défaut. La compression n'est disponible qu'avec le format tar, et le suffixe `.gz` sera automatiquement ajouté à tous les noms de fichier tar.

`-Z` *`niveau`*
`-Z [{client|server}-]`*`method`*`[:`*`detail`*`]`
`--compress=`*`niveau`*
`--compress=[{client|server}-]`*`method`*`[:`*`detail`*`]`

Réclame la compression de la sauvegarde. Si `client` ou `server` est inclus, cela indique où la compression doit avoir lieu. Compresser sur le serveur réduira la bande passante réseau mais augmentera la consommation CPU du serveur. La valeur par défaut est `client` sauf quand `--target` est utilisé. Dans ce cas, la sauvegarde n'est pas envoyée au client, donc seule la compression serveur est sensible. Quand `-Xstream`, qui est la valeur par défaut, est utilisé, la compression côté serveur n'est pas appliquée aux journaux de transaction. Pour les compresser, utilisez la compression niveau client ou indiquez `-Xfetch`.

La méthode de compression peut être `gzip`, `lz4`, `zstd` ou `none` pour aucune compression. Une chaîne de détails de compression peut être indiquée en option. Si la chaîne de détails est un entier, elle indique l niveau de compression. Sinon elle devra être une ligne d'éléments séparés par des virgules, chacun de la forme `motclé` ou `motclé=valeur`. Actuellement, les mots clés acceptés sont `level` et `workers`.

Si aucun niveau de compression n'est indiqué, le niveau de compression par défaut sera utilisé. Si seulement un niveau est indiqué sans mention d'un algorithme, la compression `gzip` sera utilisée si le niveau est supérieur à zéro, et aucune compression ne sera utilisée si le niveau est 0.

Quand le format tar est utilisé avec `gzip`, `lz4` ou `zstd`, le suffixe, respectivement, `.gz`, `.lz4` ou `.zst` sera automatiquement ajouté aux noms de fichiers tar. Quand le format plain est utilisée, la compression côté client peut ne pas être précisée mais il est toujours possible de réclamer la compression niveau serveur. Si c'est fait, le serveur compressera la sauvegarde pour transmission, et le client la décompressera et l'extraira.

Quand cette option est utilisée en combinaison avec `-Xstream`, `pg_wal.tar` sera compressé en utilisant `gzip` si la compression gzip côté client est sélectionnée, mais ne sera pas compressé

si un autre algorithme de compression est sélectionné ou si la compression côté serveur est sélectionnée.

Les options de ligne de commande suivantes contrôlent la génération de la sauvegarde et l'exécution du programme :

`-c {fast|spread}`
`--checkpoint={fast|spread}`

Configure le mode du checkpoint à immédiat (fast) ou étalé (spread, la valeur par défaut). Voir Section 26.3.3.

`-C`
`--create-slot`

Indique que le slot de réplication nommé par l'option `--slot` doit être créé avant d'exécuter la sauvegarde. Une erreur est levée si le slot existe déjà.

`-l` *`label`*
`--label=`*`label`*

Définit un nom pour la sauvegarde. Sans indication, la valeur par défaut, « `pg_basebackup base backup` », sera utilisée.

`-n`
`--no-clean`

Par défaut, quand `pg_basebackup` échoue avec une erreur, il supprime tous les répertoires qu'il aurait pu créer avant de découvrir qu'il ne peut pas terminer le travail (par exemple, le répertoire de données et le répertoire des journaux de transactions du serveur cible). Cette option désactive le nettoyage, et est donc pratique pour le débogage.

Notez que les répertoires des tablespaces ne sont pas supprimés non plus.

`-N`
`--no-sync`

Par défaut, `pg_basebackup` attendra que tous les fichiers soient écrits de manière sûre sur disque. Cette option demande à `pg_basebackup` de rendre la main sans attendre, ce qui est plus rapide, mais signifie qu'un arrêt brutal du serveur après la sauvegarde peut laisser la sauvegarde de base dans un état corrompu. De manière générale, cette option est utile durant les tests, mais ne devrait pas être utilisée dans un environnement de production.

`-P`
`--progress`

Active l'indicateur de progression. Activer cette option donnera un rapport de progression approximatif lors de la sauvegarde. Comme la base de données peut changer pendant la sauvegarde, ce n'est qu'une approximation, et peut ne pas se terminer à exactement `100%`. En particulier, lorsque les journaux de transactions sont inclus dans la sauvegarde, la quantité totale de données ne peut pas être estimée à l'avance et, dans ce cas, la taille cible estimée va augmenter une fois dépassée l'estimation totale sans les journaux.

`-r` *`taux`*
`--max-rate=`*`taux`*

Configure le taux maximum de transfert de données avec lequel les données sont récupérées du serveur source. Ceci peut se révéler utile pour limiter l'impact de pg_basebackup sur le serveur Les valeurs sont en kilooctets par seconde. Le suffixe `M` indique des mégaoctets par seconde. Un suffixe `k` est aussi accepté, mais n'a pas d'effet supplémentaire. Les valeurs valides vont de 32 ko/s à 1024 Mo/s.

Cette option concerne le transfert du répertoire de données. Le transfert des journaux de transactions n'est affecté que si la méthode de récupération est `fetch`.

`-S` *`nom_slot`*
`--slot=`*`nom_slot`*

Cette option ne peut être utilisée qu'avec l'option `-X stream`. Elle entraîne l'utilisation du slot de réplication indiqué par le flux de réplication des journaux. Si la sauvegarde de base doit être utilisée pour un serveur secondaire avec un slot, ce serveur devra alors utiliser le même nom de slot dans primary_slot_name. Ainsi, on s'assure que le serveur primaire ne supprimera pas les journaux nécessaires entre la fin de la sauvegarde et le début du lancement de la réplication par streaming sur le nouveau secondaire.

Le slot de réplication spécifié doit exister, sauf si l'option `-C` est aussi utilisée.

Si cette option n'est pas spécifiée, et que le serveur supporte les slots de réplication temporaires (versions 10 et supérieures), alors un slot de réplication temporaire sera automatiquement utilisé pour le transfert des WAL par streaming.

`-v`
`--verbose`

Active le mode verbeux. Il affichera des étapes supplémentaires pendant le démarrage et l'arrêt, ainsi que, si le rapport de progression est aussi activé, le nom exact du fichier en cours de traitement.

`--manifest-checksums=`*`algorithm`*

Spécifie l'algorithme à utiliser dans le manifeste de la sauvegarde pour les sommes de contrôle appliquées à chaque fichier. Les algorithmes disponibles sont actuellement `NONE`, `CRC32C`, `SHA224`, `SHA256`, `SHA384` et `SHA512`. Le défaut est `CRC32C`.

Si `NONE` est choisi, le manifeste ne contiendra aucune somme de contrôle. Sinon, il y en aura une pour chaque fichier de la sauvegarde, avec l'algorithme indiqué. De plus, le manifeste contiendra toujours une somme de contrôle `SHA256` de son propre contenu. Les algorithmes `SHA` sont significativement plus consommateurs de CPU que `CRC32C` ; choisir l'un d'eux peut donc augmenter la durée nécessaire à la sauvegarde.

Une fonction de hachage SHA fournit un résumé cryptographiquement sûr de chaque fichier, pour les utilisateurs voulant vérifier que la sauvegarde n'a pas été modifiée ; alors que l'algorithme CRC32C fournit une somme de contrôle bien plus rapide à calculer, bonne pour déceler des erreurs dues à des changements accidentels, mais vulnérable à des modifications ciblées. Notez que, pour qu'il serve face à un adversaire avec un accès à la sauvegarde, il faut stocker de manière sécurisée, ailleurs, le fichier manifeste, ou vérifier d'une manière ou d'une autre qu'il n'a pas été modifié depuis le moment de la sauvegarde.

pg_verifybackup peut être utilisé pour vérifier l'intégrité d'une sauvegarde grâce à son manifeste.

`--manifest-force-encode`

Dans le manifeste de la sauvegarde, force l'encodage hexadécimal de tous les noms de fichiers. Sans cette option, seuls les noms de fichiers non-UTF8 sont ainsi encodés. Cette option est surtout destinée à tester que les outils qui lisent un manifeste de sauvegarde traitent correctement ce cas.

`--no-estimate-size`

Empêche le serveur d'estimer la taille totale de la sauvegarde qui sera renvoyée, et entraînera toujours une valeur `NULL` pour la colonne `backup_total` de `pg_stat_progress_basebackup`.

Sans cette option, la sauvegarde commencera par calculer la taille de toute l'instance, puis procédera à l'envoi du contenu. Cela peut allonger légèrement la sauvegarde, en particulier avant l'envoi des premières données. Si cela est trop long, cette option permet de l'éviter.

Cette option n'est pas autorisée si l'on utilise `--progress`.

`--no-manifest`

Désactive la génération du fichier manifeste de la sauvegarde. Sans cette option, le serveur générera et enverra un manifeste, contrôlable avec pg_verifybackup. Ce manifeste est une liste de tous les fichiers présent dans la sauvegarde, à l'exception d'éventuels fichiers WAL. Pour chaque fichier, il stocke aussi la taille, le moment de la dernière modification et, optionnellement, une somme de contrôle.

`--no-slot`

Empêche la création d'un slot de réplication temporaire pour la sauvegarde.

Par défaut, si le flux de réplication est sélectionné mais qu'aucun nom de slot n'a été donné avec l'option `-S`, alors un slot de réplication temporaire est créé si cela est supporté par le serveur source).

Le but principal de cette option est de permettre de prendre une sauvegarde de base même si le serveur est à cours de slots de réplication. Utiliser les slots est presque toujours la meilleure solution, car cela empêche le serveur de supprimer les WAL nécessaires pendant la sauvegarde.

`--no-verify-checksums`

Désactive la vérification des sommes de contrôles, si elles sont activées sur le serveur sur lequel la sauvegarde est réalisée.

Par défaut, les sommes de contrôles sont vérifiées, et les erreurs produiront un code de sortie différent de zéro. Cependant, la sauvegarde ne sera pas supprimée, comme si l'option `--no-clean` avait été utilisée. Les échecs de vérification des sommes de contrôle seront aussi rapportés dans la vue pg_stat_database.

Les options de ligne de commande suivantes contrôlent la connexion au serveur source :

`-d` *`connstr`*
`--dbname=`*`connstr`*

Indique les paramètres utilisés pour se connecter au serveur sous la forme d'une chaîne de connexion ; elles surchargeront les options en ligne de commande conflictuelles.

Cette option est appelée `--dbname` par cohérence avec les autres applications clientes, mais comme pg_basebackup ne se connecte à aucune base de données particulière dans l'instance, le nom de la base de données dans la chaîne de connexion est ignorée.

`-h` *`hôte`*
`--host=`*`hôte`*

Indique le nom d'hôte de la machine sur laquelle le serveur de bases de données est exécuté. Si la valeur commence par une barre oblique (/), elle est interprétée comme le répertoire de socket de domaine Unix. La valeur par défaut est fournie par la variable d'environnement `PGHOST`, si elle est en place ; sinon une connexion sur la socket de domaine Unix est tentée.

`-p` *`port`*
`--port=`*`port`*

Indique le port TCP ou l'extension du fichier local de socket de domaine Unix sur lequel le serveur écoute les connexions. La valeur par défaut est fournie par la variable d'environnement `PGPORT`, si elle est en place ; sinon il s'agit de la valeur fournie à la compilation.

`-s` *`interval`*
`--status-interval=`*`interval`*

Spécifie le nombre de secondes entre les envois au serveur source des paquets informant de l'état en cours. Des valeurs plus petites permettent une supervision plus précise de la progression de la sauvegarde à partir du serveur source. Une valeur de zéro désactive complètement les mises à jour de statut périodiques, bien qu'une mise à jour sera toujours envoyée sur demande du serveur, pour éviter une déconnexion suite au dépassement d'un délai. La valeur par défaut est de 10 secondes.

`-U` *`nom_utilisateur`*
`--username=`*`nom_utilisateur`*

Indique le nom d'utilisateur pour la connexion.

`-w`
`--no-password`

Enpêche l'affichage d'une demande de mot de passe. Si le serveur en réclame un pour l'authentification, et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile dans les batchs et les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force pg_basebackup à demander un mot de passe avant la connexion au serveur source.

Cette option n'est jamais nécessaire, car pg_basebackup demandera automatiquement un mot de passe si le serveur exige une telle authentification. Néanmoins, pg_basebackup gaspillera une tentative de connexion pour découvrir que le serveur veut ce mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

D'autres options sont aussi disponibles :

`-V`
`--version`

Affiche la version de pg_basebackup, puis quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_basebackup, puis quitte.

Environnement

Cet outil, comme la plupart des outils PostgreSQL, utilise les variables d'environnement supportées par libpq (voir Section 34.15).

La variable d'environnement `PG_COLOR` indique s'il faut utiliser les couleurs dans les messages de diagnostics. Les valeurs possibles sont `always`, `auto`, `never`.

Notes

Au début d'une sauvegarde, un checkpoint doit être réalisé sur le serveur cible. Ceci peut prendre un certain temps (tout spécialement si l'option `--checkpoint=fast` n'est pas utilisée), pendant lequel pg_basebackup semblera inactif.

La sauvegarde inclura tous les fichiers du répertoire de données et des tablespaces, dont les fichiers de configuration, et aussi tout autre fichier placé dans le répertoire par d'autres personnes, à l'exception

de certains fichiers temporaires gérés par PostgreSQL et des fichiers du système d'exploitation. Seuls les fichiers normaux et les répertoires sont cependant copiés. Les liens symboliques utilisés pour les tablespaces sont aussi préservés. Les liens symboliques pointant vers certains répertoires connus de PostgreSQL sont copiés en tant que répertoires vides. Les autres liens symboliques et les fichiers de périphérique spéciaux sont ignorés. Voir Section 55.4 pour des détails précis.

Dans le format plain, les tablespaces seront sauvegardés avec le même chemin que sur le serveur source, sauf si l'option `--tablespace-mapping` est utilisée. Sans elle, restaurer et lancer une sauvegarde de format plain sur le même serveur ne fonctionnera pas si les tablespaces sont utilisés, car la sauvegarde devra écrire dans les mêmes répertoires que ceux des tablespaces originaux.

Quand le format tar est utilisé, il est de la responsabilité de l'utilisateur de décompresser chaque archive tar avant de démarrer un serveur PostgreSQL qui utilisera ces données. S'il existe des tablespaces supplémentaires, les archives tar les concernant doivent être décompressés au même emplacement. Dans ce cas, les liens symboliques pour ces tablespaces seront créés par le serveur, suivant le contenu du fichier `tablespace_map` inclus dans le fichier `base.tar`.

pg_basebackup fonctionne sur les serveurs de même version, ou de version plus ancienne jusqu'à la 9.1. Néanmoins, le streaming des WAL (option `-X`) ne fonctionne qu'avec un serveur en version 9.3 ou ultérieure, et le format tar (`--format=tar`) ne fonctionne qu'avec les serveurs de version 9.5 ou ultérieure.

pg_basebackup conservera les droits de groupe pour les fichiers de données si les droits de groupe sont activés sur l'instance cible.

Exemples

Pour créer une sauvegarde de base du serveur `mon_sgbd` et l'enregistrer dans le répertoire local `/usr/local/pgsql/data` :

```
$ pg_basebackup -h mon_sgbd -D /usr/local/pgsql/data
```

Pour créer une sauvegarde du serveur local avec un fichier tar compressé pour chaque tablespace, et stocker le tout dans le répertoire `sauvegarde`, tout en affichant la progression pendant l'exécution :

```
$ pg_basebackup -D sauvegarde -Ft -z -P
```

Pour créer une sauvegarde d'une base de données locale avec un seul tablespace, et la compresser avec bzip2 :

```
$ pg_basebackup -D - -Ft -X fetch | bzip2 > backup.tar.bz2
```

(Cette commande échouera s'il existe plusieurs tablespaces dans l'instance.)

Pour créer une sauvegarde d'une base locale, où le tablespace situé dans `/opt/ts` doit être déplacé vers `./backup/ts` :

```
$ pg_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

Pour créer une sauvegarde d'un serveur local avec un fichier tar pour chaque tablespace compressé avec gzip au niveau 9, enregistré dans le répertoire `backup` :

```
$ pg_basebackup -D backup -Ft --compress=gzip:9
```

Voir aussi

pg_dump, Section 28.4.5

pgbench

pgbench — Réalise un test de benchmark pour PostgreSQL

Synopsis

`pgbench -i` [*option*...] [*nom_base*]

`pgbench` [*option*...] [*nom_base*]

Description

pgbench est un programme pour réaliser simplement des tests de performance (*benchmark*) sur PostgreSQL. Il exécute la même séquence de commandes SQL en continu, potentiellement avec plusieurs sessions concurrentes puis calcule le taux de transactions moyen (en transactions par secondes). Par défaut, pgbench teste un scénario vaguement basé sur TPC-B, impliquant cinq commandes `SELECT`, `UPDATE` et `INSERT` par transaction. Toutefois, il est facile de tester d'autres scénarios en écrivant vos propres scripts de transactions.

Une sortie classique de pgbench ressemble à ceci :

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
maximum number of tries: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
number of failed transactions: 0 (0.000%)
latency average = 11.013 ms
latency stddev = 7.351 ms
initial connection time = 45.758 ms
tps = 896.967014 (without initial connection time)
```

Les sept premières lignes rapportent certains des plus importants paramètres. La sixième ligne affiche le nombre maximum d'essais pour les transactions avec des erreurs de sérialisation ou des deadlocks (pour plus d'informations, voir Failures and Serialization/Deadlock Retries). La huitième ligne remonte les nombres de transactions réalisées et prévues (ce dernier étant juste le produit du nombre de clients et du nombre de transactions par client) ; ils seront égaux à moins que l'exécution échoue avant la fin ou que des commandes SQL échouent. (Avec le mode `-T`, seul le nombre réel de transactions est affiché.) La ligne suivante affiche le nombre de transactions échouées à cause d'erreurs de sérialisation ou de deadlock (pour plus d'informations, voir Failures and Serialization/Deadlock Retries). La dernière ligne indique le nombre de transactions par seconde.

Les transactions de ce test, proche de TPC-B, nécessitent d'avoir défini au préalable quelques tables spécifiques. Il faut utiliser l'option `-i` (initialisation) de pgbench pour créer et remplir ces tables. (Si vous faites vos tests avec un script personnalisé, vous n'aurez pas besoin de cette étape, mais devrez mettre en place tout ce dont votre script aura besoin.) Une initialisation ressemble à cela :

```
pgbench -i [ autres-options
] nom_base
```

où *nom_base* est le nom de la base de données pré-existante sur laquelle on conduit les tests. (Vous aurez aussi probablement besoin des options `-h`, `-p` et/ou `-U` pour spécifier comment se connecter au serveur de base de données.)

> **Attention**
>
> `pgbench -i` crée quatre tables nommées `pgbench_accounts`, `pgbench_branches`, `pgbench_history` et `pgbench_tellers`, détruisant toute table qui porterait l'un de ces noms. Attention à utiliser une autre base de données si vous avez des tables qui portent ces noms !

Par défaut, avec un facteur d'échelle de 1, les tables contiennent initialement les nombres de lignes suivants :

```
table                   # de lignes
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

Vous pouvez (et, dans la plupart des cas, devriez) augmenter le nombre de lignes en utilisant l'option `-s`. Le facteur de remplissage `-F` peut aussi être utilisée à cet effet.

Une fois la mise en place terminée, vous pouvez lancer vos benchmarks sans inclure l'option `-i`, c'est-à-dire :

```
pgbench [ options ]
nom_base
```

Dans presque tous les cas, vous allez avoir besoin de certaines options pour rendre vos tests plus pertinents. Les options les plus importantes sont : `-c` (le nombre de clients), `-t` (le nombre de transactions), `-T` (l'intervalle de temps) et `-f` (le script à lancer). Vous trouverez ci-dessous toutes les options disponibles.

Options

La partie suivante est divisée en trois sous-parties : des options différentes sont utilisées pendant l'initialisation et pendant les tests ; certaines options sont utiles dans les deux cas.

Options d'initialisation

Pour réaliser l'initialisation, pgbench accepte les arguments suivants en ligne de commande :

dbname

Indique le nom de la base à tester. Si elle n'est pas spécifier, la variable d'environnement `PGDATABASE` est utilisée. Si elle n'est pas configurée, le nom d'utilisateur indiqué pour la connexion est utilisé.

`-i`
`--initialize`

Nécessaire pour passer en mode initialisation.

`-I` *`init_steps`*
`--init-steps=`*`init_steps`*

N'effectue qu'une partie des étapes d'initialisation habituelles. *`init_steps`* spécifie les étapes d'initialisation à exécuter, à raison d'un caractère par étape. Chaque étape est appelée dans l'ordre indiqué. La valeur par défaut est `dtgvp`. Voici la liste des différentes étapes disponibles :

`d` (Détruit)

Supprime toutes les tables pgbench déjà présentes.

`t` (crée Tables)

Crée les tables utilisées par le scénario pgbench standard, à savoir `pgbench_accounts`, `pgbench_branches`, `pgbench_history` et `pgbench_tellers`.

`g` ou `G` (génère des données, côté client ou côté serveur)

Génère des données et les charge dans les tables standards, remplaçant toutes les données déjà présentes.

Avec `g` (génération de données côté client), les données sont générées dans le client `pgbench`, puis envoyées au serveur. Cela utilise une commande `COPY` et consomme beaucoup de bande passante entre le client et le serveur. En version 14 ou supérieure, `pgbench` utilise l'option FREEZE de PostgreSQL pour accélérer les `VACUUM` suivantes, sauf en cas d'utilisation des partitions. Avec `g`, la trace affiche un message toutes les 100 000 lignes lors de la génération de `pgbench_accounts`.

Avec `G` (génération côté serveur), seules de petites requêtes sont envoyées depuis le client `pgbench`, et les données sont ensuite générées sur le serveur. Aucune bande passante significative n'est nécessaire dans cette variante, mais le serveur travaillera plus. Avec `G`, la trace n'affichera aucun message de progression pendant la génération des données.

Par défaut, l'initialisation utilise la génération des données côté client (soit `g`).

`v` (Vacuum)

Appelle `VACUUM` sur les tables standards.

`p` (clés Primaires)

Crée les clés primaires sur les tables standards.

`f` (*Foreign keys*)

Crée les contraintes de clés étrangères entre les différentes tables standards. (Notez que cette étape n'est pas exécutée par défaut.)

`-F` *`fillfactor`*
`--fillfactor=` *`fillfactor`*

Crée les tables `pgbench_accounts`, `pgbench_tellers` et `pgbench_branches` avec le facteur de remplissage (*fillfactor*) spécifié. La valeur par défaut est 100.

`-n`
`--no-vacuum`

Ne réalise pas de VACUUM après l'initialisation. (Cette option supprime l'étape d'initialisation `v`, même si elle était précisée dans `-I`.)

`-q`
`--quiet`

Passe du mode verbeux au mode silencieux, en n'affichant qu'un message toutes les 5 secondes. Par défaut, on affiche un message toutes les 100 000 lignes, ce qui engendre souvent plusieurs lignes toutes les secondes (particulièrement sur du bon matériel)

Ce paramètre n'a pas d'effet si `G` est spécifié dans l'option `-I`.

`-s` *`scale_factor`*
`--scale=` *`scale_factor`*

Multiplie le nombre de lignes générées par le facteur d'échelle (*scale factor*). Par exemple, `-s 100` va créer 10 millions de lignes dans la table `pgbench_accounts`. La valeur par défaut est 1. Lorsque l'échelle dépasse 20 000, les colonnes utilisées pour contenir les identifiants de compte (colonnes `aid`) vont être converties en grands entiers (`bigint`), de manière à être suffisamment grandes pour contenir l'espace des identifiants de compte.

`--foreign-keys`

Crée des contraintes de type clé étrangère entre les tables standards. (Cette option ajoute l'étape d'initialisation `f`, si elle n'est pas déjà présente.)

`--index-tablespace=`*`index_tablespace`*

Crée un index dans le tablespace spécifié plutôt que dans le tablespace par défaut.

`--partition-method=`*`NOM`*

Crée une table partitionnée `pgbench_accounts` par la méthode *`NOM`*. Les valeurs attendues sont `range` ou `hash`. Cette option nécessite que `--partitions` soit différente de zéro. Sans précision, le défaut est `range`.

`--partitions=`*`NUM`*

Crée une table partitionnée `pgbench_accounts` avec *`NUM`* partitions de taille à peu près égale pour le nombre de clients indiqué par l'échelle. Le défaut est `0`, ce qui signifie qu'il n'y a pas de partitionnement.

`--tablespace=`*`tablespace`*

Crée une table dans le tablespace spécifié plutôt que dans le tablespace par défaut.

`--unlogged-tables`

Crée toutes les tables en tant que tables non journalisées (*unlogged tables*) plutôt qu'en tant que tables permanentes.

Options des benchmarks

Pour réaliser un benchmark pgbench accepte les arguments suivants en ligne de commande :

`-b` *`nom_script[@poids]`*
`--builtin=`*`nom_script[@poids]`*

Ajoute le script interne spécifié à la liste des scripts à exécuter. Les scripts internes disponibles sont `tpcb-like`, `simple-update` et `select-only`. L'utilisation des préfixes non ambigus des noms de scripts internes est acceptée. En utilisant le nom spécial `list`, la commande affiche la liste des scripts internes, puis quitte immédiatement.

En option, il est possible d'écrire un poids en entier après `@` pour ajuster la probabilité de sélectionner ce script plutôt que les autres. Le poids par défaut est de 1. Voir ci-dessous pour les détails.

`-c` *`clients`*
`--client=` *`clients`*

Nombre de clients simulés, c'est-à-dire le nombre de sessions concurrentes sur la base de données. La valeur par défaut est à 1.

`-C`
`--connect`

Établit une nouvelle connexion pour chaque transaction, plutôt que de ne le faire qu'une seule fois par session cliente. C'est une option très utile pour mesurer la surcharge engendrée par la connexion.

`-d`
`--debug`

Affiche les informations de debug.

`-D` *`variable`*`=`*`value`*
`--define=`*`variable`*`=`*`value`*

Définit une variable à utiliser pour un script personnalisé Voir ci-dessous pour plus de détails. Il est possible d'utiliser plusieurs fois l'option `-D`.

`-f` *`nom_fichier[@poids]`*
`--file=`*`nom_fichier[@poids]`*

Ajoute un script de transactions nommé *`nom_fichier`* à la liste des scripts à exécuter.

En option, il est possible d'écrire un poids sous la forme d'un entier après le symbole `@` pour ajuster la probabilité de sélectionner ce script plutôt qu'unautre. Le poids par défaut est de 1. (Pour utiliser un nom de fichier incluant un caractère `@`, ajoutez un poids pour qu'il n'y ait pas d'ambiguité, par exemple `filen@me@1`.) Voir ci-dessous pour les détails.

`-j` *`threads`*
`--jobs=` *`threads`*

Nombre de processus utilisés dans pgbench. Utiliser plus d'un thread peut être utile sur des machines possédant plusieurs cœurs. Les clients sont distribués de la manière la plus uniforme possible parmi les threads. La valeur par défaut est 1.

`-l`
`--log`

Rapporte les informations sur chaque transaction dans un fichier journal. Voir ci-dessous pour plus de détails.

`-L` *`limite`*
`--latency-limit=`*`limite`*

Les transactions durant plus de *`limite`* millisecondes sont comptabilisée et rapportées séparément en tant que *late*.

Lorsqu'un bridage est spécifié (`--rate=...`), les transactions accusant un retard sur la planification supérieur à *`limite`* millisecondes, et donc sans aucune chance de respecter la limite de latence, ne sont pas du tout envoyées au serveur. Elles sont comptabilisées et rapportées séparément en tant que *skipped* (ignorées).

Avec l'option `--max-tries`, une transaction échouée à cause d'une anomalie de sérialisation ou d'un deadlock ne sera pas retentée si le temps cumulé de ses tentatives dépasse *`limit`* ms. Pour ne limiter que la durée des tentatives et non leur nombre, utiliser `--max-tries=0`. Par défaut, `--max-tries` vaut 1, et les transactions en erreur à cause de la sérialisation ou d'un deadlock

ne sont pas retentées. Voir Failures and Serialization/Deadlock Retries pour plus de détails sur comment retenter de telles transactions.

`-M` *querymode*
`--protocol=` *querymode*

Protocole à utiliser pour soumettre des requêtes au serveur :

- `simple` : utilisation du protocole de requêtes standards.
- `extended` : utilisation du protocole de requête étendu.
- `prepared` : utilisation du protocole de requête étendu avec instructions préparées.

Comme dans le mode `prepared`, pgbench réutilise le résultat de l'analyse pour la deuxième itération et les suivantes, pgbench s'exécute plus rapidement dans le mode `prepared` que dans les autres modes.

Par défaut, le protocole de requête standard est utilisé (voir Chapitre 55 pour plus d'informations).

`-n`
`--no-vacuum`

Ne réalise pas de VACUUM avant de lancer le test. Cette option est *nécessaire* si vous lancez un scénario de test personnalisé qui n'utilise pas les tables standards `pgbench_accounts`, `pgbench_branches`, `pgbench_history` et `pgbench_tellers` .

`-N`
`--skip-some-updates`

Exécute le script interne simple-update. Raccourci pour `-b simple-update`.

`-P` *sec*
`--progress=` *sec*

Affiche un rapport de progression toutes les *sec* secondes. Ce rapport inclut la durée du test, le nombre de transactions par seconde depuis le dernier rapport et la latence moyenne des transactions, ainsi que la déviation et le nombre de transactions échouées depuis le dernier rapport. Avec le bridage (option `-R`), la latence est calculée en fonction de la date de démarrage ordonnancée de la transaction et non de son temps de démarrage réel, donc elle inclut aussi la latence moyenne du temps d'ordonnancement. Si `--max-tries` est utilisée pour autoriser de nouvelles tentatives après des erreurs de sérialisation ou des deadlocks, le rapport inclut le nombre de transactions retentées et la somme de tous les nouveaux essais.

`-r`
`--report-per-command`

Affiche les statistiques suivantes pour chaque commande après la fin du benchmark : latence moyenne par requête (temps d'exécution du point de vue du client), nombre d'échecs après des erreurs de sérialisation ou des deadlocks, et le nombre de nouvelles tentatives après cela. Le rapport affiche les statistiques sur les nouveaux essais uniquement si `--max-tries` n'est pas égal à 1.

`-R` *rate*
`--rate=`*rate*

Exécute les transactions en visant le débit spécifié, au lieu d'aller le plus vite possible (le défaut). Le débit est donné en transactions par seconde. Si le débit visé est supérieur au maximum possible, la limite de débit n'aura aucune influence sur le résultat.

Pour atteindre ce débit, les transactions sont ordonnancées avec une distribution suivant une loi de Poisson. La date de démarrage prévue se calcule depuis le moment où le client a démarré et pas depuis le moment où la dernière transaction s'est achevée. Cette approche signifie que, si une transaction dépasse sa date de fin prévue, un rattrapage est encore possible pour les suivantes.

Lorsque le bridage est actif, la latence de la transaction rapportée en fin de test est calculée à partir des dates de démarrage ordonnancées, c'est-à-dire qu'elle inclut le temps où chaque transaction attend que la précédente se termine. Le temps d'attente est appelé temps de latence d'ordonnancement, et ses valeurs moyenne et maximum sont rapportées séparément. La latence de transaction par rapport au temps de démarrage réel, c'est-à-dire le temps d'exécution de la transaction dans la base, peut être récupérée en soustrayant le temps de latence d'ordonnancement à la latence précisée dans les journaux.

Si l'option `--latency-limit` est utilisée avec l'option `--rate`, une transaction peut avoir une telle latence qu'elle serait déjà supérieure à limite de latence lorsque la transaction précédente se termine, car la latence est calculée au moment de la date de démarrage planifiée. Les transactions concernées ne sont pas envoyées à l'instance, elles sont complètement ignorées et comptabilisées séparément.

Une latence de planification élevée est une indication que le système n'arrive pas à traiter les transactions à la vitesse demandée, avec les nombres de clients et threads indiqués. Lorsque le temps moyen d'exécution est plus important que l'intervalle prévu entre chaque transaction, les transactions vont prendre du retard une-à-une, et la latence de planification va continuer de croître tout le long de la durée du test. Si cela se produit, vous devrez réduire le taux de transaction que vous avez spécifié.

`-s` *`scale_factor`*
`--scale=`*`scale_factor`*

Affiche le facteur d'échelle dans la sortie de pgbench. Avec les tests internes, ce n'est pas nécessaire ; le facteur d'échelle approprié sera détecté en comptant le nombre de lignes dans la table `pgbench_branches`. Toutefois, lors de l'utilisation d'un benchmark avec un scénario personnalisé (option `-f`), le facteur d'échelle sera affiché à 1 à moins que cette option soit utilisée.

`-S`
`--select-only`

Exécute le script interne select-only. Raccourci pour `-b select-only`.

`-t` *`transactions`*
`--transactions=` *`transactions`*

Nombre de transactions lancées par chaque client. La valeur par défaut est 10.

`-T` *`seconds`*
`--time=`*`seconds`*

Lance le test pour la durée spécifiée en secondes, plutôt que pour un nombre fixe de transactions par client. Les options `-t` et `-T` ne sont pas compatibles.

`-v`
`--vacuum-all`

Réalise un VACUUM sur les quatre tables standards avant de lancer le test. Sans l'option `-n` ou `-v`, pgbench lancera un VACUUM sur les tables `pgbench_tellers` et `pgbench_branches`, puis tronquera `pgbench_history`.

`--aggregate-interval=` *`secondes`*

Taille de l'intervalle d'agrégation (en secondes). Ne peut être utilisée qu'avec l'option -l. Avec cette option, le journal contiendra des résumés par intervalle, comme décrit ci-dessous.

`--failures-detailed`

Dans les traces, par transaction et agrégées, et les rapports manuels et par script, affiche les erreurs regroupées selon les types suivants :

- erreurs de sérialisation ;

- deadlocks ;

Voir Failures and Serialization/Deadlock Retries pour plus d'informations.

`--log-prefix=`*`prefix`*

Définit le préfixe des fichiers logs créés par `--log`. Le défaut est `pgbench_log`.

`--max-tries=`*`number_of_tries`*

Autorise de nouvelles tentatives dans les transactions avec des erreurs de sérialisation ou de verrous mortels, et définit le nombre maximum de ces tentatives. Cette option peut être combinée avec `--latency-limit`, qui limite le temps total de toutes les tentatives de transactions ; cependant vous ne pouvez définir un nombre illimité de tentatives (`--max-tries=0`) sans définir `--latency-limit` ou `--time`. La valeur par défaut est 1, et les transactions avec des erreurs de sérialisation ou des deadlocks ne sont donc pas retentées. Voir Failures and Serialization/Deadlock Retries pour plus de détails sur les nouvelles tentatives sur de telles transactions.

`--progress-timestamp`

Lorsque la progression est affichée (option `-P`), utilise un horodatage de type timestamp (epoch Unix) au lieu d'un nombre de secondes depuis le début de l'exécution. L'unité est la seconde avec une précision en millisecondes après le point. Ceci aide à comparer les traces générées par différents outils.

`--random-seed=`*`graine`*

Fournit la graine du générateur de nombres aléatoires, qui produira alors une séquence d'états initiaux du générateur, un pour chaque thread. Les valeurs pour *`graine`* peuvent être `time` (par défaut, la graine est basée sur l'heure en cours), `rand` (utilise une source fortement aléatoire, et tombe en échec si aucune n'est disponible), ou une valeur entière non signée. Le générateur aléatoire est appelé depuis un script pgbench explicitement (fonctions `random...`) ou implicitement (par exemple l'option `--rate` l'utilise pour planifier les transactions). Si elle est mise en place explicitement, la valeur utilisée comme graine est affichée sur le terminal. N'importe quelle valeur autorisée pour *`graine`* peut aussi être fournie par la variable d'environnement `PGBENCH_RANDOM_SEED`. Pour garantir que la graine fournie couvre tous les cas d'usage possibles, mettez cette fonction en premier ou utilisez la variable d'environnement.

Placer cette variable explicitement permet de reproduire un run `pgbench` exactement identique, du moins en ce qui concerne les nombres aléatoires. Comme l'état du générateur aléatoire est géré par thread, `pgbench` s'exécutera à l'identique s'il y a un client par thread et pas de dépendance externe ou par rapport aux données. D'un point de vue statistique, reproduire des runs est une mauvaise idée, car cela peut masquer la variabilité des performances ou améliorer les performances excessivement, par exemple en appelant les mêmes pages qu'un run précédent. Cependant, ce peut être d'une grande aide pour déboguer, par exemple pour reproduire un cas tordu provoquant une erreur. À utiliser judicieusement.

`--sampling-rate=` *`rate`*

Taux d'échantillonnage utilisé lors de l'écriture des données dans les journaux, afin d'en réduire la quantité. Si cette option est utilisée, n'y sera écrite que la proportion indiquée des transactions. 1.0 signifie que toutes les transactions seront journalisées, 0.05 signifie que 5 % de toutes les transactions le seront.

Pensez à prendre le taux d'échantillonnage en compte en consultant le journal. Par exemple, lorsque vous évaluez le nombre de transactions par seconde, vous devrez multiplier les nombres en conséquence. (Par exemple, avec un taux d'échantillonnage de 0,01, vous n'obtiendrez que 1/100è du débit réel).

`--show-script`*`scriptname`*

Affiche le code réel du script intégré *`scriptname`* sur stderr, et quitte immédiatement.

`--verbose-errors`

Affiche des messages sur toutes les erreurs et échecs (les erreurs sans nouvelles tentatives), et mentionne quelle limite a été dépassée, et de combien, pour les échecs lors d'erreurs de sérialisation ou des deadlocks. (Notez que dans ce cas le débit peut être significativement amélioré.) Voir Failures and Serialization/Deadlock Retries pour plus d'informations.

Options courantes

pgbench accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *`hostname`*
`--host=` *`hostname`*

Le nom du serveur de base de données

`-p` *`port`*
`--port=` *`port`*

Le port d'écoute de l'instance sur le serveur de base de données

`-U` *`login`*
`--username=` *`login`*

Le nom de l'utilisateur avec lequel on se connecte

`-V`
`--version`

Affiche la version de pgbench puis quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pgbench puis quitte.

Code de sortie

Une exécution réussie se terminera avec un code d'erreur 0. Une valeur 1 indique des problèmes comme des options de ligne de commande invalides ou des erreurs internes supposée ne jamais arriver. Des erreurs dès le démarrage du benchmark, comme des échecs aux premières tentatives de connexion, échouent aussi avec le statut 1. Des erreurs pendant l'exécution, comme des erreurs de base de données ou des problèmes dans le script, provoquent une erreur de code 2. Dans ce dernier cas, pgbench affichera des résultats partiels.

Environnement

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Paramètres de connexion par défaut.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise les variables d'environnement supportées par la libpq (voir Section 34.15).

La variable d'environnement `PG_COLOR` précise si l'on doit utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto` et `never`.

Notes

Qu'est-ce qu'une « Transaction » réellement exécutée par pgbench ?

pgbench exécute des scripts de tests choisis de façon aléatoire à partir d'une sélection. Les scripts peuvent inclure des scripts internes, indiqués avec l'option `-b`, et des scripts fournis par l'utilisateur, indiqués avec l'option `-f`. Chaque script peut se voir affecter un poids relatif, à préciser après un caractère `@`, pour modifier sa probabilité de sélection. Le poids par défaut est de `1`. Les scripts avec un poids de `0` sont ignorés.

Le script interne par défaut (aussi appelé avec `-b tpcb-like`) exécute sept commandes par transaction choisies de façon aléatoire parmi `aid`, `tid`, `bid` et `delta`. Le scénario s'inspire du benchmark TPC-B, mais il ne s'agit pas réellement de TPC-B, d'où son nom.

1. `BEGIN;`
2. `UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;`
3. `SELECT abalance FROM pgbench_accounts WHERE aid = :aid;`
4. `UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;`
5. `UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;`
6. `INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);`
7. `END;`

Si vous sélectionnez le script interne `simple-update` (ou `-N`), les étapes 4 et 5 ne sont pas incluses dans la transaction. Cela évitera des contentions au niveau des mises à jour sur ces tables, mais le test ressemblera encore moins à TPC-B.

Si vous sélectionnez le script interne `select-only` (ou `-S`), alors seul le `SELECT` est exécuté.

Scripts personnalisés

pgbench est capable d'utiliser des scénarios de test de performance personnalisés, en remplaçant le script de transactions par défaut (décrit ci-dessus) par un script de transactions lu depuis un fichier spécifié avec l'option (`-f`). Dans ce cas, une « transaction » est comptabilisée comme une exécution du fichier script.

Un fichier script contient une ou plusieurs commandes SQL terminées par des points-virgules. Les lignes vides et les lignes commençant par `--` sont ignorées. Les fichiers scripts peuvent aussi contenir des « méta-commandes », qui seront interprétées par pgbench comme indiqué plus bas.

> **Note**
>
> Avant PostgreSQL 9.6, les commandes SQL comprises dans les fichiers scripts étaient terminées par un retour à la ligne. Elles ne pouvaient donc pas être écrites sur plusieurs lignes. Maintenant, un point-virgule est *requis* pour séparer des commandes SQL consécutives (bien qu'une commande SQL n'en a pas besoin si elle est suivie par une méta-commande). Si vous avez besoin de créer un fichier script qui fonctionne avec les anciennes et nouvelles versions de pgbench, assurez-vous d'écrire chaque commande SQL sur une seule ligne et en terminant avec un point-virgule.

> On part du principe que les scripts pgbench ne contiennent pas de blocs incomplets de transactions SQL. Si à l'exécution le client atteint la fin du script sans terminer le dernier bloc de transaction, il sera arrêté.

Il est possible de procéder facilement à de la substitution de variables dans les fichiers scripts. Les noms de variables doivent consister en lettres (y compris des caractères non latins), chiffres et soulignés (_), mais le premier caractère ne doit pas être un chiffre. Les variables peuvent être instanciées via l'option `-D` de la ligne de commande comme décrit ci-dessus, ou grâce aux méta-commandes décrites ci-dessous. En plus des commandes pré-définies par l'option de la ligne de commande `-D`, quelques variables sont automatiquement prédéfinies, listées sous Tableau 288. Une valeur de ces variables définie via l'option `-D` aura priorité sur la valeur définie automatiquement. Une fois définie, la valeur d'une variable peut être insérée dans les commandes SQL en écrivant `:` *`nom_variable`*. S'il y a plus d'une session par client, chaque session possède son propre jeu de variables. pgbench supports up to 255 variable uses in one statement.

Tableau 288. Variables automatiques de pgbench

Variable	Description
`client_id`	nombre unique permettant d'identifier la session client (commence à zéro)
`default_seed`	graine utilisée par défaut dans les fonctions de hachage et de permutation pseudo-aléatoire
`random_seed`	graine du générateur aléatoire (si pas remplacée avec `-D`)
`scale`	facteur d'échelle courant

Dans les fichiers de scripts, les méta-commandes commencent avec un anti-slash (\) et s'étendent jusqu'à la fin de la ligne, même si elles peuvent s'étendre sur plusieurs lignes en écrivant anti-slash puis un retour chariot. Les arguments d'une méta-commande sont séparés par des espaces vides. Les méta-commandes suivantes sont supportées :

`\gset [`*`prefix`*`]` `\aset [`*`prefix`*`]`

Cette commande peut être utilisée pour marquer la fin de requêtes SQL, prenant ainsi la place du point-virgule final (`;`).

Quand la commande `\gset` est utilisée, la requête SQL précédente doit renvoyer une ligne. Les valeurs de ses colonnes sont enregistrées dans des variables nommées d'après les noms de colonnes, préfixées avec *`prefix`*, si ce dernier est fourni.

Quand la commande `\aset` est utilisée, toutes les requêtes (séparées par `\;`) voient leurs colonnes stockées dans des variables nommées d'après elles, préfixées de *`prefix`* s'il est fourni. Si une requête ne retourne aucune ligne, aucune affectation n'est faite. On peut tester l'existence de la variable pour détecter ce cas. Si une requête retourne plus d'une ligne, la dernière valeur est conservée.

`\gset` et `\aset` ne peuvent être utilisées en mode pipeline, puisque les résultats des requêtes ne sont pas encore disponibles au moment où la commande en a besoin.

L'exemple suivant place la balance finale du compte provenant de la première requête dans la variable *`abalance`*, et remplit les variables *`p_two`* et *`p_three`* avec les entiers provenant de la troisième requête. Le résultat de la deuxième requête est ignoré. Les résultats des deux dernières requêtes combinées sont stockés dans les variables *`four`* et *`five`*.

```
UPDATE pgbench_accounts
  SET abalance = abalance + :delta
  WHERE aid = :aid
```

```
  RETURNING abalance \gset
-- compound of two queries
SELECT 1 \;
SELECT 2 AS two, 3 AS three \gset p_
SELECT 4 AS four \; SELECT 5 AS five \aset
```

\if *expression*
\elif *expression*
\else
\endif

Ce groupe de commandes implémente des blocs conditionnels imbriquables, de manière similaire au `\if` *`expression`* de `psql`. Les expressions conditionnelles sont identiques à celles avec `\set`, les valeurs autres que zéro valant `true`.

`\set` *`nom_variable expression`*

Définit la variable *`nom_variable`* à une valeur définie par *`expression`*. L'expression peut contenir la constante `NULL`, les constantes booléennes `TRUE` et `FALSE`, des constantes entières comme `5432`, des constantes double précision comme `3.14159`, des références à des variables `:`*`nomvariable`*, des opérateurs avec leur priorité et leur associativité habituelles en SQL, des appels de fonction, des expressions conditionnelles génériques SQL avec `CASE` et des parenthèses.

Les fonctions et la plupart des opérateurs retournent `NULL` en cas d'entrée à `NULL`.

En ce qui concerne les conditions, les valeurs numériques différentes de zéro valent `TRUE`, les valeurs numériques à zéro et `NULL` sont `FALSE`.

Des constantes entières ou à virgule flottante ainsi que des opérateurs arithmétiques entiers (`+`, `-`, `*` et `/`), trop larges ou trop petites, renvoient des erreurs de dépassement.

Quand aucune clause finale `ELSE` n'est fournie à un `CASE`, la valeur par défaut est `NULL`.

Exemples :

```
\set ntellers 10 * :scale
\set aid (1021 * random(1, 100000 * :scale)) % \
           (100000 * :scale) + 1
\set divx CASE WHEN :x <> 0 THEN :y/:x ELSE NULL END
```

`\sleep` *`nombre`* `[ us | ms | s ]`

Entraîne la suspension de l'exécution du script pendant la durée spécifiée en microsecondes (`us`), millisecondes (`ms`) ou secondes (`s`). Si l'unité n'est pas définie, l'unité par défaut est la seconde. Ce peut être soit un entier constant, soit une référence `:`*`nom_variable`* vers une variable retournant un entier.

Exemple :

```
\sleep 10 ms
```

`\setshell` *`nom_variable commande`* `[` *`argument`* `... ]`

Définit la variable *`nom_variable`* comme le résultat d'une commande shell nommée *`commande`* aves le(s) *`argument`*(s) donné(s). La commande doit retourner un entier sur la sortie standard.

commande et chaque *argument* peuvent être soit une constante de type text, soit une référence *:nom_variable* à une variable. Si vous voulez utiliser un *argument* commençant avec un symbole deux-points, écrivez un deux-points supplémentaire au début de l'*argument*.

Exemple :

```
\setshell variable_à_utiliser commande
 argument_litéral :variable
::literal_commencant_avec_deux_points
```

`\shell` *commande* [*argument* ...]

Identique à `\setshell`, mais le résultat de la commande sera ignoré.

Exemple :

```
\shell command
 literal_argument :variable ::literal_starting_with_colon
```

`\startpipeline`
`\endpipeline`

Ces commandes définissent le début et la fin de requêtes SQL. En mode pipeline, celles-ci sont envoyées au serveur sans attendre le résultat des requêtes précédentes. Voir Section 34.5 pour plus de détails. Le mode pipeline impose l'utilisation du protocole de requête étendu.

Opérateurs intégrés

Les opérateurs arithmétiques, de manipulation de bits, de comparaison et logiques listés dans Tableau 289 sont intégrés dans pgbench et peuvent être utilisés dans des expressions apparaissant dans `\set`. Les opérateurs sont listés par priorité croissante. Sauf indication contraire, les opérateurs prenant deux paramètres en entrée produiront un nombre en double précision, si une des entrées est en double précision, sinon le résultat produit sera entier.

Tableau 289. Opérateurs pgbench

Opérateur **Description** **Exemple(s)**
booléen `OR` *booléen* → *booléen* OU logique `5 or 0` → `TRUE`
booléen `AND` *booléen* → *booléen* ET logique `3 and 0` → `FALSE`
`NOT` *booléen* → *booléen* NON logique `not false` → `TRUE`
booléen `IS [NOT] (NULL|TRUE|FALSE)` → *booléen* Tests de valeur booléenne `1 is null` → `FALSE`
valeur `ISNULL|NOTNULL` → *booléen*

Opérateur **Description** **Exemple(s)**
Tests de nullité `1 notnull` → `TRUE`
nombre = *nombre* → *booléen* Est égal `5 = 4` → `FALSE`
nombre <> *nombre* → *booléen* N'est pas égal `5 <> 4` → `TRUE`
nombre != *nombre* → *booléen* N'est pas égal `5 != 5` → `FALSE`
nombre < *nombre* → *booléen* Inférieur à `5 < 4` → `FALSE`
nombre <= *nombre* → *booléen* Inférieur ou égal à `5 <= 4` → `FALSE`
nombre > *nombre* → *booléen* Plus grand que `5 > 4` → `TRUE`
nombre >= *nombre* → *booléen* Plus grand ou égal à `5 >= 4` → `TRUE`
entier | *entier* → *entier* OU binaire `1 | 2` → `3`
entier # *entier* → *entier* XOR binaire `1 # 3` → `2`
entier & *entier* → *entier* ET binaire `1 & 3` → `1`
~ *entier* → *entier* NON binaire `~ 1` → `-2`
entier << *entier* → *entier* Décalage binaire vers la gauche `1 << 2` → `4`
entier >> *entier* → *entier* Décalage binaire vers la droite

Opérateur **Description** **Exemple(s)**
`8 >> 2` → `2`
nombre + *nombre* → *nombre* Addition `5 + 4` → `9`
nombre - *nombre* → *nombre* Soustraction `3 - 2.0` → `1.0`
nombre * *nombre* → *nombre* Multiplication `5 * 4` → `20`
nombre / *nombre* → *nombre* Division (tronque le résultat vers zéro si les deux paramètres d'entrée sont des entiers) `5 / 3` → `1`
entier % *entier* → *entier* Modulo (reste) `3 % 2` → `1`
- *nombre* → *nombre* Opposé `- 2.0` → `-2.0`

Fonctions internes

Les fonctions listées dans Tableau 290 sont internes à pgbench et peuvent être utilisées dans des expressions apparaissant dans `\set`.

Tableau 290. pgbench Functions

Fonction **Description** **Exemple(s)**
`abs` (*nombre*) → même que l'entrée Valeur absolue `abs(-17)` → `17`
`debug` (*nombre*) → même que l'entrée Affiche l'argument dans stderr, puis le retourne. `debug(5432.1)` → `5432.1`
`double` (*nombre*) → `double` Convertit en double précision. `double(5432)` → `5432.0`
`exp` (*nombre*) → `double` Exponentielle (`e` à la puissance indiquée) `exp(1.0)` → `2.718281828459045`
`greatest` (*nombre* [, `...`]) → `double` si un argument est un double, sinon `entier`

Fonction **Description** **Exemple(s)**
Sélectionne la plus grande valeur parmi les arguments. `greatest(5, 4, 3, 2)` → `5`
`hash` (*`valeur`* [, *`graine`*]) → `entier` Alias pour `hash_murmur2`. `hash(10, 5432)` → `-5817877081768721676`
`hash_fnv1a` (*`valeur`* [, *`graine`*]) → `entier` Calcule le hash FNV-1a[1]. `hash_fnv1a(10, 5432)` → `-7793829335365542153`
`hash_murmur2` (*`valeur`* [, *`graine`*]) → `entier` Computes MurmurHash2 hash[2]. `hash_murmur2(10, 5432)` → `-5817877081768721676`
`int` (*`nombre`*) → `entier` Convertit en entier. `int(5.4 + 3.8)` → `9`
`least` (*`nombre`* [, `...`]) → `double` si un argument est un double, sinon `entier` Choisit la plus petite valeur parmi les arguments. `least(5, 4, 3, 2.1)` → `2.1`
`ln` (*`nombre`*) → `double` Logarithme naturel `ln(2.718281828459045)` → `1.0`
`mod` (*`entier`*, *`entier`*) → `entier` Modulo (reste) `mod(54, 32)` → `22`
`permute` (*`i`*, *`taille`* [, *`graine`*]) → `integer` Valeur permutée de *`i`*, dans la tranche `[0, taille)`, soit la nouvelle position de *`i`* (modulo *`taille`*) dans une permutation pseudo-aléatoire des entiers `0...taille-1`, paramétrée par la *`graine`* (voir plus bas). `permute(0, 4)` → `un entier entre 0 and 3`
`pi` () → `double` Valeur approximative de ## `pi()` → `3.14159265358979323846`
`pow` (*`x`*, *`y`*) → `double` `power` (*`x`*, *`y`*) → `double` *`x`* à la puissance *`y`* `pow(2.0, 10)` → `1024.0`
`random` (*`lb`*, *`ub`*) → `entier` Calcule un entier aléatoire uniformément distribué dans `[lb, ub]`. `random(1, 10)` → `un entier entre 1 and 10`

[1] https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function
[2] https://en.wikipedia.org/wiki/MurmurHash

Fonction **Description** **Exemple(s)**
`random_exponential` (*`lb`*, *`ub`*, *`paramètre`*) → `entier` Calcule un entier aléatoire distribué exponentiellement dans `[lb, ub]` ;voir plus bas. `random_exponential(1, 10, 3.0)` → `un entier entre 1 and 10`
`random_gaussian` (*`lb`*, *`ub`*, *`paramètre`*) → `entier` Calcule un entier aléatoire distribué de manière gaussienne dans `[lb, ub]` ;voir plus bas. `random_gaussian(1, 10, 2.5)` → `un entier entre 1 and 10`
`random_zipfian` (*`lb`*, *`ub`*, *`paramètre`*) → `entier` Calcule un entier aléatoire distribué selon la loi de Zipf dans `[lb, ub]` ;voir plus bas. `random_zipfian(1, 10, 1.5)` → `une valeur entre 1 and 10`
`sqrt` (*`nombre`*) → `double` Racine carrée `sqrt(2.0)` → `1.414213562`

La fonction `random` génère des valeurs en utilisant une distribution uniforme ; autrement dit toutes les valeurs sont dans l'intervalle spécifiée avec une probabilité identique. Les fonctions `random_exponential`, `random_gaussian` et `random_zipfian` requièrent un paramètre supplémentaire de type double qui détermine le contour précis de cette distribution.

- Pour une distribution exponentielle, *`paramètre`* contrôle la distribution en tronquant une distribution exponentielle en décroissance rapide à *`paramètre`*, puis en projetant le résultant sur des entiers entre les limites. Pour être précis :

  f(x) = exp(-paramètre * (x - min) / (max - min + 1)) / (1 - exp(-paramètre))

  Puis la valeur *`i`* entre les valeurs *`min`* et *`max`*, en les incluant, est récupérée avec la probabilité : `f(i) - f(i + 1)`.

  Intuitivement, plus *`paramètre`* est grand, plus les valeurs fréquentes proches de *`min`* sont accédées et moins les valeurs fréquentes proches de *`max`* sont accédées. Plus *`paramètre`* est proche de 0, plus la distribution d'accès sera plate (uniforme). Une approximation grossière de la distribution est que 1 % des valeurs les plus fréquentes de l'intervalle, proches de *`min`*, sont tirées *`paramètre`*% du temps. La valeur de *`paramètre`* doit être strictement positive.

- Pour une distribution gaussienne, l'intervalle correspond à une distribution normale standard (la courbe gaussienne classique en forme de cloche) tronquée à `-paramètre` à gauche et à `+paramètre` à droite. Les valeurs au milieu de l'intervalle sont plus susceptibles d'être sélectionnées. Pour être précis, si `PHI(x)` est la fonction de distribution cumulative de la distribution normale standard, avec une moyenne `mu` définie comme `(max + min) / 2.0`, avec

  f(x) = PHI(2.0 * paramètre * (x - mu) / (max - min + 1)) /
  (2.0 * PHI(paramètre) - 1)

  alors la valeur *`i`* entre *`min`* et *`max`* (inclus) est sélectionnée avec une probabilité : `f(i + 0.5) - f(i - 0.5)`. Intuitivement, plus *`paramètre`* est grand, et plus les valeurs fréquentes proches du centre de l'intervalle sont sélectionnées, et moins les valeurs fréquentes proches des bornes *`min`* et *`max`*. Environ 67 % des valeurs sont sélectionnées à partir du centre `1.0 / paramètre`, soit `0.5 / paramètre` autour de la moyenne, et 95 % dans le centre `2.0 / paramètre`,

soit `1.0 / paramètre` autour de la moyenne ; par exemple, si *`paramètre`* vaut 4.0, 67 % des valeurs sont sélectionnées depuis le quart du milieu (1.0 / 4.0) de l'intervalle (ou à partir de `3.0 / 8.0` jusqu'à `5.0 / 8.0`) et 95 % depuis la moitié du milieu (`2.0 / 4.0`) de l'intervalle (deuxième et troisième quartiles). La valeur minimale autorisée pour *`paramètre`* est 2.0.

- `random_zipfian` génère une distribution bornée selon la loi de Zipf. *`paramètre`* définit à quel point la distribution est biaisée. Plus *`paramètre`* est grand, plus fréquemment les valeurs du début de l'intervalle seront tirées. La distribution est telle que, en supposant que l'intervalle commence à 1, le ratio de probabilité d'un jet *`k`* contre un jet *`k+1`* est `((`*`k`*`+1)/`*`k`*`)**`*`parameter`*. Par exemple, `random_zipfian(1, ..., 2.5)` produit la valeur `1` à peu près `(2/1)**2.5 = 5.66` fois plus fréquemment que `2`, qui lui-même est produit `(3/2)**2.5 = 2.76` fois plus fréquemment que `3`, et ainsi de suite.

  L'implémentation de pgbench est basée sur « Non-Uniform Random Variate Generation », Luc Devroye, p. 550-551, Springer 1986. À cause des limitations de cet algorithme, la valeur *`paramètre`* est restreinte à l'intervalle [1.001, 1000].

> **Note**
>
> Lors de la conception d'un benchmark qui sélectionne des lignes de manière non uniforme, soyez conscient que les lignes choisies peuvent être corrélées avec d'autres données, comme les ID d'une séquence ou l'ordre physique des lignes, ce qui peut biaiser les mesures de performance.
>
> Pour éviter cela, pensez à la fonction `permute`, ou toute autre opération avec le même effet, pour mélanger les lignes sélectionnées et détruire ces corrélations.

Les fonctions de hachage `hash`, `hash_murmur2` et `hash_fnv1a` acceptent une valeur d'entrée et une graine optionnelle. Si la graine n'est pas fournie, la valeur de `:default_seed` est utilisée, initialisée de façon aléatoire si elle n'est pas définie par l'option de ligne de commande `-D`..

`permute` accepte en entrée une valeur, une taille, et une graine optionnelle. Elle génère une permutation pseudo-aléatoire des entiers dans la tranche `[0, taille)`, et retourne l'index de la valeur d'entrée dans les valeurs permutées. La permutation choisie est paramétrée par la graine, soit par défaut `:default_seed` si elle n'est pas fournie. Au contraire des fonctions de hachage, `permute` garantit qu'il n'y aura ni collision ni trou dans les valeurs retournées. Les valeurs d'entrée hors de l'intervalle sont interprétées modulo la taille. La fonction lève une erreur si la taille n'est pas positive. `permute` peut être utilisée pour disperser la distribution de fonctions aléatoires non uniformes comme `random_zipfian` ou `random_exponential`, afin que les valeurs les plus couramment tirées ne soient pas corrélées de manière triviale. Par exemple, le script pgbench suivant simule une charge réaliste typique des médias sociaux et des plateformes de blogs, où quelques comptes génèrent une charge excessive :

```
\set size 1000000
\set r random_zipfian(1, :size, 1.07)
\set k 1 + permute(:r, :size)
```

Dans certains cas, plusieurs distributions distinctes non corrélées entre elles sont nécessaires, et c'est là que le paramètre graine optionnel est pratique :

```
\set k1 1 + permute(:r, :size, :default_seed + 123)
\set k2 1 + permute(:r, :size, :default_seed + 321)
```

Un comportement similaire peut être approché avec `hash` :

```
\set size 1000000
\set r random_zipfian(1, 100 * :size, 1.07)
\set k 1 + abs(hash(:r)) % :size
```

Cependant, comme `hash` génère des collisions, certaines valeurs ne sont pas atteignables, et d'autres seront plus fréquentes qu'attendues par rapport à la distribution originale.

À titre d'exemple, la définition complète de la transaction style TPC-B intégrée est :

```
   \set aid random(1, 100000 * :scale)
   \set bid random(1, 1 * :scale)
   \set tid random(1, 10 * :scale)
   \set delta random(-5000, 5000)
   BEGIN;
   UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE
 aid = :aid;
   SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
   UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE
 tid = :tid;
   UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE
 bid = :bid;
   INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
 (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
   END;
```

Ce script autorise chaque itération de la transaction à référencer des lignes différentes, sélectionnées aléatoirement. (Cet exemple montre aussi pourquoi il est important que chaque session cliente ait ses propres variables -- sinon elles n'affecteront pas les différentes lignes de façon indépendante.

Journaux par transaction

Avec l'option `-l` (mais sans l'option `--aggregate-interval`), pgbench va écrire des informations sur chaque transaction dans un fichier journal. Il sera nommé *`prefix`*`.`*`nnn`*, où *`prefix`* vaut par défaut `pgbench_log`, et *`nnn`* est le PID du processus pgbench. Le préfixe peut être changé avec l'option `--log-prefix`. Si l'option `-j` est positionnée à 2 ou plus, créant plusieurs processus de travail (*worker*), chacun aura son propre fichier journal. Le premier worker utilisera le même nom pour son fichier journal que dans le cas d'un seul processus. Les fichiers journaux supplémentaires s'appelleront *`prefix`*`.`*`nnn`*`.`*`mmm`*, où *`mmm`* est un numéro de séquence, identifiant chaque worker, commençant à 1.

Dans un fichier journal, chaque ligne décrit une transaction. Elle contient les champs suivants, séparés par des espaces :

`client_id`

identifie la session client qui a exécuté la transaction

`transaction_no`

décompte les transactions exécutées par cette session

`time`

temps déroulé pour la transaction, en microsecondes

`script_no`

identifie le fichier script utilisé par la transaction (utile quand plusieurs scripts ont été fournis avec `-f` ou `-b`)

time_epoch

horodatage de la fin de la transaction (époque Unix)

time_us

fraction de seconde de la durée de la transaction, en microsecondes

schedule_lag

délai au démarrage de de la transaction, c'est-à-dire la différence entre le moment prévu du démarrage et le moment où elle a réellement démarré, en microsecondes (n'est présent que si `--rate` est précisé)

retries

nombre des nouvelles tentatives après des erreurs de sérialisation et des deadlocks pendant la transaction (n'est présent que si `--max-tries` n'est pas à 1)

Quand les options `--rate` et `--latency-limit` sont utilisées en même temps, le champ *time* pour une transaction ignorée sera rapporté en tant que `skipped`. Si la transaction se termine par une erreur, son *time* sera rapporté comme `failed`. Avec l'option `--failures-detailed`, le *time* de la transaction échouée sera attribué à la `serialization` ou à un `deadlock` en fonction du type d'erreur (voir Failures and Serialization/Deadlock Retries pour plus d'information).

Ci-dessous un extrait du fichier journal généré avec un seul client :

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

Autre exemple avec les options `--rate=100` et `--latency-limit=5` (notez la colonne supplémentaire *schedule_lag*) :

```
0 81 4621 0 1412881037 912698 3005
0 82 6173 0 1412881037 914578 4304
0 83 skipped 0 1412881037 914578 5217
0 83 skipped 0 1412881037 914578 5099
0 83 4722 0 1412881037 916203 3108
0 84 4142 0 1412881037 918023 2333
0 85 2465 0 1412881037 919759 740
```

Dans cet exemple, la transaction 82 a été en retard, elle affiche une latence (6,173 ms) supérieure à la limite de 5 ms. Les deux transactions suivantes ont été ignorées, car elles avaient déjà en retard avant même d'avoir commencé.

L'exemple suivant montre un extrait d'un journal avec des erreurs et de nouvelles tentatives, avec un nombre maximum d'essais à 10 (notez la colonne supplémentaire *retries* :

```
 3 0 47423 0 1499414498 34501 3
 3 1 8333 0 1499414498 42848 0
 3 2 8358 0 1499414498 51219 0
 4 0 72345 0 1499414498 59433 6
 1 3 41718 0 1499414498 67879 4
 1 4 8416 0 1499414498 76311 0
```

```
3 3 33235 0 1499414498 84469 3
0 0 failed 0 1499414498 84905 9
2 0 failed 0 1499414498 86248 9
3 4 8307 0 1499414498 92788 0
```

Avec l'option `--failures-detailed`, le type d'erreur est rapporté dans le *`time`* ainsi :

```
3 0 47423 0 1499414498 34501 3
3 1 8333 0 1499414498 42848 0
3 2 8358 0 1499414498 51219 0
4 0 72345 0 1499414498 59433 6
1 3 41718 0 1499414498 67879 4
1 4 8416 0 1499414498 76311 0
3 3 33235 0 1499414498 84469 3
0 0 serialization 0 1499414498 84905 9
2 0 serialization 0 1499414498 86248 9
3 4 8307 0 1499414498 92788 0
```

Dans le cas d'un test long sur du matériel qui peut supporter un grand nombre de transactions, les journaux peuvent devenir très volumineux. L'option `--sampling-rate` peut être utilisée pour journaliser seulement un extrait aléatoire des transactions effectuées.

Agrégation de la journalisation

Avec l'option `--aggregate-interval`, les fichiers journaux utilisent un format quelque peu différent. Chaque ligne décrit un intervalle d'agrégation. Elle contient les champs suivants séparés par des espaces :

`interval_start`

horodatage du démarrage de l'intervalle (époque Unix)

`num_transactions`

nombre de transactions au sein de l'intervalle

`sum_latency`

somme des latences de transaction

`sum_latency_2`

somme des carrés des latences de transaction

`min_latency`

minimum des latences de transaction

`max_latency`

maximum des latences de transaction

`sum_lag`

somme des délais de démarrage des transactions (zéro, à moins que `--rate` soit renseigné)

`sum_lag_2`

somme des carrés des délais de démarrage des transactions (zéro, à moins que `--rate` soit renseigné)

min_lag

minimum des délais de démarrage des transactions (zéro, à moins que `--rate` soit renseigné)

max_lag

maximum des délais de démarrage des transactions (zéro, à moins que `--rate` soit renseigné)

skipped

nombre de transactions passées car elles auraient démarré trop tard (zéro, à moins que `--rate` et `--latency-limit` soient renseignés)

retried

nombre de transactions retentées (zéro, à moins que `--max-tries` soit différent de 1)

retries

nombre de transactions retentées après des erreurs de sérialisation ou des deadlocks (zéro, à moins que `--max-tries` soit différent de 1)

serialization_failures

nombre de transactions ayant obtenu une erreur de sérialisation et non retentées ensuite (zéro, à moins que `--failures-detailed` soit précisé)

deadlock_failures

nombre de transactions ayant obtenu une erreur à cause d'un deadlock et non retentées ensuite (zéro, à moins que `--failures-detailed` soit précisé)

Voici un exemple de sortie générée avec ces options :

```
pgbench --aggregate-interval=10 --time=20 --client=10 --log --
rate=1000 --latency-limit=10 --failures-detailed --max-tries=10
 test

1650260552 5178 26171317 177284491527 1136 44462 2647617 7321113867
 0 9866 64 7564 28340 4148 0
1650260562 4808 25573984 220121792172 1171 62083 3037380 9666800914
 0 9998 598 7392 26621 4527 0
```

Notez que si le journal proprement dit (non agrégé) montre le script utilisé pour chaque transaction, le format agrégé ne l'affiche pas. De ce fait, si vous avez besoin des valeurs pour chaque script, vous devrez agréger ces données vous-même.

Rapport par requête

Avec l'option `-r`, pgbench collecte les statistiques suivantes pour chaque requête :

- `latency` -- partie du temps de transaction pour chaque requête. pgbench renvoie une valeur moyenne de toutes les exécutions de l'ordre.
- Le nombre d'échecs de la requête. Voir Failures and Serialization/Deadlock Retries pour plus d'informations.
- Le nombre de nouvelles tentatives après une erreur de sérialisation ou un deadlock dans cette requête. Voir Failures and Serialization/Deadlock Retries pour plus d'informations.

Le rapport n'affiche les statistiques des nouvelles tentatives que si l'option `--max-tries` ne vaut pas 1.

Toutes les valeurs sont calculées pour chaque requête exécutée par chaque client et sont affichées après la fin du benchmark.

Pour le script par défaut, le résultat aura la forme suivante :

```
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
 scaling factor: 1
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
maximum number of tries: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
number of failed transactions: 0 (0.000%)
number of transactions above the 50.0 ms latency limit: 1311/10000
 (13.110 %)
latency average = 28.488 ms
latency stddev = 21.009 ms
initial connection time = 69.068 ms
tps = 346.224794 (without initial connection time)
statement latencies in milliseconds and failures:
   0.012  0  \set aid random(1, 100000 * :scale)
   0.002  0  \set bid random(1, 1 * :scale)
   0.002  0  \set tid random(1, 10 * :scale)
   0.002  0  \set delta random(-5000, 5000)
   0.319  0  BEGIN;
   0.834  0  UPDATE pgbench_accounts SET abalance = abalance
 + :delta WHERE aid = :aid;
   0.641  0  SELECT abalance FROM pgbench_accounts WHERE aid
 = :aid;
  11.126  0  UPDATE pgbench_tellers SET tbalance = tbalance
 + :delta WHERE tid = :tid;
  12.961  0  UPDATE pgbench_branches SET bbalance = bbalance
 + :delta WHERE bid = :bid;
   0.634  0  INSERT INTO pgbench_history (tid, bid, aid, delta,
 mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
   1.957  0  END;
```

Autre exemple de sortie pour le script par défaut, en utilisant le mode de transaction sérialisable (`PGOPTIONS='-c default_transaction_isolation=serializable' pgbench ...`):

```
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
maximum number of tries: 10
number of transactions per client: 1000
number of transactions actually processed: 6317/10000
```

```
number of failed transactions: 3683 (36.830%)
number of transactions retried: 7667 (76.670%)
total number of retries: 45339
number of transactions above the 50.0 ms latency limit: 106/6317
 (1.678 %)
latency average = 17.016 ms
latency stddev = 13.283 ms
initial connection time = 45.017 ms
tps = 186.792667 (without initial connection time)
statement latencies in milliseconds, failures and retries:
  0.006     0      0  \set aid random(1, 100000 * :scale)
  0.001     0      0  \set bid random(1, 1 * :scale)
  0.001     0      0  \set tid random(1, 10 * :scale)
  0.001     0      0  \set delta random(-5000, 5000)
  0.385     0      0  BEGIN;
  0.773     0      1  UPDATE pgbench_accounts SET abalance =
 abalance + :delta WHERE aid = :aid;
  0.624     0      0  SELECT abalance FROM pgbench_accounts WHERE
 aid = :aid;
  1.098   320   3762  UPDATE pgbench_tellers SET tbalance =
 tbalance + :delta WHERE tid = :tid;
  0.582  3363  41576  UPDATE pgbench_branches SET bbalance =
 bbalance + :delta WHERE bid = :bid;
  0.465     0      0  INSERT INTO pgbench_history (tid,
 bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
 CURRENT_TIMESTAMP);
  1.933     0      0  END;
```

Toutes les statistiques sont rapportées séparément si plusieurs scripts sont spécifiés.

Notez que la collecte des informations de chronométrage supplémentaires nécessaires pour calculer la latence par requête ajoute une certaine charge. Cela va réduire la vitesse moyenne calculée pour l'exécution des transactions et réduire le débit de transactions calculé. Le ralentissement varie de manière significative selon la plateforme et le matériel. Comparer la moyenne des valeurs de TPS avec et sans intégration de la latence est une bonne manière de voir si la surcharge induite par le chronométrage est importante ou pas.

Nouvelles tentatives suite à des erreurs de sérialisation ou des deadlocks

À l'exécution de pgbench, il y a trois principaux types d'erreurs, qui comprennent :

- Les erreurs du programme principal. Ce sont les plus sérieuses, et elles entraînent toujours une sortie immédiate de pgbench avec le message d'erreur correspondant. Elles incluent :
  - les erreurs au lancement de pgbench (par exemple une valeur d'option invalide) ;
  - les erreurs dans la phase d'initialisation (par exemple quand échoue la requête de création des tables pour les scripts inclus)
  - les erreurs avant le démarrage des threads (par exemple l'impossibilité à se connecter au serveur de bases de données, une erreur de syntaxe dans la métacommande, une erreur à la création du thread) ;
  - les erreurs internes de pgbench (supposées ne jamais se produire...).
- Les erreurs quand le thread gère ses clients (par exemple si le client ne peut démarrer une connexion au serveur / si la socket pour connecter le client au serveur est devenu invalide). Dans ces cas, tous les clients du thread s'arrêtent pendant que les autres continuent de fonctionner.

- Les erreurs directes du client. Elles mènent à un arrêt immédiat de pgbench, avec le message d'erreur correspondant, uniquement dans le cas d'une erreur interne de pgbench (supposées ne jamais se produire...). Sinon, dans le pire des cas, ces erreurs n'entraînent que l'arrêt du client en échec, pendant que les autres continuent leur travail (mais certaines erreurs des clients sont gérées sans arrêter le client et sont rapportées séparément, voir plus bas). Dans la suite de cette section, on suppose que les erreurs évoquées ne sont que des erreurs du client et non des erreurs internes de pgbench.

L'exécution d'un client n'est interrompue que dans le cas d'une erreur sérieuse : par exemple, la connexion au serveur a été perdue, ou la fin du script a été atteinte sans avoir terminé la dernière transaction. De plus, si l'exécution d'une commande SQL ou d'une métacommande échoue, le client est arrêté, sauf pour une erreur de sérialisation ou un deadlock. Dans ce dernier cas, la transaction courante est annulée, ce qui comprend la restauration des variables à leurs valeurs d'avant l'exécution de la transaction (on suppose qu'un script de transaction ne contient qu'une transaction ; voir What Is the "Transaction" Actually Performed in pgbench? pour plus d'informations). Les transactions avec des erreurs de sérialisation ou des deadlocks sont répétées après le rollback, jusqu'à ce qu'elles se terminent avec succès, ou qu'elles atteignent le nombre maximum de tentatives (indiqué avec `--max-tries`) / le délai maximal pour les tentatives (indiqué avec `--latency-limit`) / la fin du benchmark (indiqué avec `--time`). Si la dernière tentative d'exécution échoue, la transaction sera comptabilisée comme échouée, mais le client n'est pas arr#té et continue de fonctionner.

> **Note**
>
> Sans définition de `--max-tries`, une transaction ne sera jamais retentée après une erreur de sérialisation ou un deadlock, car la valeur par défaut de l'option est 1. Pour limiter uniquement la durée totale des tentatives, utilisez l'option `--latency-limit` en indiquant un nombre illimité de tentatives (`--max-tries=0`). Vous pouvez aussi utiliser `--time` pour limiter la durée du benchmark alors que le nombre de tentatives est illimité.
>
> Faites attention lors de la répétition de scripts avec de multiples transactions : le script est toujours intégralement réexécuté, et les transactions réussies peuvent donc être exécutées plusieurs fois.
>
> Faites aussi attention lors de la répétition de scripts avec des commandes shell. Leur résultat n'est pas annulé, contrairement à ce qui passe avec des commandes SQL, à part pour une valeur assignée avec la commande `\setshell`.

La latence d'une transaction réussie inclut la durée entière de l'exécution, y compris les rollbacks et les diverses tentatives. La latence n'est mesurée que pour les transactions et commandes réussies, mais pas pour celles en échec.

Le rapport principal contient le nombre de transactions échouées. Si `--max-tries` n'est pas égal à 1, il contiendra aussi des statistiques sur les tentatives : le nombre total de transactions tentées plusieurs fois, et le nombre total de nouvelles tentatives. Le rapport par script hérite de tous ces champs. Le rapport par requête n'affiche les statistiques sur les tentatives que si `--max-tries` n'est pas égal à 1.

Si vous voulez regrouper les échecs par types dans les journaux par transaction et les journaux agrégés, utilisez l'option `--failures-detailed`. Si vous voulez aussi distinguer toutes les erreurs et les échecs (c'est-à-dire les erreurs sans nouvelle tentative), en incluant quelles limites sur les tentatives ont été dépassées, et de combien, pour les échecs de sérialisation ou sur deadlock, utilisez `--verbose-errors`.

Bonnes pratiques

Il est facile d'utiliser pgbench pour produire des résultats complètement dénués de sens ! Voici quelques conseils pour vous aider à obtenir des résultats pertinents.

Tout d'abord, ne croyez *jamais* en un test qui ne dure que quelques secondes. Utilisez l'option `-t` ou `-T` pour que le test dure au moins quelques minutes, de façon à lisser le bruit. Dans certains cas, il vous

faudra des heures pour récupérer des valeurs reproductibles. C'est une bonne idée de lancer plusieurs fois le test pour voir si vos chiffres sont ou pas reproductibles.

Pour le scénario de test par défaut typé TPC-B, le facteur d'échelle d'initialisation (`-s`) doit être au moins aussi grand que le nombre maximum de clients que vous avez l'intention de tester (`-c`) ; sinon vous testez surtout la contention induite par les mises à jour. Il n'y a que `-s` lignes dans la table `pgbench_branches`, et chaque transaction veut mettre à jour l'une de ces lignes, donc si la valeur de `-c` est supérieure à la valeur de `-s`, il en résultera sûrement de nombreuses transactions bloquées en attente de la fin d'autres transactions.

Le scénario par défaut est aussi assez sensible au temps écoulé depuis l'initialisation des tables : l'accumulation des lignes et espaces morts dans les tables change les résultats. Pour comprendre ces résultats, vous devez garder une trace du nombre total de mises à jour et du moment du vacuum. Si l'autovacuum est actif, il peut en résulter des variations imprévisibles dans les performances mesurées.

Une limitation de pgbench est qu'il peut lui-même devenir le goulet d'étranglement lorsqu'on essaie de tester avec un grand nombre de sessions clientes. Cela peut être atténué en utilisant pgbench depuis une machine différente du serveur de base de données, bien qu'une faible latence sur le réseau soit dans ce cas essentielle. Il peut même être utile de lancer plusieurs instances parallèles de pgbench, depuis plusieurs machines clientes vers le même serveur de base de données.

Securité

Si des utilisateurs non dignes de confiance ont accès à une base de données qui n'a pas adopté une méthode sécurisée d'utilisation des schémas, il ne faut pas exécuter pgbench dans cette base. pgbench utilise des noms non qualifiés et ne modifie pas le chemin de recherche.

pg_config

pg_config — récupèrer des informations sur la version installée de PostgreSQL

Synopsis

`pg_config` [*option*...]

Description

L'outil pg_config affiche les paramètres de configuration de la version installée de PostgreSQL. Il peut, par exemple, d'être utilisé par des paquets logiciels qui souhaitent s'interfacer avec PostgreSQL pour faciliter la recherche des fichiers d'entêtes requis et des bibliothèques.

Options

Pour utiliser pg_config, une ou plusieurs des options suivantes doivent être fournies :

`--bindir`

Afficher l'emplacement des exécutables utilisateur. Par exemple, pour trouver le programme `psql`. C'est aussi normalement l'emplacement du programme `pg_config`.

`--docdir`

Afficher l'emplacement des fichiers de documentation.

`--htmldir`

Affiche l'emplacement des fichiers de documentation HTML.

`--includedir`

Afficher l'emplacement des fichiers d'entêtes C des interfaces clientes.

`--pkgincludedir`

Afficher l'emplacement des autres fichiers d'entête C.

`--includedir-server`

Afficher l'emplacement des fichiers d'entêtes C pour la programmation du serveur.

`--libdir`

Afficher l'emplacement des bibliothèques.

`--pkglibdir`

Afficher l'emplacement des modules chargeables dynamiquement ou celui que le serveur peut parcourir pour les trouver. (D'autres fichiers de données dépendant de l'architecture peuvent aussi être installés dans ce répertoire.)

`--localedir`

Afficher l'emplacement des fichiers de support de la locale (c'est une chaîne vide si le support de la locale n'a pas été configuré lors de la construction de PostgreSQL).

`--mandir`

Afficher l'emplacement des pages de manuel.

`--sharedir`

Afficher l'emplacement des fichiers de support qui ne dépendent pas de l'architecture.

`--sysconfdir`

Afficher l'emplacement des fichiers de configuration du système.

`--pgxs`

Afficher l'emplacement des fichiers makefile d'extensions.

`--configure`

Afficher les options passées au script `configure` lors de la configuration de PostgreSQL en vue de sa construction. Cela peut être utilisé pour reproduire une configuration identique ou pour trouver les options avec lesquelles un paquet binaire a été construit. (Néanmoins, les paquets binaires contiennent souvent des correctifs personnalisés par le vendeur.) Voir aussi les exemples ci-dessous.

`--cc`

Afficher la valeur de la macro `CC` utilisée lors de la construction de PostgreSQL. Cela affiche le compilateur C utilisé.

`--cppflags`

Afficher la valeur de la macro `CPPFLAGS` utilisée lors de la construction de PostgreSQL. Cela affiche les options du compilateur C nécessaires pour l'exécution du préprocesseur (typiquement, les options `-I`).

`--cflags`

Afficher la valeur de la macro `CFLAGS` utilisée lors de la construction de PostgreSQL. Cela affiche les options du compilateur C.

`--cflags_sl`

Afficher la valeur de la macro `CFLAGS_SL` utilisée lors de la construction de PostgreSQL. Cela affiche les options supplémentaires du compilateur C utilisées pour construire les bibliothèques partagées.

`--ldflags`

Afficher la valeur de la macro `LDFLAGS` utilisée lors de la construction de PostgreSQL. Cela affiche les options de l'éditeur de liens.

`--ldflags_ex`

Afficher la valeur de la variable `LDFLAGS_EX` utilisée lors de la construction de PostgreSQL. Cela affiche les options de l'éditeur de liens uniquement pour la construction des exécutables.

`--ldflags_sl`

Afficher la valeur de la macro `LDFLAGS_SL` utilisée lors de la construction de PostgreSQL. Cela affiche les options de l'éditeur de liens utilisées pour construire seulement les bibliothèques partagées.

`--libs`

Afficher la valeur de la macro `LIBS` utilisée lors de la construction de PostgreSQL. Elle contient habituellement les options `-l` pour les bibliothèques externes auxquelles PostgreSQL est lié.

`--version`

Afficher la version de PostgreSQL.

`-?`
`--help`

Affiche de l'aide à propos des arguments en ligne de commande avec pg_config, puis quitte.

Si plusieurs options sont données, l'information est affichée dans cet ordre, un élément par ligne. Si aucune option n'est donnée, toutes les informations disponibles sont affichées avec des étiquettes.

Notes

Les options `--docdir`, `--pkgincludedir`, `--localedir`, `--mandir`, `--sharedir`, `--sysconfdir`, `--cc`, `--cppflags`, `--cflags`, `--cflags_sl`, `--ldflags`, `--ldflags_sl` et `--libs` sont apparues avec PostgreSQL 8.1. L'option `--htmldir` n'est disponible qu'à partir de PostgreSQL 8.4. The option `--ldflags_ex` was added in PostgreSQL 9.0.

Exemple

Reproduire la configuration de construction de l'installation actuelle de PostgreSQL :

```
eval ./configure `pg_config --configure`
```

La sortie de `pg_config --configure` contient les guillemets du shell de sorte que les arguments contenant des espaces soient représentés correctement. Du coup, il est nécessaire d'utiliser `eval` pour obtenir des résultats corrects.

pg_dump

pg_dump — sauvegarder une base de données PostgreSQL dans un script ou tout autre fichier d'archive

Synopsis

pg_dump [*option_connexion*...] [*option*...] [*nom_base*]

Description

pg_dump est un outil de sauvegarde d'une base de données PostgreSQL. Les sauvegardes réalisées sont cohérentes, même lors d'accès concurrents à la base de données. pg_dump ne bloque pas l'accès des autres utilisateurs (ni en lecture ni en écriture).

pg_dump sauvegarde seulement une base de données. Pour sauvegarder une instance complète ou pour sauvegarder les objets globaux communs à toutes les bases de données d'une même instance, tels que les rôles et les tablespaces, utilisez pg_dumpall.

Les extractions peuvent être réalisées sous la forme de scripts ou de fichiers d'archive. Les scripts sont au format texte et contiennent les commandes SQL nécessaires à la reconstruction de la base de données dans l'état où elle était au moment de la sauvegarde. La restauration s'effectue en chargeant ces scripts avec psql. Ces scripts permettent de reconstruire la base de données sur d'autres machines et d'autres architectures, et même, au prix de quelques modifications, sur d'autres bases de données SQL.

La reconstruction de la base de données à partir d'autres formats de fichiers archive est obtenue avec pg_restore. pg_restore permet, à partir de ces formats, de sélectionner les éléments à restaurer, voire de les réordonner avant restauration. Les fichiers d'archive sont conçus pour être portables au travers d'architectures différentes.

Utilisé avec un des formats de fichier d'archive et combiné avec pg_restore, pg_dump fournit un mécanisme d'archivage et de transfert flexible. pg_dump peut être utilisé pour sauvegarder une base de données dans son intégralité ; pg_restore peut alors être utilisé pour examiner l'archive et/ou sélectionner les parties de la base de données à restaurer. Les formats de fichier en sortie les plus flexibles sont le format « custom » (`-Fc`) et le format « directory » (`-Fd`). Ils permettent la sélection et le ré-ordonnancement de tous les éléments archivés, le support de la restauration en parallèle. De plus, ils sont compressés par défaut. Le format « directory » est aussi le seul format à permettre les sauvegardes parallélisées.

Lors de l'exécution de pg_dump, il est utile de surveiller les messages d'avertissement (affichés sur la sortie erreur standard), en particulier en ce qui concerne les limitations indiquées ci-dessous.

> **Avertissement**
>
> Restaurer une sauvegarde fait que la destination exécute du code arbitraire. Les sauvegardes et restaurations partielles ne limitent pas cela. S'il n'est pas possible de faire confiance aux superutilisateurs de la source, les requêtes SQL de restauration doivent être inspectées avant d'être exécutées. Les sauvegardes binaires peuvent être inspectées en utilisant l'option `--file` de pg_restore. Notez que le client exécutant la sauvegarde et la restauration n'a pas besoin de faire confiance aux superutilisateurs source et destination.

Options

Les options suivantes de la ligne de commande contrôlent le contenu et le format de la sortie.

nom_base

Le nom de la base de données à sauvegarder. En l'absence de précision, la variable d'environnement PGDATABASE est utilisée. Si cette variable n'est pas positionnée, le nom de l'utilisateur de la connexion est utilisé.

-a
--data-only

Seules les données sont sauvegardées, pas le schéma (définition des données). Les données des tables, les Large Objects, et les valeurs des séquences sont sauvegardées.

Cette option est similaire à --section=data mais, pour des raisons historiques, elle n'est pas identique.

-b
--blobs

Inclut les objets larges dans la sauvegarde. C'est le comportement par défaut, sauf si une des options suivantes est ajoutée : --schema, --table ou --schema-only. L'option -b n'est de ce fait utile que pour ajouter des Large Objects aux sauvegardes pour lesquelles un schéma particulier ou une table particulière a été demandée. Notez que les Large Objects sont considérés comme des données et, de ce fait, seront inclus si --data-only est utilisé, mais pas quand --schema-only l'est.

-B
--no-blobs

Exclure les objets larges de la sauvegarde.

Quand à la fois les options -b et -B sont fournies, le comportement est de produire les objets larges, quand les données sont sauvegardées, voir la documentation de -b.

-c
--clean

Les commandes de suppression (DROP) des objets de la base sont écrites avant les commandes de création. Cette option est utile quand la restauration écrase une base existante. Si un des objets n'existe pas dans la base de destination, des messages d'erreur, à ignorer, seront renvoyées lors de la restauration sauf si l'option --if-exists est aussi précisée.

Cette option n'a d'intérêt que pour le format texte. Pour les formats archive, l'option est précisée à l'appel de pg_restore.

-C
--create

La sortie débute par une commande de création de la base de données et de connexion à cette base. Peu importe, dans ce cas, la base de données de connexion à la restauration. De plus, si --clean est aussi spécifié, le script supprime puis crée de nouveau la base de données cible avant de s'y connecter.

Avec --create, la sortie inclut aussi le commentaire de la base de données, s'il a été configuré, et toute variable de configuration spécifique à cette base de données, configurée via les commandes ALTER DATABASE ... SET ... et ALTER ROLE ... IN DATABASE ... SET ... mentionnant cette base. Les droits d'accès à la base elle-même sont aussi sauvegardés, sauf si --no-acl est indiqué.

Cette option n'a d'intérêt que pour le format texte. Pour les formats archive, l'option est précisée à l'appel de pg_restore.

`-f *fichier*`
`--file=*fichier*`

La sortie est redirigée vers le fichier indiqué. Ce paramètre peut être omis pour les sorties en mode fichier, dans ce cas la sortie standard sera utilisée. Par contre, il doit être fourni pour le format 'directory' (répertoire), où il spécifie le répertoire cible plutôt qu'un fichier. Dans ce cas, le répertoire est créé par `pg_dump` et ne doit pas exister auparavant.

`-e *motif*`
`--extension=*motif*`

Ne sauvegarde que les extensions correspondant au *`motif`* indiqué. Sans cette option, seront sauvées toutes les extensions non système de la base sauvegardée. Plusieurs extensions peuvent être sélectionnées à l'aide de plusieurs paramètres `-e`. Le *`motif`* est interprété suivant les mêmes règles que la commande `\d` de psql (voir Motifs) ; ainsi plusieurs extensions peuvent être sélectionnées à l'aide de jokers dans le motif. Dans ce cas, prenez soin d'encadrer le motif de guillemets simples au besoin, pour empêcher le shell de les interpréter.

Toute configuration de table enregistrée dans `pg_extension_config_dump` est incluse dans la sauvegarde si l'extension est spécifiée dans `--extension`.

> **Note**
>
> Quand `-e` est spécifié, pg_dump ne cherche pas à sauvegarder tout autre objet dont l'extension pourrait dépendre. En conséquence, il n'y a pas de garantie que le résultat d'une sauvegarde en précisant les extensions puisse être restauré dans une base propre.

`-E *codage*`
`--encoding=*codage*`

La sauvegarde est créée dans l'encodage indiqué. Par défaut, la sauvegarde utilise celui de la base de données. Le même résultat peut être obtenu en positionnant la variable d'environnement `PGCLIENTENCODING` avec le codage désiré pour la sauvegarde. The supported encodings are described in Section 24.3.1.

`-F *format*`
`--format=*format*`

Le format de la sortie. *`format`* correspond à un des éléments suivants :

`p`

Fichier de scripts SQL en texte simple (défaut).

`c`

archive personnalisée utilisable par pg_restore. Avec le format de sortie répertoire, c'est le format le plus souple, car il permet la sélection manuelle et le réordonnancement des objets archivés au moment de la restauration. Ce format est aussi compressé par défaut.

`d`
`directory`

Produire une archive au format répertoire utilisable en entrée de pg_restore. Cela créera un répertoire avec un fichier pour chaque table et blob exporté, ainsi qu'un fichier appelé Table of Contents (Table des matières) décrivant les objets exportés dans un format machine

que pg_restore peut lire. Une archive au format répertoire peut être manipulée avec des outils Unix standard; par exemple, les fichiers d'une archive non-compressée peuvent être compressés avec l'outil gzip. Ce format est compressé par défaut et supporte les sauvegardes parallélisées.

`t`

Archive `tar` utilisable par pg_restore. Le format tar est compatible avec le format répertoire; l'extraction d'une archive au format tar produit une archive au format répertoire valide. Toutefois, le format tar ne supporte pas la compression. Par ailleurs, lors de l'utilisation du format tar, l'ordre de restauration des données des tables ne peut pas être changé au moment de la restauration.

`-j` *`njobs`*
`--jobs=`*`njobs`*

Exécute une sauvegarde parallélisée en sauvegardant *`njobs`* tables simultanément. Cette option réduit la durée de la sauvegarde mais elle augmente aussi la charge sur le serveur de base de données. Vous ne pouvez utiliser cette option qu'avec le format de sortie répertoire car c'est le seul format où plusieurs processus peuvent écrire leur données en même temps.

pg_dump ouvrira *`njobs`* + 1 connexions à la base de données. Assurez-vous donc que la valeur de max_connections est configurée suffisamment haut pour permettre autant de connexions.

Réclamer des verrous exclusifs sur les objets de la base lors de l'exécution d'une sauvegarde parallélisée peut causer l'échec de la sauvegarde. La raison en est que le processus principal de pg_dump réclame des verrous partagés (ACCESS SHARE) sur les objets que les processus fils vont sauvegarder plus tard pour s'assurer que personne ne les supprime pendant la sauvegarde. Si un autre client demande alors un verrou exclusif sur une table, ce verrou ne sera pas accepté mais mis en queue, en attente du relâchement du verrou partagé par le processus principal. En conséquence, tout autre accès à la table ne sera pas non plus accepté. Il sera lui-aussi mis en queue, après la demande de verrou exclusif. Cela inclut le processus fils essayant de sauvegarder la table. Sans autre précaution, cela résulterait en un classique « deadlock ». Pour détecter ce conflit, le processus fils pg_dump réclame un nouveau verrou partagé en utilisant l'option `NOWAIT`. Si le processus fils n'obtient pas ce verrou, quelqu'un d'autre doit avoir demandé un verrou exclusif entre temps, et il n'existe donc aucun moyen de continuer la sauvegarde. pg_dump n'a d'autre choix que d'annuler la sauvegarde.

Pour réaliser une sauvegarde parallélisée, le serveur de la base de données a besoin de supporter les images (« snapshots ») synchronisées. Cette fonctionnalité a été introduite avec PostgreSQL version 9.2 pour les serveurs primaires et version 10 pour les serveurs secondaires. Avec cette fonctionnalité, les clients de la base de données peuvent s'assurer de voir le même ensemble de données, même s'ils utilisent des connexions différentes. `pg_dump -j` utilise plusieurs connexions à la base de données ; il se connecte une première fois en tant que processus principal et une fois encore par processus fils. Sans la fonctionnalité d'images synchronisées, les différent processus ne pourraient pas garantir de voir les mêmes données sur chaque connexion, ce qui aurait pour résultat une sauvegarde incohérente.

`-n` *`motif`*
`--schema=`*`motif`*

Sauvegarde uniquement les schémas correspondant à *`schema`* ; la sélection se fait à la fois sur le schéma et sur les objets qu'il contient. Quand cette option n'est pas indiquée, tous les schémas non système de la base cible sont sauvegardés. Plusieurs schémas peuvent être indiqués en utilisant plusieurs fois l'option `-n`. De plus, le paramètre *`motif`* est interprété comme un modèle selon les règles utilisées par les commandes `\d` de psql (voir Motifs). Du coup, plusieurs schémas peuvent être sélectionnés en utilisant des caractères joker dans le modèle. Lors de l'utilisation de ces caractères, il faut faire attention à placer le modèle entre guillemets, si nécessaire, pour empêcher le shell de remplacer les jokers ; voir Exemples.

> **Note**
>
> Quand `-n` est indiqué, pg_dump ne sauvegarde aucun autre objet de la base que ceux dont les schémas sélectionnés dépendent. Du coup, il n'est pas garanti que la sauvegarde d'un schéma puisse être restaurée avec succès dans une base vide.

> **Note**
>
> Les objets qui ne font pas partie du schéma comme les objets larges ne sont pas sauvegardés quand `-n` est précisé. Ils peuvent être rajoutés avec l'option `--blobs`.

`-N` *`motif`*
`--exclude-schema=`*`motif`*

Ne sauvegarde pas les schémas correspondant au modèle *`motif`*. Le modèle est interprété selon les même règles que `-n`. `-N` peut aussi être indiqué plus d'une fois pour exclure des schémas correspondant à des modèles différents.

Quand les options `-n` et `-N` sont indiquées, seuls sont sauvegardés les schémas qui correspondent à au moins une option `-n` et à aucune option `-N`. Si `-N` apparaît sans `-n`, alors les schémas correspondant à `-N` sont exclus de ce qui est une sauvegarde normale.

`-O`
`--no-owner`

Les commandes d'initialisation des possessions des objets au regard de la base de données originale ne sont pas produites. Par défaut, pg_dump engendre des instructions `ALTER OWNER` ou `SET SESSION AUTHORIZATION` pour fixer ces possessions. Ces instructions échouent lorsque le script n'est pas lancé par un superutilisateur (ou par l'utilisateur qui possède tous les objets de ce script). L'option `-O` est utilisée pour créer un script qui puisse être restauré par n'importe quel utilisateur. En revanche, c'est cet utilisateur qui devient propriétaire de tous les objets.

Cette option n'a d'intérêt que pour le format texte. Pour les formats archive, l'option est précisée à l'appel de `pg_restore`.

`-R`
`--no-reconnect`

Cette option, obsolète, est toujours acceptée pour des raisons de compatibilité ascendante.

`-s`
`--schema-only`

Seule la définition des objets (le schéma) est sauvegardée, pas les données.

Cette option est l'inverse de `--data-only`. Elle est similaire, mais pas identique (pour des raisons historiques), à `--section=pre-data --section=post-data`.

(Ne pas la confondre avec l'option `--schema` qui utilise le mot « schema » dans un contexte différent.)

Pour exclure les données de la table pour seulement un sous-ensemble des tables de la base de données, voir `--exclude-table-data`.

`-S` *`nom_utilisateur`*
`--superuser=`*`nom_utilisateur`*

Le nom du superutilisateur à utiliser lors de la désactivation des triggers. Cela n'a d'intérêt que si l'option `--disable-triggers` est précisée. (En règle générale, il est préférable de ne pas utiliser cette option et de lancer le script produit en tant que superutilisateur.)

`-t` *`motif`*
`--table=`*`motif`*

Sauvegarde seulement les tables dont le nom correspond à *`motif`*. Plusieurs tables sont sélectionnables en utilisant plusieurs fois l'option `-t`. De plus, le paramètre *`motif`* est interprété comme un modèle suivant les règles utilisées par les commandes `\d` de psql (voir Motifs). Du coup, plusieurs tables peuvent être sélectionnées en utilisant des caractères joker dans le modèle. Lors de l'utilisation de ces caractères, il faut faire attention à placer le modèle entre guillemets, si nécessaire, pour empêcher le shell de remplacer les jokers ; voir Exemples.

Conçue pour les tables, cette option peut être utilisée pour sauvegarder la définition des vues, vues matérialisées, tables externes et séquences correspondantes. Elle ne sauvegardera pas le contenu des vues et des vues matérialisées. Le contenu des tables externes ne sera sauvegardé que si le serveur distant correspondant est précisé avec l'option `--include-foreign-data`.

Les options `-n` et `-N` n'ont aucun effet quand l'option `-t` est utilisée car les tables sélectionnées par `-t` sont sauvegardées quelle que soit la valeur des options relatives aux schémas. Les objets qui ne sont pas des tables ne sont pas sauvegardés.

> **Note**
>
> Quand `-t` est indiqué, pg_dump ne sauvegarde aucun autre objet de la base dont la (ou les) table(s) sélectionnée(s) pourrai(en)t dépendre. Du coup, il n'est pas garanti que la sauvegarde spécifique d'une table puisse être restaurée avec succès dans une base vide.

`-T` *`motif`*
`--exclude-table=`*`motif`*

Ne sauvegarde pas les tables correspondant au modèle *`motif`*. Le modèle est interprété selon les même règles que `-t`. `-T` peut aussi être indiqué plusieurs pour exclure des tables correspondant à des modèles différents.

Quand les options `-t` et `-T` sont indiquées, seules sont sauvegardées les tables qui correspondent à au moins une option `-t` et à aucune option `-T`. Si `-T` apparaît sans `-t`, alors les tables correspondant à `-T` sont exclues de ce qui est une sauvegarde normale.

`-v`
`--verbose`

Mode verbeux. pg_dump affiche des commentaires détaillés sur les objets et les heures de début et de fin dans le fichier de sauvegarde. Des messages de progression sont également affichés sur la sortie d'erreur standard. Répéter l'option entraîne l'apparition de messages de débogage supplémentaires sur la sortie d'erreur standard.

`-V`
`--version`

Affiche la version de pg_dump puis quitte.

`-x`
`--no-privileges`
`--no-acl`

Les droits d'accès (commandes grant/revoke) ne sont pas sauvegardés.

`-Z` *`0..9`*
`--compress=`*`0..9`*

Indique le niveau de compression à utiliser. Zéro signifie sans compression. Pour le format d'archive personnalisé et le format répertoire, cela signifie la compression des segments individuels des données des tables. Par défaut, la compression se fait à un niveau modéré. Pour le format texte, indiquer une valeur différente de zéro implique une compression du fichier complet, comme s'il était passé à gzip ; mais par défaut, la sortie n'est pas compressée. Le format d'archive tar ne supporte pas du tout la compression.

`--binary-upgrade`

Cette option est destinée à être utilisée pour une mise à jour en ligne. Son utilisation dans d'autres buts n'est ni recommandée ni supportée. Le comportement de cette option peut changer dans les futures versions sans avertissement.

`--column-inserts`
`--attribute-inserts`

Extraire les données en tant que commandes `INSERT` avec des noms de colonnes explicites (`INSERT INTO` *`table`* `(`*`colonne`*`, ...) VALUES ...`). Ceci rendra la restauration très lente ; c'est surtout utile pour créer des extractions qui puissent être chargées dans des bases de données autres que PostgreSQL. Toute erreur lors du rechargement causera la perte uniquement des lignes faisant partie du `INSERT` problématique, et pas du contenu complet de la table.

`--disable-dollar-quoting`

Cette option désactive l'utilisation du caractère dollar comme délimiteur de corps de fonctions, et force leur délimitation en tant que chaîne SQL standard.

`--disable-triggers`

Cette option ne s'applique que dans le cas d'une extraction de données seules. Ceci demande à pg_dump d'inclure des commandes pour désactiver temporairement les triggers sur les tables cibles pendant que les données sont rechargées. Utilisez ceci si, sur les tables, vous avez des contraintes d'intégrité ou des triggers que vous ne voulez pas invoquer pendant le rechargement.

À l'heure actuelle, les commandes émises pour `--disable-triggers` doivent être exécutées en tant que superutilisateur. Par conséquent, vous devez aussi spécifier un nom de superutilisateur avec `-S`, ou préférablement faire attention à lancer le script résultat en tant que superutilisateur.

Cette option n'a de sens que pour le format texte simple. Pour les formats d'archive, vous pouvez spécifier cette option quand vous appelez `pg_restore`.

`--enable-row-security`

Cette option est seulement adéquate lors de la sauvegarde du contenu d'une table disposant du mode de sécurité niveau ligne. Par défaut, pg_dump configurera row_security à off pour s'assurer que toutes les données de la table soient sauvegardées. Si l'utilisateur n'a pas les droits suffisant pour contourner la sécurité niveau ligne, alors une erreur est renvoyée. Ce paramètre force pg_dump à configurer row_security à on, permettant à l'utilisateur de ne sauvegarder que le contenu auquel il a le droit d'accéder.

Notez que si vous utilisez cette option actuellement, vous serez certainement intéressé à faire une sauvegarde au format `INSERT` car les politiques de sécurité ne sont pas respectées par l'instruction `COPY FROM`.

`--exclude-table-data=`*`motif`*

Ne sauvegarde pas les données pour toute table correspondant au motif indiqué par *`motif`*. Le motif est interprété selon les même règles que pour l'option `-t`. `--exclude-table-data`

peut être utilisé plusieurs fois pour exclure des tables dont le nom correspond à des motifs différents. Cette option est utile quand vous avez besoin de la définition d'une table particulière mais pas de ses données.

Pour exclure les données de toutes les tables de la base, voir `--schema-only`.

`--extra-float-digits=`*`nchiffres`*

Utilise la valeur spécifiée pour `extra_float_digits` lors de la sauvegarde de valeurs en virgule flottante, au lieu de la précision maximale disponible. Les sauvegardes de routine ne devraient pas utiliser cette option.

`--if-exists`

Utilise des commandes `DROP ... IF EXISTS` pour supprimer des objets dans le mode `--clean`. Cela permet de supprimer les erreurs « does not exist » qui seraient sinon renvoyées. Cette option n'est pas valide sauf si `--clean` est aussi indiquée.

`--include-foreign-data=`*`foreignserver`*

Sauvegarde les données de toute table externe ayant comme serveur distant *`foreignserver`*. Plusieurs serveurs distants peuvent être sélectionnés en utilisant plusieurs fois l'options `--include-foreign-data`. De plus, le paramètre *`foreignserver`* est interprété comme un motif suivant les mêmes règles que celles utilisées par les commandes `\d` de psql (voir Motifs ci-dessous), donc plusieurs serveurs distants peuvent aussi être sélectionnés en écrivant des caractères joker dans le motif. Lors de l'utilisation des jokers, faites attention à placer le motif entre guillemets pour empêcher le shell de les prendre en compte ; voir Exemples ci-dessous. Le seule exception est qu'un motif vide est interdit.

> **Note**
>
> L'utilisation de jokers dans `--include-foreign-data` pourrait donner accès à des serveurs distants inattendus. De plus, pour utiliser cette option en toute sécurité, assurez-vous que le serveur nommé a un propriétaire de confiance.

> **Note**
>
> Quand `--include-foreign-data` est précisé, pg_dump ne vérifie pas que la table externe est accessible en écriture. De ce fait, il n'y a pas de garantie que la sauvegarde des données de la table distante pourra être restaurée sans erreurs.

`--inserts`

Extraire les données en tant que commandes `INSERT` (plutôt que `COPY`). Ceci rendra la restauration très lente ; c'est surtout utile pour créer des extractions qui puissent être chargées dans des bases de données autres que PostgreSQL. Toute erreur lors du rechargement fera que seules les lignes comprises dans le `INSERT` problématique seront perdues, plutôt que le contenu entier de la table. Notez que la restauration pourrait échouer si vous avez modifié l'ordre des colonnes. L'option `--column-inserts` protège contre les changements d'ordre des colonnes au prix de lenteurs supplémentaires.

`--load-via-partition-root`

Lors de l'export de données d'une partition, faire que les instructions `COPY` ou `INSERT` ciblent la racine du partitionnement qui contient cette partition, plutôt que la partition elle-même. Ceci fait que la partition appropriée soit re-déterminée pour chaque ligne au moment du chargement. Ceci

peut être utile quand le rechargement des données se fait sur un serveur où les lignes ne tomberont pas forcément dans les mêmes partitions que celles du serveur original. Ceci pourrait arriver si la colonne de partitionnement est de type text et que les deux systèmes ont une définition différente du collationnement utilisé pour tier la colonne de partitionnement.

`--lock-wait-timeout=`*`expiration`*

Ne pas attendre indéfiniment l'acquisition de verrous partagés sur table au démarrage de l'extraction. Échouer à la place s'il est impossible de verrouiller une table dans le temps d'*`expiration`* indiqué. L'expiration peut être spécifiée dans tous les formats acceptés par `SET statement_timeout`, les valeurs autorisées dépendant de la version du serveur sur laquelle vous faites l'extraction, mais une valeur entière en millisecondes est acceptée par toutes les versions.

`--no-comments`

Ne pas sauvegarder les commentaires.

`--no-publications`

Ne pas sauvegarder les publications.

`--no-security-labels`

Ne pas sauvegarder les labels de sécurité.

`--no-subscriptions`

Ne pas sauvegarder les souscriptions.

`--no-sync`

Par défaut, `pg_dump` attendra que tous les fichiers aient été écrits de manière sûre sur disque. Cette option force `pg_dump` à rendre la main sans attendre, ce qui est plus rapide, mais signifie qu'un arrêt brutal du serveur survenant après la sauvegarde peut laisser la sauvegarde dans un état corrompu. De manière générale, cette option est utile durant les tests mais ne devrait pas être utilisée dans un environnement de production.

`--no-table-access-method`

Ne pas générer de commandes pour sélectionner les méthodes d'accès aux tables. Avec cette option, tous les objets seront créés avec la méthode d'accès par défaut au moment de la restauration.

Cette option est ignorée lors de la création d'un fichier d'archive (non texte). Pour les formats d'archivage, vous pouvez indiquer l'option lors de l'appel à `pg_restore`.

`--no-tablespaces`

Ne pas générer de commandes pour créer des tablespace, ni sélectionner de tablespace pour les objets. Avec cette option, tous les objets seront créés dans le tablespace par défaut durant la restauration.

Cette option n'a de sens que pour le format texte simple. Pour les formats d'archive, vous pouvez spécifier cette option quand vous appelez `pg_restore`.

`--no-toast-compression`

Ne génère pas de commandes pour définir les méthodes de compression TOAST. Avec cette option, toutes les colonnes seront restaurées avec la méthode de compression par défaut en vigueur au moment de la restauration.

`--no-unlogged-table-data`

Ne pas exporter le contenu des tables et séquences non journalisées (unlogged). Cette option n'a aucun effet sur le fait que la définition (schéma) des tables et séquences soit exportée ou non ; seul l'export des données de la table et de la séquence est supprimé. Les données des tables et séquences non journalisées sont toujours exclues lors d'une sauvegarde à partir d'un serveur en standby.

`--on-conflict-do-nothing`

Ajouter `ON CONFLICT DO NOTHING` aux commandes `INSERT`. Cette option n'est seulement valide que si `--inserts`, `--column-inserts` ou `--rows-per-insert` est aussi utilisée.

`--quote-all-identifiers`

Force la mise entre guillemets de tous les identifiants. Cette option est recommandée lors de la sauvegarde d'un serveur PostgreSQL dont la version majeure est différente de celle du pg_dump ou quand le résultat est prévu d'être rechargé dans une autre version majeure. Par défaut, pg_dump met entre guillemets uniquement les identifiants qui sont des mots réservés dans sa propre version majeure. Ceci peut poser parfois des problèmes de compatibilité lors de l'utilisation de serveurs de versions différentes qui auraient des ensembles différents de mots clés. Utiliser `--quote-all-identifiers` empêche ce type de problèmes au prix d'un script résultant plus difficile à lire.

`--restrict-key=`*`restrict_key`*

Utilise la chaîne fournie comme clé de `\restrict` dans psql dans la sortie de la sauvegarde. Cela peut seulement être indiqué pour les sauvegardes textes, autrement dit quand `--format` être configuré à `plain` ou quand l'option `--format` est omise. Si aucune clé de restriction n'est indiqué, pg_dump en générera une aléatoirement si nécessaire. Les clés doivent contenir uniquement des caractères alphanumériques.

Cette option a principalement pour but des tests et d'autres scénarios nécessitant une sortie répétable (par exemple pour comparer les fichiers de sauvegarde). Ce n'est pas recommandé pour une utilisation standard, car un serveur malicieux avec une connaissance de la clé pourrait injecter un code arbitraire qui sera exécuté sur la machine qui lance psql avec la sauvegarde en sortie.

`--rows-per-insert=`*`nlignes`*

Sauvegarde les données sous la forme de commandes `INSERT` (plutôt qu'avec `COPY`). Contrôle le nombre maximum de lignes par commande `INSERT`. La valeur indiquée doit être strictement positive. Toute erreur lors du rechargement provoquera uniquement la perte des lignes faisant partie du `INSERT` problématique, et non pas le contenu complet de la table.

`--section=`*`nom_section`*

Sauvegarde seulement la section nommée. Le nom de la section peut être `pre-data`, `data` ou `post-data`. Cette option peut être spécifiée plus d'une fois pour sélectionner plusieurs sections. La valeur par défaut est toutes les sections.

La section data contient toutes les données des tables ainsi que la définition des Large Objects et les valeurs des séquences. Les éléments post-data incluent la définition des index, triggers, règles et contraintes (autres que les contraintes de vérification). Les éléments pre-data incluent en tous les autres éléments de définition.

`--serializable-deferrable`

Utiliser une transaction `sérialisable` pour l'export, pour garantir que l'instantané utilisé est cohérent avec les états futurs de la base; mais ceci est effectué par l'attente d'un point dans le flux des transactions auquel aucune anomalie ne puisse être présente, afin qu'il n'y ait aucun risque que l'export échoue ou cause l'annulation d'une autre transaction pour `erreur`

`de sérialisation`. Voyez Chapitre 13 pour davantage d'informations sur l'isolation des transactions et le contrôle d'accès concurrent.

Cette option est inutile pour une sauvegarde qui ne sera utilisée qu'en cas de récupération après sinistre. Elle pourrait être utile pour une sauvegarde utilisée pour charger une copie de la base pour du reporting ou toute autre activité en lecture seule tandis que la base originale continue à être mise à jour. Sans cela, la sauvegarde serait dans un état incohérent avec l'exécution sérielle des transactions qui auront été finalement validées. Par exemple, si un traitement de type batch est exécuté, un batch pourrait apparaître comme terminé dans la sauvegarde sans que tous les éléments du batch n'apparaissent.

Cette option ne fera aucune différence si aucune transaction en lecture-écriture n'est active au lancement de pg_dump. Si des transactions en lecture-écriture sont actives, le démarrage de la sauvegarde pourrait être retardé pour une durée indéterminée. Une fois qu'il sera démarré, la performance est identique à celle de la sauvegarde sans cette option.

`--snapshot=`*`nom_snapshot`*

Utilise l'image de base de données synchronisée spécifiée lors de la sauvegarde d'une base de données (voir Tableau 9.92 pour plus de détails).

Cette option est utile lorsqu'il est nécessaire de synchroniser la sauvegarde avec un slot de réplication logique (voir Chapitre 49) ou avec une session concurrente.

Dans le cas d'une sauvegarde parallèle, le nom de l'image défini par cette option est utilisé plutôt que de prendre une nouvelle image de base.

`--strict-names`

Implique que chaque qualificateur d'extension (`-e`/ `--extension`), de schéma(`-n`/`--schema`) et de table (`-t`/`--table`) corresponde à au moins un ou une extension/schéma/table dans la base à sauvegarder. Notez que, si aucun motif d'extension/schéma/table ne trouve une correspondance, pg_dump générera une erreur, y compris sans `--strict-names`.

Cette option n'a pas d'effet sur `-N` / `--exclude-schema`, `-T` / `--exclude_table` et `--exclude-table-date`. Tout échec de correspondance pour un motif d'exclusion n'est pas considéré comme une erreur.

`--use-set-session-authorization`

Émettre des commandes SQL standard `SET SESSION AUTHORIZATION` à la place de commandes `ALTER OWNER` pour déterminer l'appartenance d'objet. Ceci rend l'extraction davantage compatible avec les standards, mais, suivant l'historique des objets de l'extraction, peut ne pas se restaurer correctement. Par ailleurs, une extraction utilisant `SET SESSION AUTHORIZATION` nécessitera certainement des droits superutilisateur pour se restaurer correctement, alors que `ALTER OWNER` nécessite des droits moins élevés.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_dump, puis quitte

Les options de ligne de commande suivantes gèrent les paramètres de connexion :

`-d` *`nom_base`*
`--dbname=`*`nom_base`*

Indique le nom de la base de données de connexion. Ceci revient à spécifier *`nom_base`* comme premier argument sans option sur la ligne de commande. Ce nom de base peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront toutes les options en ligne de commande conflictuelles.

`-h` *hôte*
`--host` *hôte*

Le nom d'hôte de la machine sur laquelle le serveur de bases de données est exécuté. Si la valeur commence par une barre oblique (/), elle est utilisée comme répertoire pour le socket de domaine Unix. La valeur par défaut est fournie par la variable d'environnement `PGHOST`, si elle est initialisée. Dans le cas contraire, une connexion sur la socket de domaine Unix est tentée.

`-p` *port*
`--port` *port*

Le port TCP ou le fichier local de socket de domaine Unix sur lequel le serveur écoute les connexions. La valeur par défaut est fournie par la variable d'environnement `PGPORT`, si elle est initialisée. Dans le cas contraire, il s'agit de la valeur fournie à la compilation.

`-U` *nomutilisateur*
`--username` *nomutilisateur*

Le nom d'utilisateur utilisé pour la connexion.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force pg_dump à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais nécessaire car pg_dump demande automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, pg_dump perd une tentative de connexion pour tester si le serveur demande un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`--role=`*nomrole*

Spécifie un rôle à utiliser pour créer l'extraction. Avec cette option, pg_dump émet une commande `SET ROLE` *nomrole* après s'être connecté à la base. C'est utile quand l'utilisateur authentifié (indiqué par `-U`) n'a pas les droits dont pg_dump a besoin, mais peut basculer vers un rôle qui les a. Certaines installations ont une politique qui est contre se connecter directement en tant que superutilisateur, et l'utilisation de cette option permet que les extractions soient faites sans violer cette politique.

Environnement

`PGDATABASE`
`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Paramètres de connexion par défaut.

`PG_COLOR`

Indique s'il faut utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise les variables d'environnement supportées par la bibliothèque libpq (voir Section 34.15).

Diagnostics

pg_dump exécute intrinsèquement des instructions `SELECT`. Si des problèmes apparaissent à l'exécution de pg_dump, psql peut être utilisé pour s'assurer qu'il est possible de sélectionner des informations dans la base de données. De plus, tout paramètre de connexion par défaut et toute variable d'environnement utilisé par la bibliothèque libpq s'appliquent.

L'activité générée par pg_dump dans la base de données est normalement collectée par le système de statistiques cumulées. Si c'est gênant, vous pouvez positionner le paramètre `track_counts` à false via `PGOPTIONS` ou la commande `ALTER USER`.

Notes

Si des ajouts locaux à la base `template1` ont été effectués, il est impératif de s'assurer que la sortie de pg_dump est effectivement restaurée dans une base vide ; dans le cas contraire, il est fort probable que la duplication des définitions des objets ajoutés engendre des erreurs. Pour obtenir une base vide de tout ajout local, on utilise `template0` à la place de `template1` comme modèle. Par exemple :

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Quand une sauvegarde des seules données est sélectionnée et que l'option `--disable-triggers` est utilisée, pg_dump engendre des commandes de désactivation des triggers sur les tables utilisateur avant l'insertion des données, puis après coup, des commandes de réactivation après l'insertion. Si la restauration est interrompue, il se peut que les catalogues systèmes conservent cette position.

Le fichier de sauvegarde produit par pg_dump ne contient pas les statistiques utilisées par l'optimiseur pour la planification des requêtes. Il est donc conseillé, pour assurer des performances optimales, de lancer `ANALYZE` après la restauration d'une sauvegarde ; voir Section 25.1.3 et Section 25.1.6 pour plus d'informations.

Parce que pg_dump est utilisé pour transférer des données vers des nouvelles versions de PostgreSQL, la sortie de pg_dump devra pouvoir se charger dans les versions du serveur PostgreSQL plus récentes que la version de pg_dump. pg_dump peut aussi extraire des données de serveurs PostgreSQL plus anciens que sa propre version. (À l'heure actuelle, les versions de serveurs supportées vont jusqu'à la 9.2.) Toutefois, pg_dump ne peut pas réaliser d'extraction de serveurs PostgreSQL plus récents que sa propre version majeure ; il refusera même d'essayer, plutôt que de risquer de fournir une extraction invalide. Par ailleurs, il n'est pas garanti que la sortie de pg_dump puisse être chargée dans un serveur d'une version majeure plus ancienne -- pas même si l'extraction a été faite à partir d'un serveur dans cette version. Charger un fichier d'extraction dans un serveur de version plus ancienne pourra requérir une édition manuelle du fichier pour supprimer les syntaxe incomprises de l'ancien serveur. L'utilisation de l'option `--quote-all-identifiers` est recommendée lors de l'utilisation avec des versions différentes, car cela permet d'empêcher la venue de problèmes provenant de listes de mots clés dans différentes versions de PostgreSQL.

Lors de la sauvegarde des souscription de réplication logique, pg_dump générera des commandes `CREATE SUBSCRIPTION` qui utilisent l'option `connect = false`, afin que la restauration des souscriptions ne fasse pas de connexions distantes pour créer un slot de réplication ou pour la copie initiale de table. De cette façon, la sauvegarde peut être restaurée sans avoir besoin d'un accès réseau aux serveurs distants. Il est alors à la charge de l'utilisateur de réactiver les souscriptions de manière adaptée. Si les hôtes impliquées ont changés, l'information de connexion pourrait nécessiter d'être changée. Il pourrait également être approprié de tronquer les tables cibles avant de lancer une nouvelle copie complète des tables. Si les utilisateurs ont pour but de copier les données originales lors du rafraichissement, ils doivent créer le slot avec `two_phase = false`. Après la synchronisation initiale, l'option `two_phase` sera automatiquement activée si la souscription a été créée à l'origine avec l'option `two_phase = true`.

Il est généralement recommendé d'utiliser l'option `-X` (`--no-psqlrc`) lors de la restauration d'une base à partir d'un script texte de pg_dump pour s'assurer d'un traitement propre et empêcher tout conflit potentiel avec des configurations personnalisées de psql.

Exemples

Sauvegarder une base appelée `ma_base` dans un script SQL :

```
$ pg_dump ma_base > base.sql
```

Pour sauvegarder une base de données dans une archive au format répertoire :

```
$ pg_dump -Fd ma_base -f rep_sauve
```

Charger ce script dans une base nouvellement créée et nommée `nouvelle_base` :

```
$ psql -X -d nouvelle_base -f base.sql
```

Sauvegarder une base dans un fichier au format personnalisé :

```
$ pg_dump -Fc ma_base > base.dump
```

Pour sauvegarder une base de données en utilisant le format répertoire et en activant la parallélisation sur cinq jobs :

```
$ pg_dump -Fd ma_base -j 5 -f rep_sauvegarde
```

Charger un fichier d'archive dans une nouvelle base nommée `nouvelle_base` :

```
$ pg_restore -d nouvelle_base base.dump
```

Pour recharger un fichier archive dans la même base de données, annuler le contenu actuel de cette base :

```
$ pg_restore -d postgres --clean --create db.dump
```

Sauvegarder la table nommée `mytab` :

```
$ pg_dump -t ma_table ma_base > base.sql
```

Sauvegarder toutes les tables du schéma `detroit` et dont le nom commence par `emp` sauf la table nommée `traces_employes` :

```
$ pg_dump -t 'detroit.emp*' -T detroit.traces_employes ma_base >
 base.sql
```

Sauvegarder tous les schémas dont le nom commence par `est` ou `ouest` et se termine par `gsm`, en excluant les schémas dont le nom contient le mot `test` :

```
$ pg_dump -n 'est*gsm' -n 'ouest*gsm' -N '*test*' ma_base >
 base.sql
```

Idem mais en utilisant des expressions rationnelles dans les options :

```
$ pg_dump -n '(est|ouest)*gsm' -N '*test*' ma_base > base.sql
```

Sauvegarder tous les objets de la base sauf les tables dont le nom commence par `ts_` :

```
$ pg_dump -T 'ts_*' ma_base > base.sql
```

Pour indiquer un nom qui comporte des majuscules dans les options `-t` et assimilées, il faut ajouter des guillemets doubles ; sinon le nom est converti en minuscules (voir Motifs). Les guillemets doubles sont interprétés par le shell et doivent donc être placés entre guillemets. Du coup, pour sauvegarder une seule table dont le nom comporte des majuscules, on utilise une commande du style :

```
$ pg_dump -t "\"NomAMajuscule\"" ma_base > ma_base.sql
```

Voir aussi

pg_dumpall, pg_restore, psql

pg_dumpall

pg_dumpall — extraire une instance de bases de données PostgreSQL dans un fichier de script

Synopsis

`pg_dumpall` [*option_connexion*...] [*option*...]

Description

pg_dumpall est un outil d'extraction (« sauvegarde ») de toutes les bases de données PostgreSQL de l'instance vers un fichier script. Celui-ci contient les commandes SQL utilisables pour restaurer les bases de données avec psql. Cela est obtenu en appelant pg_dump pour chaque base de données de la grappe. pg_dumpall sauvegarde aussi les objets globaux, communs à toutes les bases de données, autrement dit les rôles, les tablespaces et les droits pour les paramètres de configuration. (pg_dump ne sauvegarde pas ces objets.)

Puisque pg_dumpall lit les tables de toutes les bases de données, il est préférable d'avoir les droits de superutilisateur de la base de données pour obtenir une sauvegarde complète. De plus, il faut détenir des droits superutilisateur pour exécuter le script produit, afin de pouvoir créer les rôles et les bases de données.

Le script SQL est écrit sur la sortie standard. Utilisez l'option `-f`/`--file` ou les opérateurs shell pour la rediriger vers un fichier.

pg_dumpall se connecte plusieurs fois au serveur PostgreSQL (une fois par base de données). Si l'authentification par mot de passe est utilisé, un mot de passe est demandé à chaque tentative de connexion. Il est intéressant de disposer d'un fichier `~/.pgpass` dans de tels cas. Voir Section 34.16 pour plus d'informations.

> **Avertissement**
>
> Restaurer une sauvegarde fait que la destination exécute du code arbitraire. Les sauvegardes et restaurations partielles ne limitent pas cela. S'il n'est pas possible de faire confiance aux superutilisateurs de la source, les requêtes SQL de restauration doivent être inspectées avant d'être exécutées. Notez que le client exécutant la sauvegarde et la restauration n'a pas besoin de faire confiance aux superutilisateurs source et destination.

Options

Les options suivantes, en ligne de commande, contrôlent le contenu et le format de la sortie.

`-a`
`--data-only`

Seules les données sont sauvegardées, pas le schéma (définition des données).

`-c`
`--clean`

Émet des commandes SQL `DROP` pour supprimer toutes les bases, rôles et tablespaces sauvegardés avant de les recréer. Cette option est utile quand la restauration doit écraser une instance existante. Si un des objets n'existe pas dans l'instance de destination, les messages

d'erreur, à ignorer, seront renvoyés lors de la restauration sauf si l'option `--if-exists` est aussi indiquée.

`-E` *`encoding`*
`--encoding=`*`encoding`*

Crée la sauvegarde dans l'encodage spécifié. Par défaut, l'encodage de la base est utilisé. (Une autre façon d'obtenir le même résultat est de configurer la variable d'environnement `PGCLIENTENCODING` avec l'encodage désiré pour la sauvegarde.)

`-f` *`nomfichier`*
`--file=`*`nomfichier`*

Envoie le résultat dans le fichier indiqué. Si cette option n'est pas utilisée, la sortie standard est utilisée.

`-g`
`--globals-only`

Seuls les objets globaux sont sauvegardés (rôles et tablespaces), pas les bases de données.

`-O`
`--no-owner`

Les commandes permettant de positionner les propriétaires des objets à ceux de la base de données originale. Par défaut, pg_dumpall lance les instructions `ALTER OWNER` ou `SET SESSION AUTHORIZATION` pour configurer le propriétaire des éléments créés. Ces instructions échouent lorsque le script est lancé par un utilisateur ne disposant pas des droits de superutilisateur (ou ne possédant pas les droits du propriétaire de tous les objets compris dans ce script). Pour que ce qui devient alors propriétaire de tous les objets créés, l'option `-O` doit être utilisée.

`-r`
`--roles-only`

Sauvegarde seulement les rôles, pas les bases ni les tablespaces.

`-s`
`--schema-only`

Seules les définitions des objets (schéma), sans les données, sont sauvegardées.

`-S` *`username`*
`--superuser=`*`username`*

Précise le nom du superutilisateur à utiliser pour la désactivation des triggers. Cela n'a d'intérêt que lorsque `--disable-triggers` est utilisé. (Il est en général préférable de ne pas utiliser cette option et de lancer le script résultant en tant que superutilisateur.)

`-t`
`--tablespaces-only`

Sauvegarde seulement les tablespaces, ni les bases ni les rôles.

`-v`
`--verbose`

Indique l'utilisation du mode verbeux. Ainsi pg_dumpall affiche les heures de démarrage/arrêt dans le fichier de sauvegarde et les messages de progression sur la sortie standard. Répéter cette option fait apparaitre des messages de débuggage supplémentaires sur la sortie des erreurs. Cette option est aussi passée à pg_dump.

`-V`
`--version`

Affiche la version de pg_dumpall puis quitte.

`-x`
`--no-privileges`
`--no-acl`

Les droits d'accès (commandes GRANT/REVOKE) ne sont pas sauvegardés.

`--binary-upgrade`

Cette option est destinée à être utilisée pour une mise à jour en ligne. Son utilisation dans d'autres buts n'est ni recommandée ni supportée. Le comportement de cette option peut changer dans les versions futures sans avertissement.

`--column-inserts`
`--attribute-inserts`

Extraire les données en tant que commandes `INSERT` avec des noms de colonnes explicites (`INSERT INTO` *`table`* `(`*`colonne`*`, ...) VALUES ...`). Ceci rendra la restauration très lente ; c'est surtout utile pour créer des extractions qui puissent être chargées dans des bases de données autres que PostgreSQL.

`--disable-dollar-quoting`

L'utilisation du dollar comme guillemet dans le corps des fonctions est désactivée. Celles-ci sont mises entre guillemets en accord avec la syntaxe du standard SQL.

`--disable-triggers`

Cette option n'est utile que lors de la création d'une sauvegarde des seules données. pg_dumpall inclut les commandes de désactivation temporaire des triggers sur les tables cibles pendant le rechargement des données. Cette option est utile lorsqu'il existe des vérifications d'intégrité référentielle ou des triggers sur les tables qu'on ne souhaite pas voir appelés lors du rechargement des données.

Actuellement, les commandes émises par `--disable-triggers` nécessitent d'être lancées par un superutilisateur. Il est donc impératif de préciser le nom du superutilisateur avec `-S` ou, préférentiellement, de lancer le script résultant en tant que superutilisateur.

`--exclude-database=`*`motif`*

Ne pas sauvegarder les bases dont le nom correspond à *`motif`*. Plusieurs motifs peuvent être proposés en écrivant plusieurs fois l'option `--exclude-database`. Le paramètre *`motif`* est interprété comme un motif suivant les mêmes règles que les méta-commandes `\d` de psql (voir Motifs), donc plusieurs bases de données peuvent aussi être exclues en écrivant les caractères joker dans le motif. En utilisant les caractères joker, faite attention à bien mettre entre guillemets le motif si nécessaire pour empêcher qu'il ne soit interprété par le shell.

`--extra-float-digits=`*`nchiffres`*

Utiliser la valeur indiquée pour le paramètre extra_float_digits lors de la sauvegarde de données en virgule flottante au lieu de prendre la précision maximale disponible. Les sauvegardes standards ne devraient pas utiliser cette option.

`--if-exists`

Utilise des commandes `DROP ... IF EXISTS` pour supprimer des objets dans le mode `--clean`. Cela permet de supprimer les erreurs « does not exist » qui seraient sinon renvoyées. Cette option n'est pas valide sauf si `--clean` est aussi indiquée.

`--inserts`

Extraire les données en tant que commandes `INSERT` (plutôt que `COPY`). Ceci rendra la restauration très lente ; c'est surtout utile pour créer des extractions qui puissent être chargées dans des bases de données autres que PostgreSQL. Notez que la restauration peut échouer complètement si vous avez changé l'ordre des colonnes. L'option `--column-inserts` est plus sûre, mais encore plus lente.

`--load-via-partition-root`

Lors de l'export de données d'une partition, faire que les instructions `COPY` ou `INSERT` ciblent la racine du partitionnement qui contient cette partition, plutôt que la partition elle-même. Ceci fait que la partition appropriée soit re-déterminée pour chaque ligne au moment du chargement. Ceci peut être utile quand le rechargement des données se fait sur un serveur où les lignes ne tomberont pas forcément dans les mêmes partitions que celles du serveur original. Ceci pourrait arriver si la colonne de partitionnement est de type text et que les deux systèmes ont une définition différente du collationnement utilisé pour tier la colonne de partitionnement.

`--lock-wait-timeout=`*`expiration`*

Ne pas attendre indéfiniment l'acquisition de verrous partagés sur table au démarrage de l'extraction. Échouer à la place s'il est impossible de verrouiller une table dans le temps d'*`expiration`* spécifié. L'expiration peut être indiquée dans tous les formats acceptés par `SET statement_timeout`.

`--no-subscriptions`

Ne sauvegarde pas les souscriptions.

`--no-sync`

Par défaut, `pg_dumpall` attendra que tous les fichiers aient été écrits de manière sûre sur disque. Cette option force `pg_dumpall` à rendre la main sans attendre, ce qui est plus rapide, mais signifie qu'un arrêt brutal du serveur survenant après la sauvegarde peut laisser la sauvegarde dans un état corrompu. De manière générale, cette option est utile durant les tests mais ne devrait pas être utilisée dans un environnement de production.

`--no-table-access-method`

Ne pas générer de commandes pour sélectionner les méthodes d'accès aux tables. Avec cette option, tous les objets seront créés avec la méthode d'accès de table par défaut lors de la restauration.

`--no-tablespaces`

Ne pas générer de commandes pour créer des tablespace, ni sélectionner de tablespace pour les objets. Avec cette option, tous les objets seront créés dans le tablespace par défaut durant la restauration.

`--no-comments`

Ne sauvegarde pas les commentaires.

`--no-publications`

Ne sauvegarde pas les publications.

`--no-role-passwords`

Ne sauvegarde pas les mots de passe des rôles. Lors de la restauration, les rôles auront un mot de passe vide et l'authentification par mot de passe échouera toujours jusqu'à ce que le mot de

passe soit initialisé. Puisque les valeurs des mots de passe ne sont pas nécessaires quand cette option est spécifiée, l'information sur le rôle est lue depuis le catalogue système `pg_roles` au lieu de `pg_authid`. De ce fait, cette option aide aussi si l'accès à `pg_authid` est restreint par certaines politiques de sécurité.

`--no-security-labels`

Ne sauvegarde pas les labels de sécurité.

`--no-toast-compression`

N'affiche pas de commandes pour configurer les méthodes de compression des TOAST. Avec cette option, toutes les colonnes seront restaurées avec le paramètrage de compression par défaut.

`--no-unlogged-table-data`

Ne sauvegarde pas le contenu des tables non tracées dans les journaux de transactions. Cette option n'a pas d'effet sur la sauvegarde des définitions de table ; il supprime seulement la sauvegarde des données des tables.

`--on-conflict-do-nothing`

Ajoute `ON CONFLICT DO NOTHING` aux commandes `INSERT`. Cette option n'est valide que si `--inserts` ou `--column-inserts` est aussi indiquée.

`--quote-all-identifiers`

Force la mise entre guillemets de tous les identifiants. Cette option est recommandée lors de la sauvegarde d'un serveur PostgreSQL dont la version majeure est différente de celle du pg_dumpall ou quand le résultat est prévu d'être rechargé dans une autre version majeure. Par défaut, pg_dumpall met entre guillemets uniquement les identifiants qui sont des mots réservés dans sa propre version majeure. Ceci peut poser parfois des problèmes de compatibilité lors de l'utilisation de serveurs de versions différentes qui auraient des ensembles différents de mots clés. Utiliser `--quote-all-identifiers` empêche ce type de problèmes au prix d'un script résultant plus difficile à lire.

`--restrict-key=`*`restrict_key`*

Utilise la chaîne fournie comme clé de `\restrict` dans psql dans la sortie de la sauvegarde. Si aucune clé de restriction n'est indiqué, pg_dumpall en générera une aléatoirement si nécessaire. Les clés doivent contenir uniquement des caractères alphanumériques.

Cette option a principalement pour but des tests et d'autres scénarios nécessitant une sortie répétable (par exemple pour comparer les fichiers de sauvegarde). Ce n'est pas recommandé pour une utilisation standard, car un serveur malicieux avec une connaissance de la clé pourrait injecter un code arbitraire qui sera exécuté sur la machine qui lance psql avec la sauvegarde en sortie.

`--rows-per-insert=`*`nlignes`*

Sauvegarde les données sous forme de commandes `INSERT` (plutôt que `COPY`). Contrôle le nombre maximum de lignes par commande `INSERT`. La valeur spécifiée doit être un nombre supérieur à zéro. Toute erreur lors du rechargement causera seulement la perte des lignes faisant partie de l'opération `INSERT` problématique, plutôt que la part du contenu complet de la table.

`--use-set-session-authorization`

Les commandes `SET SESSION AUTHORIZATION` du standard SQL sont affichées à la place des commandes `ALTER OWNER` pour préciser le propriétaire de l'objet. Cela améliore la compatibilité de la sauvegarde vis-à-vis des standard. Toutefois, du fait de l'ordre d'apparition des objets dans la sauvegarde, la restauration peut ne pas être correcte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_dumpall, puis quitte

Les options suivantes en ligne de commande contrôlent les paramètres de connexion à la base de données.

`-d` *`connstr`*
`--dbname=`*`connstr`*

Indique les paramètres utilisés pour se connecter au serveur sous la forme d'une chaîne de connexion ; elles surchargeront les options en ligne de commande conflictuelles.

Cette option est appelée `--dbname` par cohérence avec les autres applications clientes. Comme pg_dumpall a besoin de se connecter à plusieurs bases de données, le nom de la base indiqué dans la chaîne de connexion sera ignorée. Utilisez l'option `-l` pour spécifier le nom de la base utilisée pour la connexion initiale sauvegardant les objets globaux et découvrant les bases à sauvegarder.

`-h` *`hôte`*
`--host=`*`hôte`*

Précise le nom d'hôte de la machine sur laquelle le serveur de bases de données est en cours d'exécution. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix. La valeur par défaut est prise à partir de la variable d'environnement `PGHOST`, si elle est initialisée, sinon une connexion socket de domaine Unix est tentée.

`-l` *`dbname`*
`--database=`*`dbname`*

Spécifie le nom de la base où se connecter pour la sauvegarde des objets globaux et pour découvrir les bases qui devraient être sauvegardées. Si cette option n'est pas utilisée, la base `postgres` est utilisé et, si elle n'existe pas, `template1` sera utilisée.

`-p` *`port`*
`--port=`*`port`*

Précise le port TCP ou l'extension du fichier socket de domaine Unix local sur lequel le serveur est en écoute des connexions. La valeur par défaut est la variable d'environnement `PGPORT`, si elle est initialisée, ou la valeur utilisée lors de la compilation.

`-U` *`nomutilisateur`*
`--username=`*`nomutilisateur`*

Utilisateur utilisé pour initier la connexion.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force pg_dumpall à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car pg_dumpall demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, pg_dumpall perdra une

tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Notez que le mot de passe sera demandé pour chaque base de données à sauvegarder. Habituellement, il est préférable de configurer un fichier `~/.pgpass` pour que de s'en tenir à une saisie manuelle du mot de passe.

`--role=`*`nomrole`*

Spécifie un rôle à utiliser pour créer l'extraction. Avec cette option, pg_dumpall émet une commande `SET ROLE` *`nomrole`* après s'être connecté à la base. C'est utile quand l'utilisateur authentifié (indiqué par `-U`) n'a pas les droits dont pg_dumpall a besoin, mais peut basculer vers un rôle qui les a. Certaines installations ont une politique qui est contre se connecter directement en tant que superutilisateur, et l'utilisation de cette option permet que les extractions soient faites sans violer cette politique.

Environnement

`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Paramètres de connexion par défaut

`PG_COLOR`

Indique s'il faut utiliser la couleur pour les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 34.15).

Notes

Comme pg_dumpall appelle pg_dump en interne, certains messages de diagnostic se réfèrent en fait à pg_dump.

L'option `--clean` peut être utile même si votre intention est de restaurer le script de sauvegarde sur une instance vierge. L'utilisation de `--clean` autorise le script à supprimer puis re-créer les bases de données internes `postgres` et `template1`, en s'assurant que ces bases conserveront les mêmes propriétés (par exemple la locale et l'encodage) que sur l'instance source. Sans l'option, ces bases conserveront les propriétés existantes au niveau base ainsi que le contenu pré-existant.

Une fois la restauration effectuée, il est conseillé de lancer `ANALYZE` sur chaque base de données, de façon à ce que l'optimiseur dispose de statistiques utiles. `vacuumdb -a -z` peut également être utilisé pour analyser toutes les bases de données.

On ne doit pas s'attendre à ce que le script de sauvegarde s'exécute sans erreur. En particulier, comme le script exécutera un `CREATE ROLE` pour chaque rôle existant sur l'instance source, il est certain d'obtenir une erreur « role already exists » pour le superutilisateur initial sauf si l'instance de destination a été initialisé avec un autre nom pour le superutilisateur initial. Cette erreur est sans gravité et doit être ignorée. L'utilisation de l'option `--clean` a des chances de produire des messages d'erreur supplémentaires sans risque pour les objets inexistants. Vous pouvez minimiser ces erreurs en ajoutant l'option `--if-exists`.

pg_dumpall requiert que tous les tablespaces nécessaires existent avant la restauration. Dans le cas contraire, la création de la base échouera pour une base qui ne se trouve pas dans l'emplacement par défaut.

Il est généralement recommendé d'utiliser l'option `-X` (`--no-psqlrc`) lors de la restauration d'une base à partir d'un script de pg_dumpall pour s'assurer d'une restauration propre et empêcher tout conflit potentiel avec des configurations personnalisées de psql. De plus, comme un script pg_dumpall pourrait inclure des méta-commandes psql, il pourrait être incompatible avec des outils clients autres que psql.

Exemples

Sauvegarder toutes les bases de données :

```
$ pg_dumpall > db.out
```

Pour recharger les bases de données à partir de ce fichier, vous pouvez utiliser :

```
$ psql -X -f db.out postgres
```

La base de données utilisée pour la connexion initiale n'a pas d'importance ici car le fichier de script créé par pg_dumpall contient les commandes nécessaires à la création et à la connexion aux bases de données sauvegardées. Si vous utilisez l'option `--clean`, vous devez vous connecter à la base `postgres` au début ; le script tentera de supprimer les autres bases immédiatement et ceci échouera pour la base où vous êtes connecté.

Voir aussi

Vérifier pg_dump pour des détails sur les conditions d'erreur possibles.

pg_isready

pg_isready — vérifier le statut de connexion d'un serveur PostgreSQL

Synopsis

`pg_isready` [*option-connexion*...] [*option*...]

Description

pg_isready est un outil qui vérifie le statut de connexion d'un serveur PostgreSQL. Le code de sortie indique le résultat de la vérification.

Options

`-d` *nom_base*
`--dbname=`*nom_base*

Indique le nom de la base de données de connexion. Ce nom de base peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront toutes les options en ligne de commande conflictuelles.

`-h` *hôte*
`--host=`*hôte*

Spécifie le nom d'hôte de la machine sur laquelle le serveur de bases de données est exécuté. Si la valeur commence par une barre oblique (/), elle est utilisée comme répertoire pour le socket de domaine Unix.

`-p` *port*
`--port=`*port*

Spécifie le port TCP ou l'extension du fichier local de socket de domaine Unix sur lequel le serveur écoute les connexions. La valeur par défaut est fournie par la variable d'environnement `PGPORT`, si elle est initialisée. Dans le cas contraire, il s'agit de la valeur fournie à la compilation, habituellement 5432.

`-q`
`--quiet`

N'affiche pas de message de statut. Ceci est utile pour son utilisation dans un script.

`-t` *secondes*
`--timeout=`*secondes*

Le nombre maximum de secondes à attendre lors d'une tentative de connexion et que le serveur ne répond pas. Le configurer à 0 désactive l'attente. Par défaut, la valeur est de trois secondes.

`-U` *nom_utilisateur*
`--username=`*nom_utilisateur*

Se connecter à la base en tant que l'utilisateur *nom_utilisateur* à la place du défaut.

`-V`
`--version`

Affiche la version de pg_isready, puis quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_isready, puis quitte.

Code de sortie

pg_isready renvoie `0` au shell si le serveur accepte normalement les connexions, `1` si le serveur rejette les connexions (par exemple lors du démarrage), `2` s'il n'y a pas de réponse une fois passé le délai d'attente et `3` si aucune tentative n'a eu lieu (par exemple à cause de paramètres invalides).

Environnement

`pg_isready`, comme la majorité des outils PostgreSQL, utilise les variables d'environnement supportées par libpq (voir Section 34.15).

La variable d'environnement `PG_COLOR` indique s'il faut utiliser les couleurs dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Notes

Il n'est pas nécessaire de fournir un nom d'utilisateur, un mot de passe ou une base de données valides pour obtenir le statut du serveur. Néanmoins, si des valeurs incorrectes sont fournies, le serveur tracera une tentative échouée de connexion.

Exemples

Usage standard :

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

Exécuter avec les paramètres de connexion vers une instance PostgreSQL en cours de démarrage :

```
$ pg_isready -h localhost -p 5433
localhost:5433 - rejecting connections
$ echo $?
1
```

Exécuter avec les paramètres de connexion vers une instance PostgreSQL qui ne répond pas :

```
$ pg_isready -h someremotehost
someremotehost:5432 - no response
$ echo $?
2
```

pg_receivewal

pg_receivewal — suit le flux des journaux de transactions d'un serveur PostgreSQL

Synopsis

`pg_receivewal` [*option*...]

Description

pg_receivewal est utilisé pour suivre le flux des journaux de transactions d'une instance de PostgreSQL en cours d'activité. Les journaux de transactions sont suivis en utilisant le flux du protocole de réplication, et sont écrits sous forme de fichiers dans un répertoire local. Ce répertoire peut être utilisé comme emplacement des archives dans l'optique d'une restauration utilisant le mécanisme de sauvegarde à chaud et de récupération à un instant t (*PITR*, voir Section 26.3).

pg_receivewal suit le flux des journaux de transactions en temps réel alors qu'il est généré sur le serveur principal. Il n'attend pas la fin de l'écriture d'un segment complet d'un journal de transactions comme archive_command et archive_library le font. Pour ces raisons, il n'est pas nécessaire de configurer archive_timeout lors de l'utilisation de pg_receivewal

Contrairement au receveur de WAL d'un serveur PostgreSQL standby, pg_receivewal place les données WAL sur disque par défaut uniquement quand un fichier WAL est fermé. L'option `--synchronous` doit être ajoutée pour que les données WAL soient écrites en temps réel. Comme pg_receivewal n'applique pas les WAL, vous ne devez pas lui permettre de devenir un standby synchrone quand synchronous_commit vaut `remote_apply`. Dans le cas contraire, ce sera un standby qui ne réussit jamais à rattraper son retard et causera le blocage des validations des transactions. Pour éviter ceci, vous devez soit configurer une valeur appropriée pour synchronous_standby_names, soit spécifier une valeur pour le paramètre `application_name` pour pg_receivexlog qui ne correspond pas, soit modifier la valeur du paramètre `synchronous_commit` à quelque chose d'autres que `remote_apply`.

Le journal de transactions est envoyé via une connexion PostgreSQL traditionnelle, et utilise le protocole de réplication. La connexion doit être créée avec un compte superutilisateur ou utilisateur disposant des droits `REPLICATION` (voir Section 22.2) et le fichier `pg_hba.conf` doit permettre la connexion de réplication. Le serveur doit aussi être configuré avec une valeur suffisamment haute du paramètre max_wal_senders pour laisser au moins une session disponible pour le flux.

Le point de démarrage du flux des journaux de transactions est calculé quand pg_receivewal démarre :

1. Tout d'abord, il parcourt le répertoire où les segments de journaux de transactions sont écrits et trouver le segment terminé le plus récent pour l'utiliser comme point de départ du prochain segment.

2. Si un point de démarrage ne peut pas être calculé avec la méthode précédente et si un slot de réplication est utilisé, une commande `READ_REPLICATION_SLOT` supplémentaire est exécutée pour récupérer le `restart_lsn` du slot et l'utiliser comme point de démarrage. Cette option est seulement disponible quand les journaux de transactions en flux proviennent de PostgreSQL 15 ou ultérieur.

3. Si un point de démarrage ne peut pas être calculé avec la méthode précédente, l'emplacement de vidage des journaux de transactions est utilisé comme indiqué par le serveur à partir d'une commande `IDENTIFY_SYSTEM`.

Si la connexion est perdue ou si elle ne peux pas être établie initialement, via une erreur non fatale, pg_receivewal essaiera à nouveau indéfiniment, et rétablira le flux dès que possible. Pour éviter ce comportement, utilisez le paramètre `-n`.

En l'absence d'erreurs fatales, pg_receivewal s'exécutera jusqu'à recevoir le signal SIGINT (**Control**+**C**).

Options

`-D` *répertoire*
`--directory=`*répertoire*

Répertoire dans lequel écrire le résultat.

Ce paramètre est obligatoire.

`-E` *lsn*
`--endpos=`*lsn*

Arrête automatiquement la réplication et quitte avec le code de sortie standard 0 une fois arrivé au LSN indiqué.

S'il n'y a pas d'enregistrement avec un LSN strictement identique à *lsn*, l'enregistrement sera traité.

`--if-not-exists`

Ne renvoie pas une erreur quand `--create-slot` est indiqué et qu'un slot de même nom existe déjà.

`-n`
`--no-loop`

N'effectue pas de nouvelle tentative en cas d'erreur à la connexion. À la place, le programme s'arrête en retournant une erreur.

`--no-sync`

Cette option force `pg_receivewal` à ne pas imposer de vider les données WAL sur disque. C'est plus rapide mais cela signifie aussi qu'un crash du système d'exploitation peut laisser les segments WAL corrompus. En règle général, cette option est utile pour des tests mais ne devrait pas être utilisée lors de l'archivage de journaux de transactions sur un système en production.

Cette option est incompatible avec `--synchronous`.

`-s` *intervalle*
`--status-interval=`*intervalle*

Spécifie le rythme en secondes de l'envoi des paquets au serveur informant de l'état en cours. Ceci permet une supervision plus simple du progrès à partir du serveur. Une valeur de zéro désactive complètement la mise à jour périodique du statut, bien qu'une mise à jour sera toujours envoyée si elle est réclamée par le serveur pour éviter la déconnexion après un certain délai. La valeur par défaut est de 10 secondes.

`-S` *slotname* `--slot=`*nom_slot*

Requiert l'utilisation d'un slot de réplication existant avec pg_receivewal (voir Section 27.2.6). Quand cette option est utilisée, pg_receivewal renverra une position de vidage au serveur, indiquant quand chaque segment a été synchronisé sur disque. Cela permet au serveur de supprimer ce segment s'il n'est pas utile ailleurs.

Quand le client de réplication pg_receivewal est configuré sur le serveur comme un standby synchrone, l'utilisation d'un slot de réplication renverra la position de vidage sur disque du serveur,

mais seulement lors de la fermeture d'un fichier WAL. De ce fait, cette configuration entraînera que les transactions sur le primaire attendront un long moment, ce qui aura pour effet de ne pas fonctionner de manière satisfaisante. L'option `--synchronous` (voir ci- dessous) doit être ajoutée pour que cela fonctionne correctement.

`--synchronous`

Vide les données WAL sur disque dès leur réception. De plus, envoie un paquet de statut au serveur immédiatement après le vidage, quelque soit la configuration de l'option `--status-interval`.

Cette option doit être utilisée si le client de réplication de pg_receivewal est configuré en tant que serveur standby synchrone pour s'assurer que le retour est renvoyé à temps au serveur principal.

`-v`
`--verbose`

Active le mode verbeux.

`-Z` *`niveau`*
`-Z` *`méthode`*`[:`*`détail`*`]`
`--compress=`*`niveau`*
`--compress=`*`méthode`*`[:`*`détail`*`]`

Active la compression des journaux de transactions.

La méthode de compression peut être configurée à `gzip`, `lz4` (si PostgreSQL a été configuré avec `--with-lz4`) ou `none` pour aucune compression. Une chaîne de détails sur la compression peut être indiquée en option. Si la chaîne est un entier, elle précise le niveau de compression. Sinon, cela pourrait être une liste d'éléments séparés par des virgules, chaque élément devant être de la forme `motclé` ou `motclé=valeur`. Actuellement, le seul mot clé supporté est `level`.

Si aucun niveau de compression n'est indiqué, le niveau de compression par défaut sera utilisé. Si seul un niveau est indiqué sans mention d'un algorithme, la compression `gzip` sera utilisée si le niveau est supérieur à 0, et aucune compression ne sera utilisée si le niveau vaut 0.

Le suffixe `.gz` sera automatiquement ajouté à tous les noms de fichier lors de l'utilisation de `gzip`, et le suffixe `.lz4` sera ajouté lors de l'utilisation de `lz4`.

Les options en ligne de commande qui suivent permettent de paramétrer la connexion à la base de données.

`-d` *`connstr`*
`--dbname=`*`connstr`*

Spécifie les paramètres utilisés pour se connecter au serveur, sous la forme d'une chaîne de connexion. Elles surchargent toutes les options en ligne de commande conflictuelles.

Cette option est nommée `--dbname` par cohérence avec les autres applications clients mais, comme pg_receivewal ne se connecte à aucune base de données en particulier dans l'instance, le nom de la base dans la chaîne de connexion sera ignoré.

`-h` *`hôte`*
`--host=`*`hôte`*

Spécifie le nom de l'hôte de la machine sur lequel le serveur s'exécute. Si la valeur commence par un slash, il est utilisé comme le répertoire de la socket du domaine Unix (IPC). La valeur par défaut est issue de la variable d'environnement `PGHOST`, si elle est définie, sinon une connexion à une socket du domaine Unix est tentée.

`-p` *`port`*
`--port=`*`port`*

Spécifie le port TCP ou l'extension du fichier de la socket du domaine Unix sur lequel le serveur va écouter les connexions. La valeur par défaut est issue de la variable d'environnement `PGPORT`, si elle est définie, ou d'une valeur définie lors de la compilation.

`-U` *`nom_utilisateur`*
`--username=`*`nom_utilisateur`*

L'utilisateur avec lequel se connecter.

`-w`
`--no-password`

Ne demande pas la saisie d'un mot de passe. Si le serveur nécessite un mot de passe d'authentification et qu'aucun mot de passe n'est disponible par d'autre biais comme le fichier `.pgpass`, la tentative de connexion échouera. Cette option peut être utile dans des batchs où aucun utilisateur ne pourra saisir un mot de passe.

`-W`
`--password`

Oblige pg_receivewal à demander un mot de passe avant de se connecter à la base.

Cette option n'est pas indispensable car pg_receivewal demandera automatiquement un mot de passe si le serveur nécessite une authentification par mot de passe. Cependant, pg_receivewal perdra une tentative de connexion avant de savoir si le serveur nécessite un mot de passe. Dans certain cas, il est possible d'ajouter l'option `-W` pour éviter une tentative de connexion superflue.

pg_receivewal peut réaliser une des deux actions suivantes pour contrôler les slots de réplication physiques :

`--create-slot`

Crée un slot de réplication physique avec le nom spécifié par l'option `--slot`, puis quitte.

`--drop-slot`

Supprime le slot de réplication dont le nom est spécifié par l'option `--slot`, puis quitte.

D'autres options sont aussi disponibles :

`-V`
`--version`

Affiche la version de pg_receivewal et quitte le programme.

`-?`
`--help`

Affiche l'aide concernant les arguments en ligne de commande de pg_receivewal, et quitte le programme.

Code de sortie

pg_receivewal sortira avec un code 0 quand il est terminé par le signal SIGINT. (C'est la façon normale de l'arrêter, d'où le fait qu'il ne s'agit pas d'une erreur.) Pour les erreurs fatales ou pour tout autre signal, le code de sortie sera différent de zéro.

Environnement

Cet outil, comme la plupart des autres outils PostgreSQL, utilise les variables d'environnement supportées par libpq (voir Section 34.15).

La variable d'environnement `PG_COLOR` indique s'il faut utiliser les couleurs dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Notes

Lorsque vous utilisez pg_receivewal à la place de archive_command ou archive_library comme méthode principale de sauvegarde des WAL, il est fortement recommandé d'utiliser les slots de réplication. Dans le cas contraire, le serveur est libre de recycler ou supprimer les fichiers des journaux de transactions avant qu'ils ne soient sauvegardés car il n'a aucune information, provenant soit de archive_command, soit de archive_library soit des slots de réplication, sur la quantité de WAL déjà archivée. Néanmoins, notez qu'un slot de réplication remplira l'espace disque du serveur si le receveur n'arrive pas à suivre le rythme de récupération des données WAL.

pg_receivewal conservera les droits sur le groupe pour les fichiers WAL reçus si les droits du groupe sont activés sur l'instance source.

Exemples

Pour suivre le flux des journaux de transactions du serveur `mon-serveur-de-donnees` et les stocker dans le répertoire local `/usr/local/pgsql/archive` :

```
$ pg_receivewal -h mon-serveur-de-donnees -D /usr/local/pgsql/
archive
```

Voir aussi

pg_basebackup

pg_recvlogical

pg_recvlogical — contrôle les flux de décodage logique de PostgreSQL

Synopsis

`pg_recvlogical` [*option*...]

Description

`pg_recvlogical` contrôle des slots de réplication pour le décodage logique et envoie les données par flux depuis ces slots de réplication.

Il crée une connexion en mode réplication, et est donc sujet aux même contraintes que pg_receivewal, en plus de celles de la réplication logique (voir Chapitre 49).

`pg_recvlogical` n'a pas d'équivalent aux modes d'interface SQL de décodage logique peek et get. Il envoie des confirmation de rejeu pour les données de manière paresseuse quand il les reçoit et lors d'un arrêt propre. Pour examiner les données en attente d'un slot sans les consommer, utilisez `pg_logical_slot_peek_changes`.

Quand pg_recvlogical reçoit un signal SIGHUP, il ferme le fichier de sortie courant et en ouvre un nouveau en utilisant le nom de fichier indiqué par l'option `--file`. Ceci nous permet de changer le fichier en sortie en renommant tout d'abord le fichier en cours puis en envoyant un signal SIGHUP à pg_recvlogical.

Options

Au moins une des options suivantes doit être indiquée pour sélectionner une action :

`--create-slot`

Crée un nouveau slot de réplication avec le nom spécifié avec `--slot`, utilisant le plugin de sortie spécifié avec `--plugin`, pour la base de données spécifiée par `--dbname`.

L'option `--two-phase` peut être précisée avec `--create-slot` pour activer le décodage des transactions préparées.

`--drop-slot`

Supprime le slot de réplication dont le nom est spécifié avec l'option `--slot`, puis quitte.

`--start`

Commence le transfert des modifications à partir du slot de réplication spécifié par l'option `--slot`, et continue jusqu'à être arrêté par un signal. Si le flux de modifications côté serveur se termine avec un arrêt du serveur ou une déconnexion, tente de nouveau dans une boucle, sauf si l'option `--no-loop` est ajoutée.

Le format du flux est déterminé par le plugin en sortie indiqué lors de la création du slot.

La connexion doit se faire sur la même base de données que celle utilisée pour créer le slot.

Les actions `--create-slot` et `--start` peuvent être utilisées ensemble. `--drop-slot` ne peut pas être combinée avec une autre action.

L'option de ligne de commande suivante contrôle l'emplacement et le format de sortie ainsi que les autres comportements de la réplication :

`-f` *`nom_fichier`*
`--file=`*`nom_fichier`*

Écrit les données de transactions reçues et décodées dans ce fichier. Utiliser `-` pour la sortie standard (stdout).

`-F` *`interval_secondes`*
`--fsync-interval=`*`interval_secondes`*

Précise la fréquence des appels à `fsync()` par pg_recvlogical pour s'assurer que le fichier en sortie est à coup sûr sur disque.

De temps en temps, le serveur demande au client de réaliser les écritures et de rapporter sa position au serveur. Ce paramètre permet d'aller au-delà, pour réaliser des écritures plus fréquentes.

Indiquer un intervalle de `0` désactive tous les appels à `fsync()`. Le serveur est toujours informé de la progression. Dans ce cas, des données peuvent être perdues en cas de crash.

`-I` *`lsn`*
`--startpos=`*`lsn`*

Dans le mode `--start`, la réplication commence à la position LSN désignée. Pour les détails de son effet, voir la documentation dans Chapitre 49 et Section 55.4. Ignoré dans les autres modes.

`-E` *`lsn`*
`--endpos=`*`lsn`*

Dans le mode `--start`, l'outil arrête automatiquement la réplication et quitte avec un code retour normal 0 quand il atteint le LSN spécifié. S'il est spécifié et que le mode `--start` n'est pas demandé, une erreur est levée.

S'il y a un enregistrement avec le LSN strictement égal à *`lsn`*, l'enregistrement sera produit.

L'option `--endpos` n'est pas au courant des limites de transaction et pourrait tronquer en partie la sortie d'une transaction. Toute transaction partiellement produite ne sera pas consommée et sera rejouée de nouveau quand le slot sera de nouveau lu. Les messages individuels ne sont jamais tronqués.

`--if-not-exists`

Ne renvoie pas une erreur quand `--create-slot` est spécifié et qu'un slot de ce nom existe déjà.

`-n`
`--no-loop`

Quand la connexion au serveur est perdue, ne pas tenter de nouveau dans une boucle, mais quitte simplement.

`-o` *`nom`*`[=`*`valeur`*`]`
`--option=`*`nom`*`[=`*`valeur`*`]`

Passe l'option *`nom`* au plugin en sortie avec la *`valeur`* si elle est spécifiée. Des options existent mais leurs effets dépendent du plugin utilisé en sortie.

`-P` *`plugin`*
`--plugin=`*`plugin`*

Lors de la création du slot, utiliser la sortie de plugin de décodage spécifiée. Voir Chapitre 49. Cette option n'a pas d'effet si le slot existe déjà.

`-s` *`intervalle_en_seconde`*
`--status-interval=`*`intervalle_en_seconde`*

Cette option a le même effet que l'option du même nom dans pg_receivewal. Voir la description à cet endroit.

`-S` *`nom_slot`*
`--slot=`*`nom_slot`*

Dans le mode `--start`, utilise le slot de réplication logique existant nommé *`nom_slot`*. Dans le mode `--create-slot`, créer le slot de réplication avec ce nom. Dans le mode `--drop-slot`, supprime le slot de ce nom.

`-t`
`--two-phase`

Active le décodage des transactions préparées. Cette option devrait uniquement être indiquée avec `--create-slot`

`-v`
`--verbose`

Active le mode verbeux.

Les options suivantes en ligne de commande contrôlent les paramètres de connexion à la base de données.

`-d` *`nom_base`*
`--dbname=`*`nom_base`*

La base de données où se connecter. Voir la description des actions de sa signification. Ce nom de base peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront toutes les options en ligne de commande conflictuelles. La valeur par défaut est le nom de l'utilisateur.

`-h` *`alias-ou-ip`*
`--host=`*`alias-ou-ip`*

Indique le nom d'hôte du serveur. Si la valeur commence avec un slash, elle est utilisée comme nom du répertoire pour le socket de domaine Unix. La valeur par défaut est récupérée de la variable d'environnement `PGHOST`. Si cette dernière n'est pas configurée, une connexion par socket de domaine Unix est tentée.

`-p` *`port`*
`--port=`*`port`*

Indique le port TCP ou l'extension du fichier de socket de domaine Unix, sur lequel le serveur écoute les connexions entrantes. La valeur par défaut correspond à la valeur de la variable d'environnement `PGPORT`. Si cette variable n'est pas configurée, une valeur compilée est prise en compte.

`-U` *`nom_utilisateur`*
`--username=`*`nom_utilisateur`*

Le nom d'utilisateur utilisé pour la connexion. Sa valeur par défaut est le nom de l'utilisateur du système d'exploitation.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur requiert une authentification par mot de passe et qu'un mot de passe n'est pas disponible par d'autres moyens tels que le fichier `.pgpass`, la

tentative de connexion échouera. Cette option peut être utile dans les jobs programmés et dans les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force pg_recvlogical à demander un mot de passe avant de se connecter à une base de données.

Cette option n'est jamais obligatoire, car pg_recvlogical demandera automatiquement un mot de passe si le serveur requiert une authentification par mot de passe. Néanmoins, pg_recvlogical gaspillera une tentative de connexion pour trouver que le serveur a besoin d'un mot de passe. Dans certains cas, il est préférable d'utiliser l'option `-W` pour éviter la tentative de connexion supplémentaire.

Les options supplémentaires suivantes sont disponibles :

`-V`
`--version`

Affiche la version de pg_recvlogical, puis quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_recvlogical, puis quitte.

Environnement

Cet outil, comme la plupart des autres outils PostgreSQL, utilise les variables d'environnement supportées par libpq (voir Section 34.15).

La variable d'environnement `PG_COLOR` indique s'il faut utiliser les couleurs dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Notes

pg_recvlogical conservera les droits du groupe sur les fichiers en sortie si les droits du groupe sont activés sur l'instance source.

Exemples

Voir Section 49.1 pour un exemple.

Voir aussi

pg_receivewal

pg_restore

pg_restore — restaure une base de données PostgreSQL à partir d'un fichier d'archive créé par pg_dump

Synopsis

`pg_restore` [*option_connexion*...] [*option*...] [*nom_fichier*]

Description

pg_restore est un outil pour restaurer une base de données PostgreSQL à partir d'une archive créée par pg_dump dans un des formats non textuel. Il lance les commandes nécessaires pour reconstruire la base de données dans l'état où elle était au moment de sa sauvegarde. Les fichiers d'archive permettent aussi à pg_restore d'être sélectif sur ce qui est restauré ou même de réordonner les éléments à restaurer. Les fichiers d'archive sont conçus pour être portables entre les architectures.

pg_restore peut opérer dans deux modes. Si un nom de base de données est spécifié, pg_restore se connecte à cette base de données et restaure le contenu de l'archive directement dans la base de données. Sinon, un script contenant les commandes SQL nécessaires pour reconstruire la base de données est créé et écrit dans un fichier ou sur la sortie standard. La sortie du script est équivalente à celles créées par le format en texte plein de pg_dump. Quelques-unes des options contrôlant la sortie sont du coup analogues aux options de pg_dump.

De toute évidence, pg_restore ne peut pas restaurer l'information qui ne se trouve pas dans le fichier d'archive. Par exemple, si l'archive a été réalisée en utilisant l'option donnant les « données sauvegardées par des commandes `INSERT` », pg_restore ne sera pas capable de charger les données en utilisant des instructions `COPY`.

> **Avertissement**
>
> Restaurer une sauvegarde fait que la destination exécute du code arbitraire. Les sauvegardes et restaurations partielles ne limitent pas cela. S'il n'est pas possible de faire confiance aux superutilisateurs de la source, les requêtes SQL de restauration doivent être inspectées avant d'être exécutées. Les sauvegardes binaires peuvent être inspectées en utilisant l'option `--file` de pg_restore. Notez que le client exécutant la sauvegarde et la restauration n'a pas besoin de faire confiance aux superutilisateurs source et destination.

Options

pg_restore accepte les arguments suivants en ligne de commande.

nom_fichier

Spécifie l'emplacement du fichier d'archive (ou du répertoire pour une archive au format « directory ») à restaurer. S'il n'est pas spécifié, l'entrée standard est utilisée.

`-a`
`--data-only`

Restaure seulement les données, pas les schémas (définitions des données). Les données des tables, les Large Objects, et les valeurs des séquences sont restaurées si elles sont présentes dans l'archive.

Cette option est similaire à `--section=data` mais, pour des raisons historiques, elle n'est pas identique.

`-c`
`--clean`

Avant de restaurer les objets de la base, lance des commandes SQL `DROP` pour supprimer tous les objets à restaurer. Cette option est utile pour surcharger une base existante. Si un des objets n'existe pas dans la base de destination, des messages d'erreurs, à ignorer, seront renvoyées sauf si l'option `--if-exists` est aussi indiquée.

`-C`
`--create`

Crée la base de données avant de la restaurer. Si l'option `--clean` est aussi indiquée, supprime puis crée de nouveau la base de données cible avant de s'y connecter.

Avec `--create`, pg_restore restaure aussi le commentaire de la base de données, s'il a été configuré, et toutes variables de configuration spécifiques à cette base, autrement dit toutes les commandes `ALTER DATABASE ... SET ...` et `ALTER ROLE ... IN DATABASE ... SET ...` qui mentionnent cette base. Les droits d'accès à la base elle-même sont aussi restaurés sauf si `--no-acl` est spécifié.

Quand cette option est utilisée, la base de données nommée via l'option `-d` est utilisée seulement pour exécuter les commandes `DROP DATABASE` et `CREATE DATABASE`. Toutes les données sont restaurées dans la base dont le nom se trouve dans l'archive.

`-d` *`nom_base`*
`--dbname=`*`nom_base`*

Se connecte à la base de données *`nom_base`* et restaure directement dans la base de données. Ce nom de base peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront toutes les options en ligne de commande conflictuelles.

`-e`
`--exit-on-error`

Quitte si une erreur est rencontrée lors de l'envoi des commandes SQL à la base de données. La valeur par défaut est de continuer et d'afficher le nombre d'erreurs à la fin de la restauration.

`-f` *`nom_fichier`*
`--file=`*`filename`*

Spécifie le fichier en sortie pour le script généré ou pour la liste lorsqu'elle est utilisée avec `-l`. Utilisez `-` pour stdout.

`-F` *`format`*
`--format=`*`format`*

Spécifie le format de l'archive. Il n'est pas nécessaire de le spécifier car pg_restore détermine le format automatiquement. Si spécifié, il peut être un des suivants :

`c`
`custom`

L'archive est dans le format personnalisé de pg_dump.

`d`
`directory`

L'archive est un répertoire (*directory*).

`t`
`tar`

L'archive est une archive `tar`.

`-I` *`index`*
`--index=`*`index`*

Restaure uniquement la définition des index nommés. Plusieurs index peuvent être donnés en utilisant autant de fois l'option `-I`.

`-j` *`nombre-de-jobs`*
`--jobs=`*`nombre-de-jobs`*

Exécute les parties les plus consommatrices en temps de pg_restore -- celles des chargements de données, créations d'index et créations de contraintes -- en parallèle, utilisant jusqu'à *`nombre-de-jobs`* sessions parallèles. Cette option peut réduire de beaucoup le temps pour restaurer une grosse base de données pour un serveur fonctionnant sur une machine multi-processus. Cette option est ignorée lors de la création d'un script, plutôt que lors d'une connexion directe à un serveur de bases de données.

Chaque job est un processus ou un thread, suivant le système d'exploitation, et utilise une connexion séparée au serveur.

La valeur optimale pour cette option dépend de la configuration matérielle du serveur, du client et du réseau. Les facteurs incluent le nombre de cœurs CPU et la configuration disque. Un bon moyen pour commencer est le nombre de cœurs CPU du serveur, mais une valeur plus grande que ça peut amener des temps de restauration encore meilleurs dans de nombreux cas. Bien sûr, les valeurs trop hautes apporteront des performances en baisse.

Seuls les formats d'archivage personnalisé et répertoire sont supportés avec cette option. Le fichier en entrée doit être un fichier standard (pas un tube, pas l'entrée standard). De plus, plusieurs jobs ne peuvent pas être utilisés ensemble si vous voulez l'option `--single-transaction`.

`-l`
`--list`

Liste le contenu de l'archive. Le résultat de cette opération peut être utilisé en entrée de l'option `-L`. Notez que, si vous utilisez des options de filtre telles que `-n` ou `-t` avec l'option `-l`, elles restreignent les éléments listés.

`-L` *`fichier_liste`*
`--use-list=`*`fichier_liste`*

Restaure seulement les objets qui sont listés dans le fichier *`fichier_liste`*, et les restaure dans l'ordre où elles apparaissent dans le fichier. Notez que, si des options de filtre comme `-n` et `-t` sont utilisées avec `-L`, elles ajouteront cette restriction aux éléments restaurés.

`fichier_liste` est normalement créé en éditant la sortie d'une précédente opération `-l`. Les lignes peuvent être déplacées ou supprimées, et peuvent aussi être mise en commentaire en ajoutant un point-virgule (`;`) au début de la ligne. Voir ci-dessous pour des exemples.

`-n` *`nom_schema`*
`--schema=`*`nom_schema`*

Restaure seulement les objets qui sont dans le schéma nommé. Plusieurs schémas peuvent être donnés en utilisant autant de fois l'option `-n`. Elle peut être combinée avec l'option `-t` pour ne restaurer qu'une seule table.

`-N` *`nom_schema`*
`--exclude-schema=`*`nom_schema`*

Ne pas restaurer les objets qui sont dans le schéma nommé. Plusieurs schémas à exclure peuvent être spécifiés grâce à de multiples commutateurs `-N`.

Si les commutateurs `-n` et `-N` sont tous deux présents pour le même schéma, le commutateur `-N` sera prépondérant et le schéma exclu.

`-O`
`--no-owner`

Ne pas donner les commandes initialisant les propriétaires des objets pour correspondre à la base de données originale. Par défaut, pg_restore lance des instructions `ALTER OWNER` ou `SET SESSION AUTHORIZATION` pour configurer le propriétaire des éléments du schéma créé. Ces instructions échouent sauf si la connexion initiale à la base de données est réalisée par un superutilisateur (ou le même utilisateur que le propriétaire des objets du script). Avec `-O`, tout nom d'utilisateur peut être utilisé pour la connexion initiale et cet utilisateur est le propriétaire des objets créés.

`-P` *`nom_fonction(argtype [, ...])`*
`--function=`*`nom_fonction(argtype [, ...])`*

Restaure seulement la fonction nommée. Faites attention à épeler le nom de la fonction et les arguments exactement comme ils apparaissent dans la table des matières du fichier de sauvegarde. Plusieurs fonctions peuvent être données en utilisant autant de fois l'option `-P`.

`-r`
`--no-reconnect`

Cette option est obsolète mais est toujours acceptée pour des raisons de compatibilité ascendante.

`-s`
`--schema-only`

Restaure seulement le schéma (autrement dit, la définition des données), mais pas les données, à condition que cette définition est présente dans l'archive.

Cette option est l'inverse de `--data-only`. Elle est similaire, mais pas identique (pour des raisons historiques), à `--section=pre-data --section=post-data`.

(Ne pas la confondre avec l'option `--schema` qui utilise le mot « schema » dans un contexte différent.)

`-S` *`nom_utilisateur`*
`--superuser=`*`nom_utilisateur`*

Spécifie le nom d'utilisateur du superutilisateur à utiliser pour désactiver les triggers. Ceci est seulement nécessaire si `--disable-triggers` est utilisé.

`-t` *`table`*
`--table=`*`table`*

Restaure la définition et/ou les données de la table nommée uniquement. Dans ce cadre, « table » inclut les vues, les vues matérialisées, les séquences et les tables distantes. Plusieurs tables peuvent être sélectionnées en ajoutant plusieurs options `-t`. Cette option peut être combinée avec l'option `-n` pour indiquer les tables d'un schéma particulier.

> **Note**
>
> Quand l'option `-t` est indiquée, pg_restore ne tente pas de restaurer les autres objets de la base de données qui pourraient être liés à la table sélectionnée. De ce fait, il n'y a aucune garantie qu'une restauration d'une table spécifique dans une base propre réussira.

> **Note**
>
> Cette option ne se comporte pas de la même façon que l'option `-t` de pg_dump. Il n'existe pas actuellement de support pour la recherche de motifs dans pg_restore. De plus, vous ne

pouvez pas inclure un nom de schéma dans `-t`. Et, pendant que l'option `-t` de pg_dump sauvegardera aussi les objets liés (comme les index) des tables sélectionnées, l'option `-t` de pg_restore n'incluera pas les objets liés.

Note

Dans les versions de PostgreSQL antérieures à la 9.6, cette option correspondant seulement aux tables, pas aux autres types de relation.

`-T` *`trigger`*
`--trigger=`*`trigger`*

Restaure uniquement le trigger nommé. Plusieurs triggers peuvent être donnés en utilisant autant de fois l'option `-T`..

`-v`
`--verbose`

Spécifie le mode verbeux. Ceci aura pour conséquence que pg_restore affichera des commentaires détaillés sur les objets, ainsi que les heures de début et de fin dans le fichier en sortie, et des messages de progression sur la sortie des erreurs. Répéter cette option cause l'apparition de messages supplémentaires de débuggage sur la sortie des erreurs.

`-V`
`--version`

Affiche la version de pg_restore, puis quitte.

`-x`
`--no-privileges`
`--no-acl`

Empêche la restauration des droits d'accès (commandes grant/revoke).

`-1`
`--single-transaction`

Exécute la restauration en une seule transaction (autrement dit, toutes les commandes de restauration sont placées entre un `BEGIN` et un `COMMIT`). Ceci assure l'utilisateur que soit toutes les commandes réussissent, soit aucun changement n'est appliqué. Cette option implique `--exit-on-error`.

`--disable-triggers`

Cette option n'est pertinente que lors d'une restauration des données seules. Elle demande à pg_restore d'exécuter des commandes pour désactiver temporairement les triggers sur les tables cibles pendant que les données sont rechargées. Utilisez ceci si vous avez des vérifications d'intégrité référentielle sur les tables que vous ne voulez pas appeler lors du rechargement des données.

Actuellement, les commandes émises pour `--disable-triggers` doivent être exécutées par un superutilisateur. Donc, vous devriez aussi spécifier un nom de superutilisateur avec `-S` ou, de préférence, lancer pg_restore en tant que superutilisateur PostgreSQL.

`--enable-row-security`

Cette option n'est adéquate que lors de la restauration du contenu du table disposant de l'option RLS. Par défaut, pg_restore configurera row_security à off, pour s'assurer que toutes les données

sont restaurées dans la table. Si l'utilisateur n'a pas les droits nécessaires pour contourner la sécurité au niveau ligne, alors une erreur est levée. Ce paramètre demande à pg_restore de configurer row_security à on, permettant à l'utilisateur d'essayer de restaurer le contenu de la table avec la sécurité au niveau ligne activée. Ceci pourrait échouer si l'utilisateur n'a pas le droit d'insérer des lignes dans la table.

Notez que cette option requiert aussi actuellement que la sauvegarde soit au format `INSERT` car `COPY FROM` n'est pas supportée par la sécurité au niveau ligne.

`--if-exists`

Utilise des commandes `DROP ... IF EXISTS` pour supprimer des objets dans le mode `--clean`. Cela permet de supprimer les erreurs « does not exist » qui seraient sinon renvoyées. Cette option n'est pas valide sauf si `--clean` est aussi indiquée.

`--no-comments`

Ne génère pas les commande de restauration des commentaires même si l'archive les contient.

`--no-data-for-failed-tables`

Par défaut, les données de la table sont restaurées même si la commande de création de cette table a échoué (par exemple parce qu'elle existe déjà). Avec cette option, les données de cette table seront ignorées. Ce comportement est utile si la base cible contient déjà des données pour cette table. Par exemple, les tables supplémentaires des extensions de PostgreSQL comme PostGIS pourraient avoir déjà été créées et remplies sur la base cible ; indiquer cette option empêche l'ajout de données dupliquées ou obsolètes.

Cette option est seulement efficace lors de la restauration directe d'une base, pas lors de la réalisation d'une sortie de script SQL.

`--no-publications`

Ne pas afficher les commandes pour restaurer les publications, même si l'archive les contient.

`--no-security-labels`

Ne récupère pas les commandes de restauration des labels de sécurité, même si l'archive les contient.

`--no-subscriptions`

Ne pas afficher les commandes pour restaurer les souscriptions, même si l'archive les contient.

`--no-table-access-method`

Ne pas afficher les commandes pour sélectionner les méthodes d'accès aux tables. Avec cette option, tous les objets seront créés avec la méthode d'accès par défaut lors de la restauration.

`--no-tablespaces`

Ne sélectionne pas les tablespaces. Avec cette option, tous les objets seront créés dans le tablespace par défaut lors de la restauration.

`--restrict-key=`*`restrict_key`*

Utilise la chaîne fournie comme clé de `\restrict` dans psql dans la sortie de la sauvegarde. Cela peut seulement être indiqué pour les sauvegardes textes, autrement dit quand l'option `--file` est utilisée. Si aucune clé de restriction n'est indiquée, pg_restore en générera une aléatoirement si nécessaire. Les clés doivent contenir uniquement des caractères alphanumériques.

Cette option a principalement pour but des tests et d'autres scénarios nécessitant une sortie répétable (par exemple pour comparer les fichiers de sauvegarde). Ce n'est pas recommandé pour une utilisation standard, car un serveur malicieux avec une connaissance de la clé pourrait injecter un code arbitraire qui sera exécuté sur la machine qui lance psql avec la sauvegarde en sortie.

`--section=`*`nom_section`*

Restaure seulement la section nommée. Le nom de la section peut être `pre-data`, `data` ou `post-data`. Cette option peut être spécifiée plus d'une fois pour sélectionner plusieurs sections. La valeur par défaut est toutes les sections.

La section data contient toutes les données des tables ainsi que la définition des Large Objects. Les éléments post-data consistent en la définition des index, triggers, règles et contraintes (autres que les contraintes de vérification). Les éléments pre-data consistent en tous les autres éléments de définition.

`--strict-names`

Requiert que chaque qualificateur de schéma (`-n` / `--schema`) et table (`-t` / `--table`) correspond à au moins un schéma/table dans le fichier de sauvegarde.

`--use-set-session-authorization`

Affiche les commandes `SET SESSION AUTHORIZATION` du standard SQL à la place des commandes `ALTER OWNER` pour déterminer le propriétaire de l'objet. Ceci rend la sauvegarde plus compatible avec les standards mais, suivant l'historique des objets dans la sauvegarde, pourrait ne pas restaurer correctement.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_restore, puis quitte.

pg_restore accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h `*`hôte`*
`--host=`*`hôte`*

Spécifie le nom d'hôte de la machine sur lequel le serveur est en cours d'exécution. Si la valeur commence par un slash, elle est utilisée comme répertoire du socket de domaine Unix. La valeur par défaut est prise dans la variable d'environnement `PGHOST`, si elle est initialisée, sinon une connexion socket de domaine Unix est tentée.

`-p `*`port`*
`--port=`*`port`*

Spécifie le port TCP ou l'extension du fichier socket de domaine Unix sur lequel le serveur écoute les connexions. Par défaut, l'outil utilise la variable d'environnement `PGPORT`, si elle est configurée, sinon il utilise la valeur indiquée à la compilation.

`-U `*`nom_utilisateur`*
`--username=`*`nom_utilisateur`*

Se connecte en tant que cet utilisateur

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la

tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force pg_restore à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car pg_restore demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, pg_restore perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`--role=`*`nom_rôle`*

Indique un nom de rôle utilisé pour la restauration. Cette option fait que pg_restore exécute un `SET ROLE` *`nom_rôle`* après connexion à la base de données. C'est utile quand l'utilisateur authentifié (indiqué par l'option `-U`) n'a pas les droits demandés par pg_restore, mais peut devenir le rôle qui a les droits requis. Certaines installations ont une politique contre la connexion en superutilisateur directement, et utilisent cette option pour permettre aux restaurations de se faire sans violer cette règle.

Environnement

`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Paramètres de connexion par défaut

`PG_COLOR`

Indique s'il faut utiliser les couleurs dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 34.15). Néanmoins, il ne lit pas la variable `PGDATABASE` quand le nom d'une base n'est pas fournie.

Diagnostics

Quand une connexion directe à la base de données est spécifiée avec l'option `-d`, pg_restore exécute en interne des instructions SQL. Si vous avez des problèmes en exécutant pg_restore, assurez-vous d'être capable de sélectionner des informations à partir de la base de données en utilisant, par exemple à partir de psql. De plus, tout paramètre de connexion par défaut et toute variable d'environnement utilisé par la bibliothèque libpq s'appliqueront.

Notes

Si votre installation dispose d'ajouts locaux à la base de données `template1`, faites attention à charger la sortie de pg_restore dans une base de données réellement vide ; sinon, vous avez des risques d'obtenir des erreurs dûes aux définitions dupliquées des objets ajoutés. Pour créer une base de données vide sans ajout local, copiez à partir de `template0`, et non pas de `template1`, par exemple :

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

Les limitations de pg_restore sont détaillées ci-dessous.

- Lors de la restauration des données dans une table pré-existante et que l'option `--disable-triggers` est utilisée, pg_restore émet des commandes pour désactiver les triggers sur les tables utilisateur avant d'insérer les données, puis émet les commandes pour les réactiver après l'insertion des données. Si la restauration est stoppée en plein milieu, les catalogues système pourraient être abandonnés dans le mauvais état.

- pg_restore ne peut pas restaurer les « large objects » de façon sélective, par exemple seulement ceux d'une table précisée. Si une archive contient des « large objects », alors tous les « large objects » seront restaurées (ou aucun s'ils sont exclus avec l'option `-L`, l'option `-t` ou encore d'autres options.

Voir aussi la documentation de pg_dump pour les détails sur les limitations de pg_dump.

Une fois la restauration terminée, il est conseillé de lancer `ANALYZE` sur chaque table restaurée de façon à ce que l'optimiseur dispose de statistiques utiles. Voir Section 25.1.3 et Section 25.1.6 pour plus d'informations.

Exemples

Supposons que nous avons sauvegardé une base nommée `ma_base` dans un fichier de sauvegarde au format personnalisé :

```
$ pg_dump -Fc ma_base > ma_base.dump
```

Pour supprimer la base et la re-créer à partir de la sauvegarde :

```
$ dropdb ma_base
$ pg_restore -C -d postgres ma_base.dump
```

La base nommée avec l'option `-d` peut être toute base de données existante dans le cluster ; pg_restore l'utilise seulement pour exécuter la commande `CREATE DATABASE` pour `ma_base`. Avec `-C`, les données sont toujours restaurées dans le nom de la base qui apparaît dans le fichier de sauvegarde.

Pour charger la sauvegarde dans une nouvelle base nommée `nouvelle_base` :

```
$ createdb -T template0 nouvelle_base
$ pg_restore -d nouvelle_base db.dump
```

Notez que nous n'utilisons pas `-C` et que nous nous sommes connectés directement sur la base à restaurer. De plus, notez que nous clonons la nouvelle base à partir de `template0` et non pas de `template1`, pour s'assurer qu'elle est vide.

Pour réordonner les éléments de la base de données, il est tout d'abord nécessaire de sauvegarder la table des matières de l'archive :

```
$ pg_restore -l ma_base.dump > ma_base.liste
```

Le fichier de liste consiste en un en-tête et d'une ligne par élément, par exemple :

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
```

```
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
319; 1247 25899 DOMAIN public domain0 pasha
```

Les points virgules commencent un commentaire et les numéros au début des lignes se réfèrent à l'ID d'archive interne affectée à chaque élément.

Les lignes dans le fichier peuvent être commentées, supprimées et réordonnées. Par exemple :

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

peut être utilisé en entrée de pg_restore et ne restaure que les éléments 10 et 6 dans cet ordre :

```
$ pg_restore -L mabase.liste mabase.fichier
```

Voir aussi

pg_dump, pg_dumpall, psql

pg_verifybackup

pg_verifybackup — Vérifie l'intégrité d'une sauvegarde de base d'une instance PostgreSQL

Synopsis

`pg_verifybackup` [*option*...]

Description

pg_verifybackup est utilisé pour contrôler l'intégrité d'une sauvegarde d'une instance effectuée avec `pg_basebackup`, grâce au manifeste de sauvegarde (`backup_manifest`) généré par le serveur au moment de la sauvegarde de base. La sauvegarde doit être stockée au format « plain » : un format « tar :» peut être vérifié après son extraction.

Notez que la validation effectuée par pg_verifybackup n'effectue pas toutes les vérifications effectuées par un serveur qui cherche à utiliser cette sauvegarde, et ne pourrait pas le faire. Même si vous utilisez cet outil, vous devez toujours procéder à des tests de restauration, vérifier que les bases de données résultantes fonctionnent comme prévu, et semblent contenir les bonnes données. Cependant, pg_verifybackup peut détecter de nombreux problèmes courants, dus au stockage ou une erreur de l'utilisateur.

La vérification de sauvegarde s'effectue en quatre étapes. Premièrement, `pg_verifybackup` lit le fichier `backup_manifest`. S'il n'existe pas, ne peut être lu, est malformé, ou échoue au contrôle face à sa propre somme de contrôle interne, `pg_verifybackup` se termine avec une erreur fatale.

Deuxièmement, `pg_verifybackup` tentera de vérifier que les fichiers de données en place sur le disque à ce moment sont exactement les mêmes que les fichiers que le serveur voulait envoyer, à quelques exceptions près listées plus bas. Les fichiers en trop ou manquants seront détectés, avec quelques exceptions. Cette étape ignore la présence ou la modification de `postgresql.auto.conf`, `standby.signal`, et `recovery.signal`, car il est prévisible que la procédure de sauvegarde les crée ou les modifie. Il n'y aura pas non plus de plainte à propos d'un fichier `backup_manifest` dans le répertoire cible, ou quoi que ce soit dans `pg_wal`, bien que ces fichiers ne soient pas listés dans le manifeste de sauvegarde. Seuls les fichiers sont contrôlés ; la présence ou l'absence des répertoires n'est pas vérifiée, ou indirectement : si un répertoire manque, tout fichier qu'il aurait dû contenir sera forcément porté manquant.

Ensuite, `pg_verifybackup` calcule les sommes de contrôle de tous les fichiers, les compare aux valeurs du manifeste, et émet des erreurs pour tous les fichiers où la somme de contrôle calculée ne correspond pas à celle du manifeste. Cette étape n'est pas effectuée pour les fichiers qui ont produit des erreurs à l'étape précédente, puisqu'il est déjà connu qu'ils sont problématiques. Les fichiers ignorés à l'étape précédente sont aussi ignorés à celle-ci.

Enfin, `pg_verifybackup` va utiliser le manifeste pour vérifier que les enregistrements des journaux de transaction nécessaires à la restauration de la sauvegarde sont présents, et peuvent être lus et analysés. Le `backup_manifest` contient des informations sur les enregistrements nécessaires des journaux. `pg_verifybackup` les utilise pour invoquer `pg_waldump` et analyser les journaux. L'option `--quiet` peut être utilisée pour que `pg_waldump` ne renvoie que les erreurs, sans générer d'autre sortie. Bien qu'à ce niveau les vérifications soient suffisantes pour détecter les problèmes évidents, comme des fichiers manquants ou une somme de contrôle incohérente, elles ne sont pas assez complètes pour détecter tous les problèmes possibles pouvant arriver lors d'une tentative de restauration. Par exemple, un bug du serveur générant des journaux de transaction avec les bonnes sommes de contrôle, mais spécifiant des opérations absurdes, ne peut être détecté par cette méthode.

Notez que s'il y a des fichiers WAL en plus, non nécessaires pour restaurer la sauvegarde, ils ne seront pas vérifiés par cet outil, bien qu'un appel séparé à `pg_waldump` permet de le faire. Notez

aussi que la vérification des journaux est spécifique à chaque version : vous devez utiliser la version de `pg_verifybackup`, et donc de `pg_waldump`, qui correspond à la sauvegarde à vérifier. En revanche, les tests d'intégrité des fichiers devraient fonctionner avec toutes les versions du serveur qui génèrent un fichier `backup_manifest`.

Options

pg_verifybackup accepte les arguments suivants en ligne de commande :

`-e`
`--exit-on-error`

Sort dès qu'un problème avec la sauvegarde est détecté. Si cette option n'est pas spécifiée, `pg_verifybackup` va continuer à vérifier la sauvegarde après la détection d'un problème, et rapportera tous les problèmes en tant qu'erreurs.

`-i` *path*
`--ignore=`*path*

Ignore le fichier ou répertoire spécifié (à exprimer avec un chemin relatif), lors de la comparaison des fichiers effectivement dans la sauvegarde avec ceux listés dans le fichier `backup_manifest`. Si un répertoire est spécifié, cette option affecte toute l'arborescence en-dessous. Les plaintes sur des fichiers excédentaires ou manquants, des différences de taille, ou des incohérences de sommes de contrôle, seront supprimées si le chemin relatif correspond à celui spécifié. Cette option peut être répétée plusieurs fois.

`-m` *path*
`--manifest-path=`*path*

Utiliser le fichier manifeste au chemin spécifié, plutôt que celui à la racine du répertoire de sauvegarde.

`-n`
`--no-parse-wal`

N'essaie pas d'analyser les journaux de transactions nécessaires à la restauration de la sauvegarde.

`-q`
`--quiet`

N'affiche rien si la sauvegarde est vérifiée avec succès.

`-s`
`--skip-checksums`

Ne vérifie pas les sommes de contrôle. La présence ou l'absence des fichiers et leur taille sera toujours contrôlée. C'est beaucoup plus rapide, car il n'y a pas besoin de lire les fichiers eux-mêmes.

`-w` *path*
`--wal-directory=`*path*

Essaie d'analyser les fichiers WAL stockés dans le répertoire indiqué, plutôt que ceux dans `pg_wal`. Ce peut être utile si la sauvegarde est stockée dans un emplacement différent des archives des journaux.

D'autres options sont disponibles :

`-V`
`--version`

Affiche la version de pg_verifybackup et sort.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_verifybackup, puis sort.

Exemples

Pour créer une sauvegarde de base du serveur sur `mydbserver`, et vérifier son intégrité :

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
$ pg_verifybackup /usr/local/pgsql/data
```

Pour créer une sauvegarde de base du serveur sur `mydbserver`, déplacer le manifeste hors du répertoire de la sauvegarde, et vérifier l'intégrité de la sauvegarde :

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/backup1234
$ mv /usr/local/pgsql/backup1234/backup_manifest /my/secure/
location/backup_manifest.1234
$ pg_verifybackup -m /my/secure/location/backup_manifest.1234 /usr/
local/pgsql/backup1234
```

Pour vérifier une sauvegarde, tout en ignorant un fichier ajouté manuellement au répertoire de sauvegarde, et aussi sauter la vérification des sommes de contrôle :

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
$ edit /usr/local/pgsql/data/note.to.self
$ pg_verifybackup --ignore=note.to.self --skip-checksums /usr/
local/pgsql/data
```

Voir aussi

pg_basebackup

psql

psql — terminal interactif PostgreSQL

Synopsis

`psql` [*option*...] [*nombase* [*nomutilisateur*]]

Description

psql est une interface en mode texte pour PostgreSQL. Il vous permet de saisir des requêtes de façon interactive, de les exécuter sur PostgreSQL et de voir les résultats de ces requêtes. Alternativement, les entrées peuvent être lues à partir d'un fichier ou à partir des arguments de la ligne de commande. De plus, il fournit un certain nombre de métacommandes et plusieurs fonctionnalités style shell pour faciliter l'écriture des scripts et automatiser une grande variété de tâches.

Options

`-a`
`--echo-all`

Affiche toutes les lignes non vides en entrée sur la sortie standard lorsqu'elles sont lues. (Ceci ne s'applique pas aux lignes lues de façon interactive.) C'est équivalent à initialiser la variable `ECHO` à `all`.

`-A`
`--no-align`

Bascule dans le mode d'affichage non aligné. (Le mode d'affichage par défaut est `aligned`.) Ceci est équivalent à `\pset format unaligned`.

`-b`
`--echo-errors`

Affiche les commandes SQL qui ont échoué sur la sortie standard des erreurs. C'est équivalent à configurer la variable `ECHO` à `errors`.

`-c` *commande*
`--command=`*commande*

Indique que psql doit exécuter la commande indiquée dans le paramètre *commande*. Cette option peut être répétée et combinée avec l'option `-f` dans n'importe quel ordre. Quand soit `-c` soit `-f` est utilisée, psql ne lit pas les commandes à partir de l'entrée standard ; à la place, il quitte après avoir traité toutes les options `-c` et `-f` dans la séquence indiquée.

commande doit être soit une chaîne de commande complètement analysable par le serveur (autrement dit, elle ne contient pas de fonctionnalités spécifiques à psql), soit une simple métacommande. De ce fait, vous ne pouvez pas mixer les commandes SQL et les métacommandes psql dans une option `-c`. Pour ce faire, vous pouvez utiliser plusieurs options `-c` ou envoyer la chaîne par un tube (*pipe*) dans psql, par exemple :

```
psql -c '\x' -c 'SELECT * FROM foo;'
```

or

```
echo '\x \\ SELECT * FROM foo;' | psql
```

(\\ est le séparateur de métacommandes.)

Chaque chaîne de commande SQL passée à `-c` est envoyée au serveur comme une requête unique. De ce fait, le serveur l'exécute comme une seule transaction, même si la chaîne contient plusieurs commandes SQL, sauf si des commandes `BEGIN`/`COMMIT` explicites sont incluses dans la chaîne pour la diviser en plusieurs transactions. (Voir Section 55.2.2.1 pour plus de détails sur la gestion par le serveur des chaînes contenant plusieurs requêtes.)

Si avoir plusieurs commandes exécutées dans une transaction n'est pas souhaité, utilisez plusieurs options `-c` ou envoyez les différentes commandes à psql via l'entrée standard, soit en utilisant echo comme dans l'exemple ci-dessus, soit en utilisant une redirection de type « here-document », comme ci-dessous :

```
psql <<EOF
\x
SELECT * FROM foo;
EOF
```

`--csv`

Passe au format de sortie CSV (Comma-Separated Values) Ceci est équivalent à `\pset format csv`.

`-d` *`nombase`*
`--dbname=`*`nombase`*

Indique le nom de la base de données où se connecter. Ceci est équivalent à spécifier *`nombase`* comme premier argument de la ligne de commande qui n'est pas une option. Le nom de la base, *`dbname`*, peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront toutes les options en ligne de commande conflictuelles.

`-e`
`--echo-queries`

Copie toutes les commandes SQL envoyées au serveur aussi sur la sortie standard. Ceci est équivalent à initialiser la variable `ECHO` à `queries`.

`-E`
`--echo-hidden`

Affiche les requêtes réelles générées par `\d` et autres commandes antislash. Vous pouvez utiliser ceci pour étudier les opérations internes de psql. Ceci est équivalent à initialiser la variable `ECHO_HIDDEN` à `on`.

`-f` *`nomfichier`*
`--file=`*`nomfichier`*

Lit les commandes à partir du fichier *`nomfichier`*, plutôt que l'entrée standard. Cette option peut être répétée et combinée dans tout ordre avec l'option `-c`. Quand ni `-c` ni `-f` n'est indiquée, psql ne lit pas les commandes à partir de l'entrée standard ; à la place, il termine après avoir traité toutes les options `-c` et `-f` dans la séquence indiquée. En dehors de ça, cette option est fortement équivalente à la métacommande `\i`.

Si *`nomfichier`* est un `-` (tiret), alors l'entrée standard est lue jusqu'à la détection d'une fin de fichier ou de la métacommande `\q`. Ceci peut être utilisé pour intercaler une saisie interactive

entre des entrées depuis des fichiers. Néanmoins, notez que Readline n'est pas utilisé dans ce cas (un peu comme si `-n` a été précisé).

Utiliser cette option est légèrement différent d'écrire `psql < nomfichier`. En général, les deux feront ce que vous souhaitez, mais utiliser `-f` active certaines fonctionnalités intéressantes comme les messages d'erreur avec les numéros de ligne. Il y a aussi une petite chance qu'utiliser cette option réduira la charge au démarrage. D'un autre côté, la variante utilisant la redirection de l'entrée du shell doit (en théorie) pour ramener exactement le même affichage que celui que vous auriez eu en saisissant tout manuellement.

`-F` *séparateur*
`--field-separator=`*séparateur*

Utilisez *séparateur* comme champ séparateur pour un affichage non aligné. Ceci est équivalent à `\pset fieldsep` ou `\f`.

`-h` *nomhôte*
`--host=`*nomhôte*

Indique le nom d'hôte de la machine sur lequel le serveur est en cours d'exécution. Si la valeur commence avec un slash, elle est utilisée comme répertoire du socket de domaine Unix.

`-H`
`--html`

Bascule en affichage HTML. Ceci est équivalent à `\pset format html` ou à la commande `\H`.

`-l`
`--list`

Liste toutes les bases de données disponibles puis quitte. Les autres options non relatives à la connexion sont ignorées. Ceci est similaire à la métacommande `\list`.

Quand cette option est utilisée, psql se connectera à la base de données `postgres`, sauf si une base de données différente est nommée sur la ligne de commande (via l'option `-d`, via le dernier argument de la ligne de commande, via l'enregistrement du service, mais pas via une variable d'environnement).

`-L` *nomfichier*
`--log-file=`*nomfichier*

Écrit tous les résultats des requêtes dans le fichier *nomfichier* en plus de la destination habituelle.

`-n`
`--no-readline`

N'utilise pas Readline pour l'édition de ligne et n'utilise pas l'historique des commandes (voir la section intitulée « Édition de la ligne de commande » ci-dessous).

`-o` *nomfichier*
`--output=`*nomfichier*

Dirige tous les affichages de requêtes dans le fichier *nomfichier*. Ceci est équivalent à la commande `\o`.

`-p` *port*
`--port=`*port*

Indique le port TCP ou l'extension du fichier socket de domaine local Unix sur lequel le serveur attend les connexions. Par défaut, il s'agit de la valeur de la variable d'environnement `PGPORT` ou, si elle n'est pas initialisée, le port spécifié au moment de la compilation, habituellement 5432.

`-P` *`affectation`*
`--pset=`*`affectation`*

Vous permet de spécifier les options d'affichage dans le style de `\pset` sur la ligne de commande. Notez que, ici, vous devez séparer nom et valeur avec un signe égal au lieu d'un espace. Du coup, pour initialiser le format d'affichage en LaTeX, vous devez écrire `-P format=latex`.

`-q`
`--quiet`

Indique que psql doit travailler silencieusement. Par défaut, il affiche des messages de bienvenue et diverses informations. Si cette option est utilisée, rien de ceci n'est affiché. C'est utile avec l'option `-c`. Ceci est équivalent à configurer la variable `QUIET` à `on`.

`-R` *`séparateur`*
`--record-separator=`*`séparateur`*

Utilisez *`séparateur`* comme séparateur d'enregistrement pour un affichage non aligné. Ceci est équivalent à `\pset recordsep`.

`-s`
`--single-step`

S'exécute en mode étape par étape. Ceci signifie qu'une intervention de l'utilisateur est nécessaire avant l'envoi de chaque commande au serveur, avec une option pour annuler l'exécution. Utilisez cette option pour déboguer des scripts.

`-S`
`--single-line`

S'exécute en mode simple ligne, où un retour à la ligne termine une commande SQL, de la même façon qu'un point-virgule.

> **Note**
>
> Ce mode est fourni pour ceux qui insistent pour l'avoir, mais vous n'êtes pas nécessairement encouragé à l'utiliser. En particulier, si vous mixez SQL et métacommandes sur une ligne, l'ordre d'exécution peut ne pas être toujours clair pour un utilisateur inexpérimenté.

`-t`
`--tuples-only`

Désactive l'affichage des noms de colonnes, le pied de page contenant le nombre de résultats, etc. Ceci est équivalent à la métacommande `\t` ou à `\pset tuples_only`.

`-T` *`options_table`*
`--table-attr=`*`options_table`*

Indique les options à placer à l'intérieur d'une balise `table` en HTML. Voir `\pset tableattr` pour plus de détails.

`-U` *`nomutilisateur`*
`--username=`*`nomutilisateur`*

Se connecte à la base de données en tant que l'utilisateur *`nomutilisateur`* au lieu de celui par défaut. (Vous devez aussi avoir le droit de le faire, bien sûr.)

`-v` *`affectation`*
`--set=`*`affectation`*
`--variable=`*`affectation`*

Réalise une affectation de variable, comme la métacommande `\set`. Notez que vous devez séparer le nom et la valeur, s'il y en a une, par un signe égal sur la ligne de commande. Pour désinitialiser une variable, enlevez le signe d'égalité. Pour initialiser une variable avec une valeur vide, utilisez le signe égal sans passer de valeur. Ces affectations sont réalisées lors du traitement de la ligne de commande, du coup les variables reflétant l'état de la connexion seront écrasées plus tard.

`-V`
`--version`

Affiche la version de psql et quitte.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

Notez que cette option restera positionnée pour l'ensemble de la session, et qu'elle affecte aussi l'utilisation de la métacommande `\connect` en plus de la tentative de connexion initiale.

`-W`
`--password`

Force psql à demander un mot de passe avant de se connecter à une base de données, même si le mot de passe ne sera pas utilisé.

Si le serveur requiert une authentification par mot de passe et qu'un mot de passe n'est pas disponible par d'autres sources que le fichier `.pgpass`, psql demandera un mot de passe. Néanmoins, psql perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Notez que cette option sera conservée pour la session entière, et que du coup elle affecte l'utilisation de la métacommande `\connect` ainsi que la tentative de connexion initiale.

`-x`
`--expanded`

Active le mode de formatage de table étendu. Ceci est équivalent à `\x` ou `\pset expanded`.

`-X`
`--no-psqlrc`

Ne lit pas le fichier de démarrage (ni le fichier système `psqlrc` ni le `~/.psqlrc` de l'utilisateur).

`-z`
`--field-separator-zero`

Configure le séparateur de champs pour une sortie non alignée avec un octet zéro. Ceci est équivalent à `\pset fieldsep_zero`.

`-0`
`--record-separator-zero`

Configure le séparateur d'enregistrement pour une sortie non alignée avec un octet zéro. C'est intéressant pour l'interfacer avec `xargs -0`. Ceci est équivalent à `\pset recordsep_zero`.

`-1`
`--single-transaction`

Cette option peut seulement être utilisée en combinaison avec une ou plusieurs options `-c` et/ou `-f`. Cela force psql à exécuter une commande `BEGIN` avant la première option de ce type et une commande `COMMIT` après la dernière, englobant la totalité des commandes dans une seule transaction. Si un des commandes échoue alors que la variable `ON_ERROR_STOP` était configurée, une commande `ROLLBACK` est envoyée à la place. Ceci garantit que soit toutes les commandes réussissent, soit aucun changement n'est appliqué.

Si les commandes elles-mêmes contiennent `BEGIN`, `COMMIT` ou `ROLLBACK`, cette option n'aura pas les effets désirés. De plus, si une commande en particulier ne peut pas être exécutée à l'intérieur d'un bloc de transaction, indiquer cette option causera l'échec de toute la transaction.

`-?`
`--help[=`*`thème`*`]`

Affiche de l'aide sur psql puis quitte. Le paramètre optionnel *`thème`* (par défaut à `options`) sélectionne les parties de psql à expliquer : `commands` décrit les métacommandes de psql ; `options` décrit les options en ligne de commande de psql ; et `variables` affiche de l'aide sur les variables de configuration de psql.

Code de sortie

psql renvoie 0 au shell s'il s'est terminé normalement, 1 s'il y a eu une erreur fatale de son fait (par exemple : pas assez de mémoire, fichier introuvable), 2 si la connexion au serveur s'est interrompue et que la session n'était pas interactive, 3 si une erreur est survenue dans un script et si la variable `ON_ERROR_STOP` était positionnée.

Usage

Se connecter à une base de données

psql est une application client PostgreSQL standard. Pour se connecter à une base de données, vous devez connaître le nom de votre base de données cible, le nom de l'hôte et le numéro de port du serveur ainsi que le nom de l'utilisateur sous lequel vous voulez vous connecter. On peut indiquer ces paramètres à psql à partir d'options en ligne de commande, respectivement `-d`, `-h`, `-p` et `-U`. Si un argument est rencontré qui ne correspond à aucune option, il sera interprété comme le nom de la base de données (ou le nom de l'utilisateur si le nom de la base de données est déjà donné). Toutes ces options ne sont pas requises, il y a des valeurs par défaut convenables. Si vous omettez le nom de l'hôte, psql se connectera via un socket de domaine Unix à un serveur sur l'hôte local, ou par TCP/IP sur `localhost` pour les machines qui n'ont pas de sockets de domaine Unix. Le numéro de port par défaut est déterminé au moment de la compilation. Comme le serveur de bases de données utilise la même valeur par défaut, vous n'aurez pas besoin de spécifier le port dans la plupart des cas. Le nom de l'utilisateur par défaut est votre nom d'utilisateur pour le système d'exploitation, de même pour le nom de la base de données par défaut. Notez que vous ne pouvez pas simplement vous connecter à n'importe quelle base de données avec n'importe quel nom d'utilisateur. Votre administrateur de bases de données doit vous avoir informé de vos droits d'accès.

Quand les valeurs par défaut ne sont pas idéales, vous pouvez vous épargner de la frappe en configurant les variables d'environnement `PGDATABASE`, `PGHOST`, `PGPORT` et/ou `PGUSER` avec les valeurs appropriées (pour les variables d'environnement supplémentaires, voir Section 34.15). Il est aussi pratique d'avoir un fichier `~/.pgpass` pour éviter d'avoir régulièrement à saisir les mots de passe. Voir Section 34.16 pour plus d'informations.

Une autre façon d'indiquer les paramètres de connexion est dans une chaîne *`conninfo`* ou une URI qui est utilisée à la place du nom d'une base de données. Ce mécanisme vous donne un grand contrôle sur la connexion. Par exemple :

```
$ psql "service=monservice sslmode=require"
$ psql postgresql://dbmaster:5433/mydb?sslmode=require
```

De cette façon, vous pouvez aussi utiliser LDAP pour la recherche de paramètres de connexion, comme décrit dans Section 34.18. Voir Section 34.1.2 pour plus d'informations sur toutes les options de connexion disponibles.

Si la connexion ne peut pas se faire, quelle qu'en soit la raison (c'est-à-dire droits non suffisants, serveur arrêté sur l'hôte cible, etc.), psql renverra une erreur et s'arrêtera.

Si l'entrée et la sortie standard correspondent à un terminal, alors psql fixe le paramètre d'encodage client à la valeur « auto », afin de pouvoir détecter l'encodage approprié d'après les paramètres régionaux (définis par la variable système `LC_CTYPE` pour les systèmes Unix). Si cela ne fonctionne pas comme attendu, il est possible de forcer l'encodage du client en renseignant la variable d'environnement `PGCLIENTENCODING`.

Saisir des commandes SQL

Dans le cas normal, psql fournit une invite avec le nom de la base de données sur laquelle psql est connecté suivi par la chaîne `=>`. Par exemple

```
$ psql basetest
psql (15.19)
Type "help" for help.

basetest=>
```

À l'invite l'utilisateur peut saisir des commandes SQL. Ordinairement, les lignes en entrée sont envoyées vers le serveur quand un point-virgule de fin de commande est saisi. Une fin de ligne ne termine pas une commande. Du coup, les commandes peuvent être saisies sur plusieurs lignes pour plus de clarté. Si la commande a été envoyée et exécutée sans erreur, ses résultats sont affichés sur l'écran.

Si des utilisateurs en qui vous n'avez pas confiance ont accès à une base qui n'a pas adopté la méthode sécurisée d'utilisation des schémas, démarrez votre session en supprimant de votre `search_path` les schémas ouverts en écriture au public. On peut ajouter `options=-csearch_path=` à la chaîne de connexion ou exécuter `SELECT pg_catalog.set_config('search_path', '', false)` avant toute autre commande SQL. Cette considération n'est pas propre à psql ; elle s'applique à chaque interface qui exécute des commandes SQL quelconques.

À chaque fois qu'une commande est exécutée, psql vérifie aussi les événements de notification générés par `LISTEN` et `NOTIFY`.

Alors que les blocs de commentaire de type C sont transmis au serveur pour traitement et suppression, les commentaires au standard SQL sont supprimés par psql.

Métacommandes

Tout ce que vous saisissez dans psql qui commence par un antislash non échappé est une métacommande psql, traitée par psql lui-même. Ces commandes aident à rendre psql plus utile pour l'administration ou pour l'écriture de scripts. Les métacommandes sont plus souvent appelées les commandes slash ou antislash.

Le format d'une commande psql est l'antislash suivi immédiatement d'un verbe de commande et de ses arguments. Les arguments sont séparés du verbe de la commande et les uns des autres par un nombre illimité d'espaces blancs.

Pour inclure des espaces blancs dans un argument, vous pouvez le mettre entre des guillemets simples. Pour inclure un guillemet simple dans un argument, vous devez écrire deux guillemets simples dans un texte compris entre guillemets simples. Tout ce qui est contenu dans des guillemets simples est sujet aux substitutions du style langage C : `\n` (nouvelle ligne), `\t` (tabulation), `\b` (retour arrière), `\r` (retour chariot), `\f` (saut de page), `\`*`chiffres`* (octal), and `\x`*`chiffres`* (hexadécimal). Un antislash précédant tout autre caractère dans une chaîne entre guillemets reproduit ce caractère, quel qu'il soit.

Si un deux-points sans guillemets (`:`) suivi d'un nom de variable psql apparaît dans un argument, il est remplacé par la valeur de la variable, comme décrit dans Interpolation SQL. Les formes `:'`*`variable_name`*`'` et `:"`*`variable_name`*`"` décrites ici fonctionnent également. La syntaxe `:{?`*`variable_name`*`}` permet de tester si une variable est définie. Elle est substituée par TRUE ou FALSE. Échapper le symbole deux-points avec un antislash le protège de la substitution.

Dans un argument, le texte entre des guillemets inverses (`` ` ``) est pris comme une ligne de commande, qui est passée au shell. La sortie de la commande (dont tous les retours à la ligne sont supprimés) remplace le texte entre guillemets inverses. Dans le texte à l'intérieur des guillemets inverses, ne se déroule ni échappement ni autre traitement, à l'exception de `:`*`variable_name`* où *`variable_name`* est une variable psql, qui sera remplacée par sa valeur. De plus, les occurrences de `:'`*`variable_name`*`'` sont remplacées par la valeur de la variable correctement échappée pour devenir un unique argument de commande shell (cette dernière forme est presque toujours préférable, sauf à être absolument sûr du contenu de la variable). Comme les caractères retour chariot et saut de ligne ne peuvent être échappés correctement sur toutes les plateformes, la forme `:'`*`variable_name`*`'` renvoie un message d'erreur et ne remplace pas la valeur de la variable quand ces caractères sont présents dans la valeur.

Quelques commandes prennent un identifiant SQL (comme un nom de table) en argument. Ces arguments suivent les règles de la syntaxe SQL : les lettres sans guillemets sont forcées en minuscule alors que les guillemets doubles (") protègent les lettres de la conversion de casse et autorisent l'incorporation d'espaces blancs dans l'identifiant. À l'intérieur des guillemets doubles, les guillemets doubles en paire se réduisent à un seul guillemet double dans le nom résultant. Par exemple, `FOO"BAR"BAZ` est interprété comme `fooBARbaz` et `"Un nom ""bizarre"` devient `Un nom "bizarre`.

L'analyse des arguments se termine à la fin de la ligne ou quand un autre antislash non entre guillemets est rencontré. Un antislash non entre guillemets est pris pour le début d'une nouvelle métacommande. La séquence spéciale `\\` (deux antislashs) marque la fin des arguments et continue l'analyse des commandes SQL, si elles existent. De cette façon, les commandes SQL et psql peuvent être mixées librement sur une ligne. Mais dans tous les cas, les arguments d'une métacommande ne peuvent pas continuer après la fin de la ligne.

De nombreuses métacommandes utilisent le *buffer de la requête actuelle*. Ce buffer contient simplement le texte de la commande SQL qui a été écrite mais pas encore envoyée au serveur pour exécution. Cela comprendra les lignes saisies précédentes ainsi que tout texte présent avant la métacommande de la même ligne.

Les métacommandes suivantes sont définies :

`\a`

Si le format d'affichage de table actuel est non aligné, il est basculé à aligné. S'il n'est pas non aligné, il devient non aligné. Cette commande est conservée pour des raisons de compatibilité. Voir `\pset` pour une solution plus générale.

`\c` ou `\connect [ -reuse-previous=`*`on`*`|`*`off`*` ] [ `*`nom_base`*` [ `*`nom_utilisateur`*` ] [ `*`hôte`*` ] [ `*`port`*` ] | `*`conninfo`*` ]`

Établit une nouvelle connexion à un serveur PostgreSQL. Les paramètres de connexion utilisés peuvent être spécifiés en utilisant soit la syntaxe par position (une ou plusieurs parmi le nom de la base, le nom de l'utilisateur, l'hôte et le port) soit une chaîne de connexion `conninfo` telle

qu'elle est détaillée dans Section 34.1.1. Si aucun argument n'est donné, une nouvelle connexion est réalisée en utilisant les mêmes paramètres qu'auparavant.

Utiliser `-` comme valeur d'un des paramètres *`nom_base`*, *`nom_utilisateur`*, *`hôte`* ou *`port`* est équivalent à l'omission de ce paramètre.

La nouvelle connexion peut réutiliser les paramètres de connexion de la précédente connexion ; non seulement le nom de la base, de l'utilisateur, l'hôte et le port, mais aussi les autres paramètres tels que *`sslmode`*. Par défaut, les paramètres sont ré-utilisés dans la syntaxe par position, mais pas quand une chaîne de connexion *`conninfo`* est donnée. Passer en premier argument `-reuse-previous=on` ou `-reuse-previous=off` surcharge ce comportement par défaut. Si les paramètres sont ré-utilisés, alors tout paramètre non spécifié explicitement comme un paramètre de position ou dans une chaîne de connexion *`conninfo`* est pris dans les paramètres de la connexion existante. Une exception concerne le changement du paramètre *`host`* de sa valeur précédente utilisant la syntaxe par position, et tout paramétrage de *`hostaddr`* présent dans les paramètres de la connexion existant est supprimé. De plus, tout mot de passe utilisé pour la connexion existante sera réutilisé seulement si l'utilisateur, l'hôte et le port ne sont pas modifiés. Quand la commande ne spécifie pas ou ne réutilise pas une paramètre particulier, sa valeur libpq par défaut est utilisé.

Si la nouvelle connexion est réussie, la connexion précédente est fermée. Si la tentative de connexion échoue (mauvais nom d'utilisateur, accès refusé), la connexion précédente sera conservée et psql sera en mode interactif. Mais lors de l'exécution d'un script non interactif, l'ancienne connexion est fermée et une erreur est renvoyée. Ceci pourrait terminer ou non le script ; si cela ne le fait pas, toutes les commandes d'accès à la base échoueront jusqu'à ce qu'une autre commande `\connect` soit exécutée avec succès. Cette distinction a été choisie pour faciliter la vie de l'utilisateur en cas de fautes de frappe, et en même temps comme mécanisme de sécurité pour que les scripts n'agissent pas sur la mauvaise base de données. Notez que, quand une commande `\connect` tente de réutiliser des paramètres, les valeurs ré-utilisées sont celles de la dernière connexion réussie, pas celles des tentatives échouées faites après. Néanmoins, dans le cas d'un échec d'un `\connect` non interactif, aucun paramètre ne peut être réutilisé après coup car le script pourrait s'attendre à la réutilisation des valeurs de `\connect` échoué.

Exemples :

```
=> \c mydb myuser host.dom 6432
=> \c service=foo
=> \c "host=localhost port=5432 dbname=mydb connect_timeout=10
 sslmode=disable"
=> \c -reuse-previous=on sslmode=require    -- change uniquement
 sslmode
=> \c postgresql://tom@localhost/mydb?application_name=myapp
```

`\C [ *titre* ]`

Initialise ou supprime le titre des tables affichées en résultat d'une requête. Cette commande est équivalente à `\pset title *titre*`. (Le nom de cette commande provient de « caption », car elle avait précédemment pour seul but d'initialiser l'en-tête dans une table HTML.)

`\cd [ *répertoire* ]`

Change le répertoire courant en *`répertoire`*. Sans argument, le répertoire personnel de l'utilisateur devient le répertoire courant.

> **Astuce**
>
> Pour afficher votre répertoire courant, utilisez `\! pwd`.

`\conninfo`

Affiche des informations sur la connexion en cours à la base de données.

```
\copy { table [ ( liste_colonnes ) ] } from { 'nomfichier' | program
'commande' | stdin | pstdin } [ [ with ] ( option [, ...] ) ] [ where
condition ]
\copy { table [ ( liste_colonnes ) ] | ( requête ) } to { 'nomfichier' |
program 'commande' | stdout | pstdout } [ [ with ] ( option [, ...] ) ]
```

Réalise une opération de copie côté client. C'est une opération qui exécute une commande SQL, `COPY`, mais au lieu que le serveur lise ou écrive le fichier spécifié, psql lit ou écrit le fichier en faisant le routage des données entre le serveur et le système de fichiers local. Ceci signifie que l'accès et les droits du fichier sont ceux de l'utilisateur local, pas celui du serveur, et qu'aucun droit de superutilisateur n'est requis.

Quand la clause `program` est présente, *`commande`* est exécuté par psql et les données provenant ou fournies à *`commande`* sont routées entre le serveur et le client. Encore une fois, les droits d'exécution sont ceux de l'utilisateur local, et non du serveur, et les droits superutilisateur ne sont pas nécessaires.

Pour `\copy ... from stdin`, les lignes de données sont lues depuis la même source qui a exécuté la commande, continuant jusqu'à ce que `\.` soit lu ou que le flux atteigne EOF. Cette option est utile pour remplir des tables depuis les scripts SQL même. Pour `\copy ... to stdout`, la sortie est envoyée au même endroit que la sortie des commandespsql, et le statut de la commande `COPY` *`count`* n'est pas affiché (puisqu'il pourrait être confondu avec une ligne de données). Pour lire et écrire sur les entrées et sorties de psql sans prendre en compte la source de commande courante ou l'option `\o`, écrivez `from pstdin` ou `to pstdout`.

La syntaxe de cette commande est similaire à celle de la commande SQL `COPY`. Toutes les options autres que les source/destination des données sont spécifié comme pour `COPY`. À cause de cela, des règles spéciales d'analyse sont appliquées à la métacommande `\copy`. Contrairement à la majorité des autres métacommandes, l'intégralité du reste de la ligne est toujours pris en compte en tant qu'arguments de `\copy`, et ni l'interpolation des variables ni la substitution par guillemets inverses ne seront effectuées sur les arguments.

> **Astuce**
>
> Une autre façon d'obtenir le même résultat que `\copy ... to` est d'utiliser la commande SQL `COPY ... TO STDOUT` et de la finir avec `\g` *`nom_fichier`* ou `\g |`*`programme`*. Contrairement à `\copy`, cette méthode permet à la commande d'aller sur plusieurs lignes. De plus, l'interpolation de variable et l'expansion des guillemets inverses peuvent être utilisées.

> **Astuce**
>
> Ces opérations ne sont pas aussi efficaces que la commande SQL `COPY` avec un fichier ou avec les données fournies pour/par un programme car toutes les données doivent passer via la connexion client/serveur. Pour les grandes quantités de données, la commande SQL pourrait être préférable. De plus, du fait de cette méthode de passage, `\copy ... from` dans le mode CSV traitera de façon erronée une donnée `\.` seule sur une ligne en la prenant pour un marqueur de fin de données.

`\copyright`

Affiche le copyright et les termes de distribution de PostgreSQL.

`\crosstabview [ colV [ colH [ colD [ sortcolH ] ] ] ]`

Exécute le tampon de requête actuel (tout comme `\g`) et affiche le résultat dans un tableau croisé. La requête doit renvoyer au moins trois colonnes. La colonne en sortie identifiée par *colV* devient l'en-tête vertical et la colonne en sortie identifiée par *colH* devient l'en-tête horizontal. *colD* identifie la colonne en sortie à afficher à l'intérieur de la grille. *sortcolH* identifie une colonne optionnelle de tri pour l'en-tête horizontal.

Chaque spécification de colonne peut être un numéro de colonne (en commençant à 1) ou un nom de colonne. Les règles SQL habituelles de casse et de guillemet s'appliquent aux noms de colonne. En cas d'omission, la colonne 1 est utilisée pour *colV* et la colonne 2 est utilisée pour *colH*. *colH* doit différer de *colV*. Si *colD* n'est pas indiqué, alors il doit y avoir exactement trois colonnes dans le résultat de la requête et la colonne qui n'est ni *colV* ni *colH* est utilisée pour *colD*.

L'en-tête vertical, affiché comme colonne la plus à gauche, contient les valeurs trouvées dans la colonne *colV*, dans le même ordre que dans les résultats de la requête, mais sans les duplicats.

L'en-tête horizontal, affiché comme la première ligne, contient les valeurs trouvées dans la colonne *colH*, sans duplicats. Par défaut, ils apparaissent dans le même ordre que les résultats de la requête. Mais si l'argument optionnel *sortcolH* est renseigné, il identifie une colonne dont les valeurs doivent être des entiers et les valeurs provenant de *colH* apparaîtront dans l'en-tête horizontal trié suivant les valeurs correspondantes de *sortcolH*.

À l'intérieur du tableau croisé, pour chaque valeur `x` distincte de *colH* et pour chaque valeur `y` distincte de *colV*, la cellule située à l'intersection `(x,y)` contient la valeur de la colonne `colD` dans la ligne de résultat de la requête pour laquelle la valeur de *colH* est `x` et la valeur de *colV* est `y`. Si cette ligne n'existe pas, la cellule est vide. S'il existe plusieurs lignes, une erreur est renvoyée.

`\d[S+] [ motif ]`

Pour chaque relation (table, vue, vue matérialisée, index, séquence ou table distante) ou type composite correspondant au *motif*, affiche toutes les colonnes, leurs types, le tablespace (s'il ne s'agit pas du tablespace par défaut) et tout attribut spécial tel que `NOT NULL` ou les valeurs par défaut. Les index, contraintes, règles et triggers associés sont aussi affichés. Pour les tables distantes, le serveur distant associé est aussi affiché. (Ce qui « correspond au motif » est défini dans Motifs ci-dessous.)

Pour certains type de relation, `\d` affiche des informations supplémentaires pour chaque colonne ; colonne valeur pour les séquences, expression indexée pour les index, options du wrapper de données distantes pour les tables distantes.

Le forme de la commande `\d+` est identique, sauf que des informations plus complètes sont affichées : tout commentaire associé avec les colonnes de la table est affiché, ainsi que la présence d'OID dans la table, la définition de la vue (si la relation ciblée est une vue), un réglage de replica identity autre que celui par défaut and the access method name if the relation has an access method.

Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets système.

> **Note**
>
> Si `\d` est utilisé sans argument *motif*, il est équivalent, en plus commode, à `\dtvmsE` qui affiche une liste de toutes les tables, vues, vues matérialisées, séquences et tables distantes. Ce n'est qu'un outil pratique.

`\da[S] [ motif ]`

Liste toutes les fonctions d'agrégat disponibles, avec le type en retour et les types de données sur lesquels elles opèrent. Si *motif* est spécifié, seuls les agrégats dont les noms commencent par le

motif sont affichés. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur S pour afficher les objets système.

`\dA[+] [ motif ]`

Liste les méthodes d'accès. Si *motif* est précisé, seules sont affichées les méthodes d'accès dont le nom correspond au motif. Si + est ajouté au nom de la commande, chaque méthode d'accès est listée avec sa fonction gestionnaire et sa description associées.

`\dAc[+] [access-method-pattern [input-type-pattern]]`

Liste les classes d'opérateurs (voir Section 53.33). Si *access-method-pattern* est spécifié, seules les classes d'opérateurs associées avec les méthodes d'accès dont les noms correspondent au motif sont listées. Si *input-type-pattern* est spécifié, seules les classes d'opérateurs associées avec les types en entrée dont les noms correspondent au motif sont listées. Si + est ajouté au nom de la commande, chaque classe d'opérateur est affichée avec sa famille d'opérateur associée et propriétaire.

`\dAf[+] [access-method-pattern [input-type-pattern]]`

Liste les familles d'opérateur (voir Section 38.16.5). Si *access-method-pattern* est spécifié, seules les familles d'opérateur associées avec les méthodes d'accès dont les noms correspondent au motif sont listées. Si *input-type-pattern* est spécifié, seules les familles d'opérateur associées avec les types en entrée dont les noms correspondent au motif sont listées. Si + est ajouté au nom de la commande, chaque famille d'opérateur est affichée avec son propriétaire.

`\dAo[+] [access-method-pattern [operator-family-pattern]]`

Liste les fonctions associées avec les familles d'opérateur (Section 38.16.2). Si *access-method-pattern* est spécifié, seuls les membres des familles d'opérateur associés avec les méthodes d'accès dont les noms correspondent au motif sont listés. Si *operator-family-pattern* est spécifié, seuls les membres des familles d'opérateur dont les noms correspondent au motif sont listés. Si + est ajouté au nom de la commande, chaque opérateur est listé avec sa famille d'opérateur de tri (s'il en a un).

`\dAp[+] [access-method-pattern [operator-family-pattern]]`

Listes les procédures de support associées avec les familles d'opérateur (Section 38.16.3). Si *access-method-pattern* est spécifié, seules les fonctions des familles d'opérateur associées avec les méthodes d'accès dont le nom correspond au motif sont listées. Si *operator-family-pattern* est spécifié, seules les fonctions des familles d'opérateur dont les noms correspondent au motif sont listées. Si + est ajouté au nom de la commande, chaque fonction est affiché de façon verbeuse, avec sa liste réelle de paramètres.

`\db[+] [ motif ]`

Liste tous les tablespaces disponibles. Si *motif* est spécifié, seuls les tablespaces dont le nom correspond au motif sont affichés. Si + est ajouté au nom de commande, chaque tablespace est listé avec ses options associées, sa taille sur disque, ses droits et sa description.

`\dc[S+] [ motif ]`

Liste les conversions entre les encodages de jeux de caractères. Si *motif* est spécifié, seules les conversions dont le nom correspond au motif sont listées. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur S pour afficher les objets système. Si + est ajouté au nom de la commande, chaque objet est listé avec sa description associée.

`\dconfig[+] [ pattern ]`

Liste les paramètres de configuration serveur et leurs valeurs. Si *pattern* est précisé, seuls vont apparaître les paramètres dont le nom correspond au motif. Sans *pattern*, seuls vont

apparaître les paramètres ayant une valeur personnalisée, différente de la valeur par défaut. (Utiliser `\dconfig *` pour voir tous les paramètres.) Si `+` est ajouté au nom de la commande, chaque paramètre est listé avec son type de données, son contexte et ses droits d'accès (si des droits d'accès personnalisés ont été donnés).appended to the command name, each

`\dC[+] [ motif ]`

Liste les conversions de types. Si *`motif`* est indiqué, seules sont affichées les conversions dont le type source ou cible correspond au motif. Si `+` est ajouté au nom de la commande, chaque objet est listé avec sa description associée.

`\dd[S] [ motif ]`

Affiche les descriptions des objets du type `contrainte`, `classe d'opérateurs`, `famille d'opérateur`, `règle` et `trigger`. Tous les autres commentaires peuvent être visualisés avec les commandes antislash respectives pour ces types d'objets.

`\dd` Affiche les descriptions des objets correspondant au *`motif`* ou des objets du type approprié si aucun argument n'est donné. Mais dans tous les cas, seuls les objets qui ont une description sont listés. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets système.

Les descriptions des objets peuvent être créées avec la commande SQL `COMMENT`.

`\dD[S+] [ motif ]`

Liste les domaines. Si *`motif`* est spécifié, seuls les domaines dont le nom correspond au motif sont affichés. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets système. Si `+` est ajouté au nom de la commande, chaque objet est listé avec sa description associée.

`\ddp [ motif ]`

Liste les paramètres par défaut pour les privilèges d'accès. Une entrée est affichée pour chaque rôle (et schéma, si c'est approprié) pour lequel les paramètres par défaut des privilèges ont été modifiés par rapport aux paramètres par défaut intégrés. Si *`motif`* est spécifié, seules les entrées dont le nom de rôle ou le nom de schéma correspond au motif sont listées.

La commande `ALTER DEFAULT PRIVILEGES` sert à positionner les privilèges d'accès par défaut. La signification de l'affichage des privilèges est expliquée à la page de Section 5.7.

`\dE[S+] [ motif ]`
`\di[S+] [ motif ]`
`\dm[S+] [ motif ]`
`\ds[S+] [ motif ]`
`\dt[S+] [ motif ]`
`\dv[S+] [ motif ]`

Dans ce groupe de commandes, les lettres `E`, `i`, `m`, `s`, `t` et `v` correspondent respectivement à table distante, index, vue matérialisée, séquence, table et vue. Vous pouvez indiquer n'importe quelle combinaison de ces lettres, dans n'importe quel ordre, pour obtenir la liste de tous les objets de ces types. Par exemple, `\dti` liste les index et tables. Si `+` est ajouté à la fin de la commande, chaque objet est listé avec son statut de persistence (permanent, temporaire, non journalisé), sa taille physique sur disque et sa description associée s'il y en a une. Si *`motif`* est spécifié, seuls les objets dont les noms correspondent au motif sont listés. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets système.

`\des[+] [ motif ]`

Liste les serveurs distants (mnémonique : « external servers »). Si *`motif`* est spécifié, seuls les serveurs dont le nom correspond au motif sont affichés. Si la forme `\des+` est utilisée, une

description complète de chaque serveur est affichée, incluant liste de contrôle d'accès du serveur (ACL), type, version, options et description.

`\det[+] [ motif ]`

Liste les tables distantes (mnémotechnique : « tables externes »). Si un *motif* est fourni, seules les entrées concernant les tables ou les schémas en correspondance seront listées. Si vous utilisez la forme `\det+`, les options génériques et la description de la table distante seront également affichées.

`\deu[+] [ motif ]`

Liste les correspondances d'utilisateurs (mnémonique : « external users »). Si *motif* est spécifié, seules les correspondances dont le nom correspond au motif sont affichées. Si la forme `\deu+` est utilisée, des informations supplémentaires sur chaque correspondance d'utilisateur sont affichées.

> **Attention**
>
> `\deu+` risque aussi d'afficher le nom et le mot de passe de l'utilisateur distant, il est donc important de faire attention à ne pas les divulguer.

`\dew[+] [ motif ]`

Liste les wrappers de données distantes (mnémonique : « external wrappers »). Si *motif* est spécifié, seuls les wrappers dont le nom correspond au motif sont affichés. Si la forme `\dew+` est utilisée, les ACL, options et description du wrapper sont aussi affichées.

`\df[anptwS+] [ motif [ motif_arg ... ] ]`

Liste les fonctions, ainsi que leurs types de données pour le résultat, leurs types de données pour les arguments et les types de fonctions, qui sont classés comme « agg » (agrégat), « normal », « procedure », « trigger », or « window ». Afin de n'afficher que les fonctions d'un type spécifié, ajoutez les lettres correspondantes, respectivement `a`, `n`, `p`, `t`, or `w` à la commande. Si *motif* est spécifié, seules les fonctions dont le nom correspond au motif sont affichées. Tout argument supplémentaire est un motif pour un nom de type, faisant correspondance aux noms de type du premier argument de la fonction, puis du second, et ainsi de suite. (Les fonctions correspondantes peuvent avoir plus d'arguments que le nombre de types d'argument indiqués. Pour empêcher cela, ajoutez un tiret `-` comme fin pour *motif_arg*.) Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets système. Si la forme `\df+` est utilisée, des informations supplémentaires sur chaque fonction sont affichées, incluant la volatibilité, le parallélisme, le propriétaire, la classification en sécurité, les droits d'accès, le langage, le code source et la description.

`\dF[+] [ motif ]`

Liste les configurations de la recherche plein texte. Si *motif* est spécifié, seules les configurations dont le nom correspond au motif seront affichées. Si la forme `\dF+` est utilisée, une description complète de chaque configuration est affichée, ceci incluant l'analyseur de recherche plein texte et la liste de dictionnaire pour chaque type de jeton de l'analyseur.

`\dFd[+] [ motif ]`

Liste les dictionnaires de la recherche plein texte. Si *motif* est spécifié, seuls les dictionnaires dont le nom correspond au motif seront affichés. Si la forme `\dFd+` est utilisée, des informations supplémentaires sont affichées pour chaque dictionnaire, ceci incluant le motif de recherche plein texte et les valeurs des options.

`\dFp[+] [ motif ]`

Liste les analyseurs de la recherche plein texte. Si *motif* est spécifié, seuls les analyseurs dont le nom correspond au motif seront affichés. Si la forme `\dFp+` est utilisée, une description complète

de chaque analyseur est affichée, ceci incluant les fonctions sous-jacentes et la liste des types de jeton reconnus.

`\dFt[+] [ motif ]`

Liste les motifs de la recherche plein texte. Si *`motif`* est spécifié, seuls les motifs dont le nom correspond au motif seront affichés. Si la forme `\dFt+` est utilisée, des informations supplémentaires sont affichées pour chaque motif, ceci incluant les noms des fonctions sous-jacentes.

`\dg[S+] [ pattern ]`

Liste les rôles des bases de données. (Comme les concepts d'« utilisateurs » et « groupes » ont été unifiés dans les « rôles », cette commande est maintenant équivalente à `\du`.) Par défaut, seuls les rôles créés par des utilisateurs sont affichés ; ajoutez le modificateur `S` pour inclure les rôles système. Si *`motif`* est spécifié, seuls les rôles dont le nom correspond au motif sont listés. Si la forme `\dg+` est utilisée, des informations supplémentaires sont affichées pour chaque rôle ; actuellement, cela ajoute le commentaire pour chaque rôle.

`\dl[+]`

Ceci est un alias pour `\lo_list`, qui affiche une liste des Large Objects. Si `+` est ajouté au nom de la commande, chaque Large Object est listé avec les droits associés, s'il y en a.

`\dL[S+] [ motif ]`

Affiche les langages procéduraux. Si un *`motif`* est spécifié, seuls les langages dont les noms correspondent au motif sont listés. Par défaut, seuls les langages créés par les utilisateurs sont affichés ; il faut spécifier l'option `S` pour inclure les objets système. Si `+` est ajouté à la fin de la commande, chaque langage sera affiché avec ses gestionnaire d'appels, validateur, droits d'accès, et ce même s'il s'agit d'un objet système.

`\dn[S+] [ motif ]`

Liste les schémas (espaces de noms). Si *`motif`* est spécifié, seuls les schémas dont le nom correspond au motif sont listés. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets système. Si `+` est ajouté à la fin de la commande, chaque objet sera affiché avec ses droits et son éventuelle description.

`\do[S+] [ motif [ motif_arg [ motif_arg ] ] ]`

Liste les opérateurs avec les types de leur opérande et résultat. Si *`motif`* est spécifié, seuls les opérateurs dont le nom correspond au motif sont listés. Si un *`motif_arg`* est indiqué, seuls les opérateurs préfixes dont le nom du type de l'argument droit correspond au motif sont listés. Si deux *`motif_arg`* sont indiqués, seuls les opérateurs binaires dont les noms des types d'argument correspondent à ces motifs sont listés. (Autrement, écrivez `-` pour l'argument inutilisé d'un opérateur unaire.) Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets système. Si `+` est ajouté au nom de la commande, des informations supplémentaire sur chaque opérateur est affiché, actuellement uniquement le nom de la fonction sous-jacente.

`\dO[S+] [ motif ]`

Affiche les collationnements. Si *`motif`* est spécifié, seuls les collationnements dont le nom correspond au motif sont listés. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur `S` pour afficher les objets système. Si `+` est ajouté à la fin de la commande, chacun des collationnements sera affiché avec son éventuelle description. Notez que seuls les collationnements compatibles avec l'encodage de la base de données courante sont affichés, les résultats peuvent donc varier selon les différentes bases d'une même instance.

`\dp [ motif ]`

Liste les tables, vues et séquences avec leur droits d'accès associés. Si *motif* est spécifié, seules les tables, vues et séquences dont le nom correspond au motif sont listées.

Les commandes GRANT et REVOKE sont utilisées pour configurer les droits d'accès. Les explications sur le sens de l'affichage des privilèges sont sous Section 5.7.

`\dP[itn+] [ motif ]`

Liste les tables partitionnées. Si *motif* est spécifié, seules les entrées dont le nom correspond au motif sont listées. Les modificateurs t (tables) et i (index) peuvent être ajoutés à la commande pour filtrer le type d'objet à lister. Par défaut, les tables et index partitionnés sont listés.

Si le modificateur n (« nested ») est indiqué, ou si un motif est spécifié, alors les partitions qui ne sont pas à la racine sont incluses, et une colonne indique le parent de chaque objet partitionné.

Si + est ajouté à la commande, la somme des tailles de chaque partition est aussi affichée, ainsi que la description de la relation. Si n est combiné avec +, deux tailles sont affichées : l'une incluant la taille totale des partitions directement attachées, l'autre montrant la taille totale de toutes les partitions, y compris les sous-partitions attachées indirectement.

`\drds [ role-pattern [ database-pattern ] ]`

Liste les paramètres de configuration définis. Ces paramètres peuvent être spécifiques à un rôle, spécifiques à une base, ou les deux. *role-pattern* et *database-pattern* servent à choisir sur quels rôles spécifiques ou quelles bases de données les paramètres sont listés. Si ces options sont omises, ou si on spécifie *, tous les paramètres sont listés, y compris ceux qui ne sont pas spécifiques, respectivement, à un rôle ou une base.

Les commande ALTER ROLE et ALTER DATABASE servent à définir les paramètres de configuration par rôle et par base de données.

`\dRp[+] [ pattern ]`

Liste les publications de réplication. Si *pattern* est spécifié, seules les publications dont le nom correspond au motif sont listées. Si + est ajouté à la fin du nom de la commande, les tables et les schémas associés à chaque publication sont également affichés.

`\dRs[+] [ pattern ]`

Liste les souscriptions de réplication. Si *pattern* est spécifié, seules les souscriptions dont le nom correspond au motif sont listées. Si + est ajouté à la fin du nom de la commande, des propriétés supplémentaires de la souscription sont affichées.

`\dT[S+] [ motif ]`

Liste les types de données. Si *motif* est spécifié, seuls les types dont le nom correspond au motif sont affichés. Si + est ajouté à la fin de la commande, chaque type est listé avec son nom interne et sa taille, ses valeurs autorisées si c'est un type enum, et ses permissions associées. Par défaut, seuls les objets créés par les utilisateurs sont affichés ; fournissez un motif ou le modificateur S pour afficher les objets système.

`\du[S+] [ pattern ]`

Liste les rôles de la base de données. (Depuis que les concepts des « utilisateurs » et « groupes » ont été unifiés en des « rôles », cette commande est équivalent à \dg.) Par défaut, seuls les rôles créés par des utilisateurs sont affichés. Ajoutez le modificateur S pour inclure les rôles système. Si *motif* est indiqué, seuls les rôles dont le nom correspond au motif sont listés. Si la forme \du + est utilisée, des informations supplémentaires sont affichées pour chaque rôle ; actuellement, cela ajoute le commentaire pour chaque rôle.

`\dx[+] [ motif ]`

Affiche les extensions installées. Si *`motif`* est spécifié, seules les entensions dont le nom correspond au motif sont affichées. Avec la forme `\dx+`, tous les objets dépendants de chacune des extensions correspondantes sont également listés.

`\dX [ motif ]`

Liste les statistiques étendues. Si *`motif`* est indiqué, seules les statistiques étendues dont le nom correspond à ce motif sont listées.

Le statut de chaque type de statistique étendue est affichée dans une colonne nommée d'après le type de statistique (par exemple *Ndistinct*). `defined` signifie qu'il a été réclamé lors de la création des statistiques, et NULL signifie qu'il n'a pas été réclamé. Vous pouvez utiliser `pg_stats_ext` si vous voulez savoir si la commande `ANALYZE` a été exécutée et si les statistiques sont disponibles pour le planificateur.

`\dy[+] [ motif ]`

Liste les triggers d'événements. Si *`motif`* est indiqué, seuls les triggers d'événements dont les noms correspondent au motif sont listés. Si `+` est ajouté au nom de la commande, chaque objet est listé avec sa description.

`\e` (or `\edit`) `[ nomfichier ] [ numero_ligne ]`

Si *`nomfichier`* est spécifié, le fichier est édité ; en quittant l'éditeur, le contenu du fichier est recopié dans le tampon de requête. Si aucun paramètre *`nomfichier`* n'est fourni, le tampon de requête courant est copié dans un fichier temporaire qui édité de la même manière. Ou bien, si le tampon actuel de requête est vide, la dernière requête exécutée est copiée vers un fichier temporaire et éditée de la même manière.

Si vous éditez un fichier ou la requête précédente, et que vous quittez l'éditeur sans modifier le fichier, le tampon de requête est effacé. Sinon le nouveau tampon de requête est ensuite ré-analysé suivant les règles habituelles de psql, où le tampon complet est traité comme une seule ligne. Toute requête complète est exécutée immédiatement ; c'est-à-dire que si le tampon de requête contient ou se termine par un point-virgule, tout ce qui précède est exécuté et supprimé du tampon de requête. Ce qui reste dans le tampon de requête est réaffiché. En tapant point-virgule ou `\g`, le contenu sera envoyé, tandis que `\r` annulera en effaçant le tampon de requête. Traiter le buffer comme une ligne unique affecte principalement les métacommandes : tout ce qui se trouve dans le tampon après une métacommande sera pris en tant qu'argument(s) de la métacommande, même si cela s'étend sur plusieurs lignes (du coup, vous ne pouvez pas faire de scripts de cette façon. Utilisez `\i` pour cela).

Si vous indiquez un numéro de ligne, psql positionnera le curseur sur cette ligne du fichier ou du tampon de requête. Notez que si un seul argument comportant uniquement des caractères numériques est fourni à la commande, psql considère qu'il s'agit d'un numéro de ligne, et non pas un nom de fichier.

> **Astuce**
>
> Voir dans Environnement, ci-dessous, comment configurer et personnaliser votre éditeur.

`\echo texte [ ... ]`

Affiche les arguments évalués sur la sortie standard, séparés par des espaces et suivis par une nouvelle ligne. Ceci peut être utile pour intégrer des informations sur la sortie des scripts. Par exemple :

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

Si le premier argument est `-n` sans guillemets, alors la fin de ligne n'est pas écrite (ni le premier argument).

> **Astuce**
>
> Si vous utilisez la commande `\o` pour rediriger la sortie de la requête, vous pouvez aussi utiliser `\qecho` au lieu de cette commande. See also `\warn`.

`\ef [ *description_fonction* [ *line_number* ] ]`

Cette commande récupère et édite la définition de la fonction ou procédure désignée sous la forme d'une commande `CREATE OR REPLACE FUNCTION` ou `CREATE OR REPLACE PROCEDURE`. L'édition est faite de la même façon que pour `\edit`. Si vous quittez l'éditeur sans sauvegarder, la requête est annulée. Si vous sauvegardez et quittez l'éditeur, la commande mise à jour est exécutée immédiatement si vous y ajoutez un point-virgule. Sinon elle est ré-affichée tapez `;` ou `\g` pour l'envoyer, ou `\r` pour l'annuler.

La fonction cible peut être spécifiée par son nom seul, ou par son nom et ses arguments, par exemple `foo(integer, text)`. Les types d'arguments doivent être fournis s'il y a plus d'une fonction du même nom.

Si aucune fonction n'est spécifiée, un modèle d'ordre `CREATE FUNCTION` vierge est affiché pour édition.

Si vous indiquez un numéro de ligne, psql positionnera le curseur sur cette ligne dans le corps de la fonction. (Notez que le corps de la fonction, typiquement, ne commence pas sur la première ligne du fichier.)

Contrairement à la plupart des autres métacommandes, l'intégralité du reste de la ligne est toujours pris en compte en tant qu'argument(s) de `\ef`, et ni l'interpolation des variables ni la substitution par guillemets inverses ne seront effectuées sur les arguments.

> **Astuce**
>
> Voir dans Environnement, ci-dessous, la façon de configurer et personnaliser votre éditeur.

`\encoding [ *codage* ]`

Initialise l'encodage du jeu de caractères du client. Sans argument, cette commande affiche l'encodage actuel.

`\errverbose`

Répète le message d'erreur le plus récent avec une verbosité maximale, comme si `VERBOSITY` était configuré à `verbose` et `SHOW_CONTEXT` à `always`

`\ev [ *nom_vue* [ *numero_ligne* ] ]`

Cette commande récupère et édite la définition de la vue désignée, sous la forme d'une commande `CREATE OR REPLACE VIEW`. L'édition se termine de la même façon que pour `\edit`. Si vous quittez l'éditeur sans sauvegarder, la requête est annulée. Si vous sauvegardez et quittez l'éditeur, la commande mise à jour est exécutée immédiatement si vous y ajoutez un point-virgule. Sinon, elle est réaffichée ; saisir un point- virgule ou `\g` pour l'envoyer, ou `\r` pour annuler.

Si aucune vue n'est indiquée, un `CREATE VIEW` modèle est présenté pour l'édition.

Si un numéro de ligne est indiqué, psql positionnera le curseur sur la ligne indiquée pour la définition de la vue.

Contrairement à la majorité des autres métacommandes, l'intégralité du reste de la ligne est toujours pris en compte en tant qu'arguments de `\ev`, et ni l'interpolation des variables ni la substitution par guillemets inverses ne seront effectuées sur les arguments.

`\f [ chaîne ]`

Initialise le champ séparateur pour la sortie de requête non alignée. La valeur par défaut est la barre verticale (`|`). C'est équivalent à `\pset fieldsep`.

`\g [ (option=value [...]) ] [ filename ]`
`\g [ (option=value [...]) ] [ |command ]`

Envoie le tampon de requête en entrée vers le serveur pour exécution.

Si des parenthèses apparaissent après `\g`, elles entourent une liste séparée par des espaces de clauses d'option formatées *`option=value`*, qui sont interprétées de la même façon que des commandes `\pset` *`option value`*, mais prennent effet seulement pour la durée de cette requête. Dans cette liste, les espaces ne sont pas permis autour des signes `=` mais sont requis entre chaque clause d'option. Si `=`*`value`* sont omis, l'option nommée *`option`* est modifiée de la même façon que pour `\pset` *`option`* sans *`value`* explicite.

Si un argument *`filename`* ou `|`*`command`* est donné, la commande stocke en option la sortie de la requête dans *`nomfichier`* ou envoie dans un tube (*pipe*) la sortie vers un autre shell exécutant *`commande`* au lieu de l'exécuter comme habituellement. Le fichier ou la commande n'est écrit que si la requête renvoie zéro ou plus enregistrements, mais pas si la requête échoue ou s'il s'agit d'une commande SQL ne renvoyant pas de données.

Si le tampon de la requête est vide, la dernière requête envoyée est ré-exécutée à la place. En dehors de cette exception, `\g` sans argument est essentiellement équivalent à un point-virgule. Avec des arguments, `\g` fournit une alternative « one-shot » à la commande `\o`, et permet en plus des ajustements uniques aux options de format de sortie normalement affectées par `\pset`.

Quand le dernier argument débute par `|`, alors l'intégralité du reste de la ligne est pris en tant que *`commande`* à exécuter et ni l'interpolation des variables ni la substitution par guillemets inverses n'y sont effectuées. Le reste de la ligne est simplement passé littéralement au shell.

`\gdesc`

Affiche la description (c'est-à-dire les noms et types de données des colonnes) pour le résultat de la requête contenue dans le tampon. La requête n'est pas réellement exécutée. Cependant, si elle contient une erreur de syntaxe, l'erreur sera rapportée de la façon habituelle.

Si le tampon de requête est vide, la requête la plus récemment envoyée est décrite à la place.

`\getenv psql_var env_var`

Récupère la valeur de la variable d'environnement *`env_var`* et l'affecte à la variable psql variable *`psql_var`*. Si *`env_var`* n'est pas défini dans l'environnement du processus psql, *`psql_var`* n'est pas modifié. Par exemple :

```
=> \getenv home HOME
=> \echo :home
/home/postgres
```

`\gexec`

Envoie le tampon de requête actuel au serveur, puis traite chaque colonne de chaque ligne du résultat de la requête (s'il y en a) comme une requête à exécuter. Par exemple, pour créer un index sur chaque colonne de `ma_table` :

```
=> SELECT format('create index on ma_table(%I)', attname)
-> FROM pg_attribute
-> WHERE attrelid = 'ma_table'::regclass AND attnum > 0
-> ORDER BY attnum
-> \gexec
CREATE INDEX
CREATE INDEX
CREATE INDEX
CREATE INDEX
```

Les requêtes générées sont exécutées dans l'ordre dans lequel les lignes sont renvoyées, et de gauche à droite sur chaque ligne s'il y a plus d'une colonne. Les champs NULL sont ignorés. Les requêtes générées sont envoyées litéralement au serveur pour traitement, donc elles ne peuvent pas être des métacommandes psql ni contenir des références de variables psql. Si une requête individuelle échoue, l'exécution des requêtes suivantes continue, sauf si `ON_ERROR_STOP` est configuré. L'exécution de chaque requête est sujette au traitement de `ECHO`. (Configurer `ECHO` à `all` ou à `queries` est souvent conseillé lors de l'utilisation de `\gexec`.) La trace de requêtes, le mode étape par étape, le chronométrage et les autres fonctionnalités d'exécution des requêtes s'appliquent aussi à chaque requête générée.

Si le tampon de requête courant est vide, la dernière requête envoyée est ré-exécutée à la place.

`\gset [ *préfixe* ]`

Envoie la requête courante du tampon au serveur et stocke le résultat de la requête dans des variables psql (voir Variables ci-dessous). La requête à exécuter doit renvoyer exactement une ligne. Chaque colonne de la ligne est enregistrée dans une variable séparée, nommée de la même façon que la colonne. Par exemple :

```
=> SELECT 'bonjour' AS var1, 10 AS var2
-> \gset
=> \echo :var1 :var2
bonjour 10
```

Si vous précisez un préfixe *`préfixe`*, cette chaîne est ajoutée aux noms de colonne de la requête pour créer les noms de variable à utiliser :

```
=> SELECT 'bonjour' AS var1, 10 AS var2
-> \gset result_
=> \echo :result_var1 :result_var2
bonjour 10
```

Si le résultat d'une colonne est NULL, la variable correspondante n'est pas initialisée.

Si la requête échoue ou ne renvoie pas une ligne, aucune variable n'est modifiée.

Si le tampon de requête courant est vide, la dernière requête envoyée est ré-exécutée à la place.

`\gx [ (*option=value* [...]) ] [ *filename* ]`
`\gx [ (*option=value* [...]) ] [ |*command* ]`

`\gx` est équivalent à `\g`, sauf qu'il force le mode de sortie étendue pour cette requête, comme si `expanded=on` était inclus dans la liste des options `\pset`. Voir aussi `\x`.

`\h` (ou `\help`) `[ commande ]`

Fournit la syntaxe sur la commande SQL spécifiée. Si *`commande`* n'est pas spécifiée, alors psql liste toutes les commandes pour lesquelles une aide en ligne est disponible. Si *`commande`* est un astérisque (`*`), alors l'aide en ligne de toutes les commandes SQL est affichée.

Contrairement à la plupart des autres métacommandes, l'intégralité du reste de la ligne est toujours pris en compte en tant qu'argument(s) de `\help`, et ni l'interpolation des variables ni la substitution par guillemets inverses ne seront effectuées sur les arguments.

> **Note**
>
> Pour simplifier la saisie, les commandes qui consistent en plusieurs mots n'ont pas besoin d'être entre guillemets. Du coup, il est correct de saisir **`\help alter table`**.

`\H` ou `\html`

Active le format d'affichage HTML des requêtes. Si le format HTML est déjà activé, il est basculé au format d'affichage défaut (texte aligné). Cette commande existe pour la compatibilité et la praticité, mais voyez `\pset` pour configurer les autres options d'affichage.

`\i` ou `\include` *`nomfichier`*

Lit l'entrée à partir du fichier *`nomfichier`* et l'exécute comme si elle avait été saisie sur le clavier.

Si *`nomfichier`* est `-` (tiret), l'entrée standard est lue jusqu'à une indication EOF ou la métacommande `\q`. Ceci peut être utilisé pour intercaler des entrées interactives entre des entrées de fichiers. Notez que le comportement de Readline ne sera activé que s'il est actif au niveau supérieur.

> **Note**
>
> Si vous voulez voir les lignes sur l'écran au moment de leur lecture, vous devez initialiser la variable `ECHO` à `all`.

`\if` *`expression`*
`\elif` *`expression`*
`\else`
`\endif`

Ce groupe de commandes implémente les blocs conditionnels imbriqués. Un bloc conditionnel doit commencer par un `\if` et se terminer par un `\endif`. Entre les deux, il peut y avoir plusieurs clauses `\elif`, pouvant être suivies facultativement par une unique clause `\else`. Des requêtes ordinaires et d'autres commandes antislash peuvent apparaître (et c'est généralement le cas) entre les commandes formant le bloc conditionnel.

Les commandes `\if` et `\elif` lisent leurs arguments et les évaluent en tant qu'expression booléenne. Si l'expression renvoie `true`, alors le traitement continue normalement ; sinon, les lignes sont ignorées jusqu'à un `\elif`, `\else`, ou `\endif` correspondant. Dès qu'un test `\if` ou `\elif` a réussi, les arguments des commandes `\elif` ultérieures du même bloc ne sont pas évaluées mais sont traitées comme fausses. Les lignes qui suivent un `\else` ne sont traitées que si aucune commande `\if` or `\elif` correspondante n'a réussie.

L'argument d'*`expression`* d'une commande `\if` or `\elif` est soumis à l'interpolation des variables et la substitution par guillemets inverses, tout comme n'importe quelle autre commande

antislash. Après cela, il est évalué comme la valeur d'une variable d'option on/off. Une valeur valide est n'importe quelle correspondance non sensible à la case et non-ambiguë parmi : `true`, `false`, `1`, `0`, `on`, `off`, `yes`, `no`. Par exemple, `t`, `T` et `tR` seront tous considérés comme `true`.

Les expressions ne s'évaluant pas correctement à vrai ou faux généreront un avertissement et seront traitées comme fausses.

Les lignes qui sont évitées sont analysées syntaxiquement pour identifier les requêtes et les commandes antislash, mais les requêtes ne sont pas envoyées au serveur, et les commandes antislash autres que conditionnelles (`\if`, `\elif`, `\else`, `\endif`) sont ignorées. Les commandes conditionnelles sont vérifiées seulement pour valider l'emboîtement. Les références des variables des lignes évitées ne sont pas interpolées et les substitutions par guillemets inverses ne seront pas effectuées non plus.

Toutes les commandes antislash d'un bloc conditionnel doivent apparaître dans le même fichier source. Si EOF est atteint dans le fichier d'entrée principal ou un fichier `\include` avant que tous les blocs `\if` locaux ne soient fermés, alors psql générera une erreur.

Voici un exemple :

```
-- vérifier l'existence de deux enregistrements distincts dans
 la base et
-- enregistrer les résultats dans deux variables psql
 différentes
SELECT
    EXISTS(SELECT 1 FROM customer WHERE customer_id = 123) as
 est_client,
    EXISTS(SELECT 1 FROM employee WHERE employee_id = 456) as
 est_employe
\gset
\if :est_client
    SELECT * FROM customer WHERE customer_id = 123;
\elif :est_employe
    \echo 'est un employé mais pas un client'
    SELECT * FROM employee WHERE employee_id = 456;
\else
    \if yes
        \echo 'ni un client ni un employé'
    \else
        \echo 'ce message ne s\'affichera jamais'
    \endif
\endif
```

`\ir` ou `\include_relative` *`nom_fichier`*

La commande `\ir` est similaire à `\i`, mais résout les chemins différemment. Lors d'une exécution en mode interactif, les deux commandes se comportent de la même façon. Néanmoins, lorsqu'elles sont appelées par un script, `\ir` interprète les chemins à partir du répertoire où le script est enregistré, plutôt qu'à partir du répertoire courant.

`\l[+]` ou `\list[+]` [*`motif`*]

Liste les bases de données du serveur en indiquant leur nom, propriétaire, encodage de caractères, et droits d'accès. Si *`pattern`* est spécifié, seules les bases de données dont le nom correspond au motif sont listées. Si `+` est ajouté à la fin de la commande, la taille des bases, les tablespaces par défaut et les descriptions sont aussi affichées. Les tailles ne sont disponibles que pour les bases auxquelles l'utilisateur courant a le droit `CONNECT` ou si l'utilisateur courant est un superutilisateur ou s'il a le droit du rôle `pg_read_all_stats`.

`\lo_export` *`loid nomfichier`*

Lit l'objet large d'OID *`loid`* à partir de la base de données et l'écrit dans *`nomfichier`*. Notez que ceci est subtilement différent de la fonction serveur `lo_export`, qui agit avec les droits de l'utilisateur avec lequel est exécuté le serveur de base de données et sur le système de fichiers du serveur.

> **Astuce**
>
> Utilisez `\lo_list[+]` pour trouver l'OID de l'objet large. Si `+` est ajoutée au nom de la commande, chaque Large Object est listé dans ses droits associés, s'il en a.

`\lo_import` *`nomfichier`* [*`commentaire`*]

Stocke le fichier dans un Large Object PostgreSQL. En option, il associe le commentaire donné avec l'objet. Exemple :

```
foo=> \lo_import '/home/pierre/pictures/photo.xcf' 'une
photo de moi'
lo_import 152801
```

La réponse indique que le Large Object a reçu l'ID 152801, qui peut être utilisé pour accéder de nouveau à l'objet créé. Pour une meilleure lisibilité, il est recommandé de toujours associer un commentaire compréhensible par un humain avec chaque objet. Les OID et les commentaires sont visibles avec la commande `\lo_list`.

Notez que cette commande est subtilement différente de la fonction serveur `lo_import` car elle agit en tant qu'utilisateur local sur le système de fichier local plutôt qu'en tant qu'utilisateur du serveur et de son système de fichiers.

`\lo_list`

Affiche une liste de tous les Large Objects PostgreSQL actuellement stockés dans la base de données, avec tous les commentaires fournis par eux.

`\lo_unlink` *`loid`*

Supprime le Large Object d'OID *`loid`* de la base de données.

> **Astuce**
>
> Utilisez `\lo_list` pour trouver l'OID d'un Large Object.

`\o` ou `\out` [*`nomfichier`*]
`\o` ou `\out` [`|`*`commande`*]

S'arrange pour sauvegarder les résultats des prochaines requêtes dans le fichier *`nomfichier`* ou d'envoyer les résultats à la commande shell *`commande`*. Si aucun argument n'est fourni, le résultat de la requête va sur la sortie standard.

Si l'argument commence par `|`, alors l'intégralité du reste de la ligne est considérée en tant que *`commande`* à exécuter et ni l'interpolation des variables ni la substitution par guillemets inverses ne seront effectuées. Le reste de ligne est simplement envoyée littéralement au shell.

Les « résultats de requête » incluent toutes les tables, réponses de commande et messages d'avertissement obtenus du serveur de bases de données, ainsi que la sortie de différentes

commandes antislash qui envoient des requêtes à la base de données (comme `\d`), mais sans message d'erreur.

> **Astuce**
>
> Pour intercaler du texte entre des résultats de requête, utilisez `\qecho`.

`\p` ou `\print`

Affiche le tampon de requête actuel sur la sortie standard. Si le tampon de requête actuel est vide, la requête la plus récemment exécutée est affichée à la place.

`\password [ nom_utilisateur ]`

Modifie le mot de passe de l'utilisateur indiqué (par défaut, l'utilisateur en cours). Cette commande demande le nouveau mot de passe, le chiffre et l'envoie au serveur avec la commande `ALTER ROLE`. Ceci vous assure que le nouveau mot de passe n'apparaît pas en clair dans l'historique de la commande, les traces du serveur ou ailleurs.

`\prompt [ texte ] nom`

Demande la saisie d'un texte par l'utilisateur. Ce texte sera affecté à la variable *`nom`*. Une chaîne supplémentaire, *`texte`*, peut être donnée. (Pour pouvoir saisir plusieurs mots, entourez le texte par des guillemets simples.)

Par défaut, `\prompt` utilise le terminal pour les entrées et sorties. Néanmoins, si la bascule `-f` est utilisée, `\prompt` utilise l'entrée et la sortie standard.

`\pset [ option [ valeur ] ]`

Cette commande initialise les options affectant l'affichage des tableaux de résultat de requête. *`option`* décrit l'option à initialiser. La sémantique de *`valeur`* varie en fonction de l'option sélectionnée. Pour certaines options, omettre *`valeur`* a pour conséquence de basculer ou désactiver l'option, tel que cela est décrit pour chaque option. Si aucun comportement de ce type n'est mentionné, alors omettre *`valeur`* occasionne simplement l'affichage de la configuration actuelle.

`\pset` sans aucun argument affiche l'état actuel de toutes les options d'affichage.

Les options ajustables d'affichage sont :

`border`

Le *`valeur`* doit être un nombre. En général, plus grand est ce nombre, plus les tables ont de bordures et de lignes mais ceci dépend du format. Dans le format HTML, cela se traduira directement en un attribut `border=...`. Dans la plupart des autres formats, seules les valeurs 0 (sans bordure), 1 (lignes interne de séparation) et 2 (cadre du tableau) ont un sens, et les valeurs au-dessus de 2 seront traitées de la même façon que `border = 2`. Les formats `latex` et `latex-longtable` autorisent en plus une valeur de 3 pour ajouter des lignes de séparation entre les lignes de données.

`columns`

Positionne la largeur pour le format `wrapped` , ainsi que la largeur à partir de laquelle la sortie est suffisamment longue pour nécessiter le paginateur ou pour basculer sur l'affichage vertical dans le mode étendu automatique. Si l'option est positionnée à zéro (la valeur par défaut), la largeur de la colonne est contrôlée soit par la variable d'environnement `COLUMNS`, soit par la largeur d'écran détectée si `COLUMNS` n'est pas positionnée. De plus, si `columns` vaut zéro, alors le format `wrapped` affecte seulement la sortie écran. Si `columns` ne vaut

pas zéro, alors les sorties fichier et tubes (*pipes*) font l'objet de retours à la ligne à cette largeur également.

`csv_fieldsep`

Spécifie le séparateur de champ à utiliser pour le format de sortie CSV. Lorsque le caractère de séparation apparaît dans la valeur d'un champ, le champ est affiché encadré par des guillemets, suivant les règles standards du CSV. La valeur par défaut est une virgule.

`expanded` (ou `x`)

Si une *`valeur`* est précisée, elle doit être soit `on` soit `off`, ce qui activera ou désactivera le mode étendu, soit `auto`. Si *`valeur`* est omis, la commande bascule le paramètre entre les valeurs on et off. Quand le mode étendu est activé, les résultats des requêtes sont affichés sur deux colonnes, avec le nom de la colonne sur la gauche et ses données sur la droite. Ce mode est utile si la donnée ne tient pas sur l'écran dans le mode « horizontal » habituel. Dans le mode auto, le mode étendu est utilisé quand la sortie de la requête a plus d'une colonne et est plus large que l'écran. Sinon, le mode habituel est utilisé. Le mode auto est seulement intéressant lors de l'utilisation des formats aligné et `wrapped` . Si d'autres formats sont sélectionnés, il se comporte toujours comme si le mode étendu était désactivé.

`fieldsep`

Indique le séparateur de champ à utiliser dans le mode d'affichage non aligné. De cette façon, vous pouvez créer, par exemple une sortie séparée par des tabulations, que d'autres programmes pourraient préférer. Pour configurer une tabulation comme champ séparateur, saisissez `\pset fieldsep '\t'`. Le séparateur de champ par défaut est `'|'` (une barre verticale).

`fieldsep_zero`

Configure le séparateur de champs pour qu'il utilise un octet zéro dans le format non aligné en sortie.

`footer`

Si le paramètre *`valeur`* est précisé, il doit valoir soit `on`, soit `off`, ce qui a pour effet d'activer ou de désactiver l'affichage du pied de tableau (le compte : `(`*`n`*` rows)`). Si le paramètre *`valeur`* est omis, la commande bascule entre l'affichage du pied de table ou sa désactivation.

`format`

Initialise le format d'affichage parmi `aligned`, `asciidoc`, `csv`, `html`, `latex`, `latex-longtable`, `troff-ms`, `unaligned` ou `wrapped`. Les abréviations uniques sont autorisées.

Le format `aligned` est le format de sortie standard, lisible par les humains, plaisamment formaté ; c'est le format par défaut.

Le format `unaligned` écrit toutes les colonnes d'un enregistrement sur une seule ligne, séparées par le séparateur de champ courant. Ceci est utile pour créér des sorties qui doivent être lues par d'autres programmes au format séparé par des caractère tabulation ou par des virgules, par exemple. Toutefois, le caractère séparateur n'est pas géré spécifiquement s'il apparaît dans la valeur d'une colonne ; le format CSV peut être mieux adapté pour ces cas de figure.

Le format `csv` sort les valeurs des colonnes séparées par des virgules, en appliquant les règles d'échappement décrites dans la RFC 4180[1]. Cette sortie est compatible avec le format CSV de la commande serveur `COPY`. Une ligne d'entête avec les noms des colonnes est produite

[1] https://datatracker.ietf.org/doc/html/rfc4180

sauf si le paramètre `tuples_only` est à `on`. Le titre et le pied de table ne figurent pas dans la sortie. Chaque ligne se termine par la séquence de fin de ligne qui dépend du système d'exploitation, et qui est typiquement un seul caractère de nouvelle ligne (`\n`) sur les systèmes Unix, ou une séquence constituée d'un retour chariot et nouvelle ligne (`\r\n`) sur Microsoft Windows. Un caractère de séparation de champs autre que la virgule peut être choisi avec `\pset csv_fieldsep`.

Le format `wrapped` est comme `aligned`, sauf qu'il retourne à la ligne dans les données de grande taille afin que la sortie tienne dans la largeur de colonne cible. La largeur cible est déterminée comme décrit à l'option `columns`. Notez que psql n'essaie pas de revenir à la ligne dans les titres de colonnes. Par conséquent, si la largeur totale nécessaire pour le titre de colonne est plus grande que la largeur cible, le format `wrapped` se comporte de la même manière que `aligned`.

Les formats `asciidoc`, `html`, `latex`, `latex-longtable` et `troff-ms` produisent des tableaux destinées à être inclus dans des documents utilisant les langages de balisage respectifs. Ce ne sont pas des documents complets ! Ce n'est pas forcément nécessaire en HTML mais en LaTeX, vous devez avoir une structure de document complet. Le format `latex` utilise l'environnement `tabular` de LaTeX. Le format `latex-longtable` requiert les paquets LaTeX `longtable` et `booktabs`.

`linestyle`

Positionne le style des lignes de bordure sur `ascii`, `old-ascii unicode`. Les abréviations uniques sont autorisées. (Cela signifie qu'une lettre suffit.) La valeur par défaut est `ascii`. Cette option affecte seulement les formats de sortie `aligned` et `wrapped`.

Le style `ascii` utilise les caractères basiques ASCII. Les retours à la ligne dans les données sont représentés par un symbole `+` dans la marge de droite. Quand le format `wrapped` déroule les données d'une ligne à l'autre sans caractère retour à la ligne, un point (`.`) est affiché dans la marge droite de la première ligne et à nouveau dans la marge gauche de la ligne suivante.

Le style `old-ascii` utilise des caractères basiques ASCII, utilisant le style de formatage utilisé dans PostgreSQL 8.4 and et les versions plus anciennes. Les retours à la ligne dans les données sont représentés par un symbole `:` à la place du séparateur de colonnes placé à gauche. Quand les données sont réparties sur plusieurs lignes sans qu'il y ait de caractère de retour à la ligne dans les données, un symbole `;` est utilisé à la place du séparateur de colonne de gauche.

Le style `unicode` utilise les caractères Unicode de dessin de boîte. Les retours à la ligne dans les données sont représentés par un symbole de retour à la ligne dans la marge de droite. Lorsque les données sont réparties sur plusieurs lignes, sans qu'il y ait de caractère de retour à la ligne dans les données, le symbole ellipse est affiché dans la marge de droite de la première ligne, et également dans la marge de gauche de la ligne suivante.

Quand le paramètre `border` vaut plus que zéro, l'option `linestyle` détermine également les caractères utilisés pour dessiner les lignes de bordure. Les simples caractères ASCII fonctionnent partout, mais les caractères Unicode sont plus jolis sur les affichages qui les reconnaissent.

`null`

Positionne la chaîne de caractères à afficher à la place d'une valeur null. Par défaut rien n'est affiché, ce qui peut facilement être confondu avec une chaîne de caractères vide. Par exemple, on peut préférer `\pset null '(null)'`.

`numericlocale`

Si *`valeur`* est précisée, elle doit valoir soit `on`, soit `off` afin d'activer ou désactiver l'affichage d'un caractère dépendant de la locale pour séparer des groupes de chiffres à gauche

du séparateur décimal. Si *valeur* est omise, la commande bascule entre la sortie numérique classique et celle spécifique à la locale.

`pager`

Contrôle l'utilisation d'un paginateur pour les requêtes et les affichages de l'aide de psql. Quand l'option `pager` vaut `off`, le paginateur n'est pas utilisé. Quand l'option `pager` vaut `on`, et que cela est approprié, c'est-à-dire quand la sortie est dirigée vers un terminal et ne tient pas dans l'écran, le paginateur est utilisé. L'option `pager` peut également être positionnée à `always`, ce qui a pour effet d'utiliser le paginateur pour toutes les sorties terminal, que ces dernières tiennent ou non dans l'écran. `\pset pager`, sans préciser *valeur*, bascule entre les états « paginateur activé » et « paginateur désactivé ».

Si la variable d'environnement PSQL_PAGER ou PAGER est configurée, la sortie est envoyée via un tube (*pipe*) dans le programme spécifié. Sinon, une valeur par défaut dépendant de la plateforme (comme `more`) est utilisée.

Lors de l'utilisation de la commande `\watch` pour exécuter de façon répétée une requête, la variable d'environnement PSQL_WATCH_PAGER est utilisée pour trouver le paginateur sur les systèmes Unix. Il est configuré séparément parce que les paginateurs traditionnels pourraient être troublés par le format de cette commande. Il est donc possible d'envoyer la sortie à des outils comprenant ce format de sortie de psql (tel que `pspg --stream`).

`pager_min_lines`

Si `pager_min_lines` est configuré à un numéro supérieur à la hauteur de page, le programme de pagination ne sera appelé que s'il y a au moins ce nombre de lignes à afficher. La configuration par défaut est 0.

`recordsep`

Indique le séparateur d'enregistrement (ligne) à utiliser dans le mode d'affichage non aligné. La valeur par défaut est un caractère de retour chariot.

`recordsep_zero`

Configure le séparateur d'enregistrements pour qu'il utilise un octet zéro dans le format non aligné en sortie.

`tableattr` (ou `T`)

Dans le format HTML, ceci indique les attributs à placer dans la balise `table`. Ce pourrait être par exemple `cellpadding` ou `bgcolor`. Notez que vous ne voulez probablement pas spécifier `border` puisqu'il est déjà pris en compte par `\pset border`. Si *valeur* n'est pas précisée, aucun attribut de table n'est positionné.

Dans le format `latex-longtable`, ceci contrôle la largeur proportionnelle de chaque colonne contenant un type de données aligné à gauche. Il est spécifié en tant que liste de valeurs séparées par des espaces blancs, par exemple `'0.2 0.2 0.6'`. Les colonnes en sortie non spécifiées utilisent la dernière valeur indiquée.

`title` (or `C`)

Initialise le titre de la table pour toutes les tables affichées ensuite. Ceci peut être utilisé pour ajouter des balises de description à l'affichage. Si aucun *valeur* n'est donné, le titre n'est pas initialisé.

`tuples_only` (ou `t`)

Si *valeur* est spécifiée, elle doit valoir soit `on`, soit `off`, ce qui va activer ou désactiver le mode « tuples seulement ». Si *valeur* est omise, la commande bascule entre la sortie normale et la sortie « tuples seulement ». La sortie normale comprend des informations

supplémentaires telles que les en-têtes de colonnes, les titres, et différents pieds. Dans le mode « tuples seulement », seules les données de la table sont affichées.

`unicode_border_linestyle`

Configure le style d'affichage de la bordure pour le style de ligne `unicode` soit à `single` soit à `double`.

`unicode_column_linestyle`

Configure le style d'affichage de la colonne pour le style de ligne `unicode` soit à `single` soit à `double`.

`unicode_header_linestyle`

Configure le style d'affichage de l'en-tête pour le style de ligne `unicode` soit à `single` soit à `double`.

Des exemples d'utilisation de ces différents formats sont disponibles dans la section Exemples.

> **Astuce**
>
> Il existe plusieurs raccourcis de commandes pour `\pset`. Voir `\a`, `\C`, `\f`, `\H`, `\t`, `\T` et `\x`.

`\q` ou `\quit`

Quitte le programme psql. Avec un script, seule l'exécution du script est terminée.

`\qecho` *`texte`* `[ ... ]`

Cette commande est identique à `\echo` sauf que les affichages sont écrits dans le canal d'affichage des requêtes, configuré par `\o`.

`\r` ou `\reset`

Réinitialise (efface) le tampon de requêtes.

`\restrict` *`clé_mode_restreint`*

Entrez en mode « restreint » avec la clé fournie. Dans ce mode, la seule méta-commande autorisée est `\unrestrict` pour quitter le mode restreint. La clé ne doit contenir que des caractères alphanumériques.

Cette commande a principalement comme but d'être utilisé avec les sauvegardes textes générées par pg_dump, pg_dumpall et pg_restore, mais elle peut être utile ailleurs.

Contrairement à la plupart des autres méta-commandes, le reste de la ligne est toujours pris en argument de `\unrestrict`, et ni l'interpolation de la variable ni l'expansion des guillemets inversés ne sont réalisés.

`\s [` *`nomfichier`* `]`

Envoie l'historique de la ligne de commandes de psql dans *`nomfichier`*. Si *`nomfichier`* est omis, l'historique est écrit sur la sortie standard (en utilisant le paginateur si approprié). Cette commande n'est pas disponible si psql a été construit sans le support de Readline.

`\set [` *`nom`* `[` *`valeur`* `[ ... ]]]`

Initialise la variable *`nom`* de psql à *`valeur`* ou, si plus d'une valeur est donnée, à la concaténation de toutes les valeurs. Si un seul argument est donné, la variable est configurée avec une valeur vide. Pour désinitialiser une variable, utilisez la commande `\unset`.

`\set` sans arguments affiche le nom et la valeur de toutes les variables psql actuellement configurées.

Les noms de variables valides peuvent contenir des lettres, chiffres et tirets bas (_). Voir la section Variables ci-dessous pour les détails. Les noms des variables sont sensibles à la casse.

Certaines variables sont spéciales, dans le sens qu'elles contrôlent le comportement de psql ou qu'elles sont mises à jour pour refléter l'état de la connexion. Ces variables sont documentées plus bas dans Variables.

> **Note**
>
> Cette commande est sans relation avec la commande SQL `SET`.

`\setenv` *`nom`* [*`valeur`*]

Configure la variable d'environnement *`nom`* à *`valeur`*, ou si la *`valeur`* n'est pas fournie, désinitialise la variable d'environnement. Par exemple :

```
testdb=> \setenv PAGER less
testdb=> \setenv LESS -imx4F
```

`\sf[+]` *`description_fonction`*

Cette commande récupère et affiche la définition d'une fonction ou procédure sous la forme d'une commande `CREATE OR REPLACE FUNCTION` ou `CREATE OR REPLACE PROCEDURE`. La définition est affichée via le canal de sortie courant, tel que défini par `\o`.

La fonction cible peut être spécifiée par son seul nom, ou bien par ses nom et arguments, par exemple, `foo(integer, text)`. Fournir les types des arguments devient obligatoire si plusieurs fonctions portent le même nom.

Si `+` est ajouté à la commande, les numéros de lignes sont affichés, la ligne 1 débutant à partir du corps de la fonction.

Contrairement à la majorité des autres métacommandes, l'intégralité du reste de la ligne est toujours pris en tant qu'argument(s) de `\sf` et ni l'interpolation des variables ni la substitution par guillemets inverses ne seront effectuées.

`\sv[+]` *`view_name`*

Cette commande récupère et affiche la définition de la vue nommée, dans la forme d'une commande `CREATE OR REPLACE VIEW`. La définition est affichée au travers du canal de sortie actuel, comme configuré par `\o`.

Si `+` est ajouté au nom de commande, les lignes de sorties sont numérotées à partir de 1.

Contrairement à la majorité des autres métacommandes, l'intégralité du reste de la ligne est toujours pris en tant qu'argument(s) de `\sv` et ni l'interpolation des variables ni la substitution par guillemets inverses ne sont effectuées dans les arguments.

`\t`

Bascule l'affichage des en-têtes de nom de colonne en sortie et celle du bas de page indiquant le nombre de lignes. Cette commande est équivalente à `\pset tuples_only` et est fournie pour en faciliter l'accès.

`\T` *`options_table`*

Spécifie les attributs qui seront placés dans le tag `table` pour le format de sortie HTML. Cette commande est équivalente à `\pset tableattr` *`options_table`*.

`\timing [` *`on`* `|` *`off`* `]`

Avec un paramètre, affiche ou supprime l'affichage du temps d'exécution de chaque requête. Sans paramètre, commute l'affichage entre on et off. L'affichage est en millisecondes ; les intervalles plus longs qu'une seconde sont affichés au format minutes:secondes et les champs heures et jours sont ajoutés si nécessaires.

`\unrestrict` *`clé_mode_restreint`*

Quitte le mode restreint (un mode où toutes les autres méta-commandes sont bloquées), à condition que la clé indiquée corresponde à celle donnée à `\restrict` quand le mode restreint a été activé.

Cette commande a principalement comme but d'être utilisé avec les sauvegardes textes générées par pg_dump, pg_dumpall et pg_restore, mais elle peut être utile ailleurs.

`\unset` *`nom`*

Désinitialise (supprime) la variable psql *`nom`*.

La plupart des variables qui contrôlent le comportement de psql ne peuvent pas être désinitialisées ; la commande `\unset` est interprétée comme les remettant à leur valeur par défaut. Voir Variables plus bas.

`\w` ou `\write` *`nomfichier`*
`\w` ou `\write |`*`commande`*

Place le tampon de requête en cours dans le fichier *`nomfichier`* ou l'envoie via un tube à la commande shell *`commande`*. Si le tampon de requête actuel est vide, la dernière requête exécutée est affichée à nouveau.

Si l'argument débute par `|`, alors l'intégralité du reste de la ligne est pris en tant que *`commande`* à exécuter et ni l'interpolation des variables ni la substitution par guillemets inverses n'y sont effectuées. Le reste de la ligne est simplement passé littéralement au shell.

`\warn` *`text`* `[ ... ]`

Cette commande est identique à `\echo` sauf que la sortie sera écrite sur la sortie des erreurs de psql plutôt que la sortie standard.

`\watch [` *`seconds`* `]`

Exécute en répété le tampon de requête courant (comme `\g`) jusqu'à être interrompu explicitement ou que la requête échoue. Attend le nombre spécifié de secondes (2 par défaut) entre les exécutions. Chaque résultat de requête est affiché avec un en-tête qui inclut la chaîne `\pset title` (si c'est activé), l'heure du début de la requête, et l'intervalle.

Si le tampon de requête actuel est vide, la dernière requête envoyée est exécutée à nouveau.

`\x [` *`on`* `|` *`off`* `|` *`auto`* `]`

Configure ou bascule le mode étendu de formatage en table. C'est équivalent à `\pset expanded`.

`\z [` *`motif`* `]`

Liste les tables, vues et séquences avec leur droits d'accès associés. Si un *`motif`* est spécifié, seules les tables, vues et séquences dont le nom correspond au motif sont listées.

Ceci est un alias pour `\dp` (« affichage des droits »).

`\! [ commande ]`

Sans argument, échappe vers un sous-shell ; psql reprendra quand le sous-shell se terminera. Avec un argument, exécute la commande shell *`commande`*.

Contrairement à la majorité des autres métacommandes, l'intégralité du reste de la ligne est toujours pris en compte en tant qu'arguments de `\!`, et ni l'interpolation des variables ni la substitution par guillemets inverses ne seront effectuées sur les arguments. Le reste de la ligne est simplement envoyé directement au shell.

`\? [ thème ]`

Affiche l'aide. Le paramètre optionnel *`thème`* (par défaut à `commands`) sélectionne les parties de psql à expliquer : `commands` décrit les métacommandes de psql ; `options` décrit les options en ligne de commande de psql ; et `variables` affiche de l'aide sur les variables de configuration de psql.

`\;`

Un antislash suivi d'un point-virgule n'est pas une méta-commande comme les commandes précédentes. Cela permet d'ajouter un point-virgule au tampon de requête sans autre traitement.

D'ordinaire, psql envoie une commande SQL au serveur dès qu'il atteint un point-virgule de fin de commande, y compris s'il reste du texte sur la ligne courante. Donc, par exemple :

```
select 1; select 2; select 3;
```

résultera en trois commandes SQL envoyées individuellement au serveur, les résultats de chacune étant affichés avant l'exécution de la commande suivante. Néanmoins, un point-virgule saisi avec un antislash avant, `\;`, ne déclenchera pas le traitement de la commande, pour que la commande précédente et la commande suivante soient en fait combinées et envoyées au serveur comme une seule requête. Par exemple

```
select 1\; select 2\; select 3;
```

résultera en l'envoi des trois commandes SQL en une seule requête lorsque le premier point-virgule sans antislash est atteint. Le serveur exécute une telle requête comme une seule transaction, sauf s'il y a des commandes `BEGIN`/`COMMIT` explicites incluses dans la chaîne pour la diviser en plusieurs transactions. (Voir Section 55.2.2.1 pour plus de détails sur la gestion par le serveur des chaînes de plusieurs requêtes.)

Motifs

Les différentes commandes `\d` acceptent un paramètre *`motif`* pour spécifier le(s) nom(s) d'objet à afficher. Dans le cas le plus simple, un motif est seulement le nom exact de l'objet. Les caractères à l'intérieur du motif sont normalement mis en minuscule comme pour les noms SQL ; par exemple, `\dt FOO` affichera la table nommée `foo`. Comme pour les noms SQL, placer des guillemets doubles autour d'un motif empêchera la mise en minuscule. Si vous devez inclure un guillemet double dans un motif, écrivez-le en double en accord avec les règles sur les identifiants SQL. Par exemple, `\dt "FOO""BAR"` affichera la table nommée `FOO"BAR` (et non pas `foo"bar`). Contrairement aux règles normales pour les noms SQL, vous pouvez placer des guillemets doubles simplement autour d'une partie d'un motif, par exemple `\dt FOO"FOO"BAR` affichera la table nommée `fooFOObar`.

Lorsque le paramètre *`motif`* est complètement absent, la commande `\d` affiche tous les objets visibles dans le chemin de recherche courant -- cela est équivalent à l'utilisation du motif `*`. (Un objet est dit

visible si le schéma qui le contient est dans le chemin de recherche et qu'aucun objet de même type et même nom n'apparaît avant dans le chemin de recherche. Cela est équivalent à dire que l'objet peut être référencé par son nom sans préciser explicitement le schéma.) Pour voir tous les objets de la base quelle que soit leur visibilité, utilisez le motif `*.*` .

À l'intérieur d'un motif, `*` correspond à toute séquence de caractères (et aussi à aucun) alors que `?` ne correspond qu'à un seul caractère. (Cette notation est comparable à celle des motifs de nom de fichier Unix.) Par exemple, `\dt int*` affiche les tables dont le nom commence avec `int`. Mais à l'intérieur de guillemets doubles, `*` et `?` perdent leurs significations spéciales et sont donc traités directement.

Un motif de relation qui contient un point (`.`) est interprété comme le motif d'un nom de schéma suivi par celui d'un nom d'objet. Par exemple, `\dt foo*.*bar*` affiche toutes les tables dont le nom inclut `bar` et qui sont dans des schémas dont le nom commence avec `foo`. Sans point, le motif correspond seulement aux objets qui sont visibles dans le chemin de recherche actuel des schémas. De nouveau, un point dans des guillemets doubles perd sa signification spéciale et est traité directement. Un motif de relation contenant deux points est interprété comme un nom de base suivi d'un motif de nom de schéma suivi d'un motif de nom d'objet. La partie sur le nom de la base de données ne sera pas traitée comme un motif et doit correspondre au nom de la base de données en cours, sinon une erreur sera renvoyée.

Un motif de schéma contenant un point (`.`) est interprété comme un nom de base suivi d'un motif de nom de schéma. Par exemple, `\dn mydb.*foo*` affiche tous les schémas dont le nom inclut `foo`. La partie sur le nom de la base de données ne sera pas traitée comme un motif et doit correspondre au nom de la base de données en cours, sinon une erreur sera renvoyée.

Les utilisateurs avancés peuvent utiliser des expressions rationnelles comme par exemple les classes de caractère (`[0-9]` pour tout chiffre). Tous les caractères spéciaux d'expression rationnelle fonctionnent de la façon indiquée dans Section 9.7.3, sauf pour le `.` qui est pris comme séparateur (voir ci-dessus), l'étoile (`*`) qui est transformée en l'expression rationnelle `.*` et `?` qui est transformée en `.`, et `$` qui est une correspondance littérale. Vous pouvez émuler ces caractères si besoin en écrivant `?` pour `.`, `(`*R*`+|)` pour *R*`*` et `(`*R*`|)` pour *R*`?`. `$` n'est pas nécessaire en tant que caractère d'une expression rationnelle car le motif doit correspondre au nom complet, contrairement à l'interprétation habituelle des expressions rationnelles (en d'autres termes, `$` est ajouté automatiquement à votre motif). Écrivez `*` au début et/ou à la fin si vous ne souhaitez pas que le motif soit ancré. Notez qu'à l'intérieur de guillemets doubles, tous les caractères spéciaux des expressions rationnelles perdent leur signification spéciale et sont traités directement. De plus, ces caractères sont traités littéralement dans les motifs des noms d'opérateurs (par exemple pour l'argument de `\do`).

Fonctionnalités avancées

Variables

psql fournit des fonctionnalités de substitution de variable similaire aux shells de commandes Unix. Les variables sont simplement des paires nom/valeur où la valeur peut être toute chaîne, quelle que soit sa longueur. Le nom doit consister en lettres (incluant les lettres non latines), chiffres et tirets bas.

Pour configurer une variable, utilisez la métacommande psql `\set`. Par exemple :

```
basetest=> \set foo bar
```

initialise la variable `foo` avec la valeur `bar`. Pour récupérer le contenu de la variable, précédez le nom avec un caractère deux-points, par exemple :

```
basetest=> \echo :foo
bar
```

Ceci fonctionne avec les commandes SQL et les métacommandes standards. Il y a plus de détails dans Interpolation SQL, ci-dessous.

Si vous appelez `\set` sans second argument, la variable est initialisée avec une chaîne vide. Pour désinitialiser (c'est-a-dire supprimer) une variable, utilisez la commande `\unset`. Pour afficher les valeurs de toutes les variables, appelez `\set` sans argument.

> **Note**
>
> Les arguments de `\set` sont sujets aux même règles de substitution que les autres commandes. Du coup, vous pouvez construire des références intéressantes comme `\set :foo 'quelquechose'` et obtenir des « liens doux » ou des « variables de variables » comme, respectivement, en Perl ou PHP. Malheureusement (ou heureusement ?), on ne peut rien faire d'utile avec ces constructions. D'un autre côté, `\set bar :foo` est un moyen parfaitement valide de copier une variable.

Un certain nombre de ces variables sont traitées d'une façon particulière par psql. Elles représentent certaines configurations d'options pouvant être changées à l'exécution en modifiant la valeur de la variable ou, dans certains cas, représentent un état modifiable de psql. La convention veut que tous les noms de variables traités spécialement utilisent des lettres ASCII en majuscule (et éventuellement des chiffres et des tirets bas). Pour s'assurer une compatibilité maximum dans le futur, éviter d'utiliser de tels noms de variables pour vos propres besoins.

Les variables qui contrôlent le comportement de psql ne peuvent pas être désinitialisées ou se voir affecter des valeurs incorrectes. Une commande `\unset` est autorisée mais interprétée comme remettant la variable à sa valeur par défaut. Une commande `\set` sans second argument est interprétée comme affectant `on` à la variable, pour les variables de contrôle qui acceptent cette valeur, et sera rejeté pour les autres. Les variables de contrôle qui acceptent les valeurs `on` et `off` accepteront également d'autres formes communes d'écriture des valeurs booléennes, comme `true` et `false`.

Voici une liste des variables spéciales :

`AUTOCOMMIT`

Si actif (`on`, valeur par défaut), chaque commande SQL est automatiquement validée si elle se termine avec succès. Pour suspendre la validation dans ce mode, vous devez saisir une commande SQL `BEGIN` ou `START TRANSACTION`. Lorsqu'elle est désactivée (`off`) ou non initialisée, les commandes SQL ne sont plus validées tant que vous ne lancez pas explicitement `COMMIT` ou `END`. Le mode sans autocommit fonctionne en lançant implicitement un `BEGIN`, juste avant toute commande qui n'est pas déjà dans un bloc de transaction et qui n'est pas elle-même un `BEGIN` ou une autre commande de contrôle de transaction, ou une commande qui ne peut pas être exécutée à l'intérieur d'un bloc de transaction (comme `VACUUM`).

> **Note**
>
> Dans le mode sans autocommit, vous devez annuler explicitement toute transaction échouée en saisissant `ABORT` ou `ROLLBACK`. Gardez aussi en tête que si vous sortez d'une session sans validation, votre travail est perdu.

> **Note**
>
> Le mode auto-commit est le comportement traditionnel de PostgreSQL alors que le mode sans autocommit est plus proche des spécifications SQL. Si vous préférez sans autocommit, vous pouvez le configurer dans le fichier `psqlrc` global du système ou dans votre fichier `~/.psqlrc`.

COMP_KEYWORD_CASE

Détermine la casse à utiliser lors de la complétion d'un mot clé SQL. S'il est configuré à `lower` ou `upper`, le mot complété sera, respectivement, en minuscule ou en majuscule. Si la variable est configurée à `preserve-lower` ou `preserve-upper` (valeur par défaut), le mot complété sera dans la casse du mot déjà saisi, mais les mots qui n'ont pas eu un début de saisie seront complétés, respectivement, soit en minuscule soit en majuscule.

DBNAME

Le nom de la base de données à laquelle vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être changé ou désinitialisé.

ECHO

Si cette variable est initialisée à `all`, toutes les lignes non vides saisies sont envoyées sur la sortie standard tout de suite après leur lecture. (Ceci ne s'applique pas aux lignes lues de façon interactive.) Pour sélectionner ce comportement au lancement du programme, utilisez l'option `-a`. Si `ECHO` vaut `queries`, psql affiche chaque requête sur la sortie standard comme elle est envoyée au serveur. L'option pour choisir ce comportement est `-e`. Si elle est configurée à `errors`, seules les requêtes échouées seront affichées sur la sortie standard des erreurs. L'option en ligne de commande pour ce comportement est `-b`. Si elle est configurée à `none` (valeur par défaut), alors aucune requête n'est affichée.

ECHO_HIDDEN

Quand cette variable est initialisée à `on` et qu'une commande antislash est envoyée à la base de données, la requête est d'abord affichée. Cette fonctionnalité vous aide à étudier le fonctionnement interne de PostgreSQL et fournir des fonctionnalités similaires dans vos propres programmes. (Pour sélectionner ce comportement au lancement du programme, utilisez l'option `-E`.) Si vous configurez la variable avec la valeur `noexec`, les requêtes sont juste affichées mais ne sont pas réellement envoyées au serveur ni exécutées. La valeur par défaut est `off`.

ENCODING

Le codage courant du jeu de caractères du client. Il est fixé à chaque fois que vous vous connectez à une base de données (y compris au démarrage du programme), et quand vous changez l'encodage avec `\encoding`, mais il peut être changé ou désinitialisé.

ERROR

`true` si la dernière requête SQL a échoué, `false` si elle a réussi. Voir aussi `SQLSTATE`.

FETCH_COUNT

Si cette variable est un entier plus grand que zéro, les résultats des requêtes `SELECT` sont récupérés et affichés en groupe de ce nombre de lignes, plutôt que par le comportement par défaut (récupération de l'ensemble complet des résultats avant l'affichage). Du coup, seule une petite quantité de mémoire est utilisée, quelle que soit la taille de l'ensemble des résultats. Une configuration entre 100 et 1000 est habituellement utilisée lors de l'activation de cette fonctionnalité. Gardez en tête que lors de l'utilisation de cette fonctionnalité, une requête pourrait échouer après avoir affiché quelques lignes.

> **Astuce**
>
> Bien que vous puissiez utiliser tout format de sortie avec cette fonctionnalité, le format par défaut, `aligned`, rend mal car chaque groupe de `FETCH_COUNT` lignes sera formaté séparément, modifiant ainsi les largeurs de colonnes suivant les lignes du groupe. Les autres formats d'affichage fonctionnent mieux.

`HIDE_TABLEAM`

Si cette variable est positionnée à `true`, le détail de la méthode d'accès d'une table n'est pas affiché. C'est surtout utile pour les tests de non-régression.

`HIDE_TOAST_COMPRESSION`

Si cette variable est configurée à `true`, les détails sur la méthode de compression ne sont pas affichés. Ceci est principalement utile pour les tests de régression.

`HISTCONTROL`

Si cette variable est configurée à `ignorespace`, les lignes commençant avec un espace n'entrent pas dans la liste de l'historique. Si elle est initialisée avec la valeur `ignoredups`, les lignes correspondant aux précédentes lignes de l'historique n'entrent pas dans la liste. Une valeur de `ignoreboth` combine les deux options. Si elle est configurée avec `none`), toutes les lignes lues dans le mode interactif sont sauvegardées dans la liste de l'historique.

> **Note**
>
> Cette fonctionnalité a été plagiée sans vergogne sur Bash.

`HISTFILE`

Le nom du fichier utilisé pour stocker l'historique. Si désinitialisé, le nom du fichier sera la valeur de la variable d'environnement `PSQL_HISTORY`. Si celle-ci n'est pas initialisée, la valeur par défaut sera `~/.psql_history` ou `%APPDATA%\postgresql\psql_history` sur Windows. Par exemple, mettre :

```
\set HISTFILE ~/.psql_history-:DBNAME
```

dans `~/.psqlrc` fera en sorte que psql maintienne un historique pour chaque base.

> **Note**
>
> Cette fonctionnalité a été plagiée sans vergogne sur Bash.

`HISTSIZE`

Le nombre maximum de commandes à stocker dans l'historique des commandes (par défaut 500). Aucune limite ne sera appliquée si une valeur négative est donnée.

> **Note**
>
> Cette fonctionnalité a été plagiée sans vergogne sur Bash.

`HOST`

L'hôte du serveur de la base de données sur lequel vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être changé ou désinitialisé.

`IGNOREEOF`

Si configuré à 1 ou inférieur, envoyer un caractère EOF (habituellement **Ctrl**+**D**) dans une session interactive de psql ferme l'application. Si configuré à une valeur numérique supérieure, alors

autant de caractères EOF consécutifs doivent être saisis pour terminer une session interactive. Une valeur non numérique sera interprétée comme valant 10. La valeur par défaut est 0.

> **Note**
>
> Cette fonctionnalité a été plagiée sans vergogne sur Bash.

`LASTOID`

La valeur du dernier OID affecté, renvoyée à partir d'une commande `INSERT` ou `lo_import`. La validité de cette variable est seulement garantie jusqu'à l'affichage du résultat de la commande SQL suivante. Les serveurs PostgreSQL depuis la version 12 ne gèrent plus la colonne système OID, et de ce fait LASTOID sera toujours à 0 à la suite d'une commande `INSERT` sur ces serveurs.

`LAST_ERROR_MESSAGE`
`LAST_ERROR_SQLSTATE`

Le message d'erreur principal et le code SQLSTATE associé pour la plus récente requête en échec dans la session psql en cours, ou une chaîne vide et `00000` s'il n'y a eu aucune erreur dans la session actuelle.

`ON_ERROR_ROLLBACK`

Lorsqu'il est actif (`on`), si une instruction d'un bloc de transaction génère une erreur, cette dernière est ignorée et la transaction continue. Lorsqu'il vaut `interactive`, ces erreurs sont seulement ignorées lors des sessions interactives, mais ne le sont pas lors de la lecture de scripts. Lorsqu'il est configuré à `off` (valeur par défaut), une instruction générant une erreur dans un bloc de transaction annule la transaction complète. Le mode avec `on` fonctionne en exécutant un `SAVEPOINT` implicite pour vous, juste avant chaque commande se trouvant dans un bloc de transaction, et annule jusqu'au point de sauvegarde si la commande échoue.

`ON_ERROR_STOP`

Par défaut, le traitement des commandes continue après une erreur. Quand cette variable est positionnée à `on`, le traitement sera immédiatement arrêté dès la première erreur rencontrée. En mode interactif, psql reviendra à l'invite de commande ; sinon psql quittera en renvoyant le code d'erreur 3 pour distinguer ce cas des conditions d'erreurs fatales, qui utilisent le code 1. Dans tous les cas, tout script en cours d'exécution (le script de plus haut niveau, s'il y a, et tout autre script qui pourrait avoir été appelé) sera terminé immédiatement. Si la chaîne de commande de plus haut niveau contient plusieurs commandes SQL, le traitement s'arrêtera à la commande en cours.

`PORT`

Le port du serveur de la base de données sur lequel vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être changé ou désinitialisé.

`PROMPT1`
`PROMPT2`
`PROMPT3`

Ils spécifient à quoi doit ressembler l'invite psql. Voir Invite ci-dessous.

`QUIET`

Configurer cette variable à `on` est équivalent à l'option `-q` en ligne de commande. Elle n'est probablement pas très utile en mode interactif.

`ROW_COUNT`

Le nombre de lignes renvoyées ou affectées par la dernière requête SQL, ou 0 si la requête a échoué ou si elle n'a pas renvoyée un nombre de lignes.

`SERVER_VERSION_NAME`
`SERVER_VERSION_NUM`

Le numéro de version du serveur sous la forme d'une chaîne de caractères, par exemple `9.6.2`, `10.1` ou `11beta1`, et sous sa forme numérique, par exemple `90602` ou `100001`. Ces variables sont configurées à chaque fois que vous vous connectez à une base de données (y compris au lancement du programme) mais peuvent être modifiées ou déconfigurées.

`SHOW_ALL_RESULTS`

Quand cette variable est configurée à `off`, seul le dernier résultat d'une requête combinée (`\;`) est affiché au lieu de tous. La valeur par défaut est`on`. Le comportement off est pour la compatibilité avec les anciennes versions de psql.

`SHOW_CONTEXT`

Cette variable peut être configurée avec les valeurs `never`, `errors` ou `always` pour contrôler si les champs `CONTEXT` sont affichés dans les messages du serveur. La valeur par défaut est `errors` (signifiant que ce contexte sera affiché dans les messages d'erreur et non pas dans les notes et avertissements). Ce paramètre n'a pas d'effet quand `VERBOSITY` est configuré à `terse` ou `sqlstate`. (Voir aussi `\errverbose`, à utiliser quand vous voulez une version verbose du dernier message d'erreur reçu.)

`SINGLELINE`

Configurer cette variable à `on` est équivalent à l'option `-S` en ligne de commande.

`SINGLESTEP`

Configurer cette variable à `on` est équivalent à l'option `-s` en ligne de commande.

`SQLSTATE`

Le code d'erreur (voir Annexe A) associé avec l'échec de la dernière requête SQL, ou `00000` si elle a réussi.

`USER`

L'utilisateur de la base de données sur laquelle vous êtes actuellement connecté. Ceci est configuré à chaque fois que vous vous connectez à une base de données (ainsi qu'au lancement du programme) mais peut être changé ou désinitialisé.

`VERBOSITY`

Cette variable peut être configurée avec les valeurs `default`, `verbose` (bavard), `terse` (succinct) ou `sqlstate` pour contrôler la verbosité des rapports d'erreurs. (Voir aussi `\errverbose` à utiliser quand vous avez besoin d'une version verbeuse de l'erreur que vous venez de récupérer.)

`VERSION`
`VERSION_NAME`
`VERSION_NUM`

Ces variables sont configurées au démarrage du programme pour refléter la version de psql respectivement sous la forme d'une chaîne de caractères, d'une chaîne courte (par exemple

`9.6.2`, `10.1` ou `11beta1`) d'un nombre (par exemple `90602` ou `100001`). Elles peuvent être modifiées ou désinitialisées.

Interpolation SQL

Une fonctionnalité clé des variables psql est que vous pouvez les substituer (« interpolation ») dans des requêtes SQL standards, ainsi qu'en arguments de métacommandes. De plus, psql fournit des fonctionnalités vous assurant que les valeurs des variables utilisées comme constantes et identifiants SQL sont correctement mises entre guillemets. La syntaxe pour l'interpolation d'une valeur sans guillemets est de préfixer le nom de la variable avec le symbole deux-points (`:`). Par exemple :

```
basetest=> \set foo 'ma_table'
basetest=> SELECT * FROM :foo;
```

envoie alors la requête pour la table `ma_table`. Notez que cela peut être dangereux ; la valeur de la variable est copiée de façon litérale, elle peut même contenir des guillemets non fermés, ou bien des commandes antislash. Vous devez vous assurer que cela a du sens à l'endroit où vous les utilisez.

Lorsqu'une valeur doit être utilisée comme une chaîne SQL litérale ou un identifiant, il est plus sûr de s'arranger pour qu'elle soit entre guillemets. Afin de mettre en guillemets la valeur d'une variable en tant que chaîne SQL litérale, écrivez un caractère deux-points, suivi du nom de la variable entouré par des guillemets simples. Pour mettre entre guillemet la valeur en tant qu'identifiant SQL, écrivez un caractère deux-points suivi du nom de la valeur entouré de guillemets doubles. Ces constructions gèrent correctement les guillemets et autres caractères spéciaux intégrés dans la valeur de la variable. L'exemple précédent peut s'écrire de façon plus sûre ainsi :

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :"foo";
```

L'interpolation de variables ne sera pas réalisée à l'intérieur de litéraux et d'identifiants SQL mis entre guillemets. Du coup, une construction comme `':foo'` ne fonctionne pas pour avoir un litéral entre guillemets à partir de la valeur d'une variable (il serait même dangereux que cela fonctionne car ça ne peut pas gérer correctement les guillemets embarqués dans la valeur).

Un exemple de l'utilisation de ce mécanisme est la copie du contenu d'un fichier dans la colonne d'une table. Tout d'abord, chargez le fichier dans une variable puis interpolez la valeur de la valeur en tant que chaîne de caractères :

```
basetest=> \set contenu `cat mon_fichier.txt`
basetest=> INSERT INTO ma_table VALUES (:'contenu');
```

(Notez que cela ne fonctionnera par si le fichier `mon_fichier.txt` contient des octets nuls. psql ne gère pas les octets nuls inclus dans les valeurs de variable.)

Comme des caractères deux-points peuvent légitimement apparaître dans les commandes SQL, une tentative apparente d'interpolation (comme `:nom`, `:'nom'`, ou `:"nom"`) n'est pas remplacée, sauf si la variable nommée est actuellement positionnée. Dans tous les cas, vous pouvez échapper un caractère deux-points avec un antislash pour le protéger des substitutions.

La syntaxe spéciale `:{?`*`name`*`}` renvoie TRUE ou FALSE suivant l'existence ou non de la variable, et est donc toujours substituée, sauf si le symbole deux-points est échappé avec un antislash.

La syntaxe deux-points pour les variables est du SQL standard pour les langages de requête embarqués, comme ECPG. La syntaxe avec les deux-points pour les tranches de tableau et les conversions de types sont des extensions PostgreSQL, qui peut parfois provoquer un conflit avec l'utilisation standard. La syntaxe avec le caractère deux-points pour échapper la valeur d'une variable en tant que chaîne SQL litérale ou identifiant est une extension psql.

Invite

Les invites psql peuvent être personnalisées suivant vos préférences. Les trois variables `PROMPT1`, `PROMPT2` et `PROMPT3` contiennent des chaînes et des séquences d'échappement spéciales décrivant l'apparence de l'invite. L'invite 1 est l'invite normale qui est lancée quand psql réclame une nouvelle commande. L'invite 2 est lancée lorsqu'une saisie supplémentaire est attendue lors de la saisie de la commande, par exemple parce que la commande n'a pas été terminée avec un point-virgule ou qu'un guillemet n'a pas été fermé. L'invite 3 est lancée lorsque vous exécutez une commande SQL `COPY FROM stdin` et que vous devez saisir les valeurs des lignes sur le terminal.

La valeur de la variable prompt sélectionnée est affichée littéralement sauf si un signe pourcentage (%) est rencontré. Suivant le prochain caractère, certains autres textes sont substitués. Les substitutions définies sont :

`%M`

Le nom complet de l'hôte (avec le nom du domaine) du serveur de la base de données ou `[local]` si la connexion est établie via une socket de domaine Unix ou `[local:`*`/répertoire/nom`*`]`, si la socket de domaine Unix n'est pas dans l'emplacement par défaut défini à la compilation.

`%m`

Le nom de l'hôte du serveur de la base de données, tronqué au premier point ou `[local]` si la connexion se fait via une socket de domaine Unix.

`%>`

Le numéro de port sur lequel le serveur de la base de données écoute.

`%n`

Le nom d'utilisateur de la session. (L'expansion de cette valeur peut changer pendant une session après une commande `SET SESSION AUTHORIZATION`.)

`%/`

Le nom de la base de données courante.

`%~`

Comme `%/` mais l'affichage est un ~ (tilde) si la base de données est votre base de données par défaut.

`%#`

Si l'utilisateur de la session est un superutilisateur, alors un # sinon un >. (L'expansion de cette valeur peut changer durant une session après une commande `SET SESSION AUTHORIZATION`.)

`%p`

L'identifiant du processus serveur (PID) pour cette connexion.

`%R`

Dans le prompt 1 normalement =, mais @ si la session est dans une branche inactive d'un bloc conditionnel, ou ^ en mode simple ligne, ou ! si la session est déconnectée de la base (ce qui peut arriver si `\connect` échoue). Dans le prompt 2, `%R` est remplacé par un caractère qui dépend de la raison pour laquelle psql attend des entrées supplémentaires : - si la commande n'est juste pas terminée, mais * s'il y a un commentaire `/* ... */` non terminé, un guillemet simple pour une chaîne de caractères entre guillemets simples non terminée, un guillemet double pour

un identifiant échappé non terminé, un signe dollar pour une chaîne de caractères entre dollars, ou (s'il y a une parenthèse ouvrante sans correspondance. Dans le prompt 3, %R n'a aucun effet.

%x

État de la Transaction : une chaîne vide lorsque vous n'êtes pas dans un bloc de transaction, ou * si vous y êtes, ou ! dans une transaction échouée, ou ? lorsque l'état de la transaction est indéterminé (par exemple parce qu'il n'y a pas de connexion).

%l

Le numéro de ligne dans la requête courante, en partant de 1.

%*chiffres*

Le caractère avec ce code numérique est substitué.

%:*nom*:

La valeur de la variable *nom* de psql. Voir la section Variables pour les détails.

%`*commande*`

La sortie de la *commande*, similaire à la substitution par « guillemets inverse » classique.

%[... %]

Les invites peuvent contenir des caractères de contrôle du terminal qui, par exemple, modifient la couleur, le fond ou le style du texte de l'invite, ou modifient le titre de la fenêtre du terminal. Pour que les fonctionnalités d'édition de ligne de Readline fonctionnent correctement, les caractères de contrôle non affichables doivent être indiqués comme invisibles en les entourant avec %[et %]. Des paires multiples de ceux-ci peuvent survenir à l'intérieur de l'invite. Par exemple :

```
basetest=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# '
```

a pour résultat une invite en gras (1;), jaune sur noir (33;40) sur les terminaux compatibles VT100.

%w

Espace blanc de même taille que la sortie la plus récente de PROMPT1. Cela peut être utilisé comme paramètre PROMPT2, pour que les instructions multi-lignes soient alignées avec la première ligne, mais il n'y a pas de deuxième prompt visible.

Pour insérer un pourcentage dans votre invite, écrivez %%. Les invites par défaut sont '%/%R%x%# ' pour les invites 1 et 2 et '>> ' pour l'invite 3.

> **Note**
>
> Cette fonctionnalité a été plagiée sans vergogne sur tcsh.

Édition de la ligne de commande

psql utilise la bibliothèque Readline ou la bibliothèque libedit, si disponible, pour une édition et recherche de lignes agréable. L'historique de commandes est automatiquement sauvegardée quand psql quitte et est rechargée quand psql démarre. Utiliser les flèches et control-P pour récupérer les lignes précédentes.

Vous pouvez utiliser la complétion avec la tabulation pour remplir des mots clés et des noms d'objets SQL partiellement saisis dan beaucoup de contextes (mais pas tous). Par exemple, au début d'une commande, saisir `ins` et appuyer sur table remplira avec `insert into `. Puis saisir quelques caractères du nom d'une table ou d'un schéma et presser la tabulation terminera le nom ou proposer un menu des complétions possibles quant plusieurs objets correspondent aux premiers caractères saisis. Suivant la bibliothèque utilisée, vous pourriez avoir besoin d'appuyer sur la tabulation plusieurs fois pour obtenir le menu.

La complétion par tabulation des noms d'objets SQL nécessite d'envoyer des requêtes au serveur pour trouver les correspondances possibles. Dans certains contextes, ceci peut interférer avec d'autres opérations. Par exemple, après `BEGIN`, il est souvent trop tard pour lancer `SET TRANSACTION ISOLATION LEVEL` si une requête a été exécutée entre temps pour la complétion. Si vous ne voulez pas du tout de complétion, vous pouvez la désactiver en plaçant ceci dans un fichier nommé `.inputrc` de votre répertoire personnel :

```
$if psql
set disable-completion on
$endif
```

(Ceci n'est pas une fonctionnalité psql mais Readline. Lisez sa documentation pour plus de détails.)

L'option `-n` (`--no-readline`) en ligne de commande peut aussi être utile pour désactiver l'utilisation de Readline pour une simple exécution de psql. Ceci empêche la complétion par tabulation ou l'enregistrement de l'historique de commande, et l'édition de commandes multi-lignes. C'est tout particulièrement utile quand vous avez besoin de copier/coller du texte contenant des caractères de tabulation.

Environnement

`COLUMNS`

Si `\pset columns` vaut zéro, contrôle la largeur pour le format `wrapped` et la largeur pour déterminer si une sortie large a besoin du paginateur ou doit être basculé en format vertical dans le mode automatique étendu.

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Paramètres de connexion par défaut (voir Section 34.15).

`PG_COLOR`

Indique si la couleur doit être utilisée dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

`PSQL_EDITOR`
`EDITOR`
`VISUAL`

Éditeur utilisé par les commandes `\e` `\ef` et `\ev`. Les variables sont examinées dans l'ordre donné ; la première initialisée est utilisée. Si aucun des deux n'est configuré, le système utilise `vi` par défaut sur les systèmes Unix et `notepad.exe` sur les systèmes Windows.

`PSQL_EDITOR_LINENUMBER_ARG`

Lorsque les commandes `\e` ou `\ef` sont utilisées avec un argument spécifiant le numéro de ligne, cette variable doit indiquer l'argument en ligne de commande à fournir à l'éditeur de texte.

Pour les éditeurs les plus courants, tels qu'emacs ou vi, vous pouvez simplement initialiser cette variable avec le signe +. Il faut inclure le caractère d'espacement en fin de la valeur de la variable si la syntaxe de l'éditeur nécessite un espace entre l'option à spécifier et le numéro de ligne. Par exemple :

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line '
```

La valeur par défaut est + sur les systèmes Unix (ce qui correspond à la bonne configuration pour l'éditeur par défaut, `vi`, et est utilisable généralement avec la plupart des éditeurs courants) ; par contre, il n'y a pas de valeur par défaut pour les systèmes Windows.

`PSQL_HISTORY`

Emplacement alternatif pour le fichier d'historique des commandes. L'expansion du symbole ~ est réalisée.

`PSQL_PAGER`
`PAGER`

Si les résultats d'une requête ne tiennent pas sur l'écran, ils sont envoyés à cette commande. Les valeurs typiques sont `more` ou `less`. L'utilisation du paginateur peut être désactivé en configurant `PSQL_PAGER` ou `PAGER` à une chaîne vide ou en ajustant les options relatives au paginateur avec la commande `\pset`. Ces variables sont examinées dans l'ordre listé ; la première qui est configurée est utilisée. Si aucune n'est configurée, le comportement par défaut est d'utiliser `more` sur la plupart des plateformes et `less` sur Cygwin.

`PSQL_WATCH_PAGER`

Quand une requête est exécutée de façon répétée avec la commande `\watch`, nu paginateur n'est pas utilisé par défaut. Ce comportement est modifiable en configurant le paramètre `PSQL_WATCH_PAGER` par une commande de paginateur sur les systèmes Unix. Le paginateur `pspg` (qui ne fait pas partie de PostgreSQL mais est disponible dans un grand nombre de distributions open source) peut afficher la sortie de `\watch` s'il est lancé avec l'option `--stream`.

`PSQLRC`

Emplacement alternatif pour le fichier `.psqlrc` de l'utilisateur. L'expansion du symbôle ~ est réalisée.

`SHELL`

Commande exécutée par la commande `\!`.

`TMPDIR`

Répertoire pour stocker des fichiers temporaires. La valeur par défaut est `/tmp`.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 34.15).

Fichiers

`psqlrc` et `~/.psqlrc`

Sauf si une option `-X` est fournie, psql tente de lire et exécuter les commandes provenant du fichier global au système (`psqlrc`), puis du fichier utilisateur (`~/.psqlrc`) après la connexion à la base de données mais avant d'accepter les commandes interactives. Ces fichiers sont utilisés pour

configurer le client et le serveur à votre goût, généralement en utilisant les commandes `\set` et `SET`.

Le fichier de configuration au niveau système est nommé `psqlrc`. Par défaut, il est placé dans le répertoire de configuration système de l'installation. Il est facilement identifiable en exécutant `pg_config --sysconfdir`. Habituellement, ce répertoire doit être `../etc/` relatif au répertoire contenant les exécutables PostgreSQL. Le répertoire à parcourir peut être configuré explicitement via la variable d'environnement `PGSYSCONFDIR`.

Le fichier de configuration personnel de l'utilisateur est nommé `.psqlrc` et est placé à la racine du répertoire personnel de l'utilisateur. Sur Windows, le fichier de configuration personnel est nommé `%APPDATA%\postgresql\psqlrc.conf`. Dans les deux cas, le chemin du fichier par défaut peut être surchargé en configurant la variable d'environnement `PSQLRC`.

Le fichier niveau système et le fichier de l'utilisateur peuvent être spécifiques à la version de psql en ajoutant un tiret et l'identifiant de la version mineure ou majeure, par exemple `~/.psqlrc-15` ou `~/.psqlrc-15.19`. Le fichier dont la version est la plus proche sera lu à la place d'un fichier sans indication de version. Ces suffixes de version sont ajoutés après avoir déterminer le chemin des fichiers comme expliqué ci-dessus.

`.psql_history`

L'historique de la ligne de commandes est stocké dans le fichier `~/.psql_history` ou `%APPDATA%\postgresql\psql_history` sur Windows.

L'emplacement du fichier historique peut aussi être configuré explicitement avec la variable psql `HISTFILE` ou avec la variable d'environnement `PSQL_HISTORY`.

Notes

- psql fonctionne mieux avec des serveurs de la même version ou d'une version majeure plus ancienne. Les commandes antislashs peuvent échouer si le serveur est plus récent que psql. Néanmoins, les commandes antislashs de la famille `\d` devraient fonctionner avec tous les serveurs jusqu'à la version 9.2, bien que pas nécessairement avec des serveurs plus récents que psql lui-même. Les fonctionnalités générales d'exécution de commandes SQL et d'affichage des résultats des requêtes devraient aussi fonctionner avec les serveurs d'une version majeure plus récente mais ce ne peut être garanti dans tous les cas.

  Si vous voulez utiliser psql pour vous connecter à différentes versions majeures, il est recommandé d'utiliser la dernière version de psql. Autrement, vous pouvez conserver une copie de psql pour chaque version majeure utilisée et vous assurer que la version utilisée correspond au serveur respectif. En pratique, cette complication supplémentaire n'est pas nécessaire.

- Avant PostgreSQL 9.6, l'option `-c` impliquait `-X` (`--no-psqlrc`) ; ceci n'est plus le cas.

- Avant PostgreSQL 8.4, psql autorisait le premier argument d'une commande antislash à une seule lettre à commencer directement après la commande, sans espace supplémentaire. Maintenant, un espace blanc est requis.

Notes pour les utilisateurs sous Windows

psql est construit comme une « application de type console ». Comme les fenêtres console de Windows utilisent un codage différent du reste du système, vous devez avoir une attention particulière lors de l'utilisation de caractères sur 8 bits à l'intérieur de psql. Si psql détecte une page de code problématique, il vous avertira au lancement. Pour modifier la page de code de la console, deux étapes sont nécessaires :

- Configurez la page code en saisissant **`cmd.exe /c chcp 1252`**. (1252 est une page code appropriée pour l'Allemagne ; remplacez-la par votre valeur.) Si vous utilisez Cygwin, vous pouvez placer cette commande dans `/etc/profile`.

- Configurez la police de la console par `Lucida Console` parce que la police raster ne fonctionne pas avec la page de code ANSI.

Exemples

Le premier exemple montre comment envoyer une commande sur plusieurs lignes d'entrée. Notez le changement de l'invite :

```
basetest=> CREATE TABLE ma_table (
basetest(>  premier integer not NULL default 0,
basetest(>  second text)
basetest-> ;
CREATE TABLE
```

Maintenant, regardons la définition de la table :

```
basetest=> \d ma_table
              Table "public.ma_table"
 Column  |  Type   | Collation | Nullable | Default
---------+---------+-----------+----------+---------
 premier | integer |           | not null | 0
 second  | text    |           |          |
```

Maintenant, changeons l'invite par quelque chose de plus intéressant :

```
basetest=> \set PROMPT1 '%n@%m %~%R%# '
pierre@localhost basetest=>
```

Supposons que nous avons rempli la table de données et que nous voulons les regarder :

```
pierre@localhost basetest=> SELECT * FROM ma_table;
 premier | second
---------+--------
       1 | un
       2 | deux
       3 | trois
       4 | quatre
(4 rows)
```

Vous pouvez afficher cette table de façon différente en utilisant la commande `\pset` :

```
pierre@localhost basetest=> \pset border 2
Border style is 2.
pierre@localhost basetest=> SELECT * FROM ma_table;
+---------+--------+
| premier | second |
+---------+--------+
|       1 | un     |
|       2 | deux   |
|       3 | trois  |
|       4 | quatre |
+---------+--------+
(4 rows)
```

```
pierre@localhost basetest=> \pset border 0
Border style is 0.
pierre@localhost basetest=> SELECT * FROM ma_table;
premier second
------- ------
      1 un
      2 deux
      3 trois
      4 quatre
(4 rows)

pierre@localhost basetest=> \pset border 1
Border style is 1.
pierre@localhost testdb=> \pset format csv
Output format is csv.
pierre@localhost basetest=> \pset tuples_only
Tuples only is on.
pierre@localhost basetest=> SELECT second, premier FROM
ma_table;
un,1
deux,2
trois,3
quatre,4
pierre@localhost testdb=> \pset format unaligned
Output format is unaligned.
pierre@localhost testdb=> \pset fieldsep '\t'
Field separator is "    ".
pierre@localhost testdb=> SELECT second, first FROM my_table;
un      1
deux    2
trois   3
quatre  4
```

Vous pouvez aussi utiliser les commandes courtes :

```
pierre@localhost basetest=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
pierre@localhost basetest=> SELECT * FROM ma_table;
-[ RECORD 1 ]---
premier  | 1
second   | un
-[ RECORD 2 ]---
premier  | 2
second   | deux
-[ RECORD 3 ]---
premier  | 3
second   | trois
-[ RECORD 4 ]---
premier  | 4
second   | quatre
```

De plus, ces options de format de sortie peuvent être affectées juste pour une requête en utilisant `\g` :

```
pierre@localhost basetest=> SELECT * FROM my_table
```

```
pierre@localhost basetest-> \g (format=aligned tuples_only=off
 expanded=on)
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

Voici un exemple utilisant la commande `\df` pour trouver seulementles fonctions dont le nom correspond à `int*pl` et donc le second argument est de type `bigint` :

```
testdb=> \df int*pl * bigint
                            List of functions
   Schema   |  Name   | Result data type | Argument data types |
 Type
------------+---------+------------------+---------------------
+------
 pg_catalog | int28pl | bigint           | smallint, bigint    |
 func
 pg_catalog | int48pl | bigint           | integer, bigint     |
 func
 pg_catalog | int8pl  | bigint           | bigint, bigint      |
 func
(3 rows)
```

En cas de besoin, les résultats de la requête peuvent être affichés dans une représentation croisée avec la commande `\crosstabview` :

```
testdb=> SELECT premier, second, premier > 2 AS gt2 FROM my_table;
premier | second | gt2
---------+--------+-----
      1 | un     | f
      2 | deux   | f
      3 | trois  | t
      4 | quatre | t
(4 rows)

testdb=> \crosstabview premier second
premier | un | deux | trois | quatre
---------+----+------+-------+--------
      1 | f  |      |       |
      2 |    | f    |       |
      3 |    |      | t     |
      4 |    |      |       | t
(4 rows)
```

Ce deuxième exemple montre une table de multiplication avec les lignes triées en ordre numérique inverse et les colonnes dans un ordre numérique ascendant indépendant.

```
testdb=> SELECT t1.premier as "A", t2.premier+100 AS "B",
 t1.premier*(t2.premier+100) as "AxB",
testdb(> row_number() over(order by t2.premier) AS ord
testdb(> FROM ma_table t1 CROSS JOIN ma_table t2 ORDER BY 1 DESC
testdb(> \crosstabview "A" "B" "AxB" ord
 A | 101 | 102 | 103 | 104
---+-----+-----+-----+-----
 4 | 404 | 408 | 412 | 416
 3 | 303 | 306 | 309 | 312
 2 | 202 | 204 | 206 | 208
 1 | 101 | 102 | 103 | 104
(4 rows)
```

reindexdb

reindexdb — réindexe une base de données PostgreSQL

Synopsis

`reindexdb` [*option-connexion*...] [*option*...] [`-S` | `--schema` *schéma*] ... [`-t` | `--table` *table*] ... [`-i` | `--index` *index*] ... [*nombase*]

`reindexdb` [*option-connexion*...] [*option*...] `-a` | `--all`

`reindexdb` [*option-connexion*...] [*option*...] `-s` | `--system` [*nombase*]

Description

reindexdb permet de reconstruire les index d'une base de données PostgreSQL.

reindexdb est un enrobage de la commande `REINDEX`. Il n'y a pas de différence entre la réindexation des bases de données par cette méthode et par celles utilisant d'autres méthodes d'accès au serveur.

Options

reindexdb accepte les arguments suivants en ligne de commande :

`-a`
`--all`

Réindexe toutes les bases de données.

`--concurrently`

Utilise l'option `CONCURRENTLY`. Voir REINDEX, où tous les détails de cette option sont expliqués en détail

`[-d] base`
`[--dbname=]base`

Spécifie le nom de la base à réindexer quand l'option `-a`/`--all` n'est pas utilisée. Si le nom de la base n'est pas fourni, il est lu à partir de la variable d'environnement `PGDATABASE`. Si elle n'est pas configurée, le nom de l'utilisateur pour la connexion est utilisé. Ce nom de base peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront toutes les options en ligne de commande conflictuelles.

`-e`
`--echo`

Affiche les commandes que reindexdb génère et envoie au serveur.

`-i index`
`--index=index`

Ne recrée que l'index *`index`*. Plusieurs indexes peuvent être récréés en même temps en utilisant plusieurs fois l'option `-i`.

`-j njobs`
`--jobs=njobs`

Exécute les commandes de réindexation en parallèle en exécutant *`njobs`* commandes simultanément. Cette option réduit le temps de traitement mais augmente aussi la charge sur le serveur de bases de données.

reindexdb ouvrira *njobs* connexions à la base de données, donc assurez-vous que le paramétrage de max_connections est suffisamment haut pour permettre toutes les connexions.

Notez que cette option est incompatible avec les options `--index` et `--system`.

`-q`
`--quiet`

N'affiche pas la progression.

`-s`
`--system`

Réindexe seulement les catalogues système de la base de données.

`-S` *schema*
`--schema=`*schema*

Ne réindexe que le schéma *schéma*. Plusieurs schémas peuvent être réindexés en même temps en utilisant plusieurs fois l'option `-S`.

`-t` *table*
`--table=`*table*

Ne réindexe que la table *table*. Plusieurs tables peuvent être réindexées en même temps en utilisant plusieurs fois l'option `-t`.

`--tablespace=`*tablespace*

Spécifie le tablespace dans lequel les index sont reconstruits. (Ce nom est traité comme un identifiant entre guillemets doubles.)

`-v`
`--verbose`

Affiche des informations détaillées sur le traitement.

`-V`
`--version`

Affiche la version de reindexdb, puis quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de reindexdb, puis quitte.

reindexdb accepte aussi les arguments suivants en ligne de commande pour les paramètres de connexion :

`-h` *hôte*
`--host=`*hôte*

Précise le nom d'hôte de la machine hébergeant le serveur. Si cette valeur débute par une barre oblique ('/' ou slash), elle est utilisée comme répertoire de socket UNIX.

`-p` *port*
`--port=`*port*

Précise le port TCP ou le fichier de socket UNIX d'écoute.

`-U` *nom_utilisateur*
`--username=`*nom_utilisateur*

Nom de l'utilisateur à utiliser pour la connexion.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force reindexdb à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car reindexdb demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, reindexdb perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`--maintenance-db=`*`nom-base-maintenance`*

Quand l'option `-a`/`--all` est utilisée, il se connecte à cette base pour récupérer la liste des bases à traiter. Si cette option n'est pas ajoutée, la base `postgres` sera utilisée. Si cette base n'existe pas, la base `template1` sera utilisée. Le nom de la base peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront les options en ligne de commande conflictuelles. De plus, les paramètres de la chaîne de connexion autres que le nom de la base lui-même seront réutilisés lors de la connexion aux autres bases.

Environnement

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Paramètres par défaut pour la connexion

`PG_COLOR`

Indique s'il faut utiliser des couleurs dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir la Section 34.15).

Diagnostics

En cas de difficultés, il peut être utile de consulter REINDEX et psql, pour les sections présentant les problèmes éventuels et les messages d'erreur.

Le serveur de base de données doit fonctionner sur le serveur cible. Les paramètres de connexion éventuels et les variables d'environnement utilisés par la bibliothèque cliente libpq s'appliquent.

Notes

reindexdb peut avoir besoin de se connecter plusieurs fois au serveur PostgreSQL. Afin d'éviter de saisir le mot de passe à chaque fois, on peut utiliser un fichier `~/.pgpass`. Voir la Section 34.16 pour plus d'informations.

Exemples

Pour réindexer la base de données `test` :

```
$ reindexdb test
```

Pour réindexer la table `foo` et l'index `bar` dans une base de données nommée `abcd` :

```
$ reindexdb --table=foo --index=bar abcd
```

Voir aussi

REINDEX

vacuumdb

vacuumdb — récupère l'espace inutilisé et, optionnellement, analyse une base de données PostgreSQL

Synopsis

vacuumdb [*option_de_connexion*...] [*option*...] [-t | --table *table* [(*colonne* [,...])]] ... [*nom_base*]

vacuumdb [*options_de_connexion*...] [*option*...] -a | --all

Description

vacuumdb est un outil de nettoyage d'une base de données. vacuumdb peut également engendrer des statistiques internes utilisées par l'optimiseur de requêtes de PostgreSQL.

vacuumdb est une surcouche de la commande VACUUM. Il n'y a pas de différence réelle entre exécuter des VACUUM et des ANALYZE sur les bases de données via cet outil et via d'autres méthodes pour accéder au serveur.

Options

vacuumdb accepte les arguments suivants sur la ligne de commande :

-a
--all

Nettoie toutes les bases de données.

[-d] *nom_base*
[--dbname=]*nom_base*

Spécifie le nom de la base à nettoyer ou à analyser quand l'option -a/--all n'est pas utilisée. Si le nom de la base n'est pas fourni, il est lu à partir de la variable d'environnement PGDATABASE. Si elle n'est pas configurée, le nom de l'utilisateur pour la connexion est utilisé. Ce nom de base peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront toutes les options en ligne de commande conflictuelles.

--disable-page-skipping

Désactive l'utilisation de la Visibility Map permettant d'ignorer certains blocs.

> **Note**
>
> Cette option est seulement disponible pour les serveurs exécutant PostgreSQL 9.6 et ultérieurs.

-e
--echo

Affiche les commandes que vacuumdb engendre et envoie au serveur.

-f
--full

Exécute un nettoyage « complet ».

`-F`
`--freeze`

« Gèle » agressivement les lignes.

`--force-index-cleanup`

Supprime toujours les entrées d'index pointant vers des lignes mortes.

> **Note**
>
> Cette option est seulement disponible pour les serveurs en version 12 et ultérieures.

`-j` *`njobs`*
`--jobs=`*`njobs`*

Exécute les commandes VACUUM et/ou ANALYZE en parallèle en plaçant *`njobs`* commandes simultanément. Cette option réduit la durée du traitement tout en augmentant la charge sur le serveur de bases de données.

vacuumdb ouvrira *`njobs`* connexions sur la base de données, donc assurez-vous que votre configuration du paramètre max_connections est suffisamment élevée pour accepter toutes les connexions nécessaires.

Notez que l'utilisation de ce mode avec l'option `-f` (`FULL`) pourrait causer des échecs de type deadlock si certains catalogues systèmes sont traités en parallèle.

`--min-mxid-age` *`age_mxid`*

Exécute les commandes VACUUM et ANALYZE sur les tables dont l'âge de l'identifiant multixact est d'au moins *`age_mxid`*. Cette option est utile pour prioriser les tables à traiter pour prévenir un bouclage des identifiants multixact (voir la Section 25.1.5.1).

Aux fins de cette option, l'âge de l'identifiant multixact d'une relation est le plus grand âge de la relation principale et de la relation TOAST associée si cette dernière existe. Comme les commandes exécutées par vacuumdb traitera aussi la table TOAST de la relation si nécessaire, il n'est pas nécessaire de la considérer séparément.

> **Note**
>
> Cette option est seulement utile pour les serveurs exécutant PostgreSQL 9.6 et ultérieurs.

`--min-xid-age` *`age_xid`*

Exécute les commandes VACUUM et ANALYZE sur les tables dont l'âge de l'identifiant de transaction est d'au moins *`age_xid`*. Cette option est utile pour prioriser les tables à traiter pour prévenir un bouclage des identifiants de transaction (voir la Section 25.1.5.1).

Aux fins de cette option, l'âge de l'identifiant de transaction d'une relation est le plus grand âge de la relation principale et de la relation TOAST associée si cette dernière existe. Comme les commandes exécutées par vacuumdb traitera aussi la table TOAST de la relation si nécessaire, il n'est pas nécessaire de la considérer séparément.

> **Note**
>
> Cette option est seulement utile pour les serveurs exécutant PostgreSQL 9.6 et ultérieurs.

`--no-index-cleanup`

Ne supprime pas les entrées d'index pointaint vers des lignes mortes.

> **Note**
>
> Cette option est seulement disponible pour les serveurs en version 12 et supérieures.

`--no-process-toast`

Ignore la table TOAST associée à la table en cours de traitement, s'il y en a une.

> **Note**
>
> Cette option est seulement disponible pour les serveurs en version 14 et supérieures.

`--no-truncate`

Ne tronque pas les pages vides en fin de table.

> **Note**
>
> Cette option est seulement disponible pour les serveurs en version 12 et supérieures.

`-P` *`workers_parallelisation`*
`--parallel=`*`workers_parallelisation`*

Indique le nombre de workers de parallélisation pour un *vacuum parallélisé*. Ceci permet au vacuum d'utiliser plusieurs CPU pour traiter les index. Voir VACUUM.

> **Note**
>
> Cette option est seulement disponible pour les serveurs PostgreSQL version 13 et supérieures.

`-q`
`--quiet`

N'affiche pas de message de progression.

`--skip-locked`

Ignore les relations qui ne peuvent pas immédiatement être verrouillées pour être traitées.

> **Note**
>
> Cette option est seulement utile pour les serveurs exécutant PostgreSQL 12 et ultérieurs.

`-t` *`table`* `[ (`*`colonne`* `[,...]) ]`
`--table=`*`table`* `[ (`*`colonne`* `[,...]) ]`

Ne nettoie ou n'analyse que la table *`table`*. Des noms de colonnes peuvent être précisés en conjonction avec les options `--analyze` ou `--analyze-only`. Plusieurs tables peuvent être traitées par VACUUM en utilisant plusieurs fois l'option `-t`.

> **Astuce**
>
> Lorsque des colonnes sont indiquées, il peut être nécessaire d'échapper les parenthèses. (Voir les exemples plus bas.)

`-v`
`--verbose`

Affiche des informations détaillées durant le traitement.

`-V`
`--version`

Affiche la version de vacuumdb, puis quitte.

`-z`
`--analyze`

Calcule aussi les statistiques utilisées par le planificateur.

`-Z`
`--analyze-only`

Calcule seulement les statistiques utilisées par le planificateur (donc pas de VACUUM).

`--analyze-in-stages`

Calcule seulement les statistiques utilisées par le planificateur (donc pas de VACUUM), comme `--analyze-only`. Lance les trois phases d'analyse. La première phase utilise la cible de statistiques la plus basse possible (voir default_statistics_target) pour produire des statistiques utilisables le plus rapidement, et les phases suivantes construisent les statistiques complètes.

Cette option est seulement utile pour analyser une base qui n'a actuellement pas de statistiques ou a des statistiques complètement faussées, comme c'est le cas d'une base toute nouvelle, restaurée à partir d'une sauvegarde ou par `pg_upgrade`. Soyez conscient qu'utiliser ces option dans une base ayant des statistiques existantes peut faire que les choix de l'optimiseur de statistiques deviennent temporairement pires à cause des cibles basses sur les statistiques pour la première phase.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de vacuumdb, puis quitte.

vacuumdb accepte aussi les arguments suivants comme paramètres de connexion :

`-h` *`hôte`*
`--host=`*`hôte`*

Indique le nom d'hôte de la machine qui héberge le serveur de bases de données. Si la valeur commence par une barre oblique (/), elle est utilisée comme répertoire pour la socket de domaine Unix.

`-p` *`port`*
`--port=`*`port`*

Indique le port TCP ou le fichier local de socket de domaine Unix sur lequel le serveur attend les connexions.

`-U` *`utilisateur`*
`--username=`*`utilisateur`*

Nom d'utilisateur pour la connexion.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force vacuumdb à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car vacuumdb demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, vacuumdb perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

`--maintenance-db=`*`nom_base_maintenance`*

Quand l'option `-a`/`--all` est utilisée, il se connecte à cette base pour récupérer la liste des bases à traiter. Si cette option n'est pas ajoutée, la base `postgres` sera utilisée. Si cette base n'existe pas, la base `template1` sera utilisée. Le nom de la base peut être remplacé par une chaîne de connexion. Dans ce cas, les paramètres de la chaîne de connexion surchargeront les options en ligne de commande conflictuelles. De plus, les paramètres de la chaîne de connexion autres que le nom de la base lui-même seront réutilisés lors de la connexion aux autres bases.

Environnement

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Paramètres de connexion par défaut.

`PG_COLOR`

Indique s'il faut utiliser des couleurs dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Cet outil, comme la plupart des autres outils PostgreSQL, utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 34.15).

Diagnostics

En cas de difficultés, il peut être utile de consulter VACUUM et psql, sections présentant les problèmes éventuels et les messages d'erreur.

Le serveur de base de données doit fonctionner sur le serveur cible. Les paramètres de connexion éventuels et les variables d'environnement utilisés par la bibliothèque cliente libpq s'appliquent.

Notes

vacuumdb peut avoir besoin de se connecter plusieurs fois au serveur PostgreSQL. Afin d'éviter de saisir le mot de passe à chaque fois, on peut utiliser un fichier `~/.pgpass`. Voir Section 34.16 pour plus d'informations.

Exemples

Pour nettoyer la base de données `test` :

```
$ vacuumdb test
```

Pour nettoyer et analyser une base de données nommée `grossebase` :

```
$ vacuumdb --analyze grossebase
```

Pour nettoyer la seule table `foo` dans une base de données nommée `xyzzy` et analyser la seule colonne `bar` de la table :

```
$ vacuumdb --analyze --verbose --table='foo(bar)' xyzzy
```

Voir aussi

VACUUM

Applications relatives au serveur PostgreSQL

Cette partie contient des informations de référence concernant les applications et les outils relatifs au serveur PostgreSQL. Ces commandes n'ont d'utilité que si elles sont lancées sur la machine sur laquelle le serveur fonctionne. D'autres programmes utilitaires sont listés dans la Applications client de PostgreSQL.

Table des matières

initdb 2296
pg_archivecleanup 2301
pg_checksums 2303
pg_controldata 2305
pg_ctl 2306
pg_resetwal 2312
pg_rewind 2316
pg_test_fsync 2320
pg_test_timing 2321
pg_upgrade 2325
pg_waldump 2335
postgres 2338
postmaster 2346

initdb

initdb — Créer un nouveau « cluster / instance »

Synopsis

```
initdb [option...] [ --pgdata | -D ] répertoire
```

Description

`initdb` crée une nouvelle instance de bases de données, ou « cluster », PostgreSQL. Un cluster est un ensemble de bases de données gérées par une même instance du serveur.

Créer une instance consiste à créer les répertoires dans lesquels sont stockées les données, créer les tables partagées du catalogue (tables partagées par toute l'instance, et non pas spécifique à une base), et créer les bases de données `postgres`, `template1` et `template0`. La base de données `postgres` est une base de données par défaut à destination des utilisateurs, des outils et des applications tiers. `template1` et `template0` sont des bases de données servant de source pour la copie réalisée par des commandes `CREATE DATABASE`. `template0` ne doit jamais être modifié mais vous pouvez ajouter des objets à `template1`, qui, par défaut, sera copiée dans toutes les bases créées ultérieurement. Voir Section 23.3 pour plus de détails.

`initdb` tente de créer le répertoire de données indiqué. Il se peut que la commande n'est pas les droits nécessaires si le répertoire parent du répertoire de données indiqué est possédé par root. Dans ce cas, pour réussir l'initialisation, il faut créer un répertoire de données vide en tant que root, puis utiliser `chown` pour en donner la possession au compte utilisateur de la base de données. `su` peut alors être utilisé pour prendre l'identité de l'utilisateur de la base de données et exécuter `initdb`.

`initdb` doit être exécuté par l'utilisateur propriétaire du processus serveur parce que le serveur doit avoir accès aux fichiers et répertoires créés par `initdb`. Comme le serveur ne peut pas être exécuté en tant que root, il est impératif de ne pas lancer `initdb` en tant que root. (En fait, `initdb` refuse de se lancer dans ces conditions.)

Pour des raisons de sécurité, la nouvelle instance créée par `initdb` sera seulement accessible par défaut par le propriétaire de l'instance. L'option `--allow-group-access` permet à tout utilisateur du même groupe que le propriétaire de l'instance de lire les fichiers de l'instance. Ceci est utile pour réaliser des sauvegardes en tant qu'utilisateur non privilégié.

`initdb` initialise la locale et l'encodage de jeu de caractères par défaut du cluster. Ils peuvent aussi être configurés séparément pour chaque base lors de leur création. `initdb` détermine ces paramètres pour les bases modèles servant de défaut pour toutes les autres bases. Par défaut, `initdb` utilise le fournisseur local `libc`, utilise les paramètres de la locale de l'environnement système, et détermine l'encodage à partir des paramètres de la locale. C'est est généralement suffisant, sauf s'il y a des besoins spéciaux.

Pour choisir une locale différente de celle de l'instance, utilisez l'option `--locale`. Il existe aussi des options individuelles `--lc-*` (voir ci-dessous) pour configurer les valeurs pour les catégories individuelles des locales. Notez que des paramètres incohérents pour des catégories différentes de locale peuvent donner des résultats insensés, donc c'est à utiliser avec prudence.

Il existe une alternative, la bibliothèque ICU, qui peut être utilisée pour fournir des services pour la locale. (Encore une fois, ceci configure seulement la valeur par défaut pour les bases de données créées par la suite.) Pour sélectionner cette option, indiquez `--locale-provider=icu`. Pour choisir l'identifiant ICU de la locale spécifique à appliquer, utilisez l'option `--icu-locale`. Notez que pour des raisons d'implémentation et pour supporter le code existant, `initdb` sélectionnera et initialisera les paramètres de locale en utilisant la bibliothèque C même si le fournisseur de locale ICU est utilisé.

Lors de l'exécution d'`initdb`, cette commande affichera les paramètres de la locale qu'elle a choisi. Si vous avez des besoins complexes ou si vous spécifiez plusieurs options, il est conseillé de vérifier que le résultat corresponde à la locale attendue.

Pour plus de détails sur la configuration de la locale, voir Section 24.1.

Pour modifier l'encodage par défaut, utilisez l'option `--encoding`. Section 24.3 propose plus d'options.

Options

`-A` *méthode_auth*
`--auth=`*méthode_auth*

Cette option spécifie la méthode d'authentification par défaut pour les utilisateurs locaux utilisée dans `pg_hba.conf` (lignes `host` et `local`). Voir Section 21.1 pour un aperçu des valeurs valides.

`initdb` pré-remplira les entrées de `pg_hba.conf` en utilisant la méthode d'authentification spécifiée pour les connexions qui ne sont pas pour la réplication aussi bien que pour les connexions de réplication.

N'utilisez pas `trust` à moins que vous ne fassiez entièrement confiance à tous les utilisateurs locaux sur votre système. `trust` est utilisé par défaut pour faciliter l'installation.

`--auth-host=`*méthode_auth*

Cette option spécifie la méthode d'authentification pour les utilisateurs définis dans le fichier `pg_hba.conf` et qui peuvent se connecter localement via une connexion TCP/IP (lignes `host`).

`--auth-local=`*méthode_auth*

Cette option spécifie la méthode d'authentification pour les utilisateurs définis dans le fichier `pg_hba.conf` et qui peuvent se connecter localement via une socket de domaine Unix (lignes `local`).

`-D` *répertoire*
`--pgdata=`*répertoire*

Indique le répertoire de stockage de la grappe de bases de données. C'est la seule information requise par `initdb`. Il est possible d'éviter de préciser cette option en configurant la variable d'environnement `PGDATA`. Cela permet, de plus, au serveur de bases de données (`postgres`) de trouver le répertoire par cette même variable.

`-E` *codage*
`--encoding=`*codage*

Définit l'encodage des bases de données modèles (*template*). C'est également l'encodage par défaut des bases de données créées ultérieurement. Cette valeur peut toutefois être surchargée. La valeur par défaut est déduite de la locale si le fournisseur libc est utilisé, ou `UTF8` si le fournisseur ICU est utilisé. Les jeux de caractères supportés par le serveur PostgreSQL sont décrits dans Section 24.3.1.

`-g`
`--allow-group-access`

Autorise les utilisateurs du même groupe que le propriétaire de l'instance à lire tous les fichiers de l'instance créés par `initdb`. Cette option est ignorée sur Windows car ce système ne supporte pas les droits de groupe du style POSIX.

`--icu-locale=`*locale*

Indique l'identifiant de la locale ICU si le fournisseur de locale ICU est utilisé.

`-k`
`--data-checksums`

Utilise des sommes de contrôle sur les pages de données pour aider à la détection d'une corruption par le système en entrée/sortie qui serait autrement passée sous silence. Activer les sommes de contrôle peut causer une pénalité importante sur les performances. Si elle est activée, les sommes de contrôle sont calculées pour tous les objets de chaque base de données. Tous les échecs lors des vérifications des sommes de contrôle seront rapportés dans la vue `pg_stat_database`. Voir Section 30.2 pour les détails.

`--locale=`*`locale`*

Configure la locale par défaut pour le cluster. Si cette option n'est pas précisée, la locale est héritée de l'environnement d'exécution d'`initdb`. Le support des locales est décrit dans Section 24.1.

`--lc-collate=`*`locale`*
`--lc-ctype=`*`locale`*
`--lc-messages=`*`locale`*
`--lc-monetary=`*`locale`*
`--lc-numeric=`*`locale`*
`--lc-time=`*`locale`*

Même principe que `--locale`, mais seule la locale de la catégorie considérée est configurée.

`--no-locale`

Équivalent à `--locale=C`.

`--locale-provider={libc|icu}`

Cette option configure le fournisseur de locale pour les bases de données créées dans la nouvelle instance. Il peut être surchargé dans la commande `CREATE DATABASE` quand de nouvelles bases sont créées par la suite. La valeur par défaut est `libc`.

`-N`
`--no-sync`

Par défaut, `initdb` attendra que tous les fichiers soient correctement écrit sur disque. Cette option permet à `initdb` de quitter sans attendre, ce qui est plus rapide, mais ce qui signifie aussi qu'un crash du système d'exploitation immédiatement après peut aboutir à une corruption du répertoire des données. Cette option n'est réellement utile que pour les tests et ne devrait pas être utilisée lors de la mise en place d'un serveur en production.

`--no-instructions`

Par défaut, `initdb` écrira les instructions de lancement de l'instance à la fin de son exécution. Cette option permet de ne pas afficher ces instructions. Ceci a principalement pour cible les outils qui font appel à `initdb` et pour qui ces instructions ont des chances d'être erronées.

`--pwfile=`*`nomfichier`*

Incite `initdb` à lire le mot de passe du superutilisateur à partir d'un fichier. La première ligne du fichier est utilisée comme mot de passe.

`-S`
`--sync-only`

Écrit en toute sécurité tous les fichiers de la base sur disque, puis quitte. Ceci ne réalise aucune des opérations normales d'initdb. Habituellement, cette option est utile pour s'assurer une restauration fiable après la modification de fsync de `off` à `on`.

`-T` *`config`*
`--text-search-config=`*`config`*

Définit la configuration par défaut pour la recherche de texte. Voir default_text_search_config pour de plus amples informations.

`-U` *`nomutilisateur`*
`--username=`*`nomutilisateur`*

Précise le nom de l'utilisateur défini comme superutilisateur de la base de données. Par défaut, c'est le nom de l'utilisateur qui lance `initdb`. Le nom du superutilisateur importe peu, mais postgres peut être conservé, même si le nom de l'utilisateur système diffère.

`-W`
`--pwprompt`

Force `initdb` à demander un mot de passe pour le superutilisateur de la base de données. Cela n'a pas d'importance lorsqu'aucune authentification par mot de passe n'est envisagée. Dans le cas contraire, l'authentification par mot de passe n'est pas utilisable tant qu'un mot de passe pour le superutilisateur n'est pas défini.

`-X` *`répertoire`*
`--waldir=`*`répertoire`*

Définit le répertoire de stockage des journaux de transactions.

`--wal-segsize=`*`size`*

Configure la *taille d'un segment WAL* en mégaoctets. C'est la taille d'un fichier individuel des journaux de transactions. La taille par défaut est de 16 Mo. La valeur doit être une puissance de 2 entre 1 et 1024 (mégaoctets). Cette option est seulement configurable au moment de l'initialisation. Elle ne peut plus être changée après.

Il pourrait être utile d'ajuster cette taille pour contrôler la granularité de la copie ou de l'archivage des journaux de transactions. De plus, dans les bases à gros volumes d'écritures, le grand nombre de fichiers de journaux de transactions par répertoire peut devenir un problème de performance et de gestion. Augmenter la taille des fichiers WAL réduira le nombre de fichiers WAL.

D'autres options, moins utilisées, sont disponibles :

`-d`
`--debug`

Affiche les informations de débogage du processus amorce et quelques autres messages de moindre intérêt pour le grand public. Le processus amorce est le programme qu'`initdb` lance pour créer les tables catalogues. Cette option engendre une quantité considérable de messages ennuyeux.

`--discard-caches`

Exécute le processus bootstrap avec l'option `debug_discard_caches=1`. Ceci prend beaucoup de temps et doit seulement être utilisé pour des séances de débugage intense.

`-L` *`répertoire`*

Indique à `initdb` où trouver les fichiers d'entrée nécessaires à l'initialisation du cluster. En temps normal, cela n'est pas nécessaire. Un message est affiché lorsque leur emplacement doit être indiqué de manière explicite.

`-n`
`--no-clean`

Par défaut, lorsqu'`initdb` rencontre une erreur qui l'empêche de finaliser la création du cluster, le programme supprime tous les fichiers créés avant l'erreur. Cette option désactive le nettoyage. Elle est utile pour le débogage.

D'autres options :

`-V`
`--version`

Affiche la version de initdb puis quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de initdb, puis quitte

Environnement

`PGDATA`

Indique le répertoire de stockage de la grappe de bases de données ; peut être surchargé avec l'option `-D`.

`PG_COLOR`

Indique s'il faut utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

`TZ`

Précise le fuseau horaire par défaut de l'instance. Cette valeur doit être un nom complet de fuseau horaire (voir Section 8.5.3).

Notes

`initdb` peut aussi être appelé avec `pg_ctl initdb`.

Voir aussi

pg_ctl, postgres, Section 21.1

pg_archivecleanup

pg_archivecleanup — nettoie les archives des journaux de transactions PostgreSQL

Synopsis

`pg_archivecleanup` [*option*...] *`emplacementarchive`* *`fichierwalaconserver`*

Description

pg_archivecleanup est conçu pour être utilisé avec le paramètre `archive_cleanup_command` pour nettoyer les archives de journaux de transactions quand un serveur standby est utilisé (voir Section 27.2). pg_archivecleanup peut aussi être utilisé en tant que programme autonome pour nettoyer les archives des journaux de transactions.

Pour configurer un serveur standby à utiliser pg_archivecleanup, placez ceci dans le fichier de configuration `postgresql.conf` :

```
archive_cleanup_command = 'pg_archivecleanup emplacementarchive %r'
```

où *`emplacementarchive`* est le répertoire où se trouvent les fichiers à nettoyer.

Lorsqu'il est utilisé dans archive_cleanup_command, tous les fichiers WAL précédant logiquement la valeur de l'argument `%r` seront supprimés de *`emplacementarchive`*. Ceci minimise le nombre de fichiers à conserver tout en préservant la possibilité de redémarrer après un crash. L'utilisation de ce paramètre est approprié si *`emplacementarchive`* est une aire temporaire pour ce serveur standby particulier. Cela n'est *pas* le cas quand *`emplacementarchive`* est conçu comme une aire d'archivage sur le long terme ou si plusieurs serveurs standby récupèrent les journaux à partir de ce même emplacement.

Lorsqu'il est utilisé en tant que programme autonome, tous les fichiers WAL qui précèdent logiquement *`fichierwalaconserver`* seront supprimés de *`emplacementarchive`*. Dans ce mode, si vous donnez un nom de fichier `.partial` ou `.backup`, alors seul le préfixe du fichier sera utilisé comme *`fichierwalaconserver`*. Ce traitement d'un nom de fichier `.backup` vous permet de supprimer tous les fichiers WAL archivés avant une sauvegarde de base spécifique, sans erreur. L'exemple suivant supprime tous les fichiers plus anciens que le nom `000000010000003700000010` :

```
pg_archivecleanup -d archive
 000000010000003700000010.00000020.backup

pg_archivecleanup:  keep WAL file
 "archive/000000010000003700000010" and later
pg_archivecleanup:  removing file
 "archive/00000001000000370000000F"
pg_archivecleanup:  removing file
 "archive/00000001000000370000000E"
```

pg_archivecleanup suppose que *`emplacementarchive`* est un répertoire accessible en lecture et écriture par l'utilisateur qui exécute le serveur.

Options

pg_archivecleanup accepte les arguments suivant en ligne de commande :

`-d`

Affiche plein de messages de debug sur `stderr`.

`-n`

Affiche le nom des fichiers qui auraient été supprimés sur le sortie standard (`stdout`) (permet un test).

`-V`
`--version`

Affiche la version de pg_archivecleanup, puis quitte.

`-x` *`extension`*

Fournit une extension qui sera supprimé de tous les noms de fichiers avant de décider s'ils doivent être supprimés. Ceci est utile pour nettoyer des archives qui ont été compressés lors du stockage et, de ce fait, ont une extension ajoutée par le programme de compression. Par exemple `-x .gz`.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_archivecleanup, puis quitte.

Environnement

La variable d'environnement `PG_COLOR` indique s'il faut utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto` et `never`.

Notes

pg_archivecleanup est conçu pour fonctionner avec PostgreSQL 8.0 et les versions ultérieures lorsqu'il est utilisé comme outil autonome, ou avec PostgreSQL et ultérieures quand il est utilisé comme commande de nettoyage des archives.

pg_archivecleanup est écrit en C et dispose d'un code facile à modifier, avec des sections désignées de telle façon qu'elles puissent être modifiées pour vos propres besoins.

Exemples

Sur des systèmes Linux ou Unix, vous pourriez utiliser :

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/
archive %r 2>>cleanup.log'
```

où le répertoire d'archivage est situé physiquement sur le serveur standby, pour que le paramètre `archive_command` y accède via NFS mais que les fichiers soient locaux au standby. Cette commande va :

- produire une sortie de débuggage `cleanup.log`
- supprimer les fichiers inutiles du répertoire d'archivage

pg_checksums

pg_checksums — Active, désactive ou vérifie les sommes de contrôle des fichiers de données d'une instance PostgreSQL

Synopsis

`pg_checksums` [*option*...] [[`-D` | `--pgdata`]*RÉPERTOIRE_DONNÉES*]

Description

pg_checksums vérifie, active ou désactive les sommes de contrôle des fichiers de données d'une instance PostgreSQL. Le serveur doit être arrêté proprement avant d'exécuter pg_checksums. Lors de la vérification des sommes de contrôle, le code de retour vaut zéro s'il n'y a pas d'erreur et différent de zéro s'il y a au moins une erreur détectée. Lors de l'activation ou de la désactivation des sommes de contrôle, le code de retour est différent de zéro si l'opération a échoué.

Lors de la vérification des sommes de contrôle, chaque fichier de l'instance est parcouru. Lors de l'activation des sommes de contrôle, chaque bloc des fichiers de l'instance dont le checksum a changé est réécrit. La désactivation des sommes de contrôle ne fait que mettre à jour le fichier `pg_control`.

Options

Les options suivantes sont disponibles en ligne de commande :

`-D` *`REPERTOIRE`*
`--pgdata=`*`REPERTOIRE`*

Spécifie le répertoire où l'instance de base de données est hébergée.

`-c`
`--check`

Vérifie les sommes de contrôle. Il s'agit du mode par défaut si rien d'autre n'est spécifié.

`-d`
`--disable`

Désactive les sommes de contrôle.

`-e`
`--enable`

Active les sommes de contrôle.

`-f` *`filenode`*
`--filenode=`*`filenode`*

Ne valide les sommes de contrôle que pour la relation ayant le nom de fichier *`filenode`*.

`-N`
`--no-sync`

Par défaut, `pg_checksums` va attendre que tous les fichiers soient écrits proprement sur disque. Cette option force `pg_checksums` à rendre la main sans attendre, ce qui est plus rapide, mais signifie qu'un arrêt impropre du système d'exploitation laissera le répertoire de données ciblé dans un état corrompu. Généralement, cette option est utile pour des tests mais ne devrait pas

être utilisée sur une installation de production. Cette option n'a pas d'effet lorsque `--check` est utilisée.

`-P`
`--progress`

Active le suivi de la progression. Activer cette option produira un rapport de progression lors des phases de vérification ou d'activation des sommes de contrôle.

`-v`
`--verbose`

Active l'affichage verbeux. Affichera tous les fichiers vérifiés.

`-V`
`--version`

Affiche la version de pg_checksums puis quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_checksums puis quitte.

Environnement

`PGDATA`

Indique le répertoire de stockage de l'instance de bases de données ; peut être surchargé avec l'option `-D`.

`PG_COLOR`

Indique s'il faut utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Notes

Activer les sommes de contrôle sur une grosse instance peut potentiellement prendre un long moment. Pendant cette opération, l'instance ou tout autre programme écrivant dans le répertoire des données ne peuvent pas être actifs. Dans le cas contraire, des pertes de données pourraient se produire.

Lors de l'utilisation de la réplication avec des outils qui copient directement les blocs des fichiers des relations (par exemple pg_rewind), l'activation ou la désactivation des sommes de contrôle peut mener à des corruptions de pages sous la forme de sommes de contrôle incorrectes si l'opération n'est pas menée de manière cohérente sur l'ensemble des nœuds. Lors de l'activation ou de la désactivation des sommes de contrôle dans un environnement répliqué, il est donc recommandé d'arrêter toutes les instances avant de les transformer une à une. Détruire toutes les instances secondaires, effectuant l'opération sur l'instance principale pour finalement recréer les secondaires de zéro est également considéré comme sûr.

Si pg_checksums est annulé ou tué lors de l'activation ou de la désactivation des sommes de contrôle, la configuration des sommes de contrôle des fichiers de l'instance reste inchangée, et pg_checksums peut être relancé pour effectuer à nouveau la même opération.

pg_controldata

pg_controldata — afficher les informations de contrôle d'un groupe de bases de données PostgreSQL

Synopsis

`pg_controldata` [*option*] [[`-D` | `--pgdata`]*répertoire_données*]

Description

`pg_controldata` affiche les informations initialisées lors d'`initdb`, telles que la version du catalogue. Il affiche aussi des informations sur le traitement des journaux de transactions et des checkpoints. Cette information, qui porte sur l'instance complète, n'est pas spécifique à une base de données.

Cet outil ne peut être lancé que par l'utilisateur qui a initialisé le groupe. En effet, il est nécessaire d'avoir le droit de lire le répertoire des données. Le répertoire des données peut être indiqué sur la ligne de commande ou à l'aide de la variable d'environnement `PGDATA`. Cet outil accepte les options `-V` et `--version`, qui affiche la version de pg_controldata puis arrête l'application. Il accepte aussi les options `-?` et `--help`, qui affichent les arguments acceptés.

Environnement

`PGDATA`

Emplacement du répertoire de données par défaut

`PG_COLOR`

Indique s'il faut utiliser les couleurs dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

pg_ctl

pg_ctl — initialiser, démarrer, arrêter ou contrôler le serveur PostgreSQL

Synopsis

`pg_ctl init[db]` [`-D` *répertoire_données*] [`-s`] [`-o` *options-initdb*]

`pg_ctl start` [`-D` *répertoire_données*] [`-l` *nomfichier*] [`-W`] [`-t` *secondes*] [`-s`] [`-o` *options*] [`-p` *chemin*] [`-c`]

`pg_ctl stop` [`-D` *répertoire_données*] [`-m` `s[mart]` | `f[ast]` | `i[mmediate]`] [`-W`] [`-t` *secondes*] [`-s`]

`pg_ctl logrotate` [`-D` *répertoire_données*] [`-s`]

`pg_ctl restart` [`-D` *répertoire_données*] [`-m` `s[mart]` | `f[ast]` | `i[mmediate]`] [`-W`] [`-t` *secondes*] [`-s`] [`-o` *options*] [`-c`]

`pg_ctl reload` [`-D` *répertoire_données*] [`-s`]

`pg_ctl status` [`-D` *répertoire_données*]

`pg_ctl promote` [`-s`] [`-D` *répertoire_données*] [`-W`] [`-t` *secondes*] [`-s`]

`pg_ctl kill` *nom_signal* *id_processus*

Sur Microsoft Windows, également :

`pg_ctl register` [`-D` *répertoire_données*] [`-N` *nom_service*] [`-U` *nom_utilisateur*] [`-P` *mot_de_passe*] [`-S` `a[uto]` | `d[emand]`] [`-e` *source*] [`-W`] [`-t` *secondes*] [`-s`] [`-o` *options*]

`pg_ctl unregister` [`-N` *nom_service*]

Description

pg_ctl est un outil qui permet d'initialiser une instance, de démarrer, d'arrêter, ou de redémarrer une instance PostgreSQL (postgres). Il permet également d'afficher le statut d'une instance en cours d'exécution.

Bien que le serveur puisse être démarré manuellement, pg_ctl encapsule les tâches comme la redirection des traces ou le détachement du terminal et du groupe de processus. Il fournit également des options intéressantes pour contrôler l'arrêt.

Le mode `init` ou `initdb` crée une nouvelle instance PostgreSQL. Une instance est un ensemble de bases contrôlées par une même instance du serveur. Ce mode invoque la commande `initdb`. Voir initdb pour les détails.

Le mode `start` lance une nouvelle instance. Le serveur est démarré en tâche de fond et l'entrée standard est attachée à `/dev/null` (sur `nul` sur Windows). Sur les systèmes Unix, par défaut, la sortie standard et la sortie des erreurs du serveur sont envoyées sur la sortie standard de pg_ctl (pas la sortie des erreurs). La sortie standard de pg_ctl devrait ensuite être redirigée dans un fichier standard ou dans un fichier pipe vers un autre processus comme un outil de rotation de fichiers de trace comme rotatelogs. Dans le cas contraire, `postgres` écrira sa sortie sur le terminal de contrôle. Sur Windows, par défaut, la sortie standard et la sortie des erreurs du serveur sont envoyées au terminal. Les comportements par défaut peuvent être changés en utilisant l'option `-l` pour rediriger la sortie du serveur dans un fichier de trace. L'utilisation de l'option `-l` ou d'une redirection de la sortie est recommandée.

Le mode `stop` arrête le serveur en cours d'exécution dans le répertoire indiqué. Trois méthodes différentes d'arrêt peuvent être choisies avec l'option `-m` : le mode « smart » désactive les nouvelles connexions, puis attend la déconnexion de tous les clients. Si le serveur est en mode hot standby, la récupération et la réplication en continu sont arrêtées dès que tous les clients se sont déconnectés. Le mode « fast » (la valeur par défaut) n'attend pas la déconnexion des clients. Toutes les transactions actives sont annulées et les clients sont déconnectés. Le serveur est ensuite arrêté. Le mode « immediate » tue tous les processus serveur immédiatement, sans leur laisser la possibilité de s'arrêter proprement. Ce choix conduit à une phase de récupération au redémarrage.

Le mode `restart` exécute un arrêt suivi d'un démarrage. Ceci permet de modifier les options en ligne de commande de `postgres`, ou de changer des options du fichier de configuration qui ne peuvent pas être modifiées sans un redémarrage du serveur. Si des chemins relatifs sont utilisés sur la ligne de commande durant le redémarrage du serveur, `restart` pourrait échouer à moins que pg_ctl ne soit exécuté depuis le même répertoire courant que celui durant le démarrage du serveur.

Le mode `reload` envoie simplement au processus `postgres` un signal SIGHUP. Le processus relit alors ses fichiers de configuration (`postgresql.conf`, `pg_hba.conf`, etc.). Cela permet de modifier les options des fichiers de configuration qui ne requièrent pas un redémarrage complet pour être prises en compte.

Le mode `status` vérifie si un serveur est toujours en cours d'exécution sur le répertoire de données indiqué. Si c'est le cas, le PID du serveur et les options en ligne de commande utilisées lors de son démarrage sont affichés. Si le serveur n'est pas en cours d'exécution, pg_ctl retourne une valeur de sortie de 3. Si un répertoire de données accessible n'est pas indiqué, pg_ctl retourne une valeur de sortie de 4.

Le mode `promote` commande au serveur secondaire en cours d'exécution dans le répertoire de données spécifié d'arrêter le mode de secours et de commencer des opérations en lecture / écriture.

Le mode `logrotate` effectue une rotation du fichier de traces du serveur. Pour les détails de l'utilisation de ce mode avec les outils externes de rotation de traces, voir Section 25.3.

Le mode `kill` envoie un signal à un processus spécifique. Ceci est particulièrement utile pour Microsoft Windows, qui ne possède pas de commande kill. `--help` permet d'afficher la liste des noms de signaux supportés.

Le mode `register` enregistre le serveur PostgreSQL comme service système sur Microsoft Windows. L'option `-S` permet la sélection du type de démarrage du service, soit « auto » (lance le service automatiquement lors du démarrage du serveur) soit « demand » (lance le service à la demande).

Le mode `unregister` supprime l'enregistrement du service système sur Microsoft Windows. Ceci annule les effets de la commande `register`.

Options

`-c`
`--core-files`

Tente d'autoriser la création de fichiers core suite à un arrêt brutal du serveur, sur les plateformes où cette fonctionnalité est disponible, en augmentant la limite logicielle qui en dépend. C'est utile pour le déboguage et pour diagnostiquer des problèmes en permettant la récupération d'une trace de la pile d'un processus serveur en échec.

`-D` *répertoire_données*
`--pgdata=`*datadir*

Indique l'emplacement des fichiers de configuration de la base de données sur le système de fichiers. Si cette option est omise, la variable d'environnement `PGDATA` est utilisée.

`-l` *`nom_fichier`*
`--log=`*`nom_fichier`*

Ajoute la sortie des traces du serveur dans *`nom_fichier`*. Si le fichier n'existe pas, il est créé. Par défaut, seul le propriétaire de l'instance peut accéder au fichier de trace. Si l'accès au groupe est activé dans l'instance, les utilisateurs du même groupe que celui du propriétaire de l'instance peuvent aussi le lire.

`-m` *`mode`*
`--mode=`*`mode`*

Précise le mode d'arrêt. *`mode`* peut valoir `smart`, `fast` ou `immediate`, ou la première lettre d'un de ces trois mots. Si cette option est omise, `fast` est utilisé.

`-o` *`options_initdb`*
`--options=`*`options_initdb`*

Indique les options à passer directement à la commande `postgres`. `-o` peut être utilisé plusieurs fois, et toutes les options spécifiées seront transférées.

Les *`options_initdb`* doivent habituellement être entourées de guillemets simples ou doubles pour s'assurer qu'elles sont bien passées comme un groupe.

`-p` *`chemin`*

Indique l'emplacement de l'exécutable `postgres`. Par défaut, l'exécutable `postgres` est pris à partir du même répertoire que `pg_ctl` ou, si cela échoue, à partir du répertoire d'installation codé en dur. Il n'est pas nécessaire d'utiliser cette option sauf cas inhabituel, comme lorsque des erreurs concernant l'impossibilité de trouver l'exécutable `postgres` apparaissent.

Dans le mode `init`, cette option indique de manière analogue la localisation de l'exécutable `initdb`.

`-s`
`--silent`

Affichage des seules erreurs, pas de messages d'information.

`-t` *`secondes`*
`--timeout=`*`secondes`*

Spécifie le nombre maximal de secondes à attendre lors de l'attente de la fin d'une opération (voir l'option `-w`). La valeur par défaut est la valeur de la variable d'environnement `PGCTLTIMEOUT`, ou 60 secondes si cette variable n'est pas positionnée.

`-V`
`--version`

Affiche la version de pg_ctl, puis quitte.

`-w`
`--wait`

Attend la fin de l'opération. Cela est supporté pour les modes `start`, `stop`, `restart`, `promote`, et `register`, et il s'agit de la valeur par défaut pour ces modes.

Lors d'une attente, `pg_ctl` tente de façon répétée de vérifier le fichier PID du serveur, s'endormant pour un court instant entre chaque test. Le démarrage est considéré réalisé quand le fichier PID indique que le serveur est prêt à accepter des connexions. L'arrêt est considéré réalisé quand le serveur a supprimé le fichier PID. Cette option permet d'entrer une passphrase SSL au démarrage. `pg_ctl` renvoie un code d'erreur basé sur le succès du démarrage ou de l'arrêt.

Si l'opération ne se termine pas dans le délai configuré (voir l'option option `-t`), alors `pg_ctl` quitte avec un code de sortie différent de zéro. Mais notez que l'opération pourrait continuer en tâche de fond et finalement réussir.

`-W`
`--no-wait`

N'attend pas la fin de l'opération. Il s'agit de l'opposé de l'option `-w`.

Si l'attente est désactivée, l'action demandée est déclenchée, mais il n'y a aucun retour sur son succès. Dans ce cas, le fichier de trace du serveur ou un système de supervision extérieur devra être utilisé pour vérifier le progrès et le succès de l'opération.

Dans les précédentes versions de PostgreSQL, c'était le comportement par défaut, sauf pour le mode `stop`.

`-?`
`--help`

Affiche de l'aide sur les arguments en ligne de commande de pg_ctl, puis quitte.

Si une option spécifiée est valide mais n'est pas applicable pour le mode d'opération sélectionné, pg_ctl l'ignore.

Options Windows

`-e` *`source`*

Nom de la source d'événement à utiliser par pg_ctl pour tracer dans le journal des événements lors de l'utilisation d'un service Windows. La valeur par défaut est `PostgreSQL`. Notez que ceci ne contrôle que les messages envoyés par pg_ctl ; une fois démarré, le serveur utilise la source d'événement spécifiée par son paramètre event_source. Si le serveur échoue très tôt durant le démarrage, avant que ce paramètre n'ait été positionné, il pourrait également envoyer une trace avec le nom de la source d'événement par défaut `PostgreSQL`.

`-N` *`nom_service`*

Nom du service système à enregistrer. Ce nom est utilisé à la fois comme nom de service et comme nom affiché. La valeur par défaut est `PostgreSQL`.

`-P` *`mot_de_passe`*

Mot de passe de l'utilisateur qui exécute le service.

`-S` *`start-type`*

Type de démarrage du service système à enregistrer. *`start-type`* peut valoir `auto` ou `demand` ou la première lettre de ces deux possibilités. Si cette option est omise, la valeur par défaut est `auto`.

`-U` *`nom_utilisateur`*

Nom de l'utilisateur qui exécute le service. Pour les utilisateurs identifiés sur un domaine, on utilise le format `DOMAINE\nom_utilisateur`.

Environnement

`PGCTLTIMEOUT`

Limite par défaut du nombre de secondes à attendre pour la fin de l'opération de démarrage ou d'arrêt. Si elle n'est pas configurée, l'attente est de 60 secondes.

PGDATA

Emplacement par défaut du répertoire des données.

La plupart des mode `pg_ctl` requièrent de connaitre l'emplacement du répertoire de donnée ; par conséquence, l'option `-D` est nécessaire à moins que PGDATA ne soit positionné.

Pour des variables serveurs supplémentaires qui affectent le serveur, voir postgres.

Fichiers

`postmaster.pid`

pg_ctl examine ce fichier dans le répertoire de données pour déterminer si le serveur est actuellement en cours d'exécution.

`postmaster.opts`

Si ce fichier existe dans le répertoire des données, pg_ctl (en mode `restart`) passe le contenu du fichier comme options de postgres, sauf en cas de surcharge par l'option `-o`. Le contenu de ce fichier est aussi affiché en mode `status`.

Exemples

Lancer le serveur

Démarrer un serveur, avec blocage tant que le serveur n'est pas complètement démarré :

```
$ pg_ctl start
```

Pour exécuter le serveur en utilisant le port 5433, et en s'exécutant sans `fsync` :

```
$ pg_ctl -o "-F -p 5433" start
```

Arrêt du serveur

Pour arrêter le serveur, utilisez :

```
$ pg_ctl stop
```

L'option `-m` autorise le contrôle sur la façon dont le serveur est arrêté :

```
$ pg_ctl stop -m smart
```

Redémarrage du serveur

Redémarrer le serveur est pratiquement équivalent à l'arrêter puis à le démarrer à nouveau si ce n'est que, par défaut, `pg_ctl` sauvegarde et réutilise les options en ligne de commande qui étaient passées à l'instance précédente. Pour redémarrer le serveur de la façon la plus simple, on utilise :

```
$ pg_ctl restart
```

Mais si `-o` est spécifié, cela remplace toute option précédente. Redémarrer en utilisant le port 5433 et en désactivant `fsync` après redémarrage :

```
$ pg_ctl -o "-F -p 5433" restart
```

Affichage de l'état du serveur

Exemple de statut affiché à partir de pg_ctl :

```
$ pg_ctl status

pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p"
 "5433" "-B" "128"
```

La deuxième ligne est la ligne de commande qui sera appelée en mode redémarrage.

Voir aussi

initdb, postgres

pg_resetwal

pg_resetwal — réinitialiser les WAL et les autres informations de contrôle d'une instance PostgreSQL

Synopsis

```
pg_resetwal [ -f | --force ] [ -n | --dry-run ] [option...] [ -D | --pgdata ]répertoire_données
```

Description

`pg_resetwal` efface les journaux d'écritures anticipées (*Write-Ahead Log* ou WAL) et réinitialise optionnellement quelques autres informations de contrôle stockées dans le fichier `pg_control`. Cette fonction est parfois nécessaire si ces fichiers ont été corrompus. Elle ne doit être utilisée qu'en dernier ressort quand le serveur ne démarre plus du fait d'une telle corruption.

À la suite de cette commande, le serveur doit pouvoir redémarrer. Toutefois, il ne faut pas perdre de vue que la base de données peut contenir des données inconsistantes du fait de transactions partiellement validées. Il est alors opportun de sauvegarder les données, lancer `initdb` et de les recharger. Après cela, les incohérences doivent être recherchées et le cas échéant corrigées.

Seul l'utilisateur qui a installé le serveur peut utiliser cet outil. Il requiert, en effet, un accès en lecture/écriture au répertoire des données. Pour des raisons de sécurité, `pg_resetwal` n'utilise pas la variable d'environnement `PGDATA`. Le répertoire des données doit donc être précisé sur la ligne de commande.

Si `pg_resetwal` se plaint de ne pas pouvoir déterminer de données valides pour `pg_control`, vous pouvez malgré tout le forcer à continuer en spécifiant l'option `-f` (force). Dans ce cas, des valeurs probables sont substituées aux données manquantes. La plupart des champs correspondent mais une aide manuelle pourrait être nécessaire pour le prochain OID, le prochain TID et sa date, le prochain identifiant multi-transaction et son décalage, l'adresse de début des journaux de transactions. Ces champs peuvent être configurés en utilisant les options indiquées ci-dessus. Si vous n'êtes pas capable de déterminer les bonnes valeurs pour tous ces champs, `-f` peut toujours être utilisé mais la base de données récupérée doit être traitée avec encore plus de suspicion que d'habitude : une sauvegarde immédiate et un rechargement sont impératifs. *Ne pas* exécuter d'opérations de modifications de données dans la base avant de sauvegarder ; ce type d'action risque de faire empirer la corruption.

Options

`-f`
`--force`

Force `pg_resetwal` à continuer même s'il ne peut pas déterminer de données valides pour `pg_control`, comme expliquées ci-dessus.

`-n`
`--dry-run`

L'option `-n`/`--dry-run` demande à `pg_resetwal` d'afficher les valeurs reconstruites à partir de `pg_control` et les valeurs à modifier, puis quitte sans faire aucune modification. C'est principalement un outil de débugage, mais il peut être utilisé aussi comme outil de vérification avant d'autoriser `pg_resetwal` à réaliser des modifications.

`-V`
`--version`

Affiche les informations de version, puis quitte.

`-?`
`--help`

Affiche l'aide, puis quitte.

Les options suivantes sont seulement nécessaires quand `pg_resetwal` est incapable de déterminer les valeurs appropriées lors de la lecture de `pg_control`. Des valeurs sûres peuvent être déterminées comme décrit ci-dessous. Pour les valeurs prenant des arguments numériques, les valeurs hexadécimales peuvent être précisées en utilisant le préfixe `0x`.

`-c` *`xid,xid`*
`--commit-timestamp-ids=`*`xid,xid`*

Configure manuellement le plus ancien et le plus récent identifiant de transaction pour lesquels le temps de validation peut être retrouvé.

Une valeur sûre pour la plus ancienne transaction dont le temps de validation peut être retrouvé (première partie) peut être déterminée en recherchant le numéro de fichier le plus petit numériquement dans le sous-répertoire `pg_commit_ts` du répertoire principal des données. De la même manière, une valeur sûre pour l'identifiant de transaction le plus récent dont le temps de validation peut être retrouvé (deuxième partie) peut être déterminé en recherchant le nom de fichier le plus grand numériquement dans le même répertoire. Les noms de fichiers sont en hexadécimal.

`-e` *`xid_epoch`*
`--epoch=`*`xid_epoch`*

Configure manuellement l'epoch du prochain identifiant de transaction.

L'epoch de l'identifiant de transaction n'est pas enregistré actuellement dans la base de données, en dehors du champ configuré par `pg_resetwal`, donc n'importe quelle valeur fonctionnera. Vous pourriez avoir besoin d'ajuster cette valeur pour assurer que les systèmes de réplication comme Slony-I et Skytools fonctionnent correctement -- dans ce cas, une valeur appropriée est récupérable à partir de l'état de la base de données répliquée.

`-l` *`fichier_wal`*
`--next-wal-file=`*`fichier_wal`*

Configure manuellement l'emplacement de démarrage du WAL en spécifiant le nom du prochain fichier de segment WAL.

Le nom du prochain fichier de segment WAL devrait être plus gros que le nom des segments WAL existant actuellement dans le sous-répertoire `pg_wal` sous le répertoire principal de données. Ces noms sont aussi en hexadécimal et sont composés de trois parties. La première partie est l'« identifiant de la ligne de temps » et devrait être généralement identique. Par exemple, si `000000010000000320000004A` est la plus large entrée dans largest entry in `pg_wal`, utilisez `-l 000000010000000320000004B` ou plus haut.

Notez que, en utilisant des tailles non standards pour les segments WAL, les nombres dans les noms des fichiers WAL sont différents des LSN reportés par les fonctions systèmes et les vues systèmes. Cette option prend un nom de fichier WAL, pas un LSN.

> **Note**
>
> `pg_resetwal` recherche lui-même les fichiers dans `pg_wal` et choisit une configuration par défaut pour `-l` au-dessus du dernier nom de fichier existant. De ce fait, un ajustement manuel de `-l` est seulement nécessaire si vous connaissez des fichiers de segments WAL qui ne sont pas actuellement présents dans `pg_wal`, comme les entrées d'une archive hors-ligne ou si le contenu de `pg_wal` a été entièrement perdu.

`-m` *mxid*`,`*mxid*
`--multixact-ids=`*mxid*`,`*mxid*

Configure manuellement le plus ancien et le prochain identifiants de multitransaction.

Une valeur sûre pour le prochain identifiant de multitransaction (première partie) peut être déterminée en recherchant le nom de fichier le plus élevé numériquement dans le sous-répertoire `pg_multixact/offsets` du répertoire principal des données, en ajoutant 1, puis en multipliant par 65536 (0x10000). De la même façon, une valeur sûr pour l'identifiant de multitransaction le plus ancien (deuxième partie de `-m`) peut être déterminée en recherchant le nom de fichier le moins élevé numériquement dans le même répertoire et en multipliant par 65536. Les noms de fichier sont en hexadécimal, donc la façon la plus simple de le faire est de spécifier la valeur en option en hexadécimal et d'ajouter quatre zéros.

`-u` *xid*

Configure manuellement le plus ancien identifiant de transaction non gelé.

Une valeur sûre peut être déterminée en recherchant le nom de fichier le plus petit dans le sous-répertoire `pg_xact` du répertoire des données et en le multipliant par 1048576 (0x100000). Notez que les noms de fichiers sont en hexadécimal. Il est habituellement plus simple d'indiquer la valeur de cette option en hexadécimal là-aussi. Par exemple, si `0007` est la plus petite entrée dans `pg_xact`, `-u 0x700000` fonctionnera (les cinq zéros en fin fournissent le bon multiplieur).

`-o` *oid*
`--next-oid=`*oid*

Configure manuellement le prochain OID.

Il n'existe pas de façon simple de déterminer le prochain OID, celui qui se trouve après le numéro le plus élevé dans la base de données. Heureusement, ce n'est pas critique de configurer correctement ce paramètre.

`-O` *mxoff*
`--multixact-offset=`*mxoff*

Configure manuellement le prochain décalage de multitransaction.

Une valeur sûre peut être déterminée en recherchant le nom de fichier le plus élevé numériquement dans le sous-répertoire `pg_multixact/members` du répertoire principal des données, en ajoutant 1, puis en multipliant par 52352 (0xCC80). Les noms de fichier sont en hexadécimal. Il n'existe pas de recette simple telle que celles fournies pour les autres options avec l'ajout de zéros.

`--wal-segsize=`*wal_segment_size*

Configure la nouvelle taille d'un segment WAL, en mégaoctets. La valeur doit être une puissance de 2 entre 1 et 1024 (mégaoctets). Voir la même option de initdb pour plus d'informations.

> **Note**
>
> Bien que `pg_resetwal` configurera l'adresse de début de WAL après le dernier fichier segment de WAL existant, certaines modifications de taille de segment peuvent causer la réutilisation de précédents noms de fichier WAL. Il est recommandé d'utiliser l'option `-l` avec cette option pour configurer manuellement l'adresse de début de WAL si une surcharge d'un nom de fichier WAL causerait des problèmes avec votre stratégie d'archivage.

`-x` *xid*
`--next-transaction-id=`*xid*

Configure manuellement la prochain identifiant de transaction.

Une valeur sûre peut être déterminée en recherchant le nom de fichier le plus élevé numériquement dans le sous-répertoire `pg_xact` du répertoire principal des données, en ajoutant 1, puis en multipliant par 1048576 (0x100000). Notez que les noms de fichier sont en hexadécimal. Il est généralement plus simple de spécifier la valeur de l'option en hexadécimal. Par exemple, si `0011` est l'entrée la plus élevée dans `pg_xact`, `-x 0x1200000` fonctionnera (cinq zéros à l'arrière fournissent le bon multiplicateur).

Environment

`PG_COLOR`

Précise s'il faut utiliser des couleurs dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Notes

Cette commande ne doit pas être utilisée quand le serveur est en cours d'exécution. `pg_resetwal` refusera de démarrer s'il trouve un fichier verrou du serveur dans le répertoire de données. Si le serveur s'est arrêté brutalement, un fichier verrou pourrait être toujours présent. Dans ce cas, vous pouvez supprimer le fichier verrou pour permettre l'exécution de `pg_resetwal`. Mais avant de faire cela, assurez-vous qu'aucun processus serveur n'est toujours en cours d'exécution.

`pg_resetwal` fonctionne seulement avec les serveurs de la même version majeure.

Voir aussi

pg_controldata

pg_rewind

pg_rewind — synchronise le répertoire des données de PostgreSQL avec un autre répertoire de données

Synopsis

```
pg_rewind [option...] { -D | --target-pgdata }  répertoire { --source-pgdata=répertoire | --source-server=chaine_de_connexion }
```

Description

pg_rewind est un outil qui permet de synchroniser une instance PostgreSQL avec une copie de cette même instance, a postériori de la séparation des timelines. Le scénario classique consiste à réactiver un ancien serveur primaire, après une bascule, en tant que serveur secondaire répliqué depuis le nouveau serveur primaire.

Après une synchronisation réussie, l'état du répertoire de données cible est analogue à une sauvegarde de base du répertoire source de données. Contrairement à l'exécution d'une sauvegarde de base ou l'utilisation d'un outil comme rsync, pg_rewind ne requiert pas de comparer ou copier les blocs inchangés des relations dans l'instance. Seuls les blocs modifiés de fichiers de relation existante sont copiés ; tous les autres fichiers, incluant tout nouveau fichier de relation, fichier de configuration, et segment WAL, sont copiés en totalité. Ainsi, l'opération de synchronisation est significativement plus rapide que les autres approches si la base de données est volumineuse et que seule une petite fraction de blocs diffère entre les instances.

pg_rewind inspecte l'historique de la timeline de l'instance source et de l'instance cible pour déterminer à quel moment a eu lieu la séparation, et s'attend à trouver tous les fichiers WAL jusqu'au moment de la bascule dans le répertoire `pg_wal`. Le point de divergence peut être trouvé soit sur la ligne de temps cible, soit sur la ligne de temps source, soit sur leur ancêtre commun. Dans un scénario de bascule classique, où l'instance cible a été arrêtée peu après le changement, cela ne pose aucun problème. En revanche, si l'instance cible a travaillé un certain temps après le changement, les anciens fichiers WAL peuvent ne plus être présents dans le répertoire `pg_wal`. Dans ce cas, ils peuvent être copiés à la main depuis les fichiers WAL archivés vers le répertoire `pg_wal` ou être réexécutés par pg_rewind avec l'option `-c` pour automatiquement les récupérer de l'archive WAL. L'utilisation de pg_rewind n'est pas limitée au failover : un serveur standby peut être promu, écrire quelques transactions, puis transformé de nouveau en standby avec cet outil.

Après avoir exécuté pg_rewind, il est nécessaire que la réapplication des WAL se termine pour que le répertoire de données soit dans un état cohérent. Quand le serveur cible est redémarré, il va entrer en récupération d'archive et rejouer tous les WAL générés sur le serveur source depuis le dernier checkpoint avant le point de divergence. Si certains WAL ne sont plus disponibles sur le serveur source quand pg_rewind a été exécutée, et par conséquent ne peuvent plus être copiés par la session pg_rewind, ils doivent être rendus disponible quand le serveur cible est démarré. Ceci peut être effectué en créant un fichier `recovery.signal` dans le répertoire de données cible et en configurant un restore_command approprié dans `postgresql.conf`.

pg_rewind nécessite que l'instance cible ait l'option wal_log_hints activée dans le fichier de configuration `postgresql.conf` ou que les sommes de contrôle sur les données aient été activées lorsque l'instance a été initialisée par la commande initdb. Aucune de ces options n'est active par défaut. Le paramètre full_page_writes doit lui aussi être activé. Il l'est par défaut.

Avertissement

Si pg_rewind échoue lors du traitement, le répertoire des données de la cible est très probablement dans un état inutilisable. Dans ce cas, réaliser une nouvelle sauvegarde est recommandé.

Comme pg_rewind copie entièrement les fichiers de configuration de la source, il peut être requis de corriger la configuration employée pour la reprise avant de redémarrer le serveur cible, particulièrement si la cible est réintroduite comme un standby de la source. Si vous redémarrez le serveur après que l'opération de synchronisation soit terminée mais sans reprendre la configuration, la cible peut diverger du primaire.

pg_rewind échouera immédiatement s'il trouve des fichiers qu'il ne peut pas modifier. Ceci peut arriver quand les serveurs source et cible utilisent les mêmes emplacements pour les clés et les certificats SSL qui sont habituellement en lecture seule. Si de tels fichiers sont présents sur le serveur cible, il est recommandé de les supprimer avant d'exécuter pg_rewind. Après l'avoir exécuté, certains de ces fichiers devront peut-être être copiés de la source, auquel cas il pourrait être nécessaire de supprimer les données copiées et de restaurer l'ensemble des liens utilisés avant l'exécution de l'outil.

Options

pg_rewind accepte les arguments suivants en ligne de commande :

`-D` *`répertoire`*
`--target-pgdata=`*`répertoire`*

Cette option définit le répertoire de données cible qui va être synchronisé avec le répertoire source. Cette option requiert que le serveur source ait été arrêté proprement.

`--source-pgdata=`*`répertoire`*

Cette option définit le répertoire de données source qui va être synchronisé avec le répertoire cible. Si cette option est utilisée, l'instance source doit être arrêtée proprement.

`--source-server=`*`chaine de connexion`*

Définit une chaine de connexion libpq permettant d'accéder à l'instance PostgreSQL source de façon à pouvoir la synchroniser avec la cible. Cette option requiert l'utilisation d'une connexion standard avec un rôle ayant les droits suffisants pour exécuter les fonctions utilisées par pg_rewind sur le serveur source (voir la section Notes pour les détails) ou un rôle superutilisateur.

`--no-ensure-shutdown`

pg_rewind requiert que le serveur cible soit proprement arrêté avant d'être synchronisé. Par défaut, si le serveur cible n'est pas proprement arrêté, pg_rewind le démarre en mode simple-utilisateur pour effectuer d'abord une récupération après accident, et l'arrête. En utilisant cette option, pg_rewind ignore cela et tombe en erreur immédiatement si le serveur n'est pas proprement stoppé. Les utilisateurs doivent s'attendre à gérer la situation eux-même dans ce cas.

`-R`
`--write-recovery-conf`

Crée `standby.signal` et ajoute les paramètres de connexion à `postgresql.auto.conf` dans le répertoire de sortie. `--source-server` est obligatoire avec cette option.

`-n`
`--dry-run`

Réalise toutes les opérations sans modifier le répertoire cible.

`-N`
`--no-sync`

Par défaut, `pg_rewind` attendra l'écriture certaine de tous les fichiers sur disque. Cette option permet de forcer le retour de `pg_rewind` sans attente, ce qui est plus rapide, mais signifie qu'un

crash ultérieur du système d'exploitation pourrait laisser le répertoire des données dans un état corrompu. En général, cette option est utile pour des tests, mais ne devrait pas être utilisée pour créer un serveur en production.

`-P`
`--progress`

Permet d'activer les traces. Activer cette option vous fournira les informations au fil de l'eau sur l'avancée de la copie des données depuis l'instance source.

`-c`
`--restore-target-wal`

Utilise `restore_command` définie dans la configuration de l'instance cible pour reprendre les fichiers WAL depuis l'archive WAL si ces fichiers ne sont plus disponibles dans le répertoire `pg_wal`.

`--config-file=`*`nom_fichier`*

Utilise le fichier de configuration du serveur indiqué pour l'instance cible. Ceci affecte pg_rewind quand il utilise en interne la commande postgres pour l'opération sur cette instance (lors de la récupération de `restore_command` avec l'option `-c/--restore-target-wal` et quand il force la fin d'une restauration après un crash).

`--debug`

Affiche les détails de la sortie de débogage, ce qui est surtout utile aux développeurs qui travaillent sur pg_rewind.

`-V`
`--version`

Affiche les informations concernant la version, puis quitte.

`-?`
`--help`

Affiche l'aide, puis quitte.

Environnement

Lorsque l'option `--source-server` est utilisée, pg_rewind utilise aussi les variables d'environnement supportées par la bibliothèque libpq (voir Section 34.15).

La variable d'environnement `PG_COLOR` indique s'il faut utiliser les couleurs dans les messages de diagnotiques. Les valeurs possibles sont `always`, `auto`, `never`.

Notes

Lors de l'exécution de pg_rewind sur une instance en ligne comme source, un rôle doté des droits suffisants pour exécuter les fonctions utilisées par pg_rewind sur l'instance source peut être utilisé à la place d'un superutilisateur. Voici comment créer ce rôle, nommé ici `rewind_user` :

```
CREATE USER rewind_user LOGIN;
GRANT EXECUTE ON function pg_catalog.pg_ls_dir(text, boolean,
 boolean) TO rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_stat_file(text, boolean) TO
 rewind_user;
```

```
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text) TO
 rewind_user;
GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text,
 bigint, bigint, boolean) TO rewind_user;
```

Lors de l'exécution de pg_rewind sur une instance en ligne comme source récemment promue, il est nécessaire d'exécuter un `CHECKPOINT` après la promotion pour que son fichier de contrôle reflète des informations de timeline à jour, qui est utilisé par pg_rewind pour vérifier si l'instance cible peut être rembobiné en utilisant l'instance source désignée.

Fonctionnement

L'idée de base est de copier toutes les modifications de fichiers au niveau système de fichiers de l'instance source vers l'instance cible :

1. Parcourir les journaux de transactions de l'instance cible, en commençant du dernier checkpoint avant le moment où l'historique de timeline de l'instance source a dévié de celle de l'instance cible. Pour chaque enregistrement dans les journaux de transactions, enregistrer chaque bloc de données modifié. Ceci a pour résultat une liste de tous les blocs de données modifiés dans l'instance cible, après la séparation avec l'instance source. Si certains fichiers WAL ne sont plus disponibles, essayez de ré-exécuter pg_rewind avec l'option `-c` pour chercher les fichiers manquants dans l'archive WAL.

2. Copier tous les blocs modifiés de l'instance source vers l'instance cible, soit en utilisant un accès direct au système de fichiers (`--source-pgdata`) soit en SQL (`--source-server`). Les fichiers de relations sont maintenant dans un état équivalent au moment du dernier checkpoint effectué avant le point où les timelines WAL de la source et de la cible divergent, plus l'état actuel sur la source où il existe des blocs modifiés sur la cible après divergence.

3. Copier tous les autres fichiers, incluant les nouveaux fichiers de relation, les segments WAL, `pg_xact`, et les fichiers de configuration de l'instance source sur l'instance cible. De manière similaire, les sauvegardes de base, le contenu des répertoires `pg_dynshmem/`, `pg_notify/`, `pg_replslot/`, `pg_serial/`, `pg_snapshots/`, `pg_stat_tmp/` et `pg_subtrans/` sont omis des données copiées depuis le serveur source. Les fichiers `backup_label`, `tablespace_map`, `pg_internal.init`, `postmaster.opts`, et `postmaster.pid` et `.DS_Store`, aussi bien que tous fichiers ou répertoires commençant par `pgsql_tmp`, sont omis.

4. Crée un ficher `backup_label` pour débuter la reprise des WAL au checkpoint créé lors de la bascule et configure le fichier `pg_control` avec une cohérence minimale du LSN définie comme le résultat de `pg_current_wal_insert_lsn()` lorsque la synchronisation s'effectue depuis une source activée ou le dernier checkpoint LSN quand la synchronisation s'effectue depuis une source stoppée.

5. Quand la cible est démarrée, PostgreSQL reprend tous les WAL nécessaires, donnant ainsi un répertoire de données à un état cohérent.

pg_test_fsync

pg_test_fsync — déterminer la configuration de `wal_sync_method` la plus rapide pour PostgreSQL

Synopsis

`pg_test_fsync` [*option*...]

Description

pg_test_fsync a pour but de donner une idée raisonnable de la configuration la plus rapide de wal_sync_method sur votre système spécifique, ainsi que de fournir des informations de diagnostics dans le cas où un problème d'entrées/sorties est identifié. Néanmoins, les différences montrées par pg_test_fsync pourraient ne pas faire de grosses différences sur une utilisation réelle de la base de données, tout spécialement quand de nombreux serveurs de bases de données ne sont pas limitées en performance par les journaux de transactions. pg_test_fsync rapporte la durée moyenne d'opération d'une synchronisation de fichiers en microsecondes pour chaque configuration possible de `wal_sync_method`, qui peut aussi être utilisé pour informer des efforts à optimiser la valeur de commit_delay.

Options

pg_test_fsync accepte les options suivantes en ligne de commande :

`-f`
`--filename`

Spécifie le nom du fichier où écrire les données de tests. Ce fichier doit être dans le même système de fichiers que le répertoire `pg_wal`. (`pg_wal` contient les fichiers WAL.) La valeur par défaut est de placer `pg_test_fsync.out` dans le répertoire courant.

`-s`
`--secs-per-test`

Indique le nombre de secondes de chaque test. Plus la durée est importante, et plus la précision du test est importante, mais plus cela prendra du temps. La valeur par défaut est de cinq secondes, ce qui permet au programme de terminer en moins de deux minutes.

`-V`
`--version`

Affiche la version de pg_test_fsync, puis quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_test_fsync, puis quitte.

Environnement

La variable d'environnement `PG_COLOR` indique s'il faut utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto` et `never`.

Voir aussi

postgres

pg_test_timing

pg_test_timing — mesure de l'impact du chronométrage

Synopsis

`pg_test_timing` [*option*...]

Description

pg_test_timing est un outil qui mesure l'impact du chronométrage sur votre système et confirme que l'horloge système ne prend jamais de retard. Les systèmes qui sont lents pour collecter des données chronométrées peuvent donner des résultats moins précis pour la commande `EXPLAIN ANALYSE`.

Options

pg_test_timing accepte les options de ligne de commande suivantes :

`-d` *durée*
`--duration=`*durée*

Indique la durée du test, en secondes. Des durées plus longues ont une précision bien meilleure, et ont plus de chances de détecter des problèmes avec les horloges systèmes qui prennent du retard. La durée par défaut du test est de 3 secondes.

`-V`
`--version`

Affiche la version de pg_test_timing puis termine.

`-?`
`--help`

Affiche l'aide concernant les arguments de la commande pg_test_timing, puis termine.

Utilisation

Interprétation des résultats

De bons résultats montreront que la plupart (>90%) des appels individuels de chronométrage prendront moins d'une microseconde. La moyenne de l'impact par boucle sera même plus basse, sous 100 nanosecondes. L'exemple ci-dessous tiré d'un système Intel i7-860 utilisant une source d'horloge TSC montre d'excellentes performances :

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 35.96 ns
Histogram of timing durations:
  < us   % of total      count
     1     96.40465   80435604
     2      3.59518    2999652
     4      0.00015        126
     8      0.00002         13
    16      0.00000          2
```

Notez que différentes unités sont utilisées pour le temps par boucle et les temps figurant dans l'histogramme. La boucle peut avoir une résolution de quelques nanosecondes (ns), alors que les appels individuels de chronométrage peuvent seulement descendre jusqu'à une microseconde (us).

Mesure de l'impact du chronométrage sur l'exécuteur

Lorsque l'exécuteur de requêtes exécute une instruction `EXPLAIN ANALYSE`, les opérations individuelles sont également chronométrées comme affiché par la sortie de la commande. L'impact supplémentaire de votre système peut être vérifié en comptant les lignes avec l'application psql :

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

Le système i7-860 mesuré effectue la requête de comptage en 9,8 ms alors que la commande `EXPLAIN ANALYSE` prend 16,6 ms, chaque version traitant 100 000 lignes. Cette différence de 6,8 ms signifie que l'incidence du chronométrage par ligne est de 68 ns, à peu près le double de l'estimation de pg_test_timing. Même cette relative petite incidence fait que l'instruction complète prend presque 70% de temps en plus. Sur des requêtes plus substantielles, l'impact du chronométrage serait moins problématique.

Modification de la source du chronométrage

Sur certains systèmes Linux récents, il est possible de modifier l'horloge source utilisée pour collecter les données chronométrées à n'importe quel moment. Un second exemple montre le ralentissement possible d'un passage à l'horloge plus lente acpi_pm, sur le même système utilisé pour les résultats rapides ci-dessus :

```
# cat /sys/devices/system/clocksource/clocksource0/
available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/
current_clocksource
# pg_test_timing
Per loop time including overhead: 722.92 ns
Histogram of timing durations:
  < us   % of total      count
     1     27.84870    1155682
     2     72.05956    2990371
     4      0.07810       3241
     8      0.01357        563
    16      0.00007          3
```

Dans cette configuration, la même commande `EXPLAIN ANALYSE` que ci-dessus prend 115,9 ms. Soit un impact de 1061 ns du chronométrage, à nouveau un petit multiple de ce qui est mesuré directement par cet utilitaire. Autant d'impact du chronométrage signifie que la requête actuelle elle-même prend une petite fraction du temps constaté, la plupart de celui-ci étant consommé par l'impact du chronométrage. Dans cette configuration, tous les totaux de la commande `EXPLAIN ANALYSE` entraînant beaucoup d'opérations chronométrées seront significativement augmentés par cet impact du chronomètre.

FreeBSD permet également de modifier à la volée la source du chronométrage, et il trace au démarrage l'information concernant l'horloge sélectionnée :

```
# dmesg | grep "Timecounter"
Timecounter "ACPI-fast" frequency 3579545 Hz quality 900
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounters tick every 10.000 msec
Timecounter "TSC" frequency 2531787134 Hz quality 800
# sysctl kern.timecounter.hardware=TSC
kern.timecounter.hardware: ACPI-fast -> TSC
```

D'autres systèmes peuvent n'autoriser la modification de la source du chronométrage qu'au démarrage. Sur les plus vieux systèmes Linux, le paramètre noyau « clock » est la seule manière d'effectuer ce type de modification. Et même sur certains systèmes plus récents, la seule option disponible pour une source d'horloge est « jiffies ». Jiffies est la plus vieille implémentation Linux d'horloge logicielle, qui peut avoir une bonne résolution lorsqu'elle s'appuie sur une horloge matérielle suffisamment rapide, comme dans cet exemple :

```
$ cat /sys/devices/system/clocksource/clocksource0/
available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
time.c: Detected 2400.153 MHz processor.
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
  < us   % of total      count
     1     90.23734   27694571
     2      9.75277    2993204
     4      0.00981       3010
     8      0.00007         22
    16      0.00000          1
    32      0.00000          1
```

Horloge matérielle et exactitude du chronométrage

Collectionner des données chronométrées précises est normalement effectué sur des ordinateurs en utilisant des horloges matérielles ayant différents niveaux de précision. Avec certains matériels, le système d'exploitation peut transmettre le temps de l'horloge système presque directement aux programmes. Une horloge système peut aussi provenir d'une puce qui fournit simplement des interruptions périodiques, des tic-tac à intervalles réguliers. Dans les deux cas, le noyau des systèmes d'exploitation fournit une source d'horloge qui masque ces détails. Mais la précision de cette source d'horloge et la vitesse à laquelle elle peut renvoyer des résultats est fonction du matériel sous-jacent.

Une gestion du temps inexacte peut entraîner une instabilité du système. Testez tous les changements de source d'horloge avec soin. Les réglages par défaut des systèmes d'exploitation sont parfois effectués pour favoriser la fiabilité sur la précision. Et si vous utilisez une machine virtuelle, examinez les sources d'horloge compatibles recommandées avec elle. Le matériel virtuel fait face à des difficultés additionnelles pour émuler des horloges, et il existe souvent des réglages par système d'exploitation suggérés par les vendeurs.

La source d'horloge Time Stamp Counter (TSC) est la plus précise disponible sur la génération actuelle de CPU. C'est la manière préférentielle pour suivre le temps système lorsqu'elle est supportée par le système d'exploitation et que l'horloge TSC est fiable. Il existe plusieurs sources possibles pour qu'une horloge TSC échoue à fournir une source de temps précise, la rendant non fiable. Les plus vieux systèmes peuvent avoir une horloge TSC qui varie en fonction de la température du CPU, la rendant inutilisable pour le chronométrage. Essayer d'utiliser une horloge TSC sur certains vieux CPU multi-cœurs peut renvoyer des temps qui sont incohérents entre les multiples cœurs. Ceci peut résulter en

des temps qui reculent, un problème que ce programme vérifie. Et même des systèmes plus récents peuvent échouer à fournir des chronométrages TSC précis avec des configurations très agressives en matière d'économie d'énergie.

Les systèmes d'exploitation plus récents peuvent vérifier ces problèmes connus avec l'horloge TSC et basculer vers une source plus lente, plus stable lorsqu'elles sont vues. Si votre système supporte le temps TSC mais ne l'utilise pas par défaut, c'est qu'il peut être désactivé pour une bonne raison. Et certains systèmes d'exploitation peuvent ne pas détecter correctement tous les problèmes possibles, ou autoriseront l'utilisation de l'horloge TSC y compris dans des situations où il est reconnu qu'elle n'est pas fiable.

La source High Precision Event Timer (HPET) est l'horloge préférée sur les systèmes où elle est disponible et que TSC n'est pas fiable. La puce horloge elle-même est programmable pour permettre une résolution allant jusqu'à 100 nanosecondes, mais vous pouvez ne pas constater autant de précision avec votre horloge système.

L'Advanced Configuration and Power Interface (ACPI) fournit un Power Timer (PIT), le real-time clock (RTC), l'horloge Advanced Programmable Interrupt Controller (APIC), et l'horloge Cyclone. Ces chronomètres visent une résolution de l'ordre de la milliseconde.

Voir aussi

EXPLAIN

pg_upgrade

pg_upgrade — met à jour une instance du serveur PostgreSQL

Synopsis

```
pg_upgrade -b ancien_repertoire_executables [-B nouveau_repertoire_executables] -d ancien_repertoire_configuration -D nouveau_repertoire_configuration [option...]
```

Description

pg_upgrade (antérieurement connu sous le nom pg_migrator) permet de mettre à jour les fichiers de données vers une version plus récente de PostgreSQL sans la sauvegarde et le rechargement de données typiquement requis pour les mises à jour d'une version majeure vers une autre, par exemple d'une version 9.5.8 à une version 9.6.4 ou à partir de 10.7 vers 11.2. Il n'est pas nécessaire pour les mises à jour de versions mineures, par exemple de 9.6.2 à 9.6.3 ou de 10.1 à 10.2.

Les sorties de version majeures de PostgreSQL ajoutent régulièrement de nouvelles fonctionnalités qui changent souvent la structure des tables système, mais le format interne des données stockées change rarement. pg_upgrade utilise ce fait pour effectuer des mises à jour rapides en créant de nouvelles tables systèmes et en réutilisant les anciens fichiers de données. Si jamais une future version majeure devait modifier le format d'enregistrement des données de telle sorte que l'ancien format des données soit illisible, pg_upgrade ne pourrait pas être utilisé pour ces mises à jour. (La communauté essaiera d'éviter de telles situations.)

pg_upgrade fait de son mieux pour être sûr que la nouvelle et l'ancienne instances soient compatibles au niveau binaire, par exemple en vérifiant que les options de compilation sont compatibles, y compris le format 32/64 bits des binaires. Il est également important que les modules externes soient aussi compatibles au plan binaire, bien que ceci ne puisse être vérifié par pg_upgrade.

pg_upgrade supporte les mises à jour à partir de la version 9.2.X et suivantes jusqu'à la version majeure courante de PostgreSQL, y compris les images et versions beta.

> **Avertissement**
>
> Mettre à jour une instance fait que la destination exécute du code arbitraire au choix des superutilisateurs de la source. Assurez-vous que les superutilisateurs de la source sont de confiance avant de mettre à jour.

Options

pg_upgrade accepte les arguments de ligne de commande suivants :

`-b` *repertoire_executables*
`--old-bindir=`*repertoire_executables*

l'ancien répertoire des exécutables PostgreSQL ; variable d'environnement `PGBINOLD`

`-B` *repertoire_executables*
`--new-bindir=`*repertoire_executables*

le nouveau répertoire des exécutables PostgreSQL ; la valeur par défaut est le répertoire d'installation de pg_upgrade ; variable d'environnement `PGBINNEW`

`-c`
`--check`

uniquement la vérification des instances, ne modifie aucune donnée

`-d` *`repertoire_configuration`*
`--old-datadir=`*`repertoire_configuration`*

répertoire de configuration de l'ancienne instance ; variable d'environnement`PGDATAOLD`

`-D` *`repertoire_configuration`*
`--new-datadir=`*`repertoire_configuration`*

répertoire de configuration de la nouvelle instance ; variable d'environnement `PGDATANEW`

`-j` *`njobs`*
`--jobs=`*`njobs`*

nombres de processus ou threads simultanés à utiliser

`-k`
`--link`

utiliser des liens physiques au lieu de copier les fichiers vers la nouvelle instance

`-N`
`--no-sync`

Par défaut, `pg_upgrade` attendra que tous les fichiers de l'instance mise à jour soient écrites sur disque. Cette option fait que `pg_upgrade` quitte sans attendre, ce qui est plus rapide mais signifie qu'un crash ultérieur du système d'exploitation peut laisser le répertoire de données corrompu. Généralement, cette option est utile dans le cas de tests mais elle ne devrait pas être utilisée pour une installation en production.

`-o` *`options`*
`--old-options` *`options`*

options à passer directement à l'ancienne commande `postgres` ; les invocations multiples de cette option sont cumulées

`-O` *`options`*
`--new-options` *`options`*

options à passer directement à la nouvelle commande `postgres` ; les invocations multiples de cette commande sont cumulées

`-p` *`port`*
`--old-port=`*`port`*

le numéro de port de l'ancienne instance ; variable d'environnement`PGPORTOLD`

`-P` *`port`*
`--new-port=`*`port`*

le numéro de port de la nouvelle instance ; variable d'environnement`PGPORTNEW`

`-r`
`--retain`

conserver les fichiers SQL et de traces y compris après avoir terminé avec succès

`-s` *dir*
`--socketdir=`*dir*

répertoire utilisé pour stocker les sockets lors de la mise à jour ; la valeur par défaut est le répertoire courant ; la variable d'environnement est `PGSOCKETDIR`

`-U` *username*
`--username=`*username*

nom d'utilisateur de l'instance d'installation ; variable d'environnement`PGUSER`

`-v`
`--verbose`

activer la trace interne verbeuse

`-V`
`--version`

afficher les informations de version, puis quitter

`--clone`

Utilise le clonage efficace de fichiers (connu aussi sous le nom de « reflinks » sur certains systèmes) au lieu de copier les fichiers vers la nouvelle instance. Ceci peut résulter en une copie pratiquement instantanée des fichiers de données, donnant l'avantage en vitesse de l'option `-k`/`--link` tout en laissant l'ancienne instance non modifiée.

Le clonage de fichiers est seulement supporté sur certains systèmes d'exploitation et systèmes de fichiers. Si cette option est sélectionnée alors qu'elle n'est pas supportée, l'exécution de pg_upgrade renverra une erreur. Actuellement, cette option est supportée sous Linux (à partir du noyau 4.5) avec Btrfs et XFS (pour les systèmes de fichiers créés avec le support de reflink), et sur macOS avec APFS.

`-?`
`--help`

afficher l'aide, puis quitter

Usage

Ci-dessous les étapes pour effectuer une mise à jour avec pg_upgrade :

1. **Si nécessaire, déplacez l'ancienne instance**

   Si vous utilisez un répertoire d'installation spécifique par version, exemple `/opt/PostgreSQL/15`, vous n'avez pas besoin de déplacer l'ancienne instance. Les installateurs graphiques utilisent tous des répertoires d'installation spécifiques par version.

   Si votre répertoire d'installation n'est pas spécifique par version, par exemple `/usr/local/pgsql`, il est nécessaire de déplacer le répertoire d'installation courant de PostgreSQL de telle manière à ce qu'il n'interfère pas avec la nouvelle installation de PostgreSQL. Une fois que le serveur courant PostgreSQL est éteint, il est sans danger de renommer le répertoire d'installation de PostgreSQL ; en supposant que l'ancien répertoire est `/usr/local/pgsql`, vous pouvez faire :

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

   pour renommer le répertoire.

2. **Pour les installations à partir des sources, construisez la nouvelle version**

   Construisez la nouvelle version de PostgreSQL à partir des sources avec des options de `configure` qui sont compatibles avec l'ancienne instance. pg_upgrade utilisera `pg_controldata` pour s'assurer que l'ensemble des configurations sont compatibles avant de commencer la mise à jour.

3. **Installez les nouveaux binaires PostgreSQL**

   Installez les binaires du nouveau serveur et les fichiers associés. Par défaut, pg_upgrade est inclus dans une installation.

   Pour les installations à partir des sources, si vous souhaitez installer le nouveau serveur dans un répertoire personnalisé, utilisez la variable `prefix` :

   ```
   make prefix=/usr/local/pgsql.new install
   ```

4. **Initialisez la nouvelle instance PostgreSQL**

   Initialisez la nouvelle instance en utilisant la commande `initdb`. À nouveau, utilisez des options de la commande `initdb` compatibles avec l'ancienne instance. Beaucoup d'installateurs pré-construits effectuent cette étape automatiquement. Il n'est pas nécessaire de démarrer la nouvelle instance.

5. **Installez les fichiers objets partagés d'extension**

   Beaucoup d'extensions et de modules personnalisés, qu'ils viennent de `contrib` ou d'une autre source, utilisent les fichiers d'objets partagés (ou DLL), par exemple `pgcrypto.so`. Si l'ancienne instance les utilisait, les fichiers d'objets partagés correspondant aux binaires du nouveau serveur doivent être installés dans la nouvelle instance, habituellement avec les commandes du système d'exploitation. Ne chargez pas les définitions de schéma, par exemple `CREATE EXTENSION pgcrypto`, parce qu'elles seront dupliquées à partir de l'ancienne instance. Si des mises à jour d'extensions sont disponibles, pg_upgrade l'indiquera et créera un script à exécuter plus tard pour les mettre à jour.

6. **Copier les fichiers personnalisés de recherche plein texte**

   Copiez tous les fichiers personnalisés de recherche plein texte (dictionnaire, synonymes, thésaurus, mots d'arrêt) de l'ancienne instance vers la nouvelle.

7. **Ajuster l'authentification**

   `pg_upgrade` se connectera à l'ancien et au nouveau serveur plusieurs fois, aussi vous pourriez avoir besoin de positionner l'authentification sur `peer` ou d'utiliser un fichier `~/.pgpass` (voir Section 34.16).

8. **Arrêtez les deux serveurs**

   Assurez vous que les deux serveurs sont arrêtés en utilisant, sur Unix par exemple :

   ```
   pg_ctl -D /opt/PostgreSQL/9.6 stop
   pg_ctl -D /opt/PostgreSQL/15 stop
   ```

   ou sur Windows, en utilisant les noms de services corrects :

   ```
   NET STOP postgresql-9.6
   NET STOP postgresql-15
   ```

Les serveurs standby par réplication en flux et par copie des journaux doivent être en cours d'exécution jusqu'à une étape ultérieure.

9. **Préparez la mise à jour d'un serveur standby**

   Si vous êtes en train de mettre à jour des serveurs standby en suivant la description de la section Étape 11, vérifiez en utilisant pg_controldata sur les anciennes instances primaires et standby que les anciens serveurs standby sont à jour. Vérifiez que les valeurs de « Latest checkpoint location » correspondent dans toutes les instances. De plus, assurez-vous que le paramètre `wal_level` ne soit pas configuré avec la valeur `minimal` dans le fichier de configuration `postgresql.conf` sur la nouvelle instance primaire.

10. **Lancez pg_upgrade**

    Lancez toujours le binaire pg_upgrade du nouveau serveur, pas celui de l'ancien. pg_upgrade exige la spécification des anciens et des nouveaux répertoires de données et des exécutables (`bin`). Vous pouvez aussi indiquer des valeurs pour les utilisateurs et les ports, et si vous voulez que les fichiers de données soient liées ou clonées plutôt que copiées (par défaut ce dernier).

    Si vous utilisez le mode lien, la mise à jour sera beaucoup plus rapide (pas de copie de fichiers) et utilisera moins d'espace disque, mais vous ne serez plus en mesure d'accèder à votre ancienne instance une fois que la nouvelle instance sera démarrée après la mise à jour. Le mode lien exige également que le répertoire de données de l'ancienne et de la nouvelle instance soient dans le même système de fichiers. (Les tablespaces et `pg_wal` peuvent être sur des systèmes de fichiers différents.) Le mode de clonage fournit les mêmes avantages au niveau vitesse et espace disque mais ne rend pas l'ancienne instance inutilisable une fois que la nouvelle instance a été démarrée. Le mode de clonage requiert aussi que les répertoires de données de l'ancienne et la nouvelle instances soient dans le même système de fichiers. Ce mode est seulement disponible sur certains systèmes d'exploitation et certains systèmes de fichiers.

    L'option `--jobs` permet l'utilisation de plusieurs cœurs CPU pour copier ou lier des fichiers, et pour sauvegarder et recharger les schémas des bases de données en parallèle ; un bon chiffre pour commencer est le maximum du nombre de cœurs CPU et des tablespaces. Cette option peut réduire dramatiquement le temps pour mettre à jour un serveur avec plusieurs bases de données s'exécutant sur une machine multiprocesseur.

    Pour les utilisateurs Windows, vous devez être connecté avec un compte administrateur, et lancer un shell sous l'utilisateur `postgres` en positionnant le chemin correct :

```
RUNAS /USER:postgres "CMD.EXE"
SET PATH=%PATH%;C:\Program Files\PostgreSQL\15\bin;
```

    puis lancez pg_upgrade avec les répertoires entre guillemets, par exemple :

```
pg_upgrade.exe
        --old-datadir "C:/Program Files/PostgreSQL/9.6/data"
        --new-datadir "C:/Program Files/PostgreSQL/15/data"
        --old-bindir "C:/Program Files/PostgreSQL/9.6/bin"
        --new-bindir "C:/Program Files/PostgreSQL/15/bin"
```

    Une fois démarré, `pg_upgrade` vérifiera que les deux instances sont compatibles avant d'effectuer la mise à jour. Vous pouvez utiliser `pg_upgrade --check` pour effectuer uniquement la vérification, y compris si l'ancien serveur est actuellement en fonctionnement. `pg_upgrade --check` mettra également en évidence les ajustements manuels nécessaires

que vous aurez besoin de faire après la mise à jour. Si vous désirez utiliser le mode lien ou le mode clone, vous devriez indiquer l'option `--link` ou `--clone` avec l'option `--check` pour activer les vérifications spécifiques au mode lien. `pg_upgrade` doit avoir le droit d'écrire dans le répertoire courant.

Évidemment, personne ne doit accèder aux instances pendant la mise à jour. pg_upgrade lance par défaut les serveurs sur le port 50432 pour éviter les connexions non désirées de clients. Vous pouvez utilisez le même numéro de port pour les deux instances lors d'une mise à jour car l'ancienne et la nouvelle instance ne fonctionneront pas en même temps. Cependant, lors de la vérification d'un ancien serveur en fonctionnement, l'ancien et le nouveau numéros de port doivent être différents.

Si une erreur survient lors de la restauration du schéma de la base de données, `pg_upgrade` quittera et vous devrez revenir à l'ancienne instance comme décrit ci-dessous (Étape 17). Pour réessayer `pg_upgrade`, vous aurez besoin de modifier l'ancienne instance de telle manière que la restauration du schéma par pg_upgrade réussisse. Si le problème est un module `contrib`, vous pourriez avoir besoin de désinstaller le module `contrib` de l'ancienne instance et le réinstaller dans la nouvelle instance après la mise à jour, en supposant que le module n'est pas utilisé pour stocker des données utilisateur.

11. **Mettez à jour les serveurs standby par réplication en flux et copie de journaux**

    Si vous utilisez le mode lien et avez des serveurs standby par réplication continue (voir Section 27.2.5) ou par copie des journaux de transactions (voir Section 27.2), vous pouvez suivre les étapes ci-dessous pour les mettre à jour rapidement. Vous ne lancerez pas pg_upgrade sur les serveurs standby, mais plutôt rsync sur le primaire. Ne démarrez encore aucun serveur.

    Si vous n'utilisez *pas* le mode lien, n'avez pas ou ne voulez pas utiliser rsync, ou si vous voulez une solution plus simple, ignorez les instructions de cette section et recréez simplement les serveurs standbys une fois que pg_upgrade a terminé et que le nouveau primaire fonctionne de nouveau.

    1. **Installez les nouveaux binaires PostgreSQL sur les serveurs standy**

       Assurez-vous que les nouveaux binaires et fichiers de support sont installés sur tous les serveurs standby.

    2. **Assurez vous que les nouveaux répertoires de données sur les serveurs standby n'existent *pas***

       Assurez vous que les nouveaux répertoires de données sur les serveurs standby n'existent *pas* ou sont vides. Si initdb a été lancé, détruisez les nouveaux répertoires de données des serveurs standby.

    3. **Installez les fichiers objets partagés d'extension**

       Installez les mêmes fichiers objets partagés d'extension sur les nouveaux serveurs standby que vous avez installé sur la nouvelle instance maître.

    4. **Arrêtez les serveurs standby**

       Si les serveurs standby sont encore lancés, arrêtez les maintenant en utilisant les instructions ci-dessus.

    5. **Sauvegardez les fichiers de configuration**

       Sauvegardez tous les fichiers de configuration des anciens serveurs standby que vous avez besoin de conserver, par exemple `postgresql.conf` (et tout fichier qu'il inclut), `postgresql.auto.conf`, `pg_hba.conf`, dans la mesure où ceux-ci seront réécrits ou supprimés dans l'étape suivante.

6. **Lancez rsync**

   Lors de l'utilisation du mode lien, les serveurs standbys peuvent être rapidement mis à jour en utilisant rsync. Pour cela, à partir d'un répertoire du serveur primaire situé au-dessus des répertoires de l'ancienne et de la nouvelle instance de bases de données, exécutez cette commande sur le *primaire* pour chaque serveur standby :

```
rsync --archive --delete --hard-links --size-only --
no-inc-recursive ancien_rep_config nouveau_rep_config
 repertoire_distant
```

   où `ancien_rep_config` et `nouveau_rep_config` sont relatifs au répertoire courant du serveur primaire, et `repertoire_distant` est *au-dessus* des ancien et nouveau répertoires des instances sur le serveur standby. La structure des répertoires sous les répertoires spécifiés du primaire et des secondaires doit correspondre. Consultez les pages du manuel de rsync pour des détails sur la manière de spécifier le répertoire distant, par exemple :

```
rsync --archive --delete --hard-links --size-only --no-inc-
recursive /opt/PostgreSQL/9.5 \
      /opt/PostgreSQL/9.6 standby.example.com:/opt/
PostgreSQL
```

   Vous pouvez vérifier ce que la commande va faire en utilisant l'option `--dry-run` de rsync. Alors que rsync doit être exécuté sur le primaire pour au moins un serveur standby, il est possible d'exécuter rsync sur un standby mis à jour pour mettre à jour les autres standbys tant que le standby mis à jour n'est pas démarré.

   Cela enregistre les liens créés par le mode lien de pg_upgrade qui connecte les fichiers dans les ancienne et nouvelle instances du serveur primaire. Puis, il trouve les fichiers correspondant dans l'ancienne instance du standby et crée les liens pour eux dans la nouvelle instance du serveur standby. Les fichiers qui n'ont pas été liés sur le primaire sont copiés sur à partir du serveur primaire vers le serveur secondaire. (Ils sont généralement petits.) Ceci fournit des mises à jour rapides des serveurs secondaires. Malheureusement, rsync copie sans raison les fichiers associés aux tables temporaires et non journalisées parce que ces fichiers n'existent normalement pas sur les serveurs secondaires.

   Si vous avez des tablespaces, vous aurez besoin de lancer une commande rsync similaire pour chaque répertoire de tablespace, par exemple :

```
rsync --archive --delete --hard-links --size-only --no-inc-
recursive /vol1/pg_tblsp/PG_9.5_201510051 \
      /vol1/pg_tblsp/PG_9.6_201608131 standby.example.com:/
vol1/pg_tblsp
```

   Si vous avez déplacé `pg_wal` en dehors des répertoires de données, rsync doit être lancé aussi sur ces répertoires.

7. **Configurez les serveurs standby par réplication en flux et par copie de fichiers**

   Configurez les serveurs pour les copies des fichiers de transactions. (Vous n'avez pas besoin d'exécuter les fonctions `pg_backup_start()` et `pg_backup_stop()` ou effectuer

une sauvegarde des fichiers car les secondaires sont toujours synchronisés avec le primaire.) Les slots de réplication ne sont pas copiés et doivent être créés de nouveau.

12. **Restaurez `pg_hba.conf`**

    Si vous avez modifié `pg_hba.conf`, restaurez cette configuration d'origine. Il peut être aussi nécessaire d'ajuster d'autres fichiers de configuration dans la nouvelle instance pour correspondre à l'ancienne instance, par exemple `postgresql.conf` (et tout fichier qu'il inclut), `postgresql.auto.conf`.

13. **Démarrez le nouveau serveur**

    Le nouveau serveur peut maintenant être démarré en toute sécurité, puis les autres serveurs standby synchronisés avec rsync.

14. **Traitements après mise à jour**

    Si des traitements après mise à jour sont nécessaires, pg_upgrade affichera des avertissements lors de son travail. Il générera également des scripts qui devront être lancés par l'administrateur. Les scripts se connecteront à chaque base de données qui ont besoin de traitements après mise à jour. Chaque script devrait être lancé comme suit :

    ```
    psql --username=postgres --file=script.sql postgres
    ```

    Les scripts peuvvent être lancés dans n'importe quel ordre et détruits une fois terminés.

    > **Attention**
    >
    > Généralement, il n'est pas sûr d'accèder des tables référencées dans les scripts de reconstruction avant la fin de leurs traitements ; le faire pourrait entraîner des résultats incorrects ou de médiocres performances. Les tables non référencées dans les scripts de reconstruction peuvent être accédées immédiatement.

15. **Statistiques**

    Parce que les statistiques de l'optimiseur ne sont pas transférées par `pg_upgrade`, vous serez invités à lancer une commande pour regénérer les statistiques à la fin de la mise à jour. Vous pourriez avoir besoin de positionner les paramètres de connexion pour qu'ils correspondent à votre nouvelle instance.

16. **Détruire les anciennes instances**

    Une fois que vous êtes satisfait de la mise à jour, vous pouvez détruire les répertoires de données des anciennes instances en lançant le script indiqué par `pg_upgrade` à la fin de son traitement. (La destruction automatique n'est pas possible si vous avez défini des tablespaces personnalisés dans l'ancien répertoire de données.) Vous pouvez également supprimer les anciens répertoires d'installation (par exemple `bin`, `share`).

17. **Revenir à l'ancienne instance**

    Si, après avoir lancé `pg_upgrade`, vous désirez revenir à l'ancienne instance, il y a plusieurs options :

    - Si l'option `--check` a été utilisée, l'ancienne instance n'a pas été modifiée, elle peut être redémarrée.

- Si l'option `--link` n'a *pas* été utilisée, l'ancienne instance n'a pas été modifiée, elle peut être redémarrée.
- Si l'option `--link` a été utilisée, les fichiers de données pourraient être partagés entre l'ancienne instance et la nouvelle :
  - Si `pg_upgrade` a annulé avant de réaliser les liens, l'ancienne instance n'a pas été modifiée, elle peut être redémarrée.
  - Si vous n'avez *pas* démarré la nouvelle instance, l'ancienne instance n'a pas été modifiée sauf, quand les liens ont commencé, un suffixe `.old` a été ajouté au fichier `$PGDATA/global/pg_control`. Pour utiliser de nouveau l'ancienne instance, supprimez le suffixe `.old` du fichier `$PGDATA/global/pg_control` ; vous pouvez alors redémarrer l'ancienne instance.
  - Si vous avez démarré la nouvelle instance, elle a écrit dans des fichiers partagés et il est dangereux d'utiliser l'ancienne instance. Cette dernière doit être restaurée d'une sauvegarde dans ce cas.

Notes

pg_upgrade crée différents fichiers de travail, tel que des sauvegardes de schémas, enregistré dans le sous-répertoire `pg_upgrade_output.d` du répertoire principal de la nouvelle instance. Chaque exécution crée un sous-répertoire nommé avec un horodatage formaté d'après ISO 8601 (`%Y%m%dT%H%M%S`), où tous ses fichiers générés sont stockés. `pg_upgrade_output.d` et ses fichiers seront supprimés automatiquement si pg_upgrade se termine avec succès ; en cas de problème, ses fichiers pourraient fournir des informations de debug très utiles.

pg_upgrade lance brièvement les serveurs dans les ancien et nouveau répertoires de données. Les fichiers temporaires des sockets Unix pour communiquer avec ces serveurs sont, par défaut, créés dans le répertoire courant. Dans certaines situations, le nom du chemin pour le répertoire courant pourrait être trop long pour être un nom de socket valide. Dans ce cas, vous pouvez utiliser l'option `-s` pour placer les fichiers socket dans un autre répertoire dont le nom du chemin est plus court. Pour des raisons de sécurité, assurez-vous que le répertoire n'est ni lisible ni modifiable par les autres utilisateurs. (Ceci n'est pas applicable à Windows.)

Tous les échecs, reconstructions et réindexations seront reportés par pg_upgrade s'ils affectent votre installation ; les scripts d'après mise à jour pour reconstruire les tables et index seront générés automatiquement. Si vous essayez d'automatiser la mise à jour de plusieurs instances, vous devriez constater que les instances avec des schémas de bases de données identiques ont besoin des mêmes étapes après mise à jour ; car les étapes après mise à jour sont basées sur les schémas des bases de données, et pas sur les données utilisateurs.

Pour les déploiements de tests, créez uniquement une copie du schéma de l'ancienne instance, insérez des données de tests, et faites la mise à jour.

pg_upgrade ne supporte pas la mise à jour de bases de données contenant des colonnes de table utilisant les types de données systèmes référençant les OID, nommés `reg*` :

```
regcollation
regconfig
regdictionary
regnamespace
regoper
regoperator
regproc
regprocedure
```

(`regclass`, `regrole` et `regtype` peuvent être mis à jour.)

Si vous souhaitez utiliser le mode lien et ne voulez pas que votre ancienne instance ne soit modifiée lorsque la nouvelle instance est démarré, considérez l'utilisation du mode clone. Si ce n'est pas possible, faites une copie de l'ancienne instance et faites la mise à jour à partir de cette copie. Pour faire une copie valide de l'ancienne instance, utilisez `rsync` pour effectuer une copie grossière de l'ancienne instance lancée, puis arrêtez l'ancien serveur et lancez `rsync --checksum` à nouveau pour mettre à jour la copie dans un état cohérent avec tous les changements. (L'option `--checksum` est nécessaire car `rsync` n'a une granularité sur les dates de modification de fichiers que d'une seconde.) Vous pourriez souhaiter exclure certains fichiers, par exemple `postmaster.pid`, comme documenté à Section 26.3.3. Si votre système de fichiers supporte les images de système de fichiers ou la fonctionnalité Copy-On-Write, vous pouvez utiliser ces fonctionnalités pour faire une sauvegarde de l'ancienne instance et des tablespaces, bien que l'image et les copies doivent être créées simultanément ou lorsque le serveur de bases de données est éteint.

Voir aussi

initdb, pg_ctl, pg_dump, postgres

pg_waldump

pg_waldump — affiche une version lisible du contenu des fichiers WAL (journaux de transactions) d'une instance PostgreSQL

Synopsis

`pg_waldump` [`option`...] [`startseg` [`endseg`]]

Description

`pg_waldump` affiche une version lisible des journaux de transactions (appelés aussi fichiers WAL), ce qui peut être très utile pour le debugging ou l'apprentissage.

Cet utilitaire peut seulement être lancé par l'utilisateur qui a installé l'instance car il nécessite un accès en lecture seule sur le répertoire principal des données.

Options

Les options suivantes de la ligne de commande vérifient l'emplacement et le format de la sortie :

`startseg`

On commence à lire au niveau du segment de journal spécifié. Implicitement, cela détermine le chemin dans lequel les fichiers vont être cherchés et la timeline à utiliser.

`endseg`

On arrête de lire au niveau du segment de journal spécifié.

`-b`
`--bkp-details`

Permet de renvoyer des informations détaillées sur les blocs de sauvegarde.

`-B` *`block`*
`--block=`*`block`*

Affiche seulement les enregistrements qui modifient le bloc indiqué. La relation doit aussi être indiquée avec les arguments `--relation` ou `-R`.

`-e` *`end`*
`--end=`*`end`*

Arrête la lecture à une position dans le journal spécifié, au lieu de lire jusqu'à la fin du flux.

`-f`
`--follow`

Après avoir atteint la fin d'un fichier WAL valide, la commande vérifie toutes les secondes si un nouveau fichier WAL est apparu.

`-F` *`fork`*
`--fork=`*`fork`*

Si indiqué, affiche seulement les enregistrements qui modifient des blocs dans le *fork* donné. Les valeurs valides sont `main` pour la partie principale, `fsm` pour la carte des espaces libres, `vm` pour la carte de visibilité, et `init` pour la partie init.

`-n` *`limite`*
`--limit=`*`limite`*

Affiche seulement le nombre spécifié d'enregistrements, puis s'arrête.

`-p` *`chemin`*
`--path=`*`chemin`*

Indique un répertoire où rechercher les segments de journaux de transactions ou un répertoire contenant un sous-répertoire `pg_wal` qui contient ces fichiers. Par défaut, l'outil recherche dans le répertoire courant, dans le sous-répertoire `pg_wal` du répertoire courant et dans le sous-répertoire `pg_wal` du répertoire ciblé par `PGDATA`.

`-q`
`--quiet`

N'affiche aucun message, sauf en cas d'erreurs. Cette option peut être utile quand vous voulez connaître l'intervalle d'enregistrements pouvant être analysés mais que vous n'ayez pas besoin du contenu des enregistrements.

`-r` *`rmgr`*
`--rmgr=`*`rmgr`*

N'affiche que les enregistrements générés par le gestionnaire de ressources spécifié. Vous pouvez indiquer l'option plusieurs fois pour sélectionner plusieurs gestionnaires de ressources. Si `list` est positionné comme un nom, alors cela affiche la liste des gestionnaires valides, puis quitte.

Les extensions peuvent définir des gestionnaires personnalisés de ressources, mais pg_waldump ne charge pas le module d'extension et de ce fait, ne reconnaît pas les gestionnaires personnalisés de ressources par nom. À la place, vous pouvez indiquer les gestionnaires personnalisés de ressource comme `custom###` où "###" est l'identifiant du gestionnaire de ressources sur trois chiffres. Les noms de cette forme seront toujours considérés valides.

`-R` *`tblspc/db/rel`*
`--relation=`*`tblspc/db/rel`*

Affiche seulement les enregistrements qui modifient des blocs dans la relation indiquée. La relation est indiquée avec l'OID du tablespace, l'OID de la base et le relfilenode, séparés par des slashs, par exemple `1234/12345/12345`. C'est le même format utilisé pour les relations dans la sortie du programme.

`-s` *`début`*
`--start=`*`début`*

Position dans le journal de début de lecture. Par défaut, la lecture commence au premier enregistrement valide trouvé dans le fichier le plus ancien trouvé.

`-t` *`timeline`*
`--timeline=`*`timeline`*

La timeline des journaux depuis laquelle on lit les enregistrements. Le comportement par défaut prendra la valeur trouvée dans *`startseg`*, s'il est spécifié, sinon la valeur par défaut sera 1.

`-V`
`--version`

Affiche la version de pg_waldump, puis quitte.

`-w`
`--fullpage`

Affiche seulement les enregistrements qui incluent des images de bloc complet.

`-x` *`xid`*
`--xid=`*`xid`*

N'affiche que les enregistrements marqués avec l'identifiant de transaction donné.

`-z`
`--stats[=enregistrement]`

Affiche un résumé des statistiques (nombre, taille des enregistrements et bloc complet) au lieu des enregistrements individuels. En option, il peut générer les statistiques par enregistrement plutôt que par gestionnaire de ressources.

Si pg_waldump est terminé par le signal SIGINT (**Control**+**C**), le résumé des statistiques calculées est affiché jusqu'à ce moment. Cette opération n'est pas supportée sur Windows.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de pg_waldump puis quitte.

Environment

`PGDATA`

Répertoire des données ; voir aussi l'option `-p`.

`PG_COLOR`

Indique s'il faut utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto`, `never`.

Notes

Les résultats peuvent être erronés lorsque le serveur est démarré.

Seule la timeline spécifiée est affichée (ou celle par défaut s'il n'y en a pas de spécifiée). Les enregistrements des autres timelines sont ignorés.

pg_waldump ne peut pas lire les fichiers suffixés avec `.partial`. Si ces fichiers ont tout de même besoin d'être lus, le suffixe `.partial` doit être retiré du nom du fichier.

Voir aussi

Section 30.6

postgres

postgres — Serveur de bases de données PostgreSQL

Synopsis

`postgres` [*option*...]

Description

`postgres` est le serveur de bases de données PostgreSQL. Pour qu'une application cliente puisse accéder à une base de données, elle se connecte (soit via le réseau soit localement) à un processus `postgres` en cours d'exécution. L'instance `postgres` démarre ensuite un processus serveur séparé pour gérer la connexion.

Une instance `postgres` gère toujours les données d'un seul cluster. Un cluster est un ensemble de bases de données stocké à un même emplacement dans le système de fichiers (le « répertoire des données »). Plus d'un processus `postgres` peut être en cours d'exécution sur un système à un moment donné, s'ils utilisent des répertoires différents et des ports de communication différents (voir ci-dessous). Quand `postgres` se lance, il a besoin de connaître l'emplacement du répertoire des données. Cet emplacement doit être indiquée par l'option `-D` ou par la variable d'environnement `PGDATA` ; il n'y a pas de valeur par défaut. Typiquement, `-D` ou `PGDATA` pointe directement vers le répertoire des données créé par initdb. D'autres dispositions de fichiers possibles sont discutés dans Section 20.2. Un répertoire de données est créé avec initdb.

Par défaut, `postgres` s'exécute en avant-plan et affiche ses messages dans le flux standard des erreurs. En pratique, `postgres` devrait être exécuté en tant que processus en arrière-plan, par exemple au lancement.

La commande `postgres` peut aussi être appelé en mode mono-utilisateur. L'utilisation principal de ce mode est lors du « bootstrap » utilisé par initdb. Quelque fois, il est utilisé pour du débogage et de la récupération suite à un problème (mais noter qu'exécuter un serveur en mode mono-utilisateur n'est pas vraiment convenable pour déboguer le serveur car aucune communication inter-processus réaliste et aucun verrouillage n'interviennent.) Quand il est appelé en mode interactif à partir du shell, l'utilisateur peut saisir des requêtes et le résultat sera affiché à l'écran mais dans une forme qui est plus utile aux développeurs qu'aux utilisateurs. Dans le mode mono-utilisateur, la session ouverte par l'utilisateur sera configurée avec l'utilisateur d'identifiant 1 et les droits implicites du superutilisateur lui sont donnés. Cet utilisateur n'a pas besoin d'exister, donc le mode mono-utilisateur peut être utilisé pour récupérer manuellement après certains types de dommages accidentels dans les catalogues systèmes.

Options

`postgres` accepte les arguments suivants en ligne de commande. Pour une discussion détaillée des options, consultez Chapitre 20. Vous pouvez éviter de saisir la plupart de ces options en les initialisant dans le fichier de configuration. Certaines options (sûres) peuvent aussi être configurées à partir du client en cours de connexion d'une façon dépendante de l'application, configuration qui ne sera appliquée qu'à cette session. Par exemple si la variable d'environnement `PGOPTIONS` est configurée, alors les clients basés sur libpq passeront cette chaîne au serveur qui les interprétera comme les options en ligne de commande de `postgres`.

Général

`-B` *ntampons*

Configure le nombre de tampons partagés utilisés par les processus serveur. La valeur par défaut de ce paramètre est choisi automatiquement par initdb. Indiquer cette option est équivalent à configurer le paramètre shared_buffers.

`-c` *`nom=valeur`*

Configure un parammètre d'exécution nommé. Les paramètres de configuration supportés par PostgreSQL sont décrits dans Chapitre 20. La plupart des autres options en ligne de commande sont en fait des formes courtes d'une affectation de paramètres. `-c` peut apparaître plusieurs fois pour configurer différents paramètres.

`-C` *`nom`*

Affiche la valeur d'un paramètre d'exécution nommé, ouis quitte. (Voir l'option `-c` ci-dessus pour les détails.) Cela renvoie les valeurs du `postgresql.conf`, modifiées par tout paramètre fourni lors de cet appel. Cela ne reflète pas les paramètres fournis lors de la création de l'instance.

Ceci peut être utilisé sur un serveur en cours d'exécutio pour la plupart des paramètres. Néanmoins, le serveur doit être arrêté pour certains paramètres calculés à l'exécution (par exemple shared_memory_size, shared_memory_size_in_huge_pages et wal_segment_size).

Cette option a pour but de permettre aux autres programmes d'interagir avec un outil comme pg_ctl pour récupérer des valeurs de configuration. Les applications utilisateurs devraient plutôt utiliser la commande `SHOW` ou la vue `pg_settings`.

`-d` *`niveau-débogage`*

Configure le niveau de débogage. Plus haute est sa valeur, plus importante seront les traces écrites dans les journaux. Les valeurs vont de 1 à 5. Il est aussi possible de passer `-d 0` pour une session spécifique qui empêchera le niveau des traces serveur du processus `postgres` parent d'être propagé jusqu'à cette session.

`-D` *`repdonnées`*

Indique le répertoire des fichier(s) de configuration. Voir Section 20.2 pour les détails.

`-e`

Configure le style de date par défaut à « European », c'est-à-dire l'ordre `DMY` pour les champs en entrée. Ceci cause aussi l'affichage de la date avant le mois dans certains formats de sortie de date. Voir Section 8.5 pour plus d'informations.

`-F`

Désactive les appels `fsync` pour améliorer les performances au risque de corrompre des données dans l'idée d'un arrêt brutal du système. Spécifier cette option est équivalent à désactiver le paramètre de configuration fsync. Lisez la documentation détaillée avant d'utiliser ceci !

`-h` *`hôte`*

Indique le nom d'hôte ou l'adresse IP sur lequel `postgres` attend les connexions TCP/IP d'applications clientes. La valeur peut aussi être une liste d'adresses séparées par des virgules ou `*` pour indiquer l'attente sur toutes les interfaces disponibles. Une valeur vide indique qu'il n'attend sur aucune adresse IP, auquel cas seuls les sockets de domaine Unix peuvent être utilisés pour se connecter au serveur. Par défaut, attend les connexions seulement sur localhost. Spécifier cette option est équivalent à la configurer dans le paramètre listen_addresses.

`-i`

Autorise les clients distants à se connecter via TCP/IP (domaine Internet). Sans cette option, seules les connexions locales sont autorisées. Cette option est équivalent à la configuration du paramètre `listen_addresses` à `*` dans `postgresql.conf` ou via `-h`.

Cette option est obsolète car il ne permet plus l'accès à toutes les fonctionnalités de listen_addresses. Il est généralement mieux de configurer directement `listen_addresses`.

`-k` *`directory`*

Indique le répertoire de la socket de domaine Unix sur laquelle `postgres` est en attente des connexions des applications clients. Ce paramètre peut aussi contenir une liste de répertoires séparés par des virgules. Une valeur vide précise que le serveur ne doit pas écouter à des sockets de domaine Unix, auquel cas seul les sockets TCP/IP pourront être utilisés pour se connecter. La valeur par défaut est habituellement `/tmp`, mais cela peut être changé au moment de la compilation. Spécifier cette option est équivalent à configurer le paramètre unix_socket_directories.

`-l`

Active les connexions sécurisées utilisant SSL. PostgreSQL doit avoir été compilé avec SSL pour que cette option soit disponible. Pour plus d'informations sur SSL, référez-vous à Section 19.9.

`-N` *`max-connections`*

Initialise le nombre maximum de connexions clientes que le serveur acceptera. La valeur par défaut de ce paramètre est choisi automatiquement par initdb. Indiquer cette option est équivalent à configurer le paramètre max_connections.

`-p` *`port`*

Indique le port TCP/IP ou l'extension du fichier socket de domaine Unix sur lequel `postgres` attend les connexions des applications clientes. Par défaut, la valeur de la variable d'environnement `PGPORT` environment ou, si cette variable n'est pas configurer, la valeur connue à la compilation (habituellement 5432). Si vous indiquez un port autre que celui par défaut, alors toutes les applications clientes doivent indiquer le même numéro de port soit dans les options en ligne de commande soit avec `PGPORT`.

`-s`

Affiche une information de temps et d'autres statistiques à la fin de chaque commande. Ceci est utile pour créer des rapports de performance ou pour configurer finement le nombre de tampons.

`-S` *`work-mem`*

Indique la quantité de mémoire à utiliser par les tris internes et par les hachages avant d'utiliser des fichiers disque temporaires. Voir la description du paramètre `work_mem` dans Section 20.4.1.

`-V`
`--version`

Affiche la version de postgres, puis quitte.

`--`*`nom=valeur`*

Configure un paramètre à l'exécution ; c'est une version courte de `-c`.

`--describe-config`

Cette option affiche les variables de configuration internes du serveur, leurs descriptions et leurs valeurs par défaut dans un format `COPY` délimité par des tabulations. Elle est conçue principalement pour les outils d'administration.

`-?`
`--help`

Affiche l'aide des arguments en ligne de commande sur postgres, puis quitte.

Options semi-internes

Les options décrites ici sont utilisées principalement dans un but de débogage et pouvant quelque fois aider à la récupération de bases de données très endommagées/ Il n'y a aucune raison pour

les utiliser dans la configuration d'un système en production. Elles sont listées ici à l'intention des développeurs PostgreSQL. De plus, une de ces options pourrait disparaître ou changer dans le futur sans avertissement.

`-f { s | i | o | b | t | n | m | h }`

Interdit l'utilisation de parcours et de méthode de jointure particulières. `s` et `i` désactivent respectivement les parcours séquentiels et d'index, `o`, `b` et `t` désactivent respectivement les parcours d'index seul, les parcours d'index bitmap et les parcours de TID alors que `n`, `m` et `h` désactivent respectivement les jointures de boucles imbriquées, jointures de fusion et de hachage.

Ni les parcours séquentiels ni les jointures de boucles imbriquées ne peuvent être désactivés complètement ; les options `-fs` et `-fn` ne font que décourager l'optimiseur d'utiliser ce type de plans.

`-n`

Cette option est présente pour les problèmes de débogage du genre mort brutal d'un processus serveur. La stratégie habituelle dans cette situation est de notifier tous les autres processus serveur qu'ils doivent se terminer, puis réinitialiser la mémoire partagée et les sémaphores. Tout ceci parce qu'un processus serveur errant peut avoir corrompu certains états partagés avant de terminer. Cette option spécifie seulement que `postgres` ne réinitialisera pas les structures de données partagées. Un développeur système avec quelques connaissances peut utiliser un débogueur pour examiner l'état de la mémoire partagée et des sémaphores.

`-O`

Autorise la modification de la structure des tables système. C'est utilisé par `initdb`.

`-P`

Ignore les index système lors de la lecture des tables système (mais les met à jour lors de la modification des tables). Ceci est utile lors de la récupération d'index système endommagés.

`-t pa[rser] | pl[anner] | e[xecutor]`

Affiche les statistiques en temps pour chaque requête en relation avec un des modules majeurs du système. Cette option ne peut pas être utilisée avec l'option `-s`.

`-T`

Cette option est présente pour les problèmes de débogage du genre mort brutal d'un processus serveur. La stratégie habituelle dans cette situation est de notifier tous les autres processus serveur qu'ils doivent se terminer, puis réinitialiser la mémoire partagée et les sémaphores. Tout ceci parce qu'un processus serveur errant peut avoir corrompu certains états partagés avant de terminer. Cette option spécifie seulement que `postgres` arrêtera tous les autres processus serveur en leur envoyant le signal `SIGSTOP` mais ne les arrêtera pas. Ceci permet aux développeurs système de récupérer manuellement des « core dumps » de tous les processus serveur.

`-v` *`protocole`*

Indique le numéro de version utilisé par le protocole interface/moteur pour une session particulière. Cette option est uniquement utilisée en interne.

`-W` *`secondes`*

Un délai de ce nombre de secondes survient quand un nouveau processus serveur est lancé, une fois la procédure d'authentification terminée. Ceci a pour but de permettre au développeur d'attacher un débogueur au processus serveur.

Options en mode mono-utilisateur

Les options suivantes s'appliquent uniquement en mode mono-utilisateur (voir Mode simple utilisateur ci-dessous).

`--single`

Sélectionne le mode mono-utilisateur. Cette option doit être la première sur la ligne de commande.

`base`

Indique le nom de la base à accèder. Il doit être le dernier argument. Si elle est omise, le nom de l'utilisateur est utilisé par défaut.

`-E`

Affiche toutes les commandes sur la sortie standard avant de les exécuter.

`-j`

Utilise un point-virgule suivi par deux retours à la ligne, plutôt qu'une seule comme marqueur de fin de commande.

`-r` *`fichier`*

Envoie toute la sortie des traces du serveur dans *`fichier`*. Cette option est seulement honorée quand elle est fournie en tant qu'option de ligne de commande.

Environnement

`PGCLIENTENCODING`

Jeu de caractères utilisé par défaut par tous les clients. (Les clients peuvent surcharger ce paramètre individuellement.) Cette valeur est aussi configurable dans le fichier de configuration.

`PGDATA`

Emplacement du répertoire des données par défaut

`PGDATESTYLE`

Valeur par défaut du paramètre en exécution datestyle. (Cette variable d'environnement est obsolète.)

`PGPORT`

Numéro de port par défaut (à configurer de préférence dans le fichier de configuration)

Diagnostics

Un message d'erreur mentionnant `semget` ou `shmget` indique probablement que vous devez configurer votre noyau pour fournir la mémoire partagée et les sémaphores adéquates. Pour plus de discussion, voir Section 19.4. Vous pouvez aussi repousser la configuration du noyau en diminuant shared_buffers pour réduire la consommation de la mémoire partagée utilisée par PostgreSQL, et/ou en diminuant max_connections pour réduire la consommation de sémaphores.

Un message d'erreur suggérant qu'un autre serveur est déjà en cours d'exécution devra vous demander une vérification attentive, par exemple en utilisant ls commandes

```
$ ps ax | grep postgres
```

ou

```
$ ps -ef | grep postgres
```

suivant votre système. Si vous êtes certain qu'il n'y a aucun serveur en conflit, vous pouvez supprimer le fichier verrou mentionné dans le message et tenter de nouveau.

Un message d'erreur indiquant une incapacité à se lier à un port indique que ce port est déjà utilisé par des processus autres que PostgreSQL. Vous pouvez aussi obtenir cette erreur si vous quittez `postgres` et le relancez immédiatement en utilisant le même port ; dans ce cas, vous devez tout simplement attendre quelques secondes pour que le système d'exploitation ferme bien le port avant de tenter de nouveau. Enfin, vous pouvez obtenir cette erreur si vous indiquez un numéro de port que le système considère comme réservé. Par exemple, beaucoup de versions d'Unix considèrent les numéros de port sous 1024 comme de « confiance » et permettent seulement leur accès par le superutilisateur Unix.

Notes

L'outil pg_ctl est utilisable pour lancer et arrêter le serveur `postgres` de façon sûre et confortable.

Si possible, *ne pas* utiliser `SIGKILL` pour tuer le serveur `postgres` principal. Le fait empêchera `postgres` de libérer les ressources système (c'est-à-dire mémoire partagée et sémaphores) qu'il détient avant de s'arrêter. Ceci peut poser problèmes lors du lancement d'un `postgres` frais.

Pour terminer le serveur `postgres` normalement, les signaux `SIGTERM`, `SIGINT` ou `SIGQUIT` peuvent être utilisés. Le premier attendra que tous les clients terminent avant de quitter, le second forcera la déconnexion de tous les clients et le troisième quittera immédiatement sans arrêt propre. Ce dernier amènera une récupération lors du redémarrage.

Le signal `SIGHUP` rechargera les fichiers de configuration du serveur. Il est aussi possible d'envoyer `SIGHUP` à un processus serveur individuel mais ce n'est pas perceptible.

Pour annuler une requête en cours d'exécution, envoyez le signal `SIGINT` au processus exécutant cette commande. Pour tuer un processus serveur de façon propre, envoyez le signal `SIGTERM` à ce processus. Voir aussi `pg_cancel_backend` et `pg_terminate_backend` dans Section 9.27.2 pour leur équivalents appelables avec une requête SQL.

Le serveur `postgres` utilise aussi `SIGQUIT` pour dire à ses processus-fils de terminer sans le nettoyage habituel. Ce signal *ne doit pas* être envoyé par les utilisateurs. Il est aussi déconseillé d'envoyer `SIGKILL` à un processus serveur -- le serveur `postgres` principal interprétera ceci comme un arrêt brutal et forcera tous les autres processus serveur à quitter dans le cas d'une procédure standard de récupération après arrêt brutal.

Bogues

Les options `--` ne fonctionneront pas sous FreeBSD et OpenBSD. Utilisez `-c` à la place. C'est un bogue dans les systèmes d'exploitation affectés ; une prochaine version de PostgreSQL fournira un contournement si ce n'est pas corrigé.

Mode simple utilisateur

Pour démarrer un serveur en mode mono-utilisateur, utilisez une commande comme

```
postgres --single -D /usr/local/pgsql/data autres-options ma_base
```

Fournissez le bon chemin vers le répertoire des bases avec l'option `-D` ou assurez-vous que la variable d'environnement `PGDATA` est configurée. De plus, spécifiez le nom de la base particulière avec laquelle vous souhaitez travailler.

Habituellement, le serveur en mode mono-utilisateur traite le retour chariot comme le terminateur d'une saisie ; il n'y a pas le concept du point-virgule contraitement à psql. Pour saisir une commande sur plusieurs lignes, vous devez saisir un antislash juste avant un retour chariot, sauf pour le dernier. L'antislash et le retour à la ligne qui suit sont supprimés de la saisie en entrée. Notez que ceci survient même à l'intérieur d'un chaîne litérale ou d'un commentaire.

Si vous utilisez l'option en ligne de commande `-j`, un seul retour à la ligne ne suffira pas à terminer la saisie. Dans ce cas, il faut utiliser la séquence point-virgule - retour à la ligne - retour à la ligne. Autrement dit, saisir un point-virgule suivi d'une ligne entièrement vide. La séquence antislash - retour à la ligne n'est pas traitée spécialement dans ce mode. Encore une fois, il n'y a aucune intelligence sur une séquence apparaissant à l'intérieur d'une chaîne litérale ou d'un commentaire.

Dans les modes de saisie, si vous saisissez un point-virgule qui ne se trouve ni avant ni partie prenant d'une fin de saisie, il est considéré comme un séparateur de commande. Quand vous saisissez une fin de commande, les différentes requêtes saisies seront exécutées dans la même transaction.

Pour quitter la session, saisissez EOF (habituellement, **Control**+**D**). Si vous avez saisi du texte depuis la fin de la commande précédente, alors EOF sera pris comme une fin de commande et un autre EOF sera nécessaire pour quitter.

Notez que le serveur en mode mono-utilisateur ne fournit pas de fonctionnalités avancées sur l'édition de lignes (par exemple, pas d'historique des commandes). De plus, le mode mono-utilisateur ne lance pas de processus en tâche de fond, comme par exemple les checkpoints automatiques ou la réplication.

Exemples

Pour lancer `postgres` en tâche de fond avec les valeurs par défaut, saisissez :

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

Pour lancer `postgres` avec un port spécifique, e.g. 1234 :

```
$ postgres -p 1234
```

Pour se connecter à ce serveur avec psql, indiquez le numéro de port avec l'option -p :

```
$ psql -p 1234
```

ou de configurer la variable d'environnement `PGPORT` :

```
$ export PGPORT=1234
$ psql
```

Les paramètres nommés peuvent être configurés suivant deux façons :

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

Ces deux formes surchargent le paramétrage qui pourrait exister pour `work_mem` dans `postgresql.conf`. Notez que les tirets bas dans les noms de paramètres sont écrits avec soir des

tirets bas soit des tirets sur la ligne de commande. Sauf pour les expériences à court terme, il est probablement mieux de modifier le paramétrage dans `postgresql.conf` que de se baser sur une option en ligne de commande.

Voir aussi

initdb, pg_ctl

postmaster

postmaster — Serveur de bases de données PostgreSQL

Synopsis

`postmaster` [*`option`*...]

Description

`postmaster` est un alias obsolète de `postgres`.

Voir aussi

postgres

Partie VII. Internes

Cette partie contient diverses informations utiles aux développeurs.

Table des matières

52. Présentation des mécanismes internes de PostgreSQL 2353
52.1. Chemin d'une requête 2353
52.2. Établissement des connexions 2353
52.3. Étape d'analyse 2354
52.3.1. Analyseur 2354
52.3.2. Processus de transformation 2355
52.4. Système de règles de PostgreSQL 2355
52.5. Planificateur/Optimiseur 2355
52.5.1. Engendrer les plans possibles 2356
52.6. Exécuteur 2357
53. Catalogues système 2359
53.1. Aperçu 2359
53.2. pg_aggregate 2361
53.3. pg_am 2362
53.4. pg_amop 2363
53.5. pg_amproc 2364
53.6. pg_attrdef 2364
53.7. pg_attribute 2365
53.8. pg_authid 2367
53.9. pg_auth_members 2368
53.10. pg_cast 2369
53.11. pg_class 2369
53.12. pg_collation 2372
53.13. pg_constraint 2373
53.14. pg_conversion 2375
53.15. pg_database 2375
53.16. pg_db_role_setting 2377
53.17. pg_default_acl 2377
53.18. pg_depend 2378
53.19. pg_description 2380
53.20. pg_enum 2380
53.21. pg_event_trigger 2381
53.22. pg_extension 2381
53.23. pg_foreign_data_wrapper 2382
53.24. pg_foreign_server 2383
53.25. pg_foreign_table 2383
53.26. pg_index 2384
53.27. pg_inherits 2385
53.28. pg_init_privs 2386
53.29. pg_language 2386
53.30. pg_largeobject 2387
53.31. pg_largeobject_metadata 2388
53.32. pg_namespace 2388
53.33. pg_opclass 2388
53.34. pg_operator 2389
53.35. pg_opfamily 2390
53.36. pg_parameter_acl 2390
53.37. pg_partitioned_table 2391
53.38. pg_policy 2391
53.39. pg_proc 2392
53.40. pg_publication 2395
53.41. pg_publication_namespace 2395
53.42. pg_publication_rel 2396
53.43. pg_range 2396
53.44. pg_replication_origin 2397

53.45. pg_rewrite 2397
53.46. pg_seclabel 2398
53.47. pg_sequence 2398
53.48. pg_shdepend 2399
53.49. pg_shdescription 2400
53.50. pg_shseclabel 2400
53.51. pg_statistic 2401
53.52. pg_statistic_ext 2402
53.53. pg_statistic_ext_data 2403
53.54. pg_subscription 2404
53.55. pg_subscription_rel 2405
53.56. pg_tablespace 2405
53.57. pg_transform 2406
53.58. pg_trigger 2406
53.59. pg_ts_config 2408
53.60. pg_ts_config_map 2408
53.61. pg_ts_dict 2409
53.62. pg_ts_parser 2409
53.63. pg_ts_template 2410
53.64. pg_type 2410
53.65. pg_user_mapping 2414
54. Vues système 2415
54.1. Aperçu 2415
54.2. pg_available_extensions 2416
54.3. pg_available_extension_versions 2416
54.4. pg_backend_memory_contexts 2417
54.5. pg_config 2418
54.6. pg_cursors 2418
54.7. pg_file_settings 2419
54.8. pg_group 2420
54.9. pg_hba_file_rules 2420
54.10. pg_ident_file_mappings 2421
54.11. pg_indexes 2422
54.12. pg_locks 2422
54.13. pg_matviews 2425
54.14. pg_policies 2425
54.15. pg_prepared_statements 2426
54.16. pg_prepared_xacts 2426
54.17. pg_publication_tables 2427
54.18. pg_replication_origin_status 2427
54.19. pg_replication_slots 2428
54.20. pg_roles 2429
54.21. pg_rules 2430
54.22. pg_seclabels 2431
54.23. pg_sequences 2431
54.24. pg_settings 2432
54.25. pg_shadow 2434
54.26. pg_shmem_allocations 2435
54.27. pg_stats 2436
54.28. pg_stats_ext 2437
54.29. pg_stats_ext_exprs 2438
54.30. pg_tables 2440
54.31. pg_timezone_abbrevs 2440
54.32. pg_timezone_names 2441
54.33. pg_user 2441
54.34. pg_user_mappings 2442
54.35. pg_views 2442
55. Protocole Frontend/Backend 2444

55.1. Aperçu 2444
55.1.1. Aperçu des messages 2445
55.1.2. Aperçu des requêtes étendues 2445
55.1.3. Formats et codes de format 2445
55.2. Flot de messages 2446
55.2.1. Démarrage (Start-up) 2446
55.2.2. Requête simple 2449
55.2.3. Requête étendue 2451
55.2.4. Pipelines 2454
55.2.5. Appel de fonction (*Function Call*) 2455
55.2.6. Opérations COPY 2455
55.2.7. Opérations asynchrones 2457
55.2.8. Annuler des requêtes en cours 2457
55.2.9. Arrêt 2458
55.2.10. Chiffrement SSL de la session 2458
55.2.11. Chiffrement GSSAPI des sessions 2459
55.3. Authentification SASL 2460
55.3.1. Authentification SCRAM-SHA-256 2461
55.4. Protocole de réplication en flux 2462
55.5. Logical Streaming Replication Protocol 2471
55.5.1. Paramètres de la réplication logique en flux 2471
55.5.2. Messages du protocole de réplication logique 2472
55.5.3. Flot des messages du protocole de réplication logique 2472
55.6. Types de données des messages 2473
55.7. Formats de message 2473
55.8. Champs des messages d'erreur et de notification 2491
55.9. Formats des messages de la réplication logique 2493
55.10. Résumé des changements depuis le protocole 2.0 2501
56. Conventions de codage pour PostgreSQL 2503
56.1. Formatage 2503
56.2. Reporter les erreurs dans le serveur 2504
56.3. Guide de style des messages d'erreur 2507
56.4. Conventions diverses de codage 2511
57. Support natif des langues 2514
57.1. Pour le traducteur 2514
57.1.1. Prérequis 2514
57.1.2. Concepts 2514
57.1.3. Créer et maintenir des catalogues de messages 2515
57.1.4. Éditer les fichiers PO 2516
57.2. Pour le développeur 2517
57.2.1. Mécaniques 2517
57.2.2. Guide d'écriture des messages 2518
58. Écrire un gestionnaire de langage procédural 2520
59. Écrire un wrapper de données distantes 2523
59.1. Fonctions d'un wrapper de données distantes 2523
59.2. Routines callback des wrappers de données distantes 2524
59.2.1. Routines des FDW pour parcourir les tables distantes 2524
59.2.2. Routines FDW pour optimiser le traitement après parcours/jointure 2525
59.2.3. Routines des FDW pour le parcours des jointures distantes 2527
59.2.4. Routines FDW pour la mise à jour des tables distantes 2528
59.2.5. Routines FDW pour `TRUNCATE` 2535
59.2.6. Routines FDW pour le verrouillage des lignes 2535
59.2.7. Routines FDW pour `EXPLAIN` 2537
59.2.8. Routines FDW pour `ANALYZE` 2537
59.2.9. Routines FDW pour `IMPORT FOREIGN SCHEMA` 2538
59.2.10. Routines FDW pour une exécution parallélisée 2539
59.2.11. Routines FDW pour les exécutions asynchrones 2540
59.2.12. Routines FDW pour la reparamétrisation des chemins 2541

59.3. Fonctions d'aide pour les wrapper de données distantes 2541
59.4. Planification de la requête avec un wrapper de données distantes 2543
59.5. Le verrouillage de ligne dans les wrappers de données distantes 2545
60. Écrire une méthode d'échantillonnage de table 2548
60.1. Fonctions de support d'une méthode d'échantillonnage 2549
61. Écrire un module de parcours personnalisé 2552
61.1. Créer des parcours de chemin personnalisés 2552
61.1.1. Fonctions callbacks d'un parcours de chemin personnalisé 2553
61.2. Créer des parcours de plans personnalisés 2554
61.2.1. Fonctions callbacks d'un plan de parcours personnalisé 2554
61.3. Exécution de parcours personnalisés 2555
61.3.1. Fonction callbacks d'exécution d'un parcours personnalisé 2555
62. Optimiseur génétique de requêtes (*Genetic Query Optimizer*) 2558
62.1. Gérer les requêtes, un problème d'optimisation complexe 2558
62.2. Algorithmes génétiques 2558
62.3. Optimisation génétique des requêtes (GEQO) dans PostgreSQL 2559
62.3.1. Génération par le GEQO des plans envisageables 2560
62.3.2. Tâches à réaliser pour la future implantation du GEQO 2560
62.4. Lectures supplémentaires 2561
63. Définition de l'interface des méthodes d'accès aux tables 2562
64. Définition de l'interface des méthodes d'accès aux index 2564
64.1. Structure basique de l'API pour les index 2564
64.2. Fonctions des méthodes d'accès aux index 2567
64.3. Parcours d'index 2574
64.4. Considérations sur le verrouillage d'index 2575
64.5. Vérification de l'unicité par les index 2576
64.6. Fonctions d'estimation des coûts d'index 2578
65. Enregistrements génériques des journaux de transactions 2581
66. Gestionnaires de ressources WAL personnalisées 2583
67. Index B-Tree 2585
67.1. Introduction 2585
67.2. Comportement des classes d'opérateur B-Tree 2585
67.3. Fonctions de support B-Tree 2586
67.4. Implémentation 2589
67.4.1. Structure B-Tree 2589
67.4.2. Suppression ascendante d'index 2590
67.4.3. Dédoublement 2591
68. Index GiST 2593
68.1. Introduction 2593
68.2. Classes d'opérateur internes 2593
68.3. Extensibilité 2596
68.4. Implémentation 2609
68.4.1. Méthodes de construction d'index GiST 2609
68.5. Exemples 2610
69. Index SP-GiST 2611
69.1. Introduction 2611
69.2. Classes d'opérateur internes 2611
69.3. Extensibilité 2613
69.4. Implémentation 2623
69.4.1. Limites de SP-GiST 2623
69.4.2. SP-GiST sans label de nœud 2624
69.4.3. Lignes intermédiaires « All-the-same » 2624
69.5. Exemples 2625
70. Index GIN 2626
70.1. Introduction 2626
70.2. Classes d'opérateur natives 2626
70.3. Extensibilité 2627
70.4. Implantation 2630

70.4.1. Technique GIN de mise à jour rapide 2630
70.4.2. Algorithme de mise en correspondance partielle 2631
70.5. Conseils et astuces sur GIN 2631
70.6. Limitations 2632
70.7. Exemples 2632
71. Index BRIN 2633
71.1. Introduction 2633
71.1.1. Maintenance de l'index 2633
71.2. Classes d'opérateur intégrées 2634
71.2.1. Paramètres de classe d'opérateur 2641
71.3. Extensibilité 2642
72. Index Hash 2647
72.1. Aperçu 2647
72.2. Implémentation 2648
73. Stockage physique de la base de données 2649
73.1. Emplacement des fichiers de la base de données 2649
73.2. TOAST 2651
73.2.1. Stockage TOAST sur disque 2652
73.2.2. Stockage TOAST en mémoire, hors-ligne 2653
73.3. Carte des espaces libres 2654
73.4. Carte de visibilité 2655
73.5. Fichier d'initialisation 2655
73.6. Emplacement des pages de la base de données 2655
73.6.1. Disposition d'une ligne de table 2657
73.7. Heap-Only Tuples (HOT) 2658
74. Déclaration du catalogue système et contenu initial 2660
74.1. Règles de déclaration de catalogue système 2660
74.2. Données initiales du catalogue système 2661
74.2.1. Format de fichier de données 2661
74.2.2. Affectation d'OID 2663
74.2.3. Recherche de référence d'OID 2664
74.2.4. Création automatique des types de tableau 2665
74.2.5. Recettes pour éditer les fichiers de données 2665
74.3. Format des fichiers BKI 2667
74.4. Commandes BKI 2667
74.5. Structure du fichier BKI de « bootstrap » 2668
74.6. Exemple BKI 2669
75. Comment le planificateur utilise les statistiques 2670
75.1. Exemples d'estimation des lignes 2670
75.2. Exemples de statistiques multivariées 2676
75.2.1. Dépendances fonctionnelles 2676
75.2.2. Nombre N-Distinct Multivarié 2677
75.2.3. Listes MCV 2678
75.3. Statistiques de l'optimiseur et sécurité 2679
76. Format de manifeste de sauvegarde 2681
76.1. Objet de niveau supérieur du manifeste de sauvegarde 2681
76.2. Objet fichier du manifeste de sauvegarde 2681
76.3. Objet intervalle des journaux de transaction du manifeste de sauvegarde 2682

Chapitre 52. Présentation des mécanismes internes de PostgreSQL

> **Auteur**
>
> Ce chapitre est extrait de [sim98], mémoire de maîtrise (Master's Thesis) de Stefan Simkovics. Cette maîtrise a été préparée à l'université de technologie de Vienne sous la direction du professeur (O.Univ.Prof.Dr.) Georg Gottlob et de l'assistante d'université (Univ.Ass.) Mag. Katrin Seyr.

Ce chapitre présente la structure interne du serveur PostgreSQL. La lecture des sections qui suivent permet de se faire une idée de la façon dont une requête est exécutée. Ce chapitre a, au contraire, pour but d'aider le lecteur à comprendre la suite des opérations effectuées sur le serveur depuis la réception d'une requête jusqu'au retour des résultats.

52.1. Chemin d'une requête

Ceci est un rapide aperçu des étapes franchies par une requête pour obtenir un résultat.

1. Une connexion au serveur est établie par une application. Elle transmet une requête et attend le retour des résultats.

2. L'*étape d'analyse* (*parser stage*) vérifie la syntaxe de la requête et crée un *arbre de requête* (*query tree*).

3. Le *système de réécriture* (*rewrite system*) recherche les *règles* (stockées dans les *catalogues système*) à appliquer à l'arbre de requête. Il exécute les transformations indiquées dans le *corps des règles* (*rule bodies*).

   La réalisation des *vues* est une application du système de réécriture. Toute requête utilisateur impliquant une vue (c'est-à-dire une *table virtuelle*), est réécrite en une requête qui accède aux *tables de base*, en fonction de la *définition de la vue*.

4. Le *planificateur/optimiseur* (*planner/optimizer*) transforme l'arbre de requête (réécrit) en un *plan de requête* (*query plan*) passé en entrée de l'*exécuteur*.

   Il crée tout d'abord tous les *chemins* possibles conduisant aux résultats. Ainsi, s'il existe un index sur une relation à parcourir, il existe deux chemins pour le parcours. Le premier consiste en un simple parcours séquentiel, le second utilise l'index. Le coût d'exécution de chaque chemin est estimé ; le chemin le moins coûteux est alors choisi. Ce dernier est étendu en un plan complet que l'exécuteur peut utiliser.

5. L'exécuteur traverse récursivement les étapes de l'*arbre de planification* (*plan tree*) et retrouve les lignes en fonction de ce plan. L'exécuteur utilise le *système de stockage* lors du parcours des relations, exécute les *tris* et *jointures*, évalue les *qualifications* et retourne finalement les lignes concernées.

Les sections suivantes présentent en détail les éléments brièvement décrits ci-dessus.

52.2. Établissement des connexions

PostgreSQL implémente un modèle client/serveur avec « un processus par utilisateur ». Dans ce modèle, chaque *processus client* se connecte à exactement un *processus moteur*. Comme il n'est

pas possible de savoir à l'avance combien de connexions seront créées, il est nécessaire d'utiliser un « processus superviseur » qui génère un nouveau processus moteur à chaque fois qu'une connexion est demandée. Ce processus superviseur est appelé *postmaster* et écoute sur un port TCP/IP spécifique les connexions entrantes. Dès qu'il détecte une demande de connexion, il génère un nouveau processus moteur. Ces processus moteur communiquent entre eux et avec d'autres processus de l'*instance* en utilisant des *sémaphores* et la *mémoire partagée* pour s'assurer de l'intégrité des données tout au long des accès concurrents aux données.

Le processus client est constitué de tout programme comprenant le protocole PostgreSQL décrit dans le Chapitre 55. De nombreux clients s'appuient sur la bibliothèque C libpq, mais il existe différentes implantations indépendantes du protocole, tel que le pilote Java JDBC.

Une fois que la connexion est établie, le processus client envoie une requête au processus moteur auquel il est connecté. La requête est transmise en plein texte, c'est-à-dire qu'elle n'est pas analysée par le client. Le moteur effectue l'analyse de la requête, crée un *plan d'éxécution*, exécute le plan et retourne les enregistrements récupérés au client en les transmettant par la connexion établie.

52.3. Étape d'analyse

L'*étape d'analyse* est constituée de deux parties :

- l'*analyseur*, défini dans `gram.y` et `scan.l`, est construit en utilisant les outils Unix bison et flex ;
- le *processus de transformation* fait des modifications et des ajouts aux structures de données renvoyées par l'analyseur.

52.3.1. Analyseur

L'analyseur doit vérifier que la syntaxe de la chaîne de la requête (arrivant comme un texte) est valide. Si la syntaxe est correcte, un *arbre d'analyse* est construit et renvoyé, sinon une erreur est retournée. Les analyseur et vérificateur syntaxiques sont développés à l'aide des outils Unix bien connus bison et flex.

L'*analyseur lexical*, défini dans le fichier `scan.l`, est responsable de la reconnaissance des *identificateurs*, des *mots clés SQL*, etc. Pour chaque mot clé ou identificateur trouvé, un *jeton* est engendré et renvoyé à l'analyseur.

L'analyseur est défini dans le fichier `gram.y` et consiste en un ensemble de *règles de grammaire* et en des *actions* à exécuter lorsqu'une règle est découverte. Le code des actions (qui est en langage C) est utilisé pour construire l'arbre d'analyse.

Le fichier `scan.l` est transformé en fichier source C `scan.c` en utilisant le programme flex et `gram.y` est transformé en `gram.c` en utilisant bison. Après avoir réalisé ces transformations, un compilateur C normal peut être utilisé pour créer l'analyseur. Il est inutile de modifier les fichiers C engendrés car ils sont écrasés à l'appel suivant de flex ou bison.

> **Note**
>
> Les transformations et compilations mentionnées sont normalement réalisées automatiquement en utilisant les *makefile* distribués avec les sources de PostgreSQL.

La description détaillée de bison ou des règles de grammaire données dans `gram.y` dépasse le cadre de ce manuel. Il existe de nombreux livres et documentations en relation avec flex et bison. Il est préférable d'être familier avec bison avant de commencer à étudier la grammaire donnée dans `gram.y`, au risque de ne rien y comprendre.

52.3.2. Processus de transformation

L'étape d'analyse crée un arbre d'analyse qui n'utilise que les règles fixes de la structure syntaxique de SQL. Il ne fait aucune recherche dans les catalogues système. Il n'y a donc aucune possibilité de comprendre la sémantique détaillée des opérations demandées. Lorsque l'analyseur a fini, le *processus de transformation* prend en entrée l'arbre résultant de l'analyseur et réalise l'interprétation sémantique nécessaire pour connaître les tables, fonctions et opérateurs référencés par la requête. La structure de données construite pour représenter cette information est appelée l'*arbre de requête*.

La séparation de l'analyse brute et de l'analyse sémantique résulte du fait que les recherches des catalogues système ne peuvent se dérouler qu'à l'intérieur d'une transaction. Or, il n'est pas nécessaire de commencer une transaction dès la réception d'une requête. L'analyse brute est suffisante pour identifier les commandes de contrôle des transactions (`BEGIN`, `ROLLBACK`, etc.). Elles peuvent de plus être correctement exécutées sans analyse complémentaire. Lorsqu'il est établi qu'une vraie requête doit être gérée (telle que `SELECT` ou `UPDATE`), une nouvelle transaction est démarrée si aucune n'est déjà en cours. Ce n'est qu'à ce moment-là que le processus de transformation peut être invoqué.

La plupart du temps, l'arbre d'une requête créé par le processus de transformation a une structure similaire à l'arbre d'analyse brute mais, dans le détail, de nombreuses différences existent. Par exemple, un nœud `FuncCall` dans l'arbre d'analyse représente quelque chose qui ressemble syntaxiquement à l'appel d'une fonction. Il peut être transformé soit en nœud `FuncExpr`, soit en nœud `Aggref`, selon que le nom référencé est une fonction ordinaire ou une fonction d'agrégat. De même, des informations sur les types de données réels des colonnes et des résultats sont ajoutées à l'arbre de la requête.

52.4. Système de règles de PostgreSQL

PostgreSQL supporte un puissant *système de règles* pour la spécification des *vues* et des *mises à jour de vues* ambiguës. À l'origine, le système de règles de PostgreSQL était constitué de deux implantations :

- la première, qui fonctionnait au *niveau des lignes*, était implémentée profondément dans l'*exécuteur*. Le système de règles était appelé à chaque fois qu'il fallait accéder une ligne individuelle. Cette implantation a été supprimée en 1995 quand la dernière version officielle du projet Berkeley Postgres a été transformée en Postgres95 ;

- la deuxième implantation du système de règles est une technique appelée *réécriture de requêtes*. Le *système de réécriture* est un module qui existe entre l'*étape d'analyse* et le *planificateur/optimiseur*. Cette technique est toujours implémentée.

Le système de réécriture de requêtes est vu plus en détails dans le Chapitre 41. Il n'est donc pas nécessaire d'en parler ici. Il convient simplement d'indiquer qu'à la fois l'entrée et la sortie du système sont des arbres de requêtes. C'est-à-dire qu'il n'y a pas de changement dans la représentation ou le niveau de détail sémantique des arbres. La réécriture peut être imaginée comme une forme d'expansion de macro.

52.5. Planificateur/Optimiseur

La tâche du *planificateur/optimiseur* est de créer un plan d'exécution optimal. En fait, une requête SQL donnée (et donc, l'arbre d'une requête) peut être exécutée de plusieurs façons, chacune arrivant au même résultat. Si ce calcul est possible, l'optimiseur de la requête examinera chacun des plans d'exécution possibles pour sélectionner le plan d'exécution estimé comme le plus rapide.

> **Note**
>
> Dans certaines situations, examiner toutes les façons d'exécuter une requête prend beaucoup de temps et de mémoire. En particulier, lors de l'exécution de requêtes impliquant un grand nombre de jointures. Pour déterminer un plan de requête raisonnable (mais pas

forcément optimal) en un temps raisonnable, PostgreSQL utilise un *Genetic Query Optimizer* (voir Chapitre 62) dès lors que le nombre de jointures dépasse une certaine limite (voir geqo_threshold).

La procédure de recherche du planificateur fonctionne avec des structures de données appelés *chemins*, simples représentations minimales de plans ne contenant que l'information nécessaire au planificateur pour prendre ses décisions. Une fois le chemin le moins coûteux déterminé, un *arbre plan* est construit pour être passé à l'exécuteur. Celui-ci représente le plan d'exécution désiré avec suffisamment de détails pour que l'exécuteur puisse le lancer. Dans le reste de cette section, la distinction entre chemins et plans est ignorée.

52.5.1. Engendrer les plans possibles

Le planificateur/optimiseur commence par engendrer des plans de parcours de chaque relation (table) invididuelle utilisée dans la requête. Les plans possibles sont déterminés par les index disponibles pour chaque relation. Un parcours séquentiel de relation étant toujours possible, un plan de parcours séquentiel est systématiquement créé. Soit un index défini sur une relation (par exemple un index B-tree) et une requête qui contient le filtre `relation.attribut OPR constante`. Si `relation.attribut` correspond à la clé de l'index B-tree et `OPR` est un des opérateurs listés dans la *classe d'opérateurs* de l'index, un autre plan est créé en utilisant l'index B-tree pour parcourir la relation. S'il existe d'autres index et que les restrictions de la requête font correspondre une clé à un index, d'autres plans sont considérés. Des plans de parcours d'index sont également créés pour les index dont l'ordre de tri peut correspondre à la clause `ORDER BY` de la requête (s'il y en a une), ou dont l'ordre de tri peut être utilisé dans une jointure de fusion (cf. plus bas).

Si la requête nécessite de joindre deux relations ou plus, les plans de jointure de ces dernières sont considérés après la découverte de tous les plans possibles de parcours des relations uniques. Les trois stratégies possibles de jointure sont :

- *jointure de boucle imbriquée* (*nested loop join*) : la relation de droite est parcourue une fois pour chaque ligne trouvée dans la relation de gauche. Cette stratégie est facile à implanter mais peut être très coûteuse en temps. (Toutefois, si la relation de droite peut être parcourue à l'aide d'un index, ceci peut être une bonne stratégie. Il est possible d'utiliser les valeurs issues de la ligne courante de la relation de gauche comme clés du parcours d'index à droite.) ;

- *jointure de fusion* (*merge join*) : chaque relation est triée selon les attributs de la jointure avant que la jointure ne commence. Puis, les deux relations sont parcourues en parallèle et les lignes correspondantes sont combinées pour former des lignes jointes. Ce type de jointure est intéressant parce que chaque relation n'est parcourue qu'une seule fois. Le tri requis peut être réalisé soit par une étape explicite de tri soit en parcourant la relation dans le bon ordre en utilisant un index sur la clé de la jointure ;

- *jointure de hachage* (*hash join*) : la relation de droite est tout d'abord parcourue et chargée dans une table de hachage en utilisant ses attributs de jointure comme clés de hachage. La relation de gauche est ensuite parcourue et les valeurs appropriées de chaque ligne trouvée sont utilisées comme clés de hachage pour localiser les lignes correspondantes dans la table.

Quand la requête implique plus de deux relations, le résultat final doit être construit à l'aide d'un arbre d'étapes de jointure, chacune à deux entrées. Le planificateur examine les séquences de jointure possibles pour trouver le moins cher.

Si la requête implique moins de geqo_threshold relations, une recherche quasi-exhaustive est effectuée pour trouver la meilleure séquence de jointure. Le planificateur considère préférentiellement les jointures entre paires de relations pour lesquelles il existe une clause de jointure correspondante dans la qualification `WHERE` (i.e. pour lesquelles il existe une restriction de la forme `where rel1.attr1=rel2.attr2`). Les paires jointes pour lesquelles il n'existe pas de clause de jointure ne sont considérées que lorsqu'il n'y a plus d'autre choix, c'est-à-dire qu'une relation particulière n'a pas de clause de jointure avec une autre relation. Tous les plans possibles sont créés pour chaque

paire jointe considérée par le planificateur. C'est alors celle qui est (estimée) la moins coûteuse qui est choisie.

Lorsque `geqo_threshold` est dépassé, les séquences de jointure sont déterminées par heuristique, comme cela est décrit dans Chapitre 62. Pour le reste, le processus est le même.

L'arbre de plan terminé est composé de parcours séquentiels ou d'index des relations de base, auxquels s'ajoutent les nœuds des jointures en boucle, des jointures de tri fusionné et des jointures de hachage si nécessaire, ainsi que toutes les étapes auxiliaires nécessaires, telles que les nœuds de tri ou les nœuds de calcul des fonctions d'agrégat. La plupart des types de nœud de plan ont la capacité supplémentaire de faire une *sélection* (rejet des lignes qui ne correspondent pas à une condition booléenne indiquée) et une *projection* (calcul d'un ensemble dérivé de colonnes fondé sur des valeurs de colonnes données, par l'évaluation d'expressions scalaires si nécessaire). Une des responsabilités du planificateur est d'attacher les conditions de sélection issues de la clause `WHERE` et le calcul des expressions de sortie requises aux nœuds les plus appropriés de l'arbre de plan.

52.6. Exécuteur

L'*exécuteur* prend le plan créé par le planificateur/optimiseur et l'exécute récursivement pour extraire l'ensemble requis de lignes. Il s'agit principalement d'un mécanisme de pipeline en demande-envoi. Chaque fois qu'un nœud du plan est appelé, il doit apporter une ligne supplémentaire ou indiquer qu'il a fini d'envoyer des lignes.

Pour donner un exemple concret, on peut supposer que le nœud supérieur est un nœud `MergeJoin`. Avant de pouvoir faire une fusion, deux lignes doivent être récupérées (une pour chaque sous-plan). L'exécuteur s'appelle donc récursivement pour exécuter les sous-plans (en commençant par le sous-plan attaché à l'`arbre gauche`). Le nouveau nœud supérieur (le nœud supérieur du sous-plan gauche) est, par exemple, un nœud `Sort` (NdT : Tri) et un appel récursif est une nouvelle fois nécessaire pour obtenir une ligne en entrée. Le nœud fils de `Sort` pourrait être un nœud `SeqScan`, représentant la lecture réelle d'une table. L'exécution de ce nœud impose à l'exécuteur de récupérer une ligne à partir de la table et de la retourner au nœud appelant. Le nœud `Sort` appelle de façon répétée son fils pour obtenir toutes les lignes à trier. Quand l'entrée est épuisée (indiqué par le nœud fils renvoyant un NULL au lieu d'une ligne), le code de `Sort` est enfin capable de renvoyer sa première ligne en sortie, c'est-à-dire le premier suivant l'ordre de tri. Il conserve les lignes restantes en mémoire de façon à les renvoyer dans le bon ordre en réponse à des demandes ultérieures.

Le nœud `MergeJoin` demande de la même façon la première ligne à partir du sous-plan droit. Ensuite, il compare les deux lignes pour savoir si elles peuvent être jointes ; si c'est le cas, il renvoie la ligne de jointure à son appelant. Au prochain appel, ou immédiatement, s'il ne peut pas joindre la paire actuelle d'entrées, il avance sur la prochaine ligne d'une des deux tables (suivant le résultat de la comparaison), et vérifie à nouveau la correspondance. Éventuellement, un des sous-plans est épuisé et le nœud `MergeJoin` renvoie NULL pour indiquer qu'il n'y a plus de lignes jointes à former.

Les requêtes complexes peuvent nécessiter un grand nombre de niveaux de nœuds pour les plans, mais l'approche générale est la même : chaque nœud est exécuté et renvoie sa prochaine ligne en sortie à chaque fois qu'il est appelé. Chaque nœud est responsable aussi de l'application de toute expression de sélection ou de projection qui lui a été confiée par le planificateur.

Le mécanisme de l'exécuteur est utilisé pour évaluer les cinq types de requêtes de base en SQL : `SELECT`, `INSERT`, `UPDATE`, `DELETE` et `MERGE`. Pour `SELECT`, le code de l'exécuteur de plus haut niveau a uniquement besoin d'envoyer chaque ligne retournée par l'arbre plan de la requête vers le client. `INSERT ... SELECT`, `UPDATE`, `DELETE` et `MERGE` sont en réalité des `SELECT` sous un nœud de plan haut niveau appelé `ModifyTable`.

`INSERT ... SELECT` remplit les lignes de `ModifyTable` pour insertion. Pour `UPDATE`, l'optimiseur s'arrange pour que chaque ligne traitée inclut toutes les valeurs mises à jour des colonnes, plus le *TID* (*tuple ID*, ou identifiant de ligne) de la ligne cible originale ; cette donnée est envoyée au nœud `ModifyTable`, qui utilise l'information pour créer un nouveau nœud mis à jour et pour marquer l'ancienne ligne comme supprimée. Pour `DELETE`, la seule colonne qui est réellement renvoyée par le

plan est le TID, et le nœud `ModifyTable` utilise simplement le TID pour visiter chaque ligne cible et la marquer supprimée. Pour `MERGE`, l'optimiseur joint les relations source et cible, et inclut toutes les valeurs de colonne requises par chaque clause `WHEN`, plus le TID de la ligne cible ; cette donnée est fournie au nœud `ModifyTable` qui utilise l'information pour trouver la clause `WHEN` à exécuter, puis insère, met à jour ou supprime la ligne cible, suivant le besoin exprimé.

Une simple commande `INSERT ... VALUES` crée un arbre de plan trivial consistant en un seul nœud `Result`, qui calcule seulement une ligne résultat, en l'envoyant à `ModifyTable` pour réaliser l'insertion.

Chapitre 53. Catalogues système

Les catalogues système représentent l'endroit où une base de données relationnelle stocke les métadonnées des schémas, telles que les informations sur les tables et les colonnes, et des données de suivi interne. Les catalogues système de PostgreSQL sont de simples tables. Elles peuvent être supprimées et recrées. Il est possible de leur ajouter des colonnes, d'y insérer et modifier des valeurs, et de mettre un joyeux bazar dans le système. En temps normal, l'utilisateur n'a aucune raison de modifier les catalogues système, il y a toujours des commandes SQL pour le faire. (Par exemple, `CREATE DATABASE` insère une ligne dans le catalogue `pg_database` -- et crée physiquement la base de données sur le disque.) Il y a des exceptions pour certaines opérations particulièrement ésotériques, mais au fil du temps la plupart de ces opérations ont été mises à disposition sous la forme de commandes SQL. De ce fait, le besoin de modifier directement les catalogues systèmes est de plus en plus faible.

53.1. Aperçu

Tableau 53.1 liste les catalogues système. Une documentation plus détaillée des catalogues système suit.

La plupart des catalogues système sont recopiés de la base de données modèle lors de la création de la base de données et deviennent alors spécifiques à chaque base de données. Un petit nombre de catalogues sont physiquement partagés par toutes les bases de données d'une installation de PostgreSQL. Ils sont indiqués dans les descriptions des catalogues.

Tableau 53.1. Catalogues système

Nom du catalogue	Contenu
`pg_aggregate`	fonctions d'agrégat
`pg_am`	méthodes d'accès aux relations
`pg_amop`	opérateurs des méthodes d'accès
`pg_amproc`	fonctions de support des méthodes d'accès
`pg_attrdef`	valeurs par défaut des colonnes
`pg_attribute`	colonnes des tables (« attributs »)
`pg_authid`	identifiants d'autorisation (rôles)
`pg_auth_members`	relations d'appartenance aux identifiants d'autorisation
`pg_cast`	conversions de types de données (*cast*)
`pg_class`	tables, index, séquences, vues (« relations »)
`pg_collation`	collationnement (information locale)
`pg_constraint`	contraintes de vérification, contraintes uniques, contraintes de clés primaires, contraintes de clés étrangères
`pg_conversion`	informations de conversions de codage
`pg_database`	bases de données du cluster PostgreSQL
`pg_db_role_setting`	configuration par rôle et par base de données
`pg_default_acl`	droits par défaut sur des types d'objets
`pg_depend`	dépendances entre objets de la base de données
`pg_description`	descriptions ou commentaires des objets de base de données
`pg_enum`	définitions des labels et des valeurs des enum
`pg_event_trigger`	triggers sur événement
`pg_extension`	extensions installées

Nom du catalogue	Contenu
pg_foreign_data_wrapper	définitions des wrappers de données distantes
pg_foreign_server	définitions des serveurs distants
pg_foreign_table	informations supplémentaires sur les tables distantes
pg_index	informations supplémentaires des index
pg_inherits	hiérarchie d'héritage de tables
pg_init_privs	droits initiaux des objets
pg_language	langages d'écriture de fonctions
pg_largeobject	pages de données pour les « Large Objects »
pg_largeobject_metadata	métadonnées pour les « Large Objects »
pg_namespace	schémas
pg_opclass	classes d'opérateurs de méthodes d'accès
pg_operator	opérateurs
pg_opfamily	familles d'opérateurs de méthodes d'accès
pg_parameter_acl	paramètres de configuration pour lesquels des droits ont été donnés
pg_partitioned_table	clés de partitionnement des tables
pg_policy	politiques de sécurité niveau ligne
pg_proc	fonctions et procédures
pg_publication	publications pour la réplication logique
pg_publication_namespace	correspondance schéma publication
pg_publication_rel	correspondance relation-publication
pg_range	informations sur les types d'intervalles de données
pg_replication_origin	origines de réplication enregistrées
pg_rewrite	règles de réécriture de requêtes
pg_seclabel	labels de sécurité sur les objets d'une base de données
pg_sequence	séquences
pg_shdepend	dépendances sur les objets partagés
pg_shdescription	commentaires sur les objets partagés
pg_shseclabel	labels de sécurité sur des objets partagés
pg_statistic	statistiques de l'optimiseur de requêtes
pg_statistic_ext	statistiques étendues de l'optimiseur de requêtes (définitions)
pg_statistic_ext_data	statistiques étendues de l'optimiseur de requêtes (statistiques)
pg_subscription	souscriptions pour la réplication logique
pg_subscription_rel	état des relations pour les souscriptions
pg_tablespace	*tablespaces* du cluster de bases de données
pg_transform	transformations (conversions de types de données vers les langages procéduraux)
pg_trigger	triggers
pg_ts_config	configuration de la recherche plein texte
pg_ts_config_map	configuration de la recherche plein texte pour la correspondance des lexèmes (*token*)
pg_ts_dict	dictionnaires de la recherche plein texte

Nom du catalogue	Contenu
pg_ts_parser	analyseurs de la recherche plein texte
pg_ts_template	modèles de la recherche plein texte
pg_type	types de données
pg_user_mapping	correspondance d'utilisateurs sur des serveurs distants

53.2. pg_aggregate

Le catalogue pg_aggregate stocke les informations concernant les fonctions d'agrégat. Une fonction d'agrégat est une fonction qui opère sur un ensemble de données (typiquement une colonne de chaque ligne qui correspond à une condition de requête) et retourne une valeur unique calculée à partir de toutes ces valeurs. Les fonctions d'agrégat classiques sont sum (somme), count (compteur) et max (plus grande valeur). Chaque entrée de pg_aggregate est une extension d'une entrée de pg_proc. L'entrée de pg_proc contient le nom de l'agrégat, les types de données d'entrée et de sortie, et d'autres informations similaires aux fonctions ordinaires.

Tableau 53.2. Les colonnes de pg_aggregate

Type Description
aggfnoid regproc (référence pg_proc.oid) OID pg_proc de la fonction d'agrégat
aggkind char Type d'agrégat : n pour les agrégats « normaux » (standards), o pour les agrégats d'« ensemble trié », ou h pour les agrégats d'« ensembles hypothétiques »
aggnumdirectargs int2 Nombre d'arguments directs (non agrégés) d'un ensemble trié ou d'un ensemble hypothétique, comptant un tableau variadique comme un seul argument. Si cette valeur est égale à pronargs, l'agrégat doit être variadique et le tableau variadique décrit aussi les arguments agrégés ainsi que les arguments directs finaux. Toujours à 0 pour les agrégats standards.
aggtransfn regproc (référence pg_proc.oid) Fonction de transition
aggfinalfn regproc (référence pg_proc.oid) Fonction finale (0 s'il n'y en a pas)
aggcombinefn regproc (référence pg_proc.oid) Fonction combine (zero s'il n'y en a pas)
aggserialfn regproc (référence pg_proc.oid) Fonction de sérialisation (zero s'il n'y en a pas)
aggdeserialfn regproc (référence pg_proc.oid) Fonction de désérialisation (zero s'il n'y en a pas)
aggmtransfn regproc (référence pg_proc.oid) Fonction de transition en avant pour le mode d'agrégat avec déplacement (zéro sinon)
aggminvtransfn regproc (référence pg_proc.oid) Fonction de transition inverse pour le mode d'agrégat avec déplacement (zéro sinon)
aggmfinalfn regproc (référence pg_proc.oid) Fonction finale pour le mode d'agrégat avec déplacement (zéro sinon)
aggfinalextra bool Vrai pour passer des arguments supplémentaires à aggfinalfn
aggmfinalextra bool Vrai pour passer des arguments supplémentaires à aggmfinalfn

Type Description
`aggfinalmodify` `char` Si `aggfinalfn` modifie la valeur d'état de transition : `r` s'il est en lecture seule, `s` si `aggtransfn` ne peut pas être appliqué après `aggfinalfn`, ou `w` s'il écrit sur la valeur
`aggmfinalmodify` `char` Comme `aggfinalmodify`, mais pour `aggmfinalfn`
`aggsortop` `oid` (référence `pg_operator.oid`) Opérateur de tri associé (0 s'il n'y en a pas)
`aggtranstype` `oid` (référence `pg_type.oid`) Type de la donnée interne de transition (état) de la fonction d'agrégat
`aggtransspace` `int4` Taille moyenne approximative (en octets) des données de l'état de transition, ou zéro pour utiliser une estimation par défaut
`aggmtranstype` `oid` (référence `pg_type.oid`) Type de données de la transition interne (état) de la fonction d'agrégat pour le mode d'agrégat avec déplacement (zéro sinon)
`aggmtransspace` `int4` Taille moyenne approximative (en octets) des données d'état de transition pour le mode d'agrégat avec déplacement ou zéro pour utiliser une estimation par défaut
`agginitval` `text` Valeur initiale de la fonction de transition. C'est un champ texte qui contient la valeur initiale dans sa représentation externe en chaîne de caractères. Si ce champ est NULL, la valeur d'état de transition est initialement NULL.
`aggminitval` `text` La valeur initiale de l'état de transition pour le mode d'agrégat avec déplacement. C'est un champ texte contenant la valeur initiale dans sa représentation externe sous forme de chaîne. Si ce champ est NULL, la valeur de l'état de transition commence avec NULL.

Les nouvelles fonctions d'agrégat sont enregistrées avec la commande `CREATE AGGREGATE`. La Section 38.12 fournit de plus amples informations sur l'écriture des fonctions d'agrégat et sur la signification des fonctions de transition.

53.3. `pg_am`

Le catalogue `pg_am` stocke les informations concernant les méthodes d'accès aux relations. On trouve une ligne par méthode d'accès supportée par le système. Actuellement, seuls les tables et les index ont des méthodes d'accès. Ce qui concerne les méthodes d'accès aux tables et aux index est discuté en détails dans, respectivement, Chapitre 63 et Chapitre 64.

Tableau 53.3. Colonnes de `pg_am`

Type Description
`oid` `oid` Identifiant de ligne
`amname` `name` Nom de la méthode d'accès
`amhandler` `regproc` (référence `pg_proc.oid`) OID de la fonction gestionnaire responsable de la fourniture des informations sur la méthode d'accès
`amtype` `char` `t` = table (incluant les vues matérialisées), `i` = index.

> **Note**
>
> Avant PostgreSQL 9.6, `pg_am` contenait plusieurs colonnes supplémentaires représentant des propriétés des méthodes d'accès aux index. Ces données sont maintenant uniquement visibles au niveau du code C. Néanmoins, `pg_index_column_has_property()` et les fonctions relatives ont été ajoutées pour permettre aux requêtes SQL d'inspecter les propriétés des méthodes d'accès aux index. Voir Tableau 9.71.

53.4. pg_amop

Le catalogue `pg_amop` stocke les informations concernant les opérateurs associés aux familles d'opérateurs des méthodes d'accès aux index. Il y a une ligne pour chaque opérateur membre d'une famille. Un membre d'une famille peut être soit un opérateur de *recherche* soit un opérateur de *tri*. Un opérateur peut apparaître dans plus d'une famille, mais ne peut pas apparaître dans plus d'une position à l'intérieur d'une famille.

Tableau 53.4. Colonnes de pg_amop

Type **Description**
`oid` `oid` Identifiant de ligne
`amopfamily` `oid` (référence `pg_opfamily.oid`) La famille d'opérateur
`amoplefttype` `oid` (référence `pg_type.oid`) Type de données en entrée, côté gauche, de l'opérateur
`amoprighttype` `oid` (référence `pg_type.oid`) Type de données en entrée, côté droit, de l'opérateur
`amopstrategy` `int2` Numéro de stratégie d'opérateur
`amoppurpose` `char` But de l'opérateur, soit `s` pour recherche soit `o` pour tri
`amopopr` `oid` (référence `pg_operator.oid`) OID de l'opérateur
`amopmethod` `oid` (référence `pg_am.oid`) Méthode d'accès à l'index pour cette famille d'opérateur
`amopsortfamily` `oid` (référence `pg_opfamily.oid`) La famille d'opérateur B-tree utilisée par cette entrée pour trier s'il s'agit d'un opérateur de tri ; zéro s'il s'agit d'un opérateur de recherche

Un opérateur de « recherche » indique qu'un index de cet opérateur peut être utilisé pour rechercher toutes les lignes satisfaisant une clause `WHERE` *`colonne_indexée opérateur constante`*. Cet opérateur doit évidemment renvoyer un booléen et le type de l'entrée gauche doit correspondre au type de données de la colonne de l'index.

Un opérateur de « tri » indique qu'un index de cette famille d'opérateur peut être parcouru pour renvoyer les lignes dans l'ordre représenté par une clause `ORDER BY` *`colonne_indexée opérateur constante`*. Cet opérateur peut renvoyer tout type de données triable, bien que le type de l'entrée gauche doit correspondre au type de données de la colonne de l'index. La sémantique exacte de la clause `ORDER BY` est spécifié par la colonne `amopsortfamily` qui doit référencer une famille d'opérateur B-tree pour le type de résultat de l'opérateur.

> **Note**
>
> Actuellement, il est supposé que l'ordre de tri pour un opérateur de tri est celui par défaut de la famille d'opérateur référencée, c'est-à-dire `ASC NULLS LAST`. Ceci pourrait changer en ajoutant des colonnes supplémentaires pour y indiquer explicitement les options de tri.

Une entrée dans `amopmethod` doit correspondre au `opfmethod` de sa famille d'opérateur parent (l'inclusion de `amopmethod` à ce niveau est une dénormalisation intentionnelle de la structure du catalogue pour des raisons de performance). De plus, `amoplefttype` et `amoprighttype` doivent correspondre aux champs `oprleft` et `oprright` de l'entrée `pg_operator` référencée.

53.5. `pg_amproc`

Le catalogue `pg_amproc` stocke les informations concernant les fonctions de support associées aux familles d'opérateurs de méthodes d'accès. Il y a une ligne pour chaque fonction de support appartenant à une famille.

Tableau 53.5. Colonnes de `pg_amproc`

Type **Description**
`oid` `oid` Identifiant de ligne
`amprocfamily` `oid` (référence `pg_opfamily.oid`) La famille d'opérateur
`amproclefttype` `oid` (référence `pg_type.oid`) Type de données en entrée, côté gauche, de l'opérateur associé
`amprocrighttype` `oid` (référence `pg_type.oid`) Type de données en entrée, côté droit, de l'opérateur associé
`amprocnum` `int2` Numéro de fonction de support
`amproc` `regproc` (référence `pg_proc.oid`) OID de la fonction

On interprète habituellement les champs `amproclefttype` et `amprocrighttype` comme identifiant les types de données des côtés gauche et droit d'opérateur(s) supporté(s) par une fonction particulière. Pour certaines méthodes d'accès, ils correspondent aux types de données en entrée de la fonction elle-même. Il existe une notion de fonctions de support par « défaut » pour un index, fonctions pour lesquelles `amproclefttype` et `amprocrighttype` sont tous deux équivalents à l'`opcintype` de la classe d'opérateurs de l'index.

53.6. `pg_attrdef`

Le catalogue `pg_attrdef` stocke les valeurs par défaut des colonnes. Les informations principales des colonnes sont stockées dans `pg_attribute`. Seules les colonnes pour lesquelles une valeur par défaut est explicitement indiquée (quand la table est créée ou quand une colonne est ajoutée) ont une entrée dans `pg_attrdef`.

Tableau 53.6. Colonnes de `pg_attrdef`

Type **Description**
`oid` `oid`

Type Description
Identifiant de ligne
`adrelid` `oid` (référence `pg_class.oid`) La table à laquelle appartient la colonne
`adnum` `int2` (référence `pg_attribute.attnum`) Numéro de la colonne
`adbin` `pg_node_tree` La valeur par défaut de la colonne, dans la représentation `nodeToString()`. Utilisez `pg_get_expr(adbin, adrelid)` pour la convertir en une expression SQL.

53.7. `pg_attribute`

Le catalogue `pg_attribute` stocke les informations concernant les colonnes des tables. Il y a exactement une ligne de `pg_attribute` par colonne de table de la base de données. (Il y a aussi des attributs pour les index et, en fait, tous les objets qui possèdent des entrées dans `pg_class`.)

Le terme attribut, équivalent à colonne, est utilisé pour des raisons historiques.

Tableau 53.7. Colonnes de `pg_attribute`

Type Description
`attrelid` `oid` (référence `pg_class.oid`) La table à laquelle appartient la colonne
`attname` `name` Le nom de la colonne
`atttypid` `oid` (référence `pg_type.oid`) Le type de données de la colonne (zéro pour une colonne supprimée)
`attstattarget` `int4` Contrôle le niveau de détail des statistiques accumulées pour la colonne par ANALYZE. Une valeur 0 indique qu'aucune statistique ne doit être collectée. Une valeur négative indique d'utiliser l'objectif de statistiques par défaut. Le sens exact d'une valeur positive dépend du type de données. Pour les données scalaires, `attstattarget` est à la fois le nombre de « valeurs les plus courantes » à collecter et le nombre d'histogrammes à créer.
`attlen` `int2` Une copie de `pg_type.typlen` pour le type de la colonne.
`attnum` `int2` Le numéro de la colonne. La numérotation des colonnes ordinaires démarre à 1. Les colonnes système, comme les `ctid`, ont des numéros négatifs arbitraires.
`attndims` `int4` Nombre de dimensions, si la colonne est de type tableau, sinon 0. (Pour l'instant, le nombre de dimensions des tableaux n'est pas contrôlé, donc une valeur autre que 0 indique que « c'est un tableau ».)
`attcacheoff` `int4` Toujours -1 sur disque, mais peut être mis à jour lorsque la ligne est chargée en mémoire, pour mettre en cache l'emplacement de l'attribut dans la ligne.
`atttypmod` `int4` Stocke des données spécifiques au type de données précisé lors de la création de la table (par exemple, la taille maximale d'une colonne de type `varchar`). Il est transmis aux fonctions spécifiques au type d'entrée de données et de vérification de taille. La valeur est généralement -1 pour les types de données qui n'ont pas besoin de `atttypmod`.
`attbyval` `bool`

Type Description
Une copie de `pg_type.typbyval` du type de la colonne.
`attalign` `char` Une copie de `pg_type.typalign` du type de la colonne.
`attstorage` `char` Contient normalement une copie de `pg_type.typstorage` du type de la colonne. Pour les types de données TOASTables, cette valeur peut être modifiée après la création de la colonne pour en contrôler les règles de stockage.
`attcompression` `char` La méthode de compression actuelle de la colonne. Habituellement, cette colonne vaut `'\0'` pour indiquer l'utilisation de la configuration par défaut(voir default_toast_compression). Sinon, `'p'` sélectionne la compression pglz, alors que `'l'` sélectionne la compression LZ4. Néanmoins, ce champ est ignoré à chaque fois que `attstorage` ne permet pas la compression.
`attnotnull` `bool` Indique une contrainte de non-nullité de colonne. Il est possible de changer cette colonne pour activer ou désactiver cette contrainte.
`atthasdef` `bool` Cette colonne a une expression par défaut ou une expression générée, auquel cas il y aura un enregistrement correspondant dans le catalogue `pg_attrdef` qui définit réellement l'expression. (Vérifiez `attgenerated` pour déterminer si c'est une expression par défaut ou générée.)
`atthasmissing` `bool` Cette colonne a une valeur qui est utilisée quand la colonne est complètement manquante de la ligne, ce qui arrive quand une colonne est ajoutée avec une valeur par défaut et non volatile (contrainte `DEFAULT`) après la création de la ligne. La valeur actuellement utilisée est enregistrée dans la colonne `attmissingval`.
`attidentity` `char` Si vide (`''`), alors ce n'est pas une colonne identité. Sinon, `a` signifie toujours généré alors que `d` signifie généré par défaut.
`attgenerated` `char` Si c'est un octet zéro (`''`), alors ce n'est pas une colonne générée. Sinon, `s` = stockée. (D'autres valeurs pourraient être ajoutées dans le futur.)
`attisdropped` `bool` Indique que la colonne a été supprimée et n'est plus valide. Une colonne supprimée est toujours présente physiquement dans la table, mais elle est ignorée par l'analyseur de requête et ne peut être accédée en SQL.
`attislocal` `bool` La colonne est définie localement dans la relation. Une colonne peut être simultanément définie localement et héritée.
`attinhcount` `int4` Nombre d'ancêtres directs de la colonne. Une colonne qui a au moins un ancêtre ne peut être ni supprimée ni renommée.
`attcollation` `oid` (référence `pg_collation.oid`) Le collationnement défini de la colonne, ou zéro si la colonne n'est pas un type de données collationnable.
`attacl` `aclitem[]` Droits d'accès niveau colonne, s'il y en a qui ont été spécifiquement accordés à cette colonne
`attoptions` `text[]` Options au niveau colonne, en tant que chaînes du type « motclé=valeur »

Type Description
`attfdwoptions` `text[]` Options du wrapper de données distances, au niveau colonne, en tant que chaînes du type « keyword=value »
`attmissingval` `anyarray` Cette colonne a un tableau à un élément contenant la valeur utilisée quand la colonne est complètement manquante de la ligne, comme cela peut survenir quand la colonne est ajoutée avec une valeur par défaut (`DEFAULT`) non volatile après la création de la ligne. La valeur est seulement utilisée quand `atthasmissing` est true. S'il n'y a pas de valeur, la colonne est NULL.

Dans l'entrée `pg_attribute` d'une colonne supprimée, `atttypid` est réinitialisée à 0 mais `attlen` et les autres champs copiés à partir de `pg_type` sont toujours valides. Cet arrangement est nécessaire pour s'adapter à la situation où le type de données de la colonne supprimée a été ensuite supprimé et qu'il n'existe donc plus de ligne `pg_type`. `attlen` et les autres champs peuvent être utilisés pour interpréter le contenu d'une ligne de la table.

53.8. `pg_authid`

Le catalogue `pg_authid` contient les informations concernant les identifiants pour les autorisations d'accès aux bases de données (rôles). Un rôle englobe les concepts d'« utilisateur » et de « groupe ». Un utilisateur est essentiellement un rôle qui a l'attribut de connexion (`rolcanlogin`). Tout rôle (avec ou sans `rolcanlogin`) peut avoir d'autres rôles comme membres ; voir `pg_auth_members`.

Comme ce catalogue contient les mots de passe, il ne doit pas être lisible par tout le monde. `pg_roles` est une vue, lisible par tout le monde, de `pg_authid` qui masque le champ du mot de passe.

Chapitre 22 contient des informations détaillées sur les utilisateurs et sur la gestion des droits.

Comme l'identité des utilisateurs est identique pour tout le cluster de bases de données, `pg_authid` est partagé par toutes les bases du cluster ; il n'existe qu'une seule copie de `pg_authid` par cluster, non une par base de données.

Tableau 53.8. Colonnes de `pg_authid`

Type Description
`oid` `oid` Identifiant de ligne
`rolname` `name` Nom du rôle
`rolsuper` `bool` Le rôle est superutilisateur
`rolinherit` `bool` Le rôle hérite automatiquement des droits des rôles dont il est membre
`rolcreaterole` `bool` Le rôle peut créer d'autres rôles
`rolcreatedb` `bool` Le rôle peut créer des bases de données
`rolcanlogin` `bool` Le rôle peut se connecter, c'est-à-dire qu'il peut être donné comme identifiant d'autorisation de session.
`rolreplication` `bool`

Type Description
Le rôle est un rôle de réplication. Ce type de rôle peut initier des connexions de réplication et créer/supprimer des slots de réplication.
`rolbypassrls` `bool` Le rôle contourne chaque politique de sécurité niveau ligne. Voir Section 5.8 pour plus d'informations.
`rolconnlimit` `int4` Pour les rôles qui peuvent se connecter, indique le nombre maximum de connexions concurrentes que le rôle peut initier. -1 signifie qu'il n'y a pas de limite, -2 indique que la base de données est invalide.
`rolpassword` `text` Mot de passe chiffré ; NULL si aucun. Le format dépend de la forme de chiffrement utilisé.
`rolvaliduntil` `timestamptz` Date d'expiration du mot de passe (utilisée uniquement pour l'authentification par mot de passe) ; NULL si indéfiniment valable

Pour un mot de passe chiffré en MD5, la colonne `rolpassword` commencera avec la chaîne `md5` suivi d'un hachage MD5 hexadécimal sur 32 caractères. Le hachage MD5 sera à partir du mot de passe de l'utilisateur concaténé au nom de l'utilisateur. Par exemple, si l'utilisateur `joe` a pour mot de passe `xyzzy`, PostgreSQL enregistrera le hachage MD5 de `xyzzyjoe`.

Si le mot de passe est chiffré avec SCRAM-SHA-256, il a le format :

```
SCRAM-SHA-256$<nombre
 d'itération>:<sel>$<CléEnregistrée>:<CléServeur>
```

où *`sel`*, *`CléEnregistré`* et *`CléServeur`* sont dans un format Base64. Ce format est le même que celui spécifié par la RFC 5803[1].

53.9. `pg_auth_members`

Le catalogue `pg_auth_members` contient les relations d'appartenance entre les rôles. Tout ensemble non circulaire d'appartenances est autorisé.

Parce que les identités de l'utilisateur sont valables sur l'ensemble du cluster, `pg_auth_members` est partagé par toutes les bases de données d'un cluster : il n'existe qu'une seule copie de `pg_auth_members` par cluster, pas une par base de données.

Tableau 53.9. Colonnes de `pg_auth_members`

Type Description
`roleid` `oid` (référence `pg_authid.oid`) Identifiant d'un rôle qui a un membre
`member` `oid` (référence `pg_authid.oid`) Identifiant d'un rôle qui est membre d'un `roleid`
`grantor` `oid` (référence `pg_authid.oid`) Identifiant du rôle qui a autorisé cette appartenance
`admin_option` `bool` Vrai si `member` peut donner l'appartenance à `roleid` aux autres

[1] https://datatracker.ietf.org/doc/html/rfc5803

53.10. pg_cast

Le catalogue pg_cast stocke les chemins de conversion de type de donnée, qu'il s'agisse de ceux par défaut ou ceux définis par un utilisateur.

pg_cast ne représente pas toutes les conversions de type que le système connaît, seulement celles qui ne peuvent pas se déduire à partir de règles génériques. Par exemple, la conversion entre un domaine et son type de base n'est pas représentée explicitement dans pg_cast. Autre exception importante : « les conversions automatiques d'entrée/sortie », celles réalisées en utilisant les propres fonctions d'entrée/sortie du type de données pour convertir vers ou à partir du text ou des autres types de chaînes de caractères, ne sont pas représentées explicitement dans pg_cast.

Tableau 53.10. Colonnes de pg_cast

Type Description
oid oid Identifiant de ligne
castsource oid (référence pg_type.oid) OID du type de données source
casttarget oid (référence pg_type.oid) OID du type de données cible
castfunc oid (référence pg_proc.oid) OID de la fonction à utiliser pour réaliser la conversion. 0 si la méthode ne requiert pas une fonction.
castcontext char Indique dans quel contexte la conversion peut être utilisée. e si seules les conversions explicites sont autorisées (avec CAST ou ::). a si les conversions implicites lors de l'affectation à une colonne sont autorisées, en plus des conversions explicites. i si les conversions implicites dans les expressions sont autorisées en plus des autres cas.
castmethod char Indique comment la conversion est effectuée. f signifie que la fonction indiquée dans le champ castfunc est utilisée. i signifie que les fonctions d'entrée/sortie sont utilisées. b signifie que les types sont binairement coercibles, et que par conséquent aucune conversion n'est nécessaire.

Les fonctions de transtypage listées dans pg_cast doivent toujours prendre le type source de la conversion comme type du premier argument et renvoyer le type de destination de la conversion comme type de retour. Une fonction de conversion peut avoir jusqu'à trois arguments. Le deuxième argument, s'il est présent, doit être de type integer ; il reçoit le modificateur de type associé avec le type de destination ou 1 s'il n'y en a pas. Le troisième argument, s'il est présent, doit être de type boolean ; il reçoit true si la conversion est une conversion explicite, false sinon.

Il est possible de créer une entrée pg_cast dans laquelle les types source et cible sont identiques si la fonction associée prend plus d'un argument. De telles entrées représentent les « fonctions de forçage de longueur » qui forcent la validité des valeurs de ce type pour une valeur particulière du modificateur de type.

Quand une entrée pg_cast possède des types différents pour la source et la cible et une fonction qui prend plus d'un argument, le transtypage et le forçage de longueur s'effectuent en une seule étape. Lorsqu'une telle entrée n'est pas disponible, le forçage vers un type qui utilise un modificateur de type implique deux étapes, une de transtypage, l'autre pour appliquer le modificateur.

53.11. pg_class

Le catalogue pg_class décrit les tables, et les autres objets qui contiennent des colonnes ou ressemblent à une table. Cela inclut les index (mais il faut aussi aller voir dans pg_index), les séquences (mais voir aussi pg_sequence), les vues, les vues matérialisées, les types composites et les tables TOAST ; voir relkind. Par la suite, lorsque l'on parle de « relation », on sous-entend tous ces types d'objets. Les colonnes de pg_class ne sont pas toutes significatives pour tous les types de relations.

Tableau 53.11. Colonnes de pg_class

Type Description
oid oid Identifiant de ligne
relname name Nom de la table, vue, index, etc.
relnamespace oid (référence pg_namespace.oid) OID du *namespace* qui contient la relation.
reltype oid (référence pg_type.oid) OID du type de données qui correspond au type de ligne de la table, s'il y en a un. 0 pour les index, séquences et tables TOAST qui n'ont pas d'entrée dans pg_type.
reloftype oid (référence pg_type.oid) Pour les tables typées, l'OID du type composite sous-jacent. Sinon, 0 dans tous les autres cas.
relowner oid (référence pg_authid.oid) Propriétaire de la relation.
relam oid (référence pg_am.oid) S'il s'agit d'une table ou d'un index, méthode d'accès utilisée (B-tree, hash, etc.) ; sinon zéro (cela arrive pour les séquences ainsi que pour les relations sans stockage, telles que les vues)
relfilenode oid Nom du fichier disque de la relation ; zéro signifie que c'est une relation « mapped » dont le nom de fichier est déterminé par un statut de bas niveau.
reltablespace oid (référence pg_tablespace.oid) Le *tablespace* dans lequel est stocké la relation. Si 0, il s'agit du *tablespace* par défaut de la base de données. Sans intérêt si la relation n'a pas de fichier sur disque, sauf pour les tables partitionnées, où il s'agit du tablespace dans lequel les partitions sont créées so aucun tablespace n'est précisé dans la commande de création.
relpages int4 Taille du fichier disque, exprimée en pages (de taille BLCKSZ). Ce n'est qu'une estimation utilisée par le planificateur. Elle est mise à jour par les commandes VACUUM, ANALYZE et quelques commandes DDL comme CREATE INDEX.
reltuples float4 Nombre de lignes de la table. Ce n'est qu'une estimation utilisée par le planificateur. Elle est mise à jour par VACUUM, ANALYZE et quelques commandes DDL comme CREATE INDEX. Si la table n'a jamais été traitée par un VACUUM ou un ANALYZE, reltuples contient -1 indiquant que le nombre de lignes est inconnu.
relallvisible int4 Nombre de pages marquées entièrement visibles dans la carte de visibilité de la table. Ceci n'est qu'une estimation utilisée par le planificateur. Elle est mise à jour par VACUUM, ANALYZE et quelques commandes DDL comme CREATE INDEX.
reltoastrelid oid (référence pg_class.oid) OID de la table TOAST associée à cette table. Zéro s'il n'y en a pas. La table TOAST stocke les attributs de grande taille « hors ligne » dans une table secondaire.

Type Description
`relhasindex` `bool` Vrai si c'est une table et qu'elle possède (ou possédait encore récemment) quelque index.
`relisshared` `bool` Vrai si cette table est partagée par toutes les bases de données du cluster. Seuls certains catalogues système (comme `pg_database`) sont partagés.
`relpersistence` `char` `p` = table/séquence permanente, `u` = table/séquence non journalisée, `t` = table/séquence temporaire
`relkind` `char` `r` = table, `i` = index, `S` = séquence, `t` = table TOAST, `v` = vue, `m` = vue matérialisée, `c` = type composite, `f` = table externe, `p` = table partitionnée `I` = index partitionné
`relnatts` `int2` Nombre de colonnes utilisateur dans la relation (sans compter les colonnes système). Il doit y avoir le même nombre d'entrées dans `pg_attribute`. Voir aussi `pg_attribute.attnum`.
`relchecks` `int2` Nombre de contraintes de vérification (`CHECK`) sur la table ; voir le catalogue `pg_constraint`.
`relhasrules` `bool` Vrai si la table contient (ou a contenu) des règles ; voir le catalogue `pg_rewrite`.
`relhastriggers` `bool` Vrai si la table a (ou a eu) des triggers ; voir le catalogue `pg_trigger`
`relhassubclass` `bool` Vrai si au moins une table ou un index hérite ou a hérité d'un enfant, ou a une partition
`relrowsecurity` `bool` Vrai si la table a la sécurité niveau ligne activée ; voir le catalogue `pg_policy`
`relforcerowsecurity` `bool` Vrai si la sécurité niveau ligne (si activée) s'appliquera aussi au propriétaire de la table ; voir le catalogue `pg_policy`
`relispopulated` `bool` Vrai si la relation est peuplée (ceci est vrai pour toutes les relations autres que certaines vues matérialisées)
`relreplident` `char` Colonnes utilisées pour former une « identité de réplica » pour les lignes : `d` = par défaut (clé primaire, si présente), `n` = rien, `f` = toutes les colonnes `i` = index avec `indisreplident` configuré (identique à rien si l'index utilisé a été supprimé)
`relispartition` `bool` True si la table ou l'index est une partition
`relrewrite` `oid` (référence `pg_class.oid`) Pour les nouvelles relations écrites lors d'une opération DDL qui requiert une réécriture de la table, cette colonne contient l'OID de la relation originale ; zéro sinon. Cet état est seulement visible en interne ; ce champ ne devrait jamais contenir autre chose que zéro pour une relation visible par l'utilisateur.
`relfrozenxid` `xid` Tous les ID de transaction avant celui-ci ont été remplacés par un ID de transaction permanent (« frozen »). Ceci est utilisé pour déterminer si la table doit être nettoyée (VACUUM) pour éviter un bouclage des ID de transaction (*ID wraparound*) ou pour compacter `pg_xact`. 0 (`InvalidTransactionId`) si la relation n'est pas une table.
`relminmxid` `xid`

Type Description
Tous les identifiants MultiXact avant celui-i ont été remplacés par un identifiant de transaction dans cette table. Ceci est utilisé pour tracer si la table a besoin d'être traitée par le VACUUM pour empêcher un bouclage des identifiants MultiXact ou pour permettre à `pg_multixact` d'être réduits. Cette colonne vaut zéro (`InvalidTransactionId`) si la relation n'est pas une table.
`relacl` `aclitem[]` Droits d'accès ; voir Section 5.7 pour plus de détails.
`reloptions` `text[]` Options spécifiques de la méthode d'accès, représentées par des chaînes du type « motclé=valeur »
`relpartbound` `pg_node_tree` Si la table est une partition (voir `relispartition`), représentation interne de la limite de la partition

Plusieurs des drapeaux booléens dans `pg_class` sont maintenus faiblement : la valeur true est garantie s'il s'agit du bon état, mais elle pourrait ne pas être remise à false immédiatement quand la condition n'est plus vraie. Par exemple, `relhasindex` est configurée par `CREATE INDEX` mais n'est jamais remise à false par `DROP INDEX`. C'est `VACUUM` qui le fera `relhasindex` s'il découvre que la table n'a pas d'index. Cet arrangement évite des fenêtres de vulnérabilité et améliore la concurrence.

53.12. `pg_collation`

Le catalogue `pg_collation` décrit les collationnements disponibles, qui sont essentiellement des correspondances entre un nom SQL et des catégories de locales du système d'exploitation. Voir Section 24.2 pour plus d'informations.

Tableau 53.12. Colonnes de `pg_collation`

Type Description
`oid` `oid` Identifiant de ligne
`collname` `name` Nom du collationnement (unique par schéma et encodage)
`collnamespace` `oid` (référence `pg_namespace.oid`) L'OID du schéma contenant ce collationnement
`collowner` `oid` (référence `pg_authid.oid`) Propriétaire du collationnement
`collprovider` `char` Fournisseur du collationnement : `d` = le défaut de la base de données, `c` = libc, `i` = icu
`collisdeterministic` `bool` Est-ce que le collationnement est déterministe ?
`collencoding` `int4` Encodage pour lequel le collationnement est disponible. -1 s'il fonctionne pour tous les encodages
`collcollate` `text` `LC_COLLATE` pour ce collationnement
`collctype` `text` `LC_CTYPE` pour ce collationnement
`colliculocale` `text`

Type Description
Identifiant de la locale ICU pour cette collation
`collversion` `text` Version spécifique au fournisseur du collationnement. C'est enregistré quand le collationnement est créé, puis vérifié quand il est utilisé, pour détecter les changements dans la définition du collationnement qui pourraient amener une corruption des données.

Notez que la clé unique de ce catalogue est (`collname`, `collencoding`, `collnamespace`) et non pas seulement (`collname`, `collnamespace`). PostgreSQL ignore habituellement tous les collationnement qui n'ont pas de colonne `collencoding` égale soit à l'encodage de la base de données en cours ou -1. La création de nouvelles entrées de même nom qu'une autre entrée dont `collencoding` vaut -1 est interdite. Du coup, il suffit d'utiliser un nom SQL qualifié du schéma (*schéma*.*nom*) pour identifier un collationnement bien que cela ne soit pas unique d'après la définition du catalogue. Ce catalogue a été défini ainsi car initdb le remplit au moment de l'initialisation de l'instance avec les entrées pour toutes les locales disponibles sur le système, donc il doit être capable de contenir les entrées de tous les encodages qui pourraient être utilisés dans l'instance.

Dans la base de données `template0`, il pourrait être utile de créer les collationnement dont l'encodage ne correspond pas à l'encodage de la base de données car ils pourraient correspondre aux encodages de bases de données créées par la suite à partir de ce modèle de base de données. Cela doit être fait manuellement actuellement.

53.13. `pg_constraint`

Le catalogue `pg_constraint` stocke les vérifications, clés primaires, clés uniques, étrangères et d'exclusion des tables. (Les contraintes de colonnes ne sont pas traitées de manière particulière. Elles sont équivalentes à des contraintes de tables.) Les contraintes NOT NULL sont représentées dans le catalogue `pg_attribute`, pas ici.

Les triggers de contraintes définies par des utilisateurs (créés avec `CREATE CONSTRAINT TRIGGER`) ont aussi une entrée dans cette table.

Les contraintes de vérification de domaine sont également stockées dans ce catalogue.

Tableau 53.13. Colonnes de `pg_constraint`

Type Description
`oid` `oid` Identifiant de ligne
`conname` `name` Nom de la contrainte (pas nécessairement unique !)
`connamespace` `oid` (référence `pg_namespace.oid`) OID du *namespace* qui contient la contrainte.
`contype` `char` `c` = contrainte de vérification, `f` = contrainte de clé étrangère, `p` = contrainte de clé primaire, `u` = contrainte d'unicité, `t` = contrainte trigger, `x` = contrainte d'exclusion
`condeferrable` `bool` La contrainte peut-elle être retardée (*deferable*) ?
`condeferred` `bool` La contrainte est-elle retardée par défaut ?
`convalidated` `bool` La contrainte a-t-elle été validée ? actuellement, peut seulement valoir false pour les clés étrangères et les contraintes CHECK

Type Description
`conrelid` `oid` (référence `pg_class.oid`) Table à laquelle appartient la contrainte ; zéro si ce n'est pas une contrainte de table.
`contypid` `oid` (référence `pg_type.oid`) Domaine auquel appartient la contrainte ; zéro si ce n'est pas une contrainte de domaine.
`conindid` `oid` (référence `pg_class.oid`) L'index qui force cette contrainte (unique, clé primaire, clé étrangère, d'exclusion) ; sinon zéro
`conparentid` `oid` (référence `pg_constraint.oid`) La contrainte correspondante dans la table partitionnée parent si c'est une contrainte dans une partition ; sinon zéro
`confrelid` `oid` (référence `pg_class.oid`) Si c'est une clé étrangère, la table référencée ; sinon zéro
`confupdtype` `char` Code de l'action de mise à jour de la clé étrangère : `a` = no action, `r` = restrict, `c` = cascade, `n` = set null, `d` = set default
`confdeltype` `char` Code de l'action de suppression de clé étrangère : `a` = no action, `r` = restrict, `c` = cascade, `n` = set null, `d` = set default
`confmatchtype` `char` Type de concordance de la clé étrangère : `f` = full, `p` = partial, `s` = simple
`conislocal` `bool` Cette contrainte est définie localement dans la relation. Notez qu'une contrainte peut être définie localement et héritée simultanément
`coninhcount` `int4` Le nombre d'ancêtres d'héritage directs que cette contraite possède. Une contrainte avec un nombre non nul d'ancêtres ne peut être ni supprimée ni renommée.
`connoinherit` `bool` Cette contrainte est définie localement pour la relation. C'est une contrainte non héritable.
`conkey` `int2[]` (référence `pg_attribute.attnum`) S'il s'agit d'une contrainte de table (incluant les clés étrangères mais pas les triggers de contraintes), liste des colonnes contraintes
`confkey` `int2[]` (référence `pg_attribute.attnum`) S'il s'agit d'une clé étrangère, liste des colonnes référencées
`conpfeqop` `oid[]` (référence `pg_operator.oid`) S'il s'agit d'une clé étrangère, liste des opérateurs d'égalité pour les comparaisons clé primaire/clé étrangère
`conppeqop` `oid[]` (référence `pg_operator.oid`) S'il s'agit d'une clé étrangère, liste des opérateurs d'égalité pour les comparaisons clé primaire/clé primaire
`conffeqop` `oid[]` (référence `pg_operator.oid`) S'il s'agit d'une clé étrangère, liste des opérateurs d'égalité pour les comparaisons clé étrangère/clé étrangère
`confdelsetcols` `int2[]` (references `pg_attribute.attnum`) S'il s'agit d'une clé étrangère avec une action `SET NULL` ou `SET DEFAULT` pour une suppression de ligne, liste des colonnes qui seront mises à jour. Si null, toutes les colonnes référençantes seront mises à jour.
`conexclop` `oid[]` (référence `pg_operator.oid`) Si une contrainte d'exclusion, liste les opérateurs d'exclusion par colonne

Type Description
conbin pg_node_tree Si c'est une contrainte CHECK, représentation interne de l'expression. (Il est recommandé d'utiliser pg_get_constraintdef() pour extraire la définition d'une contrainte CHECK.)

Dans le cas d'une contrainte d'exclusion, conkey est seulement utile pour les éléments contraints qui sont de simples références de colonnes. Dans les autres cas, un zéro apparaît dans conkey et l'index associé doit être consulté pour découvrir l'expression contrainte. (du coup, conkey a le même contenu que pg_index.indkey pour l'index.)

> **Note**
>
> pg_class.relchecks doit accepter le même nombre de contraintes de vérification pour chaque relation.

53.14. pg_conversion

Le catalogue pg_conversion décrit les fonctions de conversion de codage. Voir la commande CREATE CONVERSION pour plus d'informations.

Tableau 53.14. Colonnes de pg_conversion

Type Description
oid oid Identifiant de ligne
conname name Nom de la conversion (unique au sein d'un *namespace*)
connamespace oid (référence pg_namespace.oid) OID du *namespace* qui contient la conversion.
conowner oid (référence pg_authid.oid) Propriétaire de la conversion
conforencoding int4 Identifiant de l'encodage source (pg_encoding_to_char() peut traduire ce nombre par le nom de l'encodage)
contoencoding int4 Identifiant de l'encodage de destination (pg_encoding_to_char() peut traduire ce nombre par le nom de l'encodage)
conproc regproc (référence pg_proc.oid) Procédure de conversion
condefault bool Vrai s'il s'agit de la conversion par défaut

53.15. pg_database

Le catalogue pg_database stocke les informations concernant les bases de données disponibles. Celles-ci sont créées avec la commande CREATE DATABASE. Consulter le Chapitre 23 pour les détails sur la signification de certains paramètres.

Contrairement à la plupart des catalogues système, `pg_database` est partagé par toutes les bases de données d'un cluster : il n'y a qu'une seule copie de `pg_database` par cluster, pas une par base.

Tableau 53.15. Colonnes de `pg_database`

Type **Description**
`oid` `oid` Identifiant de ligne
`datname` `name` Nom de la base de données
`datdba` `oid` (référence `pg_authid.oid`) Propriétaire de la base, généralement l'utilisateur qui l'a créée
`encoding` `int4` Encodage de la base de données (la fonction `pg_encoding_to_char()` peut convertir ce nombre en nom de l'encodage)
`datlocprovider` `char` Fournisseur de la locale pour cette base de données : `c` = libc, `i` = icu
`datistemplate` `bool` Si vrai, alors cette base de données peut être clonée par tout utilisateur ayant l'attribut `CREATEDB` ; si faux, seuls les superutilisateurs et le propriétaire de la base de données peuvent la cloner.
`datallowconn` `bool` Si ce champ est faux, alors personne ne peut se connecter à cette base de données. Ceci est utilisé pour interdire toute modification de la base `template0`.
`datconnlimit` `int4` Nombre maximum de connexions concurrentes autorisées sur la base de données. -1 indique l'absence de limite.
`datfrozenxid` `xid` Tous les ID de transaction avant celui-ci ont été remplacés par un ID de transaction permanent (« frozen »). Ceci est utilisé pour déterminer si la table doit être nettoyée (VACUUM) pour éviter un bouclage des ID de transaction (*ID wraparound*) ou pour compacter `pg_xact`. C'est la valeur minimale des valeurs par table de `pg_class.relfrozenxid`.
`datminmxid` `xid` Tous les identifiants MultiXact avant celui-i ont été remplacés par un identifiant de transaction dans cette table. Ceci est utilisé pour tracer si la base de données a besoin d'être traitée par le VACUUM pour empêcher un bouclage des identifiants MultiXact ou pour permettre à `pg_multixact` d'être réduit. Il s'agit aussi de la valeur minimale des valeurs de `pg_class.relminmxid` pour chaque table.
`dattablespace` `oid` (référence `pg_tablespace.oid`) Le *tablespace* par défaut de la base de données. Dans cette base de données, toutes les tables pour lesquelles `pg_class.reltablespace` vaut zéro sont stockées dans celui-ci ; en particulier, tous les catalogues système non partagés s'y trouvent.
`datcollate` `text` LC_COLLATE pour cette base de données
`datctype` `text` LC_CTYPE pour cette base de données
`daticulocale` `text` ICU locale ID pour cette base de données
`datcollversion` `text`

Type Description
Version spécifique du fournisseur pour cette collation. Cette information est enregistrée quand la base de données est créée et est vérifiée à chaque utilisation de la base pour détecter les changements de définition de la collation, qui pourraient amener à une corruption des données.
`datacl aclitem[]` Droits d'accès ; voir Section 5.7 pour les détails.

53.16. `pg_db_role_setting`

Le catalogue `pg_db_role_setting` enregistre les valeurs par défaut qui ont été configurées pour les variables de configuration, pour chaque combinaison de rôle et de base.

Contrairement à la plupart des catalogues systèmes, `pg_db_role_setting` est partagé parmi toutes les bases de données de l'instance : il n'existe qu'une copie de `pg_db_role_setting` par instance, pas une par base de données.

Tableau 53.16. Colonnes de `pg_db_role_setting`

Type Description
`setdatabase oid` (référence `pg_database.oid`) L'OID de la base de données pour laquelle la configuration est applicable ; zéro si cette configuration n'est pas spécifique à une base de données
`setrole oid` (référence `pg_authid.oid`) L'OID du rôle pour laquelle la configuration est applicable ; zéro si cette configuration n'est pas spécifique à un rôle
`setconfig text[]` Valeurs par défaut pour les variables de configuration

53.17. `pg_default_acl`

Le catalogue `pg_default_acl` enregistre les droits initiaux à affecter aux nouveaux objets créés.

Tableau 53.17. Colonnes de `pg_default_acl`

Type Description
`oid oid` Identifiant de ligne
`defaclrole oid` (référence `pg_authid.oid`) OID du rôle associé à cette entrée
`defaclnamespace oid` (référence `pg_namespace.oid`) OID du schéma associé à cette entrée, zéro si aucun
`defaclobjtype char` Type de l'objet pour cette entrée : `r` = relation (table, vue), `S` = séquence, `f` = fonction, `T` = type, `n` = schéma
`defaclacl aclitem[]` Droits d'accès qu'auront les nouveaux objets de ce type

Une entrée `pg_default_acl` affiche les droits initiaux affectés à un objet appartenant à l'utilisateur indiqué. Il existe actuellement deux types d'entrées : des entrées « globales » avec

defaclnamespace = zéro, et des entrées « par schéma » qui référencent un schéma. Si une entrée globale est présente, alors elle *surcharge* les droits par défaut codés en dur pour le type de l'objet. Une entrée par schéma, si présente, représente les droits à *ajouter* aux droits par défaut globaux ou aux droits codés en dur.

Notez que quand une entrée de droits (ACL) dans un autre catalogue est NULL, cela veut dire que les droits par défaut codés en dur sont utilisés pour cet objet, et *non pas* ce qui pourrait être dans pg_default_acl à ce moment. pg_default_acl est seulement consulté durant la création de l'objet.

53.18. pg_depend

Le catalogue pg_depend enregistre les relations de dépendances entre les objets de la base de données. Cette information permet à la commande DROP de trouver les objets qui doivent être supprimés conjointement par la commande DROP CASCADE ou au contraire empêchent la suppression dans le cas de DROP RESTRICT.

Voir aussi pg_shdepend, qui remplit la même fonction pour les dépendances impliquant des objets partagés sur tout le cluster.

Tableau 53.18. Colonnes de pg_depend

Type Description
classid oid (référence pg_class.oid) OID du catalogue système dans lequel l'objet dépendant se trouve
objid oid (référence toute colonne OID) OID de l'objet dépendant
objsubid int4 Pour une colonne de table, ce champ indique le numéro de colonne (les champs objid et classid font référence à la table elle-même). Pour tous les autres types d'objets, cette colonne est à 0.
refclassid oid (référence pg_class.oid) OID du catalogue système dans lequel l'objet référencé se trouve.
refobjid oid (référence toute colonne OID) OID de l'objet référencé
refobjsubid int4 Pour une colonne de table, ce champ indique le numéro de colonne (les champs refobjid et refclassid font référence à la table elle même). Pour tous les autres types d'objets, cette colonne est à 0.
deptype char Code définissant la sémantique particulière de la relation de dépendance. Voir le texte.

Dans tous les cas, une entrée de pg_depend indique que l'objet de référence ne peut pas être supprimé sans supprimer aussi l'objet dépendant. Néanmoins, il y a des nuances, identifiées par deptype :

DEPENDENCY_NORMAL (n)

Une relation normale entre des objets créés séparément. L'objet dépendant peut être supprimé sans affecter l'objet référencé. Ce dernier ne peut être supprimé qu'en précisant l'option CASCADE, auquel cas l'objet dépendant est supprimé lui-aussi. Exemple : une colonne de table a une dépendance normale avec ses types de données.

DEPENDENCY_AUTO (a)

L'objet dépendant peut être supprimé séparément de l'objet référencé, mais il l'est automatiquement avec la suppression de ce dernier, quel que soit le mode RESTRICT ou

CASCADE. Exemple : une contrainte nommée sur une table est auto-dépendante de la table, elle est automatiquement supprimée avec celle-ci.

DEPENDENCY_INTERNAL (i)

L'objet dépendant est créé conjointement à l'objet référencé et fait partie intégrante de son implantation interne. Un DROP direct de l'objet dépendant est interdit (l'utilisateur est averti qu'il peut effectuer un DROP de l'objet référencé à la place). La suppression de l'objet référencé résultera en une suppression automatique de l'objet dépendant que CASCADE soit précisé ou non. Si l'objet dépendant doit être supprimé à cause de la dépendance en un autre objet en cours de suppression, la suppression est convertie en une suppression de l'objet référencé, pour que les dépendances NORMAL et AUTO de l'objet dépendant se comportent comme s'ils s'agissaient de dépendances de l'objet référencé. Par exemple, la règle ON SELECT d'une vue est rendue en interne dépendante de la vue, empêchant sa suppression tant que la vue existe. Les dépendances de la règle (comme les tables auxquelles elle fait référence) agissent comme si elles dépendaient de la vue.

DEPENDENCY_PARTITION_PRI (P)
DEPENDENCY_PARTITION_SEC (S)

L'objet dépendant a été créé lors de la création de l'objet référencé, mais il s'agit vraiment d'un détail d'implémentation interne. Néanmoins, contrairement à INTERNAL, il existe plus d'un objet référencé. L'objet dépendant ne doit pas être supprimé sauf si au moins un des objets référencés est supprimé ; si c'est le cas, l'objet dépendant doit être supprimé que CASCADE soit indiqué ou pas. De plus, contrairement à INTERNAL, une suppression de certains autres objets dont l'objet dépendant dépend ne résulte pas en une suppression automatique d'objet référencé par une patition. De ce fait, si la suppression ne cascade pas vers au moins un de ces objets via un autre chemin, elle sera refusée. (Dans la plupart des cas, l'objet dépendant partage toutes ses dépendances non-partitions avec au moins un objet référencé par partition, donc cette restriction ne résulte pas dans le blocage de suppression en cascade). Les dépendances de partitions primaire et secondaire se comportent de façon identique sauf que la dépendance primaire est préféré pour une utilisation dans les messages d'erreur. De ce fait, un objet dépendant d'une partition devrait avoir une dépendance de partition primaire et une ou plusieurs dépendances de partition secondaires. Notez que les dépendances de partition sont faites en plus, et non pas à la place, de toute dépendant que l'objet aurait normalement. Ceci simplifie les opérations ATTACH/DETACH PARTITION : les dépendances partitions ont seulement besoin d'être ajoutées ou supprimées. Par exemple, une partition d'un index est rendue dépendante de la partition de la table et de l'index partitionné, pour qu'elle disparaisse si l'un des deux est supprimé, mais pas autrement. La dépendance sur l'index parent est primaire, pour que si l'utilisateur essaie de supprimer l'index partitionné, le message d'erreur suggèrera la suppression de l'index parent à la place (et non pas de la table).

DEPENDENCY_EXTENSION (e)

L'objet dépendant est un membre de l'*extension* qui est l'objet référencé (voir pg_extension). L'objet dépendant peut être supprimé seulement via l'instruction DROP EXTENSION sur l'objet référence. Fonctionnellement, ce type de dépendance agit de la même façon qu'une dépendance INTERNAL mais il est séparé pour des raisons de clarté et pour simplifier pg_dump.

DEPENDENCY_AUTO_EXTENSION (x)

L'objet dépendant n'est pas un membre de l'extension qui est l'objet référencé (et donc ne doit pas être ignoré par pg_dump), mais il ne peut fonctionner sans l'extension et doit être supprimé automatiquement si l'extension l'est. L'objet dépendant peut aussi être supprimé tout seul. Fonctionnellement, ce type de dépendance agit de la même façon que la dépendance AUTO, mais il est séparé pour des raisons de clareté et pour simplifier pg_dump.

D'autres types de dépendance peuvent apparaître dans le futur.

Notez qu'il est bien possible que les deux objets soient liés par plus d'un enregistrement dans pg_depend. Par exemple, un index fils partitionné pourrait avoir à la fois une dépendance de type

partition sur la table partitionnée associée et une dépendance automatique de chaque colonne de la table qu'il indexe. Ce type de situation exprime l'union de plusieurs sémantiques de dépendances. Un objet dépendant peut être supprimé sans CASCADE si toutes ses dépendances satisfont la condition de suppression automatique. Dans le cas contraire, toutes les restrictions de dépendance sur les objets à supprimer ensemble doivent être satisfaites.

La plupart des objets créés lors de l'exécution d'initdb sont considérés « fixés », ce qui signifie que le système lui-même dépend de ces objets. De ce fait, il n'est pas autorisé de les supprimer. De plus, sachant que ces objets fixés ne seront pas supprimés, le mécanisme de dépendance ne s'occupe pas de créer les entrées dans pg_depend montrant les dépendances entre eux. De ce fait, par exemple, une colonne d'une table de type numeric a une dépendance de type NORMAL sur le type de données numeric, mais cette dépendance n'apparaît pas dans pg_depend.

53.19. pg_description

Le catalogue pg_description stocke les descriptions (commentaires) optionnelles de chaque objet de la base de données. Les descriptions sont manipulées avec la commande COMMENT et lues avec les commandes \d de psql. pg_description contient les descriptions prédifinies de nombreux objets internes.

Voir aussi pg_shdescription, qui offre la même fonction pour les descriptions des objets partagés au sein d'un cluster.

Tableau 53.19. Colonnes de pg_description

Type Description
objoid oid (référence toute colonne OID) OID de l'objet commenté
classoid oid (référence pg_class.oid) OID du catalogue système dans lequel apparaît l'objet
objsubid int4 Pour un commentaire de colonne de table, le numéro de la colonne. Les champs objoid et classoid font référence à la table elle-même. Pour tous les autres types de données, cette colonne est à 0.
description text Texte quelconque commentant l'objet

53.20. pg_enum

Le catalogue système pg_enum contient des entrées indiquant les valeurs et labels de chaque type enum. La représentation interne d'une valeur enum donnée est en fait l'OID de sa ligne associée dans pg_enum.

Tableau 53.20. Colonnes de pg_enum

Type Description
oid oid Identifiant de ligne
enumtypid oid (référence pg_type.oid) OID de l'entrée pg_type correspondant à cette valeur d'enum
enumsortorder float4 La position de tri de cette valeur enum dans son type enum
enumlabel name

Type Description
Le label texte pour cette valeur d'enum

Les OID des lignes de `pg_enum` suivent une règle spéciale : les OID pairs sont garantis triés de la même façon que l'ordre de tri de leur type enum. Autrement dit, si deux OID pairs appartiennent au même type enum, l'OID le plus petit doit avoir la plus petite valeur dans la colonne `enumsortorder`. Les valeurs d'OID impaires n'ont pas d'ordre de tri. Cette règle permet que les routines de comparaison d'enum évitent les recherches dans les catalogues dans la plupart des cas standards. Les routines qui créent et modifient les types enum tentent d'affecter des OID paires aux valeurs enum tant que c'est possible.

Quand un type enum est créé, ses membres sont affectés dans l'ordre des positions 1..*n*. Les membres ajoutés par la suite doivent se voir affecter des valeurs négatives ou fractionnelles de `enumsortorder`. Le seul prérequis pour ces valeurs est qu'elles soient correctement triées et uniques pour chaque type enum.

53.21. `pg_event_trigger`

Le catalogue `pg_event_trigger` enregistre les triggers sur événement. Voir Chapitre 40 pour plus d'informations.

Tableau 53.21. Colonnes de `pg_event_trigger`

Type Description
`oid` `oid` Identifiant de ligne
`evtname` `name` Nom du trigger (doit être unique)
`evtevent` `name` Identifie l'événement pour lequel ce trigger se déclenche
`evtowner` `oid` (référence `pg_authid.oid`) Propriétaire du trigger d'événement
`evtfoid` `oid` (référence `pg_proc.oid`) La fonction à appeler
`evtenabled` `char` Contrôle le mode session_replication_role dans lequel ce trigger d'événement se déclenche. `O` = le trigger se déclenche dans les modes « origin » et « local », `D` = le trigger est désactivé, `R` = le trigger se déclenche dans le mode « replica », `A` = le trigger se déclenche toujours.
`evttags` `text[]` Balises de commande pour lesquelles ce trigger va se déclencher. Si NULL, le déclenchement de ce trigger n'est pas restreint sur la base de la balise de commande.

53.22. `pg_extension`

Le catalogue `pg_extension` stocke les informations sur les extensions installées. Voir Section 38.17 pour des détails sur les extensions.

Tableau 53.22. Colonnes de `pg_extension`

Type Description
`oid` `oid`

Type Description
Identifiant de ligne
`extname` `name` Nom de l'extension
`extowner` `oid` (référence `pg_authid.oid`) Propriétaire de l'extension
`extnamespace` `oid` (référence `pg_namespace.oid`) Schéma contenant les objets exportés de l'extension
`extrelocatable` `bool` Vrai si l'extension peut être déplacée dans un autre schéma
`extversion` `text` Nom de la version de l'extension
`extconfig` `oid[]` (référence `pg_class.oid`) Tableaux d'OID de type `regclass` pour la table de configuration de l'extension, ou `NULL` si aucun
`extcondition` `text[]` Tableau de conditions de filtre (clauses `WHERE`) pour la table de configuration de l'extension, ou `NULL` si aucun

Notez que, contrairement aux autres catalogues ayant une colonne de « schéma », `extnamespace` n'est pas le schéma contenant l'extension. Les noms des extensions ne sont jamais qualifiés d'un schéma. En fait, `extnamespace` indique le schéma qui contient la plupart ou tous les objets de l'extension. Si `extrelocatable` vaut true, alors ce schéma doit en fait contenir tous les objets de l'extension, dont le nom peut être qualifié avec le nom du schéma.

53.23. `pg_foreign_data_wrapper`

Le catalogue `pg_foreign_data_wrapper` stocke les définitions des wrappers de données distantes. Un wrapper de données distantes est le mécanisme par lequel des données externes, stockées sur des serveurs distants, sont accédées.

Tableau 53.23. Colonnes de `pg_foreign_data_wrapper`

Type Description
`oid` `oid` Identifiant de ligne
`fdwname` `name` Nom du wrapper de données distantes
`fdwowner` `oid` (référence `pg_authid.oid`) Propriétaire du wrapper de données distantes
`fdwhandler` `oid` (référence `pg_proc.oid`) Référence une fonction de gestion responsable de la fourniture de routines d'exécution pour le wrapper de données distantes. Zéro si aucune fonction n'est fournie.
`fdwvalidator` `oid` (référence `pg_proc.oid`) Référence le validateur de fonction qui est responsable de vérifier la validité des options passées au wrapper de données distantes, ainsi que les options des serveurs distants et les correspondances utilisateurs du wrapper de données distantes. Zéro si aucun validateur n'est fourni.
`fdwacl` `aclitem[]` Droits d'accès ; voir Section 5.7 pour plus de détails

Type Description
`fdwoptions` `text[]` Options spécifiques pour un wrapper de données distantes, sous la forme de chaînes « motclé=valeur »

53.24. `pg_foreign_server`

Le catalogue `pg_foreign_server` stocke les définitions de serveurs distants. Un serveur distant décrit une source de données externes, comme un serveur distant. Les serveurs distants sont accédés via des wrappers de données distantes.

Tableau 53.24. Colonnes de `pg_foreign_server`

Type Description
`oid` `oid` Identifiant de ligne
`srvname` `name` Nom du serveur distant
`srvowner` `oid` (référence `pg_authid.oid`) Propriétaire du serveur distant
`srvfdw` `oid` (référence `pg_foreign_data_wrapper.oid`) OID du wrapper de données distantes pour ce serveur distant
`srvtype` `text` Type du serveur (optionnel)
`srvversion` `text` Version du serveur (optionnel)
`srvacl` `aclitem[]` Droits d'accès ; voir Section 5.7 pour les détails
`srvoptions` `text[]` Options pour le serveur distant, sous la forme de chaînes « motclé=valeur ».

53.25. `pg_foreign_table`

Le catalogue `pg_foreign_table` contient des informations supplémentaires sur les tables distantes. Une table distante est principalement représentée par une entrée dans le catalogue `pg_class`, comme toute table ordinaire. Son entrée dans `pg_foreign_table` contient les informations pertinentes aux seules tables distantes, et pas aux autres types de relation.

Tableau 53.25. Colonnes de `pg_foreign_table`

Type Description
`ftrelid` `oid` (référence `pg_class.oid`) OID de l'entrée dans le catalogue `pg_class` pour cette table distante
`ftserver` `oid` (référence `pg_foreign_server.oid`) OID du serveur distant pour cette table distante
`ftoptions` `text[]` Options de la table distante, sous la forme de chaînes « clé=valeur »

53.26. pg_index

Le catalogue pg_index contient une partie des informations concernant les index. Le reste se trouve pour l'essentiel dans pg_class.

Tableau 53.26. Colonnes de pg_index

Type Description
indexrelid oid (référence pg_class.oid) OID de l'entrée dans pg_class pour l'index
indrelid oid (référence pg_class.oid) OID de l'entrée dans pg_class de la table sur laquelle porte l'index
indnatts int2 Le nombre total de colonnes dans l'index (duplique pg_class.relnatts). Ce nombre inclut les attributs clé et supplémentaires.
indnkeyatts int2 Le nombre de *colonnes clés* dans l'index, sans compter les *colonnes supplémentaires*, qui sont simplement enregistrés et ne participent pas à la sémantique de l'index
indisunique bool Vrai s'il s'agit d'un index d'unicité
indnullsnotdistinct bool Cette valeur est seulement utilisée pour les index d'unicité. Si sa valeur est false, cet index d'unicité considérera les valeurs NULL comme distinctes les unes des autres (donc l'index peut contenir plusieurs valeurs NULL dans une colonne, ce qui correspond au comportement par défaut de PostgreSQL). Si sa valeur est true, il considérera les valeurs NULL comme égales (donc l'index peut seulement contenir une valeur NULL dans une colonne).
indisprimary bool Vrai s'il s'agit de l'index de clé primaire de la table (indisunique doit toujours être vrai quand ce champ l'est.
indisexclusion bool Vrai s'il s'agit de l'index supportant une contrainte d'exclusion
indimmediate bool Si vrai, la vérification de l'unicité est forcée immédiatement lors de l'insertion (inutile si indisunique ne vaut pas true)
indisclustered bool Vrai si la table a été réorganisée en fonction de l'index
indisvalid bool Si ce drapeau est vrai, l'index est valide pour les requêtes. Faux signifie que l'index peut être incomplet : les opérations INSERT / UPDATE peuvent toujours l'utiliser, mais il ne peut pas être utilisé sans risque pour les requêtes, et, dans le cas d'un index d'unicité, cette propriété n'est plus garantie.
indcheckxmin bool Si vrai, les requêtes ne doivent pas utiliser l'index tant que le xmin de cette ligne de pg_index est en-dessous de leur horizon d'événements TransactionXmin, car la table peut contenir des chaînes HOT cassées avec des lignes incompatibles qu'elles peuvent voir.
indisready bool Si vrai, l'index est actuellement prêt pour les insertions. Faux indique que l'index doit être ignoré par les opérations INSERT / UPDATE
indislive bool

Type Description
Si faux, l'index est en cours de suppression et devrait être complètement ignoré (y compris pour les décisions sur la sûreté de HOT)
`indisreplident` `bool` Si vrai, cet index a été choisi comme « identité de réplication » en utilisant `ALTER TABLE ... REPLICA IDENTITY USING INDEX ...`
`indkey` `int2vector` (référence `pg_attribute.attnum`) C'est un tableau de valeurs `indnatts` qui indique les colonnes de la table indexées. Par exemple, une valeur `1 3` signifie que la première et la troisième colonne de la table composent la clé de l'index. Les colonnes clés viennent avant les colonnes supplémentaires, non clés. Un 0 dans ce tableau indique que l'attribut de l'index correspondant est une expression sur les colonnes de la table plutôt qu'une simple référence de colonne.
`indcollation` `oidvector` (référence `pg_collation.oid`) Pour chaque colonne dans la clé de l'index (`indnkeyatts` values), cette colonne contient l'OID du collationnement à utiliser pour l'index, ou zéro si la colonne n'est pas d'un type de données collationnable.
`indclass` `oidvector` (référence `pg_opclass.oid`) Pour chaque colonne de la clé d'indexation (`indnkeyatts` values), contient l'OID de la classe d'opérateurs à utiliser. Voir `pg_opclass` pour plus de détails.
`indoption` `int2vector` C'est un tableau de valeurs `indnkeyatts` qui enregistrent des drapeaux d'information par colonne. La signification de ces drapeaux est définie par la méthode d'accès à l'index.
`indexprs` `pg_node_tree` Arbres d'expression (en représentation `nodeToString()`) pour les attributs d'index qui ne sont pas de simples références de colonnes. Il s'agit d'une liste qui contient un élément par entrée à 0 dans `indkey`. Nul si tous les attributs d'index sont de simples références.
`indpred` `pg_node_tree` Arbre d'expression (en représentation `nodeToString()`) pour les prédicats d'index partiels. Nul s'il ne s'agit pas d'un index partiel.

53.27. `pg_inherits`

Le catalogue `pg_inherits` enregistre l'information sur la hiérarchie d'héritage des tables et des index. Il existe une entrée pour chaque table ou index enfant direct dans la base de données. (L'héritage indirect peut être déterminé en suivant les chaînes d'entrées.)

Tableau 53.27. Colonnes de `pg_inherits`

Type Description
`inhrelid` `oid` (référence `pg_class.oid`) OID de la table fille ou de l'index fille
`inhparent` `oid` (référence `pg_class.oid`) OID de la table mère ou de l'index mère
`inhseqno` `int4` S'il y a plus d'un parent direct pour une table fille (héritage multiple), ce nombre indique dans quel ordre les colonnes héritées doivent être arrangées. Le compteur commence à 1. Les index ne peuvent pas avoir plusieurs héritages car elle peuvent seulement hériter dans le cas du partitionnement déclaratif.
`inhdetachpending` `bool`

Type Description
`true` pour une partition qui est en cours de détachement ; `false` sinon.

53.28. `pg_init_privs`

Le catalogue `pg_init_privs` enregistre des informations sur les droits initiaux des objets dans le système. Il existe une entrée pour chaque objet de la base qui n'a pas sa configuration initiale des droits.

Les objets peuvent avoir des droits initiaux, soit en ayant ces droits configurés quand le système est initialisé (par initdb) soit quand l'objet est créé lors d'un `CREATE EXTENSION` et que le script de l'extension configure les droits initiaux en utilisant la commande `GRANT`. Notez que le système gère automatiquement l'enregistrement des droits lors de l'exécution du script de l'extension et que les auteurs d'extension ont juste besoin d'utiliser les commandes `GRANT` et `REVOKE` dans leur scripts pour que les droits soient enregistrés. La colonne `privtype` indique si le droit initial a été configuré par initdb ou par une commande `CREATE EXTENSION`.

Les objects qui ont des droits initiaux configurés par initdb auront des entrées où `privtype` vaut `'i'` alors que les objets qui ont des droits initiaux configurés par `CREATE EXTENSION` auront des entrées dont `privtype` vaut `'e'`.

Tableau 53.28. Colonnes de `pg_init_privs`

Type Description
`objoid` `oid` (référence toute colonne OID) L'OID de l'objet spécifié
`classoid` `oid` (référence `pg_class.oid`) L'OID du catalogue spécifique où se trouve l'objet
`objsubid` `int4` Pour une colonne de table, c'est le numéro de la colonne (`objoid` et `classoid` font référence à la table elle-même). Pour tous les autres types d'objet, cette colonne vaut zéro.
`privtype` `char` Un code définissant le type de droit initial pour cet objet ; voir le texte.
`initprivs` `aclitem[]` Les droits d'accès initiaux ; voir Section 5.7 pour les détails

53.29. `pg_language`

Le catalogue `pg_language` enregistre les langages utilisables pour l'écriture de fonctions ou procédures stockées. Voir CREATE LANGUAGE et dans le Chapitre 42 pour plus d'information sur les gestionnaires de langages.

Tableau 53.29. Colonnes de `pg_language`

Type Description
`oid` `oid` Identifiant de ligne
`lanname` `name` Nom du langage
`lanowner` `oid` (référence `pg_authid.oid`)

Type Description
Propriétaire du langage
`lanispl` `bool` Faux pour les langages internes (comme SQL) et vrai pour les langages utilisateur. À l'heure actuelle, pg_dump utilise ce champ pour déterminer les langages à sauvegarder mais cela peut être un jour remplacé par un mécanisme différent.
`lanpltrusted` `bool` Vrai s'il s'agit d'un langage de confiance (*trusted*), ce qui signifie qu'il est supposé ne pas donner accès à ce qui dépasse l'exécution normale des requêtes SQL. Seuls les superutilisateurs peuvent créer des fonctions dans des langages qui ne sont pas dignes de confiance.
`lanplcallfoid` `oid` (référence `pg_proc.oid`) Pour les langages non-internes, ceci référence le gestionnaire de langage, fonction spéciale en charge de l'exécution de toutes les fonctions écrites dans ce langage. Zéro pour les langages internes.
`laninline` `oid` (référence `pg_proc.oid`) Ceci référence une fonction qui est capable d'exécuter des blocs de code anonyme « en ligne » (blocs DO). Zéro si les blocs en ligne ne sont pas supportés
`lanvalidator` `oid` (référence `pg_proc.oid`) Ceci référence une fonction de validation de langage, en charge de vérifier la syntaxe et la validité des nouvelles fonctions lors de leur création. 0 si aucun validateur n'est fourni.
`lanacl` `aclitem[]` Droits d'accès ;; voir Section 5.7 pour les détails.

53.30. `pg_largeobject`

Le catalogue `pg_largeobject` contient les données qui décrivent les « Large Objects » (*large objects*). Un Large Object est identifié par un OID qui lui est affecté lors de sa création. Chaque Large Object est coupé en segments ou « pages » suffisamment petits pour être facilement stockés dans des lignes de `pg_largeobject`. La taille de données par page est définie par `LOBLKSIZE`, qui vaut actuellement `BLCKSZ/4`, soit habituellement 2 ko).

Avant PostgreSQL 9.0, il n'existait pas de droits associés aux « Large Objects ». Du coup, `pg_largeobject` était lisible par tout le monde et pouvait être utilisé pour obtenir les OID (et le contenu) de tous les « Large Objects » du système. Ce n'est plus le cas ; utilisez `pg_largeobject_metadata` pour obtenir une liste des OID des « Large Objects ».

Tableau 53.30. Colonnes de `pg_largeobject`

Type Description
`loid` `oid` (référence `pg_largeobject_metadata.oid`) Identifiant du Large Object qui contient la page
`pageno` `int4` Numéro de la page au sein du Large Object, en partant de 0
`data` `bytea` Données effectivement stockées dans le Large Object. Il ne fait jamais plus de `LOBLKSIZE` mais peut faire moins.

Chaque ligne de `pg_largeobject` contient les données d'une page du Large Object, en commençant au décalage d'octet (`pageno * LOBLKSIZE`) dans l'objet. Ceci permet un stockage diffus : des pages peuvent manquer, d'autres faire moins de `LOBLKSIZE` octets même s'il ne s'agit pas de la dernière de l'objet. Les parties manquantes sont considérées comme des suites de zéro.

53.31. pg_largeobject_metadata

Le catalogue pg_largeobject_metadata contient des méta-données associées aux « Larges Objects ». Les données des « Larges Objects » sont réellement stockées dans pg_largeobject.

Tableau 53.31. Colonnes de pg_largeobject_metadata

Type Description
oid oid Identifiant de ligne
lomowner oid (référence pg_authid.oid) Propriétaire du « Larges Object »
lomacl aclitem[] Droits d'accès ; pour plus de détails, voir Section 5.7

53.32. pg_namespace

Le catalogue pg_namespace stocke les *namespace*. Un *namespace* est la structure sous-jacente aux schémas SQL : chaque *namespace* peut contenir un ensemble séparé de relations, types, etc. sans qu'il y ait de conflit de nommage.

Tableau 53.32. Colonnes de pg_namespace

Type Description
oid oid Identifiant de ligne
nspname name Nom du *namespace*
nspowner oid (référence pg_authid.oid) Propriétaire du *namespace*
nspacl aclitem[] Droits d'accès ; voir Section 5.7 pour les détails.

53.33. pg_opclass

Le catalogue pg_opclass définit les classes d'opérateurs de méthodes d'accès aux index. Chaque classe d'opérateurs définit la sémantique pour les colonnes d'index d'un type particulier et d'une méthode d'accès particulière. Une classe d'opérateurs définit essentiellement qu'une famille d'opérateur particulier est applicable à un type de données indexable particulier. L'ensemble des opérateurs de la famille actuellement utilisables avec la colonne indexée sont tous ceux qui acceptent le type de données de la colonne en tant qu'entrée du côté gauche.

Les classes d'opérateurs sont longuement décrites dans la Section 38.16.

Tableau 53.33. Colonnes de pg_opclass

Type Description
oid oid Identifiant de ligne

Type Description
`opcmethod` `oid` (référence `pg_am.oid`) Méthode d'accès à l'index pour laquelle est définie la classe d'opérateurs
`opcname` `name` Nom de la classe d'opérateurs
`opcnamespace` `oid` (référence `pg_namespace.oid`) *Namespace* de la classe d'opérateurs
`opcowner` `oid` (référence `pg_authid.oid`) Propriétaire de la classe d'opérateurs
`opcfamily` `oid` (référence `pg_opfamily.oid`) Famille d'opérateur contenant la classe d'opérateurs
`opcintype` `oid` (référence `pg_type.oid`) Type de données que la classe d'opérateurs indexe
`opcdefault` `bool` Vrai si la classe d'opérateurs est la classe par défaut pour `opcintype`
`opckeytype` `oid` (référence `pg_type.oid`) Type de données stocké dans l'index ou 0 s'il s'agit du même que `opcintype`

L'`opcmethod` d'une classe d'opérateurs doit coïncider avec l'`opfmethod` de la famille d'opérateurs qui le contient. Il ne doit pas non plus y avoir plus d'une ligne `pg_opclass` pour laquelle `opcdefault` est vrai, quelque soit la combinaison de `opcmethod` et `opcintype`.

53.34. `pg_operator`

Le catalogue `pg_operator` stocke les informations concernant les opérateurs. Voir la commande CREATE OPERATOR et la Section 38.14 pour plus d'informations.

Tableau 53.34. Colonnes de `pg_operator`

Type Description
`oprname` `name` Identifiant de ligne
`oprname` `name` Nom de l'opérateur
`oprnamespace` `oid` (référence `pg_namespace.oid`) OID du *namespace* qui contient l'opérateur
`oprowner` `oid` (référence `pg_authid.oid`) Propriétaire de l'opérateur
`oprkind` `char` `b` = opérateur infix (« les deux ») ou `l` = opérateur préfixe (« gauche »)
`oprcanmerge` `bool` L'opérateur supporte les jointures de fusion
`oprcanhash` `bool` L'opérateur supporte les jointures par découpage
`oprleft` `oid` (référence `pg_type.oid`) Type de l'opérande de gauche (zéro si opérateur préfixe)
`oprright` `oid` (référence `pg_type.oid`) Type de l'opérande de droite

Type Description
`oprresult` `oid` (référence `pg_type.oid`) Type du résultat (zéro pour un opérateur « shell » pas encore défini)
`oprcom` `oid` (référence `pg_operator.oid`) Commutateur de l'opérateur, s'il existe (zéro sinon)
`oprnegate` `oid` (référence `pg_operator.oid`) Négateur de l'opérateur, s'il existe (zéro sinon)
`oprcode` `regproc` (référence `pg_proc.oid`) Fonction codant l'opérateur (zéro pour un opérateur « shell » pas encore défini)
`oprrest` `regproc` (référence `pg_proc.oid`) Fonction d'estimation de la sélectivité de restriction de l'opérateur (zéro sinon)
`oprjoin` `regproc` (référence `pg_proc.oid`) Fonction d'estimation de la sélectivité de jointure de l'opérateur (zéro sinon)

53.35. `pg_opfamily`

Le catalogue `pg_opfamily` définit les familles d'opérateur. Chaque famille d'opérateur est un ensemble d'opérateurs et de routines de support associées codant les sémantiques définies pour une méthode d'accès particulière de l'index. De plus, les opérateurs d'une famille sont tous « compatibles », au sens défini par la méthode d'accès. Le concept de famille d'opérateur autorise l'utilisation des opérateurs inter-type de données avec des index et l'utilisation des sémantiques de méthode d'accès.

Les familles d'opérateur sont décrites dans Section 38.16.

Tableau 53.35. Colonnes de `pg_opfamily`

Type Description
`oid` `oid` Identifiant de ligne
`opfmethod` `oid` (référence `pg_am.oid`) Méthode d'accès à l'index pour la famille d'opérateur
`opfname` `name` Nom de la famille d'opérateur
`opfnamespace` `oid` (référence `pg_namespace.oid`) *namespace* de la famille d'opérateur
`opfowner` `oid` (référence `pg_authid.oid`) Propriétaire de la famille d'opérateur

La majorité des informations définissant une famille d'opérateur n'est pas dans la ligne correspondante de `pg_opfamily` mais dans les lignes associées de `pg_amop`, `pg_amproc`, et `pg_opclass`.

53.36. `pg_parameter_acl`

Le catalogue `pg_parameter_acl` enregistre les paramètres de configuration pour lesquels des droits ont été donnés pour un ou plusieurs rôles. Aucun enregistrement n'est fait pour les paramètres qui ont les droits par défaut.

Contrairement aux autres catalogues systèmes, `pg_parameter_acl` est partagé parmi toutes les bases de données d'une instance : il n'existe qu'une copie de `pg_parameter_acl` par instance, et non pas par base de données.

Tableau 53.36. `pg_parameter_acl` Columns

Type Description
`oid` `oid` Identifiant de ligne
`parname` `text` Nom d'un paramètre de configuration pour lequel des droits ont été donnés
`paracl` `aclitem[]` Droits d'accès ; voir Section 5.7 pour les détails

53.37. `pg_partitioned_table`

Le catalogue `pg_partitioned_table` enregistre des informations sur la façon dont les tables sont partitionnées.

Tableau 53.37. Colonnes de `pg_partitioned_table`

Type Description
`partrelid` `oid` (référence `pg_class.oid`) L'OID de la table partitionnée, référencé dans `pg_class`
`partstrat` `char` Stratégie de partitionnement ; `h` = partitionnement par hachage, `l` = partitionnement par liste, `r` = partitionnement par intervalles
`partnatts` `int2` Le nombre de colonnes de la clé de partitionnement
`partdefid` `oid` (référence `pg_class.oid`) L'OID de l'enregistrement dans `pg_class` pour la partition par défaut de cette table partitionnée ou zéro si cette table partitionnée n'a pas de partition par défaut.
`partattrs` `int2vector` (référence `pg_attribute.attnum`) Tableau de `partnatts` valeurs indiquant les colonnes de la table faisant partie de la clé de partitionnement. Par exemple, une valeur `1 3` signifierait que les première et troisième colonnes de la table forment la clé de partitionnement. Un zéro dans ce tableau indique que la colonne correspondante dans la clé partitionnement est une expression, plutôt qu'une simple référence de colonne.
`partclass` `oidvector` (référence `pg_opclass.oid`) Pour chaque colonne de la clé de partitionnement, ceci contient l'OID de la classe d'opérateurs à utiliser. Voir `pg_opclass` pour les détails.
`partcollation` `oidvector` (référence `pg_collation.oid`) Pour chaque colonne de la clé de partitionnement, ceci contient l'OID du collationnement à utiliser pour le partitionnement, ou zéro si la colonne n'est pas d'un type de données collationnable.
`partexprs` `pg_node_tree` Arbres d'expression (dans une représentation `nodeToString()`) pour les colonnes de la clé de partitionnement qui ne sont pas des simples références de colonne. C'est une liste avec un élément pour chaque élément 0 dans `partattrs`. NULL si tous les colonnes de la clé de partitionnement sont des références simples.

53.38. `pg_policy`

Le catalogue `pg_policy` stocke les politiques de sécurité niveau ligne pour les tables. Une politique inclue le type de commandes auquel elle s'applique (éventuellement toutes les commandes), les rôles

auxquels elle s'applique, l'expression à ajouter comme barrière de sécurité aux requêtes qui incluent la table, et l'expression à ajouter comme option `WITH CHECK` aux requêtes qui tentent d'ajouter de nouvelles lignes à la table.

Tableau 53.38. Colonnes de `pg_policy`

Type **Description**
`oid` `oid` Identifiant de ligne
`polname` `name` Le nom de la politique de sécurité
`polrelid` `oid` (référence `pg_class.oid`) La table à laquelle s'applique la politique de sécurité
`polcmd` `char` Le type de commande auquel est appliqué la politique de sécurité : `r` pour SELECT, `a` pour INSERT, `w` pour UPDATE, `d` pour DELETE ou `*` pour tous
`polpermissive` `bool` La politique est-elle permissive ou restrictive ?
`polroles` `oid[]` (référence `pg_authid.oid`) Les rôles à qui est appliquée la politique de sécurité ; zéro signifie `PUBLIC` (et apparaît normalement seule dans le tableau)
`polqual` `pg_node_tree` L'arbre de l'expression à ajouter aux barrières de sécurité pour les requêtes qui utilisent la table
`polwithcheck` `pg_node_tree` L'arbre de l'expression à ajouter aux qualifications WITH CHECK pour les requêtes qui tentent d'ajouter des lignes à la table

> **Note**
>
> Les politiques de sécurité stockées dans `pg_policy` sont seulement appliquées lorsque `pg_class.relrowsecurity` est positionné pour leur table.

53.39. `pg_proc`

Le catalogue `pg_proc` enregistre des informations sur les fonctions, procédures, fonctions d'agrégat et fonctions de fenêtrage (connues sous le nom collectif de routines). Voir CREATE FUNCTION, CREATE PROCEDURE et Section 38.3 pour plus d'informations.

Si `prokind` indique que l'enregistrement est pour une fonction d'agrégat, il devrait y avoir une ligne correspondante dans `pg_aggregate`.

Tableau 53.39. Colonnes de `pg_proc`

Type **Description**
`oid` `oid` Identifiant de ligne
`proname` `name` Nom de la fonction
`pronamespace` `oid` (référence `pg_namespace.oid`)

Type Description
OID du schéma (*namespace*) auquel appartient la fonction
`proowner` `oid` (référence `pg_authid.oid`) Propriétaire de la fonction
`prolang` `oid` (référence `pg_language.oid`) Langage de codage ou interface d'appel de la fonction
`procost` `float4` Coût d'exécution estimé (en unité de cpu_operator_cost) ; si `proretset`, il s'agit d'un coût par ligne
`prorows` `float4` Nombre estimé de lignes de résultat (zéro si `proretset` est faux)
`provariadic` `oid` (référence `pg_type.oid`) Type des données des éléments du tableau de paramètres variadic, ou zéro si la fonction n'a pas de paramètres variadiques
`prosupport` `regproc` (référence `pg_proc.oid`) Fonction optionnelle de support pour le planificateur (voir Section 38.11), ou zéro sinon
`prokind` `char` `f` pour une fonction normale, `p` pour une procédure, `a` pour une fonction d'agrégat ou `w` pour une fonction de fenêtrage
`prosecdef` `bool` Si vrai, la fonction définit la sécurité (c'est une fonction « setuid »)
`proleakproof` `bool` Cette fonction n'a pas d'effets de bord. Aucune information sur les arguments n'est reportée sauf via la valeur de retour. Toute fonction qui pourrait renvoyer une erreur dépendant des valeurs de ses arguments n'est pas considérée sans fuite.
`proisstrict` `bool` Si vrai, la fonction retourne NULL si l'un de ses arguments est NULL. Dans ce cas, la fonction n'est en fait pas appelée du tout. Les fonctions qui ne sont pas « strictes » doivent traiter les paramètres NULL.
`proretset` `bool` Si vrai, la fonction retourne un ensemble (c'est-à-dire des valeurs multiples du type défini)
`provolatile` `char` Indique si le résultat de la fonction dépend uniquement de ses arguments ou s'il est affecté par des facteurs externes. Il vaut `i` pour les fonctions « immuables », qui, pour un jeu de paramètres identique en entrée, donnent toujours le même résultat. Il vaut `s` pour les fonctions « stables », dont le résultat (pour les mêmes paramètres en entrée) ne change pas au cours du parcours (de table). Il vaut `v` pour les fonctions « volatiles », dont le résultat peut varier à tout instant. (`v` est également utilisé pour les fonctions qui ont des effets de bord, afin que les appels à ces fonctions ne soient pas optimisés.)
`proparallel` `char` `proparallel` indique si la fonction peut être utilisé en parallèle. Cette colonne vaut `s` pour les fonctions exécutables sans restriction en mode parallèle. Elle vaut `r` pour les fonctions pouvant être exécutées en parallèle mais leur exécution est restreinte au chef du groupe ; les processus de parallélisme en tâche de fond ne peuvent pas faire appel à ces fonctions. Elle vaut `u` pour les fonctions qui ne peuvent pas être exécutées en mode parallèle. La présence d'une telle fonction force une exécution non parallélisée.
`pronargs` `int2` Nombre d'arguments en entrée
`pronargdefaults` `int2`

Type Description
Nombre d'arguments qui ont des valeurs par défaut
`prorettype` `oid` (référence `pg_type.oid`) Type de données renvoyé
`proargtypes` `oidvector` (référence `pg_type.oid`) Un tableau contenant les types de données des arguments de la fonction. Ceci n'inclut que les arguments en entrée (dont les arguments `INOUT` et `VARIADIC`), et représente ainsi la signature d'appel de la fonction.
`proallargtypes` `oid[]` (référence `pg_type.oid`) Un tableau contenant les types de données des arguments de la fonction. Ceci inclut tous les arguments (y compris les arguments `OUT` et `INOUT`) ; néanmoins, si tous les arguments sont `IN`, ce champ est NULL. L'indice commence à 1 alors que, pour des raisons historiques, `proargtypes` commence à 0.
`proargmodes` `char[]` Un tableau contenant les modes des arguments de la fonction, codés avec `i` pour les arguments `IN`, `o` pour les arguments `OUT`, `b` pour les arguments `INOUT`, `v` pour les arguments `VARIADIC`, `t` pour les arguments `TABLE`. Si tous les arguments sont des arguments `IN`, ce champ est NULL. Les indices correspondent aux positions de `proallargtypes`, et non à celles de `proargtypes`.
`proargnames` `text[]` Un tableau contenant les noms des arguments de la fonction. Les arguments sans nom sont initialisés à des chaînes vides dans le tableau. Si aucun des arguments n'a de nom, ce champ est NULL. Les indices correspondent aux positions de `proallargtypes`, et non à celles de `proargtypes`.
`proargdefaults` `pg_node_tree` Arbres d'expression (en représentation `nodeToString()`) pour les valeurs par défaut. C'est une liste avec `pronargdefaults` éléments, correspondant aux *`N`* derniers arguments d'*entrée* (c'est-à-dire, les *`N`* dernières positions `proargtypes`). Si aucun des arguments n'a de valeur par défaut, ce champ vaudra null.
`protrftypes` `oid[]` (référence `pg_type.oid`) L'OID du type de données pour lequel les transformations s'appliquent. Un tableau de types de données argument/résultat pour lesquels appliquer les transformations (à partir de la clause `TRANSFORM` de la fonction). NULL sinon.
`prosrc` `text` Ce champ indique au gestionnaire de fonctions la façon d'invoquer la fonction. Il peut s'agir du code source pour un langage interprété, d'un symbole de lien, d'un nom de fichier ou de toute autre chose, en fonction du langage ou de la convention d'appel.
`probin` `text` Information supplémentaire sur la façon d'invoquer la fonction. Encore une fois, l'interprétation dépend du langage.
`prosqlbody` `pg_node_tree` Corps de fonction SQL pré-analysé. Ceci est utilisé pour les fonctions en langage SQL quand le corps est donné dans la notation du standard SQL plutôt que comme un litéral de chaîne. Il est NULL dans les autres cas.
`proconfig` `text[]` Configuration locale à la fonction pour les variables configurables à l'exécution
`proacl` `aclitem[]` Droits d'accès ; voir Section 5.7 pour plus de détails

Pour les fonctions compilées, natives ou chargées dynamiquement, `prosrc` contient le nom de la fonction en langage C(symbole de lien). Pour les fonctions en langage SQL, `prosrc` contient le texte source de la fonction si elle est indiquée sous la forme d'une chaîne litérale ; mais si le corps de la

fonction est indiquée dans le style du standard SQL, `prosrc` est inutilisé (typiquement, une chaîne vide) et `prosqlbody` contient la définition pré-analysée. Pour tous les autres types de langages, `prosrc` contient le code source de la fonction. `probin` est NULL, sauf pour les fonctions C chargées dynamiquement, pour lesquelles il donne le nom de fichier de la bibliothèque partagée qui contient la fonction.

53.40. `pg_publication`

Le catalogue `pg_publication` contient toutes les publications créées dans la base de données. Pour plus d'informations sur les publications, voir Section 31.1.

Tableau 53.40. Colonnes de `pg_publication`

Type Description
`oid` `oid` Identifiant de ligne
`pubname` `name` Nom de la publication
`pubowner` `oid` (référence `pg_authid.oid`) Propriétaire de la publication
`puballtables` `bool` Si true, cette publication inclut automatiquement toutes les tables de la base de données, ceci incluant aussi toutes les tables créées dans le futur.
`pubinsert` `bool` Si true, les opérations INSERT sont répliquées pour les tables de cette publication.
`pubupdate` `bool` Si true, les opérations UPDATE sont répliquées pour les tables de cette publication.
`pubdelete` `bool` Si true, les opérations DELETE sont répliquées pour les tables de cette publication
`pubtruncate` `bool` Si true, les opérations TRUNCATE sont répliquées pour les tables de la publication.
`pubviaroot` `bool` Si true, les opérations sur une partition feuille sont répliquées en utilisant l'identité et le schéma de son ancêtre mentionné dans la publication à la place de la sienne.

53.41. `pg_publication_namespace`

Le catalogue `pg_publication_namespace` contient la correspondance entre les schémas et les publications dans la base de données. C'est une correspondance plusieurs-à-plusieurs.

Tableau 53.41. `pg_publication_namespace` Columns

Type Description
`oid` `oid` Identifiant de ligne
`pnpubid` `oid` (references `pg_publication.oid`) Référence à la publication
`pnnspid` `oid` (references `pg_namespace.oid`) Référence au schéma

53.42. pg_publication_rel

Le catalogue pg_publication_rel contient la correspondance entre relations et publications dans la base de données. C'est une correspondance N-N (plusieurs à plusieurs). Voir aussi Section 54.17 pour une meilleure vision de cette information.

Tableau 53.42. Colonnes de pg_publication_rel

Type Description
oid oid Identifiant de ligne
prpubid oid (référence pg_publication.oid) Référence à la publication
prrelid oid (référence pg_class.oid) Référence à la relation
prqual pg_node_tree Arbre d'expression (dans la représentation de nodeToString() pour la condition qualifiante de la publication de la relation. NULL s'il n'y a pas de condition qualifiante de la publication.
prattrs int2vector (references pg_attribute.attnum) Tableau de valeurs qui indique les colonnes de table qui font partie de la publication. Par exemple, une valeur de 1 3 signifie que les première et troisième colonnes sont publiées. Une valeur NULL indique que toutes les colonnes sont publiées.

53.43. pg_range

Le catalogue pg_range enregistre des informations sur les types range. Ce sont des informations supplémentaires à celles déjà disponibles dans pg_type.

Tableau 53.43. Colonnes de pg_range

Type Description
rngtypid oid (référence pg_type.oid) OID du type range
rngsubtype oid (référence pg_type.oid) OID du type élément (sous-type) du type range
rngmultitypid oid (references pg_type.oid) OID du type multirange pour ce type range
rngcollation oid (référence pg_collation.oid) OID du collationnement utilisé pour les comparaisons d'intervalles, ou zéro si aucun
rngsubopc oid (référence pg_opclass.oid) OID de la classe d'opérateurs du sous-type, utilisée pour les comparaisons d'intervalles
rngcanonical regproc (référence pg_proc.oid) OID de la fonction de conversion d'une valeur range en sa forme canonique, ou zéro si aucune
rngsubdiff regproc (référence pg_proc.oid) OID de la fonction de renvoi de la différence entre deux valeurs d'éléments, sous la forme d'un double precision, ou zéro si aucune

rngsubopc (et rngcollation si le type de l'élément peut utilisé un collationnement) détermine l'ordre de tri utilisé par le type range. rngcanonical est utilisé quand le type de l'élément est discret.

`rngsubdiff` est optionnel mais doit être fourni pour améliorer les performances des index GiST sur le type range.

53.44. `pg_replication_origin`

Le catalogue `pg_replication_origin` contient toutes les origines de réplication créées. Pour plus d'informations sur les origines de réplication, voir Chapitre 50.

Contrairement à la plupart des catalogues systèmes, `pg_replication_origin` est partagé parmi toutes les bases de données d'une instance ; il n'existe qu'une seule copie de `pg_replication_origin` par instance, et non pas une par base.

Tableau 53.44. Colonnes de `pg_replication_origin`

Type Description
`roident` `oid` Un identifiant, unique pour l'instance, de l'origine de réplication. Ne devrait jamais quitter le système.
`roname` `text` Le nom externe, défini par l'utilisateur, d'une origine de réplication.

53.45. `pg_rewrite`

Le catalogue `pg_rewrite` stocke les règles de réécriture pour les tables et les vues.

Tableau 53.45. Colonnes de `pg_rewrite`

Type Description
`oid` `oid` Identifiant de ligne
`rulename` `name` Nom de la règle
`ev_class` `oid` (référence `pg_class.oid`) Table sur laquelle porte la règle
`ev_type` `char` Type d'événement associé à la règle : 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE
`ev_enabled` `char` Contrôle l'exécution de la règle suivant le mode session_replication_role. `O` = la règle se déclenche dans les modes « origin » et « local », `D` = la règle est désactivée, `R` = la règle s'exécute en mode « replica », `A` = la règle s'exécute à chaque fois.
`is_instead` `bool` Vrai s'il s'agit d'une règle `INSTEAD` (à la place de).
`ev_qual` `pg_node_tree` Arbre d'expression (sous la forme d'une représentation `nodeToString()`) pour la condition qualifiant la règle.
`ev_action` `pg_node_tree` Arbre de requête (sous la forme d'une représentation `nodeToString()`) pour l'action de la règle.

> **Note**
>
> `pg_class.relhasrules` doit être vrai si une table possède une règle dans ce catalogue.

53.46. `pg_seclabel`

Le catalogue `pg_seclabel` stocke les informations sur les labels de sécurité des objets de la base de données. Les labels de sécurité peuvent être manipulés avec la commande `SECURITY LABEL`. Pour visualiser plus facilement les labels de sécurité, voir Section 54.22.

Voir aussi `pg_shseclabel`, qui réaliser une opération similaire pour les labels de sécurité des objets globaux/partagés.

Tableau 53.46. Colonnes de `pg_seclabel`

Type Description
`objoid` `oid` (référence toute colonne OID) L'OID de l'objet concerné par ce label de sécurité
`classoid` `oid` (référence `pg_class.oid`) L'OID du catalogue système où cet objet apparaît
`objsubid` `int4` Pour un label de sécurité sur la colonne d'une table, cette colonne correspond au numéro de colonne (`objoid` et `classoid` font référence à la table elle-même). Pour tous les autres types d'objet, cette colonne vaut zéro.
`provider` `text` Le fournisseur du label associé avec ce label.
`label` `text` Le label de sécurité appliqué sur cet objet.

53.47. `pg_sequence`

Le catalogue `pg_sequence` contient des informations sur les séquences. Certaines de ces informations sur les séquences, comme le nom ou le schéma, sont dans `pg_class`.

Tableau 53.47. Colonnes de `pg_sequence`

Type Description
`seqrelid` `oid` (référence `pg_class.oid`) L'OID de l'enregistrement de cette séquence dans `pg_class`
`seqtypid` `oid` (référence `pg_type.oid`) Type de données de la séquence
`seqstart` `int8` Valeur de démarrage de la sequence
`seqincrement` `int8` Valeur d'incrément de la séquence
`seqmax` `int8` Valeur maximale de la séquence
`seqmin` `int8` Valeur minimale de la séquence

Type Description
`seqcache` `int8` Taille du cache de la séquence
`seqcycle` `bool` La séquence fait-elle un cycle

53.48. pg_shdepend

Le catalogue `pg_shdepend` enregistre les relations de dépendance entre les objets de la base de données et les objets partagés, comme les rôles. Cette information permet à PostgreSQL de s'assurer que tous ces objets sont déréférencés avant toute tentative de suppression.

Voir aussi `pg_depend`, qui réalise une fonction similaire pour les dépendances impliquant les objets contenus dans une seule base de données.

Contrairement à la plupart des catalogues système, `pg_shdepend` est partagé par toutes les bases de données d'un cluster : il n'existe qu'une seule copie de `pg_shdepend` par cluster, pas une par base de données.

Tableau 53.48. Colonnes de pg_shdepend

Type Description
`dbid` `oid` (référence `pg_database.oid`) L'OID de la base de données dont fait partie l'objet dépendant. ou 0 pour un objet partagé
`classid` `oid` (référence `pg_class.oid`) L'OID du catalogue système dont fait partie l'objet dépendant
`objid` `oid` (référence toute colonne OID) L'OID de l'objet dépendant
`objsubid` `int4` Pour une colonne de table, c'est le numéro de colonne (les `objid` et `classid` font référence à la table elle-même). Pour tous les autres types d'objets, cette colonne vaut zéro
`refclassid` `oid` (référence `pg_class.oid`) L'OID du catalogue système dont fait partie l'objet référencé (doit être un catalogue partagé)
`refobjid` `oid` (référence toute colonne OID) L'OID de l'objet référencé
`deptype` `char` Un code définissant les sémantiques spécifiques des relations de cette dépendance ; voir le texte.

Dans tous les cas, une entrée `pg_shdepend` indique que l'objet référencé ne peut pas être supprimé sans supprimer aussi l'objet dépendant. Néanmoins, il existe quelques différences identifiées par le `deptype` :

`SHARED_DEPENDENCY_OWNER` (`o`)

L'objet référencé (qui doit être un rôle) est le propriétaire de l'objet dépendant.

`SHARED_DEPENDENCY_ACL` (`a`)

L'objet référencé (qui doit être un rôle) est mentionné dans la liste de contrôle des accès (ACL, acronyme de *access control list*) de l'objet dépendant. (Une entrée `SHARED_DEPENDENCY_ACL`

n'est pas créée pour le propriétaire de l'objet car ce dernier a toujours une entrée SHARED_DEPENDENCY_OWNER.)

SHARED_DEPENDENCY_POLICY (r)

L'objet référencé (qui doit être un rôle) est mentionné comme la cible d'un objet de politique de sécurité dépendant.

SHARED_DEPENDENCY_TABLESPACE (t)

L'objet référencé (qui doit être un tablespace) est mentionné comme le tablespace pour une relation qui n'a pas de stockage.

D'autres types de dépendances peuvent s'avérer nécessaires dans le futur. La définition actuelle ne supporte que les rôles et les tablespaces comme objets référencés.

Comme dans le catalogue pg_depend, la plupart des objets créés pendant initdb sont considérés « fixes ». Aucun enregistrement n'est fait dans pg_shdepend pour des objets fixés soit en tant qu'objet référencée soit en tant qu'objet dépendant.

53.49. pg_shdescription

Le catalogue pg_shdescription stocke les descriptions optionelles (commentaires) des objets partagés de la base. Les descriptions peuvent être manipulées avec la commande COMMENT et visualisées avec les commandes \d de psql.

Voir aussi pg_description, qui assure les mêmes fonctions, mais pour les objets d'une seule base.

Contrairement à la plupart des catalogues systèmes, pg_shdescription est partagée par toutes les bases d'un cluster : il n'existe qu'une seule copie de pg_shdescription par cluster, et non une par base.

Tableau 53.49. Colonnes de pg_shdescription

Type Description
objoid oid (référence toute colonne OID) L'OID de l'objet concerné par la description
classoid oid (référence pg_class.oid) L'OID du catalogue système où cet objet apparaît
description text Texte arbitraire servant de description de l'objet

53.50. pg_shseclabel

Le catalogue pg_shseclabel stocke les labels de sécurité pour les objets partagés du serveur. Les labels de sécurité peuvent être manipulés avec la commande SECURITY LABEL. Pour une façon plus simple de voir les labels de sécurité, voir Section 54.22.

Voir aussi pg_seclabel, qui réalise une fonction similaire pour les labels de sécurité sur des objets internes à une base de données.

Contrairement à la plupart des catalogues systèmes, pg_shseclabel est partagé entre toutes les bases de données d'une instance ; il n'existe qu'une copie de pg_shseclabel par instance, et non pas une par base de données.

Tableau 53.50. Colonnes de pg_shseclabel

Type Description
objoid oid (référence toute colonne OID) L'OID de l'objet auquel s'applique ce label de sécurité
classoid oid (référence pg_class.oid) L'OID du catalogue système où cet objet apparaît
provider text Le fournisseur de label associé avec ce label
label text Le label de sécurité appliqué à cet objet

53.51. pg_statistic

Le catalogue pg_statistic stocke des données statistiques sur le contenu de la base de données. Les entrées sont créées par ANALYZE, puis utilisées par le planificateur de requêtes. Les données statistiques sont, par définition des approximations, même si elles sont à jour.

D'habitude, il existe une entrée, avec stainherit = false, pour chaque colonne de table qui a été analysée. Si la table a des enfants ou des partitions, une seconde entrée avec stainherit = true est aussi créé. Cette ligne représente les statistiques de la colonne sur l'arbre d'héritage, autrement dit les statistiques pour les données que vous voyez avec SELECT *colonne* FROM *table**, alors que la ligne stainherit = false représente le résultat de SELECT *column* FROM ONLY *table*.

pg_statistic stocke aussi les données statistiques des valeurs des expressions d'index. Elles sont décrites comme si elles étaient de vraies colonnes ; en particulier, starelid référence l'index. Néanmoins, aucune entrée n'est effectuée pour une colonne d'index ordinaire sans expression car cela est redondant avec l'entrée correspondant à la colonne sous-jacente de la table. Actuellement, les entrées pour les expressions d'index ont toujours stainherit = false.

Comme des statistiques différentes peuvent être appropriées pour des types de données différents, pg_statistic ne fait qu'un minimum de suppositions sur les types de statistiques qu'il stocke. Seules des statistiques extrêmement générales (comme les valeurs NULL) ont des colonnes dédiées. Tout le reste est stocké dans des « connecteurs », groupes de colonnes associées dont le contenu est identifié par un numéro de code dans l'une des colonnes du connecteur. Pour plus d'information, voir src/include/catalog/pg_statistic.h.

pg_statistic ne doit pas être lisible par le public, car même les données statistiques sont sensibles. (Exemple : les valeurs maximales et minimales d'une colonne de salaire peuvent être intéressantes). pg_stats est une vue sur pg_statistic accessible à tous, qui n'expose que les informations sur les tables accessibles à l'utilisateur courant.

Tableau 53.51. Colonnes de pg_statistic

Type Description
starelid oid (référence pg_class.oid) Table ou index à qui la colonne décrite appartient
staattnum int2 (référence pg_attribute.attnum) Numéro de la colonne décrite
stainherit bool Si vrai, les statistiques incluent les valeurs des colonnes enfants de l'héritage, pas uniquement les valeurs de la relation spécifiée

Type Description
`stanullfrac` `float4` Fraction des entrées de la colonne qui ont une valeur NULL
`stawidth` `int4` Taille moyenne, en octets, des entrées non NULL
`stadistinct` `float4` Nombre de valeurs distinctes non NULL dans la colonne. Une valeurs positive est le nombre réel de valeurs distinctes. Une valeur négative est le négatif d'un multiplieur pour le nombre de lignes dans la table ; par exemple, une colonne dans laquelle 90% des lignes ne sont pas NULL et dans laquelle chaque valeur non NULL apparaît deux fois en moyenne, pourrait être représentée avec un `stadistinct` à -0,4.
`stakind`*`N`* `int2` Numéro de code indiquant le type de statistiques stockées dans « le connecteur » numéro *`N`* de la ligne de `pg_statistic`.
`staop`*`N`* `oid` (référence `pg_operator.oid`) Opérateur utilisé pour dériver les statistiques stockées dans « le connecteur » numéro *`N`*. Par exemple, un connecteur d'histogramme montre l'opérateur `<`, qui définit l'ordre de tri des données. Zéro si le type de statistique ne requiert pas un opérateur.
`stacoll`*`N`* `oid` (référence `pg_collation.oid`) Le collationnement utilisé pour dériver les statistiques enregistrées dans le *`N`*-ième « slot ». Par exemple, un slot histogramme pour une colonne collationnable afficherait le collationnement qui définit l'ordre de tri de la donnée. Zéro pour les données sans collationnement.
`stanumbers`*`N`* `float4[]` Statistiques numériques du type approprié pour « le connecteur » numéro *`N`* ou NULL si le type de connecteur n'implique pas de valeurs numériques.
`stavalues`*`N`* `anyarray` Valeurs de données de la colonne du type approprié pour « le connecteur » numéro *`N`* ou NULL si le type de connecteur ne stocke aucune valeur de données. Chaque valeur d'élément du tableau est en fait du type de données de la colonne indiquée, ou un type en relation comme un type élément d'un tableau, si bien qu'il n'y a aucun moyen de définir le type de ces colonnes plus précisément qu'avec le type `anyarray` (tableau quelconque).

53.52. `pg_statistic_ext`

Le catalogue `pg_statistic_ext` contient des définitions des statistiques étendues pour l'optimiseur de requêtes. Chaque ligne de ce catalogue correspond à un *objet statistique* créé avec `CREATE STATISTICS`.

Normalement, il existe un enregistrement, avec `stxdinherit = false`, pour chaque objet statistique qui a été analysé. Si la table a un fils dans un héritage ou une partition, un deuxième enregistrelent avec `stxdinherit = true` est aussi créé. Cette ligne représente l'objet statistique dansl'arbre d'héritage, par exemple les statistiques pour les données que vous verriez avec `SELECT * FROM` *`table*`*, alors qu'une ligne `stxdinherit = false` représente le résultat de `SELECT * FROM ONLY` *`table`*.

Tout comme `pg_statistic`, `pg_statistic_ext_data` ne doit pas être lisible par tout le monde car son contenu peut être considéré comme sensible. (Par exemple, les combinaisons de valeurs les plus communes des colonnes pourraient être intéressantes.) `pg_stats_ext` est une vue, lisible par tout le monde, interrogeant `pg_statistic_ext` (après une jointure sur `pg_statistic_ext`) en exposant uniquement les informations sur les tables dont l'utilisateur courant est le propriétaire.

Tableau 53.52. Colonnes de `pg_statistic_ext`

Type Description
`oid` `oid` Identifiant de ligne
`stxrelid` `oid` (référence `pg_class.oid`) Table contenant les colonnes décrites par cet objet
`stxname` `name` Nom de l'objet statistique
`stxnamespace` `oid` (référence `pg_namespace.oid`) OID du schéma contenant l'objet statistique
`stxowner` `oid` (référence `pg_authid.oid`) Propriétaire de l'objet statistique
`stxstattarget` `int4` `stxstattarget` contrôle le niveau de détail des statistiques accumulées pour cet objet statistique par `ANALYZE`. Une valeur zéro indique qu'aucune statistique ne doit être récupérée. Une valeur négative indique d'utiliser le maximum des cibles statistiques des colonnes référencées, si elles sont configurés, ou la cible par défaut du système. Les valeurs positives de `stxstattarget` déterminent le nombre cible de « valeurs les plus communes » à récupérer.
`stxkeys` `int2vector` (référence `pg_attribute.attnum`) Un tableau de numéros de colonnes, indiquant les colonnes de la table couvertes par l'objet statistique ; par exemple, une valeur `1 3` signifierait que les première et troisième colonnes de la table sont couvertes
`stxkind` `char[]` Un tableau contenant des codes pour les types statistiques activés ; les valeurs valides sont : `d` pour des statistiques n-distinct, `f` pour des statistiques de dépendance fonctionnelle, `m` pour des statistiques sur les valeurs les plus communes (*most common values* ou MCV) et `e` pour des statistiques sur des expressions
`stxexprs` `pg_node_tree` Arbres d'expression (dans la représentation `nodeToString()`) pour les attributs des objets statistiques qui ne sont pas de simples références de colonnes. C'est une liste avec une un élément par expression. Null si tous les attributs des objets statistiques sont des références simples.

Un enregistrement de `pg_statistic_ext` est rempli complètement lors du `CREATE STATISTICS`, mais les vraies valeurs statistiques ne sont pas calculées à ce moment-là. Les commandes `ANALYZE` ultérieures calculent les valeurs souhaitées et peuplent l'enregistrement dans `pg_statistic_ext_data`.

53.53. `pg_statistic_ext_data`

Le catalogue `pg_statistic_ext_data` contient les données pour les statistiques étendues de l'optimiseur, définies dans `pg_statistic_ext`. Chaque ligne de ce catalogue correspond à un *objet statistique* créé avec `CREATE STATISTICS`.

Tableau 53.53. Colonnes de `pg_statistic_ext_data`

Type Description
`stxoid` `oid` (référence `pg_statistic_ext.oid`) Objet statistique étendu contenant la définition pour cette donnée
`stxdinherit` `bool`

Type Description
Si true, les statistiques incluent des valeurs des tables filles, pas seulement les valeurs dans la relation indiquée
`stxdndistinct` `pg_ndistinct` Nombre de valeurs distinctes, sérialisé sous la forme d'un type `pg_ndistinct`
`stxddependencies` `pg_dependencies` Statistiques de dépendance fonctionnelle, sérialisées sous la forme d'un type `pg_dependencies`
`stxdmcv` `pg_mcv_list` Statistiques MCV (valeurs les plus communes), sérialisées sous la forme d'un type `pg_mcv_list`.
`stxdexpr` `pg_statistic[]` Statistiques par expression, sérialisées sous la forme d'un tableau de type `pg_statistic`

53.54. `pg_subscription`

Le catalogue `pg_subscription` contient toutes les souscriptions existantes pour la réplication logique. Pour plus d'informations sur la réplication logique, voir Chapitre 31.

Contrairement à la plupart des catalogues systèmes, `pg_subscription` est partagé parmi toutes les bases de données d'une instance. Il existe une seule copie de `pg_subscription` par instance, et non pas une par base de données.

L'accès à la colonne `subconninfo` est interdite aux utilisateurs standards car elle pourrait contenir des mots de passe en clair.

Tableau 53.54. Colonnes de `pg_subscription`

Type Description
`oid` `oid` Identifiant de la ligne (attribut caché ; doit être sélectionné explicitement)
`subdbid` `oid` (référence `pg_database.oid`) OID de la base de données où réside la souscription
`subskiplsn` `pg_lsn` LSN de fin de la transaction dont les changements doivent être ignorés si le LSN est valide ; sinon `0/0`.
`subname` `name` Nom de la souscription
`subowner` `oid` (référence `pg_authid.oid`) Propriétaire de la souscription
`subenabled` `bool` Si true, la souscription est activée et doit répliquer
`subbinary` `bool` Si true, la souscription demandera que le publieur envoie des données dans le format binaire
`substream` `bool` Si true, la souscription permettra le flux de transactions en cours
`subtwophasestate` `char` Code d'état pour le mode de validation en deux phases : `d` = désactivé, `p` = activation en cours, `e` = activé

Type Description
subdisableonerr bool Si true, la souscription sera désactivée si un des workers détecte une erreur
subconninfo text Chaîne de connexion vers la base de données source
subslotname name Nom du slot de réplication dans la base de données source (aussi utilisé pour le nom origine de la réplication locale) ; null représente NONE
subsynccommit text Contient la valeur du paramètre synchronous_commit pour les processus workers de la souscription.
subpublications text[] Tableau de noms de publications souscrites. Ceci référence les publications définies dans la base d'origine. Pour plus d'informations sur les publications, voir Section 31.1.

53.55. pg_subscription_rel

Le catalogue pg_subscription_rel contient l'état de chaque relation répliquée dans chaque souscription. C'est une correspondance N-N (plusieurs à plusieurs).

Ce catalogue contient seulement les tables connues à la souscription après exécution de CREATE SUBSCRIPTION ou ALTER SUBSCRIPTION ... REFRESH PUBLICATION.

Tableau 53.55. Colonnes de pg_subscription_rel

Type Description
srsubid oid (référence pg_subscription.oid) Référence à la souscription
srrelid oid (référence pg_class.oid) Référence à la relation
srsubstate char Code d'état : i = initialisation, d = données en cours de copie, f = copie de la table terminée, s = synchronisée, r = prête (réplication normale)
srsublsn pg_lsn LSN distant du changement d'état utilisé pour la coordination de la synchronisation pour les états s ou r, sinon NULL

53.56. pg_tablespace

Le catalogue pg_tablespace enregistre les informations des *tablespaces* disponibles. Les tables peuvent être placées dans des *tablespaces* particuliers pour faciliter l'administration des espaces de stockage.

Contrairement à la plupart des catalogues système, pg_tablespace est partagée par toutes les bases de données du cluster : il n'y a donc qu'une copie de pg_tablespace par cluster, et non une par base.

Tableau 53.56. Colonnes de pg_tablespace

Type Description
oid oid Identifiant de ligne

Type Description
`spcname` `name` Nom du *tablespace*
`spcowner` `oid` (référence `pg_authid.oid`) Propriétaire du *tablespace*, habituellement l'utilisateur qui l'a créé
`spcacl` `aclitem[]` Droits d'accès ; voir Section 5.7 pour les détails.
`spcoptions` `text[]` Options au niveau tablespace, sous la forme de chaînes « motclé=valeur »

53.57. `pg_transform`

Le catalogue `pg_transform` stocke des informations à propos des transformations, qui sont un mécanisme pour adapter des types de données aux langages procéduraux. Voir CREATE TRANSFORM pour plus d'informations.

Tableau 53.57. Colonnes de `pg_transform`

Type Description
`oid` `oid` Identifiant de ligne
`trftype` `oid` (référence `pg_type.oid`) L'OID du type de donnée auquel cette transformation s'applique
`trflang` `oid` (référence `pg_language.oid`) L'OID du langage auquel cette transformation s'applique
`trffromsql` `regproc` (référence `pg_proc.oid`) L'OID de la fonction à utiliser pour la conversion du type de données envoyé au langage procédural (par exemple, les paramètres de la fonction). Zéro est stocké si le comportement par défaut doit être utilisé.
`trftosql` `regproc` (référence `pg_proc.oid`) L'OID de la fonction à utiliser pour convertir les sorties du langage procédural (par exemple, les valeurs de retour) vers le type de données. Zéro est stocké si le comportement par défaut doit être utilisé.

53.58. `pg_trigger`

Le catalogue `pg_trigger` stocke les informations concernant les triggers des tables et des vues. Voir la commande CREATE TRIGGER pour plus d'informations.

Tableau 53.58. Colonnes de `pg_trigger`

Type Description
`oid` `oid` Identifiant de ligne
`tgrelid` `oid` (référence `pg_class.oid`) Table sur laquelle porte le trigger
`tgparentid` `oid` (référence `pg_trigger.oid`) Trigger parent à partir duquel ce trigger est cloné (ceci survient quand les partitions sont créées ou attachées à une table partitionnée), zéro si ce n'est pas un clone.

Type Description
`tgname` `name` Nom du trigger (doit être unique parmi les triggers d'une table)
`tgfoid` `oid` (référence `pg_proc.oid`) Fonction à appeler
`tgtype` `int2` Masque de bits identifiant les conditions de déclenchement
`tgenabled` `char` Contrôle l'exécution du trigger suivant le mode session_replication_role. `O` = le trigger se déclenche dans les modes « origin » et « local », `D` = le trigger est désactivé, `R` = le trigger s'exécute en mode « replica », `A` = le trigger s'exécute à chaque fois.
`tgisinternal` `bool` Vrai si le trigger est généré en interne (habituellement pour forcer la contrainte identifiée par `tgconstraint`)
`tgconstrrelid` `oid` (référence `pg_class.oid`) La table référencée par une contrainte d'intégrité référentielle (zéro si le trigger n'a pas pour but une contrainte d'intégrité référentielle)
`tgconstrindid` `oid` (référence `pg_class.oid`) L'index supportant une contrainte unique, clé primaire, clé d'intégrité référentielle, contrainte d'exclusion (zéro si le trigger ne concerne pas un de ces types de contrainte)
`tgconstraint` `oid` (référence `pg_constraint.oid`) L'entrée `pg_constraint` associé au trigger, si elle existe
`tgdeferrable` `bool` Vrai si le trigger contrainte est retardable
`tginitdeferred` `bool` Vrai si le trigger de contrainte est initialement retardé
`tgnargs` `int2` Nombre de chaînes d'arguments passées à la fonction du trigger
`tgattr` `int2vector` (référence `pg_attribute.attnum`) numéros de colonne, si le trigger est spécifique à la colonne ; sinon un tableau vide
`tgargs` `bytea` Chaînes d'arguments à passer au trigger, chacune terminée par un NULL
`tgqual` `pg_node_tree` Arbre d'expression (d'après la représentation de `nodeToString()` pour la condition `WHEN` du trigger, ou NULL si aucune
`tgoldtable` `name` Nom de la clause `REFERENCING` pour `OLD TABLE`, ou NULL si aucun
`tgnewtable` `name` Nom de la clause `REFERENCING` pour `NEW TABLE`, ou NULL si aucun

Actuellement, les triggers spécifiques par colonne sont supportés seulement pour les événements `UPDATE` et, du coup, `tgattr` est valable seulement pour ce type d'événements. `tgtype` pourrait contenir des informations pour d'autres types d'événement mais ils sont supposés valides pour la table complète, quel que soit le contenu de `tgattr`.

Note

Quand `tgconstraint` est différent de zéro, `tgconstrrelid`, `tgconstrindid`, `tgdeferrable` et `tginitdeferred` sont grandement redondants avec l'entrée référencée de `pg_constraint`. Néanmoins, il est possible qu'un trigger non déferrable soit

associé à une contrainte déferrable : les contraintes de clé étrangère peuvent avoir quelques triggers déferrables et quelques triggers non déferrables.

Note

`pg_class.relhastriggers` doit valoir true si la relation possède au moins un trigger dans ce catalogue.

53.59. `pg_ts_config`

Le catalogue `pg_ts_config` contient des entrées représentant les configurations de la recherche plein texte. Une configuration spécifie un analyseur et une liste de dictionnaires à utiliser pour chacun des types d'éléments en sortie de l'analyseur. L'analyseur est présenté dans l'entrée de `pg_ts_config` mais la correspondance élément/dictionnaire est définie par des entrées supplémentaires dans `pg_ts_config_map`.

Les fonctionnalités de recherche plein texte de PostgreSQL sont expliquées en détail dans Chapitre 12.

Tableau 53.59. Colonnes de `pg_ts_config`

Type **Description**
`oid` `oid` Identifiant de ligne
`cfgname` `name` Nom de la configuration
`cfgnamespace` `oid` (référence `pg_namespace.oid`) OID du *namespace* qui contient la configuration
`cfgowner` `oid` (référence `pg_authid.oid`) Propriétaire de la configuration
`cfgparser` `oid` (référence `pg_ts_parser.oid`) OID de l'analyseur pour la configuration

53.60. `pg_ts_config_map`

Le catalogue `pg_ts_config_map` contient des entrées présentant les dictionnaires de recherche plein texte à consulter et l'ordre de consultation, pour chaque type de lexème en sortie de chaque analyseur de configuration.

Les fonctionnalités de la recherche plein texte de PostgreSQL sont expliquées en détail dans Chapitre 12.

Tableau 53.60. Colonnes de `pg_ts_config_map`

Type **Description**
`mapcfg` `oid` (référence `pg_ts_config.oid`) OID de l'entrée `pg_ts_config` qui possède l'entrée
`maptokentype` `int4` Un type de lexème émis par l'analyseur de configuration
`mapseqno` `int4` Ordre dans lequel consulter l'entrée (les plus petits `mapseqno` en premier)

Type Description
`mapdict` `oid` (référence `pg_ts_dict.oid`) OID du dictionnaire de recherche plein texte à consulter

53.61. `pg_ts_dict`

Le catalogue `pg_ts_dict` contient des entrées définissant les dictionnaires de recherche plein texte. Un dictionnaire dépend d'un modèle de recherche plein texte qui spécifie toutes les fonctions d'implantation nécessaires ; le dictionnaire lui-même fournit des valeurs pour les paramètres utilisateur supportés par le modèle. Cette division du travail permet la création de dictionnaires par des utilisateurs non privilégiés. Les paramètres sont indiqués par une chaîne, `dictinitoption`, dont le format et la signification dépendent du modèle.

Les fonctionnalités de la recherche plein texte de PostgreSQL sont expliquées en détail dans Chapitre 12.

Tableau 53.61. Colonnes de `pg_ts_dict`

Type Description
`oid` `oid` Identifiant de ligne
`dictname` `name` Nom du dictionnaire de recherche plein texte
`dictnamespace` `oid` (référence `pg_namespace.oid`) OID du *namespace* contenant le dictionnaire
`dictowner` `oid` (référence `pg_authid.oid`) Propriétaire du dictionnaire
`dicttemplate` `oid` (référence `pg_ts_template.oid`) OID du modèle de recherche plein texte du dictionnaire
`dictinitoption` `text` Chaîne d'options d'initialisation du modèle

53.62. `pg_ts_parser`

Le catalogue `pg_ts_parser` contient des entrées définissant les analyseurs de la recherche plein texte. Un analyseur est responsable du découpage du texte en entrée en lexèmes et de l'assignation d'un type d'élément à chaque lexème. Puisqu'un analyseur doit être codé à l'aide de fonctions écrites en langage C, la création de nouveaux analyseurs est restreinte aux superutilisateurs des bases de données.

Les fonctionnalités de la recherche plein texte de PostgreSQL sont expliquées en détail dans Chapitre 12.

Tableau 53.62. Colonnes de `pg_ts_parser`

Type Description
`oid` `oid` Identifiant de ligne
`prsname` `name` Nom de l'analyseur de recherche plein texte
`prsnamespace` `oid` (référence `pg_namespace.oid`) OID du *namespace* qui contient l'analyseur

Type Description
`prsstart` `regproc` (référence `pg_proc.oid`) OID de la fonction de démarrage de l'analyseur (zéro si aucune)
`prstoken` `regproc` (référence `pg_proc.oid`) OID de la fonction de récupération du jeton suivant de l'analyseur
`prsend` `regproc` (référence `pg_proc.oid`) OID de la fonction d'arrêt de l'analyseur
`prsheadline` `regproc` (référence `pg_proc.oid`) OID de la fonction headline de l'analyseur (zéro si aucune)
`prslextype` `regproc` (référence `pg_proc.oid`) OID de la fonction lextype de l'analyseur

53.63. `pg_ts_template`

Le catalogue `pg_ts_template` contient des entrées définissant les modèles de recherche plein texte. Un modèle est le squelette d'implantation d'une classe de dictionnaires de recherche plein texte. Puisqu'un modèle doit être codé à l'aide de fonctions codées en langage C, la création de nouveaux modèles est restreinte aux superutilisateurs des bases de données.

Les fonctionnalités de la recherche plein texte de PostgreSQL sont expliquées en détail dans Chapitre 12.

Tableau 53.63. Colonnes de `pg_ts_template`

Type Description
`oid` `oid` Identifiant de ligne
`tmplname` `name` Nom du modèle de recherche plein texte
`tmplnamespace` `oid` (référence `pg_namespace.oid`) OID du *namespace* qui contient le modèle
`tmplinit` `regproc` (référence `pg_proc.oid`) OID de la fonction d'initialisation du modèle (zéro si aucune)
`tmpllexize` `regproc` (référence `pg_proc.oid`) OID de la fonction lexize du modèle

53.64. `pg_type`

Le catalogue `pg_type` stocke les informations concernant les types de données. Les types basiques et d'énumération (types scalaires) sont créés avec la commande `CREATE TYPE` et les domaines avec `CREATE DOMAIN`. Un type composite est créé automatiquement pour chaque table de la base pour représenter la structure des lignes de la table. Il est aussi possible de créer des types composites avec `CREATE TYPE AS`.

Tableau 53.64. Colonnes de `pg_type`

Type Description
`oid` `oid` Identifiant de ligne
`typname` `name`

Type **Description**
Nom du type
`typnamespace` `oid` (référence `pg_namespace.oid`) OID du *namespace* qui contient le type
`typowner` `oid` (référence `pg_authid.oid`) Propriétaire du type
`typlen` `int2` Pour les types de taille fixe, `typlen` est le nombre d'octets de la représentation interne du type. Mais pour les types de longueur variable, `typlen` est négatif. -1 indique un type « varlena » (qui a un attribut de longueur), -2 indique une chaîne C terminée par le caractère NULL.
`typbyval` `bool` `typbyval` détermine si les routines internes passent une valeur de ce type par valeur ou par référence. `typbyval` doit être faux si `typlen` ne vaut pas 1, 2 ou 4 (ou 8 sur les machines dont le mot-machine est de 8 octets). Les types de longueur variable sont toujours passés par référence. `typbyval` peut être faux même si la longueur permet un passage par valeur.
`typtype` `char` `typtype` vaut `b` pour un type de base, `c` pour un type composite (le type d'une ligne de table, par exemple), `d` pour un domaine, `e` pour un enum, `p` pour un pseudo-type, `r` pour un type range ou `m` pour un type multirange. Voir aussi `typrelid` et `typbasetype`.
`typcategory` `char` `typcategory` est une classification arbitraire de types de données qui est utilisée par l'analyseur pour déterminer la conversion implicite devant être « préférée ». Voir Tableau 53.65
`typispreferred` `bool` Vrai si ce type est une cible de conversion préférée dans sa `typcategory`
`typisdefined` `bool` Vrai si le type est défini et faux s'il ne s'agit que d'un conteneur pour un type qui n'est pas encore défini. Lorsque `typisdefined` est faux, rien, à part le nom du type, le *namespace* et l'OID, n'est fiable.
`typdelim` `char` Caractère qui sépare deux valeurs de ce type lorsque le programme lit les valeurs d'un tableau en entrée. Le délimiteur est associé au type d'élément du tableau, pas au type tableau.
`typrelid` `oid` (référence `pg_class.oid`) S'il s'agit d'un type composite (voir `typtype`), alors cette colonne pointe vers la ligne de `pg_class` qui définit la table correspondante. Pour un type composite sans table, l'entrée dans `pg_class` ne représente pas vraiment une table, mais elle est néanmoins nécessaire pour trouver les lignes de `pg_attribute` liées au type. 0 pour les types autres que composites.
`typsubscript` `regproc` (references `pg_proc.oid`) OID de la fonction du gestionnaire des indices, ou zéro si ce type ne supporte pas les indices. Les types qui sont des vrais types tableau ont `typsubscript` = `array_subscript_handler`, mais les autres types pourraient avoir d'autres fonctions de gestion pour implémenter le comportement spécialisé d'indices.
`typelem` `oid` (référence `pg_type.oid`) Si `typelem` est différent de zéro, alors il identifie une autre ligne de `pg_type`, définissant le type renvoyé par indice. Ceci devrait valoir zéro si `typsubscript` vaut zéro. Néanmoins, il peut valoir zéro quand `typsubscript` ne vaut pas zéro, si le gestionnaire n'a pas besoin de `typelem` pur déterminer le type de résultat avec indice. Notez qu'une dépendance `typelem` est considéré impliqué un contenu physique du

Type Description
type élément dans ce type ; donc les modifications DDL sur le type élément pourrait être restreint par la présence de ce type.
`typarray` `oid` (référence `pg_type.oid`) Si `typarray` est différent de zéro, alors il identifie une autre ligne dans `pg_type`, qui est le type tableau « true » disposant de ce type en élément.
`typinput` `regproc` (référence `pg_proc.oid`) Fonction de conversion en entrée (format texte)
`typoutput` `regproc` (référence `pg_proc.oid`) Fonction de conversion en sortie (format texte)
`typreceive` `regproc` (référence `pg_proc.oid`) Fonction de conversion en entrée (format binaire), ou zéro s'il n'y en a pas
`typsend` `regproc` (référence `pg_proc.oid`) Fonction de conversion en sortie (format binaire), ou zéro s'il n'y en a pas
`typmodin` `regproc` (référence `pg_proc.oid`) Fonction en entrée de modification du type ou zéro si le type ne supporte pas les modificateurs
`typmodout` `regproc` (référence `pg_proc.oid`) Fonction en sortie de modification du type ou zéro pour utiliser le format standard
`typanalyze` `regproc` (référence `pg_proc.oid`) Fonction ANALYZE personnalisée ou zéro pour utiliser la fonction standard
`typalign` `char` `typalign` est l'alignement requis pour stocker une valeur de ce type. Cela s'applique au stockage sur disque ainsi qu'à la plupart des représentations de cette valeur dans PostgreSQL. Lorsque des valeurs multiples sont stockées consécutivement, comme dans la représentation d'une ligne complète sur disque, un remplissage est inséré avant la donnée de ce type pour qu'elle commence à l'alignement indiqué. La référence de l'alignement est le début de la première donnée de la séquence. Les valeurs possibles sont : • `c` = alignement `char`, aucun alignement n'est nécessaire ; • `s` = alignement `short` (deux octets sur la plupart des machines) ; • `i` = alignement `int` (quatre octets sur la plupart des machines) ; • `d` = alignement `double` (huit octets sur la plupart des machines, mais pas sur toutes).
`typstorage` `char` `typstorage` indique, pour les types varlena (ceux pour lesquels `typlen` = -1), si le type accepte le TOASTage et la stratégie par défaut à utiliser pour les attributs de ce type. Les valeurs possibles sont : • `p` (*plain*) : la valeur doit être stockée normalement (non-varlena types always use this value) ; • `e` (*external* : la valeur peut être stockée dans une relation « secondaire » « TOAST » (si la relation en a une, voir `pg_class.reltoastrelid`) ; • `m` (*main*) : la valeur peut être stockée compressée sur place ; • `x` (*extended*) : la valeur peut être stockée compressée sur place ou stockée dans une relation « secondaire ».

Type Description
x is the usual choice for toast-able types. Les colonnes m peuvent aussi être déplacées dans une table de stockage secondaire, mais seulement en dernier recours (les colonnes e et x sont déplacées les premières).
typnotnull bool Représente une contrainte non NULL.
typbasetype oid (référence pg_type.oid) S'il s'agit d'un domaine (voir typtype), alors typbasetype identifie le type sur lequel celui-ci est fondé. 0 s'il ne s'agit pas d'un domaine.
typtypmod int4 Les domaines utilisent ce champ pour enregistrer le typmod à appliquer à leur type de base (-1 si le type de base n'utilise pas de typmod). -1 si ce type n'est pas un domaine.
typndims int4 Le nombre de dimensions de tableau pour un domaine sur un tableau (c'est-à-dire dont typbasetype est un type tableau). 0 pour les types autres que les domaines sur des types tableaux.
typcollation oid (référence pg_collation.oid) typcollation spécifie le collationnement du type. Si le type ne supporte pas les collationnemens, cette colonne vaut zéro. Un type de base qui supporte les collationnements aura une valeur différente de zéro, habituellement DEFAULT_COLLATION_OID ici. Un domaine sur un type collationnable peut avoir un OID de collationnement différent de celui du type de base si ce dernier a été précisé pour le domaine.
typdefaultbin pg_node_tree Si typdefaultbin n'est pas NULL, ce champ est la représentation nodeToString() d'une expression par défaut pour le type. Ceci n'est utilisé que pour les domaines.
typdefault text NULL si le type n'a pas de valeur par défaut associée. Si typdefaultbin est non NULL, ce champ doit contenir une version lisible de l'expression par défaut représentée par typdefaultbin. Si typdefaultbin est NULL et si ce champ ne l'est pas, alors il stocke la représentation externe de la valeur par défaut du type, qui peut être passée à la fonction de conversion en entrée du type pour produire une constante.
typacl aclitem[] Droits d'accès ; voir Section 5.7 pour les détails

> **Note**
>
> Pour les types de données à largeur fixe utilisés dans les tables systèmes, il est critique que la taille et l'alignement définis dans pg_type soient en accord avec la façon dont le compilateur placera la colonne dans une structure représentant une ligne de la table.

Tableau 53.65 liste les valeurs de typcategory définies par le système. Tout ajout futur à la liste sera aussi une lettre ASCII majuscule. Tous les autres caractères ASCII sont réservés pour les catégories définies par l'utilisateur.

Tableau 53.65. Codes typcategory

Code	Catégorie
A	Types tableaux
B	Types booléens

Code	Catégorie
C	Types composites
D	Types date/time
E	Types enum
G	Types géometriques
I	Types adresses réseau
N	Types numériques
P	Pseudo-types
S	Types chaînes
R	Types range
T	Types 'Timespan' (étendue de temps, intervalle)
U	Types définis par l'utilisateur
V	Types Bit-string
X	Type unknown
Z	Types à usage interne

53.65. pg_user_mapping

Le catalogue pg_user_mapping stocke les correspondances entre utilisateurs locaux et distants. L'accès à ce catalogue est interdite aux utilisateurs normaux, utilisez la vue pg_user_mappings à la place.

Tableau 53.66. Colonnes de pg_user_mapping

Type Description
oid oid Identifiant de ligne
umuser oid (référence pg_authid.oid) OID du rôle à faire correspondre, zéro si l'utilisateur à correspondre est public
umserver oid (référence pg_foreign_server.oid) L'OID du serveur distant qui contient cette correspondance
umoptions text[] Options spécifiques à la correspondance d'utilisateurs, sous forme de chaînes « motclé=valeur ». Cette colonne s'affichera comme NULL sauf si l'utilisateur courant est celui en cours de correspondance ou si la correspondance concerne PUBLIC et que l'utilisateur courant est le propriétaire du serveur ou que l'utilisateur courant est un superutilisateur. Le but est de protéger les informations sur les mots de passe, enregistrées dans une option de la correspondance d'utilisateur.

Chapitre 54. Vues système

En plus des catalogues système, PostgreSQL fournit un certain nombre de vues internes. Certaines fournissent un moyen simple d'accéder à des requêtes habituellement utilisées dans les catalogues systèmes. D'autres vues donnent accès à l'état interne du serveur.

Le schéma d'information (Chapitre 37) fournit un autre ensemble de vues qui recouvrent les fonctionnalités des vues système. Comme le schéma d'information fait partie du standard SQL, alors que les vues décrites ici sont spécifiques à PostgreSQL, il est généralement préférable d'utiliser le schéma d'information si celui-ci apporte toutes les informations nécessaires.

Tableau 54.1 liste les vues systèmes décrites plus en détails dans la suite du document. Il existe de plus des vues permettant d'accéder à des statistiques cumulées ; elles sont décrites dans Tableau 28.2.

54.1. Aperçu

Tableau 54.1 liste les vues systèmes. Une documentation plus détaillée de chaque catalogue est disponible ci-dessous. Sauf lorsque c'est indiqué, toutes les vues décrites ici sont en lecture seule.

Tableau 54.1. Vues système

Nom de la vue	But
pg_available_extensions	extensions disponibles
pg_available_extension_versions	versions disponibles des extensions
pg_backend_memory_contexts	contextes mémoire du processus
pg_config	paramètres de configuration au moment de la compilation
pg_cursors	curseurs ouverts
pg_file_settings	résumé du contenu des fichiers de configuration
pg_group	groupe d'utilisateurs de la base de données
pg_hba_file_rules	résumé du contenu du fichier de configuration de l'authentification des clients
pg_ident_file_mappings	résumé du contenu du fichier de configuration de correspondance des noms d'utilisateur
pg_indexes	index
pg_locks	verrous actuellement détenus ou en attente currently held or awaited
pg_matviews	vues matérialisées
pg_policies	politiques de sécurité
pg_prepared_statements	instructions préparées
pg_prepared_xacts	transactions préparées
pg_publication_tables	publications et informations sur les tables associées
pg_replication_origin_status	information sur les origines de réplication, incluant la progression de la réplication
pg_replication_slots	informations sur les slots de réplication
pg_roles	rôles des bases de données
pg_rules	règles
pg_seclabels	labels de sécurité

Nom de la vue	But
pg_sequences	séquences
pg_settings	configuration
pg_shadow	utilisateurs des bases de données
pg_shmem_allocations	allocations de mémoire partagée
pg_stats	statistiques du planificateur
pg_stats_ext	statistiques étendues du planificateur
pg_stats_ext_exprs	statistiques étendues du planificateur pour les expressions
pg_tables	tables
pg_timezone_abbrevs	abréviations des fuseaux horaires
pg_timezone_names	noms des fuseaux horaires
pg_user	utilisateurs des bases de données
pg_user_mappings	user mappings
pg_views	vues

54.2. pg_available_extensions

La vue pg_available_extensions liste les extensions disponibles pour cette installation. Voir aussi le catalogue pg_extension qui affiche les extensions actuellement installées.

Tableau 54.2. Colonnes de pg_available_extensions

Type Description		
name name Nom de l'extension		
default_version text Nom de la version par défaut, ou NULL si aucune version n'est indiquée		
installed_version text Version actuellement installée pour cette extension, ou NULL si elle n'est pas installée		
comment text Chaîne de commentaire à partir du fichier de contrôle de l'extension		

La vue pg_available_extensions est en lecture seule.

54.3. pg_available_extension_versions

La vue pg_available_extension_versions liste les versions spécifiques des extensions disponibles sur cette installation. Voir aussi le catalogue pg_extension qui affiche les extensions actuellement installées.

Tableau 54.3. Colonnes de `pg_available_extension_versions`

Type Description
`name` `name` Nom de l'extension
`version` `text` Nom de la version
`installed` `bool` Vrai si cette version de l'extension est actuellement installée
`superuser` `bool` Vrai si seuls les superutilisateurs sont autorisés à installer cette extension
`trusted` `bool` Vrai si l'extension peut être installé par des utilisateurs n'ayant pas l'attribut SUPERUSER avec les droits appropriés
`relocatable` `bool` Vrai si l'extension peut être déplacée dans un autre schéma
`schema` `name` Nom du schéma dans lequel l'extension doit être installée ou `NULL` si elle est déplaçable partiellement ou complètement
`requires` `name[]` Noms des extensions requises, ou `NULL` si aucune extension supplémentaire n'est nécessaire
`comment` `text` Chaîne de commentaire provenant du fichier de contrôle de l'extension

La vue `pg_available_extension_versions` est en lecture seule.

54.4. `pg_backend_memory_contexts`

La vue `pg_backend_memory_contexts` affiche tous les contextes mémoire du processus serveur attaché à la session en cours.

`pg_backend_memory_contexts` contient une ligne pour chaque contexte mémoire.

Tableau 54.4. Colonnes de `pg_backend_memory_contexts`

Type de colonne Description
`name` `text` Nom du contexte mémoire
`ident` `text` Information d'identification du contexte mémoire. Ce champ est tronqué à 1024 octets
`parent` `text` Name du parent de ce contexte mémoire
`level` `int4` Distance de TopMemoryContext dans l'arbre des contextes
`total_bytes` `int8` Nombre total d'octets alloués pour ce contexte mémoire
`total_nblocks` `int8` Nombre total de blocs alloué pour ce contexte mémoire

Type de colonne Description
free_bytes int8 Espace libre en octets
free_chunks int8 Nombre total de morceaux libres
used_bytes int8 Espace utilisé en octets

Par défaut, la vue pg_backend_memory_contexts peut seulement être lue par les superutilisateurs ou par les rôles qui ont les droits du rôle pg_read_all_stats.

54.5. pg_config

La vue pg_config décrit les paramètres de configuration au moment de la compilation, pour la version actuellement installée de PostgreSQL. Elle a pour but d'être utilisé par les paquets logiciels souhaitant une interface vers PostgreSQL pour faciliter la recherche des fichiers d'en-tête et bibliothèques. Elle fournit les mêmes informations basiques que l'outil client pg_config de PostgreSQL.

Par défaut, la vue pg_config peut seulement être lue par des superutilisateurs.

Tableau 54.5. Colonnes de pg_config

Type Description
name text Le nom du paramètre
setting text La valeur du paramètre

54.6. pg_cursors

La vue pg_cursors liste les curseurs actuellement disponibles. Les curseurs peuvent être définis de plusieurs façons :

- via l'instruction SQL DECLARE ;
- via le message Bind du protocole frontend/backend, décrit dans le Section 55.2.3 ;
- via l'interface de programmation du serveur (SPI), décrite dans le Section 47.1.

La vue pg_cursors affiche les curseurs créés par tout moyen précédent. Les curseurs n'existent que pour la durée de la transaction qui les définit, sauf s'ils ont été déclarés avec WITH HOLD. De ce fait, les curseurs volatils (*non-holdable*) ne sont présents dans la vue que jusqu'à la fin de la transaction qui les a créés.

> **Note**
>
> Les curseurs sont utilisés en interne pour coder certains composants de PostgreSQL, comme les langages procéduraux. La vue pg_cursors peut ainsi inclure des curseurs qui n'ont pas été créés explicitement par l'utilisateur.

Tableau 54.6. Colonnes de pg_cursors

Type Description
name text Le nom du curseur
statement text La chaîne utilisée comme requête pour créer le curseur
is_holdable bool true si le curseur est persistant (*holdable*) (c'est-à-dire s'il peut être accédé après la validation de la transaction qui l'a déclaré) ; false sinon
is_binary bool true si le curseur a été déclaré binaire (BINARY) ; false sinon
is_scrollable bool true si le curseur autorise une récupération non séquentielle des lignes ; false sinon
creation_time timestamptz L'heure à laquelle le curseur a été déclaré

La vue pg_cursors est en lecture seule.

54.7. pg_file_settings

La vue pg_file_settings fournit un résumé du contenu des fichiers de configuration du serveur. Une ligne apparaît dans cette vue pour chaque entrée « nom = valeur » apparaissant dans les fichiers, avec des annotations indiquant si la valeur peut être appliquée avec succès. Des lignes additionnelles peuvent apparaître pour des problèmes non liés aux entrées « nom = valeur », comme des erreurs de syntaxe dans les fichiers.

Cette vue est utile pour vérifier que les changements envisagés dans les fichiers de configuration fonctionneront, ou pour diagnostiquer une erreur intervenue. Notez que cette vue rapporte sur le contenu *courant* des fichiers, pas sur ce qui a été appliqué dernièrement par le serveur. (La vue pg_settings est généralement suffisante pour indiquer cela.)

Par défaut, la vue pg_file_settings peut être seulement lue par les superutilisateurs.

Tableau 54.7. Colonnes de pg_file_settings

Type Description
sourcefile text Chemin complet du fichier de configuration
sourceline int4 Numéro de ligne dans le fichier de configuration où l'entrée apparaît
seqno int4 L'ordre dans lequel les entrées sont traitées (1..*n*)
name text Nom du paramètre de configuration
setting text Valeur à assigner au paramètre
applied bool Vrai si la valeur peut être appliquée avec succès
error text

Type Description
Si non NULL, un message d'erreur indiquant pourquoi cette entrée ne peut pas être appliquée

Si le fichier de configuration contient des erreurs de syntaxe ou des noms de paramètres invalides, le serveur n'essaiera pas d'appliquer les réglages correspondants, et en conséquence tous les champs correspondants auront false pour valeur `applied`. Dans de tels cas, il y aura une ou plusieurs lignes avec des champs `error` non NULL indiquant le ou les problèmes. Dans le cas contraire, les réglages individuels seront appliquées si possible. Si un réglage individuel ne peut être appliqué (par exemple, une valeur invalide, ou le réglage ne peut être modifié qu'après le démarrage du serveur), il y aura un message approprié dans le champ `error`. Une autre manière d'avoir une entrée avec un champ `applied` à false est que le réglage est réécrit par une entrée ultérieure dans le fichier de configuration. Ce cas n'est pas considéré comme une erreur, aussi rien n'apparaît dans le champ `error`.

Voir Section 20.1 pour plus d'informations concernant les diverses manières de modifier les paramètres d'exécution.

54.8. pg_group

La vue `pg_group` existe pour des raisons de compatibilité ascendante : elle émule un catalogue qui a existé avant la version 8.1 de PostgreSQL. Elle affiche les noms et membres de tous les rôles dont l'attribut `rolcanlogin` est dévalidé, ce qui est une approximation des rôles utilisés comme groupes.

Tableau 54.8. Colonnes de pg_group

Type Description
`groname` `name` (référence `pg_authid.rolname`) Nom du groupe
`grosysid` `oid` (référence `pg_authid.oid`) Identifiant du groupe
`grolist` `oid[]` (référence `pg_authid.oid`) Un tableau contenant les identifiants des rôles du groupe

54.9. pg_hba_file_rules

La vue `pg_hba_file_rules` fournit un résumé du contenu du fichier de configuration d'authentification des clients, le fichier `pg_hba.conf`. Une ligne apparaît dans cette vue pour chaque ligne non vide et qui n'est pas un commentaire, avec des annotations indiquant si la règle a pu être appliquée avec succès.

Cette vue peut être utile pour vérifier si les modifications planifiées dans le fichier de configuration de l'authentification fonctionneront ou pour diagnostiquer un échec précédent. Notez que cette vue renvoie le contenu *courant* du fichier et non pas ce qui a été chargé la dernière fois sur le serveur.

Par défaut, la vue `pg_hba_file_rules` peut seulement être lue par les superutilisateurs.

Tableau 54.9. Colonnes de pg_hba_file_rules

Type Description
`line_number` `int4` Numéro de ligne de cette règle dans `pg_hba.conf`

Type Description
`type` `text` Type de connexion
`database` `text[]` Liste des noms des bases de données pour lesquelles cette règle s'applique
`user_name` `text[]` Liste des noms d'utilisateurs et de groupes pour lesquels cette règle s'applique
`address` `text` Nom d'hôte ou adresse IP, ou une valeur parmi `all`, `samehost`, et `samenet`. NULL pour les connexions locales.
`netmask` `text` Masque d'adresse IP, ou NULL si non applicable
`auth_method` `text` Méthode d'authentification
`options` `text[]` Options spécifiées pour la méthode d'authentification
`error` `text` Si non NULL, un message d'erreur indiquant pourquoi cette ligne n'a pas pu être traitée

Habituellement, une ligne reflétant une entrée incorrecte aura uniquement des valeurs pour les champs `line_number` et `error`.

Voir Chapitre 21 pour plus d'informations sur la configuration d'authentification des clients.

54.10. `pg_ident_file_mappings`

La vue `pg_ident_file_mappings` fournit un résumé du contenu du fichier de configuration des correspondances des noms d'utilisateur, `pg_ident.conf`. Une ligne apparaît dans cette vue pour chaque ligne du fichier qui n'est ni vide ni un commentaire, avec des annotations indiquant si la carte peut être appliquée avec succès.

Cette vue peut être utile pour vérifier si les modifications prévues dans le fichier de configuration des authentifications fonctionneront ou pour diagnostique un échec précédent. Notez que cette vue donne des informations sur le contenu *actuel* du fichier, et non pas sur ce qui a été chargé en dernier par le serveur.

Par défaut, la vue `pg_ident_file_mappings` peut seulement être lue par les superutilisateurs.

Tableau 54.10. Colonnes de `pg_ident_file_mappings`

Type Description
`line_number` `int4` Numéro de ligne de cette carte dans `pg_ident.conf`
`map_name` `text` Nom de la carte
`sys_name` `text` Nom d'utilisateur détectée du client
`pg_username` `text` Nom d'utilisateur PostgreSQL démandé
`error` `text` Si non `NULL`, message d'erreur indiquant pourquoi cette ligne n'a pas été traitée

Habituellement, une ligne reflétant un enregistrement incorrect aura des valeurs uniquement pour les champs `line_number` et `error`.

Voir Chapitre 21 pour plus d'informations sur la configuration de l'authentification des clients.

54.11. `pg_indexes`

La vue `pg_indexes` fournit un accès aux informations utiles sur chaque index de la base de données.

Tableau 54.11. Colonnes de `pg_indexes`

Type Description
`schemaname` `name` (référence `pg_namespace.nspname`) Nom du schéma contenant les tables et index
`tablename` `name` (référence `pg_class.relname`) Nom de la table portant l'index
`indexname` `name` (référence `pg_class.relname`) Nom de l'index
`tablespace` `name` (référence `pg_tablespace.spcname`) Nom du *tablespace* contenant l'index (NULL s'il s'agit de celui par défaut pour la base de données)
`indexdef` `text` Définition de l'index (une commande CREATE INDEX reconstruite)

54.12. `pg_locks`

La vue `pg_locks` fournit un accès aux informations concernant les verrous détenus par les transactions actives sur le serveur de bases de données. Voir le Chapitre 13 pour une discussion plus importante sur les verrous.

`pg_locks` contient une ligne par objet verrouillable actif, type de verrou demandé et processus associé. Un même objet verrouillable peut apparaître plusieurs fois si plusieurs processus ont posé ou attendent des verrous sur celui-ci. Toutefois, un objet qui n'est pas actuellement verrouillé n'apparaît pas.

Il existe plusieurs types distincts d'objets verrouillables : les relations complètes (tables, par exemple), les pages individuelles de relations, des tuples individuels de relations, les identifiants de transaction (virtuels et permanents) et les objets généraux de la base de données (identifiés par l'OID de la classe et l'OID de l'objet, de la même façon que dans `pg_description` ou `pg_depend`). De plus, le droit d'étendre une relation est représenté comme un objet verrouillable distinct, tout comme l'est le droit de mettre à jour `pg_database.datfrozenxid`. Et enfin, les verrous informatifs peuvent être pris sur les numéros qui ont la signification définie par l'utilisateur.

Tableau 54.12. Colonnes de `pg_locks`

Type Description
`locktype` `text` Type de l'objet verrouillable : `relation`, `extend`, `frozenid`, `page`, `tuple`, `transactionid`, `virtualxid`, `spectoken`, `object`, `userlock` ou `advisory`. (See also Tableau 28.11.)
`database` `oid` (référence `pg_database.oid`) L'OID de la base de données dans laquelle existe l'objet à verrouiller, 0 si la cible est un objet partagé ou NULL si l'objet est un identifiant de transaction

Type Description
`relation` `oid` (référence `pg_class.oid`) L'OID de la relation visée par le verrouillage, ou NULL si la cible n'est ni une relation ni une partie de relation
`page` `int4` Le numéro de page visé par le verrouillage à l'intérieur de cette relation ou NULL si la cible n'est pas un tuple ou une page de relation
`tuple` `int2` Le numéro du tuple dans la page, ciblé par le verrouillage, ou NULL si la cible n'est pas un tuple
`virtualxid` `text` L'identifiant virtuel de transaction visé par le verrouillage, ou NULL si la cible n'est pas un identifiant virtuel de transaction
`transactionid` `xid` L'identifiant de transaction ciblé par le verrouillage ou NULL si la cible n'est pas un identifiant de transaction
`classid` `oid` (référence `pg_class.oid`) L'OID du catalogue système contenant la cible du verrouillage ou NULL si la cible n'est pas un objet général de la base de données
`objid` `oid` (référence toute colonne OID) L'OID de la cible du verrouillage dans son catalogue système ou NULL si la cible n'est pas un objet général de la base de données
`objsubid` `int2` Numéro de la colonne ciblée par le verrou (`classid` et `objid` font référence à la table elle-même), ou 0 si la cible est un autre objet de la base de données, ou NULL si l'objet n'est pas un objet de la base de données.
`virtualtransaction` `text` L'identifiant virtuel de la transaction qui détient ou attend le verrou.
`pid` `int4` L'identifiant du processus serveur qui détient ou attend le verrou. NULL si le verrou est possédé par une transaction préparée.
`mode` `text` Nom du type de verrou détenu ou attendu par ce processus (voir la Section 13.3.1 et Section 13.2.3)
`granted` `bool` `True` si le verrou est détenu, `false` s'il est attendu
`fastpath` `bool` `True` si le verrou a été obtenu grâce au raccourci, `false` s'il a été obtenu via la table principale des verrous
`waitstart` `timestamptz` Horodatage du début de l'attente de ce processus pour ce verrou ou NULL si le verrou est détenu. Notez que ce champ peut être NULL pendant une période de temps très courte après que l'attente ait commencé même si `granted` vaut `false`.

`granted` est `true` sur une ligne représentant un verrou tenu par le processus indiqué. False indique que le processus attend l'acquisition de ce verrou, ce qui implique qu'au moins un autre processus détient ou est en attente d'un verrou en conflit sur le même objet. Le processus en attente dormira jusqu'à ce que l'autre verrou soit relâché (ou qu'une situation de deadlock soit détecté). Un processus seul ne peut attendre qu'au plus un verrou à la fois.

Au cours de l'exécution d'une transaction, un processus serveur détient un verrou exclusif sur l'identifiant virtuel de transaction. Si un identifiant permanent est affecté à la transaction (ce qui arrive

normalement seulement si la transaction modifie l'état de la base), il détient aussi un verrou exclusif sur l'identifiant permanent de transaction jusqu'à sa fin. Quand un processus a besoin d'attendre la fin d'une autre transaction, il le fait en tentant d'acquérir un verrou partagé sur l'identfiant virtuel ou permanent de cette autre transaction . Ceci ne réussira que quand l'autre transaction se termine et relâche son verrou.

Bien que les lignes constituent un type d'objet verrouillable, les informations sur les verrous de niveau ligne sont stockées sur disque, et non en mémoire. Ainsi, les verrous de niveau ligne n'apparaissent normalement pas dans cette vue. Si un processus attend un verrou de niveau ligne, elle apparaît sur la vue comme en attente de l'identifiant permanent de la transaction actuellement détentrice de ce verrou de niveau ligne.

Les verrous consultatifs peuvent être acquis par des clés constituées soit d'une seule valeur `bigint`, soit de deux valeurs `integer`. Une clé `bigint` est affichée avec sa moitié haute dans la colonne `classid`, sa partie basse dans la colonne `objid` et `objsubid` à 1. La valeur `bigint` originale peut être recréée avec l'expression `(classid::bigint << 32) | objid::bigint`. Les clés `integer` sont affichées avec la première clé dans la colonne `classid`, la deuxième clé dans la colonne `objid` et `objsubid` à 2. La signification réelle des clés est laissée à l'utilisateur. Les verrous consultatifs sont locaux à chaque base, la colonne `database` a donc un sens dans ce cas.

Bien qu'il soit possible d'obtenir des informations sur les processus bloquant d'autres processus en joignant la vue `pg_locks` avec elle-même, c'est très difficile à réaliser correctement dans le détail. Une telle requête devrait intégrer toutes les informations sur les conflits des modes de verrous. Pire, la vue `pg_locks` n'expose pas d'informations sur les processus en avance des autres dans les queues d'attente de verrous, pas plus que des informations sur les processus exécutés en parallèle pour d'autres sessions clientes. Il est préférable d'utiliser la fonction `pg_blocking_pids()` (voir Tableau 9.66) pour identifier les processus bloqués par d'autres processus.

`pg_locks` fournit une vue globale de tous les verrous du cluster, pas seulement de ceux de la base en cours d'utilisation. Bien que la colonne `relation` puisse être jointe avec `pg_class.oid` pour identifier les relations verrouillées, ceci ne fonctionne correctement qu'avec les relations de la base accédée (celles pour lesquelles la colonne `database` est l'OID de la base actuelle ou 0).

La vue `pg_locks` affiche des données provenant du gestionnaire de verrous standards et du gestionnaire de verrous de prédicats, qui sont des systèmes autrement séparés ; de plus, le gestionnaire de verrous standards sous-divise ses verrous en verrous réguliers et en verrous rapides (*fast-path*). Cette donnée n'est pas garantie comme étant entièrement cohérente. Quand la vue est exécutée, les données des verrous fast-path (avec `fastpath` = `true`) sont récupérées à partir de chaque processus, un à la fois, sans geler l'état du gestionnaire des verrous. Donc il est possible que des verrous soient pris ou relachés pendant la récupération de l'information. Néanmoins, notez que ces verrous sont connus pour ne pas entrer en conflit avec les autres verrous déjà détenus. Après que tous les processus serveurs a été interrogés pour connaître leurs verrous fast-path, le reste du gestionnaire des verrous standards est verrouillé de manière unitaire et une image cohérente de tous les verrous restants est collectée en une seule opération atomique. Après avoir déverrouillé le gestionnaire de verrous standards, le gestionnaire de verrous de prédicats est verrouillé de la même manière et tous les verrous de prédicats sont récupérés en une action atomique. Du coup, avec l'exception des verrous fast-path, chaque gestionnaire de verrous délivrera un état cohérent des résultats mais, comme nous ne verrouillons pas les deux gestionnaires de verrous simultanément, il est possible que les verrous soient pris ou relachés après avoir interrogé du gestionnaire de verrous standards et avant avoir interrogé le gestionnaire des verrous de prédicats.

Verrouiller le gestionnaire de verrous standards ou de prédicats peut avoir un impact sur les performances de la base de données si cette vue est fréquemment interrogée. Les verrous sont bloqués le temps minimum nécessaire pour obtenir les données des gestionnaires de verrous mais cela n'élimine pas complètement la possibilité d'un impact sur les performances.

La colonne `pid` peut être jointe à la colonne `pid` de la vue `pg_stat_activity` pour obtenir plus d'informations sur la session qui détient ou attend un verrou, par exemple :

```
SELECT * FROM pg_locks pl LEFT JOIN pg_stat_activity psa
    ON pl.pid = psa.pid;
```

. De plus, si des transactions préparées sont utilisées, la colonne `virtualtransaction` peut être jointe à la colonne `transaction` de la vue`pg_prepared_xacts` pour obtenir plus d'informations sur les transactions préparées qui détiennent des verrous. (Une transaction préparée ne peut jamais être en attente d'un verrou mais elle continue à détenir les verrous qu'elle a acquis pendant son exécution.) Par exemple :

```
SELECT * FROM pg_locks pl LEFT JOIN pg_prepared_xacts ppx
    ON pl.virtualtransaction = '-1/' || ppx.transaction;
```

54.13. `pg_matviews`

La vue `pg_matviews` donne accès à des informations utiles sur chaque vue matérialisée de la base.

Tableau 54.13. Colonnes de `pg_matviews`

Type Description
`schemaname` `name` (référence `pg_namespace.nspname`) Nom du schéma contenant la vue matérialisée
`matviewname` `name` (référence `pg_class.relname`) Nom de la vue matérialisée
`matviewowner` `name` (référence `pg_authid.rolname`) Nom du propriétaire de la vue matérialisée
`tablespace` `name` (référence `pg_tablespace.spcname`) Nom du tablespace contenant la vue matérialisée (null s'il s'agit du tablespace par défaut pour cette base)
`hasindexes` `bool` Vrai si la vue matérialisée a (ou a eu) des index
`ispopulated` `bool` Vrai si la vue matérialisée est peuplée
`definition` `text` Définition de la vue matérialisée (une requête SELECT reconstruite)

54.14. `pg_policies`

La vue `pg_policies` donne accès à des informations utiles à propos des politiques de sécurité niveau ligne dans la base de données.

Tableau 54.14. Colonnes de `pg_policies`

Type Description
`schemaname` `name` (référence `pg_namespace.nspname`) Nom du schéma contenant la table sur laquelle la politique de sécurité s'applique
`tablename` `name` (référence `pg_class.relname`) Nom de la table sur laquelle la politique de sécurité s'applique
`policyname` `name` (référence `pg_policy.polname`) Nom de la politique de sécurité

Type Description
permissive text La politique est-elle permissive ou restrictive ?
roles name[] Les rôles auxquels s'applique cette politique de sécurité
cmd text Le type de commande auquel s'applique la politique de sécurité
qual text L'expression ajoutée aux qualifications de barrière de sécurité pour les requêtes auxquelles s'applique la politique de sécurité
with_check text L'expression ajoutée aux qualifications WITH CHECK pour les requêtes auxquelles s'applique la politique de sécurité

54.15. pg_prepared_statements

La vue pg_prepared_statements affiche toutes les instructions préparées disponibles pour la session en cours. Voir PREPARE pour de plus amples informations sur les instructions préparées.

pg_prepared_statements contient une ligne pour chaque instruction préparée. Les lignes sont ajoutées à la vue quand une nouvelle instruction préparée est créée et supprimée quand une instruction préparée est abandonnée (par exemple, via la commande DEALLOCATE).

Tableau 54.15. Colonnes de pg_prepared_statements

Type Description
name text L'identifiant de l'instruction préparée
statement text La requête soumise par le client pour créer cette instruction préparée. Pour les instructions préparées créées en SQL, c'est l'instruction PREPARE soumise par le client. Pour les instructions préparées créées via le protocole frontend/backend, c'est le texte de l'intruction préparée elle-même.
prepare_time timestamptz L'heure de création de l'instruction préparée
parameter_types regtype[] Les types des paramètres attendus par l'instruction préparée sous la forme d'un tableau de regtype. L'OID correspondant à un élément de ce tableau peut être obtenu en convertissant la valeur regtype en oid.
from_sql bool true si l'instruction préparée a été créée via l'instruction SQL PREPARE ; false si l'instruction a été préparée via le protocole frontend/backend
generic_plans int8 Nombre de fois où le plan générique a été sélectionné
custom_plans int8 Nombre de fois où le plan personnalisé a été sélectionné

La vue pg_prepared_statements est en lecture seule.

54.16. pg_prepared_xacts

La vue pg_prepared_xacts affiche les informations concernant les transactions actuellement préparées pour une validation en deux phases (voir PREPARE TRANSACTION pour les détails).

pg_prepared_xacts contient une ligne par transaction préparée. L'entrée est supprimée quand la transaction est validée ou annulée.

Tableau 54.16. Colonnes de pg_prepared_xacts

Type Description
transaction xid L'identifiant numérique de la transaction préparée
gid text L'identifiant global de transaction assigné à la transaction
prepared timestamptz L'heure de préparation de la transaction pour validation
owner name (référence pg_authid.rolname) Le nom de l'utilisateur qui a exécuté la transaction
database name (référence pg_database.datname) Nom de la base de données dans laquelle a été exécutée la transaction

Lors d'un accès à la vue pg_prepared_xacts, les structures de données du gestionnaire interne des transactions sont momentanément verrouillées et une copie de la vue est faite pour affichage. Ceci assure que la vue produit un ensemble cohérent de résultats tout en ne bloquant pas les opérations normales plus longtemps que nécessaire. Néanmoins, si la vue est accédée fréquemment, les performances de la base de données peuvent être impactées.

54.17. pg_publication_tables

La vue pg_publication_tables fournit des informations sur la correspondance entre les publications et les informations sur les tables qu'elles contiennent. Contrairement au catalogue pg_publication_rel sous-jacent, cette vue étend les publications définies comme FOR ALL TABLES et FOR TABLES IN SCHEMA, donc pour ces publications, il y aura une ligne pour chaque table éligible.

Tableau 54.17. Colonnes de pg_publication_tables

Type Description
pubname name (référence pg_publication.pubname) Nom de la publication
schemaname name (référence pg_namespace.nspname) Nom du schéma contenant la table
tablename name (référence pg_class.relname) Nom de la table
attnames name[] (référence pg_attribute.attname) Noms des colonnes de table incluses dans la publication. Ceci contient toutes les colonnes de la table quand l'utilisateur n'indique pas la liste des colonnes pour la table.
rowfilter text Expression pour la condition de qualification de la publication de la table.

54.18. pg_replication_origin_status

La vue pg_replication_origin_status contient des informations sur l'avancement du rejeu pour une certaine origine. Pour plus d'informations sur les origines de réplication, voir Chapitre 50.

Tableau 54.18. Colonnes de pg_replication_origin_status

Type Description
local_id oid (référence pg_replication_origin.roident) identifiant interne du nœud
external_id text (référence pg_replication_origin.roname) identifiant externe du nœud
remote_lsn pg_lsn Le LSN du nœud de l'origine jusqu'où les données ont été répliquées
local_lsn pg_lsn Le LSN de ce nœud auquel remote_lsn a été répliqué. Utilisé pour vider les enregistrements validés avant de sauvegarder les données sur disque lorsque la validation asynchrone des transactions est utilisée.

54.19. pg_replication_slots

La vue pg_replication_slots fournit une liste de tous les slots de réplication qui existent actuellement sur l'instance, avec leur état courant.

Pour plus d'informations sur les slots de réplication, voir Section 27.2.6 et Chapitre 49.

Tableau 54.19. Colonnes de pg_replication_slots

Type Description
slot_name name Un identifiant unique au niveau de l'instance pour le slot de réplication
plugin name Le nom de base de l'objet partagé contenant le plugin en sortie que ce slot logique utilise, NULL pour les slots physiques
slot_type text Le type du slot - physical ou logical
datoid oid (référence pg_database.oid) L'OID de la base de données avec laquelle ce slot est associée, ou NULL. Seuls les slots logiques ont une base de données associée
database name (référence pg_database.datname) Le nom de la base de données avec laquelle ce slot est associée, ou NULL. Seuls les slots logiques ont une base de données associée.
temporary bool True si c'est un slot de réplication temporaire. Les slots temporaires ne sont pas sauvegardés sur disque et sont automatiquement supprimés lors d'une erreur ou lorsque la session est terminée.
active bool Vrai si ce slot est actuellement utilisé
active_pid int4 L'ID du processus de la session utilisant ce slot si le slot est actuellement activement utilisé. NULL si inactif.
xmin xid

Type Description
La plus ancienne transaction dont ce slot a besoin, et que le serveur doit donc conserver. VACUUM ne peut pas traiter des lignes supprimées par des transactions plus récentes.
`catalog_xmin` `xid` La plus ancienne transaction affectant les catalogues systèmes dont ce slot a besoin et que le serveur doit donc conserver. VACUUM ne peut pas traiter des lignes du catalogues supprimées par des transactions plus récentes.
`restart_lsn` `pg_lsn` L'adresse (LSN) du plus ancien journal de transactions toujours requis par le consommateur de ce slot et qui, de ce fait, ne pourra plus être automatiquement supprimé pendant les checkpoints sauf si ce LSN est en retard de plus de max_slot_wal_keep_size à partir du LSN courant. NULL si le LSN de ce slot n'a jamais été réservé.
`confirmed_flush_lsn` `pg_lsn` L'adresse (LSN) jusqu'où le consommateur de la réplication logique a confirmé avoir reçu les données. Les données plus anciennes ne sont plus disponibles. NULL pour les slots physiques.
`wal_status` `text` Disponibilité des journaux de transactions nécessaires à ce slot. Les valeurs possibles sont : • `reserved` signifie que les journaux nécessaires sont contenus dans `max_wal_size`. • `extended` signifie que `max_wal_size` est dépassé mais que les fichiers sont toujours retenus, soit par un slot de réplication soit par `wal_keep_size`. • `unreserved` signifie que le slot ne contient plus les fichiers nécessaires et que certains seront supprimés au prochain checkpoint. Cet état peut renvoyer soit `reserved` soit `extended`. • `lost` signifie que certains journaux requis sont définitivement perdus et que ce slot ne peut plus être utilisé. Les deux derniers états sont possibles uniquement quand max_slot_wal_keep_size n'est pas négatif.
`safe_wal_size` `pg_lsn` Le nombre d'octets à écrire dans les WAL pour que ce slot ne soit pas en danger de passer dans l'état lost. Il vaut NULL pour les slots dans l'état lost, ainsi que si `max_slot_wal_keep_size` vaut `-1`.
`two_phase` `bool` True si le slot est activé pour décoder les transactions préparées. Toujours false pour les slots physiques.

54.20. `pg_roles`

La vue `pg_roles` fournit un accès aux informations des rôles de base de données. C'est tout simplement une vue accessible de `pg_authid` qui n'affiche pas le champ du mot de passe.

Tableau 54.20. Colonnes de `pg_roles`

Type Description
`rolname` `name` Nom du rôle

Type Description
`rolsuper` `bool` Le rôle est un superutilisateur
`rolinherit` `bool` Le rôle hérite automatiquement des droits des rôles dont il est membre
`rolcreaterole` `bool` Le rôle peut créer d'autres rôles
`rolcreatedb` `bool` Le rôle peut créer des bases de données
`rolcanlogin` `bool` Le rôle peut se connecter, c'est-à-dire que ce rôle peut être indiqué comme identifiant initial d'autorisation de session.
`rolreplication` `bool` Le rôle est un rôle de réplication. Ce type de rôle peut initier des connexions de réplication et créer/supprimer des slots de réplication.
`rolconnlimit` `int4` Pour les rôles autorisés à se connecter, ceci indique le nombre maximum de connexions concurrentes autorisées par rôle. -1 signifie qu'il n'y a pas de limite.
`rolpassword` `text` Ce n'est pas le mot de passe (toujours `********`)
`rolvaliduntil` `timestamptz` Estampille temporelle d'expiration du mot de passe (utilisée uniquement pour l'authentification par mot de passe) ; NULL s'il est indéfiniment valable
`rolbypassrls` `bool` Contourne toutes les politiques de sécurité niveau ligne. Voir Section 5.8 pour plus d'informations.
`rolconfig` `text[]` Valeurs par défaut de certaines variables spécifiques pour ce rôle
`oid` `oid` (référence `pg_authid.oid`) Identifiant du rôle

54.21. `pg_rules`

La vue `pg_rules` fournit un accès à des informations utiles sur les règles de réécriture des requêtes.

Tableau 54.21. Colonnes de `pg_rules`

Type Description
`schemaname` `name` (référence `pg_namespace.nspname`) Nom du schéma contenant la table
`tablename` `name` (référence `pg_class.relname`) Nom de la table pour laquelle est créée la règle
`rulename` `name` (référence `pg_rewrite.rulename`) Nom de la règle
`definition` `text` Définition de la règle (une commande de création reconstruite)

La vue `pg_rules` exclut les règles `ON SELECT` des vues, matérialisées ou non ; elles sont accessibles dans `pg_views` et `pg_matviews`.

54.22. pg_seclabels

La vue pg_seclabels fournit des informations sur les labels de sécurité. C'est une version du catalogue pg_seclabel bien plus lisible.

Tableau 54.22. Colonnes de pg_seclabels

Type Description
objoid oid (référence toute colonne OID) L'OID de l'objet concerné par ce label de sécurité
classoid oid (référence pg_class.oid) L'OID du catalogue système où cet objet apparaît
objsubid int4 Pour un label de sécurité sur une colonne d'une table, cette colonne correspond au numéro de colonne (les colonnes objoid et classoid font référence à la table). Pour tous les autres types d'objets, cette colonne vaut zéro.
objtype text Le type d'objet auquel s'applique ce label, en texte.
objnamespace oid (référence pg_namespace.oid) L'OID du schéma de cet objet si applicable ; NULL dans les autres cas.
objname text Le nom de l'objet auquel s'applique ce label, en texte.
provider text (référence pg_seclabel.provider) Le fournisseur associé à ce label
label text (référence pg_seclabel.label) Le label de sécurité appliqué à cet objet.

54.23. pg_sequences

La vue pg_sequences fournit un accès aux informations utiles sur chaque séquence de la base.

Tableau 54.23. Colonnes de pg_sequences

Type Description
schemaname name (référence pg_namespace.nspname) Nom du schéma contenant la séquence
sequencename name (référence pg_class.relname) Nom de la séquence
sequenceowner name (référence pg_authid.rolname) Nom du propriétaire de la séquence
data_type regtype (référence pg_type.oid) Type de données de la séquence
start_value int8 Valeur de démarrage de la séquence
min_value int8 Valeur minimale de la séquence
max_value int8 Valeur maximale de la séquence
increment_by int8

Type Description
Valeur d'incrément de la séquence
`cycle` `bool` La séquence fait-elle un cycle ?
`cache_size` `int8` Taille du cache de la séquence
`last_value` `int8` La dernière valeur écrite sur disque de la séquence. Si le cache est utilisé, cette valeur peut être supérieure à la dernière valeur renvoyée par la séquence.

La colonne `last_value` vaudra NULL si une des conditions suivantes est vraie :

- La séquence n'a pas encore été lue.
- L'utilisateur actuel n'a pas le droit `USAGE` ou `SELECT` sur la séquence.
- La séquence n'est pas journalisée et le server est un secondaire.

54.24. `pg_settings`

La vue `pg_settings` fournit un accès aux paramètres d'exécution du serveur. C'est essentiellement une interface alternative aux commandes `SHOW` et `SET`. Elle fournit aussi un accès à certaines informations des paramètres qui ne sont pas directement accessibles avec `SHOW`, telles que les valeurs minimales et maximales.

Tableau 54.24. Colonnes de `pg_settings`

Type Description
`name` `text` Nom du paramètre d'exécution
`setting` `text` Valeur actuelle du paramètre
`unit` `text` Unité implicite du paramètre
`category` `text` Groupe logique du paramètre
`short_desc` `text` Description brève du paramètre
`extra_desc` `text` Information supplémentaire, plus détaillée, sur le paramètre
`context` `text` Contexte requis pour positionner la valeur du paramètre (voir ci-dessous)
`vartype` `text` Type du paramètre (`bool`, `enum`, `integer`, `real` ou `string`)
`source` `text` Source de la valeur du paramètre actuel
`min_val` `text` Valeur minimale autorisée du paramètre (NULL pour les valeurs non numériques)
`max_val` `text` Valeur maximale autorisée du paramètre (NULL pour les valeurs non numériques)

Type Description
`enumvals` `text[]` Valeurs autorisées pour un paramètre enum (NULL pour les valeurs non enum)
`boot_val` `text` Valeur de paramètre prise au démarrage du serveur si le paramètre n'est pas positionné d'une autre façon
`reset_val` `text` Valeur à laquelle `RESET` ramènerait le paramètre dans la session courante
`sourcefile` `text` Fichier de configuration dans lequel ce fichier a été positionné (NULL pour les valeurs positionnées ailleurs que dans un fichier de configuration, examiné par un utilisateur qui n'est ni un superutilisateur ni un membre de `pg_read_all_settings`) ; utile lors de l'utilisation de directives `include` dans les fichiers de configuration
`sourceline` `int4` Numéro de ligne du fichier de configuration à laquelle cette valeur a été positionnée (NULL pour des valeurs positionnées ailleurs que dans un fichier de configuration, ou quand examiné par un utilisateur qui n'est ni superutilisateur ni un membre de `pg_read_all_settings`).
`pending_restart` `bool` `true` si la valeur a été modifiée dans le fichier de configuration mais a besoin d'un redémarrage ; ou `false` autrement.

Il existe différentes valeurs de `context`. Les voici, classées dans l'ordre de difficulté décroissante pour la modification d'un paramètre :

`internal`

Ces paramètres ne peuvent pas être modifiés directement ; ils reflètent des valeurs internes. Certaines sont modifiables en compilant le serveur avec des options différentes pour l'étape de configuration, ou en changeant des options lors de l'étape du initdb.

`postmaster`

Ces paramètres sont seulement appliqués au démarrage du serveur, donc toute modification nécessite un redémarrage du serveur. Les valeurs sont typiquement conservées dans le fichier `postgresql.conf` ou passées sur la ligne de commande lors du lancement du serveur. Bien sûr, tout paramètre dont la colonne `context` est inférieure peut aussi être configuré au démarrage du serveur.

`sighup`

Les modifications sur ces paramètres peuvent se faire dans le fichier `postgresql.conf` sans avoir à redémarrer le serveur. L'envoi d'un signal SIGHUP au processus père (historiquement appelé postmaster) le forcera à relire le fichier `postgresql.conf` et à appliquer les modifications. Ce processus enverra aussi le signal SIGHUP aux processus fils pour qu'ils tiennent compte des nouvelles valeurs.

`superuser-backend`

Les modifications de ces réglages peuvent être effectuées dans `postgresql.conf` sans redémarrer le serveur. Ils peuvent également être positionnés pour une session en particulier dans le paquet réseau de demande de connexion (par exemple, via la variable d'environnement `PGOPTIONS` de libpq), mais seulement si l'utilisateur se connectant est superutilisateur ou s'est vu donné les droits `SET` appropriés. Cependant, ces réglages ne changent jamais dans une session une fois qu'elle a débutée. Si vous les modifiez dans `postgresql.conf`, envoyez un signal SIGHUP à postmaster pour le forcer à relire `postgresql.conf`. Les nouvelles valeurs n'affecteront que les sessions ouvertes après cette relecture.

`backend`

Les modifications sur ces paramètres peuvent se faire dans le fichier `postgresql.conf` sans avoir à redémarrer le serveur ; ils peuvent aussi être configurés pour une session particulière dans le paquet de demande de connexion (par exemple, via la variable d'environnement `PGOPTIONS` gérée par la bibliothèque libpq) ; tous les utilisateurs peuvent faire de telles modifications pour leur session. Néanmoins, ces modifications ne changent jamais une fois que la session a démarré. Si vous les changez dans le fichier `postgresql.conf`, envoyez un signal SIGHUP à postmaster car ça le forcera à relire le fichier `postgresql.conf`. Les nouvelles valeurs affecteront seulement les sessions lancées après la relecture de la configuration.

`superuser`

Ces paramètres sont configurables partir du fichier `postgresql.conf` ou à l'intérieur d'une session via la commande `SET` ; mais seuls les superutilisateurs et les utilisateurs qui ont les droits `SET` appropriés peuvent les modifier avec `SET`. Les modifications apportées dans le fichier `postgresql.conf` affecteront aussi les sessions existantes si aucune valeur locale à la session n'a été établie avec une commande `SET`.

`user`

Ces paramètres peuvent être configurés à partir du fichier `postgresql.conf` ou à l'intérieur d'une session via la commande `SET`. Tout utilisateur est autorisé à modifier la valeur sur sa session. les modifi Any user is allowed to change his session-local value. Les modifications apportées dans le fichier `postgresql.conf` affecteront aussi les sessions existantes si aucune valeur locale à la session n'a été établie avec une commande `SET`.

Voir Section 20.1 pour plus d'informations sur les différentes façons de modifier ces paramètres.

La vue `pg_settings` n'accepte ni insertion ni suppression mais peut être mise à jour. Une requête `UPDATE` appliquée à une ligne de `pg_settings` est équivalente à exécuter la commande `SET` sur ce paramètre. Le changement affecte uniquement la valeur utilisée par la session en cours. Si un `UPDATE` est lancé à l'intérieur d'une transaction annulée par la suite, les effets de la commande `UPDATE` disparaissent à l'annulation de la transaction. Lorsque la transaction est validée, les effets persistent jusqu'à la fin de la session, à moins qu'un autre `UPDATE` ou `SET` ne modifie la valeur.

Cette vue n'affiche pas les options personnalisées sauf si le module qui les définit a été chargé par le processus exécutant la requête (via une mention dans shared_preload_libraries, un appel à la fonction C dans l'extension, ou la commande `LOAD`). Par exemple, comme les modules d'archivage sont normalement chargés seulement par le processus d'archivage et non pas par les sessions standards, cette vue n'affichera pas toute option personnalisée définie par ces modules sauf si une action spéciale est réalisée pour les charger dans le processus exécutant la requête.

54.25. pg_shadow

La vue `pg_shadow` existe pour des raisons de compatibilité ascendante : elle émule un catalogue qui a existé avant la version 8.1 de PostgreSQL. Elle affiche les propriétés de tous les rôles marqués `rolcanlogin` dans `pg_authid`.

Cette table tire son nom de la nécessité de ne pas être publiquement lisible, car elle contient les mots de passe. `pg_user` est une vue sur `pg_shadow`, publiquement accessible, car elle masque le contenu du champ de mot de passe.

Tableau 54.25. Colonnes de `pg_shadow`

Type Description
`usename` `name` (référence `pg_authid.rolname`) Nom de l'utilisateur
`usesysid` `oid` (référence `pg_authid.oid`)

Type Description
Identifiant de l'utilisateur
`usecreatedb` `bool` L'utilisateur peut créer des bases de données
`usesuper` `bool` L'utilisateur est un superutilisateur
`userepl` `bool` L'utilisateur peut initier une réplication en flux et peut faire entrer le système en mode sauvegarde et l'en faire sortir.
`usebypassrls` `bool` L'utilisateur passe outre toutes les politiques de sécurité niveau ligne, voir Section 5.8 pour plus d'informations.
`passwd` `text` Mot de passe chiffré ; NULL si aucun. Voir `pg_authid` pour des détails sur le stockage des mots de passe chiffrés.
`valuntil` `timestamptz` Estampille temporelle d'expiration du mot de passe (utilisée uniquement pour l'authentification par mot de passe)
`useconfig` `text[]` Valeurs de session par défaut des variables de configuration

54.26. `pg_shmem_allocations`

La vue `pg_shmem_allocations` affiche les allocations réalisées par le serveur pour le segment de mémoire partagée. Ceci inclut la mémoire allouée par PostgreSQL lui-même et celle allouée par les extensions utilisant le mécanisme détaillé dans Section 38.10.10.

Notez que cette vue n'inclut pas la mémoire allouée en utilisant l'infrastructure de mémoire partagée dynamique.

Tableau 54.26. Colonnes de `pg_shmem_allocations`

Type Description
`name` `text` Nom de l'allocation de mémoire partagée. NULL pour la mémoire inutilisée et `<anonymous>` pour les allocations anonymes.
`off` `int8` Décalage à partir duquel l'allocation commence. NULL pour les allocations anonymes car les détails relatifs sont inconnus.
`size` `int8` Taille de l'allocation
`allocated_size` `int8` Taille de l'allocation en incluant l'alignement. Pour les allocations anonymes, aucune information n'est disponible sur l'alignement, donc les colonnes `size` et `allocated_size` seront identiques. L'alignement n'a pas de signification pour la mémoire libre, donc les colonnes seront identiques aussi dans ce cas.

Les allocations anonymes sont des allocations réalisées directement avec with `ShmemAlloc()`, plutôt que via `ShmemInitStruct()` ou `ShmemInitHash()`.

Par défaut, la vue `pg_shmem_allocations` peut seulement être lue par les superutilisateurs et par les rôles disposant des droits du rôle `pg_read_all_stats`.

54.27. pg_stats

La vue pg_stats fournit un accès aux informations stockées dans la table système pg_statistic. Cette vue n'autorise l'accès qu'aux seules lignes de pg_statistic correspondant aux tables sur lesquelles l'utilisateur a un droit de lecture. Elle peut donc sans risque être publiquement accessible en lecture.

pg_stats est aussi conçue pour afficher l'information dans un format plus lisible que le catalogue sous-jacent -- au prix de l'extension du schéma lorsque de nouveaux types de connecteurs sont définis dans pg_statistic.

Tableau 54.27. Colonnes de pg_stats

Type Description
schemaname name (référence pg_namespace.nspname) Nom du schéma contenant la table
tablename name (référence pg_class.relname) Nom de la table
attname name (référence pg_attribute.attname) Nom de la colonne décrite par la ligne
inherited bool Si vrai, cette ligne inclut les colonnes enfant de l'héritage, pas seulement les valeurs de la table spécifiée
null_frac float4 Fraction d'entrées de colonnes qui sont NULL
avg_width int4 Largeur moyenne en octets des entrées de la colonne
n_distinct float4 Si positif, nombre estimé de valeurs distinctes dans la colonne. Si négatif, nombre de valeurs distinctes divisé par le nombre de lignes, le tout muliplié par -1. (La forme négative est utilisée quand ANALYZE croit que le nombre de valeurs distinctes a tendance à grossir au fur et à mesure que la table grossit ; la forme positive est utilisée lorsque la commande semble avoir un nombre fixe de valeurs possibles.) Par exemple, -1 indique une colonne unique pour laquelle le nombre de valeurs distinctes est identique au nombre de lignes.
most_common_vals anyarray Liste de valeurs habituelles de la colonne. (NULL si aucune valeur ne semble identique aux autres.)
most_common_freqs float4[] Liste de fréquences des valeurs les plus courantes, c'est-à-dire le nombre d'occurrences de chacune divisé par le nombre total de lignes. (NULL lorsque most_common_vals l'est.)
histogram_bounds anyarray Liste de valeurs qui divisent les valeurs de la colonne en groupes de population approximativement identiques. Les valeurs dans most_common_vals, s'il y en a, sont omises de ce calcul d'histogramme. (Cette colonne est NULL si le type de données de la colonne ne dispose pas de l'opérateur < ou si la liste most_common_vals compte la population complète.)
correlation float4 Corrélation statistique entre l'ordre physique des lignes et l'ordre logique des valeurs de la colonne. Ceci va de -1 à +1. Lorsque la valeur est proche de -1 ou +1, un parcours de l'index sur la colonne est estimé moins coûteux que si cette valeur tend vers 0, à cause de

Type Description
la réduction du nombre d'accès aléatoires au disque. (Cette colonne est NULL si le type de données de la colonne ne dispose pas de l'opérateur <.)
`most_common_elems` `anyarray` Une liste des valeurs non NULL les plus communes apparaissant parmi les valeurs de la colonne (NULL pour les types scalaires).
`most_common_elem_freqs` `float4[]` Une liste des fréquences des valeurs les plus communes, c'est-à-dire la fraction des lignes contenant au moins une instance de la valeur donnée. Deux ou trois valeurs supplémentaires suivent les fréquences par élément ; elles correspondent au minimum et au maximum des fréquences précédentes par élément, et en option la fréquence des éléments NULL. (NULL quand `most_common_elems` est NULL.)
`elem_count_histogram` `float4[]` Un histogramme du nombre de valeurs distinctes et non NULL parmi les valeurs de la colonnes, suivi de la moyenne des éléments distincts non NULL. (NULL pour les types scalaires.)

Le nombre maximum d'entrées dans les champs de type tableau est configurable colonne par colonne en utilisant la commande `ALTER TABLE SET STATISTICS` ou globalement avec le paramètre d'exécution default_statistics_target.

54.28. `pg_stats_ext`

La vue `pg_stats_ext` fournit un accès aux informations sur chaque objet statistique étendu dans la base, combinant les informations enregistrées dans les catalogues `pg_statistic_ext` et `pg_statistic_ext_data`. Cette vue autorise l'accès aux seules lignes de `pg_statistic_ext` et `pg_statistic_ext_data` correspondant aux tables dont l'utilisateur est propriétaire et, du coup, l'accès public à cette vue ne pose pas de problème de sécurité.

`pg_stats_ext` est aussi conçu pour présenter les informations dans un format plus lisible que celles des catalogues sous-jacents -- son schéma doit cependant être étendu quand de nouveaux types de statistiques étendues sont ajoutés à `pg_statistic_ext`.

Tableau 54.28. Colonnes de `pg_stats_ext`

Type Description
`schemaname` `name` (référence `pg_namespace.nspname`) Nom du schéma contenant la table
`tablename` `name` (référence `pg_class.relname`) Nom de la table
`statistics_schemaname` `name` (référence `pg_namespace.nspname`) Nom du schéma contenant l'objet de statistique étendue
`statistics_name` `name` (référence `pg_statistic_ext.stxname`) Nom de l'objet de statistiques étendues
`statistics_owner` `name` (référence `pg_authid.rolname`) Propriétaire des statistiques étendues
`attnames` `name[]` (référence `pg_attribute.attname`) Noms des colonnes includes dans l'objet statistique étendue
`exprs` `text[]` Expressions incluses dans l'objet de statistiques étendues
`kinds` `char[]`

Type Description
Types des statistiques étendues activées pour cet enregistrement
`inherited` `bool` (référence `pg_statistic_ext_data.stxdinherit`) Si true, les statistiques incluent les colonnes des enfants héritées, pas uniquement les valeurs de la relation indiquée
`n_distinct` `pg_ndistinct` Nombre de valeurs distinctes pour la combinaison de valeurs de colonnes. Si supérieur à zéro, le nombre estimé de valeurs distinctes dans la combinaison. Si inférieur à zéro, le négatif du nombre de valeurs distinctes divisé par le nombre de lignes. (La forme négative est utilisée quand `ANALYZE` croit que le nombre de valeurs distinctes pourrait bien grossir en même temps que la table ; la forme positive est utilisée quand la colonne semble avoir un nombre fixe de valeurs possibles.) Par exemple, -1 indique une combinaison unique de colonnes pour laquelle le nombre de combinaisons distinctes est identique au nombre de lignes.
`dependencies` `pg_dependencies` Statistiques de dépendance fonctionnelle
`most_common_vals` `text[]` Une liste des combinaisons de valeurs les plus communes dans les colonnes. (NULL si aucune valeur n'est plus commune que les autres.)
`most_common_val_nulls` `bool[]` Une liste de drapeaux NULL pour les combinaisons les plus communes de valeurs. (NULL quand `most_common_vals` l'est.)
`most_common_freqs` `float8[]` Une liste des fréquences des combinaisons les plus communes, autrement dit le nombre d'occurrences de chacune divisé par le nombre total de lignes. (NULL quand `most_common_vals` l'est.)
`most_common_base_freqs` `float8[]` Une liste des fréquences de base des combinaisons les plus communes, autrement dit un produit des fréquences par valeur. (NULL quand `most_common_vals` l'est.)

Le nombre maximum d'entrées dans le tableau peut être contrôlé sur une base par colonne en utilisant la commande `ALTER TABLE SET STATISTICS` ou globalement en configurant le paramètre default_statistics_target.

54.29. `pg_stats_ext_exprs`

La vue `pg_stats_ext_exprs` fournit un accès aux informations sur toutes les expressions incluses dans les objets de statistiques étendues, combinant des informations enregistrées dans les catalogues `pg_statistic_ext` et `pg_statistic_ext_data`. Cette vue donne accès seulement aux lignes de `pg_statistic_ext` et `pg_statistic_ext_data` qui correspondent aux tables dont l'utilisateur est le propriétaire et qu'il est de ce fait sûr de permettre unaccès en lecture pour cette vue.

`pg_stats_ext_exprs` est aussi conçu pour présenter l'information dans un format plus lisible que les catalogues sous-jacents -- au coût que ce schéma doit être étendu quand la structure des statistiques change dans `pg_statistic_ext`.

Tableau 54.29. Colonnes de `pg_stats_ext_exprs`

Type de colonne Description
`schemaname` `name` (references `pg_namespace.nspname`) Nom du schéma contenant la table

Type de colonne **Description**
`tablename` `name` (references `pg_class.relname`) Nom de la table sur laquelle l'objet de statistiques est défini
`statistics_schemaname` `name` (references `pg_namespace.nspname`) Nom du schéma contenant l'objet de statistiques
`statistics_name` `name` (references `pg_statistic_ext.stxname`) Nom de l'objet de statistique étendu
`statistics_owner` `name` (references `pg_authid.rolname`) Propriétaire de l'objet de statistique étendu
`expr` `text` Expression incluse dans l'objet de statistique étendu
`inherited` `bool` (référence `pg_statistic_ext_data.stxdinherit`) Si true, les statistiques incluent les colonnes des enfants héritées, pas uniquement les valeurs de la relation indiquée
`null_frac` `float4` Fraction des entrées NULL de l'expression
`avg_width` `int4` Taille moyenne en octets des entrées de l'expression
`n_distinct` `float4` Si supérieur à zéro, le nombre estimé de valeurs distinctes dans l'expression. Si inférieur à zéro, le négatif du nombre de valeurs distinctes divisé par le nombre de lignes. (La forme inverse est utilisé quand `ANALYZE` croit que le nombre de valeurs distinctes va augmenter en même temps que la table grossit ; la forme positive est utilisée quand l'expression semble avoir un nombre fixe de valeurs possibles.) Par exemple, -1 indique une expression unique pour laquelle le nombre de valeurs distinctes est identique au nombre de lignes.
`most_common_vals` `anyarray` Une liste des valeurs les plus communes de l'expression. (NULL si aucune valeur ne semble plus commune que les autres.)
`most_common_freqs` `float4[]` Une liste des fréquences des valeurs les plus communes, autrement dit nombre d'occurrences de chaque divisé par nombre total de lignes. (NULL quand `most_common_vals` l'est.)
`histogram_bounds` `anyarray` Une liste des valeurs qui divisent les valeurs de l'expression en groupes de population approximativement identique. Les valeurs dans `most_common_vals`, si présent, sont omises de ce calcul d'histogramme. (Cette expression est NULL si le type de données de l'expression n'a pas d'opérateur < ou si la liste `most_common_vals` contint la population entière.)
`correlation` `float4` Corrélation statistique entre l'ordre physique des lignes et l'ordre logique des valeurs de l'expression. Ceci va de -1 à +1. Lorsque la valeur est proche de -1 ou +1, un parcours de l'index sur l'expression est estimé moins coûteux que si cette valeur tend vers 0, à cause de la réduction du nombre d'accès aléatoires au disque. (Cette expression est NULL si le type de données de l'expression ne dispose pas de l'opérateur <.)
`most_common_elems` `anyarray` Une liste de valeurs éléments non NULL apparaissant le plus fréquence dans les valeurs de l'expression. (NULL pour les types scalaires.)
`most_common_elem_freqs` `float4[]` Une liste des fréquences des valeurs d'éléments les plus communes, autrement dit la fraction des lignes contenant au moins une instance de la valeur donnée. Deux ou trois

Type de colonne Description
valeurs supplémentaires suivent les fréquences par élément ; ce sont les minimum et maximum des fréquences précédentes par élément, et optionnellement lafréquence des éléments NULL. (NULL quand `most_common_elems` l'est.)
`elem_count_histogram` `float4[]` Un histogramme du nombre de valeurs d'éléments distinctes nonNULL parmi les valeurs de l'expression, suivi par le nombre moyen d'éléments distincts non NULL. (NULL pour les types scalaires.)

Le nombre maximum d'entrées dans les champs de type tableau peut être contrôlé colonne par colonne en utilisant l'instruction `ALTER TABLE SET STATISTICS` ou globalement en configurant le paramètre default_statistics_target.

54.30. `pg_tables`

La vue `pg_tables` fournit un accès aux informations utiles de chaque table de la base de données.

Tableau 54.30. Colonnes de `pg_tables`

Type Description
`schemaname` `name` (référence `pg_namespace.nspname`) Nom du schéma qui possède la table
`tablename` `name` (référence `pg_class.relname`) Nom de la table
`tableowner` `name` (référence `pg_authid.rolname`) Nom du propriétaire de la table
`tablespace` `name` (référence `pg_tablespace.spcname`) Nom du *tablespace* qui contient la table (NULL s'il s'agit du *tablespace* par défaut de la base)
`hasindexes` `bool` (référence `pg_class.relhasindex`) Vrai si la table comporte (ou a récemment comporté) des index
`hasrules` `bool` (référence `pg_class.relhasrules`) Vrai si la table dispose (ou disposait) de règles
`hastriggers` `bool` (référence `pg_class.relhastriggers`) Vrai si la table dispose (ou disposait) de déclencheurs
`rowsecurity` `bool` (référence `pg_class.relrowsecurity`) Vrai si la sécurité niveau ligne est activée sur la table

54.31. `pg_timezone_abbrevs`

La vue `pg_timezone_abbrevs` fournit la liste des abréviations de fuseaux horaires actuellement reconnues par les routines de saisie date/heure. Le contenu de cette vue change avec la modification du paramètre d'exécution timezone_abbreviations.

Tableau 54.31. Colonnes de `pg_timezone_abbrevs`

Type Description
`abbrev` `text` Abréviation du fuseau horaire

Type Description
`utc_offset` `interval` Décalage de l'UTC (positif signifiant à l'est de Greenwich)
`is_dst` `bool` `true` s'il s'agit d'une abréviation de fuseau horaire soumis aux changements d'heure hiver/été

Bien que la plupart des abréviations de fuseau horaire représentent des décalages fixes d'UTC, certains décalages ont variés dans l'histoire (voir Section B.4 pour plus d'informations). Pour ces cas, cette vue présente leur signification actuelle.

54.32. `pg_timezone_names`

La vue `pg_timezone_names` fournit la liste des noms de fuseaux horaires reconnus par `SET TIMEZONE`, avec les abréviations acceptées, les décalages UTC, et l'état du changement d'heure. (Techniquement, PostgreSQL n'utilise pas UTC car les secondes intercalaires ne sont pas gérées.) Contrairement aux abréviations indiquées dans `pg_timezone_abbrevs`, la majorité des noms impliquent des règles concernant les dates de changement d'heure. De ce fait, l'information associée change en fonction des frontières de changement d'heure locales. L'information affichée est calculée suivant la valeur courante de `CURRENT_TIMESTAMP`.

Tableau 54.32. Colonnes de `pg_timezone_names`

Type Description
`name` `text` Nom du fuseau horaire
`abbrev` `text` Abréviation du fuseau horaire
`utc_offset` `interval` Décalage à partir d'UTC (positif signifiant à l'est de Greenwich)
`is_dst` `bool` `true` si les changements d'heure hiver/été sont suivis

54.33. `pg_user`

La vue `pg_user` fournit un accès aux informations concernant les utilisateurs de la base de données. C'est une simple vue publiquement lisible de `pg_shadow` qui masque la valeur du champ de mot de passe.

Tableau 54.33. Colonnes de `pg_user`

Type Description
`usename` `name` Nom de l'utilisateur
`usesysid` `oid` Identifiant de l'utilisateur
`usecreatedb` `bool` L'utilisateur peut créer des bases de données
`usesuper` `bool` L'utilisateur est un superutilisateur

Type Description
`userepl` `bool` L'utilisateur peut initier une réplication en flux et peut faire entrer le système en mode sauvegarde et l'en faire sortir.
`usebypassrls` `bool` L'utilisateur passe outre toutes les politiques de sécurité niveau ligne, voir Section 5.8 pour plus d'informations.
`passwd` `text` Ce n'est pas le mot de passe (toujours `********`)
`valuntil` `timestamptz` Estampille temporelle d'expiration du mot de passe (utilisée uniquement pour l'authentification par mot de passe)
`useconfig` `text[]` Variables d'exécution par défaut de la session

54.34. `pg_user_mappings`

La vue `pg_user_mappings` donne accès aux informations sur les correspondances d'utilisateurs. C'est essentiellement une vue accessible à tous sur `pg_user_mapping` qui cache le champ d'options si l'utilisateur n'a pas le droit de l'utiliser.

Tableau 54.34. Colonnes de `pg_user_mappings`

Type Description
`umid` `oid` (référence `pg_user_mapping.oid`) OID de la correspondance d'utilisateur
`srvid` `oid` (référence `pg_foreign_server.oid`) OID du serveur distant qui contient cette correspondance
`srvname` `name` (référence `pg_foreign_server.srvname`) Nom du serveur distant
`umuser` `oid` (référence `pg_authid.oid`) OID du rôle local mis en correspondance, zéro si la correspondance d'utilisateur est public
`usename` `name` Nom de l'utilisateur local à mettre en correspondance
`umoptions` `text[]` Options spécifiques à la correspondance d'utilisateurs, sous la forme de chaînes « motclé=valeur ».

Pour protéger les mots de passe enregistrés comme option d'une correspondance d'utilisateur, la colonne `umoptions` sera renvoyée NULL sauf dans les cas suivants :

- l'utilisateur courant est l'utilisateur concerné par la correspondance et est le propriétaire du serveur ou détient le droit `USAGE` sur ce serveur ;
- l'utilisateur courant est le propriétaire du serveur et la correspondance est pour `PUBLIC` ;
- l'utilisateur courant est un superutilisateur.

54.35. `pg_views`

La vue pg_views donne accès à des informations utiles à propos de chaque vue de la base.

Tableau 54.35. Colonnes de pg_views

Type Description
schemaname name (référence pg_namespace.nspname) Nom du schéma contenant la vue
viewname name (référence pg_class.relname) Nom de la vue
viewowner name (référence pg_authid.rolname) Nom du propriétaire de la vue
definition text Définition de la vue (une requête SELECT reconstruite)

Chapitre 55. Protocole Frontend/ Backend

PostgreSQL utilise un protocole de messages pour la communication entre le *frontend* et le *backend* (clients et serveurs). Le protocole est supporté sur TCP/IP et aussi sur les sockets de domaine Unix. Le numéro de port 5432 a été enregistré auprès de l'IANA comme numéro de port TCP attribué pour les serveurs utilisant ce protocole, mais, en pratique, tout numéro de port non privilégié peut être utilisé.

Ce document décrit la version 3.0 de ce protocole, implémenté dans PostgreSQL 7.4 et ses versions ultérieures. Pour la description des versions précédentes du protocole, voir les versions précédentes de la documentation PostgreSQL. Un serveur peut accepter plusieurs versions du protocole. Le message *startup-request* initial indique au serveur la version du protocole que le client essaie d'utiliser. Si la version majeure demandée par le client n'est pas comprise par le serveur, la connexion sera rejetée (par exemple, cela surviendrait si le client demandait la version 4.0 du protocole, qui n'existe pas au moment de l'écriture de ce chapitre). Si la version mineure demandée par le client n'est pas acceptée par le serveur (par exemple si le client réclame la version 3.1 mais que le serveur n'accepte que la version 3.0), le serveur peut soit rejeter la connexion soit répondre avec un message `NegotiateProtocolVersion` contenant la version mineure la plus haute qu'il accepte pour ce protocole. Le client peut ensuite choisir soit de tenter une nouvelle connexion avec la version indiquée du protocole soit annuler la connexion.

Pour servir plusieurs clients avec efficacité, le serveur lance un nouveau processus « backend » pour chaque backend. Dans l'implémentation actuelle, un nouveau processus fils est créé immédiatement après la détection d'une connexion entrante. Cependant, ceci est transparent pour le protocole. Pour le protocole, les termes « backend » et « server » sont interchangeables ; de la même façon, les termes « frontend » et « client » sont interchangeables.

55.1. Aperçu

Le protocole a différentes phases pour le démarrage et les opérations normales. Dans la phase de démarrage, le client ouvre une connexion au serveur et s'authentifie à la satisfaction du serveur. (Cela pourrait ne nécessiter qu'un seul message ou plusieurs messages, suivant la méthode d'authentification utilisée.) Si tout va bien, le serveur envoie ensuite une information de statut au client, et entre enfin dans les opérations normales. Sauf pour le message initial startup-request, cette partie du protocole est conduite par le serveur.

Lors des opérations normales, le client envoie des requêtes et d'autres commandes au backend, et le backend envoie les résultats des requêtes ainsi que d'autres réponses. Il existe quelques cas (comme par exemple `NOTIFY`) où le backend enverra des messages non sollicités mais pour la plupart, cette portion d'une session est conduite par des demandes du client.

La fin d'une session est normalement le choix du client mais elle peut être forcée par le backend dans certains cas. Dans tous les cas, quand le backend ferme la connexion, cela annulera toute transaction ouverte (incomplète) avant de quitter.

Lors des opérations normales, les commandes SQL peuvent être exécutées via deux sous-protocoles. Dans le protocole « simple query » (requête simple), le client envoie simplement une chaîne de caractères contenant la requête. Cette chaîne est immédiatement analysée puis exécutée par le backend. Dans le protocole « extended query » (requête étendue), le traitement des requêtes est séparé en plusieurs étapes : l'analyse, le lien avec les valeurs des paramètres, et l'exécution. Ceci offre des bénéfices de flexibilité et de performance, au prix d'une complexité plus importante.

Les opérations normales ont des sous-protocoles supplémentaires pour les opérations spéciales comme `COPY`.

55.1.1. Aperçu des messages

Toute la communication se fait via un flux de message. Le premier octet d'un message identifie le type de message, et les quatre prochains octets précisent la longueur du message (cette longueur s'inclut elle-même, mais pas l'octet de type de message). Le contenu restant du message est déterminé par le type de message. Pour des raisons historiques, le tout premier message envoyé par le client (le message de démarrage) n'a pas d'octet initial de type de message.

Pour éviter de perdre la synchronisation avec le flux de message, les serveurs et les clients lisent typiquement un message entier (en utilisant le nombre d'octets) avant de commencer l'analyse de son contenu. Ceci permet une récupération facilitée si une erreur est détectée lors du traitement du contenu. Dans les situations extrêmes (comme un manque mémoire pour mettre en cache le message), le receveur peut utiliser le nombre d'octets pour déterminer la quantité de données à ignorer avant de continuer la lecture des messages.

Inversement, serveurs et clients doivent faire attention à ne jamais envoyer un message partiel. Ceci se fait habituellement en construisant le message entier dans un cache avant de l'envoyer. Si un échec de communication survient lors d'un envoi ou de la réception d'un message, la seule réaction sensible est d'abandonner la connexion parce qu'il y a très peu d'espoir de récupérer la synchronisation entre messages.

55.1.2. Aperçu des requêtes étendues

Dans le protocole de requête étendue, l'exécution de commandes SQL est divisée en plusieurs étapes. L'état conservé entre les étapes est représenté par deux types d'objets : les *requêtes préparées* et les *portails*. Une requête préparée représente le résultat de l'analyse sémantique d'une chaîne de caractères contenant une requête. Une requête préparée n'est pas prête à s'exécuter parce qu'elle pourrait manquer des valeurs spécifiques pour ses *paramètres*. Un portail représente une instruction prête à être exécutée ou une requête déjà partiellement exécutée, avec toutes valeurs manquantes de paramètres renseignées. (Pour les requêtes `SELECT`, un portail est équivalent à un curseur ouvert mais nous pouvons choisir d'utiliser un terme différent vu que les curseurs ne gèrent pas les requêtes autres que `SELECT`.)

En aperçu, le cycle d'exécution consiste en une étape *parse*, qui crée une requête préparée à partir d'une chaîne de caractères contenant la requête ; une étape *bind*, qui crée un portail à partir d'une requête préparée et des valeurs nécessaires pour les paramètres ; et une étape *execute* qui exécute la requête d'un portail. Dans le cas d'une requête qui renvoie des lignes (`SELECT`, `SHOW`, etc.), l'étape d'exécution doit connaître le nombre limité de lignes à récupérer, si bien que plusieurs étapes d'exécution pourraient être nécessaires pour terminer l'opération.

Le backend peut garder trace de plusieurs requêtes préparées et portails (mais notez qu'ils n'existent que dans une session et ne sont jamais partagés entre sessions). Les requêtes préparées et les portails existants sont référencés par des noms assignés lors de leur création. De plus, les requêtes préparées et portails « sans nom » existent. Bien que ces derniers se comportement largement de la même façon que des objets nommés, les opérations les concernant sont optimisées dans le cas d'une requête exécutée une seule fois, puis annulée, alors que les opérations sur des objets nommés sont optimisées dans un contexte d'utilisation multiple.

55.1.3. Formats et codes de format

Les données d'un type de données particulier pourraient être transmises dans un *format* parmi plusieurs. Depuis PostgreSQL 7.4, les seuls formats acceptés sont « text » et « binary », mais le protocole permet des extensions futures. Le format désiré pour toute valeur est indiqué par un *code format*. Les clients peuvent spécifier un code format pour chaque valeur transmise de paramètre et pour chaque colonne d'un résultat de requête. Le texte a le code format zéro, le binaire a le code format un, et tous les autres codes format sont réservés pour une définition future.

La représentation textuelle des valeurs correspond à la façon dont un type de données est représenté sous forme de chaîne de caractères par les fonction de conversion d'entrée/sortie. Dans la

représentation transmise, il n'existe pas de caractère nul en fin de chaîne ; le client doit ajouter un aux valeurs reçues s'il veut les traiter comme des chaînes de caractères du langage C. (Le format texte n'autorise pas les NULL embarqués.)

Les représentations binaires pour les entiers utiliser l'ordre d'octet réseau (l'octet le plus significatif en premier). Pour les autres types de données, consultez la documentation ou le code source pour apprendre leur représentation binaire. Gardez en tête que les représentations binaires des types de données complexes pourraient changer au fur et à mesure des versions du serveur ; le format texte est généralement le choix le plus portable.

55.2. Flot de messages

Cette section décrit le flot de message et la sémantique de chaque type de message. (Les détails sur la représentation exacte de chaque message apparaît dans Section 55.7.) Il existe plusieurs sous-protocoles différents dépendant de l'état de la connexion : démarrage, requête, appel de fonction, `COPY`, et fin. Il existe aussi plusieurs cas spéciaux pour les opérations asynchrones (incluant les réponses aux notifications et les annulations de commande), qui peuvent survenir à tout moment après la phase de démarrage.

55.2.1. Démarrage (Start-up)

Pour commencer une session, un client ouvre une connexion au serveur et envoie un message de démarrage. Ce message inclut les noms de l'utilisateur et de la base de données sur laquelle l'utilisateur souhaite se connecter ; il identifie aussi la version particulière du protocole à utiliser. (En option, le message de démarrage peut inclure différentes configurations de paramètres.) Le serveur utilise ensuite cette information et le contenu des fichiers de configuration (tels que `pg_hba.conf`) pour déterminer si la connexion est acceptable et le type d'authentification requis (si c'est le cas).

Ensuite, le serveur envoie un message approprié de demande d'authentification, auquel le client doit répondre avec un message approprié de réponse d'authentification (par exemple, avec un mot de passe). Pour toutes les méthodes d'authentification, sauf GSSAPI, SSPI et SASL, il existe au moins une requête et une réponse. Pour certaines méthodes, aucune réponse n'est nécessaire, et donc aucune demande d'authentification n'arrive. Pour les méthodes GSSAPI, SSPI et SASL, plusieurs échanges de paquets pourraient être nécessaires pour terminer l'authentification.

Le cycle d'authentification se termine avec le serveur rejetant la tentative de connexion (message ErrorResponse), ou renvoyant le message AuthenticationOk.

Dans cette phase, les messages possibles provenant du serveur sont :

ErrorResponse

La demande de connexion a été rejetée. Le serveur ferme immédiatement la connexion.

AuthenticationOk

L'échange d'authentification s'est terminé avec succès.

AuthenticationKerberosV5

Le client doit maintenant prendre part à un dialogue d'authentification Kerberos V5 (qui n'est pas décrit ici, et fait partie de la spécification Kerberos) avec le serveur. Si ce dialogue réussit, le serveur répond avec un AuthenticationOk, sinon il répond avec un ErrorResponse. Ceci n'est plus accepté.

AuthenticationCleartextPassword

Le client doit maintenant envoyer un message PasswordMessage contenant le mot de passe en clair. Si le mot de passe est bon, le serveur répond avec un message AuthenticationOk, sinon il répond avec un message ErrorResponse.

AuthenticationMD5Password

Le client doit maintenant envoyer un message PasswordMessage contenant le mot de passe (avec le nom de l'utilisateur) chiffré avec MD5, puis chiffré de nouveau en utilisant le sel aléatoire de quatre octets indiqué dans le message AuthenticationMD5Password. S'il s'agit du bon mot de passe, le serveur répond avec un message AuthenticationOk, sinon il répond avec nu message ErrorResponse. Le message PasswordMessage réel peut être calculé en SQL avec un `concat('md5', md5(concat(md5(concat(motdepasse, nomdutilisateur)), selaléatoire)))`. (Gardez en tête que la fonction `md5()` renvoie le résultat sous la forme d'une chaîne hexadécimale.)

AuthenticationSCMCredential

Cette réponse est uniquement possible pour les connexions locales de type domaine Unix sur les plateformes qui acceptent les messages d'identification SCM. Le client doit envoyer un message d'identification SCM, puis envoyer un seul octet de données. (Le contenu de l'octet de données est inintéressant ; il est seulement utilisé pour s'assurer que le serveur attend suffisamment longtemps pour recevoir le message d'identification.) Si l'identification est acceptable, le serveur répond avec un message AuthenticationOk, sinon il répond avec un message ErrorResponse. (Ce type de message est seulement lancé par les serveurs dont la version est antérieure à la version 9.1. Cela pourrait éventuellement être supprimé à partir de la spécification du protocole.)

AuthenticationGSS

Le client doit maintenant initier une négociation GSSAPI. Le client enverra un message GSSResponse avec la première partie du flux de données GSSAPI en réponse à cela. Si plusieurs messages sont nécessaires, le serveur répondra avec AuthenticationGSSContinue.

AuthenticationSSPI

Le client doit maintenant initier une négociation SSPI. Le client enverra un message GSSResponse avec la première partie du flux de données SSPI en réponse à cela. Si plusieurs messages sont nécessaires, le serveur répondra avec AuthenticationGSSContinue.

AuthenticationGSSContinue

Ce message contient les données de réponse de l'étape précédente de la négociation GSSAPI ou SSPI (AuthenticationGSS, AuthenticationSSPI ou un AuthenticationGSSContinue précédent). Si les données GSSAPI ou SSPI de ce message indique que plus de données sont nécessaires pour terminer l'authentification, le client doit envoyer les données dans un autre message GSSResponse. Si l'authentification GSSAPI ou SSPI est terminée par ce message, le serveur enverra ensuite un message AuthenticationOk pour indiquer une authentification réussie ou un message ErrorResponse pour indiquer un échec.

AuthenticationSASL

Le client doit maintenant initier une négociation SASL, en utilisant un des mécanismes SASL listés dans le message. Le client doit envoyer un message SASLInitialResponse avec le nom du mécanisme sélectionné, et la première partie du flux de données SASL en réponse à ceci. Si plus de messages sont nécessaires, le serveur répondra avec un message AuthenticationSASLContinue. Voir Section 55.3 pour les détails.

AuthenticationSASLContinue

Ce message contient des données de challenge provenant des étapes précédentes de la négociation SASL (AuthenticationSASL, ou d'un précédent AuthenticationSASLContinue). Le client doit répondre avec un message SASLResponse.

AuthenticationSASLFinal

L'authentification SASL a terminé avec les données supplémentaires spécifiques du mécanisme pour le client. Le serveur enverra ensuite le message AuthenticationOk pour indiquer une

authentification réussie ou un message ErrorResponse pour indiquer un échec. Ce message est envoyé uniquement si le mécanisme SASL indique que des données supplémentaires doivent être envoyées du serveur au client à la fin.

NegotiateProtocolVersion

Le serveur n'accepte pas la version mineure du protocole réclamée par le client, mais accepte une version plus ancienne du protocole ; ce message indique la version mineure la plus haute que le serveur accepté. Ce message sera aussi envoyé si le client a demandé des options non acceptées du protocole (par exemple commençant avec `_pq_.`) dans le paquet de démarrage. Ce message sera suivi par un message ErrorResponse ou par un message indiquant le succès ou l'échec de l'authentification.

Si le client n'accepte pas la méthode d'authentification demandée par le client, il doit immédiatement fermer la connexion.

Après avoir reçu le message AuthenticationOk, le client doit attendre plus de messages du serveur. Dans cette phase, un processus backend est démarré, et le client est juste un partenaire intéressé. Il est toujours possible que la demande de démarrage échoue (ErrorResponse) ou que le serveur décline le support de la version mineure demandée du protocole (NegotiateProtocolVersion), mais dans un cas normal, le backend enverra quelques messages ParameterStatus, BackendKeyData et enfin ReadyForQuery.

Lors de cette phase, le backend tentera d'appliquer toute configuration supplémentaire de paramètre d'exécution donnée lors du message de démarrage. EN cas de succès, ces valeurs deviennent les valeurs par défaut de la session. Une erreur cause un message ErrorResponse puis quitte.

Les messages possibles provenant du backend dans cette phase sont :

BackendKeyData

Ce message fournit des données de clé secrète que le client doit conserver s'il souhaite être capable d'annuler des requêtes plus tard. Le client ne doit pas répondre à ce message mais doit continuer à attendre un message ReadyForQuery.

ParameterStatus

Ce message informe le client sur la configuration actuelle (initiale) d'un paramètre serveur, tel que client_encoding ou datestyle. Le client peut ignorer ce message, ou enregistrer la configuration pour une utilisation ultérieure ; voir Section 55.2.7 pour plus de détails. Le client ne doit pas répondre à ce message, mais doit continuer à attendre un message ReadyForQuery.

ReadyForQuery

Le démarrage est terminé. Le client peut maintenant exécuter des commandes.

ErrorResponse

Le démarrage a échoué. La connexion est fermée après l'envoi de ce message.

NoticeResponse

Un message d'avertissement a été envoyé. Le client doit afficher le message mais continuer à attendre des messages ReadyForQuery ou ErrorResponse.

Le message ReadyForQuery est le même que le backend enverra après chaque cycle de commande. Suivant les besoins du client, il est raisonnable de considérer le message ReadyForQuery comme débutant un cycle de commande ou de le considérer comme terminant la phase de démarrage et chaque cycle de commandes suivant.

55.2.2. Requête simple

Un cycle de requête simple est initié par le client envoyant un message Query au backend. The message inclut une commande SQL (ou des commandes) exprimée sous la forme d'une chaîne de caractères. Le backend envoie alors une ou plusieurs réponses suivant le contenu de la chaîne de texte, et termine par un message ReadyForQuery. Ce dernier message informe le client qu'il peut envoyer une nouvelle commande. (Il n'est pas absolument nécessaire que le client attende le message ReadyForQuery avant de lancer une autre commande, mais le client prend alors la responsabilité de comprendre ce qu'il se passe si la commande précédente échoue et que les commandes suivantes déjà lancées réussissent.)

Les messages de réponse possibles du backend sont :

CommandComplete

Une commande SQL terminée normalement.

CopyInResponse

Le backend est prêt à copier des données du client vers une table ; voir Section 55.2.6.

CopyOutResponse

Le backend est prêt à copier les données d'une table vers le client ; voir Section 55.2.6.

RowDescription

Indique que les lignes sont prêtes à être renvoyées en réponse à une requête `SELECT`, `FETCH`, etc. Le contenu de ce message décrit la disposition des colonnes pour les lignes. Ceci sera suivi par un message DataRow pour chaque ligne renvoyée au client.

DataRow

Un des ensembles de lignes renvoyés par une requête `SELECT`, `FETCH`, etc.

EmptyQueryResponse

Une chaîne a été reconnue à la place d'une requête.

ErrorResponse

Une erreur est survenue.

ReadyForQuery

Le traitement de la requête est terminé. Un message séparé est envoyé pour l'indiquer parce que la chaîne de texte pourrait contenir plusieurs commandes SQL. (Le message CommandComplete marque la fin du traitement d'une commande SQL, et non pas de la chaîne complète.) Le message ReadyForQuery sera toujours envoyé, que le traitement se termine avec succès ou avec échec.

NoticeResponse

Un message d'avertissement a été envoyé en relation à la requête. Les avertissements sont en plus des autres réponses, autrement dit le backend continuera de traiter la commande.

La réponse à une requête `SELECT` (ou autres requêtes qui renvoient des ensembles de lignes, tels que `EXPLAIN` ou `SHOW`), consiste normalement en des messages RowDescription, zéro ou plusieurs messages DataRow, et enfin un message CommandComplete. `COPY` vers ou à partir du client fait appel à un protocole spécial décrit dans Section 55.2.6. Tous les autres types de requêtes produisent normalement seulement un message CommandComplete.

Comme une chaîne de texte peut contenir plusieurs requêtes (séparées par des points-virgules), il pourrait y avoir plusieurs séquences de réponses avant que le backend finisse le traitement de la chaîne. Le message ReadyForQuery est lancé quand la chaîne entière a été traitée et que le backend est prêt à accepter une nouvelle chaîne de texte.

Si une chaîne de caractères complètement vide est reçue (aucun contenu autre que les espaces blancs), la réponse est EmptyQueryResponse suivi par ReadyForQuery.

Au cas où une erreur survient, le message ErrorResponse est envoyé suivi d'un message ReadyForQuery. Tous les autres traitement de chaîne de texte sont annulés par le message ErrorResponse (même s'il reste des requêtes dans la chaîne). Notez que ceci pourrait survenir au milieu de la séquence de messages générés par une requête individuel.

Dans le mode simple requête, le format de valeurs récupérées est toujours du texte, sauf quand la commande données est un `FETCH` à partir d'un curseur déclaré avec l'option `BINARY`. Dans ce cas, les valeurs récupérées sont dans un format binaire. Les codes format donnés dans le message RowDescription indiquent le format utilisé.

Un client doit être préparé pour accepter des messages ErrorResponse et NoticeResponse à chaque fois qu'il est attendu d'autres types de message. Voir aussi Section 55.2.7 pour les messages que le backend pourrait générer à cause d'événements externes.

Une pratique recommandée est de coder des clients dans un style machine d'états qui acceptera tout type de message à tout moment où cela aurait du sens, plutôt que de coder des suppositions sur la séquence exacte des messages.

55.2.2.1. Plusieurs requêtes dans le mode requête simple

Quand un message simple Query contient plus d'une requête SQL (séparé par des points-virgules), ces requêtes sont exécutées comme une seule transaction, sauf si des commandes de contrôle de transaction explicites sont inclus pour forcer un comportement différent. Par exemple, si le message contient :

```
INSERT INTO mytable VALUES(1);
SELECT 1/0;
INSERT INTO mytable VALUES(2);
```

alors une erreur de division par zéro dans la requête `SELECT` forcera l'annulation du premier `INSERT`. De plus, comme l'exécution du message est abandonnée à la première erreur, le deuxième `INSERT` n'est jamais tenté.

Si le message contient à la place :

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELECT 1/0;
```

alors le premier `INSERT` est validé par la commande `COMMIT` explicite. Le deuxième `INSERT` et le `SELECT` sont toujours traités comme une seule transaction, donc l'échec de division par zéro annulera le second `INSERT`, mais pas le premier.

Ce comportement est implémenté en exécutant les requêtes dans un message message multi-requêtes dans un *bloc de transaction implicite* sauf s'il existe un bloc de transaction explicite pour leur exécution. La principale différence entre un bloc de transaction implicite et un bloc standard est qu'un bloc implicite est fermé automatiquement à la fin d'un message Query, soit par une validation implicite s'il n'y a pas d'erreur, soit par une annulation implicite s'il y avait une erreur. Ceci est similaire à la validation ou à l'annulation implicite qui survient pour une requête exécutée par elle-même (quand elle n'est pas dans un bloc de transaction).

Si la session est déjà dans un bloc de transaction, en résultat d'un `BEGIN` dans un message précédent, alors le message Query continue simplement ce bloc de transaction, si le message contient une ou plusieurs requêtes. Néanmoins, si le message Query contient un `COMMIT` ou un `ROLLBACK` fermant

le bloc de transaction existant, alors toutes les requêtes suivantes sont exécutées dans un bloc de transaction explicite. Inversement, si un `BEGIN` apparaît dans un message Query multi-requêtes, alors il démarre un bloc de transaction standard qui sera uniquement terminé par un `COMMIT` ou `ROLLBACK` explicite, apparaissant soit dans ce message Query soit dans un message suivant. Si le `BEGIN` suit certaines requêtes qui ont été exécutées sous la forme d'un bloc de transaction implicite, ces requêtes ne sont pas immédiatement validées ; en effet, elles sont inclues rétroactivement dans le nouveau bloc de transaction standard.

Un `COMMIT` ou un `ROLLBACK` apparaissant dans un bloc de transaction implicite est exécuté de façon normale, fermant le bloc implicite ; néanmoins, un message d'avertissement sera renvoyé car un `COMMIT` ou un `ROLLBACK` sans `BEGIN` pourrait être une erreur. Si des requêtes suivent, un nouveau bloc de transaction implicite sera démarré pour elles.

Les savepoints ne sont pas autorisés dans un bloc de transaction implicite car elles pourraient entrer en conflit avec le comportement de fermeture automatique du bloc en cas d'erreur.

Rappelez-vous que, quelque soient les commandes de contrôle des transactions présentes, l'exécution d'un message Query s'arrête à la première erreur. Donc sur cet exemple :

```
BEGIN;
SELECT 1/0;
ROLLBACK;
```

dans un seul message Query, la session sera laissée à l'intérieur d'un bloc de transaction standard en échec car `ROLLBACK` n'est pas atteint après l'erreur de division par zéro. Un autre `ROLLBACK` sera nécessaire pour restaurer la session dans un état utilisable.

Un autre comportement à noter est que l'analyse lexicale et syntaxique est réalisée sur la chaîne de caractères entière avant qu'une seule requête ne soit exécutée. De ce fait, les erreurs simples (telle qu'un mot clé mal orthographié) dans des requêtes ultérieures peuvent empêcher l'exécution des requêtes. Ceci est habituellement invisible pour les utilisateurs car les requêtes vont de toute façon être intégralement annulées quand elles font partie d'un bloc de transaction implicite. Cependant, cela peut se voir lors d'une tentative sur plusieurs transactions dans un message Query multi-requêtes. Par exemple, si une faute est intégrée à l'exemple précédent comme ceci :

```
BEGIN;
INSERT INTO mytable VALUES(1);
COMMIT;
INSERT INTO mytable VALUES(2);
SELCT 1/0;
```

alors aucune des requêtes ne sera exécutée, résultant en une différence visible, à savoir que le premier `INSERT` n'est pas validé. Les erreurs détectées lors de l'analyse sémantique ou plus tard, comme une table ou une colonne mal nommée, n'ont pas cet effet.

Enfin, notez que toutes les requêtes du message Query auront la même valeur pour la fonction `statement_timestamp()`, car cet horodatage est mis à jour seulement lors de la réception d'un message Query. De ce fait, ils auront aussi tous la même valeur pour `transaction_timestamp()`, sauf dans les cas où la chaîne de requêtes finit avec une transaction précédemment démarrée et en commence une nouvelle.

55.2.3. Requête étendue

Le protocole de requête étendue divise le protocole de requête simple décrit ci-dessus en plusieurs étapes. Le résultat des étapes préparatoires peut être ré-utilisé plusieurs fois pour améliorer l'efficacité. De plus, des fonctionnalités supplémentaires sont disponibles, tel que la possibilité de fournir des valeurs de données comme paramètres séparés au lieu d'avoir à les insérer directement dans la chaîne de la requête.

Dans le protocole étendu, le client envoie en premier lieu un message Parse, qui contient une chaîne de caractères pour la requête, avec en options quelques informations sur les types de données des paramètres, et le nom d'un objet représentant la requête préparée (une chaîne vide indiquera une requête préparée sans nom). La réponse est soit un message ParseComplete soit un message ErrorResponse. Les types de données des paramètres peuvent être des types de données indiquées par leur OID. Si aucun type n'est indiqué, l'analyseur tentera de deviner les types de données de la même façon qu'il le ferait pour des constantes de chaînes non typées.

> **Note**
>
> Un type de paramètre peut être laissé sans spécification en le configurant à zéro ou en faisant en sorte que le table d'OID des types de paramètres soit de taille inférieur au nombre de symboles de paramètres ($*n*) utilisés dans la chaîne de requête. Un autre cas spécial est qu'un type de paramètre peut être indiqué comme `void` (autrement dit, l'OID du pseudo-type `void`). Ceci permet l'utilisation de symboles de paramètre pour les paramètres de fonction qui sont en réalité des paramètres OUT. D'habitude, il n'y a pas de contexte dans lequel un paramètre `void` pourrait être utilisé, mais si un tel symbole de paramètre apparaît dans la liste de paramètres d'une fonction, il est en fait ignoré. Par exemple, un appel de fonction tel que `foo($1,$2,$3,$4)` pourrait correspondre à une fonction avec deux arguments IN et deux arguments OUT, si `$3` et `$4` sont indiqués comme étant de type `void`.

> **Note**
>
> La chaîne de requête contenue dans un message Parse ne peut pas inclure plus d'une requête SQL ; dans le cas contraire une erreur SQL est levée. Cette restriction n'existe pas dans le protocole requête simple mais elle existe dans le protocole étendu parce que permettre à des requêtes préparées ou à des portails de contenir plusieurs commandes compliquerait indument le protocole.

En cas de succès à la création, un objet nommé de requête préparé existe jusqu'à la fin de la session, sauf s'il est détruit explicitement. Une requête préparée sans nom dure seulement jusqu'à l'exécution du prochain message Parse pour une requête préparée sans nom. (Notez qu'un message Query détruit aussi une requête préparée sans nom.) Les requêtes préparées nommées doivent être explicitement fermées avant de pouvoir être redéfinies par un autre message Parse, mais ceci n'est pas requis pour une requête préparée sans nom. Les requêtes préparées nommées peuvent aussi être créées et accédées au niveau des commandes SQL en utilisant les instructions `PREPARE` et `EXECUTE`.

Une fois qu'une requête préparée existe, elle peut être préparée pour une exécution en utilisant le message Bind. Ce dernier donne le nom de la requête préparée source (ou une chaîne vide dans le cas d'une requête préparée sans nom) et les valeurs à utiliser pour tous les paramètres présents dans la requête préparée. L'ensemble de paramètres fournis doit correspondre ceux requis par la requête préparée. (Si vous avez indiqué un ou plusieurs paramètres `void` dans le message Parse, donnez des valeurs NULL pour chacune dans le message Bind.) Bind spécifie aussi le format à utiliser pour toute donnée renvoyée par la requête ; le format peut être spécifié de façon globale ou par colonne. La réponse est soit BindComplete soit ErrorResponse.

> **Note**
>
> Le choix entre une sortie texte et une sortie binaire est déterminé par les codes format donnés dans Bind, quelque soit la commande SQL impliquée. L'attribut `BINARY` dans les déclarations de curseur est hors sujet lors de l'utilisation du protocole de requête étendue.

La planification/optimisation de la requête survient typiquement quand le message Bind est traité. Si la requête préparée n'a pas de paramètres ou si elle est exécutée de façon répétée, le serveur pourrait sauvegarde le plan créé et le ré-utiliser lors des messages Bind suivants pour la même requête préparée. Néanmoins, il le fera seulement s'il trouve qu'un plan générique peut être créé, sans être trop inefficace par rapport à un plan dépendant des valeurs spécifiques fournies pour les paramètres. Ceci survient de façon transparente pour ce qui concerne le protocole.

S'il est créé avec succès, un objet portail nommé dure jusqu'à la fin de la transaction en cours, sauf en cas de destruction explicite. Un portail non nommé est détruit à la fin de la transaction ou dès l'exécution du prochain message Bind pour un portal non nommé. (Notez qu'un simple message Query détruit aussi le portail sans nom.) Les portails nommés doivent être fermés explicitement avant de pouvoir être redéfinis par un autre message Bind, mais ceci n'est pas requis pour un portail non nommé. Les portails nommés peuvent aussi être utilisés et accédés au niveau des commandes SQL, en utilisant les instructions `DECLARE CURSOR` et `FETCH`.

Une fois que le portail existe, il peut être exécuté en utilisant un message Execute. Ce dernier spécifie le nom du portail (une chaîne vide indique un portail non nommé) et un nombre maximum de lignes de résultat (zéro signifiant « récupère toutes les lignes »). Ce nombre a seulement un sens pour les portails contenant des commandes renvoyant des ensembles de lignes ; dans les autres cas, la commande est toujours exécutée jusqu'à sa fin, et le nombre de lignes est ignoré. Les réponses possibles à Exécute sont les mêmes que celles décrites ci-dessus pour les requêtes lancées via le protocole de requête simple, sauf que Execute ne cause pas l'exécution de ReadyForQuery ou RowDescription.

Si Execute termine avant la fin de l'exécution d'un portail (à cause de l'atteinte d'un nombre de lignes résultats différent de zéro), il enverra un message PortalSuspended ; l'apparence de ce message indique au client qu'un autre Execute devrait être exécuté contre le même portail pour terminer l'opération. Le message CommandComplete indiquant la fin de la commande SQL source n'est pas envoyé jusqu'à la fin de l'exécution du portail. De ce fait, une phase Execute est toujours terminée par l'apparition d'exactement un de ces messages : CommandComplete, EmptyQueryResponse (si le portail a été créé à partir d'une chaîne vide de requête), ErrorResponse ou PortalSuspended.

À la fin de chaque série de messages du protocole de requête étendue, le client doit envoyer un message Sync. Ce message sans paramètre fait que le backend ferme la transaction en cours si elle n'est pas à l'intérieur d'un bloc de transaction `BEGIN`/`COMMIT` (« fermer » signifiant une validation s'il n'y a pas d'erreur et une annulation dans le cas d'une erreur). Une réponse ReadyForQuery est ensuite produite. Le but de Sync est de fournir un point de resynchronisation pour les erreurs. Quand une erreur est détectée lors du traitement de tout message du protocole de requête étendu, le backend envoie un message ErrorResponse, puis lit et annule les messages jusqu'à la réception d'un Sync, envoie un message ReadyForQuery et enfin retourne à un traitement habituelle des messages. (Mais notez qu'une erreur dans le traitement du message Sync n'est pas ignoré -- ceci assure qu'il y a bien un et un seul ReadyForQuery envoyé pour chaque Sync.)

> **Note**
>
> Sync ne ferme pas un bloc de transaction ouvert avec `BEGIN`. Il est possible de détecter cette situation car le message ReadyForQuery inclut des informations de statut de la transaction.

En plus de ces opérations fondamentales et requises, il existe plusieurs opérations optionnelles pouvant être utilisées avec le protocole de requête étendue.

Le message Describe (variant du portail) indique le nom d'un portail existant (ou une chaîne vide pour le portail non nommé). La réponse est un message RowDescription décrivant les lignes qui seront retournées par l'exécution du portail ; ou un message NoData si le portail ne contient pas une requête qui renverra des lignes ; ou ErrorResponse s'il n'existe pas un tel portail.

Le message Describe (variant de la requête) indique le nom d'une requête préparée existante (ou une chaîne vide pour la requête préparée non nommée). La réponse est un message ParameterDescription décrivant les paramètres nécessaires pour la requête, suivi par un message RowDescription décrivant

les lignes qui seront renvoyés quand la requête sera enfin exécutée (ou un message NoData si la requête ne renverra pas de lignes). Le message ErrorResponse est renvoyé si cette requête préparée n'existe pas. Notez que, comme Bind n'a pas été exécutée, les formats à utiliser pour les colonnes renvoyés ne sont pas encore connus du backend ; les champs de code format seront à zéro dans le message RowDescription dans ce cas.

> **Astuce**
>
> Dans la plupart des scénarios, le client doit envoyer une variante ou l'autre de Describe avant d'envoyer le message Execute, pour s'assurer qu'il sait comment interpréter les résultats qu'il récupérera.

Le message Close ferme une requête préparée ou un portail existant, et libère les ressources. Ce n'est pas une erreur d'envoyer Close pour un nom inexistant de requête préparée ou de portail. La réponse est habituellement CloseComplete, mais pourrait être ErrorResponse si des difficultés sont rencontrées lors de la libération des ressources. Notez que fermer implicitement une requête préparée ferme tout portail ouvert qui était construit par cette requête.

Le message Flush ne cause pas la génération d'une sortie spécifique mais force le backend à renvoyer toutes les données en attente dans les buffers de sortie. Un Flush doit être envoyé après toute commande de requête étendue sauf Sync, si le client souhaite examiner les résultats de cette commande avant de lancer d'autres commandes. Sans Flush, les messages renvoyés par le backend seront combinés dans le plus petit nombre de paquets pour minimiser la surcharge réseau.

> **Note**
>
> Le message Query en requête simple est approximativement équivalent à une série Parse, Bind, Describe portail, Execute, Close, Sync, en utilisant une requête préparée et un portail non nommés et aucun paramètre. Une différence est qu'il acceptera plusieurs requêtes SQL dans la chaîne de requête, réalisant automatiquement les séquences bind/describe/execute pour chaque requête, les unes à la suite des autres. Une autre différence est qu'il ne renvoie pas les messages ParseComplete, BindComplete, CloseComplete et NoData.

55.2.4. Pipelines

L'utilisation du protocole de requête étendue autorise les *pipelines*, autrement dit l'envoi d'une série de requêtes sans attendre que les premières se terminent. Ceci réduit le nombre d'aller/retour réseau nécessaire pour terminer une série d'opérations. Néanmoins, l'utilisateur doit faire attention au comportement souhaité si une des étapes échoue car les requêtes suivantes seront déjà envoyées au serveur.

Une façon de gérer cela est de transformer la série complète de requête en une seule transaction, donc de l'entourer des commandes `BEGIN` ... `COMMIT`. Cela n'aide cependant pas les personnes qui souhaiteraient que certaines commandes soient validées indépendamment des autres.

Le protocole de requête étendue fournit un autre moyen pour gérer cette problématique. Il s'agit d'oublier d'envoyer les messages Sync entre les étapes qui sont dépendantes. Comme, après une erreur, le moteur ignorera les messages des commandes jusqu'à ce qu'il trouve un message Sync, cela autorise les commandes ultérieures d'un pipeline d'être automatiquement ignorées si une commande précédente échoue, sans que le client ait à gérer cela explicitement avec des commandes `BEGIN` et `COMMIT`. Les segments à valider indépendamment dans le pipeline peuvent être séparées par des messages Sync.

Si le client n'a pas exécuté un `BEGIN` explicite, alors chaque Sync implique un `COMMIT` implicite si les étapes prédédentes ont réussi ou un `ROLLBACK` implicite si elles ont échoué. Néanmoins, il existe quelques commandes DDL (comme `CREATE DATABASE`) qui ne peuvent pas être exécutées dans

un bloc de transaction. Si une de ces commandes est exécutée dans un pipeline, cela échouera sauf s'il s'agit de la première commande du pipeline. De plus, en cas de succès, cela forcera une validation immédiate pour préserver la cohérence de la base. De ce fait, un Sync suivant immédiatement une des ces commandes n'aura pas d'effet autre que de répondre avec ReadyForQuery.

Lors de l'utilisation de cette méthode, la fin du pipelin doit être déterminée en comptant les messages ReadyForQuery et en attendant que cela atteigne le nombre de Sync envoyés. Compter les réponses de fin de commande n'est pas fiable car certaines commandes pourraient être ignorées et donc ne pas produire de message de fin.

55.2.5. Appel de fonction (*Function Call*)

Le sous-protocole d'appel de fonction permet au client de demander un appel direct de toute fonction qui existe dans le catalogue système `pg_proc` de la base de données. Le client doit avoir le droit d'exécution sur la fonction.

> **Note**
>
> Le sous-protocole d'appel de fonction est une ancienne fonctionnalité qu'il est certainement préférable d'éviter dans du nouveau code. Des résultats similaires peuvent être accomplis en configurant une requête préparée qui exécute `SELECT function($1, ...)`. Le cycle de l'appel de fonction peut ensuite être remplacé avec les messages Bind/Execute.

Un cycle d'appel de fonction est initié par le client en envoyant un message FunctionCall au backend. Le backend envoie alors un ou plusieurs messages de réponse suivant le résultat de l'appel de fonction, et termine avec un message ReadyForQuery. Ce message informe le client qu'il peut envoyer une nouvelle requête ou un nouvel appel de fonction.

Les messages de réponse possible provenant du backend sont :

ErrorResponse

Une erreur est survenue

FunctionCallResponse

L'appel de fonction s'est terminé et a renvoyé le résultat donné dans le message. (Notez que le protocole d'appel de fonction peut seulement gérer un résultat scalaire simple, pas un type ligne ou un ensemble de résultats.)

ReadyForQuery

Le traitement de l'appel de fonction est terminé. Le message ReadyForQuery sera toujours envoyé, que le traitement termine avec succès ou avec une erreur.

NoticeResponse

Un message d'avertissement a été lancé en relation avec l'appel de fonction. Les notes sont en plus des autres réponses, autrement dit le backend continuera à traiter la commande.

55.2.6. Opérations COPY

La commande `COPY` permet une transfert de données en masse très rapide du ou à partir du serveur. Les opérations de copie vers le serveur (*copy-in*) ou à partir du serveur (*copy-out*) font basculer la connexion dans un sous-protocole distinct, qui dure jusqu'à la fin de l'opération.

Le mode `Copy-in` (transfert de données vers le serveur) est initié quand le backend exécute une requête SQL `COPY FROM STDIN`. Le backend envoie un message CopyInResponse au client. Le client doit alors envoyer zéro ou plusieurs messages CopyData, formant un flux de données en entrée.

(Les limites du message ne sont pas nécessaires liées aux limites des lignes, bien qu'il s'agit souvent d'un choix raisonnable.) Le client peut terminer le mode `copy-in` en envoyant soit un message CopyDone (permettant une fin avec succès) ou un message CopyFail (qui causera l'échec de la requête SQL `COPY` avec un message d'erreur). Le backend annule alors le mode de traitement de commande dans lequel il était entré au début de la commande `COPY`, qui sera soit le protocole de requête simple ou celui de requête étendu. Il enverra ensuite soit un message CommandComplete (en cas de réussite) soit un message ErrorResponse (dans le cas contraire).

Dans le cas d'une erreur détecté par le backend lors du mode `copy-in` (incluant la réception d'un message CopyFail), le backend enverra un message ErrorResponse. Si la commande `COPY` a été lancée via un message de requête étendue, le backend ignorera maintenant les messages du client jusqu'à la réception d'un message Sync. Après, il enverra un message ReadyForQuery et retournera à un traitement normal. Si la commande `COPY` a été lancé via un message Query, le reste de ce message est ignoré et le message ReadyForQuery est envoyé. Dans tous les cas, les messages CopyData, CopyDone ou CopyFail suivants envoyés par le client seront simplement ignorés.

Le backend ignorera les messages Flush et Sync reçus lors du mode `copy-in`. La réception de tout autre message qui ne concerne pas la copie constitue une erreur qui annulera l'état `copy-in` comme décrit ci-dessus. (L'exception pour Flush et Sync est pour simplifier les bibliothèques clientes qui envoient toujours Flush ou Sync après un message Execute, sans vérifier si la commande à exécuter est un `COPY FROM STDIN`.)

Le mode *Copy-out* (transfert de données à partir du serveur) est initié lorsque le backend exécute une requête SQL `COPY TO STDOUT`. Le backend envoie un message CopyOutResponse au client, suivi par zéro ou plusieurs messages CopyData (toujours un par ligne), suivi par un message CopyDone. Le backend annulera alors le mode de traitement de commande dans lequel il était avant le lancement de `COPY`. Enfin, il envoie CommandComplete. Le client ne peut pas annuler le transfert (sauf en fermant la connexion ou en envoyant une requête Cancel), mais il peut ignorer les messages CopyData et CopyDone indésirables.

Dans le cas d'une erreur détectée par le backend en mode `copy-out`, le backend lancera un message ErrorResponse et reviendra en traitement normal. Le client devrait traiter la réception de ErrorResponse comme terminant le mode `copy-out`.

Il est possible que les messages NoticeResponse et ParameterStatus soient mélangés entre des messages CopyData ; les clients doivent gérer ces cas, et devraient être préparés aussi pour tout type de message asynchrone (voir Section 55.2.7). Sinon, tout type de message autre que CopyData ou CopyDone peut être traité comme terminant le mode `copy-out`.

Il existe un autre mode relatif à la copie, appelé *copy-both*, qui permet des transferts de données en masse et très rapides vers *et* à partir du serveur. Le mode `copy-both` est initié quand un backend en mode walsender exécute une instruction `START_REPLICATION`. Le backend envoie un message CopyBothResponse au client. Le backend et le client peuvent ensuite envoyer des messages CopyData jusqu'à ce que l'un des deux envoie un message CopyDone. Une fois que le client a envoyé un message CopyDone, le connexion passe du mode `copy-both` au mode `copy-out` et le client ne peut plus envoyer de messages CopyData. De la même façon, quand le serveur envoie un message CopyDone, la connexion passe en mode `copy-in`, et le serveur ne peut plus envoyer de messages CopyData. Une fois que les deux ont envoyé un message CopyDone, le mode de copie est terminé, et le backend revient au mode de traitement des commandes. Dans le cas où une erreur est détectée par le backend pendant le mode `copy-both`, le backend enverra un message ErrorResponse, ignorera les messages du client jusqu'à ce qu'un message Sync ne soit reçu, puis enverra un message ReadyForQuery avant de revenir au traitement normal. Le client doit traiter la réception d'un message ErrorResponse comme terminant la copie dans les deux directions ; aucun message CopyDone ne devra être envoyé dans ce cas. Voir Section 55.4 pour plus d'informations sur le sous-protocole transmis sur le mode `copy-both`.

Les messages CopyInResponse, CopyOutResponse et CopyBothResponse incluent des champs qui informent le client du nombre de colonnes par ligne et des codes format utilisés pour chaque colonne. (Sur l'implémentation actuelle, toutes les colonnes d'une opération `COPY` donnée utiliseront le même format, mais le design des messages ne le force pas.)

55.2.7. Opérations asynchrones

Il existe différents cas où le backend enverra des messages qui ne sont pas spécifiquement demandés par le flux de commande du client. Les clients doivent être préparés à gérer aussi ces messages à tout moment, même quand une requête n'est pas en cours. Au minimum, un client devrait vérifier ces cas avant de continuer à lire la réponse à une requête.

Il est possible que des messages NoticeResponse soient générés à cause d'une activité externe ; par exemple, si l'administrateur de la base de données ordonne un arrêt « rapide » de la base de données, le backend enverra un message NoticeResponse indiquant ce fait avant de fermer la connexion. Les clients doivent donc être prêts à accepter et afficher les messages NoticeResponse, même quand la connexion n'exécute pas de requêtes.

Les messages ParameterStatus seront générés à chaque fois qu'une valeur change pour un des paramètres pour lesquels le backend pense que le client doit être notifié. La plupart du temps, ceci survient en réponse à une commande SQL `SET` exécutée par le client, et ce cas est effectivement synchrone -- mais il est aussi possible qu'un changement de valeur de paramètres survient parce que l'administrateur a modifié un fichier de configuration, puis envoyé le signal SIGHUP au serveur. De plus, si une commande `SET` est annulé, un message ParameterStatus approprié sera généré pour rapporter la valeur effective.

Il existe actuellement un ensemble, codé en dur, de paramètres pour lesquels un message ParameterStatus sera généré. En voici la liste : `server_version`, `server_encoding`, `client_encoding`, `application_name`, `default_transaction_read_only`, `in_hot_standby`, `is_superuser`, `session_authorization`, `DateStyle`, `IntervalStyle`, `TimeZone`, `integer_datetimes` et `standard_conforming_strings`. (`server_encoding`, `TimeZone` et `integer_datetimes` ne sont pas rapportés avant la version 8.0 ; `standard_conforming_strings` n'est pas rapporté avant la version 8.1 ; `IntervalStyle` n'est pas rapporté avant la version 8.4 ; `application_name` n'est pas rapporté avant la version 9.0 ; `default_transaction_read_only` et `in_hot_standby` ne sont pas rapportés avant la version 14.) Notez que `server_version`, `server_encoding` et `integer_datetimes` sont des pseudos-paramètres qui ne peuvent pas changer après le démarrage. Cet ensemble pourrrait être modifié dans le futur, voire même devenir configurable. En conséquence, un client devrait simplement ignorer les messages ParameterStatus pour les paramètres qu'il ne comprend pas ou qui ne sont pas intéressants pour lui.

Si un client exécute une commande `LISTEN`, alors le backend enverra un message NotificationResponse (à ne pas confondre avec NoticeResponse !) à chaque fois qu'une commande `NOTIFY` est exécutée pour le même nom de canal.

> **Note**
>
> Actuellement, NotificationResponse peut seulement être envoyé en dehors d'une transaction, et donc il ne surviendra pas au milieu d'une série de réponses à des commandes, bien qu'il puisse survenir juste avant un message ReadyForQuery. Il est déconseillé de concevoir la logique du client comme assumant cela. Une bonne pratique est d'être capable d'accepter un message NotificationResponse à tout moment dans le protocole.

55.2.8. Annuler des requêtes en cours

Lors du traitement d'une requête, le client pourrait demander l'annulation de la requête. La demande d'annulation n'est pas envoyée directement sur la connexion ouverte sur le backend pour des raisons d'efficacité de l'implémentation : nous ne voulons pas avoir le backend en train de vérifier constamment pour une nouvelle entrée en provenance du client lors du traitement de la requête. Les demandes

d'annulation sont relativement peu fréquentes, donc nous pouvons les rendre un peu compliquées pour éviter un pénalité sur les cas normaux.

Pour demander une annulation, le client ouvre une nouvelle connexion au serveur et envoie un message CancelRequest, plutôt que le message StartupMessage qui serait habituellement envoyé lors d'une nouvelle connexion. Le serveur traitera cette demande, puis fermera la connexion. Pour des raisons de sécurité, aucune réponse directe n'est faite au message de demande d'annulation.

Un message CancelRequest sera ignoré sauf s'il contient la même donnée clé (PID et clé sécrète) passée au client lors de la connexion. Si la demande correspond au PID et à la clé secrète pour un backend en cours d'exécution d'une requête, le traitement de la requête est annulé. (Dans l'implémentation actuelle, ceci se fait en envoyant un signal spécial au processus backend qui traite la requête.)

Le signal d'annulation pourrait avoir un effet ou pas -- par exemple, s'il arrive après que le backend ait terminé le traitement de la requête, il n'aura aucun effet. Si l'annulation est réelle, la commande en cours est arrêtée avec un message d'erreur.

Le résultat de tout ceci est que, pour des raisons de sécurité et d'efficacité, le client n'a pas de façon de savoir si la requête d'annulation a réussi. Il doit continuer d'attendre que le backend réponse à la requête. Exécuter une annulation améliore simplement les chances que la requête en cours termine rapidement, et améliore les chances qu'elle échouera avec un message d'erreur au lieu d'un succès.

Comme la requête d'annulation est envoyée sur une nouvelle connexion au serveur et non pas via le lien standard de communication client/backend, il est possible d'envoyer une demande d'annulation pour tout processus, pas uniquement pour le client dont la requête est à annuler. Ceci apporte une flexibilité supplémentaire lors de la construction d'applications multi-processus. Cela introduit aussi un risque de sécurité dans le fait que des personnes non autorisées pourraient tenter d'annuler des requêtes. Le risque de sécurité est mitigé par le besoin d'une clé secrète générée dynamiquement, à fournir lors de demandes d'annulation.

55.2.9. Arrêt

La procédure normale et propre d'arrêt est que le client envoie un message Terminate et ferme immédiatement la connexion. Sur réception de ce message, le backend ferme la connexion et quitte.

Dans les rares cas (tel qu'un arrêt de base demandé par l'administrateur), le backend pourrait se déconnecter sans demande du client. Dans ce genre de cas, le backend tentera d'envoyer un message d'erreur ou d'information, donnant la raison de la déconnexion avant de fermer la connexion.

D'autres scénarios d'arrêt surgissent de différents cas d'échec, tel qu'un core dump d'un bout ou de l'autre, une perte du lien de communication, une perte de la synchronisation sur les messages, etc. Si soit le client soit le backend voit une fin inattendue de la connexion, il doit faire son ménage et quitter. Le client a l'option de lancer un nouveau backend en recontactant le serveur s'il ne veut pas quitter. Fermer la connexion est aussi conseillé si un type de message non reconnu est reçu car cela indique probablement la perte de synchronisation des messages.

Pour un arrêt normal ou anormal, toute transaction ouverte est annulée, et non pas validée. Néanmoins, notez que si un client se déconnecter alors qu'une requête autre qu'un `SELECT` est en cours de traitement, le backend finira probablement la requête sans se rendre compte de la déconnexion. Si la requête est exécutée en dehors d'un bloc de transaction (séquence `BEGIN` ... `COMMIT`), alors ses résultats pourraient être validées avant que le backend ne s'aperçoive de la déconnexion.

55.2.10. Chiffrement SSL de la session

Si PostgreSQL a été compilé avec le support de SSL, les communications client/backend peuvent être chiffrés en utilisant SSL. Cela fournit une sécurité sur la communication pour les environnements où des attaquants pourraient être en mesure de capturer le trafic de la session. Pour plus d'informations sur le chiffrement des sessions PostgreSQL avec SSL, voir Section 19.9.

Pour initier une connexion chiffrée avec SSL, le client envoie dès le début un message SSLRequest au lieu de StartupMessage. Le serveur répond avec un seul octet contenant `S` ou `N`, indiquant s'il est, respectivement, prêt ou non à utiliser SSL. Le client pourrait fermer la connexion à ce point s'il n'est pas satisfait par la réponse. Pour continuer après un `S`, le client réalise une poignée de main SSL de démarrage (non décrit ici, car partie de la spécification SSL) avec le serveur. Si la poignée de main est réussie, le client continue avec l'habituel StartupMessage. Dans ce cas, le message StartupMessage et toutes les données qui suivent seront chiffrés avec SSL. Pour continuer après un `N`, le client envoie le message habituel StartupMessage et continue sans chiffrement. (Il est aussi possible de lancer un message GSSENCRequest après une réponse `N` pour tenter l'utilisation du chiffrement GSSAPI à la place du chiffrement SSL.)

Le client doit aussi se préparer à gérer un message ErrorMessage en réponse à un message SSLRequest. Le client ne doit pas afficher ce message d'erreur à l'utilisateur/application car le serveur n'a pas été authentifié (CVE-2024-10977[1]). Dans ce cas, la connexion doit être fermée mais le client pourrait choisir d'ouvrir une connexion propre et continuer sans demander SSL.

Quand le chiffrement SSL peut être réalisé, le serveur est supposé envoyer l'octet `S`, puis attendre que le client initie une poignée de main SSL. Si des octets supplémentaires sont disponibles en lecture, cela signifie probablement qu'une attaque de type *man-in-the-middle* est en cours et tente de réaliser la technique *buffer-stuffing* (CVE-2021-23222[2]). Les clients doivent être codés soit pour lire exactement un octet de la socket avant de renvoyer la socket à leur bibliothèque SSL ou de la traiter comme une violation du protocole s'ils voient qu'ils ont déjà des octets supplémentaires.

Un message initial SSLRequest peut aussi être utilisé dans une connexion en cours d'ouverture pour envoyer un message CancelRequest.

Alors que le protocole lui-même ne fournit pas de moyens pour que le serveur force un chiffrement SSL, l'administrateur peut configurer le serveur de telle façon qu'il rejette les sessions non chiffrées. Cela se fait pendant la vérification de l'authentification.

55.2.11. Chiffrement GSSAPI des sessions

Si PostgreSQL a été compilé avec le support de GSSAPI, les communications client/backend peuvent être chiffrées en utilisant GSSAPI. Ceci fournit une sécurité de la communication dans les environnements où les attaquants pourraient être en mesure de capturer le trafic de la session. Pour plus d'informations sur le chiffrement des sessions PostgreSQL avec GSSAPI, voir Section 19.10.

Pour initier une connexion chiffrée avec GSSAPI, le client doit au départ envoyer un message GSSENCRequest plutôt qu'un message StartupMessage. Le serveur répond avec un seul octet contenant `G` ou `N`, indiquant s'il est, respectivement, disposé ou non pour réaliser un chiffrement GSSAPI. Le client pourrait fermer la connexion à ce stage s'il n'est pas satisfait par la réponse. Pour continuer après un `G`, en utilisant les fonctions C de GSSAPI comme indiqué dans la RFC 2744[3] ou un équivalent, le client réalise une initialisation de GSSAPI en appelant `gss_init_sec_context()` dans une boucle et en envoyant le résultat au serveur, en commençant avec une entrée vide, et en continuant avec chaque résultat du serveur jusqu'à la fin. Lors de l'envoi de résultats de `gss_init_sec_context()` au serveur, il ajoute la longueur du message sous la forme d'un entier sous quatre octets dans l'ordre des octets du réseau. Pour continuer après un `N`, il envoie le message habituel StartupMessage puis continue sans chiffrement. (Une autre solution est de lancer un message SSLRequest après une réponse `N` pour tenter l'utilisation d'un chiffrement SSL à la place du chiffrement GSSAPI.)

Le client doit aussi être préparé à gérer une réponse ErrorMessage pour GSSENCRequest provenant du serveur. Le client ne doit pas afficher ce message d'erreur à l'utilisation/application carle serveur n'a pas été authentifié (CVE-2024-10977[4]). Dans ce cas, la connexion doit être termée, mais le client peut choisir d'ouvrir une nouvelle connexion et de continuer sans demander de chiffrement GSSAPI.

[1] https://www.postgresql.org/support/security/CVE-2024-10977/
[2] https://www.postgresql.org/support/security/cve-2021-23222/
[3] https://datatracker.ietf.org/doc/html/rfc2744
[4] https://www.postgresql.org/support/security/CVE-2024-10977/

Quand le chiffrement GSSAPI peut être réalisé, le serveur est supposé envoyer l'octet `G`, puis attendre que le client initie une poignée de main GSSAPI. Si des octets supplémentaires sont disponibles en lecture, cela signifie probablement qu'une attaque de type *man-in-the-middle* est en cours et tente de réaliser la technique *buffer-stuffing* (CVE-2021-23222[5]). Les clients doivent être codés soit pour lire exactement un octet de la socket avant de renvoyer la socket à leur bibliothèque GSSAPI ou de la traiter comme une violation du protocole s'ils voient qu'ils ont déjà des octets supplémentaires à lire.

Un message initial GSSENCRequest peut aussi être utilisé dans une connexion en cours d'ouverture pour envoyer un message CancelRequest.

Une fois que le chiffrement GSSAPI a été établi avec succès, le client utilise la fonction `gss_wrap()` pour chiffrer le message StartupMessage habituel et toutes les données qui suivent, ajoutant la longueur du résultat de `gss_wrap()` sous la forme d'un entier sous quatre octets dans l'ordre d'octets du réseau à la charge chiffrée réelle. Notez que le serveur acceptera seulement des paquets chiffrés provenant du client qui font moins de 16 Ko ; `gss_wrap_size_limit()` devrait être utilisé par le client pour déterminer la taille du message non chiffré qui sera contenu dans cette limite, et les messages plus larges devront être divisés en plusieurs appels à `gss_wrap()`. Les segments typiques sont 8 Ko de données non chiffrées, résultant en des paquets chiffrés légèrement supérieurs à 8 Ko mais bien contenu dans les 16 Ko maximum. Il est attendu que le serveur n'envoie pas de paquets de plus de 16 ;Ko au client.

Alors que le protocole lui-même ne fournit pas de moyens pour que le serveur force un chiffrement GSSAPI, l'administrateur peut configurer le serveur de telle façon qu'il rejette les sessions non chiffrées. Cela se fait pendant la vérification de l'authentification.

55.3. Authentification SASL

SASL est un cadre pour l'authentification dans des protocoles orientés connexions. Actuellement, PostgreSQL implémente deux mécanismes d'authentification SASL, SCRAM-SHA-256 et SCRAM-SHA-256-PLUS. D'autres pourraient être ajoutés dans le futur. Les étapes ci-dessous illustrent comment l'authentification SASL est réalisé en général, alors que la sous-section suivante donne plus de détails sur SCRAM-SHA-256 et SCRAM-SHA-256-PLUS.

Flot de message d'authentification SASL

1. Pour commencer un échange d'authentification SASL, le serveur envoie un message AuthenticationSASL. Il inclut une liste des mécanismes d'authentification SASL que le serveur peut accepter, dans l'ordre de préférence du serveur.

2. Le client sélectionne un des mécanismes supportés dans la liste, et envoie un message SASLInitialResponse au serveur. Le message inclut le nom du mécanisme sélectionné, et un optionnel `Initial Client Response`, si le mécanisme sélectionné utilise cela.

3. Un ou plusieurs messages `server-challenge` et `client-response` suivront. Chaque message `server-challenge` est envoyé dans un message AuthenticationSASLContinue, suivi par une réponse d'un client dans un message SASLResponse. Les particularités des messages sont spécifiques au mécanisme.

4. Enfin, quand l'échange d'authentification est terminé avec succès, le serveur envoie un message AuthenticationSASLFinal, suivi immédiatement par un message AuthenticationOk. Le message AuthenticationSASLFinal contient des données supplémentaires du serveur au client, dont le contenu est spécifique au mécanisme d'authentification sélectionné. Si le mécanisme d'authentification n'utilise pas les données supplémentaires envoyées en fin, le message AuthenticationSASLFinal n'est pas envoyé.

En cas d'erreur, le serveur peut annuler l'authentification à tout moment, et envoie un message ErrorMessage.

[5] https://www.postgresql.org/support/security/cve-2021-23222/

55.3.1. Authentification SCRAM-SHA-256

Les mécanismes SASL implémentés actuellement sont `SCRAM-SHA-256` et son variant avec la liaison de canal `SCRAM-SHA-256-PLUS`. Ils sont décrits en détails dans RFC 7677[6] et RFC 5802[7].

Quand SCRAM-SHA-256 est utilisé dans PostgreSQL, le serveur ignorera le nom d'utilisateur que le client envoie dans `client-first-message`. Le nom d'utilisateur qui était déjà envoyé dans le message de démarrage est utilisé à la place. PostgreSQL accepte plusieurs encodages de caractères, alors que SCRAM impose l'utilisation d'UTF-8 pour le nom d'utilisateur, donc il pourrait être impossible de représenter le nom d'utilisateur PostgreSQL en UTF-8.

La spécification SCRAM impose que le mot de passe soit aussi en UTF-8, et soit traité avec l'algorithme *SASLprep*. Néanmoins, PostgreSQL ne requiert pas l'utilisation d'UTF-8 pour le mot de passE. Quand le mot de passe d'un utilisateur est configuré, il est traité avec SASLprep comme s'il était en UTF-8, quelque soit l'encodage réel. Néanmoins, s'il ne s'agit pas d'une séquence légale d'octets UTF-8 ou s'il contient des séquences d'octets UTF-8 qui sont interdites par l'algorithme SASLprep, le mot de passe brut sera utilisé sans le traitement de SASLprep, au lieu de renvoyer une erreur. Ceci permet au mot de passe d'être normalisé quand il est en UTF-8, en autorisant toujours l'utilisation d'un mot de passe non UTF-8, sans nécessiter que le système connaisse l'encodage du mot de passe.

La liaison de canal (*Channel binding*) est accepté par PostgreSQL si ce dernier a été compilé avec le support de SSL. Le nom du mécanisme SASL pour SCRAM avec liaison de canal est `SCRAM-SHA-256-PLUS`. Le type de liaison utilisé par PostgreSQL est `tls-server-end-point`.

Dans SCRAM sans liaison de canal, le serveur choisit un nombre aléatoire qui est transmis au client pour être mélangé avec le mot de passe fourni par l'utilisateur dans le hachage transmis du mot de passe. Bien que ceci empêche le hachage du mot de passe d'être retransmis avec succès dans une session ultérieure, cela n'empêche pas un faux serveur entre le serveur réel et le client de passer la valeur aléatoire du serveur et de s'authentifier avec succès.

SCRAM avec liaison de canal empêche ce type d'attaques *man-in-the-middle* en mélangeant la signature du certificat du serveur dans le hachage transmis du mot de passe. Bien qu'un faux serveur peut retransmettre le certificat du vrai serveur, il n'a pas accès à la clé privée correspondant à ce certificat et ne peut donc pas prouver qu'il en est le propriétaire, imposant de ce fait un échec de la connexion SSL.

Exemple

1. Le serveur envoie un message AuthenticationSASL. Il inclut une liste des mécanismes d'authentification SASL que le serveur peut accepter. Ce sera `SCRAM-SHA-256-PLUS` et `SCRAM-SHA-256` si le serveur dispose du support de SSL. Dans le cas contraire, ce sera uniquement le dernier.

2. Le client répond en envoyant un message SASLInitialResponse, qui indique le mécanisme choisi, `SCRAM-SHA-256` ou `SCRAM-SHA-256-PLUS`. (Un client est libre de choisir un mécanisme mais pour une sécurité plus forte, il devra choisir la variante à liaison de canal s'il le gère.) Dans le champ `Initial Client response`, le message contient le message SCRAM `client-first-message`. Le message `client-first-message` contient aussi le type de liaison de canal choisi par le client.

3. Le serveur envoie un message AuthenticationSASLContinue, avec un message SCRAM `server-first-message` comme contenu.

4. Le client envoie un message SASLResponse, avec le message SCRAM `client-final-message` comme contenu.

[6] https://datatracker.ietf.org/doc/html/rfc7677
[7] https://datatracker.ietf.org/doc/html/rfc5802

5. Le serveur envoie un message AuthenticationSASLFinal, avec le message SCRAM `server-final-message`, suivi immédiatement par un message AuthenticationOk.

55.4. Protocole de réplication en flux

Pour initier une réplication en flux, le client envoie le paramètre `replication` dans le message de démarrage. Une valeur booléenne à `true` (ou `on`, `yes`, `1`) indique au backend d'entrer dans le mode walsender de réplication physique, dans lequel un petit ensemble de commandes de réplication, affichées ci-dessous, peut être exécuté à la place de requêtes SQL.

Passer `database` comme valeur du paramètre `replication` instruit le backend à entrer en mode walsender de réplication logique, se connectant à la base de données spécifiée par le paramètre `dbname`. Dans le mode walsender de réplication logique, les commandes de réplication affichées ci-dessous peuvent être exécutées ainsi que des commandes SQL standards.

Dans les deux modes walsender, seul le protocole de requête simple peut être utilisé.

Dans le but de tester des commandes de réplication, vous pouvez faire une connexion de réplication via l'outil psql ou tout autre outil utilisant la bibliothèque libpq avec une chaîne de connexion incluant l'option `replication`, par exemple :

```
psql "dbname=postgres replication=database" -c "IDENTIFY_SYSTEM;"
```

Néanmoins, il est souvent plus utile d'utiliser pg_receivewal (pour la réplication physique) ou pg_recvlogical (pour la réplication logique).

Les commandes de réplication sont tracées dans les journaux applicatifs du serveur quand log_replication_commands est activé.

Les commandes acceptées en mode réplication sont :

`IDENTIFY_SYSTEM`

Demande au serveur de s'identifier. Le serveur répond avec un ensemble de résultats contenant une ligne composée de quatre champs :

`systemid` (`text`)

L'identifiant système unique de l'instance. Ceci peut être utilisé pour vérifier que la sauvegarde utilisée pour initialiser le serveur secondaire provient bien de la même instance.

`timeline` (`int4`)

Identifiant de timeline actuel. Aussi utile pour vérifier que le serveur secondaire est cohérent avec le primaire.

`xlogpos` (`text`)

Emplacement actuel de vidage des journaux de transactions. Utile pour obtenir un emplacement connu dans les journaux de transactions où commencer le flux.

`dbname` (`text`)

Base de données où se connecter, ou NULL.

`SHOW` *`nom`*

Demande au serveur d'envoyer la configuration actuelle d'un paramètre. C'est similaire à la commande SQL SHOW.

nom

Le nom d'un paramètre. Les paramètres disponibles sont documentées dans Chapitre 20.

TIMELINE_HISTORY *tli*

Demande au serveur d'envoyer le fichier d'historique des timelines pour la timeline *tli*. Le serveur répond avec un ensemble de résultat d'une seule ligne comprenant deux champs. Bien que les champs soient du type text, elles renvoient réellement des octets bruts sans conversion d'encodage :

filename (text)

Nom du fichier d'historique des timelines, par exemple 00000002.history.

content (text)

Contenu du fichier d'historique des timelines.

CREATE_REPLICATION_SLOT *nom_slot* [TEMPORARY] { PHYSICAL | LOGICAL *plugin_sortie* } [(*option* [, ...])]

Crée un slot de réplication physique ou logique. Voir Section 27.2.6 pour plus d'informations sur les slots de réplication.

nom_slot

Le nom du slot à créer. Ce doit être un nom valide pour un slot de réplication (voir Section 27.2.6.1).

output_plugin

Le nom du plugin de sortie utilisé pour le décodage logique (voir Section 49.6).

TEMPORARY

Indique que ce slot de réplication est un slot temporaire. Les slots temporaires ne sont pas sauvegardés sur disque et sont automatiquement supprimés en cas d'erreur ou quand la session est terminée.

Les options suivantes sont acceptées :

TWO_PHASE [*boolean*]

Si true, ce slot de réplication logique accepte le décodage des validations en deux phases. Avec cette option, les commandes relatives au 2PC telles que PREPARE TRANSACTION, COMMIT PREPARED et ROLLBACK PREPARED sont décodées et transmises. La transaction sera décodée et transmise au moment du PREPARE TRANSACTION. La valeur par défaut est false.

RESERVE_WAL [*boolean*]

Si true, ce slot de réplication physique réserve immédiatement les WAL. Sinon, les WAL sont seulement réservés à la connexion d'un client de réplication en flux. La valeur par défaut est false.

SNAPSHOT { 'export' | 'use' | 'nothing' }

Décide ce qu'il faut faire du snapshot créé lors de l'initialisation du slot logique. 'export', qui est la valeur par défaut, exportera le snapshot à utiliser dans d'autres sessions. Cette option ne peut pas être utilisée à l'intérieur d'une transaction. 'use' utilisera le snapshot pour la transaction courante exécutant la commande. Cette option doit être utilisée dans une

transaction, et CREATE_REPLICATION_SLOT doit être la première commande exécutée dans cette transaction. Enfin, 'nothing' utilisera uniquement le snapshot pour le décodage logique comme d'habitude mais ne fera rien de plus avec.

En réponse à cette commande, le serveur enverra un ensemble de résultat d'une ligne contenant les champs suivants :

slot_name (text)

Le nom du slot de réplication nouvellement créé.

consistent_point (text)

L'emplacement dans les WAL à partir duquel le slot devient cohérent. C'est l'emplacement le plus proche à partir duquel le flux peut commencer sur ce slot de réplication.

snapshot_name (text)

L'identifiant du snapshot exporté par la commande. Le snapshot est valide jusqu'à l'exécution d'une nouvelle commande sur cette connexion ou jusqu'à la fermeture de la connexion de réplication. NULL si le slot créé est physique.

output_plugin (text)

Le nom du plugin de sortie utilisé par le slot de réplication nouvellement créé. NULL si le slot créé est physique.

CREATE_REPLICATION_SLOT *slot_name* [TEMPORARY] { PHYSICAL [RESERVE_WAL] | LOGICAL *output_plugin* [EXPORT_SNAPSHOT | NOEXPORT_SNAPSHOT | USE_SNAPSHOT | TWO_PHASE] }

Pour la compatibilité avec les anciennes versions, cette syntaxe alternative pour CREATE_REPLICATION_SLOT est toujours acceptée.

READ_REPLICATION_SLOT *slot_name*

Lit certaines informations associées avec un slot de réplication. Renvoie une ligne avec des valeurs NULL si le slot de réplication n'existe pas. Cette commande est actuellement uniquement acceptée pour les slots de réplication physique.

En réponse à cette commande, le serveur renverra un ensemble de résultats d'une ligne, contenant les champs suivants :

slot_type (text)

Le type du slot de réplication, soit physical soit NULL.

restart_lsn (text)

Le restart_lsn du slot de réplication.

restart_tli (int8)

L'identifiant de timeline associé avec restart_lsn, suivant l'historique actuelle des timelines.

START_REPLICATION [SLOT *slot_name*] [PHYSICAL] *XXX/XXX* [TIMELINE *tli*]

Demande au serveur de commencer le flux de WAL, en commençant à l'emplacement *XXX/XXX*. Si l'option TIMELINE est spécifiée, le flux commence sur la timeline *tli* ; sinon la timeline actuelle du serveur est sélectionnée. Le serveur peut répondre avec une erreur, par exemple si la section demandée de WAL a déjà été recyclée. En cas de succès, le serveur répond avec un message CopyBothResponse, puis commence le flux de WAL vers le client.

Si un nom de slot est fourni via *slot_name*, il sera mis à jour au fur et à mesure de la progression de la réplication pour que le serveur se rappelle des segments WAL et, si hot_standby_feedback vaut on, des transactions toujours nécessaires pour le secondaire.

Si le client demande une timeline qui n'est pas la plus récente mais qui fait partie de l'historique du serveur, celui-ci diffusera l'intégralité du WAL correspondant à cette timeline, depuis le point de départ demandé jusqu'au moment où le serveur a changé de timeline. Si le client demande la diffusion en continu exactement à la fin d'une ancienne timeline, le serveur ignore complètement le mode COPY.

Après le flux de tous les WAL d'une timeline qui n'est pas la dernière, le serveur terminera le flux en sortant du mode COPY. Quand le client reconnait cette situation en quittant lui aussi le mode COPY, le serveur envoie un ensemble de résultats d'une ligne et de deux colonnes, indiquant la prochaine timeline dans l'historique du serveur. La première colonne est l'identifiant de la prochaine timeline (type int8), et la deuxième colonne est l'emplacement des WAL où la bascule a eu lieu (type text). Habituellement, la position de bascule est la fin du WAL en cours de flux, mais il existe des cas particuliers où le serveur peut envoyer certains WAL de l'ancienne timeline qu'il n'a pas encore rejoué avant la promotion. Enfin, le serveur envoie deux messages CommandComplete (un qui termine le CopyData et l'autre qui termine le START_REPLICATION lui-même), et est prêt à accepter une nouvelle commande.

Les données WAL sont envoyées sous la forme d'une série de messages CopyData. (Ceci permet le mélange d'autres informations ; en particulier, le serveur peut envoyer un message ErrorResponse s'il rencontre un échec après le début du flux.) La charge de chaque message CopyData du serveur vers le client contient un message sur un des formats suivants :

XLogData (B)

Byte1('w')

Identifie le message comme des données WAL.

Int64

Le début des données WAL dans ce message.

Int64

La fin actuelle des WAL sur le serveur.

Int64

L'horloge système du serveur au moment de la transmission, en microsecondes depuis le 1er janvier 2000 à minuit.

Byte*n*

Une section du flux de données WAL.

Un enregistrement WAL seul n'est jamais divisé en deux messages XLogData. Quand un enregistrement WAL arrive sur une limite de page d'un WAL et est, de ce fait, déjà divisé en utilisant les enregistrements de continuité, il peut être divisé sur la limite de la page. Autrement dit, le premier enregistrement WAL principal et ses enregistrements de continuité peuvent être envoyés dans différents messages XLogData.

Primary keepalive message (B)

Byte1('k')

Identifie le message comme un keepalive de l'envoyeur.

Int64

La fin actuelle du WAL sur le serveur.

Int64

L'horloge système du serveur au moment de la transmission, en microsecondes depuis le 1er janvier 2000.

Byte1

1 signifie que le client devrait répondre à ce message aussi vite que possible pour éviter une déconnexion par dépassement de délai d'attente.

Le processus en réception peut envoyer des réponses à l'émetteur à tout moment, en utilisant un des formats de message suivants (aussi dans la charge d'un message CopyData) :

Standby status update (F)

Byte1('r')

Identifie le message comme mise à jour du statut du receveur.

Int64

L'emplacement du dernier octet du WAL + 1 réçu et écrit sur disque par le secondaire.

Int64

L'emplacement du dernier octet du WAL + 1 vidé sur disque par le secondaire.

Int64

L'emplacement du dernier octet du WAL + 1 appliqué sur disque par le secondaire.

Int64

L'horloge système du serveur au moment de la transmission, en microsecondes depuis le 1er janvier 2000.

Byte1

Si 1, le client demande au serveur de répondre immédiatement à ce message. Ceci peut être utilisé pour appeler le serveur, pour tester si la connexion est toujours active.

Hot standby feedback message (F)

Byte1('h')

Identifie le message comme un message de retour d'informations du serveur secondaire.

Int64

L'horloge système du serveur au moment de la transmission, en microsecondes depuis le 1er janvier 2000.

Int32

Le xmin global actuel du secondaire, excluant le catalog_xmin de tout slot de réplication. Si cette valeur et le catalog_xmin suivant sont à 0, ceci est traité comme une notification

que le retour du secondaire ne sera plus envoyé sur cette connexion. Des messages ultérieurs différents de zéro pourraient ré-initier le mécanisme de retour.

Int32

Le nombre epoch du xmin xid global sur le serveur secondaire.

Int32

Le catalog_xmin le plus bas de tout slot de réplication sur le secondaire. Configuré à 0 si aucun catalog_xmin n'existe sur le secondaire ou si le retour d'informations du secondaire est en cours de désactivation.

Int32

Le nombre epoch du XID catalog_xmin sur le secondaire.

`START_REPLICATION SLOT` *`slot_name`* `LOGICAL` *`XXX/XXX`* [(*`option_name`* [*`option_value`*] [, ...])]

Demande au serveur de commencer le flux de WAL pour la réplication logique, en commençant soit à la position WAL *`XXX/XXX`* soit à la position `confirmed_flush_lsn` du slot (voir Section 54.19), suivant la position la plus grande. Ce comportement facilite la tache pour les clients qui évitent ainsi de mettre à jour le statut LSN local quand il n'y a aucune donnée à traiter. Cependant, commencer à un LSN différent de celui demandé pourrait ne pas récupérer certaines types d'erreurs clients ; donc le client pourrait souhaiter vérifier que `confirmed_flush_lsn` correspond à ses attentes avant de lancer `START_REPLICATION`.

Le serveur peut répondre avec une erreur, par exemple si le slot n'existe pas. En cas de succès, le serveur répond avec un message CopyBothResponse, puis lance le flux de WAL vers le client.

Le message à l'intérieur des messages CopyBothResponse sont du même format documenté que pour `START_REPLICATION ... PHYSICAL`, ceci incluant les deux messages CommandComplete.

Le plugin de sortie associé avec le slot sélectionné est utilisé pour traiter la sortie pour le flux.

`SLOT` *`slot_name`*

Le nom du slot pour lequel envoyer en flux les changements. Ce paramètre est requis, et doit correspondre à un slot existant de réplication logique créé avec `CREATE_REPLICATION_SLOT` dans le mode `LOGICAL` mode.

`XXX/XXX`

L'emplacement dans le WAL à partir duquel commence le flot.

`option_name`

Le nom d'une option passée au plugin de décodage logique du slot. Voir Section 55.5 pour les options qui sont acceptées par le plugin standard (`pgoutput`).

`option_value`

Valeur optionnelle, sous le forme d'une constante de type chaîne de caractères, associée à l'option indiquée.

`DROP_REPLICATION_SLOT` *`slot_name`* [`WAIT`]

Supprime un slot de réplication, libérant toute ressource réservée.

slot_name

Le nom du slot à supprimer.

WAIT

Cette option force la commande à attendre si le slot est actif et jusqu'à ce que ce dernier devienne inactif. Dans le cas contraire, une erreur est levée (comportement par défaut).

BASE_BACKUP [(*option* [, ...])]

Demande au serveur le lancement d'une sauvegarde en flux. Le système sera automatiquement placé en mode sauvegarde avant que la sauvegarde ne commence, et sorti de ce mode quand la sauvegarde est terminée. Les options suivantes sont acceptées :

LABEL *'label'*

Configure le label de la sauvegarde. Si aucun label n'est donné, le label `base backup` sera utilisé. Les règles de guillemet pour le label sont les mêmes que pour une chaîne SQL standard lorsque standard_conforming_strings vaut `on`.

TARGET *'target'*

Indique au serveur où envoyer la sauvegarde. Si la cible est `client` (ce qui est la valeur par défaut), les données de la sauvegarde sont envoyées au client. Si elle vaut `server`, les données de la sauvegarde sont écrites sur le serveur dans le chemin indiqué par l'option TARGET_DETAIL. Si elle vaut `blackhole`, les données de la sauvegarde ne sont pas envoyées ; elles sont simplement ignorées.

La cible `server` nécessite l'attribut SUPERUSER ou être membre du rôle `pg_write_server_files`.

TARGET_DETAIL *'detail'*

Fournit des informations supplémentaires sur la cible de sauvegarde.

Actuellement, cette option peut seulement être utilisée quand la cible de sauvegarde est `server`. Elle indique le répertoire du serveur, répertoire dans lequel la sauvegarde sera écrite.

PROGRESS [*boolean*]

Si configuré à true, réclame des informations nécessaires pour générer un rapport de progression. Cela enverra une taille approximative dans l'en-tête de chaque tablespace, qui peut être utilisé pour calculer le reste du flux. Le calcul se fait en énumérant toutes les tailles de fichier avant que le transfert ne commence, et cela pourrait avoir un impact négatif sur les performances. En particulier, cela pourrait prendre du temps avant que les premières données ne soient envoyées. Comme les fichiers de la base peuvent évoluer pendant la sauvegarde, la taille n'est qu'approximative, et pourrait soit augmenter soit diminuer entre le moment de l'approximation et celui de l'envoi des fichiers. La valeur par défaut est false.

CHECKPOINT { 'fast' | 'spread' }

Configure le type de checkpoint à réaliser au début de la sauvegarde. La valeur par défaut est `spread`.

WAL [*boolean*]

Si configuré à true, inclut les segments WAL nécessaires dans la sauvegarde. Cela inclura tous les fichiers entre le début et la fin de la sauvegarde, et ils seront placés dans le répertoire `pg_wal` du répertoire principal. La valeur par défaut est false.

`WAIT [ boolean ]`

Si configuré à true, la sauvegarde attendra jusqu'à ce que le dernier segment WAL requis soit archivé ou émettra un message d'avertissement si l'archivage des journaux de transactions n'est pas activé. Si false, la sauvegarde n'attendra pas et ne préviendra pas, laissant le client responsable de la vérification de la disponibilité du dernier segment WAL. La valeur par défaut est false.

`COMPRESSION 'method'`

Demande au serveur de compresser la sauvegarde en utilisant la méthode indiquée. Actuellement, les méthodes acceptées sont `gzip`, `lz4` et `zstd`.

`COMPRESSION_DETAIL detail`

Donne les détails sur la méthode de compression choisie. Ceci devrait être utilisé en conjonction avec l'option `COMPRESSION`. Si la valeur est un entier, elle indique le niveau de compression. Sinon, elle peut être une liste d'éléments séparés par des virgules, chaque élément étant de la forme *`motclé`* ou *`motclé=valeur`*. Actuellement, les mots clés acceptés sont `level` et `workers`.

Le mot clé `level` configure le niveau de compression. Pour `gzip`, le niveau de compression devrait être un entier compris entre `1` et `9` (par défaut `Z_DEFAULT_COMPRESSION` ou `-1`), pour `lz4` un entier compris entre 1 et 12 (par défaut `0` pour le mode de compression rapide), et pour `zstd` un entier compris entre `ZSTD_minCLevel()` (habituellement `-131072`) et `ZSTD_maxCLevel()` (habituellement `22`), (par défaut `ZSTD_CLEVEL_DEFAULT` ou `3`).

Le mot clé `workers` configure le nombre de threads devant être utilisé pour paralléliser la compression. Cette fonctionnalité n'est disponible qu'avec `zstd`.

`MAX_RATE rate`

Limite la quantité maximale de données transférées du serveur au client par unité de temps. L'unité attendue est du Ko par seconde. Si cette option est indiqué, la valeur doit être soit égale à zéro, soit être compris dans l'intervalle allant de 32 ko à 1 Go (valeurs inclues). Si un zéro est passé ou si l'option n'est pas indiquée, aucune restriction n'est imposée pendant le transfert.

`TABLESPACE_MAP [ boolean ]`

Si true, inclut des informations sur les liens symboliques présents dans le répertoire `pg_tblspc` dans un fichier nommé `tablespace_map`. Le fichier de cartographie des tablespaces inclut, pour chaque tablespace, le nom du lien symbolique dans le répertoire `pg_tblspc/` et le chemin complet de ce lien symbolique. La valeur par défaut est false.

`VERIFY_CHECKSUMS [ boolean ]`

Si true, les sommes de contrôle sont vérifiées lors d'une sauvegarde si elles sont activées. Si false, elles sont ignorées. La valeur par défaut est true.

`MANIFEST manifest_option`

Quand cette option est indiquée avec une valeur `yes` ou `force-encode`, un manifest de sauvegarde est créé et envoyé avec la sauvegarde. Le manifeste est une liste de chaque fichier présent dans la sauvegarde à l'exception des fichiers WAL qui y sont inclus. Il enregistre aussi la taille, l'horodatage de la dernière modification et en option une somme de contrôle pour chaque fichier. Une valeur de `force-encode` force tous les noms de fichier à être codé en hexadécimal ; sinon ce type d'encodage est réalisé seulement pour les fichiers dont le nom est une séquence d'octets non UTF8. `force-encode` existe principalement pour des tests, pour être sûr que les clients qui lisent le manifeste de sauvegarde puissent gérer ce cas. Pour la compatibilité avec les anciennes versions, la valeur par défaut est `MANIFEST 'no'`.

`MANIFEST_CHECKSUMS` *`checksum_algorithm`*

Précise l'algorithme des sommes de contrôle, à appliquer à chaque fichier inclus dans le manifeste de sauvegarde. Actuellement, les seuls algorithmes disponibles sont `NONE`, `CRC32C`, `SHA224`, `SHA256`, `SHA384` et `SHA512`. La valeur par défaut est `CRC32C`.

Quand la sauvegarde est lancée, le serveur enverra tout d'abord deux ensembles de résultats ordinaires, suivis par un ou plusieurs résultats CopyOutResponse.

Le premier ensemble de résultat contient la position de départ de la sauvegarde, sur une liste ligne comprenant deux colonnes. La première colonne contient la position de départ donné dans un format correspondant à XLogRecPtr, et la deuxième colonne contient l'identifiant de la timeline correspondante.

Le deuxième ensemble de résultat a une ligne pour chaque tablespace. Voici la liste des champs de cette ligne :

`spcoid` (`oid`)

L'OID du tablespace, ou NULL s'il s'agit du répertoire principal.

`spclocation` (`text`)

Le chemin complet du répertoire du tablespace, ou NULL si c'est le répertoire principal.

`size` (`int8`)

La taille approximative du tablespace, en kilooctets (1024 octets), si le rapport de progression a été demandé ; sinon NULL.

Après le deuxième ensemble de résultats, un message CopyOutResponse sera envoyé. La chaque de chaque message CopyData contiendra un message dans un des formats suivants :

new archive (B)

Byte1('n')

Identifie le messaage comme indiquant le début d'une nouvelle archive. Il y a aura une archive pour le répertoire principal des données et une archive pour chaque tablespace supplémentaire ; chacune utilisera le format tar (en suivant le format « ustar interchange format » spécifié dans le standard POSIX 1003.1-2008).

String

Le nom du fichier pour cette archive.

String

Pour le répertoire principal des données, une chaîne vide. Pour les autres tablespaces, le chemin complet vers le répertoire à partir duquel cette archive a été créée.

manifest (B)

Byte1('m')

Identifie le message comme indiquant le début d'un manifeste de sauvegarde.

archive or manifest data (B)

Byte1('d')

Identifie le message comme contenant une archive ou des données du manifeste.

Byte*n*

Octets des données.

progress report (B)

Byte1('p')

Identifie le message comme un rapport de progression.

Int64

Le nombre d'octets du tablespace actuel pour lequel le traitement a terminé.

Après que le CopyOutResponse et tous ces types de réponses aient été envoyés, un ensemble de résultat ordinaire final sera envoyé, contenant la position finale dans les WAL de la sauvegarde, dans le même format que la position de départ.

L'archive tar pour le répertoire de données principal et pour chaque tablespace contiendra tous les fichiers des répertoires, qu'ils appartiennent à PostgreSQL ou pas (fichiers ajoutées dans le même répertoire). Les fichiers exclus sont :
- `postmaster.pid`
- `postmaster.opts`
- `pg_internal.init` (trouvé dans plusieurs répertoires)
- Différents fichiers et répertoires temporaires créés pendant les opérations du serveur PostgreSQL, tels que les fichiers ou répertoire commençant avec `pgsql_tmp` et les objets temporaires.
- Relations non journalisées, sauf pour le type init qui est requis pour recréer à vide les relations non journalisés lors d'une restauration.
- `pg_wal`, en incluant les sous-répertoires. Si la sauvegarde est exécuté avec les fichiers WAL inclus, une version synthétisée de `pg_wal` sera inclus, mais elle contiendra seulement les fichiers nécessaires pour que la sauvegarde fonctionne, et pas le reste du contenu.
- `pg_dynshmem`, `pg_notify`, `pg_replslot`, `pg_serial`, `pg_snapshots`, `pg_stat_tmp` et `pg_subtrans` sont copiés sous la forme de répertoires vides (même s'il s'agit de lien symbolique).
- Fichiers autres que les fichiers et répertoires standards, tels que des liens symboliques (autre que pour les répertoires indiqués ci-dessus), les fichiers périphériques et les fichiers du système d'exploitation. Les liens symboliques dans `pg_tblspc` sont maintenus.)

Propriétaire, groupe et droits sont configurés si le système de fichiers sous-jacent l'accepte.

55.5. Logical Streaming Replication Protocol

Cette section décrit le protocole de réplication logique qui consiste en un flux de message démarré par la commande `START_REPLICATION SLOT` *`nom_slot`* `LOGICAL`.

Le protocole de réplication logique en flux se base sur les primitives du protocole de réplication physique en flux.

Le décodage logique PostgreSQL accepte des plugins de sortie. `pgoutput` est le plugin standard utilisé par la réplication logique native.

55.5.1. Paramètres de la réplication logique en flux

En utilisant la commande de réplication logique `START_REPLICATION`, le plugin `pgoutput` accepte les options suivantes :

proto_version

La version du protocole. Actuellement, les versions `1`, `2` et `3` sont supportées. Une option valide est requise.

La version 2 n'est supportée qu'à partir de la version 14 et permet la transmission de transactions volumineuses non terminées.

La version 3 n'est supportée qu'à partir de la version 15 et permet l'utilisation de la validation en deux phases (2PC) avec la réplication logique.

publication_names

Liste séparée par des virgules contenant les noms de publication auxquelles s'abonner (recevoir les modifications). Les noms individuels de publication sont traités comme des noms d'objets standard et peuvent être mis entre guillemets si nécessaire. Il est requis d'indiquer au moins un nom de publication.

binary

Option booléenne pour utiliser le mode de transfert binaire. Le mode binaire est plus rapide que le mode texte mais un peu moins robuste.

messages

Option booléenne pour activer l'envoi de messages qui sont écrits par `pg_logical_emit_message`.

streaming

Option booléenne pour activer l'envoi en flux des transactions en cours. Il est requis d'activer au minimum la version 2 du protocole.

two_phase

Option booléenne pour activer les transactions en deux phases. Il est requis d'activer au minimum la version3 du protocole.

55.5.2. Messages du protocole de réplication logique

Les messages individuels du protocole sont discutés dans les sous-sections suivantes. Les messages individuels sont décrits dans Section 55.9.

Tous les message haut niveau du protocole commencent avec un octet de type de message. Bien que représenté dans le code comme un caractère, c'est en fait un octet signé sans encodage associé.

Comme le protocole de réplication en flux fournit une longueur de message, il n'est pas nécessaire que les messages haut niveau du protocole embarquent une longueur dans leur en-tête.

55.5.3. Flot des messages du protocole de réplication logique

À l'exception de la commande `START_REPLICATION` et des messages de progression du rejeu, toutes les informations se déplacent uniquement du backend vers le client.

Le protocole de réplication logique envoie les transactions individuelles l'une après l'autre. Cela signifie que tous les messages entre une paire de messages Begin et Commit appartiennent à la même transaction. De façon similaire, tous les messages entre une paire de messages Begin Prepare et Prepare appartiennent à la même transaction. Il envoie aussi les changements de grosses transactions en cours entre une paire de messages Stream Start et Stream Stop. Le dernier lot d'une telle transaction contient un message Stream Commit ou Stream Abort.

Chaque transaction envoyée contient zéro ou plusieurs messages DML (Insert, Update, Delete). Dans le cas d'une configuration en cascade, elle peut aussi contenir des messages Origin. Le message origine

indique que la transaction provient d'un autre nœud de réplication. Comme un nœud de réplication peut être à peu près tout dans le cas du protocole de réplication logique, le seul identifiant est le nom de l'origine. C'est au client de gérer ceci. Le message Origin est toujours envoyé avant tout message DML dans la transaction.

Chaque message DML contient un OID de relation, identifiant la relation concernée. Avant le premier message DML pour un OID de relation donné, un message Relation sera envoyé, décrivant le schéma de cette relation. Après cela, un nouveau message Relation sera envoyé si la définition de la relation a changé depuis l'envoi du dernier message Relation. (Le protocole suppose que le client est capable de se rappeler de cette méta donnée pour toutes les relations impliquées.)

Les messages Relation identifient les types des colonnes par leur OID. Dans le cas d'un type interne, il est supposé que le client peut rechercher cet OID localement, donc aucune donnée supplémentaire n'est nécessaire. Pour les OID de type utilisateur, un message Type sera envoyé avant le message Relation pour fournir le nom du type associé à cet OID. De ce fait, un client qui a besoin d'identifier spécifiquement les types des colonnes d'une relation devrait placer en cache le contenu des messages Type, et consulter ce cache pour vérifier si l'OID du type y est défini. Dans le cas contraire, il sera possible de rechercher localement l'OID du type.

55.6. Types de données des messages

Cette section décrit les types de données de base utilisés dans les messages.

Int*n*(*i*)

Un entier sur *n* bits dans l'ordre d'octet réseau (l'octet le plus significatif en premier). Si *i* est précisé, il s'agit de la valeur exacte qui apparaîtra, sinon la valeur est variable. Par exemple, Int16, Int32(42).

Int*n*[*k*]

Un tableau d'entiers de *k* *n* bits, chacun dans l'ordre d'octet réseau. La longueur du tableau *k* est toujours déterminée par un champ précédent dans le message. Par exemple, Int16[M].

String(*s*)

Une chaîne terminée par un octet nul (chaîne du type C). Il n'y a pas de limitation spécifique de longueur sur les chaînes. Si *s* est indiqué, il s'agit de la valeur exacte qui apparaîtra. Sinon la valeur est variable. Par exemple, String, String("user").

> **Note**
>
> *Il n'y a pas de limite prédéfinie* sur la longueur d'une chaîne qui peut être renvoyée par le backend. Une bonne stratégie de codage pour un client est d'utiliser un tampon expansif pour que tout ce qui puisse être stocké en mémoire soit accepté. Si ce n'est pas faisable, lire la chaîne et ignorer les caractères en fin qui ne rentrent pas dans le tampon de taille fixe.

Byte*n*(*c*)

Exactement *n* octets. Si la largeur du champ *n* n'est pas une constante, il est toujours déterminable à partir d'un champ précédent dans le message. Si *c* est indiqué, il s'agit de la valeur exacte. Par exemple, Byte2, Byte1('\n').

55.7. Formats de message

Cette section décrit le format détaillé de chaque message. Tous sont marqués pour indiquer qu'ils peuvent être envoyés par un client (F, pour *frontend* en anglais), par un backend (B) ou les deux (F &

B). Notez que, bien que chaque message inclut un décompte d'octets au début, le format de message est défini pour que la fin du message puisse être trouvée sans faire référence au nombre d'octets. Ceci aide à vérifier la validité du message. (Le message CopyData est une exception car il fait partie d'un flux de données ; le contenu de tout message CopyData individuel ne peut pas être interprété indépendamment.)

AuthenticationOk (B)

Byte1('R')

Identifie le message comme une demande d'authentification.

Int32(8)

Longueur du contenu du message en octets, longueur inclue.

Int32(0)

Indique que l'authentification est réussie.

AuthenticationKerberosV5 (B)

Byte1('R')

Identifie le message comme une demande d'authentification.

Int32(8)

Longueur du contenu du message en octets, longueur inclue.

Int32(2)

Indique que l'authentification Kerberos V5 est requise.

AuthenticationCleartextPassword (B)

Byte1('R')

Identifie le message comme une demande d'authentification.

Int32(8)

Longueur du contenu du message en octets, longueur incluse.

Int32(3)

Indique que l'authentification par mot de passe en clair en requise.

AuthenticationMD5Password (B)

Byte1('R')

Identifie le message comme une demande d'authentification.

Int32(12)

Longueur du contenu du message en octets, longueur inclue.

Int32(5)

Indique que le mot de passe chiffré avec MD5 est requis.

Byte4

Le sel à utiliser lors du chiffrement du mot de passe.

AuthenticationSCMCredential (B)

Byte1('R')

Identifie le message comme une demande d'authentification.

Int32(8)

Longueur du contenu du message en octets, longueur incluse.

Int32(6)

Indique qu'un message d'identification SCM est requis.

AuthenticationGSS (B)

Byte1('R')

Identifie le message comme une demande d'authentification.

Int32(8)

Longueur du contenu du message en octets, longueur inclue.

Int32(7)

Indique qu'une authentification GSSAPI est requise.

AuthenticationGSSContinue (B)

Byte1('R')

Identifie le message comme une demande d'authentification.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int32(8)

Indique que ce message contient des données GSSAPI ou SSPI.

Byte*n*

Données d'authentification GSSAPI ou SSPI.

AuthenticationSSPI (B)

Byte1('R')

Identifie le message comme une demande d'authentification.

Int32(8)

Longueur du contenu du message en octets, longueur incluse.

Int32(9)

Indique que l'authentification SSPI est requise.

AuthenticationSASL (B)

Byte1('R')

Identifie le message comme une demande d'authentification.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int32(10)

Indique que l'authentification SASL est requise.

Le corps du message est une liste de mécanismes d'authentification SASL, dans l'ordre de préférence du serveur. Un octet zéro est requis en fin après le nom du dernier mécanisme d'authentification. Pour chaque mécanisme, il y a ce qui suit :

String

Nom d'un mécanisme d'authentification SASL.

AuthenticationSASLContinue (B)

Byte1('R')

Identifie le message comme une demande d'authentification.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int32(11)

Indique que ce message contient un challenge SASL.

Byte*n*

Données SASL, spécifiques au mécanisme SASL utilisé.

AuthenticationSASLFinal (B)

Byte1('R')

Identifie le message comme une demande d'authentification.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int32(12)

Indique que l'authentification SASL s'est terminée.

Byte*n*

Données supplémentaires SASL, spécifique au mécanisme SASL utilisé.

BackendKeyData (B)

Byte1('K')

Identifie le message comme une donnée clé d'annulation. Le client doit conserver ces valeurs s'il souhaite être capable d'exécuter ultérieurement des messages CancelRequest.

Int32(12)

Longueur du contenu du message en octets, longueur incluse.

Int32

Identifiant du processus de ce backend.

Int32

Clé secrète de ce backend.

Bind (F)

Byte1('B')

Identifie le message comme une commande Bind.

Int32

Longueur du contenu du message en octets, longueur inclue.

String

Le nom du portail de destination (une chaîne vide sélectionne le portail non nommé).

String

Le nom de la requête préparée source (une chaîne vide sélectionne la requête préparée non nommée).

Int16

Le nombre de codes format des paramètres (dénoté *C* ci-dessous). Il peut valoir zéro pour indiquer qu'il n'y a pas de paramètres ou que les paramètres peuvent tous utiliser le format par défaut (text) ; ou un, auquel cas le code format spécifié est appliqué à tous les paramètres ; ou il peut être égal au nombre réel de paramètres.

Int16[*C*]

Les codes format des paramètres. Il n'y a que deux valeurs possibles : zéro pour texte, un pour binaire.

Int16

Le nombre de valeurs de paramètres (potentiellement zéro). Ce nombre doit correspondre au nombre de paramètres nécessaires pour la requête.

Ensuite, la paire suivante de champs apparaît pour chaque paramètre :

Int32

La longueur de la valeur du paramètre en octets (ce nombre ne s'inclut pas). Peut valoir zéro. En cas particulier, -1 indique une valeur NULL du paramètre.

Byte*n*

La valeur du paramètre, dans le format indiqué par le code format associé. *n* est la longueur ci-dessus.

Après le dernier paramètre, les champs suivants apparaissent :

Int16

Le nombre de codes formats des colonnes de résultat (dénoté par *R* ci-dessous). Ce nombre peut valoir zéro pour indiquer qu'il n'y a pas de colonnes de résultats ou que les colonnes de résultats devraient toutes utiliser le format par défaut (texte) ; ou un, auquel cas le code format spécifié est appliqué à toutes les colonnes du résultat (s'il y en a) ; ou il peut être égal au nombre réel de colonnes résultats de la requête.

Int16[*R*]

Les codes format des colonnes du résultat. Il n'existe que deux possibilités : zéro pour texte, un pour binaire.

BindComplete (B)

Byte1('2')

Identifie le message comme un indicateur de fin de Bind.

Int32(4)

Longueur du contenu du message en octets, longueur inclue.

CancelRequest (F)

Int32(16)

Longueur du contenu du message en octets, longueur inclue.

Int32(80877102)

Le code de demande d'annulation. La valeur est choisie pour contenir `1234` sur les 16 bits les plus significatifs, et `5678` sur les 16 bits les moins significatifs. (Pour éviter la confusion, ce code ne doit pas être le même que tout numéro de version de protocole.)

Int32

L'identifiant de processus du backend cible.

Int32

La clé secrète du backend cible.

Close (F)

Byte1('C')

Identifie le message comme une commande Close.

Int32

Longueur du contenu du message en octets, longueur inclue.

Byte1

'S' pour fermer une requête préparée ; ou 'P' pour fermer un portail.

String

Le nom de la requête préparée ou du portail à fermer (une chaîne vide sélectionne la requête préparée ou le portail non nommé).

CloseComplete (B)

Byte1('3')

Identifie le message comme un indicateur de fin de Close.

Int32(4)

Longueur du contenu du message en octets, longueur inclue.

CommandComplete (B)

Byte1('C')

Identifie le message comme une réponse de fin de commande.

Int32

Longueur du contenu du message en octets, longueur inclue.

String

La balise de la commande. C'est habituellement un mot simple qui identifie la commande SQL terminée.

Pour une commande `INSERT`, la balise est `INSERT` *`oid lignes`*, où *`lignes`* correspond au nombre de lignes insérées. *`oid`* était l'identifiant d'objet de la ligne insérée si *`lignes`* valait 1 et que la table cible avait des OID, mais les colonnes systèmes OID ne sont plus supportées ; de ce fait, *`oid`* est toujours à 0.

Pour une commande `DELETE`, la balise est `DELETE` *`lignes`* où *`lignes`* correspond au nombre de lignes supprimées.

Pour une commande `UPDATE`, la balise est `UPDATE` *`lignes`* où *`lignes`* correspond au nombre de lignes mises à jour.

Pour une commande `MERGE`, la balise est `MERGE` *`lignes`* où *`lignes`* correspond au nombre de lignes insérées, mises à jour ou supprimées.

Pour une commande `SELECT` ou `CREATE TABLE AS`, la balise est `SELECT` *`lignes`* où *`lignes`* correspond au nombre de lignes récupérées.

Pour une commande `MOVE`, la balise est `MOVE` *`lignes`* où *`lignes`* correspond au nombre de lignes de déplacement du curseur.

Pour une commande `FETCH`, la balise est `FETCH` *`lignes`* où *`lignes`* est le nombre de lignes qui ont été récupérées du curseur.

Pour une commande `COPY`, la balise est `COPY` *`lignes`* où *`lignes`* est le nombre de lignes copiées. (Note : le nombre de lignes apparaît seulement dans PostgreSQL 8.2 et les versions ultérieures.)

CopyData (F & B)

Byte1('d')

Identifie le message comme des données `COPY`.

Int32

Longueur du contenu du message en octets, longueur inclue.

Byte*n*

Données qui forment une partie d'un flux de données `COPY`. Les messages envoyés à partir du backend correspondront toujours à des lignes de données simples, mais les messages envoyés par les clients pourraient diviser le flux de données de façon arbitraire.

CopyDone (F & B)

Byte1('c')

Identifie le message comme un indicateur de fin de `COPY`.

Int32(4)

Longueur du contenu du message en octets, longueur inclue.

CopyFail (F)

Byte1('f')

Identifie le message comme un indicateur d'échec de `COPY`.

Int32

Longueur du contenu du message en octets, longueur inclue.

String

Un message d'erreur à rapporter comme cause de l'échec.

CopyInResponse (B)

Byte1('G')

Identifie le message comme une réponse Start Copy In. Le client doit maintenant envoyer des données `copy-in` (s'il n'est pas préparé à le faire, envoie un message CopyFail).

Int32

Longueur du contenu du message en octets, longueur inclue.

Int8

0 indique que le format global de `COPY` est du texte (lignes séparées par des retours à la ligne, colonnes séparées par des caractères séparateurs, etc.). 1 indique que le format global de copie est binaire (similaire au format DataRow). Voir COPY pour plus d'informations.

Int16

Le nombre de colonnes dans les données à copier (dénoté *N* ci-dessous).

Int16[*N*]

Les codes format à utiliser pour chaque colonne. Il n'y a que deux possibilités : zéro pour texte, un pour binaire. Tous doivent être à zéro pour que le format de copie global soit du texte.

CopyOutResponse (B)

Byte1('H')

Identifie le message comme une réponse Start Copy Out. Ce message sera suivi par les données `copy-out`.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int8

0 indique que le format global de `COPY` est du texte (lignes séparées par des retours à la ligne, colonnes séparées par des caractères séparateurs, etc.). 1 indique que le format global de copie est binaire (similaire au format DataRow). Voir COPY pour plus d'informations.

Int16

Le nombre de colonnes des données à copier (dénoté *N* ci-dessous).

Int16[*N*]

Les codes format à utiliser pour chaque colonne. Il n'y a que deux possibilités : zéro pour texte, un pour binaire. Ils doivent être tous à zéro pour que le format global de copie soit du texte.

CopyBothResponse (B)

Byte1('W')

Identifie le message comme une réponse Start Copy Both. Ce message est utilisé uniquement pour la réplication en flux.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int8

0 indique que le format global de `COPY` est du texte (lignes séparées par des retours à la ligne, colonnes séparées par des caractères séparateurs, etc.). 1 indique que le format global de copie est binaire (similaire au format DataRow). Voir COPY pour plus d'informations.

Int16

Le nombre de colonnes dans les données à copier (dénoté *N* ci-dessous).

Int16[*N*]

Les codes format à utiliser pour chaque colonne. Il n'y a que deux possibilités : zéro pour texte, un pour binaire. Ils doivent être tous à zéro pour que le format global de copie soit du texte.

DataRow (B)

Byte1('D')

Identifie le message comme une ligne de données.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int16

Le nombre de valeurs de colonnes (potentiellement zéro).

Ensuite, la paire de champs suivants apparaît pour chaque colonne :

Int32

La longueur de la valeur de la colonne, en octets (ce décompte ne s'inclue pas lui-même). Peut valoir zéro. Cas particulier, -1 indique une valeur de colonne NULL.

Byte*n*

La valeur de la colonne, dans le format indiqué par le code format associé. *n* est la longueur ci-dessus.

Describe (F)

Byte1('D')

Identifie le message comme une commande Describe.

Int32

Longueur du contenu du message en octets, longueur inclue.

Byte1

'S' pour décrire une requête préparée ; ou 'P' pour décrire un portail.

String

Le nom de la requête préparée ou du portail à décrire (une chaîne vide sélectionne la requête préparée ou le portail non nommé).

EmptyQueryResponse (B)

Byte1('I')

Identifie le message comme une réponse à une chaîne vide de requête. (Cela se substitue à CommandComplete.)

Int32(4)

Longueur du contenu du message en octets, longueur inclue.

ErrorResponse (B)

Byte1('E')

Identifie le message comme une erreur.

Int32

Longueur du contenu du message en octets, longueur inclue.

Le corps du message consiste en un ou plusieurs champs identifiés, suivi par un octet zéro comme terminaison. Les champs peuvent apparaître dans n'importe quel ordre. Pour chaque champ, il existe ce qui suit :

Byte1

Un code identifiant le type de champ ; si zéro, c'est le message de terminaison, et aucune chaîne ne suit. Les types de champs actuellement définis sont listés dans Section 55.8. Comme plus de types de champs pourraient être ajoutés dans le futur, les clients devraient ignorer silencieusement les champs dont le type n'est pas reconnu.

String

La valeur du champ.

Execute (F)

Byte1('E')

Identifie le message comme une commande Execute.

Int32

Longueur du contenu du message en octets, longueur inclue.

String

Le nom du portail à exécuter (une chaîne vide sélectionne le portail non nommé).

Int32

Nombre maximum de lignes à renvoyer si le portail contient une requête qui renvoie des lignes (ignoré sinon). Zéro indique « pas de limite ».

Flush (F)

Byte1('H')

Identifie le message comme une commande Flush.

Int32(4)

Longueur du contenu du message en octets, longueur inclue.

FunctionCall (F)

Byte1('F')

Identifie le message comme un appel de fonction.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int32

Indique l'identifiant d'objet de la fonction à appeler.

Int16

Le nombre de codes format des arguments (dénoté *C* ci-dessous). Ce nombre peut valoir zéro pour indiquer qu'il n'y a pas d'arguments ou que tous les arguments utilisent le format par défaut (texte) ; ou un, auquel cas le code format spécifié est appliqué à tous les arguments ; ou être égal au nombre réel d'arguments.

Int16[*C*]

Les codes formats des arguments. Il n'existe que deux possibilités : zéro pour texte, un pour binaire.

Int16

Indique le nombre d'arguments à fournir à la fonction.

Ensuite, la paire suivant de champs apparaît pour chaque argument :

Int32

La longueur de la valeur de l'argument, en octets (ce décompte ne s'inclue pas lui-même). Peut valoir zéro. Cas particulier, -1 indique une valeur de colonne NULL.

Byte*n*

La valeur de l'argument, dans le format indiqué par le code format associé. *n* est la longueur ci-dessus.

Après le dernier argument, le champ suivant apparaît :

Int16

Le code format pour le résultat de la fonction. Il n'existe que deux possibilités : 0 pour texte, 1 pour binaire.

FunctionCallResponse (B)

Byte1('V')

Identifie le message comme un résultat d'appel de fonction.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int32

La longueur du résultat de la fonction, en octets (ce décompte ne s'inclue pas lui-même). Peut valoir zéro. Cas particulier, -1 indique une valeur de colonne NULL.

Byte*n*

La valeur du résultat de la fonction, dans le format indiqué par le code format associé. *n* est la longueur ci-dessus.

GSSENCRequest (F)

Int32(8)

Longueur du contenu du message en octets, longueur incluse.

Int32(80877104)

Le code de demande de chiffrement GSSAPI. La valeur est choisie pour contenir `1234` sur les 16 bits les plus significatifs, et `5680` sur les 16 bits les moins significatifs. (Pour éviter de la confusion, ce code ne doit pas être le même que le numéro de version du protocole.)

GSSResponse (F)

Byte1('p')

Identifie le message comme une réponse GSSAPI ou SSPI. Notez que ceci est aussi utilisé pour les messages de réponse SASL et mot de passe. Le type exact du message peut être déduit du contexte.

Int32

Longueur du contenu du message en octets, longueur incluse.

Byte*n*

Données spécifiques du message GSSAPI/SSPI.

NegotiateProtocolVersion (B)

Byte1('v')

Identifie le message comme un message de négociation de version du protocole.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int32

Version mineure la plus récente supportée par le serveur pour la version majeure réclamée par le client.

Int32

Nombre d'options du protocole non reconnues par le serveur.

Puis, pour l'option de protocole non reconnue par le serveur, il y a ce qui suit :

String

Le nom de l'option.

NoData (B)

Byte1('n')

Identifie le message comme un indicateur d'absence de données.

Int32(4)

Longueur du contenu du message en octets, longueur inclue.

NoticeResponse (B)

Byte1('N')

Identifie le message comme une information.

Int32

Longueur du contenu du message en octets, longueur inclue.

Le corps du message consiste en un ou plusieurs champs identifiés, suivi par un octet zéro comme terminaison. Les champs peuvent apparaître dans n'importe quel ordre. Pour chaque champ, il existe ce qui suit :

Byte1

Un code identifiant le type de champs ; si zéro, ceci est le message de terminaison et aucune chaîne de caractères ne suit. Les types de champs actuellement définis sont listés dans Section 55.8. Comme plus de types de champs pourraient être ajoutés dans le futur, les clients devraient ignorer silencieusement les champs dont le type leur est inconnu.

String

La valeur du champ.

NotificationResponse (B)

Byte1('A')

Identifie le message comme une réponse à une notification.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int32

L'identifiant du processus backend notifiant.

String

Le nom du canal où la notification a été levée.

String

La « charge » passée par le processus notifiant.

ParameterDescription (B)

Byte1('t')

Identifie le message comme une description de paramètre.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int16

Le nombre de paramètres utilisés par la requête (peut valoir zéro).

Ensuite, pour chaque paramètre, il y a ce qui suit :

Int32

Indique l'identifiant d'objet du type de données du paramètre.

ParameterStatus (B)

Byte1('S')

Identifie le message comme un rapport de statut d'un paramètre.

Int32

Longueur du contenu du message en octets, longueur inclue.

String

Le nom du paramètre.

String

La valeur actuelle de ce paramètre.

Parse (F)

Byte1('P')

Identifie le message comme une commande Parse.

Int32

Longueur du contenu du message en octets, longueur inclue.

String

Le nom de la requête préparée de destination (une chaîne vide sélectionne la requête préparée non nommée).

String

La chaîne de requête à analyser.

Int16

Le nombre de types de données des paramètres (peut valoir zéro). Notez que cela n'est pas une indication du nombre de paramètres qui pourraient apparaître dans la requête de chaîne, seulement le nombre de types que le client souhaite pré-déclarer.

Ensuite, pour chaque paramètre, il y a ce qui suit :

Int32

Indique l'identifiant d'objet du type de données du paramètre. Placer un zéro ici est équivalent à laisser le type sans spécification.

ParseComplete (B)

Byte1('1')

Identifie le message comme un indicateur de fin de Parse.

Int32(4)

Longueur du contenu du message en octets, longueur inclue.

PasswordMessage (F)

Byte1('p')

Identifie le message comme une réponse de mot de passe. Notez que c'est aussi utilisé pour les messages de réponse pour GSSAPI, SSPI et SASL. Le type exact de message peut être déduit du contexte.

Int32

Longueur du contenu du message en octets, longueur inclue.

String

Le mot de passe (chiffré si demandé).

PortalSuspended (B)

Byte1('s')

Identifie le message comme un indicateur de portal-suspended. Notez que ceci survient uniquement si la limite de lignes a été atteint pour un message Execute.

Int32(4)

Longueur du contenu du message en octets, longueur inclue.

Query (F)

Byte1('Q')

Identifie le message comme une simple requête.

Int32

Longueur du contenu du message en octets, longueur inclue.

String

La requête elle-même.

ReadyForQuery (B)

Byte1('Z')

Identifie le type de message. ReadyForQuery est envoyé quand le backend est prêt pour un nouveau cycle de requête.

Int32(5)

Longueur du contenu du message en octets, longueur inclue.

Byte1

Indicateur du statut de transaction du backend. Les valeurs possibles sont '`I`' si inactif (et pas dans un bloc de transaction) ; '`T`' si dans un bloc de transaction ; ou '`E`' si dans un bloc de transaction en échec (les requêtes seront rejetées jusqu'à la fin du bloc).

RowDescription (B)

Byte1('T')

Identifie le message comme une description de ligne.

Int32

Longueur du contenu du message en octets, longueur inclue.

Int16

Indique le nombre de champs dans une ligne (peut valoir zéro).

Ensuite, pour chaque champ, il y a ce qui suit :

String

Le nom du champ.

Int32

Si le champ peut être identifié comme une colonne d'une table précise, l'identifiant d'objet de la table ; sinon zéro.

Int16

Si le champ peut être identifié comme une colonne d'une table précise, le numéro de la colonne ; sinon zéro.

Int32

L'identifiant d'objet du type de données du champ.

Int16

La taille du type de données (voir `pg_type.typlen`). Notez que les valeurs négatives désignent des types à taille variable.

Int32

Le modifieur du type (voir `pg_attribute.atttypmod`). La signification du modifieur dépend du type.

Int16

Le code format en cours d'utilisation pour le champ. Actuellement, ce sera soit zéro (texte) soit un (binaire). Dans un RowDescription renvoyé à partir de la variante requête de Describe, le code format n'est pas encore connu et sera toujours zéro.

SASLInitialResponse (F)

Byte1('p')

Identifie le message comme une réponse SASL initiale. Notez que ce message est aussi utilisé pour les messages de réponse de GSSAPI, SSPI et mot de passe. Le type exact de message est déduit du contexte.

Int32

Longueur du contenu du message en octets, longueur inclue.

String

Nom du mécanisme d'authentification SASL que le client a sélectionné.

Int32

Longueur du "Initial Client Response" spécifique au mécanisme SASL, ou -1 en cas d'absence de `Initial Response`.

Byte*n*

"Initial Response" spécifique au mécanisme SASL.

SASLResponse (F)

Byte1('p')

Identifie le message comme une réponse SASL. Notez que ce message est aussi utilisé par les messages réponses de GSSAPI, SSPI et mot de passe. Le type exact de message peut être déduit du contexte.

Int32

Longueur du contenu du message en octets, longueur inclue.

Byte*n*

Données du message spécifique au mécanisme SASL.

SSLRequest (F)

Int32(8)

Longueur du contenu du message en octets, longueur incluse.

Int32(80877103)

Le code de demande de chiffrement SSL. La valeur est choisie pour contenir `1234` sur les 16 bits les plus significatifs, et `5679` sur les 16 bits les moins significatifs. (Pour éviter de la confusion, ce code ne doit pas être le même que le numéro de version du protocole.)

StartupMessage (F)

Int32

Longueur du contenu du message en octets, longueur incluse.

Int32(196608)

Le numéro de version du protocole. Les 16 bits les plus significatifs sont le numéro de version majeure (3 pour le protocole décrit ici). Les 16 bits les moins significatifs sont le numéro de version mineure (0 pour le protocole décrit ici).

Le numéro de version du protocole est suivi par une ou plusieurs paires de nom/valeur de paramètre. Un octet zéro est requis comme terminaison après la dernière paire nom/valeur. Les paramètres peuvent apparaître dans n'importe quel ordre. `user` est requis, les autres sont optionnels. Chaque paramètre est spécifié ainsi :

String

Le nom du paramètre. Actuellement, les noms reconnus sont :

`user`

Le nom de l'utilisateur pour la connexion à la base. Requis, pas de valeur par défaut.

`database`

La base de données où se connecter. Par défaut, le nom de l'utilisateur.

`options`

Arguments de la ligne de commande pour le backend. (C'est abandonné au profit de la configuration de paramètres individuels.) Les espaces dans cette chaîne permettent de séparer les arguments, à moins qu'ils ne soient échappés avec un caractère antislash (\) ; écrire \\ pour représenter un antislash litéral.

`replication`

Utilisé pour se connecter mode de réplication en flux, où un petit ensemble de commandes de réplication peuvent être exécutées au lieu de requêtes SQL. Les valeurs peuvent être `true`, `false` ou `database`, et la valeur par défaut est `false`. Voir Section 55.4 pour plus de détails.

En plus de tout cela, d'autres paramètres pourraient être listés. Les noms de paramètre commençant avec `_pq_.` sont réservés pour une utilisation en tant qu'extensions du protocole, alors que les autres sont traités comme des paramètres d'exécution à configurer au démarrage du backend. De telles configurations seront appliquées au démarrage du serveur (après analyse des arguments en ligne de commande s'il y en a) et agiront comme valeurs par défaut de la session.

String

La valeur du paramètre.

Sync (F)

Byte1('S')

Identifie le message comme une commande Sync.

Int32(4)

Longueur du contenu du message en octets, longueur incluse.

Terminate (F)

Byte1('X')

Identifie le message comme une fin de session.

Int32(4)

Longueur du contenu du message en octets, longueur incluse.

55.8. Champs des messages d'erreur et de notification

Cette section décrit les champs qui peuvent apparaître dans des messages ErrorResponse et NoticeResponse. Chaque type de champ a un jeton d'identification sur un seul octet. Notez que tout type de champ devrait apparaître au moins une fois par message.

`S`

Sévérité : le contenu du champ est `ERROR`, `FATAL`, ou `PANIC` (dans un message d'erreur), ou `WARNING`, `NOTICE`, `DEBUG`, `INFO` ou `LOG` (dans un message de notification), ou une traduction localisée d'un de ces messages. Toujours présent.

`V`

Sévérité : le contenu du champ est `ERROR`, `FATAL`, ou `PANIC` (dans un message d'erreur), ou `WARNING`, `NOTICE`, `DEBUG`, `INFO`, ou `LOG` (dans un message de notification). Ceci est identique au champ `S` sauf que son contenu n'est jamais localisé. Il est présent uniquement dans les messages générés par un serveur PostgreSQL en version 9.6 ou ultérieure.

`C`

Code : le code SQLSTATE pour l'erreur (voir Annexe A). Non localisable. Toujours présent.

`M`

Message : le message d'erreur principal, lisible par un humain. Il doit être précis mais court (typiquement une ligne). Toujours présent.

`D`

Détail : un message d'erreur secondaire et optionnel apportant plus d'informations sur le problème. Pourrait utiliser plusieurs lignes.

`H`

Astuce : une suggestion optionnelle pour corriger le problème. Ceci doit différer du Détail parce qu'il offre un conseil (potentiellement inapproprié) plutôt que des faits certains. Pourrait utiliser plusieurs lignes.

`P`

Position : la valeur du champ est un entier décimal ASCII, indiquant la position du curseur sur l'erreur comme un index sur la chaîne requête originale. Le premier caractère est à l'index 1, et les positions sont mesurées en caractères, et non pas en octets.

`p`

Position interne : ceci est défini de la même façon que le champ `P`, mais est utilisé quand la position du curseur fait référence à une commande générée en interne plutôt qu'une commande exécutée par le client. Le champ `q` apparaîtra toujours quand ce champ apparaît.

`q`

Requête interne : le texte d'une commande générée en interne. Cela pourrait être une requête SQL exécutée par une fonction PL/pgSQL par exemple.

`W`

Où : une indication du contexte dans lequel l'erreur s'est produite. Actuellement, cela inclut la pile d'appels des fonctions en langage procédural et les requêtes générées en interne. La trace est composée d'une entrée par ligne, la plus récente en premier.

`s`

Nom du schéma : si l'erreur était associée à un objet particulier de la base, le nom du schéma contenant cet objet, s'il existe.

`t`

Nom de la table : si l'erreur était associée à une table particulière, le nom de la table. (Référez-vous au champ du nom de schéma pour le nom du schéma de la table.)

`c`

Nom de la colonne : si l'erreur était associée à une colonne particulière de la table, le nom de la colonne. (Référez-vous aux champs de nom de schéma et de table pour identifier la table.)

`d`

Nom du type de données : si l'erreur était associée à un type de données particulier, le nom du type de données. (Référez-vous au champ du nom de schéma pour le nom du schéma du type de données.)

`n`

Nom de contrainte : si l'erreur était associée à une contrainte particulière, le nom de la contrainte. Référez-vous aux champs listés ci-dessus pour la table ou le domaine associé. (Dans ce but, les index sont traités comme des contraintes, même si elles n'ont pas été créées avec une syntaxe de contrainte.)

`F`

Fichier : le nom du fichier source où l'erreur a été rapportée.

`L`

Ligne : le numéro de ligne dans le fichier source où l'erreur a été rapportée.

`R`

Routine : le nom de la routine ayant rapporté l'erreur.

> **Note**
>
> Les champs du nom du schéma, de la table, de la colonne, du type de données et de la contrainte sont fournis seulement pour un nombre limité de types d'erreur ; voir Annexe A. Les clients ne doivent pas supposer que la présence d'un de ces champs garantie la présence d'un autre champ. Les sources des erreurs observent les relations notées ci-dessus, mais les fonctions définies par les utilisateurs pourraient utiliser ces champs d'une autre façon. Dans la même veine, les clients ne devraient pas supposer que ces champs parlent d'objets actuels dans la base actuelle.

Le client est responsable du formatage des informations affichées ; en particulier, il doit diviser les lignes longues si nécessaires. Les caractères de retour chariot apparaissant dans les champs du message d'erreur doivent être traités comme des changements de paragraphes, pas comme des changements de lignes.

55.9. Formats des messages de la réplication logique

Cette section décrit le format détaillé de chaque message de réplication logique. Ces messages sont renvoyés soit par l'interface SQL du slot de réplication soit par un walsender. Dans ce dernier cas, ils sont encapsulés à l'intérieur de messages WAL du protocole de réplication, comme décrit dans Section 55.4, et obéissent généralement au même flux de message que la réplication physique.

Begin

Byte1('B')

Identifie le message comme un message begin.

Int64 (XLogRecPtr)

Le LSN final de la transaction.

Int64 (TimestampTz)

Horodatage de la validation de la transaction. La valeur est un nombre de microsecondes depuis l'epoch PostgreSQL (1er janvier 2000).

Int32 (TransactionId)

Xid de la transaction.

Message

Byte1('M')

Identifie le message comme un message de décodage logique.

Int32 (TransactionId)

Xid de la transaction (seulement présent pour les transactions en flux). Ce champ est disponible depuis la version 2 du protocole.

Int8

Drapeaux ; soit 0 pour aucun drapeau, ou 1 si le message de décodage logique est transactionnel.

Int64 (XLogRecPtr)

Le LSN du message de décodage logique.

String

Le préfixe du message de décodage logique.

Int32

Longueur du contenu.

Byte*n*

Le contenu du message de décodage logique.

Commit

Byte1('C')

Identifie le message comme un message de Commit.

Int8(0)

Drapeaux ; actuellement inutilisé.

Int64 (XLogRecPtr)

Le LSN du commit.

Int64 (XLogRecPtr)

Le LSN de fin de la transaction.

Int64 (TimestampTz)

Horodatage de la validation de la transaction. La valeur est en nombre de microsecondes depuis l'epoch PostgreSQL (1er janvier 2000).

Origin

Byte1('O')

Identifie le message comme un message Origin.

Int64 (XLogRecPtr)

Le LSN de la validation sur le serveur d'origine.

String

Nom de l'origine.

Notez qu'il peut y avoir plusieurs messages Origin dans une même transaction.

Relation

Byte1('R')

Identifie le message comme un message Relation.

Int32 (TransactionId)

XID de la transaction (seulement présent pour les transactions en flux). Ce champ est disponible à partir de la version 2 du protocole.

Int32 (Oid)

OID de la relation.

String

Schéma (chaîne vide pour `pg_catalog`).

String

Nom de la relation.

Int8

Configuration de l'identité de réplication pour la relation (identique à `relreplident` dans `pg_class`).

Int16

Nombre de colonnes.

Ensuite, la partie suivante du message apparaît pour chaque colonne inclus dans la publication, à l'exception des colonnes générées :

Int8

Drapeaux pour la colonne. Actuellement, peut valoir soit 0 pour aucun drapeau ou 1 pour marquer la colonne comme faisant partie de la clé.

String

Nom de la colonne.

Int32 (Oid)

OID du type de données de la colonne.

Int32

Modifieur de type de la colonne (`atttypmod`).

Type

Byte1('Y')

Identifie le message comme un message Type.

Int32 (TransactionId)

XID de la transaction (seulement présent pour les transactions en flux). Ce champ est disponible depuis la version 2 du protocole.

Int32 (Oid)

OID du type de données.

String

Schéma (chaîne vide pour `pg_catalog`).

String

Nom du type de données.

Insert

Byte1('I')

Identifie le message comme un message Insert.

Int32 (TransactionId)

XID de la transaction (seulement présent pour les transactions en flux). Ce champ est disponible depuis la version 2 du protocole.

Int32 (Oid)

OID de la relation correspondant à l'identifiant du message Relation message.

Byte1('N')

Identifie le message TupleData suivant comme une nouvelle ligne.

TupleData

Message TupleData représentant le contenu de la nouvelle ligne.

Update

Byte1('U')

Identifie le message comme un message Update.

Int32 (TransactionId)

XID de la transaction (seulement présent pour les transactions en flux). Ce champ est disponible depuis la version 2 du protocole.

Int32 (Oid)

OID de la relation correspondant à l'identifiant du message Relation message.

Byte1('K')

Identifie les sous-messages TupleData suivants comme une clé. Ce champ est optionnel et est présent uniquement si la mise à jour a modifié des données dans une des colonnes qui font partie de l'index REPLICA IDENTITY.

Byte1('O')

Identifie les sous-messages TupleData suivants comme une ancienne ligne. Ce champ est optionnel et est présent uniquement si la table ciblée par la mise à jour a REPLICA IDENTITY configuré à FULL.

TupleData

Message TupleData représentant le contenu de l'ancienne ligne ou la clé primaire. Seulement présent si la partie 'O' ou 'K' précédente est présente.

Byte1('N')

Identifie le message TupleData suivant comme la nouvelle ligne.

TupleData

Message TupleData représentant le contenu de la nouvelle ligne.

Le message Update peut contenir soit une partie d'un message 'K' soit une partie d'un message 'O', mais jamais les deux.

Delete

Byte1('D')

Identifie le message comme un message Delete.

Int32 (TransactionId)

XID de la transaction (seulement présent pour les transactions en flux). Ce champ est disponible depuis la version 2 du protocole.

Int32 (Oid)

OID de la relation correspondant ç l'identifiant du message Relation message.

Byte1('K')

Identifie le sous-message TupleData suivant comme une clé. Ce champ est présent si la table ciblée par la suppression utilise un index comme REPLICA IDENTITY.

Byte1('O')

Identifie le message TupleData suivant comme une ancienne ligne. Ce champ est présent si la table ciblée par la suppression a le REPLICA IDENTITY configuré à FULL.

TupleData

Message TupleData représentant le contenu de l'ancienne ligne ou clé primaire, suivant le champ précédent.

Le message Delete pourrait contenir soit une partie d'un message 'K' soit une partie d'un message 'O', mais jamais les deux.

Truncate

Byte1('T')

Identifie le message comme un message Truncate.

Int32 (TransactionId)

XID de la transaction (seulement présent pour les transactions en flux). Ce champ est disponible depuis la version 2 du protocole.

Int32

Nombre de relations.

Int8

Options pour `TRUNCATE` : 1 pour `CASCADE`, 2 pour `RESTART IDENTITY`

Int32 (Oid)

OID de la relation correspondant à l'identifiant du message Relation. Ce champ est répété pour chaque relation.

Les messages suivants (Stream Start, Stream Stop, Stream Commit et Stream Abort) sont disponibles depuis la version 2 du protocole.

Stream Start

Byte1('S')

Identifie le message comme un message de début de flux.

Int32 (TransactionId)

Xid de la transaction.

Int8

Une valeur de 1 indique qu'il s'agit du premier segment pour cet XID, et une valeur de 0 pour tout autre segment de flux.

Stream Stop

Byte1('E')

Identifie le message comme un message d'arrêt de flux.

Stream Commit

Byte1('c')

Identifie le message comme un message de validation du flux.

Int32 (TransactionId)

XID de la transaction.

Int8(0)

Drapeaux ; actuellement inutilisé.

Int64 (XLogRecPtr)

Le LSN de la validation.

Int64 (XLogRecPtr)

Le LSN final de la transaction.

Int64 (TimestampTz)

Horodatage de la validation de la transaction. La valeur est un nombre de microsecondes depuis l'epoch PostgreSQL (1er janvier 2000).

Stream Abort

Byte1('A')

Identifie le message comme un message d'annulation de flux.

Int32 (TransactionId)

XID de la transaction.

Int32 (TransactionId)

XID de la sous-transaction (identique à l'XID de la transaction pour les transactions haut niveau).

Les messages suivants (Begin Prepare, Prepare, Commit Prepared, Rollback Prepared, Stream Prepare) sont disponibles depuis la version 3 du protocole.

Begin Prepare

Byte1('b')

Identifie le message comme le début d'une transaction préparée.

Int64 (XLogRecPtr)

Le LSN de la préparation de la transaction.

Int64 (XLogRecPtr)

Le LSN final de la transaction préparée.

Int64 (TimestampTz)

Horodatage de la phase de préparation de la transaction. La valeur est en nombre de microsecondes depuis l'epoch PostgreSQL (1er janvier 2000).

Int32 (TransactionId)

XID de la transaction.

String

Le GID défini par l'utilisateur pour la transaction préparée.

Prepare

Byte1('P')

Identifie le message comme un message de transaction préparée.

Int8(0)

Drapeaux ; actuellement inutilisé.

Int64 (XLogRecPtr)

Le LSN de la préparation.

Int64 (XLogRecPtr)

Le LSN final de la transaction préparée.

Int64 (TimestampTz)

Horodatage de la phase de préparation de la transaction. La valeur est en nombre de microsecondes depuis l'epoch PostgreSQL (1er janvier 2000).

Int32 (TransactionId)

XID de la transaction.

String

Le GID défini par l'utilisateur pour la transaction préparée.

Commit Prepared

Byte1('K')

Identifie le message comme le message de validation d'une transaction préparée.

Int8(0)

Drapeaux ; actuellement inutilisé.

Int64 (XLogRecPtr)

Le LSN de la validation de la transaction préparée.

Int64 (XLogRecPtr)

Le LSN final de la validation de la transaction préparée.

Int64 (TimestampTz)

Horodatage de la validation de la transaction. La valeur est en nombre de microsecondes depuis l'epoch PostgreSQL (1er janvier 2000).

Int32 (TransactionId)

XID de la transaction.

String

Le GID défini par l'utilisateur pour la transaction préparée.

Rollback Prepared

Byte1('r')

Identifie le message comme le message d'annulation d'une transaction préparée.

Int8(0)

Drapeaux ; actuellement inutilisé.

Int64 (XLogRecPtr)

Le LSN final de la transaction préparée.

Int64 (XLogRecPtr)

Le LSN final de la transaction préparée annulée.

Int64 (TimestampTz)

Horodatage de la phase de préparation de la transaction. La valeur est en nombre de microsecondes depuis l'epoch PostgreSQL (1er janvier 2000).

Int64 (TimestampTz)

Horodatage de la phase d'annulation de la transaction. La valeur est en nombre de microsecondes depuis l'epoch PostgreSQL (1er janvier 2000).

Int32 (TransactionId)

XID de la transaction.

String

Le GID défini par l'utilisateur pour la transaction préparée.

Stream Prepare

Byte1('p')

Identifie le message comme un message de préparation en flux 2PC.

Int8(0)

Drapeaux ; actuellement inutilisé.

Int64 (XLogRecPtr)

Le LSN de la phase de préparation.

Int64 (XLogRecPtr)

Le LSN final de la transaction préparée.

Int64 (TimestampTz)

Horodatage de la phase de préparation de la transaction. La valeur est en nombre de microsecondes depuis l'epoch PostgreSQL (1er janvier 2000).

Int32 (TransactionId)

XID de la transaction.

String

Le GID défini par l'utilisateur pour la transaction préparée.

Les parties de message suivants sont partagées par les messages ci-dessus.

TupleData

Int16

Nombre de colonnes.

Ensuite, un des sous-messages suivants apparaît pour chaque colonne, à l'exception des colonnes générées) :

Byte1('n')

Identifie la donnée comme une valeur NULL.

ou

Byte1('u')

Identifie la valeur TOAST non modifiée (la valeur réelle n'est pas envoyée).

ou

Byte1('t')

Identifie la donnée comme une valeur formatée en texte.

ou

Byte1('b')

Identifie la donnée comme une valeur formatée en binaire.

Int32

Longueur de la valeur de la colonne.

Byte*n*

La valeur de la colonne, en format soit binaire soit texte. (Comme spécifié dans l'octet de format précédent.) *n* est la longueur ci-dessus.

55.10. Résumé des changements depuis le protocole 2.0

Cette section fournit une liste rapide des changements, au bénéfice des développeurs essayant de mettre à jour leur bibliothèque client existante avec le protocole 3.0.

Le paquet de démarrage initial utilise un format flexible de liste de chaînes de caractères, au lieu d'un format fixé. Notez que les valeurs par défaut d'une session pour les paramètres d'exécution peuvent maintenant être spécifiées directement dans le paquet de démarrage. (En fait, vous pouvez le faire avant en utilisant le champ `options` mais, étant donné la largeur limité de ce champ et le manque de moyen pour mettre entre guillemets les espaces blancs dans les valeurs, ce n'était pas une technique très sûre.)

Tous les messages ont maintenant un compteur de longueur suivant immédiatement l'octet du type de message (sauf pour les paquets de démarrage qui n'ont pas d'octet de type). De plus, notez que le message PasswordMessage a maintenant un octet de type

Les messages ErrorResponse et NoticeResponse ('`E`' et '`N`') contiennent maintenant plusieurs champs, à partir desquels le code du client peut assembler un message d'erreur avec le niveau de verbosité désiré. Notez que les champs individuels ne finiront habituellement pas avec un retour chariot, alors que la chaîne seule envoyée dans le protocole précédent le faisait toujours.

Le message ReadyForQuery ('`Z`') inclut un indicateur de statut de transaction.

La distinction entre les types de message BinaryRow et DataRow n'existe plus ; le type de message DataRow sert à renvoyer des données dans tous les formats. Notez que la disposition de DataRow a changé pour faciliter l'analyse. De plus, la représentation des valeurs binaires a changé : elle n'est plus directement liée à la représentation interne du serveur.

Il existe un nouveau sous-protocole de « requête étendue », qui ajoute les types de message client Parse, Bind, Execute, Describe, Close, Flush et Sync, ainsi que les types de message backend ParseComplete, BindComplete, PortalSuspended, ParameterDescription, NoData et CloseComplete. Les clients existants n'ont pas besoin de s'occuper de ce sous-protocole, mais l'utiliser pourrait permettre des améliorations en performance et en fonctionnalité.

Les données de `COPY` sont maintenant encapsulées dans des messages CopyData et CopyDone. Il existe une façon bien définie de reprendre après des erreurs survenues pendant un `COPY`. Le « `\.` » de la dernière ligne n'est plus nécessaire, et n'est plus envoyé lors d'un `COPY OUT`. (Il est toujours reconnu comme terminaison lors d'un `COPY IN`, mais son utilisation est abandonnée et finira par être supprimée.) Le `COPY` binaire est supporté. Les messages CopyInResponse et CopyOutResponse incluent des champs indiquant le nombre de colonnes et le format de chaque colonne.

La disposition des messages FunctionCall et FunctionCallResponse a changé. FunctionCall accepte maintenant de passer des arguments NULL aux fonctions. Il peut aussi gérer de passer des paramètres et de récupérer des résultats au format texte ou binaire. Il n'y a plus de raison de considérer FunctionCall comme une faille potentielle de sécurité car il n'offre pas d'accès direct aux représentations internes des données du serveur.

Le backend envoie des messages ParameterStatus ('`S`') lors du démarrage de la connexion pour tous les paramètres qu'il considère intéressant pour la bibliothèque cliente. De plus, un message ParameterStatus est envoyé à chaque fois que la valeur active est modifiée pour chacun de ces paramètres.

Le message RowDescription ('`T`') apporte de nouveaux champs pour l'OID de la table et le numéro de la colonne pour chaque colonne de la ligne décrite. Il affiche aussi le code format pour chaque colonne.

Le message CursorResponse ('`P`') n'est plus généré par le backend.

Le message NotificationResponse ('`A`') a un champ supplémentaire de type chaîne de caractères, qui peut porter une « charge » passé depuis l'événement `NOTIFY`.

Le message EmptyQueryResponse ('`I`') incluait un paramètre de chaîne vide ; il a été supprimé.

Chapitre 56. Conventions de codage pour PostgreSQL

56.1. Formatage

Le formatage du code source utilise un espacement de quatre colonnes pour les tabulations, avec préservation de celles-ci (c'est-à-dire que les tabulations ne sont pas converties en espaces). Chaque niveau logique d'indentation est une tabulation supplémentaire.

Les règles de disposition (positionnement des parenthèses, etc.) suivent les conventions BSD. En particulier, les accolades pour les blocs de contrôle `if`, `while`, `switch`, etc. ont leur propre ligne.

Limiter la longueur des lignes pour que le code soit lisible avec une fenêtre de 80 colonnes. (Cela ne signifie pas que vous ne devez jamais dépasser 80 colonnes. Par exemple, diviser un long message d'erreur en plusieurs morceaux arbitraires pour respecter la consigne des 80 colonnes ne sera probablement pas un grand gain en lisibilité.)

Pour maintenir un style de codage cohérent, ne pas utiliser les commentaires style C++ (`//`). pgindent les remplacera par `/* ... */`.

Le style préféré pour les blocs multilignes de commentaires est :

```
/*
 * le commentaire commence ici
 * et continue ici
 */
```

Notez que les blocs de commentaire commençant en colonne 1 seront préservés par pgindent, mais qu'il déplacera (au niveau de la colonne) les blocs de commentaires indentés comme tout autre texte. Si vous voulez préserver les retours à la ligne dans un bloc indenté, ajoutez des tirets comme ceci :

```
    /*----------
     * le commentaire commence ici
     * et continue ici
     *----------
     */
```

Bien que les correctifs (patchs) soumis ne soient absolument pas tenus de suivre ces règles de formatage, il est recommandé de le faire. Le code est passé dans pgindent avant la sortie de la prochaine version, donc il n'y a pas de raison de l'écrire avec une autre convention de formatage. Une bonne règle pour les correctifs est de « faire en sorte que le nouveau code ressemble au code existant qui l'entoure ».

Le répertoire `src/tools/editors` contient des fichiers d'exemples de configuration qui peuvent être employés avec les éditeurs Emacs, xemacs ou vim pour valider que le format du code écrit respecte ces conventions.

Si vous voulez exécuter pgindent localement pour aider à rendre votre code plus proche du style du projet, voir le répertoire `src/tools/pgindent`.

Les outils de parcours de texte more et less peuvent être appelés de la manière suivante :

```
more -x4
less -x4
```

pour qu'ils affichent correctement les tabulations.

56.2. Reporter les erreurs dans le serveur

Les messages d'erreur, d'alertes et de traces produites dans le code du serveur doivent être créés avec `ereport` ou son ancien cousin `elog`. L'utilisation de cette fonction est suffisamment complexe pour nécessiter quelques explications.

Il y a deux éléments requis pour chaque message : un niveau de sévérité (allant de `DEBUG` à `PANIC`) et un message texte primaire. De plus, il y a des éléments optionnels, le plus commun d'entre eux est le code identifiant de l'erreur qui suit les conventions SQLSTATE des spécifications SQL. `ereport` en elle-même n'est qu'une macro shell qui existe principalement pour des convenances syntaxiques faisant ressembler la génération de messages à un seul appel d'une fonction dans un code source C. Le seul paramètre directement accepté par `ereport` est le niveau de sévérité. Le message texte primaire et les autres éléments de messages optionnels sont produits par appel de fonctions auxiliaires, comme `errmsg`, dans l'appel à `ereport`.

Un appel typique à `ereport` peut ressembler à :

```
ereport(ERROR,
        errcode(ERRCODE_DIVISION_BY_ZERO),
        errmsg("division by zero"));
```

Le niveau de sévérité de l'erreur est ainsi positionné à `ERROR` (une erreur banale). L'appel à `errcode` précise l'erreur SQLSTATE en utilisant une macro définie dans `src/include/utils/errcodes.h`. L'appel à `errmsg` fournit le message texte primaire.

Vous verrez aussi fréquemment cet ancien style, avec un ensemble supplémentaire de parenthèses entourant les appels de fonction auxiliaire :

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

Les parenthèses supplémentaires sont nécessaires pour les versions de PostgreSQL antérieures à la version 12, et sont depuis optionnelles.

Exemple plus complexe :

```
ereport(ERROR,
        errcode(ERRCODE_AMBIGUOUS_FUNCTION),
        errmsg("function %s is not unique",
               func_signature_string(funcname, nargs,
                                     NIL, actual_arg_types)),
        errhint("Unable to choose a best candidate function. "
                "You might need to add explicit typecasts."));
```

Cela illustre l'utilisation des codes de formatage pour intégrer des valeurs d'exécution dans un message texte. Un message « conseil », optionnel, est également fourni. Les appels de fonction auxiliaire peuvent être écrits dans n'importe quel ordre, bien que la convention soit de faire apparaître `errcode` et `errmsg` en premier.

Si le niveau de sévérité est `ERROR` ou plus, `ereport` annule l'exécution de la requête en cours et ne rend pas la main à l'appelant. Si le niveau de sévérité est moins qu'`ERROR`, `ereport` rend la main normalement.

Les routines auxiliaires disponibles pour `ereport` sont :

- `errcode(sqlerrcode)` précise le code SQLSTATE de l'identifiant erreur pour la condition. Si cette routine n'est pas appelée, l'identifiant l'erreur est, par défaut, `ERRCODE_INTERNAL_ERROR` quand le niveau de sévérité de l'erreur est `ERROR` ou plus haut, `ERRCODE_WARNING` quand le niveau d'erreur est `WARNING` et `ERRCODE_SUCCESSFUL_COMPLETION` pour `NOTICE` et inférieur. Bien que ces valeurs par défaut soient souvent commodes, il faut se demander si elles sont appropriées avant d'omettre l'appel à `errcode()`.

- `errmsg(const char *msg, ...)` indique le message texte primaire de l'erreur et les possibles valeurs d'exécutions à y insérer. Les insertions sont précisées par les codes de formatage dans le style `sprintf`. En plus des codes de formatage standard acceptés par `sprintf`, le code `%m` peut être utilisé pour insérer le message d'erreur retourné par `strerror` pour la valeur courante de `errno`. [1] `%m` ne nécessite aucune entrée correspondante dans la liste de paramètres pour `errmsg`. Notez que la chaîne de caractères du message sera passée à travers `gettext` pour une possible adaptation linguistique avant que les codes de formatage ne soient exécutés.

- `errmsg_internal(const char *msg, ...)` fait la même chose que `errmsg` à l'exception que la chaîne de caractères du message ne sera ni traduite ni incluse dans le dictionnaire de messages d'internationalisation. Cela devrait être utilisé pour les cas qui « ne peuvent pas arriver » et pour lesquels il n'est probablement pas intéressant de déployer un effort de traduction.

- `errmsg_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` est identique à `errmsg` mais avec le support pour plusieurs formes de pluriel du message. *`fmt_singular`* est le format singulier de l'anglais, *`fmt_plural`* est le format pluriel en anglais, *`n`* est la valeur entière qui détermine la forme utilisée. Les arguments restants sont formatés suivant la chaîne de format sélectionnée. Pour plus d'informations, voir Section 57.2.2.

- `errdetail(const char *msg, ...)` fournit un message « détail » optionnel ; cela est utilisé quand il y a des informations supplémentaires qu'il semble inadéquat de mettre dans le message primaire. La chaîne de caractères du message est traitée de la même manière que celle de `errmsg`.

- `errdetail_internal(const char *msg, ...)` est identique à `errdetail`, sauf que le message ne sera ni traduit ni inclut dans le dictionnaire des messages à traduire. Elle doit être utilisée pour les messages de niveau détail pour lequel un effort de traduction est inutile, par exemple parce qu'ils sont trop techniques pour que cela soit utile à la majorité des utilisateurs.

- `errdetail_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` est identique à `errdetail` mais avec le support de plusieurs formes de pluriel pour le message. Pour plus d'information, voir Section 57.2.2.

- `errdetail_log(const char *msg, ...)` est identique à `errdetail` sauf que cette chaîne ne va que dans les traces du serveur. Elle n'est jamais envoyée au client. Si `errdetail` (ou un de ses équivalents ci-dessus) et `errdetail_log` sont utilisées ensemble, alors une chaîne est envoyés au client et l'autre dans les traces du serveur. C'est utile pour les détails d'erreur qui concernent la sécurité ou qui sont trop techniques pour être inclus dans le rapport envoyé au client.

- `errdetail_log_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` est identique à `errdetail_log`, mais avec le support de plusieurs formes de pluriel pour le message. Pour plus d'informations, voir Section 57.2.2.

- `errhint(const char *msg, ...)` fournit un message « conseil » optionnel ; cela est utilisé pour offrir des suggestions sur la façon de régler un problème, par opposition aux détails effectifs

[1] C'est-à-dire la valeur qui était courante quand l'appel à `ereport` a été atteinte ; les changements d'`errno` dans les routines auxiliaires de rapports ne l'affecteront pas. Cela ne sera pas vrai si vous devez écrire explicitement `strerror(errno)` dans la liste de paramètres de `errmsg` ; en conséquence ne faites pas comme ça.

au sujet de ce qui a mal tourné. La chaîne de caractères du message est traitée de la même manière que celle de `errmsg`.

- `errhint_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` est identique à `errhint`, mais avec le support de plusieurs formes plurielles du message. Pour plus d'informations, voir Section 57.2.2.

- `errcontext(const char *msg, ...)` n'est normalement pas appelée directement depuis un site de message de `ereport` mais plutôt elle est utilisée dans les fonctions de rappels `error_context_stack` pour fournir des informations à propos du contexte dans lequel une erreur s'est produite, comme les endroits courants dans la fonction PL. La chaîne de caractères du message est traitée de la même manière que celle de `errmsg`. À l'inverse des autres fonctions auxiliaires, celle-ci peut être appelée plus d'une fois dans un appel de `ereport` ; les chaînes successives ainsi fournies sont concaténées et séparées pas des caractères d'interlignes (NL).

- `errposition(int cursorpos)` spécifie l'endroit textuel d'une erreur dans la chaîne de caractères de la requête. Actuellement, c'est seulement utile pour les erreurs détectées dans les phases d'analyses lexicales et syntaxiques du traitement de la requête.

- `errtable(Relation rel)` spécifie une relation dont le nom et le schéma doivent être inclus comme champs auxiliaires du rapport d'erreur.

- `errtablecol(Relation rel, int attnum)` indique une colonne dont le nom, le nom de la table et le nom du schéma doivent être inclus comme champs auxiliaires du rapport d'erreur.

- `errtableconstraint(Relation rel, const char *conname)` spécifie une contrainte de table dont le nom, le nom de la table et le nom du schéma doivent être inclus comme champs du rapport d'erreur. Les index doivent être considérés comme des contraintes dans ce but, qu'ils soient ou non associés à une entrée dans `pg_constraint`. Faites attention à fournir la relation principale sous-jacente et non pas l'index lui-même, via `rel`.

- `errdatatype(Oid datatypeOid)` spécifie un type de données dont le nom et le nom du schéma doivent être inclus comme champs auxiliaires dans le rapport d'erreur.

- `errdomainconstraint(Oid datatypeOid, const char *conname)` spécifie une contrainte de domaine dont le nom, le nom du domaine et le nom du schéma doivent être inclus comme champs auxiliaires du rapport d'erreur.

- `errcode_for_file_access()` est une fonction commode qui sélectionne l'identifiant d'erreur SQLSTATE approprié pour une défaillance dans l'appel système relatif à l'accès d'un fichier. Elle utilise le `errno` sauvegardé pour déterminer quel code d'erreur générer. Habituellement cela devrait être utilisé en combinaison avec `%m` dans le texte du message d'erreur primaire.

- `errcode_for_socket_access()` est une fonction commode qui sélectionne l'identifiant d'erreur SQLSTATE approprié pour une défaillance dans l'appel système relatif à un socket.

- `errhidestmt(bool hide_stmt)` peut être appelé pour indiquer la suppression de la portion `STATEMENT:` d'un message dans le journal applicatif de postmaster. Habituellement, c'est approprié si le texte du message contient déjà l'instruction en cours.

- `errhidecontext(bool hide_ctx)` peut être appelé pour spécifier la suppression de la portion `CONTEXT:` d'un message dans les traces de postmaster. Ceci devrait seulement être utilisé pour les messages verbeux de débogage où l'inclusion répétée de contexte ferait grossir les journaux trop fortement.

> **Note**
>
> Au moins une des fonctions `errtable`, `errtablecol`, `errtableconstraint`, `errdatatype` ou `errdomainconstraint` doivent être utilisées dans un appel à

`ereport`. Ces fonctions existent pour permettre aux applications d'extraire le nom de l'objet de la base associé à l'erreur sans avoir à examiner le texte du messahe d'erreur potentiellement traduit. Ces fonctions doivent être utilisées dans les rapports d'erreur pour lesquels il est probable que les applications voudraient une gestion automatique des erreurs. À partir de PostgreSQL 9.3, une couverture complète existe pour les erreurs de la classe SQLSTATE 23 (violation des contraintes d'intégrité), mais il est probable que cette couverture soit étendue dans les prochaines versions.

Il y a une plus ancienne fonction nommée `elog`, qui est toujours largement utilisée. Un appel à `elog` :

```
elog(niveau, "chaine format", ...);
```

est strictement équivalent à :

```
ereport(level, errmsg_internal("chaine format", ...));
```

Le code d'erreur SQLSTATE est toujours celui par défaut, la chaîne de caractères du message n'est pas sujette à traduction. Par conséquent, `elog` ne devrait être utilisé que pour les erreurs internes et l'enregistrement de trace de débogage de bas niveau. N'importe quel message susceptible d'intéresser les utilisateurs ordinaires devrait passer par `ereport`. Néanmoins, il y a suffisamment de contrôles des erreurs internes qui « ne peuvent pas arriver » dans le système, pour que `elog` soit toujours largement utilisée ; elle est préférée pour ces messages du fait de sa simplicité d'écriture.

Des conseils sur l'écriture de bons messages d'erreur peuvent être trouvés dans la Section 56.3.

56.3. Guide de style des messages d'erreur

Ce guide de style est fourni dans l'espoir de maintenir une cohérence et un style facile à comprendre dans tous les messages générés par PostgreSQL.

Ce qui va où

Le message primaire devrait être court, factuel et éviter les références aux détails d'exécution comme le nom de fonction spécifique. « Court » veut dire « devrait tenir sur une ligne dans des conditions normales ». Utilisez un message détail si nécessaire pour garder le message primaire court ou si vous sentez le besoin de mentionner les détails de l'implémentation comme un appel système particulier qui échoue. Les messages primaires et détails doivent être factuels. Utilisez un message conseil pour les suggestions à propos de quoi faire pour fixer le problème, spécialement si la suggestion ne pourrait pas toujours être applicable.

Par exemple, au lieu de :

```
IpcMemoryCreate: shmget(clé=%d, taille=%u, 0%o) a échoué : %m
(plus un long supplément qui est basiquement un conseil)
```

écrivez :

```
Primaire:    Ne peut pas créer un segment en mémoire partagée : %m
Détail:      L'appel système qui a échoué était shmget(key=%d,
 size=%u, 0%o).
Astuce:     Le supplément
```

Raisonnement : garder le message primaire court aide à le garder au point et laisse les clients présenter un espace à l'écran sur la supposition qu'une ligne est suffisante pour les messages d'erreurs. Les messages détails et conseils peuvent être relégués à un mode verbeux ou peut-être dans une fenêtre pop-up détaillant l'erreur. De plus, les détails et les conseils devront normalement être supprimés des traces du serveur pour gagner de l'espace. La référence aux détails d'implémentation est à éviter puisque les utilisateurs ne sont pas supposés connaître tous les détails.

Formatage

N'émettez pas d'hypothèses spécifiques à propos du formatage dans les messages textes. Attendez-vous à ce que les clients et les traces du serveur enveloppent les lignes pour correspondre à leurs propres besoins. Dans les longs messages, les caractères d'interlignes (\n) peuvent être utilisés pour indiquer les coupures suggérées d'un paragraphe. Ne terminez pas un message avec un caractère d'interlignes. N'utilisez pas des tabulations ou d'autres caractères de formatage (dans les affichages des contextes d'erreurs, les caractères d'interlignes sont automatiquement ajoutés pour séparer les niveaux d'un contexte comme dans les appels aux fonctions).

Raisonnement : les messages ne sont pas nécessairement affichés dans un affichage de type terminal. Dans les interfaces graphiques ou les navigateurs, ces instructions de formatage sont, au mieux, ignorées.

Guillemets

Les textes en anglais devraient utiliser des guillemets doubles quand la mise entre guillemets est appropriée. Les textes dans les autres langues devraient uniformément employer un genre de guillemets qui est conforme aux coutumes de publication et à la sortie visuelle des autres programmes.

Raisonnement : le choix des guillemets doubles sur celui des guillemets simples est quelque peu arbitraire mais tend à être l'utilisation préférée. Certains ont suggéré de choisir le type de guillemets en fonction du type d'objets des conventions SQL (notamment, les chaînes de caractères entre guillemets simples, les identifiants entre guillemets doubles). Mais ceci est un point technique à l'intérieur du langage avec lequel beaucoup d'utilisateurs ne sont pas familiers ; les conventions SQL ne prennent pas en compte les autres genres de termes entre guillemets, ne sont pas traduites dans d'autres langues et manquent un peu de sens aussi.

Utilisation des guillemets

Utilisez toujours les guillemets pour délimiter les noms de fichiers, les identifiants fournis par les utilisateurs et les autres variables qui peuvent contenir des mots. Ne les utilisez pas pour marquer des variables qui ne contiennent pas de mots (par exemple, les noms d'opérateurs).

Il y a des fonctions au niveau du serveur qui vont, au besoin, mettre entre guillemets leur propre flux de sortie (par exemple, `format_type_be()`). Ne mettez pas de guillemets supplémentaires autour du flux de sortie de ce genre de fonctions.

Raisonnement : les objets peuvent avoir un nom qui crée une ambiguïté une fois incorporé dans un message. Soyez prudent en indiquant où un nom commence et fini. Mais n'encombrez pas les messages avec des guillemets qui ne sont pas nécessaires ou qui sont dupliqués.

Grammaire et ponctuation

Les règles sont différentes pour les messages d'erreurs primaires et pour les messages détails/conseils :

Messages d'erreurs primaires : ne mettez pas en majuscule la première lettre. Ne terminez pas un message avec un point. Ne pensez même pas à finir un message avec un point d'exclamation.

Messages détails et conseils : utilisez des phrases complètes et toutes terminées par des points. Mettez en majuscule le premier mot des phrases. Placez deux espaces après le point si une autre phrase suit (pour un texte en anglais... cela pourrait être différent dans une autre langue).

Chaînes de contexte d'erreur: Ne mettez pas en majuscule la première lettre et ne terminer pas la chaîne avec un point. Les chaînes de contexte ne sont normalement pas des phrases complètes.

Raisonnement : éviter la ponctuation rend plus facile, pour les applications clientes, l'intégration du message dans des contextes grammaticaux variés. Souvent, les messages primaires ne sont de toute

façon pas des phrases complètes (et s'ils sont assez longs pour être sur plusieurs phrases, ils devraient être divisés en une partie primaire et une partie détail). Cependant, les messages détails et conseils sont longs et peuvent avoir besoin d'inclure de nombreuses phrases. Pour la cohérence, ils devraient suivre le style des phrases complètes même s'il y a seulement une phrase.

Majuscule contre minuscule

Utilisez les minuscules pour les mots d'un message, inclus la première lettre d'un message d'erreur primaire. Utilisez les majuscules pour les commandes et les mots-clé SQL s'ils apparaissent dans le message.

Raisonnement : il est plus facile de rendre toutes les choses plus cohérentes au regard de cette façon, puisque certains messages sont des phrases complètes et d'autres non.

Éviter la voix passive

Utilisez la voix active. Utilisez des phrases complètes quand il y a un sujet (« A ne peut pas faire B »). Utilisez le style télégramme, sans sujet, si le sujet est le programme lui-même ; n'utilisez pas « Je » pour le programme.

Raisonnement : le programme n'est pas humain. Ne prétendez pas autre chose.

Présent contre passé

Utilisez le passé si une tentative de faire quelque chose échouait, mais pourrait peut-être réussir la prochaine fois (peut-être après avoir corriger certains problèmes). Utilisez le présent si l'échec est sans doute permanent.

Il y a une différence sémantique non triviale entre les phrases de la forme :

```
n'a pas pu ouvrir le fichier "%s": %m
```

et :

```
ne peut pas ouvrir le dossier "%s"
```

La première forme signifie que la tentative d'ouverture du fichier a échoué. Le message devrait donner une raison comme « disque plein » ou « le fichier n'existe pas ». Le passé est approprié parce que la prochaine fois le disque peut ne plus être plein ou le fichier en question peut exister.

La seconde forme indique que la fonctionnalité d'ouvrir le fichier nommé n'existe pas du tout dans le programme ou que c'est conceptuellement impossible. Le présent est approprié car la condition persistera indéfiniment.

Raisonnement : d'accord, l'utilisateur moyen ne sera pas capable de tirer de grandes conclusions simplement à partir du temps du message mais, puisque la langue nous fournit une grammaire, nous devons l'utiliser correctement.

Type de l'objet

En citant le nom d'un objet, spécifiez quel genre d'objet c'est.

Raisonnement : sinon personne ne saura ce qu'est « foo.bar.baz ».

Crochets

Les crochets sont uniquement utilisés (1) dans les synopsis des commandes pour indiquer des arguments optionnels ou (2) pour indiquer l'indice inférieur d'un tableau.

Raisonnement : rien de ce qui ne correspond pas à l'utilisation habituelle, largement connue troublera les gens.

Assembler les messages d'erreurs

Quand un message inclut du texte produit ailleurs, il est intégré dans ce style :

```
n'a pas pu ouvrir le fichier %s: %m
```

Raisonnement : il serait difficile d'expliquer tous les codes d'erreurs possibles pour coller ceci dans une unique phrase douce, ainsi une certaine forme de ponctuation est nécessaire. Mettre le texte inclus entre parenthèses a été également suggéré, mais ce n'est pas naturel si le texte inclus est susceptible d'être la partie la plus importante du message, comme c'est souvent le cas.

Raisons pour les erreurs

Les messages devraient toujours indiquer la raison pour laquelle une erreur s'est produite. Par exemple :

```
MAUVAIS :  n'a pas pu ouvrir le fichier %s
MEILLEUR : n'a pas pu ouvrir le fichier %s (échec E/S)
```

Si aucune raison n'est connue, vous feriez mieux de corriger le code.

Nom des fonctions

N'incluez pas le nom de la routine de rapport dans le texte de l'erreur. Nous avons d'autres mécanismes pour trouver cela quand c'est nécessaire et, pour la plupart des utilisateurs, ce n'est pas une information utile. Si le texte de l'erreur n'a plus beaucoup de sens sans le nom de la fonction, reformulez-le.

```
+BAD:    pg_strtoint32: error in "z": cannot parse "z"
BETTER: invalid input syntax for type integer: "z"
```

Évitez de mentionner le nom des fonctions appelées, au lieu de cela dites ce que le code essayait de faire :

```
MAUVAIS :  ouvrir() a échoué : %m
MEILLEUR : n'a pas pu ouvrir le fichier %s: %m
```

Si cela semble vraiment nécessaire, mentionnez l'appel système dans le message détail (dans certains cas, fournir les valeurs réelles passées à l'appel système pourrait être une information appropriée pour le message détail).

Raisonnement : les utilisateurs ne savent pas tout ce que ces fonctions font.

Mots délicats à éviter

Incapable. « Incapable » est presque la voix passive. Une meilleure utilisation est « ne pouvait pas » ou « ne pourrait pas » selon les cas.

Mauvais. Les messages d'erreur comme « mauvais résultat » sont vraiment difficile à interpréter intelligemment. Cela est mieux d'écrire pourquoi le résultat est « mauvais », par exemple, « format invalide ».

Illégal. « Illégal » représente une violation de la loi, le reste est « invalide ». Meilleur encore, dites pourquoi cela est invalide.

Inconnu. Essayez d'éviter « inconnu ». Considérez « erreur : réponse inconnue ». Si vous ne savez pas qu'elle est la réponse, comment savez-vous que cela est incorrect ? « Non reconnu » est souvent un meilleur choix. En outre, assurez-vous d'inclure la valeur pour laquelle il y a un problème.

```
MAUVAIS :  type de nœud inconnu
MEILLEUR : type de nœud non reconnu : 42
```

Trouver contre Exister. Si le programme emploie un algorithme non trivial pour localiser une ressource (par exemple, une recherche de chemin) et que l'algorithme échoue, il est juste de dire que le programme n'a pas pû « trouver » la ressource. D'un autre côté, si l'endroit prévu pour la ressource est connu mais que le programme ne peut pas accéder à celle-ci, alors dites que la ressource n'« existe » pas. Utilisez « trouvez » dans ce cas-là semble faible et embrouille le problème.

May vs. Can vs. Might. « May » suggère un droit (par exemple *You may borrow my rake.*) et a peu d'utilité dans la documentation et dans les messages d'erreur. « Can » suggère une capacité (par exemple *I can lift that log.*), et « might » suggère une possibilité (par exemple *It might rain today.*). Utiliser le bon mot clarifie la signification et aide les traducteurs.

Contractions. Éviter les contractions comme « can't » ; utilisez « cannot » à la place.

Non négatif. Éviter « non-negative » car c'est ambigu sur l'acceptation ou non de zéro. Il est préférable d'utiliser « greater than zero » ou « greater than or equal to zero ».

Orthographe appropriée

Orthographiez les mots en entier. Par exemple, évitez :

- spec (NdT : spécification)
- stats (NdT : statistiques)
- params (NdT : paramètres)
- auth (NdT : authentification)
- xact (NdT : transaction)

Raisonnement : cela améliore la cohérence.

Adaptation linguistique

Gardez à l'esprit que les textes des messages d'erreur ont besoin d'être traduit en d'autres langues. Suivez les directives dans la Section 57.2.2 pour éviter de rendre la vie difficile aux traducteurs.

56.4. Conventions diverses de codage

Standard C

Le code dans PostgreSQL devrait seulement se baser sur les fonctionnalités disponibles dans le standard C99. Ceci signifie qu'un compilateur se conformant au standard C99 doit être capable de compiler PostgreSQL, à l'exception possible de quelques parties dépendantes de la plateforme.

Certaines fonctionnalités incluses dans le standard C99 ne sont actuellement pas autorisées dans le code principal de PostgreSQL. Ceci inclut actuellement les tableaux de longueur variable, les déclarations mêlées dans le code, les commentaires `//`, les noms de caractères universels. Les raisons incluent entre autre la portabilité et les pratiques historiques.

Les fonctionnalités des révisions ultérieures du standard C ou du compilateur peuvent être utilisées si un contournement est fourni.

Par exemple `_Static_assert()` et `__builtin_constant_p` sont actuellement utilisés, même s'ils font partie, respectivement, d'une révision plus récente du standard C et d'une

extension GCC. S'ils ne sont pas disponibles, nous retournons respectivement vers l'utilisation d'un remplacement C99 compatible réalisant les mêmes vérifications, mais émet des messages plutôt incompréhensibles et nous n'utilisons pas `__builtin_constant_p`.

Macros du style fonctions et fonctions inline

Les macros avec arguments et les fonctions `static inline` peuvent être utilisés. Ces dernières sont préférables s'il y a un risque de multiples évaluations si elles sont écrites en tant que macro, comme le cas avec

```
#define Max(x, y)       ((x) > (y) ? (x) : (y))
```

ou quand la macro deviendrait très longue. Dans d'autres cas, il est possible d'utiliser des macros ou au moins plus facilement. Par exemple parce que des expressions de types divers ont besoin d'être passées à la macro.

Quand la définition d'une fonction inline référence des symboles (autrement dit des variables, des fonctions) uniquement disponibles dans le moteur, la fonction pourrait ne pas être visible lorsqu'elle est incluse dans le code frontend.

```
#ifndef FRONTEND
static inline MemoryContext
MemoryContextSwitchTo(MemoryContext context)
{
    MemoryContext old = CurrentMemoryContext;

    CurrentMemoryContext = context;
    return old;
}
#endif   /* FRONTEND */
```

Dans cet exemple, `CurrentMemoryContext`, qui est seulement disponible dans le moteur, est référencé et la fonction est donc cachée avec un `#ifndef FRONTEND`. Cette règle existe parce que certains compilateurs émettent des références aux symboles contenus dans les fonctions inline même si la fonction n'est pas utilisée.

Écrire des gestionnaires de signaux

Pour pouvoir être exécuté à l'intérieur d'un gestionnaire de signal, le code doit être écrit avec beaucoup d'attention. Le problème fondamental est qu'une gestion de signal peut interrompre le code à tout moment, sauf s'il est bloqué. Si le code à l'intérieur d'un gestionnaire de signal utilise le même état que le code en dehors, un grand chaos peut survenir. Comme exemple, pensez à ce qui arriverait si un gestionnaire de signal essaie d'obtenir un verrou qui est déjà détenu par le code interrompu.

En dehors d'arrangements spéciaux, le code dans les gestionnaires de signaux doit seulement appeler des fonctions saines de signal asynchrone (d'après la définition de POSIX) et accèder à des variables de type `volatile sig_atomic_t`. Quelques fonctions dans `postgres` sont aussi déclarées comme saines pour les signaux, notamment `SetLatch()`.

Dans la plupart des cas, les gestionnaires de signaux ne devraient rien faire de plus que de noter qu'un signal est arrivé, et réveiller du code à l'extérieur du gestionnaire en utilisant un *latch*. Voici un exemple d'un tel gestionnaire :

```
static void
```

```
handle_sighup(SIGNAL_ARGS)
{
    int         save_errno = errno;

    got_SIGHUP = true;
    SetLatch(MyLatch);

    errno = save_errno;
}
```

`errno` est sauvegardé puis restauré parce que `SetLatch()` pourrait le modifier. Si cela n'était pas fait, le code interrompi qui était en train d'inspecter `errno` pourrait voir la mauvaise valeur.

Appeler des pointeurs de fonction

Pour plus de clarté, il est préféré de déréférencer explicitement un pointeur de fonction lors de l'appel de cette fonction si le pointeur est une simple variable, par exemple :

```
(*emit_log_hook) (edata);
```

(même si `emit_log_hook(edata)` fonctionnerait aussi). Quand le pointeur de fonction fait partie d'une structure, la ponctuation supplémentaire peut et devrait habituellement être omise. Par exemple :

```
paramInfo->paramFetch(paramInfo, paramId);
```

Chapitre 57. Support natif des langues

57.1. Pour le traducteur

Les programmes PostgreSQL (serveur et client) peuvent afficher leur message dans la langue préférée de l'utilisateur -- si les messages ont été traduits. Créer et maintenir les ensembles de messages traduits nécessite l'aide de personnes parlant leur propre langue et souhaitant contribuer à PostgreSQL. Il n'est nul besoin d'être un développeur pour cela. Cette section explique comment apporter son aide.

57.1.1. Prérequis

Les compétences dans sa langue d'un traducteur ne seront pas jugées -- cette section concerne uniquement les outils logiciels. Théoriquement, seul un éditeur de texte est nécessaire. Mais ceci n'est vrai que dans le cas très improbable où un traducteur ne souhaiterait pas tester ses traductions des messages. Lors de la configuration des sources, il faudra s'assurer d'utiliser l'option `--enable-nls`. Ceci assurera également la présence de la bibliothèque libintl et du programme `msgfmt` dont tous les utilisateurs finaux ont indéniablement besoin. Pour tester son travail, il sera utile de suivre les parties pertinentes des instructions d'installation.

Pour commencer un nouvel effort de traduction ou pour faire un assemblage de catalogues de messages (décrit ci-après), il faudra installer respectivement les programmes `xgettext` et `msgmerge` dans une implémentation compatible GNU. Il est prévu dans le futur que `xgettext` ne soit plus nécessaire lorsqu'une distribution empaquetée des sources est utilisée (en travaillant à partir du Git, il sera toujours utile). GNU Gettext 0.10.36 ou ultérieure est actuellement recommandé.

Toute implémentation locale de gettext devrait être disponible avec sa propre documentation. Une partie en est certainement dupliquée dans ce qui suit mais des détails complémentaires y sont certainement disponibles.

57.1.2. Concepts

Les couples de messages originaux (anglais) et de leurs (possibles) traductions sont conservés dans les *catalogues de messages*, un pour chaque programme (bien que des programmes liés puissent partager un catalogue de messages) et pour chaque langue cible. Il existe deux formats de fichiers pour les catalogues de messages : le premier est le fichier « PO » (pour "Portable Object" ou Objet Portable), qui est un fichier texte muni d'une syntaxe spéciale et que les traducteurs éditent. Le second est un fichier « MO » (pour "Machine Object" ou Objet Machine), qui est un fichier binaire engendré à partir du fichier PO respectif et qui est utilisé lorsque le programme internationalisé est exécuté. Les traducteurs ne s'occupent pas des fichiers MO ; en fait, quasiment personne ne s'en occupe.

L'extension du fichier de catalogue de messages est, sans surprise, soit `.po`, soit `.mo`. Le nom de base est soit le nom du programme qu'il accompagne soit la langue utilisée dans le fichier, suivant la situation. Ceci peut s'avérer être une source de confusion. Des exemples sont `psql.po` (fichier PO pour psql) ou `fr.mo` (fichier MO en français).

Le format du fichier PO est illustré ici :

```
# commentaire

msgid "chaîne originale"
msgstr "chaîne traduite"

msgid "encore une originale"
```

```
msgstr "encore une de traduite"
"les chaînes peuvent être sur plusieurs lignes, comme ceci"

...
```

Les chaînes msgid sont extraites des sources du programme. (Elles n'ont pas besoin de l'être mais c'est le moyen le plus commun). Les lignes msgstr sont initialement vides puis complétées avec les chaînes traduites. Les chaînes peuvent contenir des caractères d'échappement de style C et peuvent être sur plusieurs lignes comme le montre l'exemple ci-dessus (la ligne suivante doit démarrer au début de la ligne).

Le caractère # introduit un commentaire. Si une espace fine suit immédiatement le caractère #, c'est qu'il s'agit là d'un commentaire maintenu par le traducteur. On trouve aussi des commentaires automatiques qui n'ont pas d'espace fine suivant immédiatement le caractère #. Ils sont maintenus par les différents outils qui opèrent sur les fichiers PO et ont pour but d'aider le traducteur.

```
#. commentaire automatique
#: fichier.c:1023
#, drapeau, drapeau
```

Les commentaires du style #. sont extraits du fichier source où le message est utilisé. Il est possible que le développeur ait ajouté des informations pour le traducteur, telles que l'alignement attendu. Les commentaires #: indiquent l'emplacement exact où le message est utilisé dans le source. Le traducteur n'a pas besoin de regarder le source du programme, mais il peut le faire s'il subsiste un doute sur l'exactitude d'une traduction. Le commentaire #, contient des drapeaux décrivant le message d'une certaine façon. Il existe actuellement deux drapeaux : `fuzzy` est positionné si le message risque d'être rendu obsolète par des changements dans les sources. Le traducteur peut alors vérifier ceci et supprimer ce drapeau. Notez que les messages « fuzzy » ne sont pas accessibles à l'utilisateur final. L'autre drapeau est `c-format` indiquant que le message utilise le format de la fonction C `printf`. Ceci signifie que la traduction devrait aussi être de ce format avec le même nombre et le même type de paramètres fictifs. Il existe des outils qui vérifient que le message est une chaîne au format printf et valident le drapeau c-format en conséquence.

57.1.3. Créer et maintenir des catalogues de messages

OK, alors comment faire pour créer un catalogue de messages « vide » ? Tout d'abord, se placer dans le répertoire contenant le programme dont on souhaite traduire les messages. S'il existe un fichier `nls.mk`, alors ce programme est préparé pour la traduction.

S'il y a déjà des fichiers `.po`, alors quelqu'un a déjà réalisé des travaux de traduction. Les fichiers sont nommés *`langue`*`.po`, où *`langue`* est le code de langue sur deux caractères (en minuscules) tel que défini par l' ISO 639-1, le code du pays composé de deux lettres en minuscule[1], c'est-à-dire `fr.po` pour le français. S'il existe réellement un besoin pour plus d'une traduction par langue, alors les fichiers peuvent être renommés *`langue_region`*`.po` où *`region`* est le code de langue sur deux caractères (en majuscules), tel que défini par l'ISO 3166-1, le code du pays sur deux lettres en majuscule[2], c'est-à-dire `pt_BR.po` pour le portugais du Brésil. Si vous trouvez la langue que vous souhaitez, vous pouvez commencer à travailler sur ce fichier.

Pour commencer une nouvelle traduction, il faudra préalablement exécuter la commande :

```
make init-po
```

Ceci créera un fichier *`nomprog`*`.pot`. (`.pot` pour le distinguer des fichiers PO qui sont « en production ». Le `T` signifie « template » (NdT : modèle en anglais). On copiera ce fichier sous le nom

[1] https://www.loc.gov/standards/iso639-2/php/English_list.php
[2] https://www.iso.org/iso/country_names_and_code_elements

langue`.po`. On peut alors l'éditer. Pour faire savoir qu'une nouvelle langue est disponible, il faut également éditer le fichier `nls.mk` et y ajouter le code de la langue (ou de la langue et du pays) avec une ligne ressemblant à ceci :

```
AVAIL_LANGUAGES := de fr
```

(d'autres langues peuvent apparaître, bien entendu).

À mesure que le programme ou la bibliothèque change, des messages peuvent être modifiés ou ajoutés par les développeurs. Dans ce cas, il n'est pas nécessaire de tout recommencer depuis le début. À la place, on lancera la commande :

```
make update-po
```

qui créera un nouveau catalogue de messages vides (le fichier pot avec lequel la traduction a été initiée) et le fusionnera avec les fichiers PO existants. Si l'algorithme de fusion a une incertitude sur un message particulier, il le marquera « fuzzy » comme expliqué ci-dessus. Le nouveau fichier PO est sauvegardé avec l'extension `.po.new`.

57.1.4. Éditer les fichiers PO

Les fichiers PO sont éditables avec un éditeur de texte standard. Le traducteur doit seulement modifier l'emplacement entre les guillemets après la directive msgstr, peut ajouter des commentaires et modifier le drapeau fuzzy (NdA : Il existe, ce qui n'est pas surprenant, un mode PO pour Emacs, que je trouve assez utile).

Les fichiers PO n'ont pas besoin d'être entièrement remplis. Le logiciel retournera automatiquement à la chaîne originale si une traduction n'est pas disponible ou est laissée vide. Soumettre des traductions incomplètes pour les inclure dans l'arborescence des sources n'est pas un problème ; cela permet à d'autres personnes de récupérer le travail commencé pour le continuer. Néanmoins, les traducteurs sont encouragés à donner une haute priorité à la suppression des entrées fuzzy après avoir fait une fusion. Les entrées fuzzy ne seront pas installées ; elles servent seulement de référence à ce qui pourrait être une bonne traduction.

Certaines choses sont à garder à l'esprit lors de l'édition des traductions :

- S'assurer que si la chaîne originale se termine par un retour chariot, la traduction le fasse bien aussi. De même pour les tabulations, etc.

- Si la chaîne originale est une chaîne au format `printf`, la traduction doit l'être aussi. La traduction doit également avoir les même spécificateurs de format et dans le même ordre. Quelquefois, les règles naturelles de la langue rendent cela impossible ou tout au moins difficile. Dans ce cas, il est possible de modifier les spécificateurs de format de la façon suivante :

  ```
  msgstr "Die Datei %2$s hat %1$u Zeichen."
  ```

  Le premier paramètre fictif sera alors utilisé par le deuxième argument de la liste. Le *chiffre*`$` a besoin de suivre immédiatement le %, avant tout autre manipulateur de format (cette fonctionnalité existe réellement dans la famille des fonctions `printf`, mais elle est peu connue, n'ayant que peu d'utilité en dehors de l'internationalisation des messages).

- Si la chaîne originale contient une erreur linguistique, on pourra la rapporter (ou la corriger soi-même dans le source du programme) et la traduire normalement. La chaîne corrigée peut être fusionnée lorsque les programmes sources auront été mis à jour. Si la chaîne originale contient une erreur factuelle, on la rapportera (ou la corrigera soi-même) mais on ne la traduira pas. À la place, on marquera la chaîne avec un commentaire dans le fichier PO.

- Maintenir le style et le ton de la chaîne originale. En particulier, les messages qui ne sont pas des phrases (`cannot open file %s`, soit `ne peut pas ouvrir le fichier %s`) ne devraient probablement pas commencer avec une lettre capitale (si votre langue distingue la casse des lettres) ou finir avec un point (si votre langue utilise des marques de ponctuation). Lire Section 56.3 peut aider.

- Lorsque la signification d'un message n'est pas connue ou s'il est ambigu, on pourra demander sa signification sur la liste de diffusion des développeurs. Il est possible qu'un anglophone puisse aussi ne pas le comprendre ou le trouver ambigu. Il est alors préférable d'améliorer le message.

57.2. Pour le développeur

57.2.1. Mécaniques

Cette section explique comment implémenter le support natif d'une langue dans un programme ou dans une bibliothèque qui fait partie de la distribution PostgreSQL. Actuellement, cela s'applique uniquement aux programmes C.

Ajouter le support NLS à un programme

1. Le code suivant est inséré dans la séquence initiale du programme :

   ```
   #ifdef ENABLE_NLS
   #include <locale.h>
   #endif

   ...

   #ifdef ENABLE_NLS
   setlocale(LC_ALL, "");
   bindtextdomain("nomprog", LOCALEDIR);
   textdomain("nomprog");
   #endif
   ```

   (*nomprog* peut être choisi tout à fait librement).

2. Partout où un message candidat à la traduction est trouvé, un appel à `gettext()` doit être inséré. Par exemple :

   ```
   fprintf(stderr, "panic level %d\n", lvl);
   ```

   devra être changé avec :

   ```
   fprintf(stderr, gettext("panic level %d\n"), lvl);
   ```

   (`gettext` est défini comme une opération nulle si NLS n'est pas configuré).

   Cela peut engendrer du fouillis. Un raccourci habituel consiste à utiliser :

   ```
   #define _(x) gettext(x)
   ```

   Une autre solution est envisageable si le programme effectue la plupart de ses communications via une fonction ou un nombre restreint de fonctions, telle `ereport()` pour le moteur. Le fonctionnement interne de cette fonction peut alors être modifiée pour qu'elle appelle `gettext` pour toutes les chaînes en entrée.

3. Un fichier `nls.mk` est ajouté dans le répertoire des sources du programme. Ce fichier sera lu comme un makefile. Les affectations des variables suivantes doivent être réalisées ici :

   `CATALOG_NAME`

   Le nom du programme tel que fourni lors de l'appel à `textdomain()`.

   `AVAIL_LANGUAGES`

   Liste des traductions fournies -- initialement vide.

   `GETTEXT_FILES`

   Liste des fichiers contenant les chaînes traduisibles, c'est-à-dire celles marquées avec `gettext` ou avec une solution alternative. Il se peut que cette liste inclut pratiquement tous les fichiers sources du programme. Si cette liste est trop longue, le premier « fichier » peut être remplacé par un + et le deuxième mot représenter un fichier contenant un nom de fichier par ligne.

   `GETTEXT_TRIGGERS`

   Les outils qui engendrent des catalogues de messages pour les traducteurs ont besoin de connaître les appels de fonction contenant des chaînes à traduire. Par défaut, seuls les appels à `gettext()` sont reconnus. Si _ ou d'autres identifiants sont utilisés, il est nécessaire de les lister ici. Si la chaîne traduisible n'est pas le premier argument, l'élément a besoin d'être de la forme `func:2` (pour le second argument). Si vous avez une fonction qui supporte les messages au format pluriel, l'élément ressemblera à `func:1,2` (identifiant les arguments singulier et pluriel du message).

Le système de construction s'occupera automatiquement de construire et installer les catalogues de messages.

57.2.2. Guide d'écriture des messages

Voici quelques lignes de conduite pour l'écriture de messages facilement traduisibles.

- Ne pas construire de phrases à l'exécution, telles que :

  ```
  printf("Files were %s.\n", flag ? "copied" : "removed");
  ```

  L'ordre des mots d'une phrase peut être différent dans d'autres langues. De plus, même si `gettext()` est correctement appelé sur chaque fragment, il pourrait être difficile de traduire séparément les fragments. Il est préférable de dupliquer un peu de code de façon à ce que chaque message à traduire forme un tout cohérent. Seuls les nombres, noms de fichiers et autres variables d'exécution devraient être insérés au moment de l'exécution dans le texte d'un message.

- Pour des raisons similaires, ceci ne fonctionnera pas :

  ```
  printf("copied %d file%s", n, n!=1 ? "s" : "");
  ```

  parce que cette forme présume de la façon dont la forme plurielle est obtenue. L'idée de résoudre ce cas de la façon suivante :

  ```
  if (n==1)
      printf("copied 1 file");
  else
      printf("copied %d files", n):
  ```

sera source de déception. Certaines langues ont plus de deux formes avec des règles particulières. Il est souvent préférable de concevoir le message de façon à éviter le problème, par exemple ainsi :

```
printf("number of copied files: %d", n);
```

Si vous voulez vraiment construire un message correctement pluralisé, il existe un support pour cela, mais il est un peu étrange. Quand vous générez un message d'erreur primaire ou détaillé dans `ereport()`, vous pouvez écrire quelque-chose comme ceci :

```
errmsg_plural("copied %d file",
              "copied %d files",
              n,
              n)
```

Le premier argument est la chaîne dans le format approprié pour la forme au singulier en anglais, le second est le format de chaîne approprié pour la forme plurielle en anglais, et le troisième est la valeur entière déterminant la forme à utiliser. Des arguments additionnels sont formatés suivant la chaîne de formatage comme d'habitude. (Habituellement, la valeur de contrôle de la pluralisation sera aussi une des valeurs à formater, donc elle sera écrite deux fois.) En anglais, cela n'importe que si *n* est égale à 1 ou est différent de 1, mais dans d'autres langues, il pourrait y avoir plusieurs formes de pluriel. Le traducteur voit les deux formes anglaises comme un groupe et a l'opportunité de fournir des chaînes de substitution supplémentaires, la bonne étant sélectionnée suivant la valeur à l'exécution de *n*.

Si vous avez besoin de pluraliser un message qui ne va pas directement à `errmsg` ou `errdetail`, vous devez utiliser la fonction sous-jacente `ngettext`. Voir la documentation gettext.

- Lorsque quelque chose doit être communiqué au traducteur, telle que la façon dont un message doit être aligné avec quelque autre sortie, on pourra faire précéder l'occurrence de la chaîne d'un commentaire commençant par `translator`, par exemple :

  ```
  /* translator: This message is not what it seems to be. */
  ```

  Ces commentaires sont copiés dans les catalogues de messages de façon à ce que les traducteurs les voient.

Chapitre 58. Écrire un gestionnaire de langage procédural

Tous les appels de fonctions écrites dans un langage autre que celui de l'interface « version 1 » pour les langages compilés (ce qui inclut les fonctions dans les langages procéduraux utilisateur, les fonctions SQL), passent par une fonction spécifique au langage du *gestionnaire d'appels*. Le gestionnaire d'appels exécute la fonction de manière appropriée, par exemple en interprétant le code source fourni. Ce chapitre décrit l'écriture du gestionnaire d'appels d'un nouveau langage procédural.

Le gestionnaire d'appel d'un langage procédural est une fonction « normale » qui doit être écrite dans un langage compilé tel que le C, en utilisant l'interface version-1, et enregistrée sous PostgreSQL comme une fonction sans argument et retournant le type `language_handler`. Ce pseudo-type spécial identifie la fonction comme gestionnaire d'appel et empêche son appel à partir des commandes SQL. Pour plus de détails sur les conventions d'appels et le chargement dynamique en langage C, voir Section 38.10.

L'appel du gestionnaire d'appels est identique à celui de toute autre fonction : il reçoit un pointeur de `structure FunctionCallInfoBaseData` qui contient les valeurs des arguments et d'autres informations de la fonction appelée. Il retourne un résultat `Datum` (et, initialise le champ `isnull` de la structure `FunctionCallInfoBaseData` si un résultat SQL NULL doit être retourné). La différence entre un gestionnaire d'appels et une fonction ordinaire se situe au niveau du champ `flinfo->fn_oid` de la structure `FunctionCallInfoBaseData`. Dans le cas du gestionnaire d'appels, il contiendra l'OID de la fonction à appeler, et non pas celui du gestionnaire d'appels lui-même. Le gestionnaire d'appels utilise ce champ pour déterminer la fonction à exécuter. De plus, la liste d'arguments passée a été dressée à partir de la déclaration de la fonction cible, et non pas en fonction du gestionnaire d'appels.

C'est le gestionnaire d'appels qui récupère l'entrée de la fonction dans la table système `pg_proc` et analyse les types des arguments et de la valeur de retour de la fonction appelée. La clause `AS` de la commande `CREATE FUNCTION` se situe dans la colonne `prosrc` de `pg_proc`. Il s'agit généralement du texte source du langage procédural lui-même (comme pour PL/Tcl) mais, en théorie, cela peut être un chemin vers un fichier ou tout ce qui indique au gestionnaire d'appels les détails des actions à effectuer.

Souvent, la même fonction est appelée plusieurs fois dans la même instruction SQL. L'utilisation du champ `flinfo->fn_extra` évite au gestionnaire d'appels de répéter la recherche des informations concernant la fonction appelée. Ce champ, initialement `NULL`, peut être configuré par le gestionnaire d'appels pour pointer sur l'information concernant la fonction appelée. Lors des appels suivants, si `flinfo->fn_extra` est différent de `NULL`, alors il peut être utilisé et l'étape de recherche d'information évitée. Le gestionnaire d'appels doit s'assurer que `flinfo->fn_extra` pointe sur une zone mémoire qui restera allouée au moins jusqu'à la fin de la requête en cours, car une structure de données `FmgrInfo` peut être conservée aussi longtemps. Cela peut être obtenu par l'allocation des données supplémentaires dans le contexte mémoire spécifié par `flinfo->fn_mcxt` ; de telles données ont la même espérance de vie que `FmgrInfo`. Le gestionnaire peut également choisir d'utiliser un contexte mémoire de plus longue espérance de vie de façon à mettre en cache sur plusieurs requêtes les informations concernant les définitions des fonctions.

Lorsqu'une fonction en langage procédural est appelée via un trigger, aucun argument ne lui est passé de façon traditionnelle mais le champ `context` de `FunctionCallInfoBaseData` pointe sur une structure `TriggerData`. Il n'est pas `NULL` comme c'est le cas dans les appels de fonctions standard. Un gestionnaire de langage doit fournir les mécanismes pour que les fonctions de langages procéduraux obtiennent les informations du trigger.

Un modèle de gestionnaire de langage procédural écrit sous la forme d'une extension C est fourni dans `src/test/modules/plsample`. C'est un exemple fonctionnel démontrant une façon de créer un tel gestionnaire, de traiter des paramètres et de renvoyer une valeur. Il suffit de remplacer les points de suspension par quelques milliers de lignes de code pour compléter ce modèle.

Lorsque la fonction du gestionnaire est compilée dans un module chargeable (voir Section 38.10.5), les commandes suivantes enregistrent le langage procédural défini dans l'exemple :

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS 'nomfichier'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Bien que fournir un gestionnaire d'appels est suffisant pour créer un langage de procédures minimal, il existe deux autres fonctions qui peuvent être fournies pour faciliter l'utilisation du langage. Ce sont les fonctions de validation (*validator*) et de traitement en ligne (*inline handler*). Une fonction de validation peut être fournie pour activer une vérification spécifique au langage lors du CREATE FUNCTION. Une fonction de traitement en ligne sera utilisé pour supporter les blocs de code anonymes exécutés via la commande DO.

Si une fonction de validation est fournie par un langage de procédures, elle doit être déclarée comme une fonction prenant un seul paramètre, de type `oid`. Le résultat de la validation est ignoré, donc elle peut renvoyer le type `void`. La fonction de validation sera appelée à la fin de la commande `CREATE FUNCTION` qui a créé ou mis à jour une fonction écrite dans ce langage. L'OID passé en argument est l'OID de la fonction, disponible dans le catalogue `pg_proc`. La fonction de validation doit récupérer cette ligne de la façon habituelle et réaliser les vérifications appropriées. Tout d'abord, elle appelle `CheckFunctionValidatorAccess()` pour diagnostiquer les appels explicites au validateur que l'utilisateur ne peut pas réaliser via `CREATE FUNCTION`. Les vérifications typiques incluent la vérification du support des types en arguments et en sortie, ainsi que la vérification syntaxique du corps de la requête pour ce langage. Si la fonction de validation est satisfait par la fonction, elle quitte sans erreur. Si, par contre, elle trouve une erreur, elle doit rapporter cette erreur au travers du mécanisme `ereport()` standard. Renvoyer une erreur forcera une annulation de la transaction et empêchera du même coup l'enregistrement de la fonction dont la définition est erronée.

Les fonctions de validation devraient typiquement accepter le paramètre check_function_bodies : s'il est désactivé, alors toute vérification coûteuse ou spécifique au contexte devrait être abandonnée. Si le langage permet l'exécution de code à la compilation, le validateur doit supprimer les vérifications qui impliquerait une telle exécution. En particulier, ce paramètre est désactivé par pg_dump, pour qu'il puisse charger le langage de procédures sans avoir à s'inquiéter des effets de bord et des dépendances possibles dans le corps des procédures stockées avec d'autres objets de la base de données. (À cause de cela, le gestionnaire d'appels doit éviter de supposer que la fonction de validation a vérifié complètement la fonction. Le but d'avoir une fonction de validation n'est pas d'éviter au gestionnaire d'appels de faire des vérifications, mais plutôt de notifier immédiatement à l'utilisateur si des erreurs évidentes apparaissent dans la commande `CREATE FUNCTION`.) Bien que le choix de ce qui est à vérifier est laissé à la discrétion de la fonction de validation, il faut noter que le code de `CREATE FUNCTION` exécute seulement les clauses `SET` attachées à la fonction quand le paramètre `check_function_bodies` est activé. Du coup, les vérifications dont les résultats pourraient être affectés par les paramètres en question doivent être ignorées quand `check_function_bodies` est désactivé pour éviter des échecs erronés lors du chargement d'une sauvegarde.

Si une fonction de traitement en ligne est fournie au langage de procédures, elle doit être déclarée comme une fonction acceptant un seul paramètre de type `internal`. Le résultat de la fonction de traitement en ligne est ignoré, donc elle peut renvoyer le type `void`. Elle sera appelée quand une instruction `DO` est exécutée pour ce langage. Le paramètre qui lui est fourni est un pointeur vers une structure `InlineCodeBlock`, structure contenant des informations sur les paramètres de l'instruction `DO`, en particulier le texte du bloc de code anonyme à exécuter. La fonction doit exécuter ce code.

Il est recommandé de placer toutes les déclarations de fonctions ainsi que la commande `CREATE LANGUAGE` dans une *extension* pour qu'une simple commande `CREATE EXTENSION` suffise à installer le langage. Voir Section 38.17 pour plus d'informations sur l'écriture d'extensions.

Les langages procéduraux inclus dans la distribution standard sont de bons points de départ à l'écriture de son propre gestionnaire de langage. Les sources se trouvent dans le répertoire `src/pl`. La page de référence de CREATE LANGUAGE contient aussi certains détails utiles.

Chapitre 59. Écrire un wrapper de données distantes

Toutes les opérations sur une table distante sont gérées via un wrapper de données distantes. Ce dernier est un ensemble de fonctions que PostgreSQL appelle. Le wrapper de données distantes est responsable de la récupération des données à partir de la source de données distante et de leur renvoi à l'exécuteur PostgreSQL. Si la mise à jour de tables distantes doit être supportée, le wrapper doit aussi gérer cela. Ce chapitre indique comment écrire un nouveau wrapper de données distantes.

Les wrappers de données distantes incluent dans la distribution standard sont de bons exemples lorsque vous essayez d'écrire les vôtres. Regardez dans le sous-répertoire `contrib` du répertoire des sources. La page de référence CREATE FOREIGN DATA WRAPPER contient aussi des détails utiles.

> **Note**
>
> Le standard SQL spécifie une interface pour l'écriture des wrappers de données distantes. Néanmoins, PostgreSQL n'implémente pas cette API car l'effort nécessaire pour cela serait trop important. De toute façon, l'API standard n'est pas encore très adoptée.

59.1. Fonctions d'un wrapper de données distantes

Le développeur d'un FDW doit écrire une fonction de gestion (handler) et, en option, une fonction de validation. Les deux fonctions doivent être écrites dans un langage compilé comme le C en utilisant l'interface version-1. Pour les détails sur les conventions d'appel et le chargement dynamique en langage C, voir Section 38.10.

La fonction de gestion renvoie simplement une structure de pointeurs de fonctions callback qui seront appelées par le planificateur, l'exécuteur et différentes commandes de maintenance. La plupart du travail dans l'écriture d'une FDW se trouve dans l'implémentation de ces fonctions callback. La fonction de gestion doit être enregistrée dans PostgreSQL comme ne prenant aucun argument et renvoyant le pseudo-type `fdw_handler`. Les fonctions callback sont des fonctions en C et ne sont pas visibles ou appelables avec du SQL. Les fonctions callback sont décrites dans Section 59.2.

La fonction de validation est responsable de la validation des options données dans les commandes `CREATE` et `ALTER` pour son wrapper de données distantes, ainsi que pour les serveurs distants, les correspondances d'utilisateurs et les tables distants utilisant le wrapper. La fonction de validation doit être enregistrée comme prenant deux arguments : un tableau de texte contenant les options à valider et un OID représentant le type d'objet avec lequel les options sont associées. Ce dernier correspond à l'OID du catalogue système où sera stocké l'objet, donc

- `AttributeRelationId`
- `ForeignDataWrapperRelationId`
- `ForeignServerRelationId`
- `ForeignTableRelationId`
- `UserMappingRelationId`

Si aucune fonction de validation n'est fournie, les options ne sont pas vérifiées au moment de la création ou de la modification de l'objet.

59.2. Routines callback des wrappers de données distantes

La fonction de gestion d'une FDW renvoie une structure `FdwRoutine` allouée avec palloc. Elle contient des pointeurs vers les fonctions de callback décrites ci-dessous. Les fonctions relatives aux parcours sont requises, le reste est optionnel.

Le type de structure `FdwRoutine` est déclaré dans `src/include/foreign/fdwapi.h`, où vous trouverez plus de détails.

59.2.1. Routines des FDW pour parcourir les tables distantes

```
void
GetForeignRelSize (PlannerInfo *root,
                   RelOptInfo *baserel,
                   Oid foreigntableid);
```

Obtient des estimations de la taille de la relation pour une table distante. Elle est appelée au début de la planification d'une requête parcourant une table distante. `root` est l'information globale du planificateur sur la requête ; `baserel` est l'information du planificateur sur la table ; et `foreigntableid` est l'OID provenant de `pg_class` pour cette table distante. (`foreigntableid` pourrait être obtenu à partir de la structure de données du planificateur mais il est directement fourni pour ne pas avoir à faire cet effort.)

Cette fonction doit mettre à jour `baserel->rows` pour que cela corresponde au nombre de lignes renvoyées par un parcours de table après avoir pris en compte le filtre réalisé par les clauses de restriction. La valeur initiale de `baserel->rows` est une estimation par défaut, qui doit être remplacée si possible. La fonction pourrait aussi choisir de mettre à jour `baserel->width` si elle peut calculer une meilleure estimation de la largeur moyenne d'une ligne du résultat. (La valeur initiale est basée sur les types de données des colonnes et sur les valeurs de largeur moyenne des colonnes, mesurées par le dernier `ANALYZE`.) De plus, cette fonction pourrait mettre à jour `baserel->tuples` s'il peut calculer une meilleure estimation du nombre total de lignes de ma table distante. (La valeur initiale provient de `pg_class.reltuples` qui représente le nombre total de lignes vues par le dernier `ANALYZE` ; ça sera `-1` si aucun `ANALYZE` n'a été exécuté sur la table distante.)

Voir Section 59.4 pour plus d'informations.

```
void
GetForeignPaths (PlannerInfo *root,
                 RelOptInfo *baserel,
                 Oid foreigntableid);
```

Crée les chemins d'accès possibles pour un parcours sur une table distante. Cette fonction est appelée lors de la planification de la requête. Les paramètres sont identiques à ceux de `GetForeignRelSize`, qui a déjà été appelée.

Cette fonction doit générer au moins un chemin d'accès (nœud `ForeignPath`) pour un parcours sur une table distante et doit appeler `add_path` pour ajouter chaque chemin à `baserel->pathlist`. Il est recommandé d'utiliser `create_foreignscan_path` pour construire les nœuds `ForeignPath`. La fonction peut générer plusieurs chemins d'accès, c'est-à-dire un chemin

qui a un champ `pathkeys` valide pour représenter un résultat pré-trié. Chaque chemin d'accès doit contenir les estimations de coûts et peut contenir toute information privée au FDW qui est nécessaire pour identifier la méthode attendue du parcours spécifique.

Voir Section 59.4 pour plus d'informations.

```
ForeignScan *
GetForeignPlan (PlannerInfo *root,
                RelOptInfo *baserel,
                Oid foreigntableid,
                ForeignPath *best_path,
                List *tlist,
                List *scan_clauses,
                Plan *outer_plan);
```

Crée un nœud de plan `ForeignScan` à partir du chemin d'accès distant sélectionné. Cette fonction est appelé à la fin de la planification de la requête. Les paramètres sont identiques à ceux de la fonction `GetForeignRelSize`, avec en plus le `ForeignPath` sélectionné (précédemment produit par `GetForeignPaths`, `GetForeignJoinPaths` ou `GetForeignUpperPaths`), la liste cible à émettre par le nœud du plan, les clauses de restriction forcées par le nœud du plan, et le sous-plan externe de `ForeignScan`, utilisé pour les vérifications réalisées par `RecheckForeignScan`. (Si le chemin est pour une jointure plutôt qu'une relation de base, `foreigntableid` est `InvalidOid`.)

Cette fonction doit créer et renvoyer un nœud `ForeignScan`. Il est recommandé d'utiliser `make_foreignscan` pour construire le nœud `ForeignScan`.

Voir Section 59.4 pour plus d'informations.

59.2.2. Routines FDW pour optimiser le traitement après parcours/jointure

Si un FDW supporte l'exécution distante de jointure après parcours, comme une agrégation distante, il doit fournir cette fonction callback :

```
void
GetForeignUpperPaths(PlannerInfo *root,
                     UpperRelationKind stage,
                     RelOptInfo *input_rel,
                     RelOptInfo *output_rel,
                     void *extra);
```

Crée les chemins d'accès possibles pour le traitement *relation de niveau supérieur*, qui est le terme de l'optimiseur pour tout traitement après parcours/jointure, comme les agrégats, les fonctions de fenêtrage, le tri et les mises à jour de table. Cette fonction optionnelle est appelée lors de l'optimisation de la requête. Actuellement, elle est seulement appelée si toutes les relations de base impliquées appartiennent au même FDW. Cette fonction doit générer des chemins `ForeignPath` pour tout traitement post-parcours/jointure que le FDW sait réaliser à distance (utilisez `create_foreign_upper_path` pour les construire), et appeler `add_path` pour ajouter ces chemins à la relation indiquée du niveau supérieur. Tout comme `GetForeignJoinPaths`, il n'est pas nécessaire que cette fonction réussisse à créer des chemins, étant donnée qu'il est toujours possible d'utiliser des chemins de traitement local.

Le paramètre `stage` identifie l'étape post- parcours/jointure est en cours de considération. `output_rel` est la relation supérieure devant recevoir les chemins représentation le traitement

de cette étape, et `input_rel` est la relation représentant la source de cette étape. Le paramètre `extra` fournit des détails supplémentaires. Pour le moment, il est uniquement positionné pour `UPPERREL_PARTIAL_GROUP_AGG` ou `UPPERREL_GROUP_AGG`, auquel cas il pointe vers une structure `GroupPathExtraData` ; ou pour `UPPERREL_FINAL`, auquel cas il pointe vers une structure `FinalPathExtraData`. (Notez que les chemins `ForeignPath` ajoutés à `output_rel` n'auront typiquement pas de dépendances directes avec les chemins de `input_rel` car leur traitement se fait en externe. Néanmoins, examiner les chemins précédemment générés pour l'étape de traitement précédente peut se révéler utile pour éviter un travail redondant de planification.)

Voir Section 59.4 pour plus d'informations.

```
void
BeginForeignScan (ForeignScanState *node,
                  int eflags);
```

Commence l'exécution d'un parcours distant. L'appel se fait lors du démarrage de l'exécuteur. Cette fonction doit réaliser toutes les initialisation nécessaires avant le démarrage du parcours, mais ne doit pas commencer à exécuter le vrai parcours (cela se fera lors du premier appel à `IterateForeignScan`). Le nœud `ForeignScanState` est déjà créé mais son champ `fdw_state` vaut toujours NULL. Les informations sur la table à parcourir sont accessibles via le nœud `ForeignScanState` (en particulier à partir du nœud sous-jacent `ForeignScan` qui contient toute information privée au FDW fournie par `GetForeignPlan`). `eflags` contient les bits de drapeaux décrivant le mode opératoire de l'exécuteur pour ce nœud du plan.

Notez que quand `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` est vraie, cette fonction ne doit pas réaliser d'actions visibles en externe. Elle doit seulement faire le minimum requis pour que l'état du nœud soit valide pour `ExplainForeignScan` et `EndForeignScan`.

```
TupleTableSlot *
IterateForeignScan (ForeignScanState *node);
```

Récupère une ligne de la source distante, la renvoyant dans un emplacement de ligne de table (le champ `ScanTupleSlot` du nœud doit être utilisé dans ce but). Renvoie NULL s'il n'y a plus de lignes disponibles. L'infrastructure d'emplacement de ligne de table permet qu'une ligne physique ou virtuelle soit renvoyée. Dans la plupart des cas, la deuxième possibilité (virtuelle), est préférable d'un point de vue des performances. Notez que cette fonction est appelée dans un contexte mémoire dont la durée de vie est très courte et qui sera réinitialisé entre chaque appel. Créez un contexte mémoire dans `BeginForeignScan` si vous avez besoin d'un stockage qui tient plus longtemps ou utilisez le champ `es_query_cxt` de `EState`.

Les lignes renvoyées doivent correspondre à la liste cible `fdw_scan_tlist` si elle a été fournie, sinon elles doivent correspondre au type de ligne de la table distante parcourue. Si vous choisissez d'optimiser en récupérant d'avance des colonnes non nécessaires, vous devriez insérer des valeurs NULL dans les positions de ces colonnes, ou sinon générer une liste `fdw_scan_tlist` avec ces colonnes omises.

Notez que l'exécuteur de PostgreSQL ne se préoccupe pas de savoir si les lignes renvoyées violent les contraintes définies sur la table distante -- mais le planificateur s'en préoccupe, et peut optimiser les requêtes incorrectement si il y a des lignes visibles dans la table distante qui ne satisfont pas une contrainte déclarée. Si une contrainte est violée lorsque l'utilisateur a déclaré que la contrainte devrait être vrai, il peut être approprié de lever une erreur (de la même manière que vous devriez le faire dans le cas où les types de données ne correspondent pas).

```
void
```

```
ReScanForeignScan (ForeignScanState *node);
```

Recommence le parcours depuis le début. Notez que les paramètres dont dépent le parcours peuvent avoir changés de valeur, donc le nouveau parcours ne va pas forcément renvoyer les mêmes lignes.

```
void
EndForeignScan (ForeignScanState *node);
```

Termine le parcours et relâche les ressources. Il n'est habituellement pas nécessaire de relâcher la mémoire allouée via palloc. Par contre, les fichiers ouverts et les connexions aux serveurs distants doivent être nettoyés.

59.2.3. Routines des FDW pour le parcours des jointures distantes

Si un FDW permet d'effectuer des jointures distantes (autrement qu'en récupérant les données des deux tables et en faisant la jointure localement), il devrait fournir cette fonction callback :

```
void
GetForeignJoinPaths (PlannerInfo *root,
                     RelOptInfo *joinrel,
                     RelOptInfo *outerrel,
                     RelOptInfo *innerrel,
                     JoinType jointype,
                     JoinPathExtraData *extra);
```

Crée les chemins possibles d'accès pour une jointure de deux (ou plus) tables distantes qui toutes proviennent du même serveur distant. Cette fonction optionnelle est appelée durant la planification de la requête. De la même façon que `GetForeignPaths`, cette fonction devrait générer des chemins `ForeignPath` pour le paramètre `joinrel` fourni (utilisez `create_foreign_join_path` pour les construire), et appeler la fonction `add_path` pour ajouter ces chemins à l'ensemble des chemins à considérer pour la jointure. Mais contrairement à `GetForeignPaths`, il n'est pas nécessaire que cette fonction réussisse à créer au moins un chemin, dans la mesure où des chemins entraînant des jointures locales sont toujours possibles.

Notez que cette fonction sera invoquée de manière répétitive pour la même jointure, avec des combinaisons différentes de relations internes ou externes ; il est de la responsabilité du FDW de minimiser les tâches dupliquées.

Si un chemin `ForeignPath` est choisi pour la jointure, il représentera l'ensemble du processus de jointure ; les chemins générés pour les tables qui la composent et les jointures auxiliaires ne seront pas utilisés. Les traitements suivants des chemins de jointure procèdent essentiellement de la même manière que pour un chemin parcourant une simple table distante. Une différence est que le `scanrelid` résultant du nœud du plan `ForeignScan` devrait être mis à zéro, dans la mesure où il ne représente aucune relation simple ; à la place, le champ `fd_relids` du nœud `ForeignScan` représente l'ensemble des relations qui ont été jointes. (Le dernier champ est positionné automatiquement par le code interne du planificateur, et n'a pas besoin d'être rempli par le FDW.) Une autre différence est que, comme la liste des colonnes pour une jointure distante ne peut être trouvée dans les catalogues systèmes, le FDW doit remplir `fdw_scan_tlist` avec une liste appropriée de nœuds `TargetEntry`, représentant l'ensemble des colonnes qu'il fournira à l'exécution dans les lignes qu'il retournera.

Voir Section 59.4 pour des informations supplémentaires.

59.2.4. Routines FDW pour la mise à jour des tables distantes

Si un FDW supporte la modification des tables distantes, il doit fournir certaines ou toutes les fonctions callback suivant les besoins et les capacités du FDW :

```
void
AddForeignUpdateTargets (PlannerInfo *root,
                         Index rtindex,
                         RangeTblEntry *target_rte,
                         Relation target_relation);
```

Les opérations UPDATE et DELETE sont réalisées contre des lignes précédemment récupérées par des fonctions de parcours de table. Le FDW peut avoir besoin d'informations supplémentaires, comme l'identifiant de la ligne ou les valeurs des colonnes formant la clé primaire pour s'assurer qu'il peut identifier la ligne exacte à mettre à jour ou à supprimer. Pour supporter cela, cette fonction peut ajouter des colonnes cibles supplémentaires cachées à la liste des colonnes qui doivent être récupérées de la table distante pendant une opération UPDATE ou DELETE.

Pour faire cela, construisez une Var représentant une valeur supplémentaire dont vous avez besoin, et passez la à add_row_identity_var, avec un nom pour la colonne cachée. (Vous pouvez effectuer cela plusieurs fois, si plusieurs colonnes sont nécessaires.) Vous devez choisir une colonne cachée distincte pour chaque Var différente dont vous avez besoin, à l'exception des Var identiques, sauf pour le champ varno, qui peuvent, et doivent partager un même nom de colonne. Le cœur du système utilise les colonnes cachées nommées tableoid pour une colonne tableoid de table, ctid ou ctid*N* pour ctid, wholerow pour une Var de ligne entière marquée avec vartype = RECORD, et wholerow*N* pour une Var de ligne entière avec vartype égal au type de ligne déclaré de la table. Ré-utiliser ces noms quand vous le pouvez (le planificateur combinera les demandes dupliquées pour les colonnes cachées identiques). Si vous avez besoin d'autres type de colonnes cachées en dehors de celles-ci, il est préférable de choisir un nom préfixé avec le nom de votre extension pour éviter des conflits avec d'autres FDW.

Si le pointeur AddForeignUpdateTargets est initialisé à NULL, aucune expression cible supplémentaire ne sera ajoutée. (Ceci rend impossible l'implémentation des opérations DELETE bien que l'UPDATE est toujours faisable si le FDW se base sur une clé primaire ne changeant pas pour identifier les lignes.)

```
List *
PlanForeignModify (PlannerInfo *root,
                   ModifyTable *plan,
                   Index resultRelation,
                   int subplan_index);
```

Réalise toute opération supplémentaire de planification nécessaire pour une insertion, mise à jour ou suppression sur une table distante. Cette fonction génère l'information privée du FDW qui sera attachée au nœud du plan ModifyTable qui réalise la mise à jour. Cette information privée doit avoir la forme d'une List, et sera réalisée par BeginForeignModify lors de l'exécution.

root est l'information globale du planificateur sur la requête. plan est le nœud du plan ModifyTable qui est complet sauf pour le champ fdwPrivLists. resultRelation identifie la table distante cible par son index rangetable. subplan_index identifie la cible du nœud de plan ModifyTable en comptant à partir de zéro ; utilisez ceci si vous voulez indexer dans les sous-structures « per-target-relation » des nœuds du plan.

Voir Section 59.4 pour plus d'informations.

Si le pointeur `PlanForeignModify` est initialisé à `NULL`, aucune action supplémentaire n'est réalisée au moment de la planification, et la liste `fdw_private` renvoyée par `BeginForeignModify` vaudra NIL.

```
void
BeginForeignModify (ModifyTableState *mtstate,
                    ResultRelInfo *rinfo,
                    List *fdw_private,
                    int subplan_index,
                    int eflags);
```

Commence l'exécution d'une opération de modification de la table distante. Cette routine est appelée lors du démarrage de l'exécuteur. Elle doit réaliser toute initialisation nécessaire avant de procéder aux modifications de la table. En conséquence, `ExecForeignInsert/ExecForeignBatchInsert`, `ExecForeignUpdate` ou `ExecForeignDelete` seront appelées pour chaque ligne à insérer, mettre à jour ou supprimer.

`mtstate` est l'état général du nœud de plan `ModifyTable` en cours d'exécution ; les données globales sur le plan et l'état d'exécution sont disponibles via cette structure. `rinfo` est la structure `ResultRelInfo` décrivant la table distante cible. (Le champ `ri_FdwState` de `ResultRelInfo` est disponible pour que le FDW enregistre tout état privé dont il aurait besoin pour réaliser cette opération.) `fdw_private` contient les données privées générées par `PlanForeignModify`. `subplan_index` identifie la cible du nœud de plan `ModifyTable`. `eflags` contient les bits de drapeaux décrivant le mode opératoire de l'exécuteur pour ce nœud de plan.

Notez que quand `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` est vrai, cette fonction ne devrait réaliser aucune action visible externe ; il devrait seulement faire le minimum requis pour rendre l'état du nœud valide pour `ExplainForeignModify` et `EndForeignModify`.

Si le pointeur `BeginForeignModify` est initialisé à `NULL`, aucune action n'est prise lors du démarrage de l'exécuteur.

Notez que cette fonction est aussi appelée lors de l'insertion de lignes déplacées dans une partition de type table distante ou lors de l'exécution de `COPY FROM` sur une table distante, auquel cas elle est appelée d'une façon différente que dans le cas d'un `INSERT`. Voir les fonctions callback décrites ci-dessous permettant au FDW de le supporter. described below that allow the FDW to support that.

```
TupleTableSlot *
ExecForeignInsert (EState *estate,
                   ResultRelInfo *rinfo,
                   TupleTableSlot *slot,
                   TupleTableSlot *planSlot);
```

Insère une ligne dans la table distante. `estate` est un état global de l'exécution de la requête. `rinfo` est la structure `ResultRelInfo` décrivant la table distante cible. `slot` contient la ligne à insérer ; ça correspondra à la définition du type de la ligne de la table distante. `planSlot` contient la ligne qui a été générée par le sous-plan du nœud `ModifyTable` ; cela diffère du `slot` qui contient aussi les colonnes supplémentaires. (Le `planSlot` a typiquement peu d'intérêt pour `INSERT` mais est fourni pour être complet.)

La valeur de retour est soit un emplacement contenant les données effectivement insérées (elles peuvent différer des données fournies, par exemple suite à l'action de triggers), soit NULL si aucune ligne n'a été insérée (là aussi typiquement le résultat d'un trigger). Le `slot` peut être ré-utilisé dans ce contexte.

Les données du slot retourné ne sont utilisées que si la commande INSERT a une clause RETURNING ou implique une vue WITH CHECK OPTION ; ou si la table étrangère a un trigger AFTER ROW. Les triggers ont besoin de toutes les colonnes, mais le FDW peut choisir d'optimiser en ne retournant que certaines colonnes, voire aucune, selon la clause RETURNING ou les contraintes WITH CHECK OPTION. Cependant, un slot doit être retourné pour indiquer un succès, sinon le nombre de lignes retournées rapporté par la requête sera faux.

Si le pointeur ExecForeignInsert est initialisé à NULL, les tentatives d'insertion dans la table distante échoueront avec un message d'erreur.

Notez que cette fonction est aussi appelée à l'insertion de lignes reroutées vers une partition qui est une table étrangère, ou à l'exécution de COPY FROM sur une table étrangère, auxquels cas elle sera appelée d'une manière différente du cas d'un INSERT. Voir les fonctions callback décrites plus bas qui permettent au FDW de supporter cela.

```
TupleTableSlot **
ExecForeignBatchInsert(EState *estate,
                       ResultRelInfo *rinfo,
                       TupleTableSlot **slots,
                       TupleTableSlot **planSlots,
                       int *numSlots);
```

Insère plusieurs lignes en bloc dans la table distante. Les paramètres sont les même que pour ExecForeignInsert mis à part que slots et planSlots contiennent de plusieurs lignes et *numSlots spécifie le nombre de lignes dans ces tableaux.

La valeur de retour est un tableau d'emplacements (« slots ») contenant les données qui sont réellement insérées (et peut différer des données fournies, par exemple avec le résultat d'actions de trigger.) Les emplacements en entrée peuvent être réutilisés à ce propos. Le nombre de lignes insérées avec succès est retourné dans *numSlots.

Les données dans l'emplacement retour sont utilisés uniquement si l'expression INSERT implique une vue WITH CHECK OPTION ou si une table distance a un trigger AFTER ROW. Les triggers nécessitent toutes les colonnes, mais le FDW peut choisir, par optimisation distante, de retourner toutes ou quelques unes des colonnes, en fonction du contenu des contraintes WITH CHECK OPTION.

Si les pointeurs ExecForeignBatchInsert ou GetForeignModifyBatchSize sont mis à NULL, les tentatives d'insérer dans la table distante utiliseront ExecForeignInsert. Cette fonction n'est pas utilisée si l'expression INSERT dispose de la clause RETURNING.

Notez que cette fonction est aussi appelée lors de l'insertion de lignes dirigées dans une partition de table distante. Voir les fonctions callback décrites ci-dessous qui permettent au FDW de supporter cela.

```
int
GetForeignModifyBatchSize(ResultRelInfo *rinfo);
```

Affiche le nombre maximum de lignes qu'un simple appel à ExecForeignBatchInsert peut gérer pour la table distante spécifiée. L'exécuteur passe au plus le nombre donnée de lignes à ExecForeignBatchInsert. rinfo est la structure ResultRelInfo décrivant la cible de la table distante. Le FDW doit fournir le serveur distant et/ou l'option de table distante pour que l'utilisateur affecte cette valeur ou une valeur codée en dur.

Si le pointeur de ExecForeignBatchInsert ou GetForeignModifyBatchSize est mis à NULL, les tentatives d'insertion dans la table distante utiliseront ExecForeignInsert.

```
TupleTableSlot *
ExecForeignUpdate (EState *estate,
```

```
                    ResultRelInfo *rinfo,
                    TupleTableSlot *slot,
                    TupleTableSlot *planSlot);
```

Met à jour une ligne dans la table distante. `estate` est l'état global de l'exécution de la requête. `rinfo` est la structure `ResultRelInfo` décrivant la table distante cible. `slot` contient les nouvelles données de la ligne ; elles correspondront à la définition du type de ligne pour la table distante. `planSlot` contient la ligne qui a été générée par le sous-plan du nœud `ModifyTable`. Contrairement au `slot`, cette ligne contient seulement les nouvelles valeurs pour les colonnes modifiées dans la requête. De ce fait, ne vous appuyez pas sur les nombres d'attributs de la table distante à indexer dans `planSlot`. De plus, `planSlot` contient typiquement des colonnes « inutiles » (*junk*) supplémentaires. En particulier, toutes les colonnes inutiles demandées par `AddForeignUpdateTargets` seront disponibles pour ce slot.

La valeur de retour est soit un emplacement contenant la nouvelle ligne modifiée (elle peut différer des données fournies suite, par exemple, à l'exécution d'un trigger), ou NULL si aucune ligne n'a été réellement mise à jour (là-encore typiquement l'action d'un trigger). L'emplacement `slot` fourni peut être réutilisé dans ce contexte.

Les données du slot retourné ne sont utilisées que si la commande >`UPDATE` a une clause `RETURNING` ou implique une vue `WITH CHECK OPTION` ; ou si la table étrangère a un trigger `AFTER ROW`. Les triggers ont besoin de toutes les colonnes, mais le FDW peut choisir d'optimiser en ne retournant que certaines colonnes, voire aucune, selon la clause `RETURNING` ou les contraintes `WITH CHECK OPTION`. Cependant, un slot doit être retourné pour indiquer un succès, sinon le nombre de lignes retournées rapporté par la requête sera faux.

Si le pointeur `ExecForeignUpdate` est initialisé à `NULL`, les tentatives de mise à jour de la table distante échoueront avec un message d'erreur.

```
TupleTableSlot *
ExecForeignDelete (EState *estate,
                   ResultRelInfo *rinfo,
                   TupleTableSlot *slot,
                   TupleTableSlot *planSlot);
```

Supprime une ligne de la table distante. `estate` est l'état global de l'exécution de la requête. `rinfo` est la structure `ResultRelInfo` décrivant la table distante cible. `slot` ne contient rien d'utile à l'appel de la fonction mais peut être utilisé pour contenir la ligne renvoyée. `planSlot` contient la ligne générée par le sous-plan du nœud du plan `ModifyTable` ; en particulier, elle contient toute colonne supplémentaire réclamée par `AddForeignUpdateTargets`. Les colonnes supplémentaires doivent être utilisées pour identifier la ligne à supprimer.

La valeur de retour est soit un slot contenant la ligne supprimée soit NULL si aucune ligne n'a été supprimée (par exemple suite a déclenchement d'un trigger). Le `slot` fourni en référence peut être utilisé pour contenir la ligne à renvoyer.

Les données placées dans l'emplacement sont utilisées seulement si la requête `DELETE` dispose de la clause `RETURNING` ou si la table externe a un trigger `AFTER ROW`. Les triggers nécessitent toutes les colonnes mais le FDW pourrait choisir d'optimiser en ne renvoyant que certaines colonnes ou toutes suivant le contenu de la clause `RETURNING`. Néanmoins, un slot doit être renvoyé pour indiquer le succès. Dans le cas contraire, le nombre de lignes rapporté par la requête sera faux.

Si le pointeur `ExecForeignDelete` est initialisé à `NULL`, les tentatives de suppression dans la table distante échoueront avec un message d'erreur.

```
void
```

```
EndForeignModify (EState *estate,
                  ResultRelInfo *rinfo);
```

Termine la mise à jour et libère les ressources. Il n'est normalement pas importante de libérer la mémoire prise avec palloc mais, par exemple, les fichiers ouverts et les connexions vers des serveurs distants doivent être nettoyés.

Si le pointeur vers `EndForeignModify` est initialisé à `NULL`, aucune action n'a lieu pendant l'arrêt de l'exécuteur.

Les lignes insérées dans une table partitionnée à l'aide d'`INSERT` ou `COPY FROM` sont redirigées vers les partitions. Si un FDW supporte la redirection des lignes pour les partitions déclarées comme table distantes, il devra également fournir les fonctions de callback suivantes. Ces fonctions sont également appelées quand `COPY FROM` est exécuté sur une table distante.

```
void
BeginForeignInsert(ModifyTableState *mtstate,
                   ResultRelInfo *rinfo);
```

Débute l'exécution d'une opération d'insertion sur une table distante. Cette routine est appelée juste avant que la première ligne soit insérée dans la table distante quand il s'agit de la partition choisie par la redirection de ligne ou quand il s'agit de la cible spécifiée dans une commande `COPY FROM`. Elle devrait effectuer toute initialisation nécessaire avant l'insertion elle-même. Ensuite, `ExecForeignInsert` ou `ExecForeignBatchInsert` seront appelées pour chaque ligne devant être insérée dans la table distante.

`mtstate` est l'état général du nœud de plan `ModifyTable` en cours d'exécution ;; les données globales sur le plan et l'état d'exécution sont disponibles via cette structure. `rinfo` est la structure `ResultRelInfo` décrivant la table distante cible. (Le champ `ri_FdwState` de `ResultRelInfo` est disponible pour que le FDW enregistre tout état privé dont il aurait besoin pour réaliser cette opération.)

Quand elle est appelée par une commande `COPY FROM`, les données globales liées au plan contenues dans `mtstate` ne sont pas fournies et le paramètre `planSlot` de `ExecForeignInsert/ExecForeignBatchInsert` appelée par la suite pour chaque ligne insérée vaut `NULL`, que la table distante soit la partition choisie par la redirection de ligne ou que cela soit la cible spécifiée dans la commande.

Si le pointeur `BeginForeignInsert` est initialisé à `NULL`, aucune action n'est faite pour l'initialisation.

Notez que si le FDW ne supporte pas les tables étrangères comme partitions routables et/ou l'exécution de `COPY FROM` sur des tables étrangères, cette fonction, ou `ExecForeignInsert` appelée après, doivent renvoyer une erreur si nécessaire.

```
void
EndForeignInsert(EState *estate,
                 ResultRelInfo *rinfo);
```

Termine l'opération d'insertion et libère les ressources. Il n'est habituellement pas nécessaire de libérer la mémoire allouée via palloc. Par contre, les fichiers ouverts et les connexions aux serveurs distants doivent être libérés par exemple.

Si le pointeur `EndForeignInsert` est initialisé à `NULL`, aucune action n'est faite pour la fin de l'opération d'insertion.

```
int
IsForeignRelUpdatable (Relation rel);
```

Indique les opérations de mise à jour supportées par la table distante indiquée. La valeur de retour doit être un masque de bits correspondant aux numéros d'événement des règles, indiquant les opérations supportées par la table disante, en utilisant l'énumération `CmdType`. Autrement dit `(1 << CMD_UPDATE) = 4` pour `UPDATE`, `(1 << CMD_INSERT) = 8` pour `INSERT` et `(1 << CMD_DELETE) = 16` pour `DELETE`.

Si le pointeur `IsForeignRelUpdatable` est configuré à `NULL`, les tables distantes sont supposées accepter les INSERT, UPDATE et DELETE si le connecteur FDW fournit respectivement les fonctions `ExecForeignInsert`, `ExecForeignUpdate` et `ExecForeignDelete`. Cette fonction est uniquement nécessaire si le FDW supporte quelques tables modifiables et d'autres qui ne le sont pas. (Et même là, il est possible de renvoyer une erreur dans la routine d'exécution au lieu de vérifier avec cette fonction. Néanmoins, cette fonction est utilisé pour déterminer l'état modifiable des tables qui sera affiché dans les vues `information_schema`.)

Certaines insertions, mises à jour et suppressions vers des tables distantes peuvent être optimisées en implémentant un ensemble alternatif d'interfaces. Les interfaces habituelles pour les insertions, mises à jour et suppressions récupèrent les lignes du serveur distant, puis modifient les lignes, une par une. Dans certains cas, cette approche ligne par ligne est nécessaire mais elle peut s'avérer inefficace. S'il est possible pour le serveur distant de déterminer les lignes à modifier sans avoir à les récupérer, et qu'il n'y a pas de structures locales qui pourraient affecter l'opération (triggers locaux au niveau ligne, colonnes calculées automatiquement ou contraintes `WITH CHECK OPTION` provenant des vues parents), alors il est possible de s'arranger pour que l'opération entière soit réalisée sur le serveur distant. Les interfaces décrites ci-dessous rendent cela possible.

```
bool
PlanDirectModify (PlannerInfo *root,
                  ModifyTable *plan,
                  Index resultRelation,
                  int subplan_index);
```

Décide si l'exécution d'une modification directement sur le serveur distant est sûre. Dans ce cas, renvoie `true` après avoir réalisé les actions d'optimisation nécessaire pour cela. Dans le cas contraire, renvoie `false`. Cette fonction optionnelle est appelée lors de la planification de la requête. Si cette fonction réussit, `BeginDirectModify`, `IterateDirectModify` et `EndDirectModify` seront appelées à l'étape d'exécution. Dans le cas contraire, la modification de la table sera exécutée en utilisant les fonctions de modification de la table décrites ci-dessus. Les paramètres sont les mêmes que pour `PlanForeignModify`.

Pour exécuter la modification directe sur le serveur distant, cette fonction doit ré-écrire le sous-plan cible avec un nœud de plan `ForeignScan` qui exécute la modification directe sur le serveur distant. Les champs `operation` et `resultRelation` du `ForeignScan` doivent être configurés de façon approprié. `operation` doit être affecté à la valeur de l'énumération `CmdType` correspondante au type d'expression (et qui est `CMD_UPDATE` pour `UPDATE`, `CMD_INSERT` pour `INSERT` et `CMD_DELETE` pour `DELETE`), et l'argument `resultRelation` doit être copié dans le champ `resultRelation`.

Voir Section 59.4 pour plus d'informations.

Si le pointeur `PlanDirectModify` est configuré à `NULL`, aucune tentative ne sera réalisée pour exécuter une modification directe sur le serveur distant.

```
void
```

```
BeginDirectModify (ForeignScanState *node,
                   int eflags);
```

Prépare une exécution d'une modification directe sur le serveur distant. Cette fonction est appelée lors du démarrage de l'exécuteur. Elle doit réaliser toute initialisation nécessaire avant la modification directe, qui doit être réalisée lors du premier appel à `IterateDirectModify`). Le nœud `ForeignScanState` a déjà été créée mais son champ `fdw_state` vaut toujours NULL. Des informations sur la table à modifier sont disponibles au travers du nœud `ForeignScanState` (en particulier, à partir du nœud `ForeignScan` sous-jacent, qui contient des informations privées au FDW fournies par `PlanDirectModify`). `eflags` contient des bits d'informations décrivant le mode d'opération de l'exécuteur pour ce nœud de plan.

Notez que quand `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` est vrai, cette fonction ne doit pas réaliser d'actions visibles extérieurement ; elle doit seulement faire le minimum requis pour rendre valide l'état du nœud pour `ExplainDirectModify` et `EndDirectModify`.

Si le pointeur `BeginDirectModify` est configuré à `NULL`, aucune tentative ne sera réalisée pour exécuter une modification directe sur le serveur distant.

```
TupleTableSlot *
IterateDirectModify (ForeignScanState *node);
```

Quand la requête `INSERT`, `UPDATE` ou `DELETE` ne contient pas de clause `RETURNING`, renvoie simplement NULL après une modification directe sur le serveur distant. Quand la requête contient cette clause, récupère un résultat contenant la donnée nécessaire pour le traitement du `RETURNING`, le renvoyant dans un slot de ligne de table (le champ `ScanTupleSlot` du nœud doit être utilisé pour cela). Les données insérées, mises à jour ou supprimées doivent être enregistrées dans le champ `node->resultRelInfo->ri_projectReturning->pi_exprContext->ecxt_scantuple`. Renvoie NULL s'il n'y a plus de lignes disponibles. Notez que cette fonction est appelée dans un contexte mémoire à court terme qui sera réinitialisée à chaque appel. Créez un contexte mémoire dans `BeginDirectModify` si vous avez besoin d'un stockage d'une durée de vie plus importante ou utilisez `es_query_cxt` du champ `EState` du nœud.

Les lignes renvoyées doivent correspondre à la liste cible `fdw_scan_tlist` si une liste a été fournie. Sinon, elles doivent correspondre au type de ligne de la table externe en cours de modification. Si vous choisissez d'optimiser la récupération des colonnes inutiles pour le traitement de `RETURNING`, vous devez placer des valeurs NULL à la position de ces colonnes ou générer une liste `fdw_scan_tlist` en omettant les colonnes inutiles.

Que la requête ait la clause ou non, le nombre de lignes rapporté par la requête doit être incrémenté par le FDW lui-même. Quand la requête n'a pas de clause, le FDW doit aussi incrémenté le nombre de lignes pour le nœud `ForeignScanState` dans le cas d'un `EXPLAIN ANALYZE` case.

Si le pointeur `IterateDirectModify` est configuré à `NULL`, aucune tentative ne sera réalisée pour exécuter une modification directe sur le serveur distant.

```
void
EndDirectModify (ForeignScanState *node);
```

Nettoie après une modification directe sur le serveur distant. Il n'est normalement pas important de relâcher la mémoire allouée avec palloc mais, par exemple, des fichiers et des connexions ouvertes sur le serveur distant doivent fermés.

Si le pointeur `EndDirectModify` est configuré à `NULL`, aucune tentative ne sera réalisée pour exécuter une modification directe sur le serveur distant.

59.2.5. Routines FDW pour TRUNCATE

```
void
ExecForeignTruncate(List *rels,
                    DropBehavior behavior,
                    bool restart_seqs);
```

Tronque les tables distantes. Cette fonction est appelée quand TRUNCATE est exécuté sur une table distante. `rels` est une liste de structure de données `Relation` des table distantes à tronquer.

Le comportement est soit `DROP_RESTRICT`, soit `DROP_CASCADE` indiquant que l'option `RESTRICT` ou `CASCADE`, respectivement, a été indiquée dans la commande `TRUNCATE` originelle.

Si `restart_seqs` vaut `true`, la commande `TRUNCATE` originelle a requis le comportement `RESTART IDENTITY`, sinon le comportement `CONTINUE IDENTITY` a été requis.

Notez que les options `ONLY` spécifiées dans la commande `TRUNCATE` originelle ne sont pas passées à `ExecForeignTruncate`. Ce comportement est similaire aux fonctions callback des `SELECT`, `UPDATE` et `DELETE` sur une table distante.

`ExecForeignTruncate` est invoquée une fois par serveur distant pour lequel des tables distantes sont tronquées. Cela signifie que toutes les tables distantes incluses dans `rels` doivent appartenir au même serveur.

Si le pointeur de `ExecForeignTruncate` est mis à `NULL`, les tentatives de tronquer des tables distantes échoueront avec un message d'erreur.

59.2.6. Routines FDW pour le verrouillage des lignes

Si un FDW veut supporter le *verrouillage tardif de lignes* (comme décrit à Section 59.5), il doit fournir les fonctions callbacks suivantes :

```
RowMarkType
GetForeignRowMarkType (RangeTblEntry *rte,
                       LockClauseStrength strength);
```

Indique quelle option de marquage de ligne utiliser pour une table distante. `rte` est le nœud `RangeTblEntry` pour la table et `strength` décrit la force du verrou requis par la clause `FOR UPDATE/SHARE`, si applicable. Le résultat doit être un membre du type énumération `RowMarkType`.

Cette fonction est appelée durant la planification de la requête pour chaque table distante qui apparaît dans une requête `UPDATE`, `DELETE`, ou `SELECT FOR UPDATE/SHARE` et n'est pas la cible d'une commande `UPDATE` ou `DELETE`.

Si le pointeur de fonction `GetForeignRowMarkType` est positionné à `NULL`, l'option `ROW_MARK_COPY` est toujours utilisée. (Ceci implique que la fonction `RefetchForeignRow` ne sera jamais appelée, aussi elle n'a pas besoin d'être fournie non plus.)

Voir Section 59.5 pour plus d'informations.

```
void
RefetchForeignRow (EState *estate,
                   ExecRowMark *erm,
                   Datum rowid,
                   TupleTableSlot *slot,
```

```
                                bool *updated);
```

Récupère à nouveau une ligne à partir de la table distante, après l'avoir verrouillée si nécessaire. `estate` est l'état global d'exécution de la requête. `erm` est la structure `ExecRowMark` décrivant la table distante cible et le type de verrou ligne (si applicable) à prendre. `rowid` identifie la ligne à récupérer. `slot` ne contient rien d'utile lors de l'appel, mais peut être utilisé pour retourner l'enregistrement renvoyé. `updated` est un paramètre de sortie.

Cette fonction doit renvoyer l'enregistrement dans le slot fourni à cet effet, ou l'effacer si le verrou sur la ligne ne peut être obtenu. Le type de verrou de ligne à acquérir est défini par `erm->markType`, qui est la valeur retournée précédemment par la fonction `GetForeignRowMarkType`. (`ROW_MARK_REFERENCE` signifie qu'il faut juste re-récupérer l'enregistrement sans poser aucun verrou, et `ROW_MARK_COPY` ne sera jamais envoyé à cette fonction.)

En complément, `*updated` devrait être positionné à `true` si ce qui a été récupéré est une version mise à jour de la ligne plutôt que la même version obtenue précédemment. (Si le FDW ne peut être sûr à propos de cette information, retourner toujours `true` est recommandé.)

Notez que par défaut, l'échec pour prendre un verrou ligne devrait avoir pour conséquence de lever une erreur ; renvoyer un emplacement vide n'est approprié que si l'option `SKIP LOCKED` est spécifiée par `erm->waitPolicy`.

`rowid` est la valeur de `ctid` précédemment lue pour la ligne récupérée à nouveau. Bien que la valeur `rowid` est envoyée comme type `Datnum`, elle ne peut être actuellement que de type `tid`. L'API de la fonction est choisie dans l'espoir qu'il sera possible d'autoriser d'autre types de données pour les identifiants des lignes dans le futur.

Si le pointeur de fonction `RefetchForeignRow` est positionné sur `NULL`, les tentatives de récupération à nouveau des lignes échoueront avec un message d'erreur.

Voir Section 59.5 pour plus d'informations.

```
bool
RecheckForeignScan (ForeignScanState *node, TupleTableSlot *slot);
```

Vérifie à nouveau qu'une ligne retournée précédemment correspond toujours au parcours et aux qualificatifs de jointures, et éventuellement fournit une version modifiée de la ligne. Pour les wrappers de données distantes qui ne supportent pas les jointures (*join push-down*), il sera plus pratique de positionner ce pointeur de fonction à `NULL` et, à la place, configurer `fdw_recheck_quals` de manière appropriée. Cependant lorsque des jointures externes sont poussées au serveur distant, il n'est pas suffisant d'appliquer à nouveau les vérifications applicables à toutes les tables de base à la ligne résultat, même si tous les attributs nécessaires sont présents, parce que l'impossibilité de mettre en correspondance certains qualificatifs pourrait résulter en la mise à NULL de certaines colonnes, plutôt qu'aucune ligne ne soit retournée. `RecheckForeignScan` peut vérifier à nouveau les qualificatifs et renvoyer true si ils sont toujours satisfaits et false dans le cas contraire, mais elle peut aussi stocker une ligne de remplacement dans l'emplacement fourni.

Pour implémenter le support des jointures, un wrapper de données distantes construira typiquement un plan alternatif local qui est utilisé uniquement pour les revérifications ; celui-ci deviendra le sous-plan externe de `ForeignScan`. Lorsqu'une revérification est requise, ce sous-plan peut être exécuté et la ligne résultante peut être stockée dans l'emplacement. Ce plan n'a pas besoin d'être efficace car aucune table de base ne retournera plus d'une ligne ; par exemple, il peut réaliser toutes les jointures comme des boucles imbriquées. La fonction `GetExistingLocalJoinPath` peut être utilisée pour rechercher des chemins existants dans un chemin de jointure local convenable, qui est utilisable comme plan de jointure local alternatif. `GetExistingLocalJoinPath` recherche un chemin sans paramètre dans la liste de chemins de la relation de jointure spécifiée (si un tel chemin n'existe pas, elle renvoie NULL, ce qui fait que le FDW pourrait construire un chemin local lui-même ou pourrait choisir de ne pas créer de chemins d'accès pour cette jointure).

59.2.7. Routines FDW pour EXPLAIN

```
void
ExplainForeignScan (ForeignScanState *node,
                    ExplainState *es);
```

Affiche une sortie EXPLAIN supplémentaire pour un parcours de table distante. Cette fonction peut faire appel à ExplainPropertyText et aux fonctions relatives pour ajouter des champs à la sortie d'EXPLAIN. Les champs drapeaux dans es peuvent être utilisés pour déterminer ce qui doit être affiché, et l'état du nœud ForeignScanState peut être inspecté pour fournir des statistiques d'exécution dans le cas du EXPLAIN ANALYZE.

Si le pointeur ExplainForeignScan vaut NULL, aucune information supplémentaire n'est affichée lors de l'EXPLAIN.

```
void
ExplainForeignModify (ModifyTableState *mtstate,
                      ResultRelInfo *rinfo,
                      List *fdw_private,
                      int subplan_index,
                      struct ExplainState *es);
```

Affiche une sortie supplémentaire pour EXPLAIN lors de la mise à jour d'une table distante. Cette fonction peut appeler ExplainPropertyText et les fonctions en relation pour ajouter des champs à la sortie d'EXPLAIN. Les champs drapeaux de es peuvent être utilisés pour déterminer quoi afficher, et l'état du nœud ModifyTableState peut être inspecté pour fournir des statistiques en exécution dans le cas du EXPLAIN ANALYZE. Les quatre premiers arguments sont les mêmes que pour BeginForeignModify.

Si le pointeur ExplainForeignModify vaut NULL, aucune information supplémentaire n'est affichée lors de l'EXPLAIN.

```
void
ExplainDirectModify (ForeignScanState *node,
                     ExplainState *es);
```

Affiche une sortie EXPLAIN supplémentaire pour une modification directe sur le serveur distant. Cette fonction peut appeler ExplainPropertyText et les fonctions relatives pour ajouter des champs à la sortie d'EXPLAIN. Les champs flag dans es peuvent être utilisés pour déterminer ce qui doit être affiché, et l'état du nœud ForeignScanState peut être inspecté pour fournir des statistiques à l'exécution dans le cas d'un EXPLAIN ANALYZE.

Si le pointeur ExplainDirectModify est configuré à NULL, aucune information supplémentaire n'est affichée pendant un EXPLAIN.

59.2.8. Routines FDW pour ANALYZE

```
bool
AnalyzeForeignTable (Relation relation,
                     AcquireSampleRowsFunc *func,
                     BlockNumber *totalpages);
```

Cette fonction est appelée quand ANALYZE est exécuté sur une table distante. Si le wrapper de données distantes peut récupérer des statistiques pour cette table distante, il doit renvoyer `true`, et fournir un pointeur vers une fonction qui récupérera un échantillon de lignes à partir de la table dans *`func`*, ainsi que la taille estimée de la table en blocs dans *`totalpages`*. Sinon, il doit renvoyer `false`.

Si le wrapper de données distantes ne supporte pas la récupération de statistiques quelque soit la table, le pointeur `AnalyzeForeignTable` doit être configuré à `NULL`.

Si fourni, la fonction de récupération de l'échantillon doit avoir la signature suivante :

```
int
AcquireSampleRowsFunc(Relation relation,
                      int elevel,
                      HeapTuple *rows,
                      int targrows,
                      double *totalrows,
                      double *totaldeadrows);
```

Un échantillon récupéré au hasard et comprenant au plus *`targrows`* lignes doit être récupéré à partir de la table et stocké dans le tableau *`rows`* fourni par l'appelant. Le nombre réel de lignes récupérées doit être renvoyé. De plus, les estimations du nombre total de lignes vivantes et mortes doivent être enregistrées dans les paramètres en sortie appelés *`totalrows`* et *`totaldeadrows`*. (Configurez *`totaldeadrows`* à zéro si le wrapper de données distantes ne connaît pas le concept des lignes mortes.)

59.2.9. Routines FDW pour `IMPORT FOREIGN SCHEMA`

```
List *
ImportForeignSchema (ImportForeignSchemaStmt *stmt, Oid serverOid);
```

Obtient une liste des commandes de création de tables distantes. Cette fonction est appelée lors de l'exécution de IMPORT FOREIGN SCHEMA, et il lui est passé l'arbre d'analyse pour cette instruction, ainsi que l'OID du serveur distant à utiliser. Elle devrait renvoyer une liste de chaînes C, chacune d'entre elles devant contenir une commande CREATE FOREIGN TABLE. Ces chaînes seront analysées et exécutées par le serveur principal.

À l'intérieur de la structure `ImportForeignSchemaStmt`, `remote_schema` est le nom du schéma distant à partir duquel les tables sont à importer. `list_type` indique comment filtrer les noms de tables : `FDW_IMPORT_SCHEMA_ALL` signifie que toutes les tables dans le schéma distant devraient être importées (dans ce cas, `table_list` est vide), `FDW_IMPORT_SCHEMA_LIMIT_TO` signifie d'inclure seulement les tables listées dans `table_list`, et `FDW_IMPORT_SCHEMA_EXCEPT` signifie d'exclure les tables listées dans `table_list`. `options` est une liste d'options utilisées pour le processus d'import. La signification des options relève du FDW. Par exemple, un FDW pourrait utiliser une option pour définir si les attributs `NOT NULL` des colonnes devraient être importés. Ces options n'ont pas besoin d'avoir une quelconque relation avec celles supportées par le FDW pour les objets base de données.

Le FDW peut ignorer le champ `local_schema` de `ImportForeignSchemaStmt`, parce que le serveur principal insérera automatiquement ce nom dans les commandes `CREATE FOREIGN TABLE` analysées.

Le FDW n'a pas besoin de mettre en place lui-même le filtrage spécifié par `list_type` et `table_list`, dans la mesure où le serveur principal ignorera automatiquement les commandes renvoyées pour les tables exclues selon ces options. Cependant, il est souvent utile d'éviter

le travail de création des commandes pour les tables exclues dès le départ. La fonction `IsImportableForeignTable()` peut être utile pour tester si une table distante donnée passera ou pas le filtre.

Si le FDW ne supporte pas l'import de définition de tables, le pointeur de fonction `ImportForeignSchema` peut être positionné à `NULL`.

59.2.10. Routines FDW pour une exécution parallélisée

Un nœud `ForeignScan` peut, en option, supporter une exécution parallélisée. Un `ForeignScan` parallélisée sera exécutée par plusieurs processus et devra renvoyer chaque ligne une fois seulement au travers de tous les processus coopérant. Pour faire cela, les processus peuvent se coordonner avec des ensembles de taille fixe de mémoire partagée dynamique. Cette mémoire partagée n'est pas garantie d'être placée à la même adresse pour chaque processus, donc il ne doit pas contenir de pointeurs. Les fonctions suivantes sont toutes optionnelles généralement, mais elles sont requises si une exécution parallèle doit être supportée.

```
bool
IsForeignScanParallelSafe(PlannerInfo *root, RelOptInfo *rel,
                          RangeTblEntry *rte);
```

Teste si un parcours peut être réalisé avec un processus parallèle. Cette fonction sera seulement appelée quand le planificateur pense qu'un plan parallélisé est possible, et doit renvoyer true si un tel plan est sûr pour ce parcours. Ceci ne sera généralement pas le cas si la source de données distante a des sémantiques transactionnelles, sauf si la connexion du processus supplémentaire peut être en quelque sorte partagée dans le même contexte transactionnelle que celui du processus maître

Si ce callback n'est pas défini, il est supposé que le parcours doit avoir lieu au niveau du processus maître. Notez que renvoyer true ne signifie pas que le parcours sera parallélisé. Cela signifie seulement qu'il est possible de l'effectuer avec des processus parallèles. De ce fait, il peut être utile de définir cette méthode même quand l'exécution parallélisée n'est pas supportée.

```
Size
EstimateDSMForeignScan(ForeignScanState *node, ParallelContext
 *pcxt);
```

Estime la quantité de mémoire partagée dynamique requis pour une opération parallélisée. Cette valeur pourrait être supérieure à la quantité réellement utilisée mais elle ne peut pas être inférieure. La valeur renvoyée est en octets. Cette fonction est optionnelle et peut être omise si elle n'est pas nécessaire. Mais si elle est omise, les trois fonctions suivantes peuvent elles-aussi être omises parce qu'aucune mémoire partagée ne sera allouée pour une utilisation avec le FDW.

```
void
InitializeDSMForeignScan(ForeignScanState *node, ParallelContext
 *pcxt,
                         void *coordinate);
```

Initialise la mémoire partagée dynamique qui sera requise pour une opération parallélisée ; `coordinate` pointe vers une partie de mémoire partagée de même taille que la valeur de retour de `EstimateDSMForeignScan`. Cette fonction est optionnelle et peut être omise si nécessaire.

```
void
```

```
ReInitializeDSMForeignScan(ForeignScanState *node, ParallelContext
 *pcxt,
                           void *coordinate);
```

Ré-initialise la mémoire partagée dynamique requise pour les opérations parallélisées quand le nœud du plan pour le parcours distant va être ré-exécuté. Cette fonction est optionnelle et peut être omise si nécessaire. La pratique recommandée est que cette fonction réinitialise seulement l'état partagé alors que la fonction `ReScanForeignScan` réinitialise seulement l'état local. Actuellement, cette fonction sera appelée avant `ReScanForeignScan` mais il est préférable de ne pas se baser sur cet ordre.

```
void
InitializeWorkerForeignScan(ForeignScanState *node, shm_toc *toc,
                            void *coordinate);
```

Initialise un état local d'un processus parallèle suivant l'état partagé configuré dans le processus maître par `InitializeDSMForeignScan`. Cette fonction est optionnelle et peut être omise si nécessaire.

```
void
ShutdownForeignScan(ForeignScanState *node);
```

Libères les ressources quand il anticipé que le nœud ne sera pas exécuté entièrement. Cette fonction ne sera pas appelée dans tous les cas; parfois, `EndForeignScan` peut être appelée sans que cette fonction ait été appelée avant. Puisque le segment DSM utilisé par les requêtes parallèles est détruit juste après que ce callback soit appelé, les wrappers de données distantes qui désirent effectuer des actions avant que le segment DSM disparaissent devraient implémenter cette méthode.

59.2.11. Routines FDW pour les exécutions asynchrones

Un nœud `ForeignScan` peut, optionellement, supporter une exécution asynchrone tel que décrit dans `src/backend/executor/README`. Les fonctions suivantes sont toutes optionnelles, mais sont toutes requises si l'exécution asynchrone doit être supportée.

```
bool
IsForeignPathAsyncCapable(ForeignPath *path);
```

Vérifie si un chemin `ForeignPath` donné peut parcourir une relation distante sous-jacente asynchrone. Cette fonction ne sera appelée uniquement à la fin d'une planification de requête quand le chemin donné est un enfant direct du chemin `AppendPath` et quand le planificateur estime que l'exécution asynchrone améliore les performances, et devra retourner vrai (« true ») si un chemin donné est capable de parcourir la relation distante de façon asynchrone.

Si cette fonction n'est pas définie, il est assumé que le chemin donné parcourt la relation distante en utilisant `IterateForeignScan`.(Ceci implique que les fonctions callbacks décrites ci-dessous ne seront jamais appelées, et n'ont pas besoin d'être fournies.)

```
void
ForeignAsyncRequest(AsyncRequest *areq);
```

Produit une ligne de manière asynchrone à partir du nœud `ForeignScan`. `areq` est la structure `AsyncRequest` décrivant le nœud `ForeignScan` et le nœud parent `Append` requis par les lignes

qui en viennent. Cette fonction doit conserver la ligne dans un slot spécifié par `areq->result`, et défini avec `areq->request_complete` mis à `true` ; ou, s'il est nécessaire d'attendre un événement extérieur au cœur du serveur, tel que des I/O réseau, et qu'une ligne ne peut être générée immédiatement, le drapeau doit être mis à `false`, et défini avec `areq->callback_pending` mis à `true` pour que le nœud `ForeignScan` puisse recevoir un callback depuis les fonctions callbacks décrites ci-dessous. S'il n'y a plus de lignes diponibles, mettre le slot à NULL ou fournir un slot vide, et mettre le drapeau `areq->request_complete` à `true`. Il est recommandé d'utiliser `ExecAsyncRequestDone` ou `ExecAsyncRequestPending` pour affecter les paramètres de sortie dans `areq`.

```
void
ForeignAsyncConfigureWait(AsyncRequest *areq);
```

Configure un événement de descripteur de fichier pour lequel le nœud `ForeignScan` souhaite attendre. Cette fonction sera appelée uniquement quand le nœud `ForeignScan` a le drapeau `areq->callback_pending` affecté, et devra ajouter l'événement au `as_eventset` du nœud parent `Append` décrit par `areq`. Voir les commentaires pour `ExecAsyncConfigureWait` dans `src/backend/executor/execAsync.c` pour des informations additionnelles. Quand l'événement de descripteur de fichier a lieu, `ForeignAsyncNotify` sera appelée.

```
void
ForeignAsyncNotify(AsyncRequest *areq);
```

Traite un événement pertinent qui s'est déroulé, puis produit une ligne de façon asynchrone à partir du nœud `ForeignScan`. Cette fonction devra affecter les paramètres de sortie dans `areq` de la même façon que `ForeignAsyncRequest`.

59.2.12. Routines FDW pour la reparamétrisation des chemins

```
List *
ReparameterizeForeignPathByChild(PlannerInfo *root, List
 *fdw_private,
                                 RelOptInfo *child_rel);
```

Cette fonction est appelée lors de la conversion d'un chemin paramétré par le plus haut parent de la relation enfant `child_rel` spécifiée devant être paramétrée avec la relation enfant. Cette fonction est utilisée pour reparamétrer n'importe quel chemin ou traduire n'importe quel nœud d'expression enregistré dans le membre `fdw_private` du `ForeignPath` spécifié. Le callback peut utiliser `reparameterize_path_by_child`, `adjust_appendrel_attrs` ou `adjust_appendrel_attrs_multilevel` selon son besoin.

59.3. Fonctions d'aide pour les wrapper de données distantes

Plusieurs fonctions d'aide sont exportées à partir du cœur du serveur, pour que les auteurs de wrappers de données distantes puissent accéder facilement aux attributs des objets en relation avec les wrappers, comme par exemple les options d'un wrapper. Pour utiliser une de ces fonctions, vous avez besoin d'inclure le fichier en-tête `foreign/foreign.h` dans votre fichier source. Cet en-tête définit aussi les types de structures qui sont renvoyés par ces fonctions.

```
ForeignDataWrapper *
GetForeignDataWrapperExtended(Oid fdwid, bits16 flags);
```

Cette fonction renvoie un objet `ForeignDataWrapper` pour le FDW avec l'OID précisé. Un objet `ForeignDataWrapper` contient les propriétés du FDW (voir `foreign/foreign.h` pour les détails). `flags` est un masque de bit OU indiquant des options supplémentaires. Il peut prendre la valeur `FDW_MISSING_OK`, auquel cas un `NULL` est retourné comme résultat à l'appelant au lieu d'une erreur d'objet indéfini.

```
ForeignDataWrapper *
GetForeignDataWrapper(Oid fdwid);
```

Cette fonction renvoie un objet `ForeignDataWrapper` pour le wrapper de données distantes de l'OID spécifié. Un objet `ForeignDataWrapper` contient les propriétés du wrapper (voir `foreign/foreign.h` pour les détails).

```
ForeignServer *
GetForeignServerExtended(Oid serverid, bits16 flags);
```

Cette fonction renvoie un objet `ForeignServer` pour le FDW avec l'OID précisé. Un objet `ForeignServer` contient les propriétés du FDW (voir `foreign/foreign.h` pour les détails). `flags` est un masque de bit OU indiquant des options supplémentaires. Il peut prendre la valeur `FSV_MISSING_OK`, auquel cas un `NULL` est retourné comme résultat à l'appelant au lieu d'une erreur d'objet indéfini.

```
ForeignServer *
GetForeignServer(Oid serverid);
```

Cette fonction renvoie un objet `ForeignServer` pour le serveur distant de l'OID donné. Un objet `ForeignServer` contient les propriétés du serveur (voir `foreign/foreign.h` pour les détails).

```
UserMapping *
GetUserMapping(Oid userid, Oid serverid);
```

Cette fonction renvoie un objet `UserMapping` pour la correspondance utilisateur du rôle donné sur le serveur donné. (S'il n'existe pas de correspondance utilisateur, la fonction renvoie la correspondance pour `PUBLIC` ou une erreur si cette dernière n'existe pas non plus.) Un objet `UserMapping` contient les propriétés de la correspondance utilisateur (voir `foreign/foreign.h` pour les détails).

```
ForeignTable *
GetForeignTable(Oid relid);
```

Cette fonction renvoie un objet `ForeignTable` pour la table distante de l'OID donné. Un objet `ForeignTable` contient les propriétés de la table distante (voir `foreign/foreign.h` pour les détails).

```
List *
GetForeignColumnOptions(Oid relid, AttrNumber attnum);
```

Cette fonction renvoie les opérations du wrapper de données distantes par colonne pour l'OID de la table distante donnée et le numéro de l'attribut sous la forme d'une liste de `DefElem`. NIL est renvoyé sur la colonne n'a pas d'options.

Certains types d'objets ont des fonctions de recherche basées sur le nom en plus de celles basées sur l'OID :

```
ForeignDataWrapper *
GetForeignDataWrapperByName(const char *name, bool missing_ok);
```

Cette fonction renvoie un objet `ForeignDataWrapper` pour le wrapper de données distante du nom indiqué. Si le wrapper n'est pas trouvé, cette fonction renvoie NULL si missing_ok vaut true, et renvoie une erreur sinon.

```
ForeignServer *
GetForeignServerByName(const char *name, bool missing_ok);
```

Cette fonction renvoie un objet `ForeignServer` pour le serveur distant du nom donné. Si le serveur n'est pas trouvé, cette fonction renvoie NULL si missing_ok vaut true, et renvoie une erreur sinon.

59.4. Planification de la requête avec un wrapper de données distantes

Les fonctions d'appels d'un wrapper de données distantes, `GetForeignRelSize`, `GetForeignPaths`, `GetForeignPlan`, `PlanForeignModify`, `GetForeignJoinPaths`, `GetForeignUpperPaths` et `PlanDirectModify` doivent s'intégrer au fonctionnement du planificateur de PostgreSQL. Voici quelques notes sur ce qu'elles doivent faire.

Les informations dans `root` et `baserel` peuvent être utilisées pour réduire la quantité d'informations qui doivent être récupérées sur la table distante (et donc réduire le coût) `baserel->baserestrictinfo` est tout particulièrement intéressant car il contient les qualificatifs de restriction (clauses `WHERE`) qui doivent être utilisées pour filtrer les lignes à récupérer. (Le wrapper lui-même n'est pas requis de respecter ces clauses car l'exécuteur du moteur peut les vérifier à sa place.) `baserel->reltargetlist` peut être utilisé pour déterminer les colonnes à récupérer ; mais notez qu'il liste seulement les colonnes qui doivent être émises par le nœud `ForeignScan`, et non pas les colonnes qui sont utilisées pour satisfaire l'évaluation des qualificatifs et non renvoyées par la requête.

Divers champs privés sont disponibles pour que les fonctions de planification du wrapper de données distantes conservent les informations. Habituellement, tout ce que vous stockez dans les champs privées doit avoir été alloué avec la fonction palloc, pour que l'espace soit récupéré à la fin de la planification.

`baserel->fdw_private` est un pointeur `void` disponible pour que les fonctions de planification du wrapper y stockent des informations correspondant à la table distante spécifique. Le planificateur du moteur n'y touche pas sauf lors de son initialisation à NULL quand le nœud `RelOptInfo` est créé. Il est utile de passer des informations de `GetForeignRelSize` à `GetForeignPaths` et/ou `GetForeignPaths` à `GetForeignPlan`, évitant du coup un recalcul.

`GetForeignPaths` peut identifier la signification de chemins d'accès différents pour enregistrer des informations privées dans le champ `fdw_private` des nœuds `ForeignPath`. `fdw_private` est déclaré comme un pointeur `List` mais peut contenir réellement n'importe quoi car le planificateur du moteur n'y touche pas. Néanmoins, une bonne pratique est d'utiliser une représentation qui est

affichable par `nodeToString`, pour son utilisation avec le support du débogage disponible dans le processus.

`GetForeignPlan` peut examiner le champ `fdw_private` du nœud `ForeignPath`, et peut générer les listes `fdw_exprs` et `fdw_private` à placer dans le nœud de plan `ForeignScan`, où elles seront disponibles au moment de l'exécution. Les deux listes doivent être représentées sous une forme que `copyObject` sait copier. La liste `fdw_private` n'a pas d'autres restrictions et n'est pas interprétée par le processus moteur. La liste `fdw_exprs`, si non NULL, devrait contenir les arbres d'expressions qui devront être exécutées. Ces arbres passeront par un post-traitement par le planificateur qui les rend complètement exécutables.

Dans `GetForeignPlan`, habituellement, la liste cible fournie peut être copiée dans le nœud du plan tel quel. La liste `scan_clauses` fournie contient les mêmes clauses que `baserel->baserestrictinfo` mais ces clauses pourraient être ré-ordonnées pour une meilleure efficacité à l'exécution. Dans les cas simples, le wrapper peut seulement supprimer les nœuds `RestrictInfo` de la liste `scan_clauses` (en utilisant `extract_actual_clauses`) et placer toutes les clauses dans la liste des qualificatifs du nœud. Cela signifie que toutes les clauses seront vérifiées par l'exécuteur au moment de l'exécution. Les wrappers les plus complexes peuvent être capables de vérifier certaines clauses en interne, auquel cas ces clauses peuvent être supprimées de la liste de qualificatifs du nœud du plan pour que le planificateur ne perde pas de temps à les vérifier de nouveau.

Comme exemple, le wrapper peut identifier certaines clauses de restriction de la forme *`variable_distante`* = *`sous_expression`*, qui, d'après lui, peut être exécuté sir le serveur distant en donnant la valeur évaluée localement de la *`sous_expression`*. L'identification réelle d'une telle clause doit survenir lors de l'exécution de `GetForeignPaths` car cela va affecter l'estimation ddu coût pour le chemin. Le champ `fdw_private` du chemin pourrait probablement inclure un pointeur vers le nœud `RestrictInfo` de la clause identifiée. Puis, `GetForeignPlan` pourrait supprimer cette clause de scan_clauses et ajouter la *`sous_expression`* à `fdw_exprs` pour s'assurer qu'elle soit convertie en une forme exécutable. Il pourrait aussi placer des informations de contrôle dans le champ `fdw_private` du nœud pour dire aux fonctions d'exécution ce qu'il faudra faire au moment de l'exécution. La requête transmise au serveur distant va impliquer quelque chose comme `WHERE` *`variable_distante`* `= $1`, avec la valeur du paramètre obtenu à l'exécution à partir de l'évaluation de l'arbre d'expression `fdw_exprs`.

Toutes les clauses enlevées de la liste des qualificatifs du nœud du plan doivent être à la place ajoutées à `fdw_recheck_quals` ou vérifiées à nouveau par `RecheckForeignScan` pour permettre un fonctionnement correct au niveau d'isolation `READ COMMITED`. Lorsqu'une mise à jour concurrente survient pour une autre table concernée par la requête, l'exécuteur peut avoir besoin de vérifier que tous les qualificatifs originaux sont encore satisfaits pour la ligne, éventuellement avec un ensemble différent de valeurs pour les paramètres. L'utilisation de `fdw_recheck_quals` est typiquement plus facile que de mettre en place les vérifications à l'intérieur de `RecheckForeignScan`, mais cette méthode sera insuffisante lorsque des jointures externes ont été poussées, dans la mesure où les lignes jointes dans ce cas peuvent avoir certaines colonnes à NULL sans rejeter la ligne entièrement.

Un autre champ `ForeignScan` qui peut être rempli par les FDW est `fdw_scan_tlist`, qui décrit les lignes renvoyées par le FDW pour ce nœud du plan. Pour les parcours simples de tables distantes, il peut être positionné à `NIL`, impliquant que les lignes renvoyées ont le type de ligne déclaré pour la table distante. Une valeur différente de `NIL` doit être une liste cible (liste de `TargetEntry`) contenant des variables et/ou expressions représentant les colonnes renvoyées. Ceci peut être utilisé, par exemple, pour montrer que le FDW a omis certaines colonnes qu'il a noté comme non nécessaire à la requête. Aussi, si le FDW peut calculer des expressions utilisées par la requête de manière moins coûteuse que localement, il pourrait ajouter ces expressions à `fdw_scan_tlist`. Notez que les plans de jointure (créés à partir des chemins construits par `GetForeignJoinPaths`) doivent toujours fournir `fdw_scand_tlist` pour décrire l'ensemble des colonnes qu'ils retourneront.

Le wrapper de données distantes devrait toujours construire au moins un chemin qui dépend seulement des clauses de restriction de la table. Dans les requêtes de jointure, il pourrait aussi choisir de construire des chemins qui dépendent des clauses de jointures. Par exemple, *`variable_distante`* `=` *`variable_local`*. De telles clauses ne se trouveront pas dans

`baserel->baserestrictinfo` mais doivent être dans les listes de jointures des relations. Un chemin utilisant une telle clause est appelé un « parameterized path ». Il doit identifier les autres relations utilisées dans le(s) clause(s) de jointure sélectionnée(s) avec une valeur convenable pour `param_info` ; utilisez `get_baserel_parampathinfo` pour calculer cette valeur. Dans `GetForeignPlan`, la portion *`local_variable`* de la clause de jointure pourra être ajoutée à `fdw_exprs`, et ensuite à l'exécution, cela fonctionne de la même façon que pour une clause de restriction standard.

Si un FDW supporte les jointures distantes, `GetForeignJoinPaths` devrait produire `ForeignPath` pour les jointures distantes potentielles essentiellement de la même manière que `GetForeignPaths` le fait pour les tables de base. L'information à propos de la jointure envisagée peut être passée à `GetForeignPlan` de la même manière que décrit ci-dessus. Cependant, `baserestrictinfo` n'est pas applicable pour les tables d'une jointure ; à la place, les clauses de jointure applicables pour une jointure particulière sont passées à `GetForeignJoinPaths` comme un paramètre séparé (`extra->restrictlist`).

Un FDW pourrait supporter en plus l'exécution direct de certaines actions d'un plan, qui sont au-dessus du niveau d'un parcours ou d'une jointure, comme par exemple un regroupement ou un agrégat. Pour proposer ce genre d'options, le FDW doit générer des chemins et les insérer dans la *relation de niveau supérieur* appropriée. Par exemple, un chemin représentant un agrégat distant doit être inséré dans la relation `UPPERREL_GROUP_AGG`, en utilisant `add_path`. Ce chemin sera comparé suivant son coût et celui d'un agrégat local réalisé en lisant un chemin de parcours simple de la relation externe (notez qu'un tel chemin doit aussi être fourni... dans le cas contraire, une erreur est renvoyée lors de l'optimisation). Si le chemin de l'agrégat distant gagne (ce qui sera généralement le cas), il sera converti en un plan standard en appelant `GetForeignPlan`. L'endroit recommendé pour générer de tels chemins est dans la fonction callback `GetForeignUpperPaths`, qui est appelée pour chaque relation supérieure (autrement dit, chaque étape de traitement post-parcours/jointure) si toutes les relations de base de la requête viennent du même FDW.

`PlanForeignModify` et les autres callbacks décrits dans Section 59.2.4 sont conçus autour de la supposition que la relation externe sera parcourue de la façon standard et qu'ensuite, les mises à jour individuelles de lignes seront réalisées par un nœud local `ModifyTable`. Cette approche est nécessaire dans le cas général où une mise à jour nécessite de lire des tables locales ainsi que des tables externes. Néanmoins, si l'opération pouvoit être exécutée entièrement par le serveur distant, le FDW pourrait générer un plan représentant cela et l'insérer dans la relation de niveau supérieur `UPPERREL_FINAL`, où il serait comparé avec l'approche `ModifyTable`. Cette approche pourrait être utilisé pour implémenter un `SELECT FOR UPDATE` distant, plutôt que d'utiliser les callbacks de verrouillage de ligne décrits dans Section 59.2.6. Gardez à l'exprit qu'un chemin inséré dans `UPPERREL_FINAL` est responsable de l'implémentation de *tout* le comportement de cette requête.

Lors de la planification d'un `UPDATE` ou d'un `DELETE`, `PlanForeignModify` et `PlanDirectModify` peuvent rechercher la structure `RelOptInfo` pour la table distante et utiliser la donnée `baserel->fdw_private` créée précédemment par les fonctions de planification de parcours. Néanmoins, pour un `INSERT`, la table cible n'est pas parcourue, donc il n'existe aucun `RelOptInfo` pour elle. La structure `List` renvoyée par `PlanForeignModify` a les mêmes restrictions que la liste `fdw_private` d'un nœud de plan `ForeignScan`, c'est-à-dire qu'elle doit contenir seulement les structures que `copyObject` sait copier.

Une commande `INSERT` avec une clause `ON CONFLICT` ne supporte pas la spécification d'une cible de conflit, dans la mesure où les contraintes uniques ou les contraintes d'exclusion sur les tables distantes ne sont pas localement connues. Ceci entraîne également que `ON CONFLICT DO UPDATE` n'est pas supporté car la spécification est obligatoire ici.

59.5. Le verrouillage de ligne dans les wrappers de données distantes

Si le mécanisme de stockage sous-jacent à un FDW a un concept de verrouillage individuel des lignes pour prévenir des mises à jour concurrentes de ces lignes, il est généralement intéressant

pour le FDW d'effectuer des verrouillages de niveau ligne avec une approximation aussi proche que possible de la sémantique utilisée pour les tables ordinaires de PostgreSQL. Ceci implique de multiples considérations.

Une décision clef à prendre est si il vaut mieux effectuer un *verrouillage précoce* ou un *verrouillage tardif*. Dans le verrouillage précoce, une ligne est verrouillée lorsqu'elle est récupérée pour la première fois à partir du stockage sous-jacent, alors qu'avec le verrouillage tardif, la ligne est verrouillée seulement lorsque le besoin est connu et nécessaire. (La différence survient parce que certaines lignes peuvent être abandonnées par des restrictions vérifiées localement ou des conditions de jointure.) Le verrouillage précoce est beaucoup plus simple et évite des allers-retours supplémentaires vers le stockage distant, mais il peut entraîner des verrouillages de lignes qui n'auraient pas eu besoin de l'être, résultant en une réduction de la concurrence voire même des deadlocks inattendus. De plus, le verrouillage tardif n'est possible seulement que si la ligne à verrouiller peut être identifiée de manière unique à nouveau plus tard. Idéalement, l'identifiant de ligne devrait identifier une version spécifique de la ligne, comme les TID de PostgreSQL le font.

Par défaut, PostgreSQL ignore les considérations de verrouillage lorsqu'il s'interface avec les FDW, mais un FDW peut effectuer un verrouillage précoce sans un support explicite du code du serveur principal. Les fonctions de l'API décrites dans le Section 59.2.6, qui ont été ajoutées dans la version 9.5 de PostgreSQL, autorise un FDW à utiliser un verrouillage tardif si il le désire.

Une considération supplémentaire est que dans le niveau d'isolation `READ COMMITTED`, PostgreSQL peut avoir besoin de vérifier à nouveau les restrictions et conditions de jointures avec une version mise à jour de certaines lignes. Vérifier à nouveau des conditions de jointure requiert d'obtenir à nouveau des copies des lignes non ciblées qui étaient auparavant jointes à la ligne cible. En travaillant avec des tables standards PostgreSQL, ceci est effectué en incluant les TID des tables non ciblées dans la liste des colonnes projetées via la jointure, puis en récupérant à nouveau les lignes non ciblées si nécessaire. Cette approche maintient l'ensemble des données jointes compact, mais il demande une capacité peu coûteuse de récupération à nouveau, ainsi qu'un TID qui peut identifier de manière unique la version de la ligne à récupérer à nouveau. Par défaut, donc, l'approche utilisée avec les tables distantes est d'inclure une copie de la ligne entière récupérée dans la liste de colonnes projetée via la jointure. Ceci n'impose rien au FDW mais peut entraîner des performances réduites des jointures par fusion ou hachage. Un FDW qui remplit les conditions pour récupérer à nouveau peut choisir de le faire.

Pour une commande `UPDATE` ou `DELETE` sur une table distante, il est recommandé que l'opération de `ForeignScan` sur la table cible effectue un verrouillage précoce sur les lignes qu'elle récupère, peut-être via un équivalent de la commande `SELECT FOR UPDATE`. Un FDW peut détecter si une table est la cible d'une commande `UPDATE`/`DELETE` lors de la planification en comparant son relid à `root->parse->resultRelation`, ou lors de l'exécution en utilisant la fonction `ExecRelationIsTargetRelation()`. Une possibilité alternative est d'effectuer un verrouillage tardif à l'intérieur des fonctions callback `ExecForeignUpdate` ou `ExecForeignDelete`, mais aucun support spécial n'est fourni pour cela.

Pour les tables distantes qui sont verrouillées par une commande `SELECT FOR UPDATE/SHARE`, l'opération `ForeignScan` peut encore effectuer un verrouillage précoce en récupérant des lignes avec l'équivalent de la commande `SELECT FOR UPDATE/SHARE`. Pour effectuer à la place un verrouillage tardif, fournissez les fonctions callback définies à Section 59.2.6. Dans `GetForeignRowMarkType`, sélectionner l'option rowmark `ROW_MARK_EXCLUSIVE`, `ROW_MARK_NOKEYEXCLUSIVE`, `ROW_MARK_SHARE` ou `ROW_MARK_KEYSHARE` en fonction de la force du verrouillage demandé. (Le code du serveur principal agira de la même manière indépendamment de l'option choisie parmi ces quatre options.) Ailleurs, vous pouvez détecter si une table distante a été verrouillée par ce type de commandes en utilisant la fonction `get_plan_rowmark` lors de la planification ou la fonction `ExecFindRowMark` lors de l'exécution ; vous devez vérifier non seulement si une structure rowmark non nulle est renvoyée, mais également que son champ `strength` n'est pas égal à `LCS_NONE`.

Enfin, pour les tables distantes qui sont utilisées dans une commande `UPDATE`, `DELETE` ou `SELECT FOR UPDATE/SHARE` sans demande de verrouillage de ligne, vous pouvez passer outre le choix par défaut de copier les lignes entières dans la fonction `GetForeignRowMarkType`

en sélectionnant l'option ROW_MARK_REFERENCE lorsqu'elle voit comme valeur de puissance de verrouillage LCS_NONE. Ceci aura pour conséquence d'appeler RefetchForeignRow avec cette valeur pour le champ markType ; elle devrait alors récupérer à nouveau la ligne sans prendre aucun nouveau verrouillage. (Si vous avez une fonction GetForeignRowMarkType mais ne souhaitez pas récupérer à nouveau des lignes non verrouillées, sélectionnez l'option ROW_MARK_COPY pour LCS_NONE.)

Voir les commentaires dans src/include/nodes/lockoptions.h, pour RowMarkType et dans src/include/nodes/plannodes.h pour PlanRowMark, et les commentaires pour ExecRowMark dans src/include/nodes/execnodes.h pour des informations complémentaires.

Chapitre 60. Écrire une méthode d'échantillonnage de table

L'implémentation de la clause TABLESAMPLE de PostgreSQL supporte l'utilisation de méthodes personnalisées d'échantillonnage de table, en plus des méthodes BERNOULLI et SYSTEM qui sont requises par le standard SQL. La méthode d'échantillonnage détermine les lignes de la table sélectionnées lorsque la clause TABLESAMPLE est utilisée.

Au niveau SQL, une méthode d'échantillonnage de table est représentée par une simple fonction, classiquement implémentée en C, et qui a la signature suivante :

```
method_name(internal) RETURNS tsm_handler
```

Le nom de la fonction est le même que le nom de la méthode apparaissant dans la clause TABLESAMPLE. L'argument internal est factice (il a toujours une valeur de zéro) qui sert uniquement à interdire que cette fonction soit appelée directement à partir d'une commande SQL. Le résultat de cette fonction doit être une structure allouée avec palloc de type TsmRoutine, qui contient des pointeurs de fonction supportant la méthode d'échantillonnage. Ces fonctions sont des fonctions C pleines et entières qui ne sont ni visibles ni appellables au niveau SQL. Les fonctions de support sont décrites dans le Section 60.1.

En plus des pointeurs de fonction, la structure TsmRoutine doit fournir ces champs additionnels :

`List *parameterTypes`

Il s'agit d'une liste d'OID contenant les OID des types de données du ou des paramètre(s) qui seront acceptés par la clause TABLESAMPLE lorsque cette méthode d'échantillonnage sera utilisée. Par exemple, pour les méthodes incluses, cette liste contient un simple élément avec la valeur FLOAT4OID, qui représente le pourcentage d'échantillonnage. Les méthodes d'échantillonnage personnalisées peuvent avoir des paramètres en plus ou différents.

`bool repeatable_across_queries`

Si true, la méthode d'échantillonnage peut renvoyer des échantillons identiques pour des requêtes successives, si les mêmes paramètres et la valeur de graine de la clause REPEATABLE sont fournis à chaque fois et que le contenu de la table n'a pas changé. Lorsque positionné à false, la clause REPEATABLE n'est pas acceptée comme valable pour la méthode d'échantillonnage.

`bool repeatable_across_scans`

Si true, la méthode d'échantillonnage peut renvoyer des échantillons identiques pour des parcours successifs dans la même requête (en supposant des paramètres, une graine et une image de la base inchangés). Lorsque positionné à false, le planificateur ne sélectionnera pas des plans qui requièrent de parcourir la table échantillonnée plus d'une fois, dans la mesure où ceci pourrait entraîner des résultats de sortie incohérents.

La structure TsmRoutine est déclarée dans le fichier src/include/access/tsmapi.h, auquel il convient de se référer pour des détails supplémentaires.

Les méthodes d'échantillonnage de table incluses dans la distribution standard sont de bonnes références pour écrire la vôtre. Jeter un œil dans le répertoire src/backend/access/tablesample de l'arbre des sources pour les méthodes incluses, et dans le répertoire contrib pour des méthodes additionnelles.

60.1. Fonctions de support d'une méthode d'échantillonnage

La fonction du gestionnaire TSM renvoie une structure `TsmRoutine` allouée avec palloc contenant des pointeurs vers les fonctions de support décrites ci-dessous. La plupart des fonctions sont obligatoires, mais certaines sont optionnelles, et leurs pointeurs peuvent être NULL.

```
void
SampleScanGetSampleSize (PlannerInfo *root,
                         RelOptInfo *baserel,
                         List *paramexprs,
                         BlockNumber *pages,
                         double *tuples);
```

Cette fonction est appelée durant la planification. Elle doit estimer le nombre de pages de la relation qui seront lues lors d'un simple parcours, et le nombre de lignes qui seront sélectionnées lors du parcours. (Par exemple, cela pourrait être déterminé en estimant la fraction échantillonnée, puis en multipliant `baserel->pages` et `baserel->tuples` par ce chiffre, après s'être assuré d'avoir arrondi ces chiffres à des valeurs entières.) La liste `paramexprs` contient les expressions qui sont les paramètres de la clause `TABLESAMPLE`. Il est recommandé d'utiliser la fonction `estimate_expression_value` pour essayer de réduire ces expressions à des constantes, si leurs valeurs sont nécessaires pour les besoins de l'estimation ; mais la fonction doit renvoyer les estimations des tailles même si elles ne peuvent être réduites, et elle ne devrait pas échouer même si les valeurs apparaissent invalides (rappelez-vous qu'il s'agit uniquement d'une estimation de valeurs futures à l'exécution). Les paramètres `pages` et `tuples` sont les valeurs de sortie.

```
void
InitSampleScan (SampleScanState *node,
                int eflags);
```

Initialise pour l'exécution d'un nœud du plan SampleScan. La fonction est appelée au démarrage de l'exécuteur. Elle devrait effectuer toutes les initialisations nécessaires avant que le traitement ne puisse commencer. Le nœud `SampleScanState` a déjà été créé, mais son champ `tsm_state` est NULL. La fonction peut allouer via palloc les données internes d'état nécessaires à la fonction d'échantillonnage, et enregistrer un pointeur dans `node->tsm_state`. Des informations à propos de la table à parcourir sont accessibles via d'autres champs du nœud `SampleScanState` (mais veuillez noter que le descripteur du parcours `node->ss.ss_currentScanDesc` n'est pas encore positionné à ce stade). `eflags` contient un ensemble de bits décrivant le mode opératoire de l'exécuteur pour ce nœud du plan.

Lorsque `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` est true, le parcours ne sera pas encore effectué. Dans ce cas, cette fonction devrait effectuer uniquement le minimum requis pour mettre dans un état valide le nœud pour la commande `EXPLAIN` et la fonction `EndSampleScan`.

Cette fonction est optionnelle (positionnez alors le pointeur sur NULL), auquel cas la fonction `BeginSampleScan` doit effectuer toutes les initialisations nécessaires à la méthode d'échantillonnage.

```
void
BeginSampleScan (SampleScanState *node,
                 Datum *params,
                 int nparams,
                 uint32 seed);
```

Débute l'exécution d'un parcours d'échantillonnage. Cette fonction est appelée juste avant la première tentative de récupération d'une ligne, et peut être appelée à nouveau si le parcours a besoin d'être relancé. Des informations sur la table à parcourir sont accessibles via les champs de la structure du nœud `SampleScanState` (mais notez que le descripteur du parcours `node->ss.ss_currentScanDesc` n'est pas encore positionné à ce stade). Le tableau `params`, de longueur `nparams`, contient les valeurs des paramètres indiqués dans la clause `TABLESAMPLE`. Ces paramètres seront en nombre et de types spécifiés par la méthode d'échantillonnage dans la liste `parameterTypes`, et ont été vérifiés comme n'étant pas null. `seed`> contient une graine à usage de la méthode d'échantillonnage pour générer des nombres aléatoires ; il s'agit d'un hash dérivé de la valeur de la clause `REPEATABLE` si fournie, ou du résultat de la fonction `random()` dans le cas contraire.

Cette fonction peut ajuster les champs `node->use_bulkread` et `node->use_pagemode`. Si `node->use_bulkread` est true, ce qui est le cas par défaut, le parcours utilisera une stratégie d'accès aux tampons mémoires qui encourage le recyclage des tampons après usage. Il peut être raisonnable de mettre cette valeur à `false` si le parcours doit visiter seulement une petite fraction des pages de la table. Si `node->use_pagemode` est `true`, ce qui est la valeur par défaut, le parcours effectuera une vérification de la visibilité avec un unique passage pour l'ensemble des lignes composant chaque page visitée. Il peut être raisonnable de mettre cette valeur à `false` si le parcours doit sélectionner seulement une petite fraction des lignes de chaque page visitée. Ceci aura pour conséquence un nombre moindre de vérifications de visibilité effectuées, mais chacune sera plus coûteuse car elle demandera plus de verrouillages.

Si la méthode d'échantillonnage est marquée comme `repeatable_across_scans`, elle doit être capable de sélectionner le même ensemble de lignes lors d'un parcours relancé à nouveau comme elle l'a fait à l'origine, c'est-à-dire qu'un nouvel appel à la fonction `BeginSampleScan` doit engendrer la sélection des mêmes lignes que précédemment (dans la mesure où les paramètres de la clause `TABLESAMPLE` et la graine ne changent pas).

```
BlockNumber
NextSampleBlock (SampleScanState *node, BlockNumber nblocks);
```

Renvoie le numéro du bloc de la page suivante à parcourir, ou `InvalidBlockNumber` si il n'y a plus de pages à parcourir.

Cette fonction peut être omise (mettez le pointeur à la valeur NULL), auquel cas le code du serveur effectuera un parcours séquentiel de l'ensemble de la relation. Un tel parcours peut utiliser un parcours synchronisé, aussi la méthode d'échantillonnage ne peut pas supposer que les pages de la relation sont visitées dans le même ordre à chaque parcours.

```
OffsetNumber
NextSampleTuple (SampleScanState *node,
                 BlockNumber blockno,
                 OffsetNumber maxoffset);
```

Renvoie le décalage de la ligne suivante à echantillonner sur la page spécifiée, ou `InvalidOffsetNumber` si il n'y a plus de lignes à échantillonner. `maxoffset` est le décalage le plus grand utilisé sur la page.

> **Note**
>
> Il n'est pas explicitement indiqué à la fonction `NextSampleTuple` les décalages dans l'intervalle `1 .. maxoffset` qui contiennent des lignes valides. Ce n'est normalement pas

un problème dans la mesure où le code du serveur ignore les requêtes pour échantillonner des lignes manquantes ou non visibles ; ceci ne devrait pas entraîner de biais dans l'échantillon. Cependant, si nécessaire, la fonction peut utiliser `node->donetuples` pour examiner le nombre de lignes renvoyées valides et visibles.

Note

La fonction `NextSampleTuple` ne doit *pas* assumer que `blockno` est le même numéro de page que celui renvoyé par le plus récent appel à la fonction `NextSampleBlock`. Le numéro a été renvoyé par un précédent appel à la fonction `NextSampleBlock`, mais le code du serveur est autorisé à appeler `NextSampleBlock` en amont du parcours des pages, pour rendre possible la récupération en avance. Il est acceptable d'assumer qu'une fois le parcours d'une page débuté, les appels successifs à la fonction `NextSampleTuple` se réfèrent tous à la même page jusqu'à ce que `InvalidOffsetNumber` soit retourné.

```
void
EndSampleScan (SampleScanState *node);
```

Termine le parcours et libère les ressources. Il n'est normalement pas important de libérer la mémoire allouée via palloc, mais toutes les ressources visibles à l'extérieur doivent être nettoyées. Cette fonction peut être omise (positionnez le pointeur sur la valeur NULL) dans la plupart des cas où de telles ressources n'existent pas.

Chapitre 61. Écrire un module de parcours personnalisé

PostgreSQL supporte un ensemble de fonctionnalités expérimentales destinées à permettre à des modules d'extension d'ajouter de nouveaux types de parcours au système. Contrairement aux wrapper de données distantes, qui sont seulement en charge de savoir comment parcourir leurs propres tables distantes, un module de parcours personnalisé peut fournir une méthode alternative de parcours de n'importe quelle relation du système. Typiquement, la motivation pour écrire un module de parcours personnalisé serait d'utiliser des optimisations non supportées par le système de base, telles que la mise en cache ou certaines formes d'accélération matérielles. Ce chapitre décrit les grandes lignes de l'écriture d'un nouveau module de parcours personnalisé.

Développer un nouveau type de parcours personnalisé est un processus en trois étapes. Premièrement, lors de la planification, il est nécessaire de générer des chemins d'accès représentant un parcours utilisant la stratégie proposée. Deuxièmement, si l'un de ces chemins d'accès est sélectionné par le planificateur comme la stratégie optimale pour parcourir une relation particulière, le chemin d'accès doit être converti en plan. Finalement, il doit être possible d'exécuter le plan et de générer le même résultat qui aurait été généré pour tous les autres chemins d'accès visant la même relation.

61.1. Créer des parcours de chemin personnalisés

Un module de parcours personnalisé ajoutera habituellement des chemins pour une relation de base en mettant en place le hook suivant, qui est appelé après que le code de base ait généré tous les chemins d'accès possibles pour la relation (sauf les chemins Gather, qui sont réalisés après cet appel pour qu'ils puissent utiliser les chemins partiels ajoutés par le hook :

```
typedef void (*set_rel_pathlist_hook_type) (PlannerInfo *root,
                                            RelOptInfo *rel,
                                            Index rti,
                                            RangeTblEntry *rte);
extern PGDLLIMPORT set_rel_pathlist_hook_type
 set_rel_pathlist_hook;
```

Bien que cette fonction puisse être utilisée pour examiner, modifier ou supprimer des chemins générés par le système de base, un module de parcours personnalisé se limitera généralement lui-même à générer des objets `CustomPath` et à les ajouter à `rel` en utilisant la fonction `add_path`. Le module de parcours personnalisé a la charge d'initialiser l'objet `CustomPath`, qui est déclaré comme suit :

```
typedef struct CustomPath
{
    Path      path;
    uint32    flags;
    List     *custom_paths;
    List     *custom_private;
    const CustomPathMethods *methods;
} CustomPath;
```

`path` doit être initialisé comme pour tous les autres chemins, y compris l'estimation du nombre de lignes, le coût de départ et le coût total, et l'ordre de tri fourni par ce chemin. `flags` est un masque de bits, qui indique si le fournisseur de parcours peut accepter certaines fonctionnalités

optionnelles. `flags` devrait inclure `CUSTOMPATH_SUPPORT_BACKWARD_SCAN` si le chemin personnalisé supporte le parcours inverse et `CUSTOM_SUPPORT_MARK_RESTORE` si il peut supporter le marquage et la restauration, et `CUSTOMPATH_SUPPORT_PROJECTION` s'il peut réaliser projections. (Si `CUSTOMPATH_SUPPORT_PROJECTION` n'est pas configuré, le nœud de parcours se verra seulement demandé de produire le Vars de la relation parcourue ; alors que s'il est configuré, le nœud du parcours doit être caoavle d'évaluer les expressions scalaires sur ces Vars.) Une liste optionnelle `custom_paths` est une liste de nœuds `Path` utilisés par ce nœud de chemin personnalisé ; ils seront transformés en nœuds `Plan` par le planificateur. `custom_private` peut être utilisé pour stocker les données privées du chemin personnalisé. Les données privées devraient être stockées dans une forme qui puisse être traitée par `nodeToString`, de telle manière que les routines de debuggage qui essaient d'imprimer le chemin personnalisé fonctionnent comme prévu. `methods` doit pointer vers un objet (généralement alloué statiquement) implémentant les méthodes obligatoires d'un chemin personnalisé, qui sont détaillées ci-dessous.

Un module de parcours personnalisé peut également fournir des chemins de jointure. De la même manière que pour les relations de base, un tel chemin doit produire la même sortie qui serait normalement produite par la jointure qu'il remplace. Pour réaliser ceci, le module de jointure devrait mettre en place le hook suivant, puis, à l'intérieur de cette fonction, créer un ou des chemins `CustomPath` pour la relation de jointure.

```
typedef void (*set_join_pathlist_hook_type) (PlannerInfo *root,
                                             RelOptInfo *joinrel,
                                             RelOptInfo *outerrel,
                                             RelOptInfo *innerrel,
                                             JoinType jointype,
                                             JoinPathExtraData
 *extra);
extern PGDLLIMPORT set_join_pathlist_hook_type
 set_join_pathlist_hook;
```

Cette fonction sera appelée de manière répétée pour la même relation de jointure, avec différentes combinaisons de relations internes ou externes ; la fonction a la charge de minimiser la duplication des travaux.

61.1.1. Fonctions callbacks d'un parcours de chemin personnalisé

```
Plan *(*PlanCustomPath) (PlannerInfo *root,
                         RelOptInfo *rel,
                         CustomPath *best_path,
                         List *tlist,
                         List *clauses,
                         List *custom_plans);
```

Convertit un chemin personnalisé en un plan finalisé. La valeur de retour sera généralement un objet `CustomScan`, que la fonction callback doit allouer et initialiser. Voir Section 61.2 pour plus de détails.

```
List *(*ReparameterizeCustomPathByChild) (PlannerInfo *root,
                                          List *custom_private,
                                          RelOptInfo *child_rel);
```

Cette fonction callback est appelée lors de la conversion d'un chemin à paramètres par le parent de la relation `child_rel`. La fonction callback est utilisé pour re-paramétrer tout chemin ou

traduire des nœuds d'expression sauvegardé dans le membre `custom_private` donné d'un `CustomPath`. La fonction callback pourrait utiliser `reparameterize_path_by_child`, `adjust_appendrel_attrs` ou `adjust_appendrel_attrs_multilevel` comme requis.

61.2. Créer des parcours de plans personnalisés

Un parcours personnalisé est représenté dans un arbre de plans finalisé en utilisant la structure suivante :

```
typedef struct CustomScan
{
    Scan      scan;
    uint32    flags;
    List     *custom_plans;
    List     *custom_exprs;
    List     *custom_private;
    List     *custom_scan_tlist;
    Bitmapset *custom_relids;
    const CustomScanMethods *methods;
} CustomScan;
```

`scan` doit être initialisé comme pour tous les autres parcours, y compris le coût estimé, les listes cibles, les qualifications, et ainsi de suite. `flags` est un masque de bits avec la même signification que dans `CustomPath`. `custom_plans` peut être utilisé pour stocker des nœuds enfants de type `Plan`. `custom_exprs` devrait être utilisé pour stocker des arbres d'expressions qui devront être corrigés par `setrefs.c` et `subselect.c`, tandis que `custom_private` devrait être utilisé pour stocker d'autres données privées qui sont seulement utilisées par le module de parcours personnalisé lui-même. `custom_scan_tlist` peut être à NIL lors du parcours d'une relation de base, indiquant que le parcours personnalisé renvoie des lignes parcourues qui correspondent au type des lignes de la relation de base. Dans le cas contraire, il s'agit d'une liste de cibles décrivant les lignes actuellement parcourues. `custom_scan_tlist` devrait être fourni pour les jointures, et peut être fourni pour les parcours dont le module de parcours personnalisé peut calculer certaines expressions non variables. `custom_relids` est positionné par le code du serveur sur l'ensemble des relations (index de l'ensemble des tables) que ce nœud de parcours gère ; sauf lorsque ce parcours remplace une jointure, il aura alors un seul membre. `methods` doit pointer sur un objet (généralement alloué statiquement) implémentant les méthodes requises d'un parcours personnalisé, lesquelles sont détaillées ci-dessous.

Lorsqu'un `CustomScan` parcourt une simple relation, `scan.scanrelid` doit être l'index dans l'ensemble des tables de la table à parcourir. Lorsqu'il remplace une jointure, `scan.scanrelid` devrait être à zéro.

Les arbres de plan doivent pouvoir être dupliqués en utilisant la fonction `copyObject`, aussi les données stockées dans les champs « custom » doivent consister en des nœuds que cette fonction peut gérer. De plus, les modules de parcours personnalisés ne peuvent pas substituer une structure plus large qui incorporerait une structure de type `CustomScan`, comme il est possible pour les structures `CustomPath` ou `CustomScanState`.

61.2.1. Fonctions callbacks d'un plan de parcours personnalisé

```
Node *(*CreateCustomScanState) (CustomScan *cscan);
```

Alloue une structure `CustomScanState` pour ce `CustomScan`. L'allocation actuelle sera souvent plus grande que requis pour une structure ordinaire `CustomScanState` car beaucoup de modules voudront incorporer celui-ci comme le premier champ d'une structure plus large. La valeur renvoyée doit avoir la marque du nœud et le champ `methods` positionnés correctement, les autres champs devraient être laissés à zéro à ce stade ; après que la fonction `ExecInitCustomScan` ait effectué une initialisation basique, la fonction `BeginCustomScan` sera appelée pour permettre au module de parcours personnalisé d'effectuer ce qu'il a besoin de faire.

61.3. Exécution de parcours personnalisés

Lorsqu'un `CustomScan` est exécuté, l'état de son exécution est représenté par un `CustomScanState`, qui est déclaré comme suit :

```
typedef struct CustomScanState
{
    ScanState ss;
    uint32    flags;
    const CustomExecMethods *methods;
} CustomScanState;
```

`ss` est initialisé comme tous les autres états de parcours, sauf que si le parcours est pour une jointure plutôt qu'une relation, `ss.ss_currentRelation` est laissé à NULL. `flags` est un masque de bits avec la même signification que dans `CustomPath` et `CustomScan`. `methods` doit pointer vers un objet (généralement alloué statiquement) implémentant les méthodes requises d'un état de parcours personnalisé, qui sont détaillées ci-dessous. Typiquement, une structure `CustomScanState`, qui n'a pas besoin de supporter la fonction `copyObject`, sera actuellement une structure plus grande incorporant la structure ci-dessus comme premier membre.

61.3.1. Fonction callbacks d'exécution d'un parcours personnalisé

```
void (*BeginCustomScan) (CustomScanState *node,
                         EState *estate,
                         int eflags);
```

Complète l'initialisation de la structure `CustomScanState`. Les champs standards ont été initialisés par la fonction `ExecInitCustomScan`, mais tous les champs privés devraient être initialisés ici.

```
TupleTableSlot *(*ExecCustomScan) (CustomScanState *node);
```

Récupère la ligne suivante du parcours. Si il existe des lignes restantes, la fonction devrait remplir `pg_ResultTupleSlot` avec la ligne suivante dans le sens actuel du parcours, puis renvoyer le slot de la ligne. Dans le cas contraire, `NULL` ou un slot vide devrait être renvoyé.

```
void (*EndCustomScan) (CustomScanState *node);
```

Nettoie les données privées associées avec le `CustomScanState`. Cette méthode est obligatoire, mais elle n'a pas besoin de faire quoi que ce soit si il n'y a pas de données associées ou des données qui seront nettoyées automatiquement.

```
void (*ReScanCustomScan) (CustomScanState *node);
```

Repositionne au début le parcours en cours et prépare à parcourir de nouveau la relation.

```
void (*MarkPosCustomScan) (CustomScanState *node);
```

Enregistre la position du parcours courant de telle manière qu'elle puisse être restaurée par la fonction callback `RestrPosCustomScan`. Cette fonction callback est facultative, et n'a besoin d'être fournie que si le drapeau `CUSTOMPATH_SUPPORT_MARK_RESTORE` est positionné.

```
void (*RestrPosCustomScan) (CustomScanState *node);
```

Restaure la position précédente du parcours telle que sauvegardée par la fonction `MarkPosCustomScan`. Cette fonction callback est facultative, et n'a besoin d'être fournie que si le drapeau `CUSTOMPATH_SUPPORT_MARK_RESTORE` est positionné.

```
Size (*EstimateDSMCustomScan) (CustomScanState *node,
                               ParallelContext *pcxt);
```

Estime la quantité de mémoire partagée dynamique qui sera requise pour l'opération parallèlisée. Elle pourrait être plus importante que la quantité réellement utilisée, mais elle ne doit pas être moindre. La valeur en retour est en octets. Cette fonction est optionnelle. Elle n'est nécessaire que si ce type de parcours supporte une exécution parallélisée.

```
void (*InitializeDSMCustomScan) (CustomScanState *node,
                                 ParallelContext *pcxt,
                                 void *coordinate);
```

Initialise la mémoire partagée dynamique requise pour une opération parallélisée. L'argument `coordinate` pointe vers une partie de la mémoire partagée de taille identique à la valeur en retour de `EstimateDSMCustomScan`. Cette fonction est optionnelle. Elle n'est nécessaire que si ce type de parcours supporte une exécution parallélisée.

```
void (*ReInitializeDSMCustomScan) (CustomScanState *node,
                                   ParallelContext *pcxt,
                                   void *coordinate);
```

Ré-initialise la mémoire partagée dynamique requise pour des opérations parallélisées lorsque le nœud du plan pour le parcours personnalisé doit être réalisé de nouveau. Cette fonction est optionnelle et doit seulement être fournie si le fournisseur de ce parcours personnalisé supporte les exécutions parallélisées. La pratique recommandée est que cette fonction réinitialise seulement l'état partagé alors que la fonction `ReScanCustomScan` réinitialise seulement l'état local. Actuellement, cette fonction sera appelée avant `ReScanCustomScan` mais il est préférable de ne pas se fier à l'ordre des opérations.

```
void (*InitializeWorkerCustomScan) (CustomScanState *node,
                                    shm_toc *toc,
```

```
                                    void *coordinate);
```

Initialise un état local d'un processus en parallèle basé sur la configuration de l'état partagée dans le processus principal par `InitializeDSMCustomScan`. Cette fonction est optionnelle. Elle n'est nécessaire que si le fournisseur de ce parcours personnalisé supporte une exécution parallélisée.

```
void (*ShutdownCustomScan) (CustomScanState *node);
```

Libère les ressources quand il est anticipé que le nœud ne sera pas exécuté entièrement. Cette fonction ne sera pas appelée dans tous les cas; parfois, `EndCustomScan` peut être appelée sans que cette ait été appelée avant. Puisque le segment DSM utilisé par les requêtes parallèles est détruit juste après que ce callback soit appelé, les modules de parcours personnalisés qui désirent effectuer des actions avant que le segment DSM disparaissent devraient implémenter cette méthode.

```
void (*ExplainCustomScan) (CustomScanState *node,
                           List *ancestors,
                           ExplainState *es);
```

Envoie sur la sortie des informations additionnelles pour la commande `EXPLAIN` d'un nœud du plan d'un parcours personnalisé. Cette fonction est facultative. Les données communes enregistrées dans la structure `ScanState`, tel que la liste des cibles et la relation parcourue, seront montrées même sans cette fonction callback, mais la fonction permet l'affichage d'états additionnels, privés.

Chapitre 62. Optimiseur génétique de requêtes (*Genetic Query Optimizer*)

Auteur

Écrit par Martin Utesch (<utesch@aut.tu-freiberg.de>) de l'Institut de Contrôle Automatique à l'Université des Mines et de Technologie de Freiberg, en Allemagne.

62.1. Gérer les requêtes, un problème d'optimisation complexe

De tous les opérateurs relationnels, le plus difficile à exécuter et à optimiser est la jointure (*join*). Le nombre de plans de requêtes possibles croît exponentiellement avec le nombre de jointures de la requête. Un effort supplémentaire d'optimisation est nécessité par le support de différentes *méthodes de jointure* (boucles imbriquées, jointures de hachage, jointures de fusion...) pour exécuter des jointures individuelles et différents *index* (B-tree, hash, GiST et GIN...) pour accéder aux relations.

L'optimiseur standard de requêtes pour PostgreSQL réalise une *recherche quasi-exhaustive* sur l'ensemble des stratégies alternatives. Cet algorithme, introduit à l'origine dans la base de données System R d'IBM, produit un ordre de jointure quasi-optimal mais peut occuper beaucoup de temps et de mémoire à mesure que le nombre de jointures d'une requête augmente. L'optimiseur ordinaire de requêtes de PostgreSQL devient donc inapproprié pour les requêtes qui joignent un grand nombre de tables.

L'Institut de Contrôle Automatique de l'Université des Mines et de Technologie basé à Freiberg, en Allemagne, a rencontré des difficultés lorsqu'il s'est agi d'utiliser PostgreSQL comme moteur d'un système d'aide à la décision reposant sur une base de connaissance utilisé pour la maintenance d'une grille de courant électrique. Le SGBD devait gérer des requêtes à nombreuses jointures pour la machine d'inférence de la base de connaissances. Le nombre de jointures de ces requêtes empêchait l'utilisation de l'optimiseur de requête standard.

La suite du document décrit le codage d'un *algorithme génétique* de résolution de l'ordonnancement des jointures qui soit efficace pour les requêtes à jointures nombreuses.

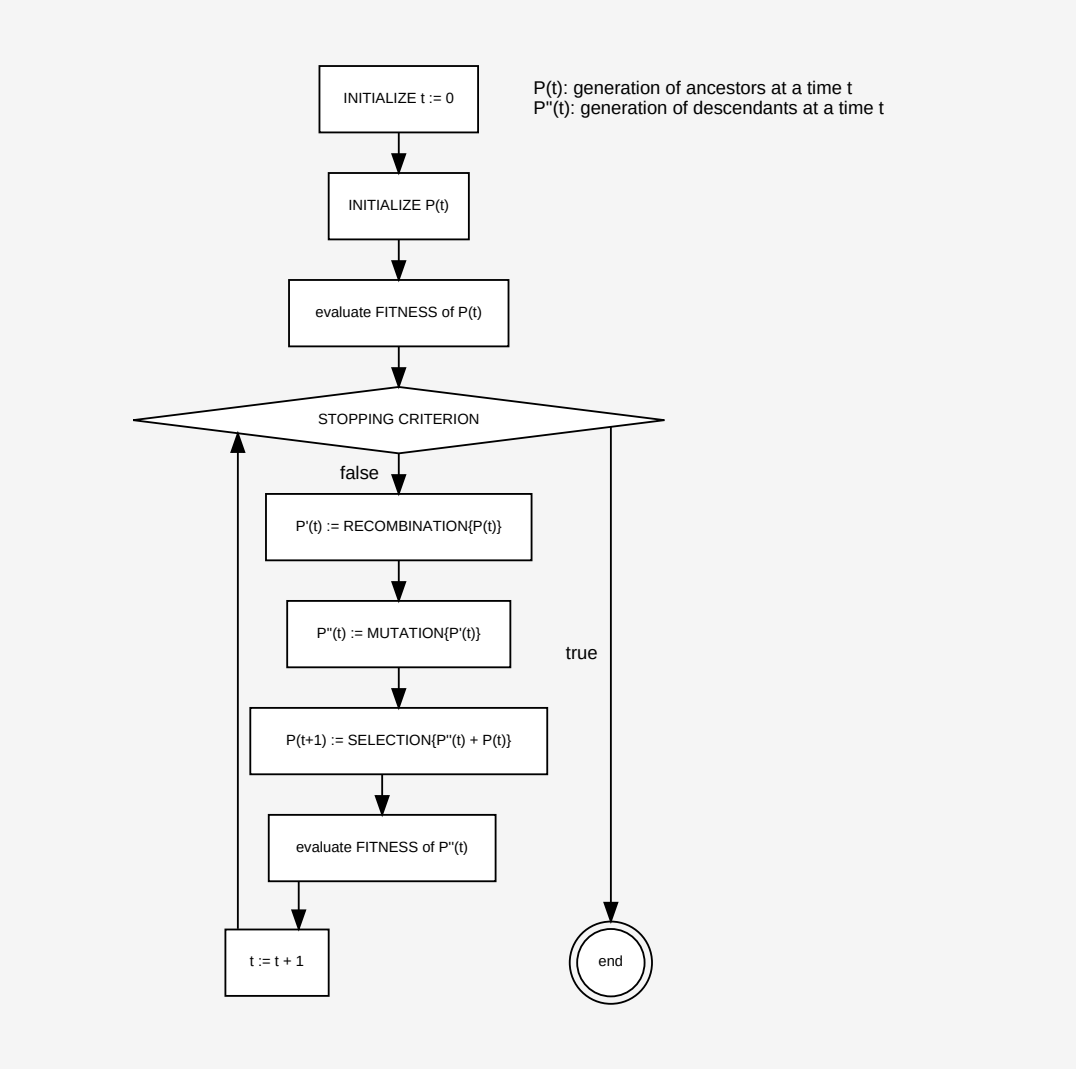
62.2. Algorithmes génétiques

L'algorithme génétique (GA) est une méthode d'optimisation heuristique qui opère par recherches aléatoires. L'ensemble des solutions possibles au problème d'optimisation est considéré comme une *population* d'*individus*. Le degré d'adaptation d'un individu à son environnement est indiqué par sa *valeur d'adaptation* (*fitness*).

Les coordonnées d'un individu dans l'espace de recherche sont représentées par des *chromosomes*, en fait un ensemble de chaînes de caractères. Un *gène* est une sous-section de chromosome qui code la valeur d'un paramètre simple en cours d'optimisation. Les codages habituels d'un gène sont *binary* ou *integer*.

La simulation des opérations d'évolution (*recombinaison*, *mutation* et *sélection*) permet de trouver de nouvelles générations de points de recherche qui présentent une meilleure adaptation moyenne que leurs ancêtres. Figure 62.1 illustre ces étapes.

Figure 62.1. Structure d'un algorithme génétique



Selon la FAQ de comp.ai.genetic, on ne peut pas réellement affirmer qu'un GA n'est pas purement une recherche aléatoire. Un GA utilise des processus stochastiques, mais le résultat est assurément non-aléatoire (il est mieux qu'aléatoire).

62.3. Optimisation génétique des requêtes (GEQO) dans PostgreSQL

Le module GEQO utilise une approche du problème d'optimisation des requêtes similaire à celui du voyageur de commerce (TSP). Les plans de requêtes possibles sont codés comme des chaînes d'entiers. Chaque chaîne représente l'ordre de jointure d'une relation de la requête à une autre. Par exemple, l'arbre de jointure

```
/\
/\ 2
/\ 3
4  1
```

est codé avec la chaîne d'entiers '4-1-3-2', ce qui signifie : première jointure entre les relations '4' et '1', puis '3' et enfin '2', avec 1, 2, 3, 4 les identifiants des relations pour l'optimiseur de PostgreSQL.

Les caractéristiques spécifiques de l'implantation du GEQO sont :

- l'utilisation d'un algorithme génétique monostable (ou à état stable) (remplacement des individus les moins ciblés au lieu d'un remplacement global de génération) permet une convergence rapide vers des plans de requêtes améliorés ; c'est indispensable au traitement des requêtes dans un temps raisonnable ;
- l'utilisation de croisements (recombinaisons) aux limites est particulièrement adapté pour la restriction des pertes aux limites lors de la résolution du problème du voyageur de commerce par un algorithme génétique ;
- la mutation en tant qu'opérateur génétique est rendue obsolète afin d'éviter la nécessité de mécanismes de réparation lors de la génération de tournées valides du problème du voyageur de commerce.

Diverses parties du module GEQO sont adaptées de l'algorithme Genitor de D. Whitley.

Le module GEQO permet à l'optimiseur de requêtes de PostgreSQL de supporter les requêtes disposant de jointures importantes de manière efficace via une recherche non exhaustive.

62.3.1. Génération par le GEQO des plans envisageables

Le processus de planification du GEQO utilise le code standard du planificateur pour créer les plans de parcours des relations individuelles. Les plans de jointure sont alors développés à l'aide de l'approche génétique. Comme décrit plus bas, chaque plan de jointure candidat est représenté par une séquence à laquelle joindre les relations de base. Lors de l'étape initiale, l'algorithme produit simplement quelques séquences de jointure aléatoirement. Pour chaque séquence considérée, le code du planificateur standard est invoqué pour estimer le coût de la requête à l'aide de cette séquence. (Pour chaque étape de la séquence, les trois stratégies de jointure sont considérées ; et tous les plans de parcours initiaux sont disponibles. Le coût estimé est le moins coûteux.) Les séquences dont le coût est moindre sont considérées « plus adaptées » que celle de coût plus élevé. L'algorithme génétique élimine les candidats les moins adaptés. De nouveaux candidats sont alors engendrés par combinaison de gènes de candidats à forte valeur d'adaptation -- par l'utilisation de portions aléatoires de plans peu coûteux pour créer de nouvelles séquences. Ce processus est répété jusqu'à ce qu'un nombre prédéterminé de séquences aient été considérées ; la meilleure séquence rencontrée pendant la recherche est utilisée pour produire le plan final.

Ce processus est intrinsèquement non-déterministe, du fait des choix aléatoires effectués lors de la sélection initiale de la population et lors des « mutations » des meilleurs candidats qui s'en suivent. Pour éviter des modifications surprenantes du plan sélectionné, chaque exécution de l'algorithme relance son générateur aléatoire de numéros avec le paramètre geqo_seed. Tant que `geqo_seed` et les autres paramètres GEQO sont fixes, le même plan sera généré pour une même requête (ainsi que pour certaines informations du planificateur comme les statistiques). Pour expérimenter différents chemins de recherche, modifiez `geqo_seed`.

62.3.2. Tâches à réaliser pour la future implantation du GEQO

Un gros travail est toujours nécessaire pour améliorer les paramètres de l'algorithme génétique. Dans le fichier `src/backend/optimizer/geqo/geqo_main.c`, pour les routines `gimme_pool_size` et `gimme_number_generations`, il faut trouver un compromis pour que les paramètres satisfassent deux besoins concurrents :

- l'optimisation du plan de requête ;
- le temps de calcul.

Dans l'implantation courante, l'adaptation de chaque séquence de jointure candidate est estimée par l'exécution ab-initio du code standard de sélection de jointure et d'estimation de coût utilisé par

le planificateur. Avec l'hypothèse que différents candidats utilisent des sous-séquences de jointure similaires, une grande partie du travail est répétée. Ce processus peut être grandement accéléré en retenant les estimations de coût des sous-jointures. Le problème consiste à éviter d'étendre inutilement la mémoire en mémorisant ces états.

À un niveau plus basique, il n'est pas certain qu'optimiser une requête avec un algorithme génétique conçu pour le problème du voyageur de commerce soit approprié. Dans le cas du voyageur de commerce, le coût associé à une sous-chaîne quelconque (tour partiel) est indépendant du reste du tour, mais cela n'est certainement plus vrai dans le cas de l'optimisation de requêtes. Du coup, la question reste posée quant au fait que la recombinaison soit la procédure de mutation la plus efficace.

62.4. Lectures supplémentaires

Les ressources suivantes contiennent des informations supplémentaires sur les algorithmes génétiques :

- The Hitch-Hiker's Guide to Evolutionary Computation[1] (FAQ de news://comp.ai.genetic) ;
- Evolutionary Computation and its application to art and design[2], par Craig Reynolds ;
- [elma04]
- [fong]

[1] http://www.faqs.org/faqs/ai-faq/genetic/part1/
[2] https://www.red3d.com/cwr/evolve.html

Chapitre 63. Définition de l'interface des méthodes d'accès aux tables

Ce chapitre explique l'interface entre le système PostgreSQL et les *méthodes d'accès aux tables*, qui gèrent le stockage des tables. Le cœur du système connaît très peu de choses sur ces méthodes d'accès en dehors de ce qui est spécifié ici, donc il est possible de développer de nouvelles méthodes d'accès en écrivant un code supplémentaire.

Chaque méthode d'accès aux tables est décrite par une ligne dans le catalogue système `pg_am`. L'enregistrement dans `pg_am` précise un nom et une *fonction de gestion* pour la méthode d'accès à la table. Ces enregistrements peuvent être créés et supprimés en utilisant les commandes SQL CREATE ACCESS METHOD et DROP ACCESS METHOD.

Une fonction de gestion de la méthode d'accès aux tables doit être déclarée comme acceptant un seul argument de type `internal` et doit renvoyer le pseudo-type `table_am_handler`. L'argument est une valeur bateau qui sert simplement à empêcher l'appel de fonctions de gestion à partir de simples commandes SQL. Le résultat de la fonction doit être un pointeur vers une structure de type `TableAmRoutine`, qui contient tout ce que le cœur du moteur a besoin de savoir pour utiliser la méthode d'accès aux tables. La valeur de retour doit avoir une durée de vie valable pour toute la durée d'exécution du serveur, ce qui se fait habituellement en la définissant comme une variable de type `static const` de façon globale. La structure `TableAmRoutine`, aussi appelée *structure API* de la méthode d'accès, définit le comportement de la méthode d'accès en utilisant des callbacks. Ces callbacks sont des pointeurs vers des fonctions C et ne sont pas visibles ou appelables au niveau du langage SQL. Tous les callbacks et leurs comportements sont définis dans la structure `TableAmRoutine` (avec des commentaires dans le structure définissant les prérequis des callbacks). La plupart des callbacks ont des fonctions wrapper, documentées du point de vue de l'utilisateur (plutôt que de l'implémenteur) de la méthode d'accès aux tables. Pour les détails, voir le fichier `src/include/access/tableam.h`[1].

Pour implémenter une méthode d'accès, un implémenteur aura habituellement besoin d'implémenter un type spécifique AM de ligne (voir `src/include/executor/tuptable.h`[2]), qui permet à du code en dehors de la méthode d'accès de détenir des références sur les lignes de l'AM, et d'accéder aux colonnes de la ligne.

Actuellement, la façon dont un AM enregisrre les données est très peu contrainte. Par exemple, il est possible, mais non requis, d'utiliser le cache disque de PostgreSQL. Au cas où il est utilisé, il est sensé d'utiliser la disposition standard d'une page de PostgreSQL comme décrit dans Section 73.6.

Une importante contrainte de l'API de méthode d'accès aux tables est qu'actuellement, si l'AM veut accepter les modifications et/ou les index, il est nécessaire que chaque ligne ait un identifiant de ligne (TID) consistant en un numéro de bloc et un numéro d'enregistrement (voir aussi Section 73.6). IL n'est pas strictement nécessaire que les sous-parties des TIDs aient cette même signification, avoir un `heap`, mais si le support de parcours de bitmap est souhaité (c'est optionnel), le numéro de bloc doit fournir une localisation.

Pour la sécurité des données, un AM doit utiliser les WAL de PostgreSQL ou une implémentation personnalisée. Si les WAL sont choisis, des enregistrements génériques des WAL peuvent être utilisés. Vous pouvez aussi implémenter un nouveau type d'enregistrements WAL (voir Gestionnaire personnalisé de ressources WAL).

Pour implémenter le support transactionnel d'une façon qui permet à différentes méthodes d'accès aux tables d'être accédées dans une seule transaction, il est généralement nécessaire d'intégrer ce qui se trouve dans `src/backend/access/transam/xlog.c`.

[1] https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/include/access/tableam.h;hb=head
[2] https://git.postgresql.org/gitweb/?p=postgresql.git;a=blob;f=src/include/executor/tuptable.h;hb=head

Tout développeur d'une nouvelle `méthode d'accès aux tables` peut se référer à l'implémentation existante de `heap` dans `src/backend/access/heap/heapam_handler.c` pour les détails de son implémentation.

Chapitre 64. Définition de l'interface des méthodes d'accès aux index

Ce chapitre définit l'interface entre le cœur du système de PostgreSQL et les *méthodes d'accès aux index*, qui gèrent chaque type d'index. Le système principal ne sait rien des index en dehors de ce qui est spécifié ici. Il est donc possible de développer des types d'index entièrement nouveaux en écrivant du code supplémentaire.

Tous les index de PostgreSQL sont techniquement des *index secondaires* ; c'est-à-dire que l'index est séparé physiquement du fichier de la table qu'il décrit. Chaque index est stocké dans sa propre *relation* physique et est donc décrit par une entrée dans le catalogue `pg_class`. Le contenu d'un index est entièrement contrôlé par la méthode d'accès à l'index. En pratique, toutes les méthodes d'accès aux index les divisent en pages de taille standard de façon à utiliser le gestionnaire de stockage et le gestionnaire de tampon pour accéder au contenu de l'index. (De plus, toutes les méthodes existantes d'accès aux index utilisent la disposition de page standard décrite dans Section 73.6 et la plupart ont le même format pour les en-têtes de ligne de l'index ; mais ce ne sont pas des obligations pour toutes les méthodes d'accès.)

Dans les faits, un index est une correspondance entre certaines valeurs de données clés et les identifiants des lignes (*tuple identifiers*, ou TIDs), dans leurs différentes versions, dans la table parente de l'index. Un TID consiste en un numéro de bloc et un numéro d'élément dans ce bloc (voir Section 73.6). L'information est suffisante pour récupérer une version d'une ligne particulière à partir de la table. Les index n'ont pas directement connaissance de l'existence éventuelle, à cause du MVCC, de plusieurs versions de la même ligne logique ; pour un index, chaque ligne est un objet indépendant qui a besoin de sa propre entrée. En conséquence, la mise à jour d'une ligne crée toujours de nouvelles entrées dans l'index pour cette ligne, même si les valeurs de la clé ne changent pas. (Les lignes HOT sont une exception ; mais les index ne s'en occupent pas). Les entrées d'index pour les lignes mortes sont nettoyées (par le VACUUM) lorsque les lignes mortes elles-même sont nettoyées.

64.1. Structure basique de l'API pour les index

Chaque méthode d'accès à un index est décrite par une ligne dans le catalogue système `pg_am`. Elle indique un nom et une *fonction gestionnaire* pour la méthode d'accès à l'index. Ces entrées peuvent être créées et supprimées en utilisant les commandes SQL CREATE ACCESS METHOD et DROP ACCESS METHOD respectivement.

Une fonction gestionnaire de méthode d'accès aux index doit être déclarée avec un seul argument de type `internal` et en retour le pseudo-type `index_am_handler`. L'argument est une valeur sans utilité sinon pour empêcher les fonctions gestionnaires d'être appelées directement à partir d'une commande SQL. Le résultat de la fonction doit être une structure, allouée avec palloc, de type `IndexAmRoutine`, et contenant tout ce que le code interne a besoin de savoir pour utiliser la méthode d'accès à l'index. La structure `IndexAmRoutine`, aussi appelée *API struct* de la méthode, inclut les champs spécifiant les propriétés fixes de la méthode d'accès, comme le support des index multi-colonnes. Plus important, elle contient les pointeurs vers les fonctions de la méthode d'accès, qui se chargent de tout le travail d'accès aux index. Ces fonctions de support sont de simples fonctions en C et ne sont ni visibles ni appelables au niveau SQL. Elles sont décrites dans Section 64.2.

La structure `IndexAmRoutine` est définie ainsi :

```
typedef struct IndexAmRoutine
{
    NodeTag     type;
```

```
   /*
    * Nombre total de stratégies (opérateurs) par lesquels nous
pouvons
    * traverser la méthode d'accès ou chercher dedans. Zéro si la
méthode
    * n'a pas de jeu de stratégies fixé.
    */
   uint16      amstrategies;
   /* nombre total de fonctions support utilisées par cette
méthode d'accès */
   uint16      amsupport;
   /* opclass options support function number or 0 */
   uint16      amoptsprocnum;
   /* la méthode supporte-t-elle un ORDER BY sur la colonne
indexée ? */
   bool        amcanorder;
   /* la méthode supporte-t-elle un ORDER BY sur le résultat d'un
opérateur appliqué à une colonne indexée ? */
   bool        amcanorderbyop;
   /* la méthode supporte-t-elle le parcours à rebours ? */
   bool        amcanbackward;
   /* la méthode supporte-t-elle les index UNIQUE ? */
   bool        amcanunique;
   /* la méthode supporte-t-elle les index multi-colonnes ? */
   bool        amcanmulticol;
   /* la méthode exige-t-elle un parcours pour une contrainte sur
la première colonne de l'index ? */
   bool        amoptionalkey;
   /* la méthode gère-t-elle les qualificatifs ScalarArrayOpExpr ?
*/
   bool        amsearcharray;
   /* la méthode gère-elle les qualificatifs IS NULL/IS NOT NULL ?
*/
   bool        amsearchnulls;
   /* le type de la valeur dans l'index peut-elle différer du type
de la colonne ? */
   bool        amstorage;
   /* un index de ce type peut-il être la cible de la commande
CLUSTER ? */
   bool        amclusterable;
   /* la méthode gère-t-elle les verrous sur prédicat ? */
   bool        ampredlocks;
   /* la méthode gère-t-elle les parcours parallélisés ? */
   bool        amcanparallel;
   /* la méthode gère-t-elle les colonnes incluses avec la clause
INCLUDE ? */
   bool        amcaninclude;
   /* does AM use maintenance_work_mem? */
   bool        amusemaintenanceworkmem;
   /* OR of parallel vacuum flags */
   uint8       amparallelvacuumoptions;
   /* type de données stocké dans l'index, ou InvalidOid si
variable */
   Oid         amkeytype;

   /* fonctions d'interfaçage */
   ambuild_function ambuild;
   ambuildempty_function ambuildempty;
```

```
    aminsert_function aminsert;
    ambulkdelete_function ambulkdelete;
    amvacuumcleanup_function amvacuumcleanup;
    amcanreturn_function amcanreturn;   /* peut être NULL */
    amcostestimate_function amcostestimate;
    amoptions_function amoptions;
    amproperty_function amproperty;     /* peut être NULL */
    ambuildphasename_function ambuildphasename;   /* peut être NULL
 */
    amvalidate_function amvalidate;
    amadjustmembers_function amadjustmembers; /* peut être NULL */
    ambeginscan_function ambeginscan;
    amrescan_function amrescan;
    amgettuple_function amgettuple;     /* peut être NULL */
    amgetbitmap_function amgetbitmap;   /* peut être NULL */
    amendscan_function amendscan;
    ammarkpos_function ammarkpos;       /* peut être NULL */
    amrestrpos_function amrestrpos;     /* peut être NULL */

    /* interface functions to support parallel index scans */
    amestimateparallelscan_function amestimateparallelscan;    /*
 can be NULL */
    aminitparallelscan_function aminitparallelscan;    /* can be
 NULL */
    amparallelrescan_function amparallelrescan;    /* can be NULL
 */
} IndexAmRoutine;
```

Pour être utile, une méthode d'accès à l'index doit aussi avoir une ou plusieurs *familles d'opérateurs* et *classes d'opérateurs* définies dans `pg_opfamily`, `pg_opclass`, `pg_amop` et `pg_amproc`. Ces entrées permettent au planificateur de déterminer les types de requêtes qui peuvent être utilisés avec les index de cette méthode d'accès. Les familles et classes d'opérateurs sont décrites dans Section 38.16, qui est un élément requis pour comprendre ce chapitre.

Un index individuel est défini par une entrée dans `pg_class` en tant que relation physique, et une entrée dans `pg_index` affichant son contenu logique -- c'est-à-dire ses colonnes et leur sémantique, telles que récupérées par les classes d'opérateurs associées. Les colonnes de l'index (valeurs clés) peuvent être de simples colonnes de la table sous-jacente ou des expressions des lignes. Habituellement, la méthode d'accès ne s'intéresse pas à la provenance des valeurs clés (elles lui arrivent toujours pré-calculées), mais plutôt aux informations de la classe d'opérateurs dans `pg_index`. On peut accéder aux entrées de ces deux catalogues via la structure de données `Relation` passée à toute opération sur l'index.

Certains champs de `IndexAmRoutine` ont des implications peu évidentes. Les besoins de `amcanunique` sont discutés dans Section 64.5. L'option `amcanmulticol` indique que la méthode d'accès supporte les index à clés multi-colonnes alors que `amoptionalkey` autorise des parcours lorsqu'aucune restriction indexable n'est fournie pour la première colonne de l'index. Quand `amcanmulticol` est faux, `amoptionalkey` indique essentiellement que la méthode d'accès autorise les parcours complets de l'index sans clause de restriction. Les méthodes d'accès supportant les colonnes multiples *doivent* supporter les parcours sans restriction sur une ou toutes les colonnes après la première ; néanmoins, elles peuvent imposer une restriction sur la première colonne de l'index, ce qui est signalé par `amoptionalkey` à false . Une raison pour une méthode d'accès d'index d'initialiser `amoptionalkey` à false est de ne pas indexer les valeurs NULL. Comme la plupart des opérateurs indexables sont stricts et ne peuvent donc pas renvoyer true pour des entrées NULL, à première vue on ne voudra pas stocker d'entrées pour les valeurs NULL : un parcours d'index ne peut de toute façon pas les retourner. Néanmoins, cette raison ne vaut pas pour un parcours d'index sans restriction pour une colonne d'index donnée. En pratique, cela signifie que les index avec `amoptionalkey` à true doivent indexer les valeurs NULL, car le planificateur peut décider de les utiliser sans aucune clé de

parcours. Une limite liée : une méthode d'accès qui supporte des colonnes multiples *doit* supporter l'indexation des NULL dans les colonnes après la première, car le planificateur supposera l'index utilisable avec les requêtes ne restreignant pas ces colonnes. Par exemple, considérons un index sur (a,b) et une requête avec `WHERE a = 4`. Le système supposera que l'index est utilisable pour les lignes où `a = 4`, ce qui est faux si l'index omet les lignes où `b` est NULL. Néanmoins, on peut omettre les lignes où la première colonne indexée est NULL. Du coup, une méthode d'accès d'index ne s'occupant pas des valeurs NULL peut aussi affecter `amsearchnulls` à true, indiquant ainsi qu'elle supporte les clauses `IS NULL` et `IS NOT NULL` dans les conditions de recherche.

L'option `amcaninclude` indique si la méthode d'accès supporte les colonnes « included », c'est-à-dire qu'elle peut enregistrer (sans traiter) des colonnes supplémentaires en dehors des colonnes clés. En particulier, la combinaison de `amcanmulticol=false` et de `amcaninclude=true` est sensible : cela signifie qu'il peut seulement y avoir une colonne clé, mais qu'il peut y avoir une ou plusieurs colonnes inclues. De plus, les colonnes inclues doivent être autorisées à être configurées à NULL, indépendemment de `amoptionalkey`.

64.2. Fonctions des méthodes d'accès aux index

Les fonctions de construction et de maintenance que doit fournir une méthode d'accès aux index dans `IndexAmRoutine` sont :

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

Construit un nouvel index. La relation de l'index a été créée physiquement mais elle est vide. Elle doit être remplie avec toute donnée figée nécessaire à la méthode d'accès, ainsi que les entrées pour toutes les lignes existant déjà dans la table. Habituellement, la fonction `ambuild` appelle `table_index_build_scan()` pour parcourir la table à la recherche des lignes existantes et calculer les clés à insérer dans l'index. La fonction doit renvoyer une structure allouée par palloc contenant les statistiques du nouvel index.

```
bool
void
ambuildempty (Relation indexRelation);
```

Construit un index vide et l'écrit dans le fichier d'initialisation (`INIT_FORKNUM`) de la relation. Cette méthode n'est appelée que pour les index non journalisés ; l'index vide écrit dans ce fichier écrasera le fichier de la relation à chaque redémarrage du serveur.

```
bool
aminsert (Relation indexRelation,
          Datum *values,
          bool *isnull,
          ItemPointer heap_tid,
          Relation heapRelation,
          IndexUniqueCheck checkUnique,
          IndexInfo *indexInfo);
```

Insère une nouvelle ligne dans un index existant. Les tableaux `values` et `isnull` donnent les valeurs de clés à indexer et `heap_tid` est le TID à indexer. Si la méthode d'accès supporte les index uniques (son drapeau `amcanunique` vaut true), alors `checkUnique` indique le type de vérification

d'unicité nécessaire. Elle varie si la contrainte unique est déferrable ou non ; voir Section 64.5 pour les détails. Normalement, la méthode d'accès a seulement besoin du paramètre `heapRelation` lors de la vérification d'unicité (car elle doit vérifier la visibilité de la ligne dans la table).

La valeur booléenne `indexUnchanged` donne une idée sur la nature de la ligne à indexer. Quand elle vaut true, la ligne est un duplicat d'une ligne existant dans l'index. La nouvelle ligne est une version ultérieure de ligne MVCC logiquement non modifiée. Ceci survient quand un `UPDATE` a lieu et qu'il ne modifie par les colonnes couvertes par l'index, mais qu'une nouvelle version est malgré tout nécessaire. La méthode d'accès de l'index peut utiliser cette information pour décider d'application la suppression du bas vers le haut dans certaines parties de l'index où de nombreuses versions de la même ligne logique s'accumulent. Notez que la mise à jour d'une colonne non clé ou d'une colonne qui apparaît seulement dans un prédicat d'index partiel n'affecte pas la valeur de `indexUnchanged`. Le code détermine la valeur de `indexUnchanged` de chaque ligne utilisant une approche avec une surcharge faible qui permet à la fois les faux positifs et les faux négatifs. Les méthodes d'accès des index ne doivent pas traiter `indexUnchanged` comme une source d'informations faisant autorité sur la visibilité ou le versionnement des lignes.

La valeur booléenne résultante n'a de sens que si `checkUnique` vaut `UNIQUE_CHECK_PARTIAL`. Dans ce cas, un résultat true signifie que la nouvelle entrée est reconnue comme unique alors que false indique qu'elle pourrait ne pas être unique (et une vérification d'unicité déferrée doit être planifiée). Dans les autres cas, renvoyer une constante false est recommandé.

Certains index peuvent ne pas indexer toutes les lignes. Si la ligne ne doit pas être indexée, `aminsert` devrait s'achever sans rien faire.

Si l'AM de l'index souhaite mettre en cache des données entre plusieurs insertions successives dans l'index au sein d'un même ordre SQL, il peut allouer de l'espace dans `indexInfo->ii_Context` et stocker un pointeur vers les données dans `indexInfo->ii_AmCache` (qui sera initialement NULL).

```
IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);
```

Supprime un ou des tuple(s) de l'index. Il s'agit d'une opération de « suppression en masse » à implémenter par un parcours complet de l'index et la vérification de chaque entrée pour voir si elle doit être supprimée. La fonction `callback` en argument doit être appelée, sous la forme `callback(`*`TID`*`, callback_state) returns bool`, pour déterminer si une entrée d'index particulière, identifiée par son TID, doit être supprimée. Elle doit retourner NULL ou une structure issue d'un palloc contenant des statistiques sur les effets de la suppression. La fonction peut renvoyer NULL si aucune information n'a besoin d'être envoyée à `amvacuumcleanup`.

À cause d'un `maintenance_work_mem` limité, la suppression de nombreux tuples peut nécessiter d'appeler `ambulkdelete` à plusieurs reprises. L'argument `stats` est le résultat du dernier appel pour cet index (il est NULL au premier appel dans une opération `VACUUM`). Ceci permet à la méthode d'accumuler des statistiques sur toute l'opération. Typiquement, `ambulkdelete` modifie et renvoie la même structure si le `stats` fourni n'est pas NULL.

```
IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
                 IndexBulkDeleteResult *stats);
```

Nettoyer après une opération `VACUUM` (zéro à plusieurs appels à `ambulkdelete`). La fonction n'a pas d'autre but que de retourner des statistiques sur les index, mais elle peut réaliser un nettoyage en masse (réclamer les pages d'index vides, par exemple). `stats` est le retour de l'appel

à ambulkdelete, ou NULL si ambulkdelete n'a pas été appelée car aucune ligne n'avait besoin d'être supprimée. Si le résultat n'est pas NULL, il s'agit d'une structure allouée par palloc. Les statistiques qu'elle contient seront utilisées pour mettre à jour pg_class, et sont rapportées par VACUUM si VERBOSE est indiqué. La fonction peut retourner NULL si l'index n'a pas été modifié lors de l'opération de VACUUM mais, dans le cas contraire, il faut retourner des statistiques correctes.

amvacuumcleanup sera aussi appelé à la fin d'une opération ANALYZE. Dans ce cas, stats vaut toujours NULL et toute valeur de retour sera ignorée. Ce cas peut être repéré en vérifiant info->analyze_only. Il est recommandé que la méthode d'accès ne fasse rien en dehors du nettoyage après insertion pour ce type d'appel, et cela seulement au sein d'un processus autovacuum.

```
bool
amcanreturn (Relation indexRelation, int attno);
```

Vérifie si l'index supporte les *parcours d'index seul* sur la colonne indiquée, en renvoyant la valeur originale indexée de la colonne. Le numéro d'attribut est indexé à partir de 1, c'est-à-dire que le champ attno de la première colonne est 1. Renvoie true si supporté, false sinon. Cette fonction renvoie toujours true pour les colonnes inclues (si elles sont supportées) car il y a peu d'intérêt à une colonne inclue qui ne peut pas être récupérée. Si la méthode d'accès ne supporte pas du tout les parcours d'index seul, le champ amcanreturn de la structure IndexAmRoutine peut être mis à NULL.

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

Estime les coûts d'un parcours d'index. Cette fonction est décrite complètement dans Section 64.6 ci-dessous.

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

Analyse et valide le tableau reloptions d'un index. Cette fonction n'est appelée que lorsqu'il existe un tableau reloptions non NULL pour l'index. *reloptions* est un tableau avec des entrées de type text et de la forme *nom=valeur*. La fonction doit construire une valeur de type bytea à copier dans le rd_options de l'entrée relcache de l'index. Les données contenues dans le bytea sont définies par la méthode d'accès. La plupart des méthodes d'accès standard utilisent la structure StdRdOptions. Lorsque *validate* est true, la fonction doit remonter un message d'erreur clair si une des options n'est pas reconnue ou a des valeurs invalides ; quand *validate* est false, les entrées invalides sont ignorées silencieusement. (*validate* est faux lors du chargement d'options déjà stockées dans pg_catalog ; une entrée invalide ne peut se trouver que si la méthode d'accès a modifié ses règles pour les options et, dans ce cas, il faut ignorer les entrées obsolètes.) Pour obtenir le comportement par défaut, il suffit de retourner NULL.

```
bool
amproperty (Oid index_oid, int attno,
```

```
IndexAMProperty prop, const char *propname,
bool *res, bool *isnull);
```

Permet aux méthodes d'accès de surcharger le comportement par défaut de `pg_index_column_has_property` et des fonctions liées. Si la méthode d'accès n'a pas de comportement spécial lors des demandes de propriétés d'index, le champ `amproperty` dans sa structure `IndexAmRoutine` peut être NULL. Dans le cas contraire, la méthode `amproperty` sera appelée avec *`index_oid`* et *`attno`* tous les deux à zéro pour les appels à `pg_indexam_has_property`, ou avec un *`index_oid`* valide et *`attno`* à zéro pour les appels à `pg_index_has_property`, ou avec un *`index_oid`* valide et un *`attno`* supérieur à zéro pour les appels à `pg_index_column_has_property`. *`prop`* est une valeur enum identifiant la propriété testée, alors que *`propname`* est le nom de la propriété originale. Si le code principal ne reconnaît pas le nom de la propriété, alors *`prop`* vaut `AMPROP_UNKNOWN`. Les méthodes d'accès peuvent définir les noms de propriétés personnalisées en cherchant une correspondance avec *`propname`* (utilisez `pg_strcasecmp` pour être cohérent avec le code principal) ; pour les noms connus du code principal, il est préférable d'inspecter *`prop`*. Si la méthode `amproperty` renvoie `true`, alors elle a passé le test de propriété : elle doit renvoyer le booléen `*res` ou mettre `*isnull` à `true` pour renvoyer un NULL. (Les deux variables référencées sont initialisées à `false` avant l'appel.) Si la méthode `amproperty` renvoie `false`, alors le code principal continuera avec sa logique habituelle pour tester la propriété.

Les méthodes d'accès supportant les opérateurs de tri devraient implémenter le test de `AMPROP_DISTANCE_ORDERABLE` car le code principal ne sait pas le faire et renverra NULL. Il pourrait aussi être avantageux d'implémenter un test sur `AMPROP_RETURNABLE` si cela peut être fait de façon plus simple que d'ouvrir l'index et d'appeler `amcanreturn`, comme le fait le code principal. Le comportement par défaut devrait être satisfaisant pour toutes les autres propriétés standard.

```
char *
ambuildphasename (int64 phasenum);
```

Retourne le nom textuel du numéro de phase de construction donnée. Les numéros de phase sont ceux reportés durant une construction d'index via l'interface `pgstat_progress_update_param`. Les noms de phase sont ensuite exposés dans la vue `pg_stat_progress_create_index`.

```
bool
amvalidate (Oid opclassoid);
```

Valide les entrées du catalogue pour la classe d'opérateurs indiquée, à condition que la méthode d'accès puisse le faire raisonnablement. Par exemple, ceci pourrait inclure le test de la présence de toutes les fonctions de support. La fonction `amvalidate` doit renvoyer false si la classe d'opérateurs est invalide. Les problèmes devraient être rapportés avec les messages `ereport`, typiquement au niveau `INFO`.

```
void
amadjustmembers (Oid opfamilyoid,
                 Oid opclassoid,
                 List *operators,
                 List *functions);
```

Valide l'opérateur new proposé et les membres fonctions d'une famille d'opérateur, si la méthode d'accès peut raisonnablement le faire, et configure leur types de dépendance si le défaut n'est pas satisfaisant. C'est appelée lors du `CREATE OPERATOR CLASS` et lors du `ALTER OPERATOR FAMILY ADD` ; dans ce dernier cas, *`opclassoid`* vaut `InvalidOid`. Les arguments `List`

sont des listes de structures `OpFamilyMember`, comme définies dans `amapi.h`. Les tests réalisés par cette fonction seront typiquement un sous-ensemble de ceux réalisés par `amvalidate`, parce que `amadjustmembers` ne peut pas assumer qu'elle voit un ensemble complet de membres. Par exemple, il serait raisonnable de vérifier la signature d'une fonction d'appui, mais pas de vérifier si toutes les fonctions d'appui requises sont fournies. Tout problème peut être rapporté en renvoyant une erreur. Les champs relatifs aux dépendances des structures `OpFamilyMember` sont initialisés par le cœur de PostgreSQL pour créer les dépendances dures sur la classe d'opérateur si c'est `CREATE OPERATOR CLASS`, ou des dépendances douces sur la famille d'opérateur si c'est `ALTER OPERATOR FAMILY ADD`. `amadjustmembers` peut ajuster ces champs si d'autres comportements sont plus appropriés. Par exemple, GIN, GiST et SP-GiST configurent toujours les membres opérateurs pour avoir des dépendances douces sur la famille d'opérateur car la connexion entre un opérateur et une classe d'opérateurs est relativement faible dans ces types d'index ; donc il est raisonnable de permettre l'ajout et la suppression libres de membres d'opérateurs. Les fonctions d'appui optionnelles ont typiquement des dépendances douces, pour être supprimées si nécessaire.

Le but d'un index est bien sûr de supporter les parcours de lignes qui correspondent à une condition `WHERE` indexable, souvent appelée *qualificateur* (*qualifier*) ou *clé de parcours* (*scan key*). La sémantique du parcours d'index est décrite plus complètement dans Section 64.3, ci-dessous. Une méthode d'accès à l'index peut supporter les parcours d'accès standards (« plain index scan »), les parcours d'index « bitmap » ou les deux. Les fonctions liées au parcours qu'une méthode d'accès à l'index doit ou devrait fournir sont :

```
IndexScanDesc
ambeginscan (Relation indexRelation,
             int nkeys,
             int norderbys);
```

Prépare un parcours d'index. Les paramètres `nkeys` et `norderbys` indiquent le nombre de qualificateurs et d'opérateurs de tri qui seront utilisés dans le parcours. Ils peuvent servir pour l'allocation d'espace. Notez que les valeurs réelles des clés de parcours ne sont pas encore fournies. Le résultat doit être une structure palloc. Pour des raisons d'implémentation, la méthode d'accès à l'index *doit* créer cette structure en appelant `RelationGetIndexScan()`. Dans la plupart des cas, `ambeginscan` ne fait pas grand-chose d'autre que cet appel et parfois l'acquisition de verrous ; les parties intéressantes du début du parcours sont dans `amrescan`.

```
void
amrescan (IndexScanDesc scan,
          ScanKey keys,
          int nkeys,
          ScanKey orderbys,
          int norderbys);
```

Démarre ou relance un parcours d'index, possiblement avec de nouvelles clés d'index. (Pour relancer en utilisant des clés déjà passées, passer NULL à `keys` et/ou `orderbys`.) Notez que le nombre de clés ou d'opérateurs de tri ne doit pas être plus grand que ce qui a été passé à la fonction `ambeginscan`. En pratique, le relancement est utilisé quand une nouvelle ligne externe est sélectionnée par une jointure de boucle imbriquée, donc avec une nouvelle valeur de comparaison, mais la structure de clé de parcours reste la même.

```
bool
amgettuple (IndexScanDesc scan,
            ScanDirection direction);
```

Récupérer la prochaine ligne d'un parcours donné, dans la direction donnée (vers l'avant ou l'arrière de l'index). Renvoie true si une ligne a été obtenue, false s'il ne reste aucune ligne. Dans le cas true, le

TID de la ligne est stocké dans la structure `scan`. « success » signifie juste que l'index contient une entrée qui correspond aux clés de parcours, pas que la ligne existe toujours dans la table ou qu'elle sera visible dans l'instantané (*snapshot*) de l'appelant. En cas de succès, `amgettuple` doit passer `scan->xs_recheck` à true ou false. True signifie que ce n'est pas certain et que les conditions représentées par les clés de parcours doivent être de nouveau vérifiées sur la ligne dans la table après récupération. Cette différence permet de supporter les opérateurs d'index « à perte ». Notez que cela ne s'appliquera qu'aux conditions de parcours ; un prédicat partiel d'index n'est jamais revérifié par les appelants à `amgettuple`.

Si l'index supporte les parcours d'index seul (c'est-à-dire que `amcanreturn` renvoie true pour chacune de ces colonnes), alors, en cas de succès, la méthode d'accès doit aussi vérifier `scan->xs_want_itup`, et si ce dernier est true, elle doit renvoyer les données indexées originales de cette entrée d'index. Les colonnes pour lesquelles `amcanreturn` renvoie false peuvent êre renvoyées comme NULL. Les données peuvent être retournées sous la forme d'un pointeur d'`IndexTuple` stocké dans `scan->xs_itup`, avec un descripteur de lignes dans `scan->xs_itupdesc`; ou sous la forme d'un pointeur `HeapTuple` stocké dans `scan->xs_hitup`, avec le descripteur de ligne `scan->xs_hitupdesc`. (Le second format devrait être utilisé lors de la reconstruction des données qui pourraient ne pas tenir dans un `IndexTuple`.) Dans tous les cas, la gestion de la donnée référencée par le pointeur est de la responsabilité de la méthode d'accès. Les données doivent rester bonnes au moins jusqu'au prochain appel à `amgettuple`, `amrescan` ou `amendscan` pour le parcours.

La fonction `amgettuple` a seulement besoin d'exister si la méthode d'accès supporte les parcours d'index standards. Si ce n'est pas le cas, le champ `amgettuple` de la structure `IndexAmRoutine` doit être NULL.

```
int64
amgetbitmap (IndexScanDesc scan,
             TIDBitmap *tbm);
```

Récupère toutes les lignes du parcours sélectionné et les ajoute au `TIDBitmap` fourni par l'appelant (c'est-à-dire un OU de l'ensemble des identifiants de ligne dans l'ensemble où se trouve déjà le bitmap). Le nombre de lignes récupérées est renvoyé (cela peut n'être qu'une estimation car certaines méthodes d'accès ne détectent pas les duplicats). Lors de l'insertion d'identifiants de ligne dans le bitmap, `amgetbitmap` peut indiquer que la vérification des conditions du parcours est requis pour des identifiants précis de transactions. C'est identique au paramètre de sortie `xs_recheck` de `amgettuple`. Note : dans l'implémentation actuelle, le support de cette fonctionnalité est fusionné avec le support du stockage à perte du bitmap lui-même, et du coup les appelants revérifient à la fois les conditions du parcours et le prédicat de l'index partiel (si c'en est un) pour les lignes à revérifier. Cela ne sera pas forcément toujours vrai. `amgetbitmap` et `amgettuple` ne peuvent pas être utilisés dans le même parcours d'index ; il existe d'autres restrictions lors de l'utilisation de `amgetbitmap`, comme expliqué dans Section 64.3.

En plus de supporter des parcours d'index ordinaires, certains types d'index peuvent souhaiter supporter des *parcours d'index parallèle*, qui permettent à de multiples processus clients de coopérer afin de réaliser un parcours d'index. La méthode d'accès à l'index devrait s'arranger pour que chaque processus participant au parcours retourne un sous-ensemble des lignes qui devraient être traitées par un parcours d'index ordinaire, non parallèle, mais de telle façon que l'union de tous ces sous ensembles soit identique aux ensemble de lignes qui seraient retournés par un parcours d'index ordinaire, non parallèle. En outre, bien qu'il n'y ait pas besoin d'un ordre de tri global des lignes retournée par un parcours parallèle, l'ordre du sous ensemble de lignes retourné par chaque processus participant au parcours d'index doit correspondre à l'ordre demandé. Les fonctions suivantes peuvent être implémentée pour supporter les parcours d'index parallèles :

```
Size
amestimateparallelscan (void);
```

Estime et retourne le nombre d'octets de mémoire partagée dynamique dont la méthode d'accès aura besoin pour effectuer le parcours d'index. (Ce chiffre est en plus, et non à la place, de la quantité d'espace nécessaire pour les données indépendantes de l'AM dans `ParallelIndexScanDescData`.)

Il n'est pas nécessaire d'implémenter cette fonction pour les méthodes d'accès qui ne supportent pas les parcours d'index parallèles, où pour lesquelles le nombre d'octets de stockage additionnels vaut zéro.

```
void
aminitparallelscan (void *target);
```

Cette fonction sera appelée pour initialiser la mémoire partagée dynamique au début du parcours parallèle. *`target`* pointera vers au moins le nombre d'octets précédemment retourné par `amestimateparallelscan`, et cette fonction pourra utiliser cette quantité d'espace pour stocker n'importe quelle donnée dont elle a besoin.

Il n'est pas nécessaire d'implémenter cette fonction pour les méthodes d'accès qui ne spportent pas les parcours d'index parallèles ou dans le cas où l'espace de mémoire partagé requis ne nécessite pas d'initialisation.

```
void
amparallelrescan (IndexScanDesc scan);
```

Si implémentée, cette fonction sera appelée lorsqu'un parcours d'index parallèle doit recommencer. Elle devrait réinitialiser tout état partagé mis en place par `aminitparallelscan` de telle manière à ce que le parcours sera recommencé depuis le début.

La fonction `amgetbitmap` ne doit exister que si la méthode d'accès supporte les parcours d'index « bitmap ». Dans le cas contraire, le champ `amgetbitmap` de la structure `IndexAmRoutine` doit être NULL.

```
void
amendscan (IndexScanDesc scan);
```

Terminer un parcours et libérer les ressources. La structure `scan` elle-même ne doit pas être libérée, mais tout verrou posé en interne par la méthode d'accès doit être libéré, ainsi qu'à tout autre mémoire allouée par `ambeginscan` et les autres fonctions relatives aux parcours.

```
void
ammarkpos (IndexScanDesc scan);
```

Marquer la position courante du parcours. La méthode d'accès n'a besoin de mémoriser qu'une seule position par parcours.

La fonction `ammarkpos` n'a besoin d'être fournie que si la méthode supporte les parcours ordonnés. Dans le cas contraire, le champ `ammarkpos` dans sa structure `IndexAmRoutine` peut être NULL.

```
void
amrestrpos (IndexScanDesc scan);
```

Restaurer le parcours à sa plus récente position marquée.

64.3. Parcours d'index

Dans un parcours d'index, la responsabilité de la méthode d'accès est de recracher tous les TID de toutes les lignes qu'on lui a dit correspondre aux *clés de parcours*. La méthode d'accès n'est impliquée *ni* dans la récupération de ces lignes dans la table parente de l'index, *ni* dans les tests de qualification ou autre condition.

Une clé de parcours est une représentation interne d'une clause `WHERE` de la forme *`clé_index opérateur constante`*, où la clé est une des colonnes de l'index et l'opérateur un des membres de la famille d'opérateurs associée à cette colonne. Un parcours d'index a entre zéro et plusieurs clés de parcours assemblées implicitement avec des AND -- les lignes renvoyées doivent satisfaire toutes les conditions indiquées.

La méthode d'accès peut indiquer que l'index est *à perte* ou nécessite une vérification pour une requête particulière ; ceci implique que le parcours d'index renvoie toutes les entrées qui correspondent à la clé de parcours, plus éventuellement des entrées supplémentaires qui ne correspondent pas. La machinerie du parcours d'index du système principal applique alors les conditions de l'index à la ligne pour vérifier si elle doit effectivement être retenue. Si l'option de vérification n'est pas indiquée, le parcours d'index doit renvoyer exactement l'ensemble d'entrées correspondantes.

La méthode d'accès doit s'assurer qu'elle trouve correctement toutes les entrées correspondantes aux clés de parcours données, et seulement celles-ci. De plus, le système principal se contente de transférer toutes les clauses `WHERE` qui correspondent aux clés d'index et aux familles d'opérateurs, sans analyse sémantique permettant de déterminer si elles sont redondantes ou contradictoires. Par exemple, avec `WHERE x > 4 AND x > 14`, où `x` est une colonne indexée par B-tree, c'est à la fonction B-tree `amrescan` de déterminer que la première clé de parcours est redondante et peut être annulée. Le supplément de pré-traitement nécessaire lors de `amrescan` dépend du niveau de réduction des clés de parcours en une forme « normalisée » nécessaire à la méthode d'accès à l'index.

Certaines méthodes d'accès renvoient des entrées d'index dans un ordre bien défini, d'autres non. Il existe en fait deux façons différentes permettant à une méthode d'accès de fournir une sortie triée :

- Les méthodes d'accès qui renvoient toujours les entrées dans l'ordre naturel des données (comme les B-tree) doivent configurer `amcanorder` à true. Actuellement, ces méthodes d'accès doivent utiliser des nombres de stratégie compatibles avec les B-tree pour les opérateurs d'égalité et de tri.

- Les méthodes d'accès qui supportent les opérateurs de tri doivent configurer `amcanorderbyop` à true. Ceci indique que l'index est capable de renvoyer les entrées dans un ordre satisfaisant `ORDER BY` *`clé_index opérateur constante`*. Les modificateurs de parcours de cette forme peuvent être passés à `amrescan` comme décrit précédemment.

La fonction `amgettuple` dispose d'un argument `direction`, qui peut être soit `ForwardScanDirection` (le cas normal), soit `BackwardScanDirection`. Si le premier appel après `amrescan` précise `BackwardScanDirection`, alors l'ensemble des entrées d'index correspondantes est à parcourir de l'arrière vers l'avant plutôt que dans la direction normale (d'avant en arrière). `amgettuple` doit donc renvoyer la dernière ligne correspondante dans l'index, plutôt que la première, comme cela se fait normalement. (Cela ne survient que pour les méthodes d'accès qui initialise `amcanorder` à true.) Après le premier appel, `amgettuple` doit être préparé pour continuer le parcours dans la direction adaptée à partir de l'entrée la plus récemment renvoyée. (Mais si `amcanbackward` vaut false, tous les appels suivants auront la même direction que le premier.)

Les méthodes d'accès qui supportent les parcours ordonnés doivent supporter le « marquage » d'une position dans un parcours pour retourner plus tard à la position marquée. La même position peut être restaurée plusieurs fois. Néanmoins, seule une position par parcours a besoin d'être conservée en mémoire ; un nouvel appel à `ammarkpos` surcharge la position anciennement marquée. Une méthode d'accès qui ne supporte pas les parcours ordonnés n'a pas besoin de fournir les fonctions `ammarkpos` et `amrestrpos` dans sa structure `IndexAmRoutine` ; configurez ces pointeurs à NULL dans ce cas.

Les positions du parcours et du marquage doivent être conservées de façon cohérente dans le cas d'insertions et de suppressions concurrentes dans l'index. Il est acceptable qu'une entrée tout juste insérée ne soit pas retournée par un parcours qui l'aurait trouvée si l'entrée avait existé au démarrage du parcours. De même est-il correct qu'un parcours retourne une telle entrée lors d'un re-parcours ou d'un retour arrière, alors même qu'il ne l'a pas retournée lors du parcours initial. De même, une suppression concurrente peut être, ou non, visible dans les résultats d'un parcours. Il est primordial qu'insertions et suppressions ne conduisent pas le parcours à oublier ou dupliquer des entrées qui ne sont pas insérées ou supprimées.

Si l'index stocke les valeurs originales des données indexées (et pas une représentation à perte), il est utile de supporter les parcours d'index seul, pour lesquels l'index renvoie la donnée réelle et non pas juste le TID de la ligne dans la table. Ceci n'évitera des I/O que si la carte de visibilité montre que le TID est sur une page dont toutes les lignes sont visibles par toutes les transactions en cours. Sinon, la ligne de la table doit être visitée de toute façon pour s'assurer de sa visibilité pour la transaction en cours. Mais cela ne concerne pas la méthode d'accès.

Un parcours d'index peut utiliser `amgetbitmap` à la place de `amgettuple` pour récupérer toutes les lignes en un unique appel. Cette méthode peut s'avérer nettement plus efficace que `amgettuple` parce qu'elle permet d'éviter les cycles de verrouillage/déverrouillage à l'intérieur de la méthode d'accès. En principe, `amgetbitmap` a les mêmes effets que des appels répétés à `amgettuple`, mais plusieurs restrictions ont été imposées pour simplifier la procédure. En premier lieu, `amgetbitmap` renvoie toutes les lignes en une fois et le marquage ou la restauration des positions de parcours n'est pas supporté. Ensuite, les lignes sont renvoyées dans un bitmap qui n'a pas d'ordre spécifique, ce qui explique pourquoi `amgetbitmap` ne prend pas de `direction` en argument. (Les opérateurs de tri ne seront jamais fournis non plus pour un tel parcours.) De plus, il n'existe aucune disposition pour les parcours d'index seul avec `amgetbitmap` car il n'y a aucun moyen de renvoyer le contenu des lignes d'index. Enfin, `amgetbitmap` ne garantit pas le verrouillage des lignes renvoyées, avec les implications précisées dans Section 64.4.

Notez qu'il est permis à une méthode d'accès d'implémenter seulement `amgetbitmap` et pas `amgettuple`, ou vice versa, si son fonctionnement interne ne convient qu'à une seule des API.

64.4. Considérations sur le verrouillage d'index

Les méthodes d'accès aux index doivent gérer des mises à jour concurrentes de l'index par plusieurs processus. Le système principal de PostgreSQL obtient `AccessShareLock` sur l'index lors d'un parcours d'index et `RowExclusiveLock` lors de sa mise à jour (ce qui inclut le `VACUUM` simple). Comme ces types de verrous ne sont pas conflictuels, la méthode d'accès est responsable de la finesse du verrouillage dont elle a besoin. Un verrou de type `ACCESS EXCLUSIVE` sur l'intégralité de l'index entier n'est posé qu'à la création de l'index, sa destruction ou lors d'un `REINDEX` (un verrou `SHARE UPDATE EXCLUSIVE` est pris à la place avec l'option `CONCURRENTLY`).

Construire un type d'index qui supporte les mises à jour concurrentes requiert une analyse complète et subtile. Pour les types d'index B-tree et hash, on peut lire les implications sur les décisions de conception dans `src/backend/access/nbtree/README` et `src/backend/access/hash/README`.

En plus des besoins de cohérence interne de l'index, les mises à jour concurrentes créent des problèmes de cohérence entre la table parente (*heap*) et l'index. Comme PostgreSQL sépare les accès et les mises à jour de la table et ceux de l'index, il existe des fenêtres temporelles pendant lesquelles l'index et l'en-tête peuvent être incohérents. Ce problème est géré avec les règles suivantes :

- une nouvelle entrée dans la table est effectuée avant son entrée dans l'index. (Un parcours d'index concurrent peut alors ne pas voir l'entrée dans la table. Ce n'est pas gênant dans la mesure où un lecteur de l'index ne s'intéresse pas à une ligne non validée. Voir Section 64.5) ;
- lorsqu'une entrée de la table va être supprimée (par `VACUUM`), on doit d'abord supprimer toutes les entrées d'index ;

- un parcours d'index doit maintenir un lien sur la page d'index contenant le dernier élément renvoyé par `amgettuple`, et `ambulkdelete` ne peut supprimer des entrées de pages liées à d'autres processus. La raison figure ci-dessous.

Sans la troisième règle, il serait possible qu'un lecteur d'index voit une entrée dans l'index juste avant qu'elle ne soit supprimée par un `VACUUM` et arrive à l'entrée correspondante de la table après sa suppression par le `VACUUM`. Cela ne pose aucun problème sérieux si cet élément est toujours inutilisé quand le lecteur l'atteint, car tout emplacement vide est ignoré par `heap_fetch()`. Mais que se passe-t-il si un troisième moteur a déjà ré-utilisé l'emplacement de l'élément pour quelque chose d'autre ? Lors de l'utilisation d'un instantané (*snapshot*) compatible MVCC, il n'y a pas de problème car le nouvel occupant de l'emplacement est certain d'être trop récent pour apparaître dans l'instantané. En revanche, avec un instantané non-compatible MVCC (tel que `SnapshotAny`), une ligne qui ne correspond pas aux clés de parcours peut être acceptée ou retournée. Ce scénario peut être évité en imposant que les clés de parcours soient re-confrontées à la table dans tous les cas, mais cela est trop coûteux. À la place, un lien sur une page d'index est utilisé comme *proxy* pour indiquer que le lecteur peut être « en route » depuis l'entrée d'index vers l'entrée de table correspondante. Bloquer `ambulkdelete` sur un tel lien assure que `VACUUM` ne peut pas supprimer l'entrée de la table avant que le lecteur n'en ait terminé avec elle. Cette solution est peu coûteuse en temps d'exécution, et n'ajoute de surcharge du fait du blocage que dans les rares cas où il y a vraiment un conflit.

Cette solution requiert que les parcours d'index soient « synchrones » : chaque ligne de la table doit être récupérée immédiatement après récupération de l'entrée d'index correspondante. Cela est coûteux pour plusieurs raisons. Un parcours « asynchrone », où l'on récupère de nombreux TID depuis l'index et où l'on ne visite la table que plus tard, requiert moins de surcharge de verrouillage de l'index et autorise un modèle d'accès à la table plus efficace. D'après l'analyse ci-dessus, l'approche synchrone doit être utilisée pour les instantanés non compatibles avec MVCC, mais un parcours asynchrone est possible pour une requête utilisant un instantané MVCC.

Dans un parcours d'index `amgetbitmap`, la méthode d'accès ne bloque l'index pour aucune des lignes renvoyées. C'est pourquoi de tels parcours ne sont fiables qu'avec les instantanés compatibles MVCC.

Quand le drapeau `ampredlocks` n'est pas en place, tout parcours par cette méthode d'accès au sein d'une transaction sérialisable acquerra un verrou prédicat non bloquant sur l'index complet. Ceci génèrera un conflit de lecture/écriture à l'insertion d'une ligne dans cet index par une transaction sérialisable concurrente. Si certains motifs de tels conflits sont détectés dans un ensemble de transactions sérialisables concurrentes, une de ces transactions peut être annulée pour protéger l'intégrité des données. Quand le drapeau est en place, il indique que la méthode d'accès implémente un verrou prédicat plus fin, qui tend à réduire la fréquence d'annulation de telles requêtes.

64.5. Vérification de l'unicité par les index

PostgreSQL assure les contraintes d'unicité SQL par des *index uniques*, qui sont des index qui refusent des entrées multiples pour un même clé. Une méthode d'accès qui supporte cette fonctionnalité initialise `amcanunique` à true. (À ce jour, seul B-tree le supporte). Les colonnes listées dans la clause `INCLUDE` ne sont pas considérées lors de la vérification d'unicité.

Du fait de MVCC, il est toujours nécessaire de permettre à des entrées dupliquées d'exister physiquement dans un index : elles peuvent faire référence à des versions successives d'une même ligne logique. Nous voulons garantir qu'aucune image MVCC n'inclut deux lignes avec les mêmes clés d'index. Cela se résume aux cas suivants, à vérifier à l'insertion d'une nouvelle ligne dans un index d'unicité :

- si une ligne valide conflictuelle a été supprimée par la transaction courante, pas de problème. (En particulier, comme un UPDATE supprime toujours l'ancienne version de la ligne avant d'insérer la nouvelle version, cela permet un UPDATE sur une ligne sans changer la clé) ;

- si une ligne conflictuelle a été insérée par une transaction non encore validée, l'insérteur potentiel doit attendre de voir si la transaction est validée. Si elle est annulée, alors il n'y a pas de conflit. Si elle

est validée sans avoir supprimé la ligne conflictuelle, il y a violation de la contrainte d'unicité. (En pratique, on attend que l'autre transaction finisse et le contrôle de visibilité est effectué à nouveau dans son intégralité) ;

- de façon similaire, si une ligne conflictuelle validée est supprimée par une transaction encore non validée, l'inséreur potentiel doit attendre la validation ou l'annulation de cette transaction et recommencer le test.

De plus, immédiatement avant de lever une violation d'unicité en fonction des règles ci-dessus, la méthode d'accès doit revérifier l'état de la ligne en cours d'insertion. Si elle est validée mais est déjà morte, alors aucune erreur ne survient. (Ce cas ne peut pas survenir lors du scénario ordinaire d'insertion d'une ligne tout juste créée par la transaction en cours. Cela peut néanmoins arriver lors d'un `CREATE UNIQUE INDEX CONCURRENTLY`.)

La méthode d'accès à l'index doit appliquer elle-même ces tests, ce qui signifie qu'elle doit accéder à la table pour vérifier le statut de validation de toute ligne présentant une clé dupliquée au regard du contenu de l'index. C'est sans doute moche et non modulaire, mais cela permet d'éviter un travail redondant : si un test séparé était effectué, alors la recherche d'une ligne conflictuelle dans l'index serait en grande partie répétée lors de la recherche d'une place pour la nouvelle entrée d'index. Qui plus, est, il n'y a pas de façon évidente d'éviter des *race conditions*, sauf si la recherche de conflit est partie intégrante de l'insertion d'une nouvelle entrée d'index.

Si la contrainte unique est déferrable, il y a une complication supplémentaire : nous devons être capable d'insérer une entrée d'index pour une nouvelle ligne mais devons déferrer toute erreur de violation de l'unicité jusqu'à la fin de l'instruction, voire après. Pour éviter des recherches répétées et inutiles dans l'index, la méthode d'accès doit faire une vérification préliminaire d'unicité dès l'insertion initiale. Si elle ne montre pas de conflit avec une ligne visible, nous avons terminé. Sinon, nous devons planifier une nouvelle vérification quand il sera temps de forcer la contrainte. Si, lors de la nouvelle vérification, la ligne insérée et d'autres lignes de la même clé sont vivantes, alors l'erreur doit être rapportée. (Notez que, dans ce contexte, « vivant » signifie réellement « toute ligne dans la chaîne HOT de l'entrée d'index est vivante ».) Pour implanter ceci, la fonction `aminsert` reçoit un paramètre `checkUnique` qui peut avoir une des valeurs suivantes :

- `UNIQUE_CHECK_NO` indique qu'aucun test d'unicité ne doit être fait (ce n'est pas un index unique).
- `UNIQUE_CHECK_YES` indique qu'il s'agit d'un index unique non déferrable et la vérification de l'unicité doit se faire immédiatement, comme décrit ci-dessus.
- `UNIQUE_CHECK_PARTIAL` indique que la contrainte unique est déferrable. PostgreSQL utilisera ce mode pour insérer l'entrée d'index de chaque ligne. La méthode d'accès doit autoriser les entrées dupliquées dans l'index et rapporter tout duplicat potentiel en renvoyant FALSE à partir de `aminsert`. Pour chaque ligne pour laquelle FALSE est renvoyé, une revérification déferrée sera planifiée.

  La méthode d'accès doit identifier toute ligne qui pourrait violer la contrainte unique, mais rapporter des faux positifs n'est pas une erreur. Cela permet de faire la vérification sans attendre la fin des autres transactions ; les conflits rapportés ici ne sont pas traités comme des erreurs, et seront revérifiés plus tard, à un moment où ils ne seront peut-être plus en conflit.
- `UNIQUE_CHECK_EXISTING` indique qu'une revérification déferrée d'une ligne a été rapportée en violation potentielle d'unicité. Bien que cela soit implémenté par un appel à `aminsert`, la méthode d'accès ne doit *pas* insérer une nouvelle entrée d'index dans ce cas. L'entrée d'index est déjà présente. À la place, la méthode d'accès doit vérifier s'il existe une autre entrée d'index vivante. Si c'est le cas et que la ligne cible est toujours vivante, elle doit rapporter l'erreur.

  Il est recommendé que, dans un appel à `UNIQUE_CHECK_EXISTING`, la méthode d'accès vérifie en plus que la ligne cible ait réellement une entrée existante dans l'index et de lever une erreur si ce n'est pas le cas. C'est une bonne idée car les valeurs de la ligne d'index passées à `aminsert` auront été recalculées. Si la définition de l'index implique des fonctions qui ne sont pas vraiment immutables, nous pourrions être en train de vérifier une mauvaise partie de l'index. Vérifier que la

ligne cible est trouvée dans la revérification permet de s'assurer que nous recherchons les mêmes valeurs de la ligne comme elles ont été utilisées lors de l'insertion originale.

64.6. Fonctions d'estimation des coûts d'index

La fonction `amcostestimate` reçoit des informations décrivant un parcours d'index possible, incluant des listes de clauses WHERE et ORDER BY considérées comme utilisables avec l'index. Elle doit renvoyer une estimation du coût de l'accès à l'index et de la sélectivité des clauses WHERE (c'est-à-dire la fraction des lignes de la table parente qui seront récupérées lors du parcours de l'index). Dans les cas simples, pratiquement tout le travail de l'estimateur de coût peut être effectué en appelant des routines standard dans l'optimiseur ; la justification d'une fonction `amcostestimate` est de permettre aux méthodes d'accès de fournir des connaissances spécifiques liées au type d'index, au cas où il serait possible d'améliorer les estimations standards.

Chaque fonction `amcostestimate` doit avoir la signature :

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation,
                double *indexPages);
```

Les trois premiers paramètres sont des entrées :

`root`

Information du planificateur sur la requête en cours de traitement.

`path`

Le chemin d'accès considéré pour l'index. Tous les champs, en dehors du coût et de la sélectivité, sont valides.

`loop_count`

Le nombre de répétitions du parcours d'index à prendre en compte dans les estimations de coût. Il sera généralement supérieur à 1 lors d'un parcours avec paramètres à utiliser à l'intérieur d'une jointure de boucle imbriquée. Notez que l'estimation de coût doit toujours être pour un seul parcours ; une valeur plus importante de *`loop_count`* signifie qu'on pourrait constater l'effet du cache avec plusieurs parcours.

Les cinq derniers paramètres sont les sorties, passées par référence :

`*indexStartupCost`

Renvoie le coût du lancement du traitement de l'index.

`*indexTotalCost`

Renvoie le coût du traitement total de l'index.

`*indexSelectivity`

Renvoie la sélectivité de l'index.

`*indexCorrelation`

Renvoie le coefficient de corrélation entre l'ordre de parcours de l'index et l'ordre sous-jacent de la table.

`*indexPages`

Configuré au nombre de pages feuilles de l'index.

Les fonctions d'estimation de coûts doivent être écrites en C, pas en SQL ou dans tout autre langage procédural, parce qu'elles doivent accéder aux structures de données internes du planificateur/optimiseur.

Les coûts d'accès aux index doivent être calculés avec les paramètres utilisés par `src/backend/optimizer/path/costsize.c` : la récupération d'un bloc disque séquentiel a un coût de `seq_page_cost`, une récupération non séquentielle a un coût de `random_page_cost`, et le coût de traitement d'une ligne d'index doit habituellement être considéré comme `cpu_index_tuple_cost`. De plus, un multiple approprié de `cpu_operator_cost` doit être ajouté pour tous les opérateurs de comparaison impliqués lors du traitement de l'index (spécialement l'évaluation des `indexQuals`).

Les coûts d'accès doivent inclure tous les coûts dus aux disques et aux CPU associés au parcours d'index proprement dit, mais *pas* les coûts de récupération ou de traitement des lignes de la table parente identifiées par l'index.

Le « coût de lancement » est la partie du coût total de parcours à dépenser avant de commencer à récupérer la première ligne. Pour la plupart des index, on peut prendre zéro, mais un type d'index avec un grand coût au démarrage peut nécessiter une valeur supérieure à zéro.

`indexSelectivity` doit être la fraction estimée des lignes de la table parente qui sera récupérée lors du parcours d'index. Dans le cas d'une requête à perte, ce sera typiquement plus élevé que la fraction des lignes qui satisfont les conditions de qualification données.

`indexCorrelation` doit être initialisé à la corrélation (valeur entre -1.0 et 1.0) entre l'ordre de l'index et celui de la table. Cela permet d'ajuster l'estimation du coût de récupération des lignes de la table parente.

`indexPages` doit être configuré au nombre de pages feuilles. Ceci est utilisé pour estimer le nombre de workers pour les parcours d'index parallélisés.

Quand *`loop_count`* est supérieur à 1, les nombres renvoyés doivent être des moyennes attendues pour tout parcours de l'index.

Estimation du coût

Un estimateur typique de coût exécute le traitement ainsi :

1. Estime et renvoie la fraction des lignes de la table parente visitées d'après les conditions de qualification données. En l'absence de toute connaissance spécifique sur le type d'index, on utilise la fonction standard de l'optimiseur `clauselist_selectivity()` :

```
+*indexSelectivity = clauselist_selectivity(root, path-
>indexquals,
                                            path->indexinfo-
>rel->relid,
                                            JOIN_INNER, NULL);
```

2. Estime le nombre de lignes d'index visitées lors du parcours. Pour de nombreux types d'index, il s'agit de *`indexSelectivity`* multiplié par le nombre de lignes dans l'index, mais cela peut

valoir plus (la taille de l'index en pages et lignes est disponible à partir de la structure `path->indexinfo`).

3. Estime le nombre de pages d'index récupérées pendant le parcours. Ceci peut être simplement *`indexSelectivity`* multiplié par la taille en pages de l'index.

4. Calcule le coût d'accès à l'index. Un estimateur générique peut le faire ainsi :

```
    /*
     * On suppose généralement que les pages d'index sont lues
     * séquentiellement, elles coûtent donc seq_page_cost each,
et pas random_page_cost.
     * Nous ajoutons l'évaluation des qualificateurs pour
chaque ligne d'index.
     * Tous les coûts sont supposés être payés de manière
incrémentale pendant le parcours.
     */
    cost_qual_eval(&index_qual_cost, path->indexquals, root);
    *indexStartupCost = index_qual_cost.startup;
    *indexTotalCost = seq_page_cost * numIndexPages +
        (cpu_index_tuple_cost + index_qual_cost.per_tuple) *
numIndexTuples;
```

   Néanmoins, le calcul ci-dessus ne prend pas en compte l'amortissement des lectures des index à travers des parcours répétés.

5. Estime la corrélation de l'index. Pour un index ordonné sur un seul champ, cela peut se trouver dans pg_statistic. Si la corrélation est inconnue, l'estimation conservatrice est zéro (pas de corrélation).

Des exemples de fonctions d'estimation du coût sont disponibles dans `src/backend/utils/adt/selfuncs.c`.

Chapitre 65. Enregistrements génériques des journaux de transactions

Bien que tous les modules internes traçant dans les journaux de transactions ont leur propre type d'enregistrements WAL, il existe aussi un type d'enregistrement générique. Ce type d'enregistrement décrit les modifications de pages d'une façon générique. Ceci est utile pour les extensions fournissant des méthodes d'accès personnalisés.

En comparaison avec les gestionnaires de ressources WAL personnalisées, un WAL générique est plus simple à implémenter pour une extension et ne nécessite pas que la bibliothèque de l'extension soit chargée pour être applicable aux enregistrements.

> **Note**
>
> Les enregistrements WAL génériques sont ignorés lors du décodage logique. Si le décodage logique est requis pour votre extension, pensez à un gestionnaire de ressources WAL personnalisées.

L'API de construction des enregistrements génériques pour les journaux de transactions est définie dans `access/generic_xlog.h` et implémentée dans `access/transam/generic_xlog.c`.

Pour réaliser une mise à jour de données tracée dans les journaux de transactions en utilisant le système d'enregistrement générique, suivez ces étapes :

1. `state = GenericXLogStart(relation)` -- lance la construction d'un enregistrement générique pour la relation spécifiée.

2. `page = GenericXLogRegisterBuffer(state, buffer, flags)` -- enregistre un tampon à modifier dans l'enregistrement générique actuel du journal de transactions. Cette fonction renvoie un pointeur vers une copie temporaire de la page du tampon, où les modifications doivent survenir. (Ne modifiez pas le contenu du tampon.) Le troisième argument est un masque de bits pour les drapeaux applicables à l'opération. Actuellement, le seul drapeau disponible est `GENERIC_XLOG_FULL_IMAGE`, qui indique qu'une image d'une page complète doit être incluse dans l'enregistrement WAL, plutôt qu'un delta. Typiquement, ce drapeau doit être configuré si le bloc est nouveau ou s'il a été complètement réécrit. `GenericXLogRegisterBuffer` peut être répété si l'action tracée doit modifier plusieurs blocs.

3. Réalisez des modifications à l'image des pages obtenue à l'étape précédente.

4. `GenericXLogFinish(state)` -- applique les modifications aux tampons et émet l'enregistrement générique.

La construction d'enregistrements de journaux de transactions peut être annulée entre n'importe laquelle des étapes ci-dessus en appelant la fonction `GenericXLogAbort(state)`. Ceci annulera toutes les modifications aux copies d'image de bloc.

Merci de noter les points suivants lors de l'utilisation de la fonctionnalité d'enregistrements génériques pour les journaux de transactions :

- Aucune modification directe des tampons n'est autorisée ! Toutes les modifications doivent se faire dans les copies récupérées de `GenericXLogRegisterBuffer()`. Autrement dit, le code

réalisant les enregistrements génériques ne doit jamais appeler lui-même `BufferGetPage()`. Néanmoins, il est de la responsabilité de l'appelant du bloquer/débloquer et de verrouiller/déverrouiller les tampons au bon moment. Un verrou exclusif doit être obtenu et conservé pour chaque tampon cible avant l'appel à `GenericXLogRegisterBuffer()` et jusqu'à l'appel à `GenericXLogFinish()`.

- Les enregistrements de tampons (étape 2) et les modifications des images de page (étape 3) peuvent être librement mélangés. Les deux étapes peuvent donc être répétées dans n'importe quelle séquence. Gardez en tête que les tampons doivent être enregistrés dans le même ordre que l'obtention des verrous lors de leur rejeu.

- Le nombre maximum de tampons qui peut être enregistré pour un enregistrement générique dépend de la constante `MAX_GENERIC_XLOG_PAGES`. Une erreur est renvoyée si cette limite est dépassée.

- Un enregistrement générique suppose que les blocs à modifier aient un schéma standard et, en particulier, qu'il n'y ait pas de données utiles entre `pd_lower` et `pd_upper`.

- Comme vous modifiez des copies de pages de tampon, `GenericXLogStart()` ne commence pas une section critique. De ce fait, vous pouvez faire de l'allocation mémoire de façon sûre, en renvoyant des erreurs le cas échéant, entre `GenericXLogStart()` et `GenericXLogFinish()`. La seule section réellement critique se trouve dans `GenericXLogFinish()`. Il n'est pas non plus nécessaire de s'inquiéter lors de l'appel de `GenericXLogAbort()` pendant une sortie en erreur.

- `GenericXLogFinish()` fait attention à marquer les tampons comme modifiés et à configurer leur LSN. Vous n'avez pas besoin de le faire explicitement.

- Pour les relations non journalisées, tout fonctionne de la même façon sauf qu'aucun enregistrement n'est réellement émis. De ce fait, vous n'avez pas besoin de faire une quelconque vérification explicite pour les relations non journalisées.

- La fonction de rejeu des enregistrements génériques acquiert des verrous exclusifs sur les tampons dans le même ordre qu'ils ont été enregistrés. Après l'exécution du rejeu, les verrous sont relâchés dans le même ordre.

- Si `GENERIC_XLOG_FULL_IMAGE` n'est pas spécifié pour un tampon enregistré, l'enregistrement générique contient un delta entre les anciennes et les nouvelles images. Ce delta est basé sur une comparaison octet par octet. Ceci n'est pas spécialement compact dans le cas d'un déplacement de données dans une page, et pourrait être amélioré dans le futur.

Chapitre 66. Gestionnaires de ressources WAL personnalisées

Ce chapitre explique l'interface entre le cœur du système PostgreSQL et les gestionnaires de ressources WAL personnalisées, qui permettent aux extensions de s'intégrer directement avec les WAL.

Une extension, en particulier une méthode d'accès à une table ou une méthode d'accès à un index, peut avoir besoin d'utiliser des WAL pour la restauration, la réplication et/ou le décodage logique. Les gestionnaires de ressources personnalisés sont une alternative plus flexible aux enregistrements génériques des WAL (qui ne prennent pas en charge le décodage logique), mais plus complexes à implémenter dans une extension.

Pour créer un nouveau gestionnaire de ressources WAL personnalisées, définissez d'abord une structure `RmgrData` avec l'implémentation des méthodes du gestionnaire de ressources. Reportez-vous à `src/backend/access/transam/README` et `src/include/access/xlog_internal.h` dans la source de PostgreSQL.

```
/*
 * Table de méthodes pour les gestionnaires de ressources.
 *
 * Cette structure doit rester synchronisée avec la définition de
 PG_RMGR dans
 * rmgr.c.
 *
 * rm_identify doit renvoyer un nom pour l'enregistrement relatif à
 xl_info
 * (sans référence au rmid). Par exemple, XLOG_BTREE_VACUUM serait
 nommé
 * "VACUUM". rm_desc peut ensuite être appelé pour obtenir des
 détails
 * supplémentaires pour l'enregistrement, s'ils sont disponibles
 (par exemple,
 * le dernier bloc).
 *
 * rm_mask prend en entrée une page modifiée par le gestionnaire de
 ressources
 * et masque les bits qui ne doivent pas être marqués par
 * wal_consistency_checking.
 *
 * RmgrTable[] est indexé par les valeurs de RmgrId (voir
 rmgrlist.h).
 * Si rm_name est NULL, l'entrée RmgrTable correspondante est
 considérée comme
 * invalide.
 */
typedef struct RmgrData
{
    const char *rm_name;
    void        (*rm_redo) (XLogReaderState *record);
    void        (*rm_desc) (StringInfo buf, XLogReaderState
 *record);
    const char *(*rm_identify) (uint8 info);
    void        (*rm_startup) (void);
    void        (*rm_cleanup) (void);
    void        (*rm_mask) (char *pagedata, BlockNumber blkno);
```

```
    void        (*rm_decode) (struct LogicalDecodingContext *ctx,
                              struct XLogRecordBuffer *buf);
} RmgrData;
```

Ensuite, enregistrez votre nouveau gestionnaire de ressources.

```
/*
 * Enregistrer un nouveau gestionnaire de ressources WAL
 personnalisées.
 *
 * Les ID de gestionnaire de ressources doivent être uniques au
 monde pour
 * toutes les extensions. Se référer à
 * https://wiki.postgresql.org/wiki/CustomWALResourceManagers pour
 réserver un
 * RmgrId unique pour votre extension, afin d'éviter les conflits
 avec les
 * extensions d'autres développeurs. Lors du développement,
 utilisez
 * RM_EXPERIMENTAL_ID pour éviter la réservation inutile d'un
 nouvel ID.
 */
extern void RegisterCustomRmgr(RmgrId rmid, RmgrData *rmgr);
```

`RegisterCustomRmgr` doit être appelé depuis la fonction _PG_init du module d'extension. Lors du développement d'une nouvelle extension, utilisez `RM_EXPERIMENTAL_ID` pour *`rmid`*. Lorsque vous êtes prêt à publier l'extension pour les utilisateurs, réservez un nouvel ID de gestionnaire de ressources sur la page Custom WAL Resource Manager[1].

Mettez le module d'extension implémentant le gestionnaire de ressources personnalisées dans shared_preload_libraries afin qu'il soit chargé assez tôt lors du démarrage de PostgreSQL .

> **Note**
>
> L'extension doit rester dans shared_preload_libraries tant que des enregistrements personnalisés de WAL peuvent exister dans le système. Sinon, PostgreSQL ne pourra pas appliquer ou décoder les enregistrements personnalisés de WAL, ce qui peut empêcher le serveur de démarrer.

[1] https://wiki.postgresql.org/wiki/CustomWALResourceManagers

Chapitre 67. Index B-Tree

67.1. Introduction

PostgreSQL inclut une implémentation de la structure standard d'index btree (*multi-way balanced tree*) N'importe quel type de données pouvant être trié dans un ordre linéaire clairement défini peut être indexé par un index btree. La seule limitation est qu'une entrée d'index ne peut dépasser approximativement un tiers de page (après la compression TOAST si cela est possible).

Puisque chaque classe d'opérateur btree impose un ordre de tri sur son type de données, les classes d'opérateur btree (ou, en réalité, les familles d'opérateur) ont finies par être utilisées par PostgreSQL comme représentation et connaissance générale des sémantiques de tri. En conséquence, elles ont acquis certaines fonctionnalités qui vont au delà de ce qui serait nécessaire pour simplement supporter les index btree, et des parties du systèmes qui sont éloignées des méthodes d'accès btree les utilisant.

67.2. Comportement des classes d'opérateur B-Tree

Comme montré dans Tableau 38.3, une classe d'opérateur btree doit fournir cinq opérateurs de comparaison, <, <=, =, >= et >. On pourrait supposer que <> devraient également faire partie de la classe d'opérateur, mais ce n'est pas le cas car cela ne serait presque jamais utile d'utiliser une clause WHERE <> dans une recherche d'index. (Dans certains cas, le planificateur traite <> comme s'il était associé avec une classe d'opérateur btree ; mais il trouve cet opérateur via le lien du négateur de l'opérateur =, plutôt que depuis `pg_amop`.)

Quand plusieurs types de données partagent des sémantiques de tri presque identiques, leurs classes d'opérateurs peuvent être regroupées dans une famille d'opérateur. Il est avantageux de procéder ainsi car cela permet au planificateur de faire des déductions quant aux comparaisons entre plusieurs types. Chaque classe d'opérateur au sein d'une famille devrait contenir les opérateurs concernant un seul type (et les fonctions de support associées), tandis que les opérateurs de comparaison inter-types et les fonctions de support sont « libres » dans la famille. Il est recommandé qu'un ensemble complet d'opérateurs inter-types soit inclus dans la famille, afin d'assurer que le planificateur puisse représenter n'importe quelle condition de comparaison qu'il pourrait déduire depuis la transitivité.

Il y a des supposition basiques qu'une famille d'opérateur btree doit satisfaire :

- Un opérateur = doit être une relation d'équivalence ; c'est-à-dire que pour toutes les valeurs non nulles *A*, *B*, *C* du type de données :
  - *A* = *A* est vrai (*loi de réflexivité*)
  - si *A* = *B*, alors *B* = *A* (*loi de symétrie*)
  - si *A* = *B* et *B* = *C*, alors *A* = *C* (*loi de transitivité*)
- Un opérateur < doit être une relation de tri forte ; c'est-à-dire, pour toutes les valeurs non nulles *A*, *B*, *C* :
  - *A* < *A* est faux (*loi d'antiréflexivité*)
  - si *A* < *B* et *B* < *C*, alors *A* < *C* (*loi de transitivité*)
- De plus, le tri est total ; c'est-à-dire, pour toutes les valeurs non nulles *A*, *B* :
  - exactement une seule des expressions *A* < *B*, *A* = *B*, et *B* < *A* est vraie (*loi de trichotomie*)

  (Bien entendu, la loi de trichotomie justifie la définition de la fonction de support de comparaison).

Les trois autres opérateurs sont définis avec = et < de manière évidente, et doivent se comporter de manière cohérentes avec ceux-ci.

Pour une famille d'opérateurs supportant plusieurs types de données, les lois définies auparavant doivent continuer à s'appliquer quand *A*, *B*, *C* sont pris de n'importe quel type de données de la famille. Les lois de transitivité sont les plus délicates à garantir, car, dans des situations inter-types, elles représentent des déclarations comme quoi les comportements de deux ou trois différents opérateurs sont cohérents. Comme exemple, mettre `float8` et `numeric` dans la même famille d'opérateurs ne fonctionnerait pas, du moins pas avec les sémantiques actuelles qui définissent que les valeurs de type `numeric` sont converties en `float8` pour la comparaison vers un `float8`. Du fait de la précision limitée du type `float8`, cela veut dire que des valeurs `numeric` distinctes seraient considérées par la comparaison comme égales à la même valeur `float8`, et par conséquent la loi de transitivé échouerait.

Une autre exigence pour les familles contenant plusieurs types est que les transtypages implicites ou de coercition binaire qui sont définis entre les types de données inclus dans la famille d'opérateurs ne doivent pas changer l'ordre de tri associé.

La raison pour laquelle les index B-tree nécessitent que ces lois soient vérifiées pour un même type de données devraient être tout à fait claires : sans celles-ci, il n'y a pas d'ordre avec lequel organiser les clés. En outre, les recherches d'index utilisant une clé de comparaison d'un type de données différent nécessitent que la comparaison se comporte sainement à travers deux types de données. Les extensions à trois types de données ou plus au sein d'une famille ne sont pas strictement requis par le mécanisme d'index B-tree lui-même, mais le planificateur se repose sur eux pour des besoins d'optimisation.

67.3. Fonctions de support B-Tree

Comme montré dans Tableau 38.9, btree définit une fonction de support obligatoire et quatre facultatives. Les cinq méthodes définies par l'utilisateur sont :

`order`

Pour chaque combinaison de types de données pour laquelle une famille d'opérateur B-tree fournit des opérateurs de comparaison, elle doit fournir une fonction de support de comparaison inscrite dans `pg_amproc` avec la fonction de support 1 et `amproclefttype`/`amprocrighttype` égaux aux types de données gauche et droit pour la comparaison (c'est-à-dire les mêmes types de données que l'opérateur correspondant a inscrit dans `pg_amop`). La fonction de comparaison doit prendre en entrée deux valeurs non nulles *A* et *B* et retourner une valeur `int32` qui est < `0`, `0`, ou > `0` quand, respectivement *A* < *B*, *A* = *B*, ou *A* > *B*. Une valeur de retour NULL est également interdite : toutes les valeurs du type de données doivent être comparables. Voir `src/backend/access/nbtree/nbtcompare.c` pour plus d'exemples.

Si les valeurs comparées sont d'un type avec collation, l'identifiant de collation approprié sera passé à la fonction de support de comparaison, en utilisant le mécanisme standard `PG_GET_COLLATION()`.

`sortsupport`

De manière facultative, une famille d'opérateurs B-tree peut fournir une ou plusieurs fonctions *sort support*, inscrites comme fonctions de support numéro 2. Ces fonctions permettent d'implémenter des comparaisons dans l'optique de tri de manière plus efficace qu'appeler naivement la fonction de support de comparaison. Les API impliquées pour cela sont définies dans `src/include/utils/sortsupport.h`.

`in_range`

De manière facultative, une famille d'opérateurs B-tree peut fournir une ou plusieurs fonctions de support *in_range* inscrites comme fonction de support numéro 3. Celles-ci ne sont pas utilisées durant les opérations d'index B-tree ; mais plutôt, elles étendent les sémantiques de la famille

d'opérateurs de telle manière qu'elles puissent supporter les clauses de fenêtrage contenant les types de limite de cadre `RANGE` *`décalage`* `PRECEDING` et `RANGE` *`décalage`* `FOLLOWING` (voir Section 4.2.8). Fondamentalement, les informations supplémentaires fournies sont comment additionner et soustraire une valeur d'un *`décalage`* d'une manière qui est compatible avec le tri de données de la famille.

Une fonction `in_range` doit avoir la signature

```
in_range(val type1, base type1, offset type2, sub bool, less
 bool)
returns bool
```

`val` et *`base`* doivent être du même type, qui est un des types supportés par la famille d'opérateur (c'est-à-dire un type pour lequel elle fournit un tri). Cependant, *`offset`* peut être d'un type de données différent, qui peut par ailleurs ne pas être supporté par la famille. Un exemple est que la famille `time_ops` incluse par défaut fournit une fonction `in_range` qui a un *`offset`* de type `interval`. Une famille peut fournir des fonctions `in_range` pour n'importe lesquels des types de données qu'elle supporte, et un ou plusieurs types *`offset`*. Chaque fonction `in_range` devrait être inscrite dans `pg_amproc` avec `amproclefttype` égal à `type1` et `amprocrighttype` égal à `type2`.

Les sémantiques essentielles pour une fonction `in_range` dépendent des deux paramètres de drapeau booléens. Elle devrait ajouter ou soustraire *`base`* et *`offset`*, puis comparer *`val`* au résultat, comme ceci :

- si !*`sub`* et !*`less`*, renvoyer *`val`* >= (*`base`* + *`offset`*)
- si !*`sub`* et *`less`*, renvoyer *`val`* <= (*`base`* + *`offset`*)
- si *`sub`* et !*`less`*, renvoyer *`val`* >= (*`base`* - *`offset`*)
- si *`sub`* et *`less`*, renvoyer *`val`* <= (*`base`* - *`offset`*)

Avant de procéder, la fonction devrait vérifier le signe d' *`offset`* : s'il est inférieur à zéro, lever l'erreur `ERRCODE_INVALID_PRECEDING_OR_FOLLOWING_SIZE` (22013) avec un message d'erreur tel que « taille précédente ou suivante invalide dans la fonction de fenêtrage ». (Cela est requis par le standard SQL, bien que des familles d'opérateur non standards pourraient peut être choisir d'ignorer cette restriction, puisqu'il n'y a pas vraiment de nécessité de sémantique dans ce cas.) Cette exigence est déléguée à la fonction `in_range` si bien que le code du moteur n'a pas besoin de comprendre ce que « inférieur à zéro » signifie pour un type de données particulier.

Une autre attente est que les fonctions `in_range` devraient, si applicable, éviter de générer une erreur si *`base`* + *`offset`* ou *`base`* - *`offset`* devait causer un débordement. Le résultat de comparaison correct peut être déterminé même si cette valeur devait être en dehors de l'intervalle des valeurs du type de données. Notez que si le type de données inclut des concepts tels que « infinity » ou « NaN », des précautions supplémentaires pourraient être nécessaires pour s'assurer que les résultats de `in_range` soient en accord avec l'ordre de tri normal de la famille d'opérateurs.

Les résultats de la fonction `in_range` doivent être cohérents avec l'ordre de tri imposé par la famille d'opérateurs. Pour être précis, pour n'importe quelles valeurs fixées de *`offset`* et *`sub`*, alors :

- Si `in_range` avec *`less`* = true est vrai pour certains *`val1`* et *`base`*, il doit être vrai pour chaque *`val2`* <= *`val1`* avec le même *`base`*.
- Si `in_range` avec *`less`* = true est faux pour certains *`val1`* et *`base`*, il doit être faux pour chaque *`val2`* >= *`val1`* avec le même *`base`*.

- Si `in_range` avec *`less`* = true est vrai pour certains *`val`* et *`base1`*, il doit être vrai pour chaque *`base2`* >= *`base1`* avec le même *`val`*.
- Si `in_range` avec *`less`* = true est faux pour certains *`val`* et *`base1`*, il doit être faux pour chaque *`base2`* <= *`base1`* avec le même *`val`*.

Des déclarations similaires avec des conditions inversées continuent à s'appliquer quand *`less`* = false.

Si le type est trié (`type1`) par rapport à une collation, l'OID de collation approprié sera passé à la fonction `in_range` en utilisant le mécanisme standard `PG_GET_COLLATION()`.

Les fonctions `in_range` n'ont pas besoin de gérer les valeurs en entrée NULL, et typiquement elles seront marquées comme strict.

`equalimage`

Une famille d'opérateurs B-tree facultative peut fournir les fonctions de support `equalimage` (« l'égalité implique une égalité d'image »), inscrites comme fonctions de support numéro 4. Ces fonctions permettent au code du moteur de déterminer quand il est sûr d'appliquer l'optimisation de dédoublonnage B-tree. Actuellement, les fonctions `equalimage` sont seulement appelées lors de la construction ou reconstruction d'un index.

Une fonction `equalimage` doit avoir comme signature

```
equalimage(opcintype oid) returns bool
```

La valeur retournée est une information statique relative à une classe d'opérateur et une collation. Retourner `true` indique que la fonction `order` pour la classe d'opérateurs est garantie de retourner seulement `0` (« les arguments sont égaux ») quand ses arguments *`A`* et *`B`* sont aussi interchangeables sans aucune perte d'information sémantique. Ne pas inscrire une fonction `equalimage` ou retourner `false` indique que cette condition ne peut être tenue.

L'argument *`opcintype`* est le `pg_type.oid` du type de données que la classe d'opérateurs indexe. Ceci est une commodité qui permet de réutiliser la même fonction `equalimage` sous-jacente entre plusieurs classes d'opérateurs. Si *`opcintype`* est un type de données collationné, l'identifiant de collation appropriée sera passé à la fonction `equalimage`, par le mécanisme standard `PG_GET_COLLATION()`.

Tant que la classe d'opérateurs est concernée, retourner `true` indique que le dédoublement est sûr (ou sûr pour la collation dont l'identifiant a été passé à sa fonction `equalimage`). Cependant, le code du moteur considérera le dédoublonnage sécurisé pour un index, si *chaque* colonne indexée utilise une classe d'opérateur ayant inscrit une fonction `equalimage`, et si chaque fonction retourne `true` par appel.

L'égalité d'image est *presque* la même condition qu'une simple égalité bit à bit. Il n'y a qu'une seule et subtile différence : en indexant un type de données « varlena », la représentation sur disque de deux images de données égales peuvent ne pas être identiques bit à bit, à cause des incohérences lors de l'application de la compression TOAST sur les données en entrée. Dans les règles, quand une fonction `equalimage` d'une classe d'opérateurs retourne `true`, on peut présumer sans se tromper que la fonction C `datum_image_eq()` correspondra avec la fonction `order` de la classe d'opérateurs (sous réserve que le même identifiant de collation soit passé aux fonctions `equalimage` et `order`).

Le code du moteur est fondamentalement incapable de déduire quoi que ce soit au sujet du statut « l'égalité implique l'égalité d'image » d'une classe d'opérateurs incluse dans une famille de types de données multiples en se basant sur les détails d'autres classes d'opérateur de la même famille. Aussi, il n'est pas pertinent pour une famille d'opérateurs d'inscrire une fonction `equalimage` inter-type, et essayer déclenchera une error. En effet, le statut de « l'égalité implique l'égalité d'image » ne dépend pas juste de la sémantique de l'ordre/égalité, qui est plus ou moins définie

au niveau de la famille d'opérateurs. En général, les sémantiques d'un type particulier de données doivent être considérées séparément.

La convention suivie par les classes d'opérateurs incluses avec la distribution principale PostgreSQL est d'inscrire une fonction générique `equalimage`. La plupart des classes d'opérateurs inscrivent `btequalimage()`, qui indique que le dédoublonnage est sécurisé sans conditions. Les classes d'opérateurs pour les types de données collationnés comme `text` inscrivent `btvarstrequalimage()`, qui indique que le dédoublonnage est sécurisé avec les collations déterministes. La bonne pratique pour une extension tierce est d'inscrire leur propre fonction personnalisée pour garder le contrôle.

`options`

En option, une famille d'opérateurs B-tree peut fournir des fonctions de support des `options` (« options spécifiques à la classe d'opérateurs »), enregistrées sous le numéro 5 des fonctions de support. Ces fonctions définissent un ensemble de paramètres visibles à l'utilisateur et contrôlant le comportement de la classe d'opérateurs.

Une fonction de support `options` doit avoir cette signature

```
options(relopts local_relopts *) returns void
```

La fonction se voit fournie un pointeur vers une structure `local_relopts` qui doit être remplie avec un ensemble d'options spécifiques à une classe d'opérateurs. Les options sont accessibles à partir des autres fonctions de support en utilisant les macros `PG_HAS_OPCLASS_OPTIONS()` et `PG_GET_OPCLASS_OPTIONS()`.

Actuellement, aucune classe d'opérateurs B-Tree n'a de fonction de support `options`. B-tree n'autorise pas une représentation flexible des clés comme GiST, SP-GiST, GIN et BRIN le font. Donc, `options` n'a probablement pas beaucoup d'intérêt pour la méthode d'accès aux index B-tree actuellement. Néanmoins, cette fonction de support a été ajouté au B-tree par cohérence, et trouvera probablement son utilité lors des prochaines évolutions du B-tree dans PostgreSQL.

67.4. Implémentation

Cette section couvre les détails de l'implémentation des index B-Tree qui peuvent être utiles pour les utilisateurs avancés. Voir `src/backend/access/nbtree/README` dans les sources de la distribution pour une description plus en détails de l'implémentation du B-Tree.

67.4.1. Structure B-Tree

Les index B-Tree de PostgreSQL sont des structures arborescentes multi-niveaux, où chaque niveau de l'arbre peut être utilisé comme une liste doublement chaînée de pages. Une seule métapage est stockée à une position fixe au début du premier segment de fichier de l'index. Toutes les autres pages sont soit des pages feuilles, soit des pages internes. Les pages feuilles sont les pages de plus bas niveau de l'arbre. Tous les autres niveaux consistent en des pages internes. Chaque page feuille contient des tuples qui pointent sur les enregistrements en table. Chaque page interne contient des tuples qui pointent vers le niveau inférieur suivant dans l'arbre. En général, 99% des pages sont des pages feuilles. Aussi bien les pages internes que les pages feuilles emploient le format standard de page décrit dans Section 73.6.

Des nouvelles pages feuilles sont ajoutées à un index B-Tree quand un tuple entrant ne peut pas tenir dans une page feuille existante. Une opération de *fractionnement de page* a alors lieu et libère de la place pour les éléments qui appartiennent à la page surchargée en déplaçant une portion de ces éléments dans une nouvelle page. Le fractionnement de page doit aussi insérer un *lien descendant* vers la nouvelle page dans la page parente, ce qui peut causer à son tour le fractionnement du parent. Le fractionnement de page se produit en « cascade vers les niveaux supérieurs » de façon récursive. Si la page racine ne peut finalement pas porter le lien descendant, une opération de *fractionnement de page*

racine se produit. Elle ajoute un nouveau niveau dans la structure arborescente en créant une nouvelle page racine un niveau au dessus de la page racine d'origine.

67.4.2. Suppression ascendante d'index

Les index B-Tree n'ont pas directement connaissance que sous MVCC, il peut y avoir plusieurs versions existantes de la même ligne logique d'une table ; pour un index, chaque ligne est un objet indépendant qui a besoin de sa propre entrée d'index. Le « renouvellement des versions » (*version churn* dans la version originale) des lignes peut parfois s'accumuler et nuire à la latence et au débit des requêtes. Ceci se produit typiquement avec des charges élevées en UPDATE où la plupart des mises à jour individuelles ne peuvent appliquer l'optimisation HOT. Changer la valeur de seulement une colonne couverte par un index durant un UPDATE nécessite *toujours* un nouveau ensemble d'entrées d'index -- un pour *chaque et tous les* index sur la table. Notons en particulier que cela inclut les index qui ne sont pas « modifiés logiquement » par la commande UPDATE. Tous les index nécessiteront une entrée d'index physique successeur qui pointe vers la dernière version en table. Chaque nouvelle entrée à l'intérieur de chaque index aura besoin, en général, de coexister avec l'entrée « modifiée » originale pour une courte période de temps (typiquement jusque peu après que la transaction de l'UPDATE soit validée).

Les index B-Tree suppriment de manière incrémentielle les entrées d'index de renouvellement de version en effectuant des passes de *suppression ascendante de l'index*. Chaque passe de suppression est déclenchée en réaction à un « fractionnement de page de renouvellement de version » anticipée. Ceci se produit avec les index qui ne sont pas modifiés logiquement par les déclarations UPDATE, dans lesquels des accumulations concentrées de versions obsolètes dans des blocs particuliers auraient lieu autrement. Un fractionnement de bloc sera normalement évité, bien qu'il est possible que certains niveaux d'implémentation d'heuristique échoueront même à identifier et à supprimer une ligne à renouveller (*garbage tuple* dans la version originale) d'index (dans ce cas, un fractionnement de bloc ou une passe de dédoublement résout le problème de l'entrée d'une nouvelle ligne qui ne rentre pas dans le bloc feuille). Le pire nombre de versions que n'importe quel parcours d'index doit traverser (pour n'importe quel enregistrement unique logique) est un contributeur important pour la réactivité et le débit globaux du système. Une passe de suppression ascendante d'index cible les lignes à renouveller supposées dans un bloc feuille unique par des distinctions *qualitatives* impliquant enregistrements logiques et versions. Ceci diffère des nettoyages d'index « descendants » effectués par les processus de l'autovacuum, qui sont déclenchés quand certains seuils *quantitatifs* au niveau table sont dépassés (voir Section 25.1.6).

> **Note**
>
> Les opérations de suppression effectuées à l'intérieur des index B-Tree ne sont pas toutes des opérations de suppression ascendantes. Il y a une catégorie de suppression d'entrées d'index : *la suppression d'entrée d'index simple* (*simple index tuple deletion* en version originale). C'est une opération de maintenance différée qui supprime les entrées d'index en toute sécurité (ceux dont le bit LP_DEAD de son élément identifiant est déjà affecté). Comme les suppressions ascendantes d'index, la suppression d'index simple a lieu au moment où un fractionnement de bloc est attendu comme moyen d'éviter ce fractionnement.
>
> La suppression simple est opportuniste dans le sens où elle peut seulement s'effectuer quand les parcours récents d'index mettent à jour les bits LP_DEAD des éléments affectés lors d'une passe. Avant PostgreSQL 14, la seule catégorie de suppression B-Tree était la suppression simple. Les principales différences entre elle et les suppressions ascendantes sont que seule la première est dirigée de manière opportuniste par l'activité des passes de parcours d'index, tandis que la nouvelle ne cible spécifiquement que le renouvellement de version des UPDATE qui ne modifient pas logiquement les colonnes indexées.

Les suppressions ascendantes d'index effectuent la grande majorité des nettoyages des entrées à renouveller d'index pour certains index et certaines charges de travail. C'est le cas pour les index B-

Tree sujets à un renouvellement de version significatif par les UPDATE qui ne modifient logiquement que rarement ou jamais les colonnes couvertes par l'index. La valeur moyenne et la pire valeur possible de versions par enregistrement logique peuvent être maintenues basses grâce aux passes incrémentales de suppression ciblée. Il est possible que la taille sur disque de certains index n'augmentera jamais même d'un simple bloc malgré un renouvellement de version *continu* par les UPDATE. Même si cela était le cas, un « nettoyage » exhaustif par une opération VACUUM (typiquement exécutée par un processus autovacuum), sera éventuellement requis comme une partie de nettoyage *collectif* de la table et chacun de ses index.

À la différence du VACUUM, la suppression ascendante d'index ne fournit pas de solides garanties pour déterminer quel peut être la plus ancienne entrée à renouveller dans l'index. Aucun index ne peut permettre de conserver des entrées d'index « à renouvellement flottant » qui seront morts avant le moment de conservation limite partagé collectivement par la table et tous ses index. Cette constante fondamentale au niveau table implique qu'il est sans danger de recycler les TID d'une table. C'est ainsi qu'il est possible pour les enregistrements logiques distincts de réutiliser les mêmes TID dans une table au cours du temps (bien que cela ne peut jamais se produire avec deux enregistrements logiques dont l'espérance de vie couvre le même cycle VACUUM).

67.4.3. Dédoublement

Un doublon est un tuple de page feuille (un tuple qui pointe sur un enregistrement en table) où *toutes* les valeurs des colonnes clés de l'index correspondent aux valeurs de colonnes respectives d'au moins un autre tuple de page feuille dans le même index. Les tuples doublons sont assez communs en pratique. Les index B-Tree peuvent utiliser une représentation spéciale gérant efficacement l'espace pour les doublons lorsqu'une fonctionnalité est activée : le *dédoublement*.

Le dédoublement fonctionne en fusionnant périodiquement les groupes d'enregistrements doublons ensemble, formant une *liste d'affectation* unique pour chaque groupe. Le ou les valeurs de colonnes clés n'apparaissent qu'une fois dans cette représentation. Elles sont suivies par un tableau trié des TID pointant sur les lignes en table. Ceci réduit significativement la taille de stockage des index où chaque valeur (ou chaque combinaison distincte de valeur de colonne) apparait plusieurs fois en moyenne. La latence des requêtes peut sensiblement diminuer. Le débit général des requêtes peut augmenter sensiblement. Le coût supplémentaire de la routine de vacuum d'index peut aussi être notablement réduite.

> **Note**
>
> Le dédoublement B-Tree est tout aussi efficace avec des « duplicats » contenant une valeur NULL, même si les valeurs NULL ne sont jamais égales d'après l'opérateur = de toute classe d'opérateurs B-Tree. Pour toute implémentation comprenant la structure disque B-Tree, NULL est simplement une autre valeur du domaine des valeurs indexées.

Le processus de dédoublement se déroule avec le moins d'effort possible, quand un nouveau élément est inséré et ne peut rentrer dans une page feuille existante, mais seulement quand la suppression d'entrée d'index ne peut pas libérer suffisamment d'espace pour le nouvel élément (typiquement la suppression est brièvement considérée puis ignorée). Contrairement à la liste chainée d'enregistrements GIN, la liste chainée d'enregistrements B-Tree n'a pas besoin de s'étendre à chaque fois qu'un nouveau doublon est inséré ; ils sont simplement une représentation physique différente du contenu logique de la page feuille. Ce concept priorise l'uniformité des performances sur des charges de travail mixte en lecture-écriture. La plupart des applications clientes verront un bénéfice modéré sur les performances en utilisant le dédoublement. Le dédoublement est activé par défaut.

CREATE INDEX et REINDEX appliquent la déduplication pour créer les listes de lignes, bien que la stratégie utilisée soit un peu différente. Chaque groupe de lignes ordinaires dupliquées rencontré dans l'entrée triée prise à partir de la table est assemblé en une liste *avant* d'être ajouté à la page feuille en cours. Les listes individuelles sont assemblées avec autant de TID que possible. Les pages feuilles

sont écrites de la façon habituelle, sans passe de déduplication séparée. Cette stratégie convient bien à `CREATE INDEX` et `REINDEX` car ce sont des opérations de groupe en lot unique.

Les charges de travail majoritaires en écriture et qui ne bénéficient pas du dédoublement du fait qu'il y a peu ou pas de doublons dans les index, encoureront une pénalité stable et légère de performance (sauf si le dédoublement est explicitement désactivé). Le paramètre de stockage `deduplicate_items` peut être utilisé pour désactiver le dédoublement au niveau de chaque index. Il n'y a jamais de pénalité de performance avec des charges de travail en lecture seule, puisque la lecture de liste chainée des tuples est au moins aussi efficace que la lecture de la représentation standard des tuples. Désactiver le dédoublement n'est en général pas utile.

Il est parfois possible pour des index uniques (autant que pour des contraintes uniques) d'utiliser le dédoublement. Cela permet aux blocs feuilles d'« absorber » temporairement les doublons supplémentaires des renouvellement de version. Le dédoublement des index uniques augmente les suppressions ascendantes d'index, spécialement dans les cas où une longue transaction garde un instantané qui bloque la collecte des éléments à nettoyer. Le but est de gagner du temps pour que la stratégie de suppression ascendante d'index devienne encore efficace. Retarder les fractionnements de blocs jusqu'à ce qu'une transaction longue finisse naturellement peut permettre à une passe de suppression ascendante de réussir là où une passe de suppression précédente a échoué.

> **Astuce**
>
> Une heuristique particulière est utilisée pour déterminer si une passe de dédoublement peut prendre place dans un index unique. Elle peut souvent directement passer au fractionnement de page feuille, évitant ainsi une pénalité de performance par gaspillage de cycles de dédoublement inutiles. Si vous êtes préoccupés par le coût additionnel du dédoublement, veuillez considérer le paramètre `deduplicate_items = off` de manière sélective. Conserver le dédoublement activé par index distinct n'a guère d'impacts uniques.

Le dédoublement ne peut pas être utilisé dans tous les cas à cause des restrictions au niveau de l'implémentation. L'innocuité du dédoublement est déterminé quand `CREATE INDEX` ou `REINDEX` est exécutée.

Notez que le dédoublement est considéré comme non sécurisé et ne peut être utilisé dans les cas suivants qui impliquent des différences significatives au niveau sémantique parmi des données identiques :

- `text`, `varchar`, et `char` ne peuvent être dédoublonnés quand une collation *non déterministique* est utilisée. La différence de casse et des accents doit être préservée parmi les données égales.
- `numeric` ne peut pas utilisé le dédoublonnement. La précision des nombres doit être préservée parmi les données identiques.
- `jsonb` ne peut être dédoublonné, depuis que la classe d'opérateur B-Tree pour le type `jsonb` utilise en interne un type `numeric`.
- `float4` et `float8` ne peuvent être dédoublonnés. Ces types ont une représentation distincte pour `-0` et `0`, qui sont cependant considérés égaux. Cette différence doit être préservée.

Une autre restriction au niveau de l'implémentation existe et pourra être levée dans une version future de PostgreSQL:

- Les types conteneur (tel que les types compostes, tableaux ou intervalle) ne peuvent être dédoublonnés.

Une autre restriction au niveau de l'implémentation s'applique quel que soient les classes d'opérateurs ou collations employées :

- Les index `INCLUDE` ne peuvent pas être dédoublonnés.

Chapitre 68. Index GiST

68.1. Introduction

GiST est un acronyme de *Generalized Search Tree*, c'est-à-dire arbre de recherche généralisé. C'est une méthode d'accès balancée à structure de type arbre, qui agit comme un modèle de base dans lequel il est possible d'implanter des schémas d'indexage arbitraires. B-trees, R-trees et de nombreux autres schémas d'indexage peuvent être implantés en GiST.

GiST a pour avantage d'autoriser le développement de types de données personnalisés avec les méthodes d'accès appropriées, par un expert en types de données, plutôt que par un expert en bases de données.

Quelques informations disponibles ici sont dérivées du site web[1] du projet d'indexage GiST de l'université de Californie à Berkeley et de la thèse de Marcel Kornacker, Méthodes d'accès pour les systèmes de bases de données de la prochaine génération[2]. L'implantation GiST de PostgreSQL est principalement maintenu par Teodor Sigaev et Oleg Bartunov. Leur site web[3] fournit de plus amples informations.

68.2. Classes d'opérateur internes

La distribution de PostgreSQL inclut les classes d'opérateur GiST indiquées dans Tableau 68.1. (Quelques modules optionnels décrits dans Annexe F fournissent des classes d'opérateur GiST supplémentaires.)

Tableau 68.1. Classes d'opérateur GiST internes

Nom	Opérateurs indexables	Opérateurs de tri
box_ops	<< (box, box)	<-> (box, point)
	&< (box, box)	
	&& (box, box)	
	&> (box, box)	
	>> (box, box)	
	~= (box, box)	
	@> (box, box)	
	<@ (box, box)	
	&<| (box, box)	
	<<| (box, box)	
	|>> (box, box)	
	|&> (box, box)	
circle_ops	<< (circle, circle)	<-> (circle, point)
	&< (circle, circle)	
	&> (circle, circle)	
	>> (circle, circle)	
	<@ (circle, circle)	
	@> (circle, circle)	

[1] http://gist.cs.berkeley.edu/
[2] http://www.sai.msu.su/~megera/postgres/gist/papers/concurrency/access-methods-for-next-generation.pdf.gz
[3] http://www.sai.msu.su/~megera/postgres/gist/

Nom	Opérateurs indexables	Opérateurs de tri
	~= (circle, circle)	
	&& (circle, circle)	
	|>> (circle, circle)	
	<<| (circle, circle)	
	&<| (circle, circle)	
	|&> (circle, circle)	
inet_ops	<< (inet, inet)	
	<<= (inet, inet)	
	>> (inet, inet)	
	>>= (inet, inet)	
	= (inet, inet)	
	<> (inet, inet)	
	< (inet, inet)	
	<= (inet, inet)	
	> (inet, inet)	
	>= (inet, inet)	
	&& (inet, inet)	
multirange_ops	= (anymultirange, anymultirange)	
	&& (anymultirange, anymultirange)	
	&& (anymultirange, anyrange)	
	@> (anymultirange, anyelement)	
	@> (anymultirange, anymultirange)	
	@> (anymultirange, anyrange)	
	<@ (anymultirange, anymultirange)	
	<@ (anymultirange, anyrange)	
	<< (anymultirange, anymultirange)	
	<< (anymultirange, anyrange)	
	>> (anymultirange, anymultirange)	
	>> (anymultirange, anyrange)	
	&< (anymultirange, anymultirange)	
	&< (anymultirange, anyrange)	
	&> (anymultirange, anymultirange)	
	&> (anymultirange, anyrange)	
	-|- (anymultirange, anymultirange)	

Nom	Opérateurs indexables	Opérateurs de tri
	-|- (anymultirange, anyrange)	
point_ops	|>> (point, point)	<-> (point, point)
	<< (point, point)	
	>> (point, point)	
	<<| (point, point)	
	~= (point, point)	
	<@ (point, box)	
	<@ (point, polygon)	
	<@ (point, circle)	
poly_ops	<< (polygon, polygon)	<-> (polygon, point)
	&< (polygon, polygon)	
	&> (polygon, polygon)	
	>> (polygon, polygon)	
	<@ (polygon, polygon)	
	@> (polygon, polygon)	
	~= (polygon, polygon)	
	&& (polygon, polygon)	
	<<| (polygon, polygon)	
	&<| (polygon, polygon)	
	|&> (polygon, polygon)	
	|>> (polygon, polygon)	
range_ops	= (anyrange, anyrange)	
	&& (anyrange, anyrange)	
	&& (anyrange, anymultirange)	
	@> (anyrange, anyelement)	
	@> (anyrange, anyrange)	
	@> (anyrange, anymultirange)	
	<@ (anyrange, anyrange)	
	<@ (anyrange, anymultirange)	
	<< (anyrange, anyrange)	
	<< (anyrange, anymultirange)	
	>> (anyrange, anyrange)	
	>> (anyrange, anymultirange)	
	&< (anyrange, anyrange)	
	&< (anyrange, anymultirange)	
	&> (anyrange, anyrange)	
	&> (anyrange, anymultirange)	
	-|- (anyrange, anyrange)	
	-|- (anyrange, anymultirange)	

Nom	Opérateurs indexables	Opérateurs de tri
tsquery_ops	<@ (tsquery, tsquery)	
	@> (tsquery, tsquery)	
tsvector_ops	@@ (tsvector, tsquery)	

Pour des raisons historiques, la classe d'opérateurs `inet_ops` n'est pas la classe par défaut pour les types `inet` et `cidr`. Pour l'utiliser, mentionnez le nom de la classe dans la commande `CREATE INDEX`, par exemple

```
CREATE INDEX ON ma_table USING GIST (ma_colonne_inet inet_ops);
```

68.3. Extensibilité

L'implantation d'une nouvelle méthode d'accès à un index a toujours été un travail complexe. Il est, en effet, nécessaire de comprendre le fonctionnement interne de la base de données, tel que le gestionnaire de verrous ou le WAL.

L'interface GiST dispose d'un haut niveau d'abstraction, ce qui autorise le codeur de la méthode d'accès à ne coder que la sémantique du type de données accédé. La couche GiST se charge elle-même de la gestion des accès concurrents, des traces et de la recherche dans la structure en arbre.

Cette extensibilité n'est pas comparable à celle des autres arbres de recherche standard en termes de données gérées. Par exemple, PostgreSQL supporte les B-trees et les index de hachage extensibles. Cela signifie qu'il est possible d'utiliser PostgreSQL pour construire un B-tree ou un hachage sur tout type de données. Mais, les B-trees ne supportent que les prédicats d'échelle (<, =, >), les index de hachage que les requêtes d'égalité.

Donc, lors de l'indexation d'une collection d'images, par exemple, avec un B-tree PostgreSQL, seules peuvent être lancées des requêtes de type « est-ce que imagex est égale à imagey », « est-ce que imagex est plus petite que imagey » et « est-ce que imagex est plus grande que imagey ». En fonction de la définition donnée à « égale à », « inférieure à » ou « supérieure à », cela peut avoir une utilité. Néanmoins, l'utilisation d'un index basé sur GiST permet de créer de nombreuses possibilités de poser des questions spécifiques au domaine, telles que « trouver toutes les images de chevaux » ou « trouver toutes les images surexposées ».

Pour obtenir une méthode d'accès GiST fonctionnelle, il suffit de coder plusieurs méthodes utilisateur définissant le comportement des clés dans l'arbre. Ces méthodes doivent être suffisamment élaborées pour supporter des requêtes avancées, mais pour toutes les requêtes standard (B-trees, R-trees, etc.) elles sont relativement simples. En bref, GiST combine extensibilité, généralité, ré-utilisation de code et interface claire.

Une classe d'opérateurs d'index GiST doit fournir cinq méthodes, et six supplémentaires optionnelles. La précision de l'index est assurée par l'implantation des méthodes `same`, `consistent` et `union` alors que l'efficacité (taille et rapidité) de l'index dépendra des méthodes `penalty` et `picksplit`. Deux fonctions optionnelles sont `compress` et `decompress`, qui permettent à un index d'avoir des données internes de l'arbre d'un type différent de ceux des données qu'il indexe. Les feuilles doivent être du type des données indexées alors que les autres nœuds peuvent être de n'importe quelle structure C (mais vous devez toujours suivre les règles des types de données de PostgreSQL dans ce cas, voir ce qui concerne `varlena` pour les données de taille variable). Si le type de données interne de l'arbre existe au niveau SQL, l'option `STORAGE` de la commande `CREATE OPERATOR CLASS` peut être utilisée. La huitième méthode, optionnelle, est `distance`, qui est nécessaire si la classe d'opérateurs souhaite supporter les parcours ordonnées (intéressant dans le cadre des recherches du voisin-le-plus-proche, *nearest-neighbor*). La neuvième méthode, optionnelle, nommée `fetch`, est nécessaire si la classe d'opérateurs souhaite supporter les parcours d'index seuls, sauf quand la méthode `compress` est omise. La dixième méthode, optionnelle, est `options` et est nécessaire si l'opérateur de classe a

des paramètres définis par l'utilisateur. La onzième méthode, `sortsupport`, est aussi optionnelle et est utilisée pour accélérer la construction d'un index GiST.

`consistent`

Étant donné une entrée d'index `p` et une valeur de requête `q`, cette fonction détermine si l'entrée de l'index est cohérente (« consistent » en anglais) avec la requête ; c'est-à-dire, est-ce que le prédicat « *`colonne_indexée opérateur_indexable`* `q` » soit vrai pour toute ligne représentée par l'entrée de l'index ? Pour une entrée de l'index de type feuille, c'est l'équivalent pour tester la condition indexable, alors que pour un nœud interne de l'arbre, ceci détermine s'il est nécessaire de parcourir le sous-arbre de l'index représenté par le nœud. Quand le résultat est `true`, un drapeau `recheck` doit aussi être renvoyé. Ceci indique si le prédicat est vrai à coup sûr ou seulement peut-être vrai. Si `recheck` = `false`, alors l'index a testé exactement la condition du prédicat, alors que si `recheck` = `true`, la ligne est seulement un correspondance de candidat. Dans ce cas, le système évaluera automatiquement l'*`opérateur_indexable`* avec la valeur actuelle de la ligne pour voir s'il s'agit réellement d'une correspondance. Cette convention permet à GiST de supporter à la fois les structures sans pertes et celles avec perte de l'index.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_consistent(internal, data_type,
 smallint, oid, internal)
RETURNS bool
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
PG_FUNCTION_INFO_V1(my_consistent);

Datum
my_consistent(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber)
 PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    bool       *recheck = (bool *) PG_GETARG_POINTER(4);
    data_type  *key = DatumGetDataType(entry->key);
    bool        retval;

    /*
     * determine return value as a function of strategy, key and
 query.
     *
     * Use GIST_LEAF(entry) to know where you're called in the
 index tree,
     * which comes handy when supporting the = operator for
 example (you could
     * check for non empty union() in non-leaf nodes and
 equality in leaf
     * nodes).
     */

    *recheck = true;        /* or false if check is exact */
```

```
    PG_RETURN_BOOL(retval);
}
```

Ici, `key` est un élément dans l'index et `query` la valeur la recherchée dans l'index. Le paramètre `StrategyNumber` indique l'opérateur appliqué de votre classe d'opérateurs. Il correspond à un des nombres d'opérateurs dans la commande `CREATE OPERATOR CLASS`.

Suivant les opérateurs inclus dans la classe, le type de données de `query` pourrait varier avec l'opérateur car il sera du type de ce qui se trouver sur le côté droit de l'opérateur, qui pourrait être différent du type de la donnée indexée apparaissant du côté gauche. (Le squelette de code ci-dessus suppose qu'un seul type est possible ; dans le cas contraire, récupérer la valeur de l'argument `query` pourrait devoir dépendre de l'opérateur.) Il est recommendé que la déclaration SQL de la fonction `consistent` utilise le type de la donnée indexée de la classe d'opérateurs pour l'argument `query`, même si le type réel pourrait être différent suivant l'opérateur.

`union`

Cette méthode consolide l'information dans l'arbre. Suivant un ensemble d'entrées, cette fonction génère une nouvelle entrée d'index qui représente toutes les entrées données.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_union(internal, internal)
RETURNS storage_type
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
PG_FUNCTION_INFO_V1(my_union);

Datum
my_union(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *)
 PG_GETARG_POINTER(0);
    GISTENTRY  *ent = entryvec->vector;
    data_type  *out,
               *tmp,
               *old;
    int         numranges,
                i = 0;

    numranges = entryvec->n;
    tmp = DatumGetDataType(ent[0].key);
    out = tmp;

    if (numranges == 1)
    {
        out = data_type_deep_copy(tmp);

        PG_RETURN_DATA_TYPE_P(out);
    }

    for (i = 1; i < numranges; i++)
    {
```

```
            old = out;
            tmp = DatumGetDataType(ent[i].key);
            out = my_union_implementation(out, tmp);
        }

        PG_RETURN_DATA_TYPE_P(out);
    }
```

Comme vous pouvez le voir dans ce squelette, nous gérons un type de données où `union(X, Y, Z) = union(union(X, Y), Z)`. C'est assez simple pour supporter les types de données où ce n'est pas le cas, en implantant un autre algorithme d'union dans cette méthode de support GiST.

Le résultat de la fonction `union` doit être une valeur du type de stockage de l'index, quelqu'il soit (il pourrait être ou non différent du type de la colonne indexée). La fonction `union` doit renvoyer un pointeur vers la mémoire nouvellement allouée avec `palloc()`. Vous ne pouvez pas seulement renvoyer la valeur en entrée directement, même s'il n'y a pas de changement de type.

Comme indiqué ci-dessus, le premier argument `internal` de la fonction `union` est en réalité un pointeur `GistEntryVector`. Le deuxième argument est un pointeur vers une variable entière qui peut être ignorée. (Il était requis que la fonction `union` enregistre la taille de sa valeur résultat dans cette variable, mais ce n'est plus nécessaire.)

`compress`

Convertit l'élément de données dans un format compatible avec le stockage physique dans une page d'index. Si la méthode `compress` est omise, les éléments des données sont enregistrés dans l'index sans modification.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_compress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *retval;

    if (entry->leafkey)
    {
        /* replace entry->key with a compressed version */
        compressed_data_type *compressed_data =
 palloc(sizeof(compressed_data_type));

        /* fill *compressed_data from entry->key ... */

        retval = palloc(sizeof(GISTENTRY));
        gistentryinit(*retval, PointerGetDatum(compressed_data),
                      entry->rel, entry->page, entry->offset,
 FALSE);
```

```
        }
        else
        {
            /* typically we needn't do anything with non-leaf
     entries */
            retval = entry;
        }

        PG_RETURN_POINTER(retval);
    }
```

Vous devez adapter *`compressed_data_type`* au type spécifique que vous essayez d'obtenir pour compresser les nœuds finaux.

`decompress`

Convertit la représentation enregistrée d'un élément des données dans un format manipulable par les autres méthodes GiST dans la classe d'opérateur. Si la méthode `decompress` est omise, il est supposé que les autres méthodes GiST peuvent fonctionner directement dans le format de la donnée. (`decompress` n'est pas nécessairement l'inverse de la méthode `compress` ; en particulier, si `compress` est à perte, alors il est impossible pour `decompress` de reconstruire exactement la donnée originale. `decompress` n'est pas nécessairement équivalent à `fetch`, car les autres méthodes GiST pourraient ne pas nécessiter la reconstruction complète des données.)

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_decompress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
PG_FUNCTION_INFO_V1(my_decompress);

Datum
my_decompress(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(PG_GETARG_POINTER(0));
}
```

Le squelette ci-dessus est convenable dans le cas iù aucune décompression n'est nécessaire. (Mais, bien sûr, omettre la méthode est encore plus simple et même recommendé dans ce cas.)

`penalty`

Renvoie une valeur indiquant le « coût » d'insertion d'une nouvelle entrée dans une branche particulière de l'arbre. Les éléments seront insérés dans l'ordre des pénalités moindres (`penalty`) de l'arbre. Les valeurs renvoyées par `penalty` doivent être positives ou nulles. Si une valeur négative est renvoyée, elle sera traitée comme valant zéro.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_penalty(internal, internal,
 internal)
RETURNS internal
```

```
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;  -- in some cases penalty functions need not
 be strict
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
PG_FUNCTION_INFO_V1(my_penalty);

Datum
my_penalty(PG_FUNCTION_ARGS)
{
    GISTENTRY  *origentry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *newentry = (GISTENTRY *) PG_GETARG_POINTER(1);
    float      *penalty = (float *) PG_GETARG_POINTER(2);
    data_type  *orig = DatumGetDataType(origentry->key);
    data_type  *new = DatumGetDataType(newentry->key);

    *penalty = my_penalty_implementation(orig, new);
    PG_RETURN_POINTER(penalty);
}
```

Pour des raisons historiques, la fonction `penalty` ne renvoie pas seulement un résultat de type `float` ; à la place, il enregistre la valeur à l'emplacement indiqué par le troisième argument. La valeur de retour est ignorée, bien que, par convention, l'adresse de l'argument est renvoyée.

La fonction `penalty` est crucial pour de bonnes performances de l'index. Elle sera utilisée lors de l'insertion pour déterminer la branche à suivre pour savoir où ajouter la nouvelle entrée dans l'arbre. Lors de l'exécution de la requête, plus l'arbre sera bien balancé, plus l'exécution sera rapide.

`picksplit`

Quand une division de page est nécessaire pour un index, cette fonction décide des entrées de la page qui resteront sur l'ancienne page et de celles qui seront déplacées sur la nouvelle page.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_picksplit(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
PG_FUNCTION_INFO_V1(my_picksplit);

Datum
my_picksplit(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *)
 PG_GETARG_POINTER(0);
    GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
    OffsetNumber maxoff = entryvec->n - 1;
    GISTENTRY  *ent = entryvec->vector;
    GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
    int         i,
```

```
                nbytes;
    OffsetNumber *left,
                *right;
    data_type  *tmp_union;
    data_type  *unionL;
    data_type  *unionR;
    GISTENTRY **raw_entryvec;

    maxoff = entryvec->n - 1;
    nbytes = (maxoff + 1) * sizeof(OffsetNumber);

    v->spl_left = (OffsetNumber *) palloc(nbytes);
    left = v->spl_left;
    v->spl_nleft = 0;

    v->spl_right = (OffsetNumber *) palloc(nbytes);
    right = v->spl_right;
    v->spl_nright = 0;

    unionL = NULL;
    unionR = NULL;

    /* Initialize the raw entry vector. */
    raw_entryvec = (GISTENTRY **) malloc(entryvec->n *
 sizeof(void *));
    for (i = FirstOffsetNumber; i <= maxoff; i =
 OffsetNumberNext(i))
        raw_entryvec[i] = &(entryvec->vector[i]);

    for (i = FirstOffsetNumber; i <= maxoff; i =
 OffsetNumberNext(i))
    {
        int         real_index = raw_entryvec[i] - entryvec-
>vector;

        tmp_union = DatumGetDataType(entryvec-
>vector[real_index].key);
        Assert(tmp_union != NULL);

        /*
         * Choose where to put the index entries and update
 unionL and unionR
         * accordingly. Append the entries to either v->spl_left
 or
         * v->spl_right, and care about the counters.
         */

        if (my_choice_is_left(unionL, curl, unionR, curr))
        {
            if (unionL == NULL)
                unionL = tmp_union;
            else
                unionL = my_union_implementation(unionL,
 tmp_union);

            *left = real_index;
            ++left;
            ++(v->spl_nleft);
```

```
        }
        else
        {
            /*
             * Same on the right
             */
        }
    }

    v->spl_ldatum = DataTypeGetDatum(unionL);
    v->spl_rdatum = DataTypeGetDatum(unionR);
    PG_RETURN_POINTER(v);
}
```

Notez que le résultat de la fonction `picksplit` est fourni en modifiant la structure `v` en référence. La valeur de retour réelle est ignorée, bien que la convention est de passer l'adresse de `v`.

Comme `penalty`, la fonction `picksplit` est cruciale pour de bonnes performances de l'index. Concevoir des implantations convenables des fonctions `penalty` et `picksplit` est le challenge d'un index GiST performant.

`same`

Renvoie true si les deux entrées de l'index sont identiques, faux sinon. (Un « enregistrement d'index » est une valeur du type de stockage de l'index, pas nécessairement le type original de la colonne indexée.)

La déclaration SQL de la fonction ressemble à ceci :

```
CREATE OR REPLACE FUNCTION my_same(storage_type, storage_type,
 internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut alors suivre ce squelette :

```
PG_FUNCTION_INFO_V1(my_same);

Datum
my_same(PG_FUNCTION_ARGS)
{
    prefix_range *v1 = PG_GETARG_PREFIX_RANGE_P(0);
    prefix_range *v2 = PG_GETARG_PREFIX_RANGE_P(1);
    bool       *result = (bool *) PG_GETARG_POINTER(2);

    *result = my_eq(v1, v2);
    PG_RETURN_POINTER(result);
}
```

Pour des raisons historiques, la fonction `same` ne renvoie pas seulement un résultat booléen ; à la place, il doit enregistrer le drapeau à l'emplacement indiqué par le troisième argument. La valeur de retour est ignoré, bien qu'il soit par convention de passer l'adresse de cet argument.

`distance`

À partir d'une entrée d'index `p` et une valeur recherchée `q`, cette fonction détermine la « distance » entre l'entrée de l'index et la valeur recherchée. Cette fonction doit être fournie si la classe d'opérateurs contient des opérateurs de tri. Une requête utilisant l'opérateur de tri sera implémentée en renvoyant les entrées d'index dont les valeurs de « distance » sont les plus petites, donc les résultats doivent être cohérents avec la sémantique de l'opérateur. Pour une entrée d'index de type feuille, le résultat représente seulement la distance vers l'entrée d'index. Pour un nœud de l'arbre interne, le résultat doit être la plus petite distance que toute entrée enfant représente.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_distance(internal, data_type,
 smallint, oid, internal)
RETURNS float8
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

Et le code correspondant dans le module C peut correspondre à ce squelette :

```
PG_FUNCTION_INFO_V1(my_distance);

Datum
my_distance(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber)
 PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    /* bool *recheck = (bool *) PG_GETARG_POINTER(4); */
    data_type  *key = DatumGetDataType(entry->key);
    double      retval;

    /*
     * determine return value as a function of strategy, key and
 query.
     */

    PG_RETURN_FLOAT8(retval);
}
```

Les arguments de la fonction `distance` sont identiques aux arguments de la fonction `consistent`.

Quelques approximations sont autorisées pour déterminer la distance, pour que le résultat ne soit jamais plus grand que la distance réelle de l'entrée. De ce fait, par exemple, une distance dans une *bounding box* est généralement suffisante dans les applications géométriques. Pour un nœud d'un arbre interne, la distance renvoyée ne doit pas être plus grande que la distance vers tous les nœuds cibles. Si la distance renvoyée n'est pas exacte, la fonction doit configurer `*recheck` à true. (Ceci n'est pas nécessaire pour les nœuds de l'arbre interne ; en ce qui les concerne, le calcul est supposé toujours inexact.) Dans ce cas, l'exécuteur calculera la distance précise après la récupération de la ligne à partir de la pile, et réordonnera les lignes si nécessaires.

Si la fonction distance renvoie `*recheck = true` pour tout nœud feuille, le type de retour de l'opération de tri original doit être `float8` ou `float4`, et les valeurs résultats de la fonction

distance doivent être comparables à ceux de l'opérateur original de tri, car l'exécuteur triera en utilisant les résultats de la fonction de distance et les résultats recalculés de l'opérateur de tri. Dans le cas contraire, les valeurs de résultats de la fonction distance peuvent être toute valeur `float8` finie, tant est que l'ordre relatif des valeurs résultats correspond à l'ordre renvoyé par l'opérateur de tri. (l'infinité, positif comme négatif, est utilisé en interne pour gérer des cas comme les valeurs NULL, donc il n'est pas recommandé que les fonctions `distance` renvoient ces valeurs.)

`fetch`

Convertit la représentation compressée de l'index pour un élément de données vers le type de données original pour les parcours d'index seuls. Les données renvoyées doivent être une copie exacte, sans perte de la valeur indexée à l'origine.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_fetch(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

L'argument est un pointeur vers une structure `GISTENTRY`. En entrée, son champ `key` contient une donnée non NULL compressée. La valeur de retour est une autre structure `GISTENTRY` dont le champ `key` contient la même donnée que l'original, mais non compressée. Si la fonction de compression de la classe d'opérateurs ne fait rien pour les enregistrements feuilles, la méthode `fetch` peut renvoyer l'argument tel quel. Ou, si la classe d'opérateurs n'a pas de fonction de compression, la méthode `fetch` peut aussi être omise car elle ne ferait rien de toute façon.

Le code correspondant dans le module C doit alors suivre ce squelette :

```
PG_FUNCTION_INFO_V1(my_fetch);

Datum
my_fetch(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    input_data_type *in = DatumGetPointer(entry->key);
    fetched_data_type *fetched_data;
    GISTENTRY  *retval;

    retval = palloc(sizeof(GISTENTRY));
    fetched_data = palloc(sizeof(fetched_data_type));

    /*
     * Convertit 'fetched_data' en un Datum du type de données
 original.
     */

    /* remplit *retval à partir de fetched_data. */
    gistentryinit(*retval, PointerGetDatum(converted_datum),
                  entry->rel, entry->page, entry->offset,
 FALSE);

    PG_RETURN_POINTER(retval);
}
```

Si la méthode de compression est à perte pour les entrées feuilles, la classe d'opérateurs ne supporte pas les parcours d'index seuls, et ne doit pas définir une fonction `fetch`.

`options`

Permet la définition des paramètres visibles par l'utilisateur et contrôlant le comportement des classes d'opérateurs.

La déclaration SQL de la fonction doit ressembler à ça :

```
CREATE OR REPLACE FUNCTION my_options(internal)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

La fonction est passée sous forme de pointeur à une structure `local_relopts`, qui doit être nécessairement alimentée avec un ensemble d'options spécifiques de classe d'opérateurs. Les options peuvent être accédées par d'autres fonctions support en utilisant les macros `PG_HAS_OPCLASS_OPTIONS()` et `PG_GET_OPCLASS_OPTIONS()`.

Un exemple d'implémentation de my_options() et de l'utilisation des paramètres depuis une autre fonction support est donné ci-dessous :

```
typedef enum MyEnumType
{
    MY_ENUM_ON,
    MY_ENUM_OFF,
    MY_ENUM_AUTO
} MyEnumType;

typedef struct
{
    int32   vl_len_;    /* varlena header (do not touch
 directly!) */
    int     int_param;  /* integer parameter */
    double  real_param; /* real parameter */
    MyEnumType enum_param; /* enum parameter */
    int     str_param;  /* string parameter */
} MyOptionsStruct;

/* String representation of enum values */
static relopt_enum_elt_def myEnumValues[] =
{
    {"on", MY_ENUM_ON},
    {"off", MY_ENUM_OFF},
    {"auto", MY_ENUM_AUTO},
    {(const char *) NULL}   /* list terminator */
};

static char *str_param_default = "default";

/*
 * Sample validator: checks that string is not longer than 8
 bytes.
 */
static void
validate_my_string_relopt(const char *value)
```

```
{
    if (strlen(value) > 8)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_PARAMETER_VALUE),
                 errmsg("str_param must be at most 8 bytes")));
}

/*
 * Sample filler: switches characters to lower case.
 */
static Size
fill_my_string_relopt(const char *value, void *ptr)
{
    char   *tmp = str_tolower(value, strlen(value),
 DEFAULT_COLLATION_OID);
    int     len = strlen(tmp);

    if (ptr)
        strcpy((char *) ptr, tmp);

    pfree(tmp);
    return len + 1;
}

PG_FUNCTION_INFO_V1(my_options);

Datum
my_options(PG_FUNCTION_ARGS)
{
    local_relopts *relopts = (local_relopts *)
 PG_GETARG_POINTER(0);

    init_local_reloptions(relopts, sizeof(MyOptionsStruct));
    add_local_int_reloption(relopts, "int_param", "integer
 parameter",
                            100, 0, 1000000,
                            offsetof(MyOptionsStruct,
 int_param));
    add_local_real_reloption(relopts, "real_param", "real
 parameter",
                             1.0, 0.0, 1000000.0,
                             offsetof(MyOptionsStruct,
 real_param));
    add_local_enum_reloption(relopts, "enum_param", "enum
 parameter",
                             myEnumValues, MY_ENUM_ON,
                             "Valid values are: \"on\", \"off\"
 et \"auto\".",
                             offsetof(MyOptionsStruct,
 enum_param));
    add_local_string_reloption(relopts, "str_param", "string
 parameter",
                               str_param_default,
                               &validate_my_string_relopt,
                               &fill_my_string_relopt,
                               offsetof(MyOptionsStruct,
 str_param));
```

```
    PG_RETURN_VOID();
}

PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    int     int_param = 100;
    double  real_param = 1.0;
    MyEnumType enum_param = MY_ENUM_ON;
    char   *str_param = str_param_default;

    /*
     * Normally, when opclass contains 'options' method, then
 options are always
     * passed to support functions.  However, if you add
 'options' method to
     * existing opclass, previously defined indexes have no
 options, so the
     * check is required.
     */
    if (PG_HAS_OPCLASS_OPTIONS())
    {
        MyOptionsStruct *options = (MyOptionsStruct *)
 PG_GET_OPCLASS_OPTIONS();

        int_param = options->int_param;
        real_param = options->real_param;
        enum_param = options->enum_param;
        str_param = GET_STRING_RELOPTION(options, str_param);
    }

    /* the rest implementation of support function */
}
```

Comme la représentation de la clé dans GiST est flexible, elle peut dépendre des paramètres définis par l'utilisateur. Voir `gtsvector_options()` pour exemple.

`sortsupport`

Renvoie une fonction de comparaison pour trier lesdonnées d'une façon qui préserve l'emplacement. Elle est utilisée par les commandes `CREATE INDEX` et `REINDEX`. La qualité de l'index créé dépend decomment l'ordre de tri déterminé par la fonction de comparaison préserve l'emplacement des entrées.

La méthode `sortsupport` est optionnelle. Si elle n'est pas fournie, `CREATE INDEX` construit l'index en insérant chaque ligne dans l'arbre en utilisant les fonctions `penalty` et `picksplit`, qui sont plus lentes.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_sortsupport(internal)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

L'argument est un pointeur vers une structure `SortSupport`. Au minimum, la fonction doit remplir son champ de comparaison. La fonction prends trois arguments : deux Datums à comparer, et un pointeur vers la structure `SortSupport`. Les Datums sont les deux valeurs indexées dans le format où elles sont stockées dans l'index ; c'est-à-dire dans le format renvoyé par la méthode `compress`. L'API complète est définie dans `src/include/utils/sortsupport.h`.

Le code correspondant dans un module C pourrait suivre ce squelette :

```
PG_FUNCTION_INFO_V1(my_sortsupport);

static int
my_fastcmp(Datum x, Datum y, SortSupport ssup)
{
  /* établir l'ordre entre x et y en calculant une valeur de tri
 z */

  int z1 = ComputeSpatialCode(x);
  int z2 = ComputeSpatialCode(y);

  return z1 == z2 ? 0 : z1 > z2 ? 1 : -1;
}

Datum
my_sortsupport(PG_FUNCTION_ARGS)
{
  SortSupport ssup = (SortSupport) PG_GETARG_POINTER(0);

  ssup->comparator = my_fastcmp;
  PG_RETURN_VOID();
}
```

Toutes les méthodes de support GiST sont habituellement appelées dans des contextes mémoires à durée limitée. En fait, `CurrentMemoryContext` sera réinitialisé après le traitement de chaque ligne. Il n'est donc pas très important de s'inquiéter de libérer avec pfree tout ce que vous avez alloué avec palloc. Néanmoins, dans certains cas, une méthode de support peut avoir besoin de cacher des données à utiliser lors des prochains appels. Pour cela, allouez les données à durée de vie longue dans `fcinfo->flinfo->fn_mcxt` et conservez un pointeur vers ces données dans `fcinfo->flinfo->fn_extra`. Ce type de données va survivre pendant toute la durée de l'opération sur l'index (par exemple, un seul parcours d'index GiST, une construction d'index ou l'insertion d'une ligne dans un index). Faites attention à libérer avec pfree la valeur précédente lors du remplacement d'une valeur `fn_extra`. Dans le cas contraire, une perte mémoire s'accumulera pendant la durée de l'opération.

68.4. Implémentation

68.4.1. Méthodes de construction d'index GiST

La façon la plus simple de construire un index GiST est justement d'insérer toutes les entrées, une par une. Ceci tend à être lent pour les index volumineux parce que, si les entrées d'index sont réparties partout dans l'index et que l'index est suffisamment gros pour ne pas tenir en cache, un grand nombre d'accès disque aléatoires seront nécessaires. PostgreSQL accepte deux méthodes alternatives pour une construction initiale d'un indexGiST : le mode *sorted* (trié) et le mode *buffered* (tampon).

La méthode par tri est seulement disponible si chacune des classes d'opérateur utilisée par l'index fournit une fonction `sortsupport`, comme décrit dans Section 68.3. Si c'est le cas, cette méthode est généralement la meilleure, donc elle est utilisée par défaut.

La méthode par cache fonctionne en n'insérant pas les lignes directement et immédiatement dans l'index. Il peut réduire fortement la quantité d'accès disques aléatoires nécessaire pour les ensembles de données non triés. Pour les ensembles de données bien triés, le bénéfice est plus petit, voire inexistant car seul un petit nombre de blocs reçoit de nouvelles lignes à un instant t, et ces blocs tiennent en cache même si l'index complet ne le peut pas.

La méthode par cache a besoin d'appeler la fonction `penalty` plus souvent que ne le fait la méthode simple, ce qui consomme des ressources CPU supplémentaires. De plus, le cache a besoin d'un espace disque supplémentaire allant jusqu'à la taille de l'index résultant. L'utilisation de tampons peut aussi influencer la qualité de l'index résultant, de façon positive et négative. Cette influence dépend de plusieurs facteurs, comme la distribution des données en entrée et de l'implémentation de la classe d'opérateurs.

Si le tri n'est pas possible, alors par défaut la construction d'un index GiST bascule sur la méthode par cache quand la taille de l'index atteint effective_cache_size. L'utilisation du cache peut être forcée ou empêchée manuellement avec le paramètre `buffering` de la commande CREATE INDEX. Le comportement par défaut est bon dans la plupart des cas mais désactiver le cache pourrait accélérer un peu la construction si les données en entrée sont triées.

68.5. Exemples

La distribution source de PostgreSQL inclut plusieurs exemples de méthodes d'indexation implémentées selon GiST. Le système principal fournit des fonctionnalités de recherche plein texte (indexation des `tsvector` et `tsquery`) ainsi que des fonctionnalités équivalentes aux R-Tree pour certains types de données géométriques (voir `src/backend/access/gist/gistproc.c`). Les modules `contrib` suivants contiennent aussi des classes d'opérateurs GiST :

`btree_gist`

Fonctionnalités équivalentes aux B-Tree pour plusieurs types de données

`cube`

Indexation de cubes multi-dimensionnels

`hstore`

Module pour le stockage des paires (clé, valeur)

`intarray`

RD-Tree pour tableaux uni-dimensionnels de valeurs int4

`ltree`

Indexation des structures de type arbre

`pg_trgm`

Similarité textuelle par correspondance de trigrammes

`seg`

Indexation pour les « nombres flottants »

Chapitre 69. Index SP-GiST

69.1. Introduction

SP-GiST est une abréviation pour les espaces géographiques partitionnées avec GiST. SP-GiST supporte les arbres de recherche partitionnés, qui facilitent le développement d'un grand nombre de structures de données non balancées différentes, comme les *quadtree*, les arbres k-d et les arbres de *radix*. Le principal intérêt de ces structures et la division régulière de l'espace de recherche en partitions de taille égales. Les recherches qui correspondent bien avec la règle de partitionnement peuvent être très rapides.

Ces fameuses structures de données ont été initialement conçues pour une exécution en mémoire. Dans la mémoire principale, elles sont généralement conçues comme un ensemble de nœuds alloués dynamiquement et reliés entre eux par des pointeurs. Cette organisation ne peut pas être transposée directement sur disque car ces suites de pointeurs peuvent nécessiter un nombre d'accès disque trop important. Au contraire, les structures de données adaptées au disque devraient permettre de charger simultanément un grand nombre de données (*high fanout*) pour minimiser les accès disque. Le challenge proposé par SP-GiST est de faire correspondre les nœuds des arbres de recherche avec les pages du disque de manière à ce qu'une recherche ne nécessite qu'un faible nombre d'accès disque, même si il nécessite de traverser plusieurs nœuds.

Tout comme GiST, SP-GiST est destiné à permettre le développement de types de données personnalisées, disposant des méthodes d'accès appropriées, par un expert du domaine plutôt que par un expert en base de données.

Une partie des informations fournies ici sont extraites du site web[1] du projet d'indexation SP-GiST de l'université Purdue. L'implémentation de SP-GiST dans PostgreSQL est principalement maintenue par Teodor Sigaev et Oleg Bartunov, plus d'informations sont disponibles sur leur site web[2].

69.2. Classes d'opérateur internes

La distribution de PostgreSQL inclut les classes d'opérateur SP-GiST indiquées dans Tableau 69.1.

Tableau 69.1. Classes d'opérateur SP-GiST internes

Nom	Opérateurs indexables	Opérateurs d'ordre
box_ops	<< (box,box)	<-> (box,point)
	&< (box,box)	
	&> (box,box)	
	>> (box,box)	
	<@ (box,box)	
	@> (box,box)	
	~= (box,box)	
	&& (box,box)	
	<<| (box,box)	
	&<| (box,box)	
	|&> (box,box)	
	|>> (box,box)	
inet_ops	<< (inet,inet)	

[1] https://www.cs.purdue.edu/spgist/
[2] http://www.sai.msu.su/~megera/wiki/spgist_dev

Nom	Opérateurs indexables	Opérateurs d'ordre
	<<= (inet,inet)	
	>> (inet,inet)	
	>>= (inet,inet)	
	= (inet,inet)	
	<> (inet,inet)	
	< (inet,inet)	
	<= (inet,inet)	
	> (inet,inet)	
	>= (inet,inet)	
	&& (inet,inet)	
kd_point_ops	|>> (point,point)	<-> (point,point)
	<< (point,point)	
	>> (point,point)	
	<<| (point,point)	
	~= (point,point)	
	<@ (point,box)	
poly_ops	<< (polygon,polygon)	<-> (polygon,point)
	&< (polygon,polygon)	
	&> (polygon,polygon)	
	>> (polygon,polygon)	
	<@ (polygon,polygon)	
	@> (polygon,polygon)	
	~= (polygon,polygon)	
	&& (polygon,polygon)	
	<<| (polygon,polygon)	
	&<| (polygon,polygon)	
	|>> (polygon,polygon)	
	|&> (polygon,polygon)	
quad_point_ops	|>> (point,point)	<-> (point,point)
	<< (point,point)	
	>> (point,point)	
	<<| (point,point)	
	~= (point,point)	
	<@ (point,box)	
range_ops	= (anyrange,anyrange)	
	&& (anyrange,anyrange)	
	@> (anyrange,anyelement)	
	@> (anyrange,anyrange)	

Nom	Opérateurs indexables	Opérateurs d'ordre
	<@ (anyrange,anyrange)	
	<< (anyrange,anyrange)	
	>> (anyrange,anyrange)	
	&< (anyrange,anyrange)	
	&> (anyrange,anyrange)	
	-|- (anyrange,anyrange)	
text_ops	= (text,text)	
	< (text,text)	
	<= (text,text)	
	> (text,text)	
	>= (text,text)	
	~<~ (text,text)	
	~<=~ (text,text)	
	~>=~ (text,text)	
	~>~ (text,text)	
	^@ (text,text)	

Sur les deux classes d'opérateurs pour le type `point`, `quad_point_ops` est celui par défaut. `kd_point_ops` gère les mêmes opérateurs mais utilise une structure de données différente pour l'index, structure pouvant offrir de meilleures performances pour certaines utilisations.

Les classes d'opérateurs `quad_point_ops`, `kd_point_ops` et `poly_ops` supportent l'ordre d'opérateur `<->`, qui active la recherche de type voisin-le-plus-proche (`k-NN`) sur des ensembles de données composés de point ou polygon.

69.3. Extensibilité

SP-GiST offre une interface avec un haut niveau d'abstraction, imposant au développeur des méthodes d'accès de n'implémenter que des méthodes spécifiques à un type de donnée spécifié. Le cœur de SP-GiST est responsable de l'efficacité du stockage sur le disque et de la recherche dans la structure arborescente. Il s'occupe aussi de la concurrence d'accès et des journaux.

Les lignes des feuilles d'un arbre SP-GiST contiennent habituellement des valeurs du même type de données que la colonne indexée, bien qu'il soit possible qu'ils contiennent des représentations à perte de la colonne indexée. Les enregistrements des feuilles stockés à la racine représenteront directement la valeur originale de la donnée indexée, mais les enregistrements des feuilles à des niveaux plus bas pourraient ne contenir qu'une valeur partielle, telle qu'un suffixe. Dans ce cas, les classes d'opérateurs des fonctions supportées devront être capables de reconstruire la valeur originale en utilisant les informations accumulées dans les lignes intermédiaires au travers du parcours de l'arbre et vers le niveau le plus bas.

Quand un index SP-GiST est créé avec des colonnes `INCLUDE`, les valeurs de ces colonnes sont aussi stockées dans des enregistrements feuilles. Les colonnes `INCLUDE` ne concernent pas la classe d'opérateurs SP-GiST, donc elles ne seront pas discutées avec plus de détails ici.

Les lignes internes sont plus complexes car elles relient des points dans l'arbre de recherche. Chaque ligne intermédiaire contient un ensemble d'au moins un *nœud*, qui représente des groupes de valeurs similaires de feuilles. Un nœud contient un lien qui mène vers un autre nœud de niveau inférieur, ou une petite liste de lignes de feuilles qui appartiennent toutes à la même page d'index. Chaque nœud a un *label* qui le décrit. Par exemple, dans un arbre *radix*, le label du nœud peut être le caractère suivant de la chaîne de caractère. (Sinon, une classe d'opérateurs peut omettre les labels des nœuds si elle fonctionne avec un ensemble fixe de nœuds pour les enregistrements internes ; voir Section 69.4.2.) En option, une ligne intermédiaire peut avoir une valeur de *préfixe* qui décrit tous ses membres. Dans un arbre *radix*, cela peut être le préfixe commun des chaînes représentant les données. La valeur du préfixe n'est pas nécessairement réellement un préfixe, mais peut être toute donnée utilisée par la classe d'opérateurs. Par exemple, pour un *quadtree*, il peut stocker le barycentre des quatre points représenté par chaque feuille. Une ligne intermédiaire d'un *quadtree* contiendra aussi quatre nœuds correspondants à des points autour de ce point central.

Quelques algorithmes de recherche arborescente nécessite la connaissance du niveau (ou profondeur) de la ligne en cours, et ainsi le cœur de SP-GiST fournit aux classes d'opérateurs la possibilité de gérer le décompte des niveaux lors du parcours de l'arbre. Il fournit aussi le moyen de reconstruire de façon incrémentale la valeur représentée lorsque cela est nécessaire, et pour passer des données supplémentaires (appelées *valeurs traverses*) lors de la descente de l'arbre.

> **Note**
>
> Le code du cœur de SP-GiST tient aussi compte des valeurs NULL. Bien que les index SP-GiST stockent des entrées pour les valeurs NULL dans les colonnes indexées, cette implémentation reste non apparente au code de l'index de classe d'opérateur : aucune valeur NULL d'index ou de condition de recherche ne sera jamais transmis aux méthodes de la classe d'opérateurs (il est convenu que les opérateurs SP-GiST sont stricts et ainsi ne peuvent trouver des valeurs NULL). Le cas des valeurs NULL n'est ainsi plus abordé dans les paragraphes qui suivent.

Un index de classe d'opérateurs pour SP-GiST peut proposer cinq méthodes personnalisées, et deux optionnelles. Chacune de ces cinq méthodes obligatoires doit suivre la convention qui consiste à accepter deux arguments de type `internal`, le premier étant un pointeur vers une structure C contenant les valeurs en entrée de cette méthode, et le second étant un pointeur vers une structure C où les valeurs en sortie seront placées. Quatre de ces méthodes retournent `void` car leurs résultats sont présents dans la structure en sortie. Mais la méthode `leaf_consistent` retourne une valeur de type `boolean`. Les méthodes ne doivent modifier aucun des champs de la structure en entrée. Dans tous les cas, la structure en sortie est initialisée avec des zéros avant l'appel à la méthode personnalisée. La sixième méthode, optionnelle, `compress` accepte un `datum` à indexer comme seul argument et renvoie une valeur convenable pour un enregistrement physique dans une ligne feuille. La septième méthode, optionnelle, est appelée `options` et accepte un pointeur `internal` vers une structure C, remplie avec les paramètres spécifiques à la classe d'opérateurs, et renvoie `void`.

Les cinq méthodes personnalisées sont :

`config`

Retourne des informations statiques concernant l'implémentation des index, incluant les OID du type de données du préfixe et le type de données du label du nœud.

La déclaration SQL de la fonction doit ressembler à :

```
CREATE FUNCTION ma_configuration(internal, internal) RETURNS
 void ...
```

Le premier argument est un pointeur vers une structure C `spgConfigIn`, qui contient les données en entrée de la fonction. Le second argument est un pointeur vers une structure C `spgConfigOut`, qui permet à la fonction d'y spécifier les données en sortie.

```
typedef struct spgConfigIn
{
    Oid         attType;        /* Le type de donnée à indexer
 */
} spgConfigIn;

typedef struct spgConfigOut
{
    Oid         prefixType;     /* Le type de donnée des préfixe
 des tuples intermédiaires */
    Oid         labelType;      /* Le type de donnée des labels
 de nœud des tuples intermédiaires */
    Oid         leafType;       /* Type de données pour les
 valeurs de tuple feuille */
    bool        canReturnData;  /* Opclass peut reconstruire les
 données originales */
    bool        longValuesOK;   /* Opclass sait gérer les
 valeurs plus grandes qu'une page */
} spgConfigOut;
```

`attType` est fourni pour gérer les index polymorphiques de classe d'opérateurs. Pour les types de données ordinaires de classe d'opérateurs (fixés), il aura toujours la même valeur et peut ainsi être ignoré.

Pour les classes d'opérateurs qui n'utilisent pas de préfixe, `prefixType` peut être défini à `VOIDOID`. De la même façon, pour les classes d'opérateurs qui n'utilisent pas de label de nœud, `labelType` peut être défini à `VOIDOID`. `canReturnData` peut être défini à true si la classe d'opérateurs est capable de reconstruire la valeur d'index fournie initialement. `longValuesOK` doit être défini à true uniquement lorsque `attType` est de longueur variable et que la classe d'opérateur est capable de segmenter les grandes valeurs en répétant les suffixes (voir Section 69.4.1).

`leafType` doit correspondre au type de stockage de l'index défini par l'entrée de catalogue `opckeytype` de la classe d'opérateurs. (Notez que `opckeytype` peut valoir zéro, impliquant que le type de stockage est le même que le type en entrée de la classe d'opérateurs, ce qui est la situation la plus commune.) Pour des raisons de compatibilité ascendante, la méthode `config` peut configurer `leafType` à toute autre valeur, et cette valeur sera utilisée ; mais ceci est abandonné car le contenu de l'index est alors incorrectement identifié dans les catalogues. De plus, il est autorisé de laisser leave `leafType` non initialisé (zéro) ; ceci est interprété comme signifiant que le type de stockage de l'index est dérivé de `opckeytype`.

Quand `attType` et `leafType` sont différents, alors la méthode optionnelle `compress` doit être fournie. La méthode `compress` est responsable de la transformation des datums pour les indexer de `attType` vers `leafType`.

`choose`

Choisit une méthode pour insérer une nouvelle valeur dans une ligne intermédiaire.

La déclaration SQL de la fonction doit ressembler à :

```
CREATE FUNCTION mon_choix(internal, internal) RETURNS void ...
```

Le premier argument est un pointeur vers une structure C `spgChooseIn`, qui contient les données en entrée de la fonction. Le second argument est un pointeur vers une structure C `spgChooseOut`, qui permet à la fonction d'y spécifier les données en sortie.

```
typedef struct spgChooseIn
{
    Datum       datum;          /* donnée initiale à indexer */
    Datum       leafDatum;      /* donnée en cours à stocker
 dans la feuille */
    int         level;          /* niveau en cours (à partir de
 0) */

    /* Données issues de la ligne intermédiaire */
    bool        allTheSame;     /* la ligne contient des valeurs
 équivalentes ? */
    bool        hasPrefix;      /* la ligne a-t-elle un préfixe?
 */
    Datum       prefixDatum;    /* si c'est le cas, la valeur de
 ce préfixe */
    int         nNodes;         /* nombre de nœuds dans la ligne
 intermédiaire */
    Datum      *nodeLabels;     /* valeurs du label du nœud
 (NULL sinon) */
} spgChooseIn;

typedef enum spgChooseResultType
{
    spgMatchNode = 1,           /* descend dans le nœud existant
 */
    spgAddNode,                 /* ajoute un nœud dans la ligne
 intermédiaire */
    spgSplitTuple               /* scinde une ligne
 intermédiaire (modifie son préfixe) */
} spgChooseResultType;

typedef struct spgChooseOut
{
    spgChooseResultType resultType;     /* code d'action, voir
 plus bas */
    union
    {
        struct                  /* resultats de spgMatchNode */
        {
            int         nodeN;      /* descend dans ce nœud (à
 partir de 0) */
            int         levelAdd;   /* incrémente le niveau de
 cette valeur */
            Datum       restDatum;  /* nouvelle valeur de la
 feuille */
        }           matchNode;
        struct                  /* résultats de spgAddNode */
        {
            Datum       nodeLabel;  /* nouveau label du nœud */
            int         nodeN;      /* là où l'insérer (à partir
 de 0) */
```

```
        }           addNode;
        struct                  /* résultats pour spgSplitTuple
 */
        {
    /* Information pour former une ligne de niveau supérieur
 avec un nœud fils */
            bool        prefixHasPrefix;    /* la ligne doit-
elle avoir un préfixe ? */
            Datum       prefixPrefixDatum;  /* si oui, sa valeur
 */
            int         prefixNNodes;       /* nombre de nœuds
 */
            Datum      *prefixNodeLabels;   /* leurs labels (ou
 NULL si
                                             * aucun label) */
            int         childNodeN;         /* quel nœud a un
 nœud fils */

            /* Informations pour former une nouvelle ligne
 intermédaire de niveau inférieur
   à partir de tous les anciens nœuds */
            bool        postfixHasPrefix;   /* la ligne doit-
elle avoir un préfixe ? */
            Datum       postfixPrefixDatum; /* si oui, sa valeur
 */
        }           splitTuple;
    }           result;
} spgChooseOut;
```

`datum` est la valeur initiale de type `spgConfigIn.attType` de la donnée qui a été insérée dans l'index. `leafDatum` est une valeur de type `spgConfigOut.leafType` qui est initialement un résultat de la méthode `compress` appliquée à `datum` quand la méthode `compress` est fournie, ou la même valeur que `datum` dans le cas contraire. `leafDatum` peut changer à des niveaux inférieurs de l'arbre si la fonction `choose` ou `picksplit` change cette valeur. Lorsque la recherche liée à l'insertion atteint une feuille, la valeur actuelle de `leafDatum` sera stockée dans la nouvelle ligne de feuille créée. `level` est le niveau actuel de la ligne intermédiaire, en considérant que 0 est le niveau racine. `allTheSame` est true si la ligne intermédiaire actuelle est marquée comme contenant plusieurs nœuds équivalents. (voir Section 69.4.3). `hasPrefix` est vrai si la ligne intermédiaire actuelle contient un préfixe ; si c'est le cas, `prefixDatum` est sa valeur. `nNodes` est le nombre de nœuds enfants contenus dans la ligne intermédiaire, et `nodeLabels` est un tableau des valeurs de leurs labels, ou NULL s'il n'y a pas de labels.

La fonction `choose` peut déterminer si la nouvelle valeur correspond à un des nœuds enfants existants, ou si un nouvel enfant doit être ajouté, ou si la nouvelle valeur n'est pas consistante avec les préfixes de ligne et qu'ainsi la ligne intermédiaire doit être découpée pour créer un préfixe moins restrictif.

Si la nouvelle valeur correspond à un des nœuds enfants existants, définir `resultType` à `spgMatchNode`. et définir `nodeN` à l'index (à partir de 0) du nœud dans le tableau de nœud. Définir `levelAdd` à l'incrément de `level` nécessaire pour descendre au travers de ce nœud, ou le laisser à 0 si la classe d'opérateurs n'utilise pas de niveaux. Définir `restDatum` à la valeur de `leafDatum` si la classe d'opérateurs ne modifie pas les valeurs d'un niveau au suivant, ou dans le cas contraire, définir la valeur modifiée pour être utilisée comme valeur de `leafDatum` au niveau suivant.

Si un nouveau nœud enfant doit être ajouté, définir `resultType` à `spgAddNode`. Définir `nodeLabel` au label à utiliser pour le nouveau nœud, et définir `nodeN` à l'index (de 0) auquel insérer le nœud dans le tableau de nœud. Après que ce nœud a été ajouté, la fonction `choose`

sera appelée à nouveau avec la ligne intermédiaire modifiée. Cet appel devrait produire un résultat `spgMatchNode`.

Si la nouvelle valeur est cohérente avec le préfixe de ligne, définir `resultType` à `spgSplitTuple`. Cette action déplace tous les nœuds existants dans le nouveau niveau inférieur de la ligne intermédiaire, et remplace la ligne intermédiaire existant avec une ligne qui dispose d'un unique nœud qui est lié à la nouvelle ligne intermédiaire de niveau inférieur. Définir `prefixHasPrefix` pour indiquer si les nouvelles lignes supérieures doivent avoir un préfixe, et si c'est le cas, définir `prefixPrefixDatum` à la valeur du préfixe. Cette nouvelle valeur de préfixe doit être suffisamment moins restrictive que l'original pour accepter que la nouvelle valeur soit indexée. Définir `prefixNNodes` au nombre de nœuds nécessaires pour la nouvelle ligne et définir `prefixNodeLabels` à un tableau alloué avec palloc de leurs labels, ou à NULL si les labels des nœuds ne sont pas nécessaires. Noter que la taille totale de la nouvelle ligne supérieure ne doit pas dépasser la taille totale de la ligne qu'elle remplace ; cela contraint les longueurs des nouveaux préfixes et labels. Définir `postfixHasPrefix` pour indiquer si la nouvelle ligne intermédiaire de niveau inférieur aura un préfixe, et dans ce cas définir `postfixPrefixDatum` à la valeur du préfixe. La combinaison de ces deux préfixes et le label additionnel doit avoir la même signification que le préfixe original car il n'y a pas de moyen de modifier le label du nœud qui est déplacé vers la nouvelle ligne de niveau inférieur, ni de modifier une quelconque entrée d'index enfant. Après que ce nœud ait été découpé, la fonction `choose` sera appelée à nouveau avec la ligne intermédiaire de remplacement. Cet appel devrait retourner un `spgAddNode` car, à priori, le label du nœud ajouté lors de l'étape de découpage ne correspondra pas à la nouvelle valeur. Ainsi, après cette étape, il y aura une troisième étape qui retournera finalement `spgMatchNode` et permettra l'insertion pour descendre au niveau feuille.

`picksplit`

Décide de la manière à suivre pour créer une ligne intermédiaire à partir d'un ensemble de lignes de feuilles.

La déclaration de fonction SQL doit ressembler à :

```
CREATE FUNCTION mon_decoupage(internal, internal) RETURNS
 void ...
```

Le premier argument est un pointeur vers une structure C `spgPickSplitIn`, qui contient les données en entrée de la fonction. Le second argument est un pointeur vers une structure C `spgPickSplitOut`, qui permet à la fonction d'y spécifier les données en sortie.

```
typedef struct spgPickSplitIn
{
    int         nTuples;        /* nombre de lignes feuilles */
    Datum      *datums;         /* leur données (tableau de
 taille nTuples) */
    int         level;          /* niveau actuel (à partir de 0)
 */
} spgPickSplitIn;

typedef struct spgPickSplitOut
{
    bool        hasPrefix;      /* les nouvelles lignes
 intermédiaires doivent-elles avoir un préfixe ? */
    Datum       prefixDatum;    /* si oui, la valeur du préfixe
 */

    int         nNodes;         /* nombre de nœud pour une
 nouvelle ligne intermédiaire */
```

```
    Datum      *nodeLabels;     /* leurs labels (ou NULL s'il
 n'y a aucun label) */

    int        *mapTuplesToNodes;   /* index du nœud de chaque
 ligne feuille */
    Datum      *leafTupleDatums;    /* données à stocker dans
 chaque nouvelle ligne feuille */
} spgPickSplitOut;
```

`nTuples` est le nombre de lignes feuilles fournies. `datums` est un tableau de leurs données de type `spgConfigOut.leafType`. `level` est le niveau actuel que les lignes feuille concernées partagent, qui deviendra le niveau de la nouvelle ligne intermédiaire.

Définir `hasPrefix` pour indiquer que la nouvelle ligne intermédiaire doit avoir un préfixe, et dans ce cas, définir `prefixDatum` à la valeur de ce préfixe. Définir `nNodes` pour indiquer le nombre de nœuds que contiendra la nouvelle ligne intermédiaire, et spécifier dans `nodeLabels` un tableau de leurs labels, ou NULL si les labels ne sont pas nécessaires. Attribuer à `mapTuplesToNodes` un tableau des index (à partir de zéro) des nœuds auxquels seront assignés chaque ligne feuille. Attribuer à `leafTupleDatums` un tableau des valeurs à stocker dans la nouvelle ligne de feuilles (ces valeurs seront les mêmes que celles des données `datums` fournies en paramètre si la classe d'opérateurs ne modifie pas les données d'un niveau à un autre). À noter que la fonction `picksplit` est responsable de l'allocation de mémoire des tableaux `nodeLabels`, `mapTuplesToNodes` et `leafTupleDatums`.

Si plus d'une ligne de feuille est fournie, il est nécessaire que la fonction `picksplit` les classent en plus d'un nœud. Dans le cas contraire, il ne sera pas possible de répartir les lignes des feuilles sur des pages différentes, ce qui est pourtant l'objectif de cette opération. À cet effet, si la fonction `picksplit` se termine après avoir réparti toutes les lignes des feuilles dans le même nœud, le code du moteur de SP-GiST ne tiendra pas compte de cette décision, et générera une ligne intermédiaire dans lequel chaque ligne de feuille sera assigné aléatoirement à plusieurs nœuds de labels identiques. De telles lignes sont marquées `allTheSame` pour garder une trace de cette décision. Les fonctions `choose` et `inner_consistent` doivent tenir compte de ces lignes intermédiaires. Voir Section 69.4.3 pour plus d'informations.

`picksplit` peut être appliqué à une unique ligne de feuille lorsque la fonction `config` définit `longValuesOK` à true et qu'une valeur plus large qu'une page est donnée en paramètre. Dans ce cas, l'objectif de la fonction est d'extraire un préfixe et de produire une donnée de feuille moins longue. Cet appel sera répété jusqu'à ce que la donnée de la feuille soit suffisamment petite pour tenir dans une page. Voir Section 69.4.1 pour plus d'information.

`inner_consistent`

Retourne un ensemble de nœuds (branches) à suivre durant une recherche arborescente.

La déclaration SQL de cette fonction doit ressembler à :

```
CREATE FUNCTION ma_suite_de_nœuds(internal, internal) RETURNS
 void ...
```

Le premier argument est un pointeur vers une structure C `spgInnerConsistentIn`, qui contient les données en entrée de la fonction. Le second argument est un pointeur vers une structure C `spgInnerConsistentOut`, qui permet à la fonction d'y spécifier les données en sortie.

```
typedef struct spgInnerConsistentIn
{
```

```
    ScanKey     scankeys;       /* tableau d'opérateurs et de
 valeurs de comparaison */
    ScanKey     orderbys;       /* tableau d'opérateurs de tri
 et comparaison */
                                 * de valeur */
    int         nkeys;          /* taille du tableau scankeys */
    int         norderbys;      /* taille du tableau orderbys */

    Datum       reconstructedValue;     /* valeur reconstruite
 au niveau parent */
    MemoryContext traversalMemoryContext;   /* placer les
 nouvelles valeurs ici */
    int         level;          /* niveau actuel (à partir de
 zéro) */
    bool        returnData;     /* retourner la valeur
 originale ? */

    /* Données du tuple intermédiaire en cours */
    bool        allTheSame;     /* la ligne est-elle identifiée
 comme all-the-same ? */
    bool        hasPrefix;      /* la ligne a-t-elle un
 préfixe ? */
    Datum       prefixDatum;    /* dans ce cas, la valeur du
 préfixe */
    int         nNodes;         /* nombre de nœuds dans la ligne
 intermédiaire */
    Datum      *nodeLabels;     /* labels du nœud (NULL si pas
 de labels) */
    void      **traversalValues;        /* valeurs traverses
 spécifiques de la classe d'opérateurs */
    double    **distances;              /* distances associées
 */
} spgInnerConsistentIn;

typedef struct spgInnerConsistentOut
{
    int         nNodes;         /* nombre de nœuds enfants à
 visiter */
    int        *nodeNumbers;    /* leurs index dans le tableau
 de nœuds */
    int        *levelAdds;      /* l'incrément à apporter au
 niveau pour chaque enfant */
    Datum      *reconstructedValues;    /* valeurs reconstruites
 associées */
} spgInnerConsistentOut;
```

Le tableau `scankeys`, de longueur `nkeys`, décrit les conditions de recherche d'index. Ces conditions sont combinées avec un opérateur et seules les entrées d'index qui correspondent à toutes ces conditions sont conservées (à noter que `nkeys` = 0 implique que toutes les entrées d'index sont conservées). Généralement, la fonction `inner_consistent` ne tient compte que des champs `sk_strategy` et `sk_argument` de chaque entrée de tableau, qui fournissent respectivement l'opérateur indexé et la valeur de comparaison. En particulier, il n'est pas nécessaire de vérifier si `sk_flags` est NULL car le moteur de SP-GiST aura complété cette valeur. Le tableau `orderbys`, de longueur `norderbys`, décrit les opérateurs de tri (s'il y en a) de la même manière. `reconstructedValue` est la valeur reconstruite pour la ligne parent. La valeur est `(Datum) 0` au niveau le plus haut ou si la fonction `inner_consistent` ne fournit pas de valeur pour le niveau supérieur. `traversalValue` est un pointer vers toute

donnée traverse passée à l'appel précédent de `inner_consistent` sur l'enregistrement parent de l'index, ou NULL à la racine. `traversalMemoryContext` est le contexte mémoire de stockage des valeurs traverses en sortie (voir ci-dessous). `level` est le niveau actuel de la ligne intermédiaire, en commençant à 0 pour le niveau racine. `returnData` est `true` pour la valeur reconstruite pour cette requête. Ce n'est le cas que si la fonction `config` définit `canReturnData`. `allTheSame` est true si la ligne intermédiaire en cours est marquée « all-the-same ». Dans ce cas, tous les nœuds ont le même label (si un label est défini) et ainsi soit ils correspondent tous à la requête, soit aucun ne correspond (voir Section 69.4.3). `hasPrefix` est true si la ligne intermédiaire en cours contient un préfixe. Dans ce cas, `prefixDatum` est sa valeur. `nNodes` est le nombre de nœuds enfants de la ligne intermédiaire, et `nodeLabels` est un tableau de leurs labels, ou NULL si les nœuds n'ont pas de labels.

`nNodes` doit être défini comme le nombre de nœuds enfants qui doivent être visités durant la recherche, et `nodeNumbers` doit être défini comme le tableau de leurs index. Si la classe d'opérateurs effectue le suivi des niveaux, définir `levelAdds` comme un tableau des incréments à ajouter aux niveaux pour descendre vers chaque nœud à visiter (dans la plupart des cas, les incréments seront les mêmes pour chaque nœud, mais ce n'est pas systématique, et ainsi un tableau est employé). Si la reconstruction de la valeur est nécessaire, définir `reconstructedValues` en un tableau des valeurs reconstruites pour chaque nœud enfant à visiter. Sinon, laisser `reconstructedValues` à la valeur NULL. Si une recherche triée est exécutée, initialise `distances` à un tableau de valeurs de distance suivant le tableau `orderbys` (les nœuds avec les plus petites distances seront traitées en premier). Laisse NULL sinon. Les valeurs reconstruites sont supposées être de type `spgConfigOut.leafType`. (Néanmoins, comme le coeur du système ne fera rien avec elles sauf potentiellementles copier, il est suffisant qu'elles aient les mêmes propriétés `typlen` et `typbyval` que `leafType`.) S'il est souhaitable de passer les informations supplémentaires hors bande (« valeurs traverses ») pour diminuer les niveaux de l'arbre de recherche, initialiser `traversalValues` en un tableau des valeurs traverses appropriées, un pour chaque nœud enfants à visiter ; sinon laisser `traversalValues` à NULL. Notez que la fonction `inner_consistent` est responsable de l'allocation mémoire des tableaux `nodeNumbers`, `levelAdds`, `distances`, `reconstructedValues` et `traversalValues` dans le contexte mémoire actuel. Néanmoins, toute valeur traverse en sortie pointée par le tableau `traversalValues` devrait être allouée dans `traversalMemoryContext`. Chaque valeur traverse doit être un morceau simple alloué avec la fonction palloc.

`leaf_consistent`

Retourne true si une ligne de feuille satisfait une requête.

La déclaration SQL de cette fonction doit ressembler à :

```
CREATE FUNCTION ma_fonction_leaf_consistent(internal, internal)
 RETURNS bool ...
```

Le premier argument est un pointeur vers une structure C `spgLeafConsistentIn`, qui contient les données en entrée de la fonction. Le second argument est un pointeur vers une structure C `spgLeafConsistentOut`, qui permet à la fonction d'y spécifier les données en sortie.

```
typedef struct spgLeafConsistentIn
{
    ScanKey     scankeys;       /* tableau d'opérateurs et de
 valeurs de comparaison */
    ScanKey     orderbys;       /* tableau d'opérateurs de tri
 et comparaison */
                                 * de valeurs */
    int         nkeys;          /* taille du tableau scankeys */
```

```
    int         norderbys;      /* taille du tableau orderbys */

    Datum       reconstructedValue;     /* valeur reconstruite
 au parent */
    void       *traversalValue; /* valeur traverse spécifique à
 la classe d'opérateurs */
    int         level;          /* niveau actuel (à partir de
 zéro) */
    bool        returnData;     /* les données originales
 doivent-elles être reconstruites ? */

    Datum       leafDatum;      /* données de la ligne de
 feuille */
} spgLeafConsistentIn;

typedef struct spgLeafConsistentOut
{
    Datum       leafValue;      /* données originales
 reconstruites, le cas échéant */
    bool        recheck;        /* définir à true si l'opérateur
 doit être revérifié */
    Datum       leafValue;        /* valeur d'origine
 reconstruite, le cas échéant */
    bool        recheck;          /* positionné à true si
 l'opérateur doit être revérifié */
    bool        recheckDistances; /* positionné à true si les
 distances doivent être revérifiées */
    double     *distances;        /* associated distances */
} spgLeafConsistentOut;
```

Le tableau `scankeys`, de longueur `nkeys`, décrit les conditions de recherche dans l'index. Ces conditions sont uniquement combinées avec AND -- Seules les entrées d'index qui satisfont toutes les conditions satisfont la requête (Notez que `nkeys` = 0 implique que toutes les entrées de l'index satisfont la requête). Généralement, la fonction de recherche ne tient compte que des champs `sk_strategy` et `sk_argument` de chaque entrée du tableau, qui correspondent respectivement à l'opérateur indexable et à la valeur de comparaison. En particulier, il n'est pas nécessaire de vérifier `sk_flags` pour savoir que la valeur de comparaison est NULL car le code du cœur de SP-GiST filtre ces conditions. Le tableau `orderbys`, de taille `norderbys`, décrit les opérateurs de tri de la même manière. `reconstructedValue` est la valeur reconstruite pour la ligne parent ; Il s'agit de `(Datum) 0` au niveau racine ou si la fonction `inner_consistent` ne fournit pas de valeur au niveau parent. `traversalValue` est un pointeur vers toute donnée traverse passée lors de l'appel précédent à `inner_consistent` de l'enregistrement parent de l'index ou NULL à la racine. `level` est le niveau actuel de la ligne de feuille, qui commence à zéro pour le niveau racine. `returnData` est `true` s'il est nécessaire de reconstruire les données pour cette requête. Cela ne sera le cas que lorsque la fonction `config` vérifie `canReturnData`. `leafDatum` est la valeur de la clé stockée de `spgConfigOut.leafType` dans la ligne de feuille en cours.

La fonction doit retourner `true` si la ligne de feuille correspond à la requête ou `false` sinon. Dans le cas où la valeur serait `true`, et que `returnData` est `true` alors `leafValue` doit être défini à la valeur originale, de type `spgConfigIn.attType` fournie pour être indexée pour cette ligne de feuille. `recheck` peut être défini à `true` si la correspondance est incertaine et ainsi l'opérateur doit être réappliqué à la pile de ligne courante pour vérifier la correspondance. Si une recherche triée est effectuée, positionne `distances` à un tableau de distance suivant le tableau `orderbys` Laisse NULL sinon. Si au moins une des distances retournées n'est pas exacte, positionne `recheckDistances` à true. Dans ce cas, l'exécuteur recalculera la distance exacte après avoir récupéré toutes les lignes de la table, et réordonnera les lignes si besoin.

Les méthodes optionnelles définies par l'utilisateur sont :

`Datum compress(Datum in)`

Convertit un élément de données dans un format convenable pour un stockage physique dans un enregistrement feuille d'un index. Elle accepte une valeur de type `spgConfigIn.attType` et renvoie une valeur de type `spgConfigOut.leafType`. La valeur en sortie ne doit pas contenir un pointeur TOAST hors ligne.

Note : la méthode `compress` est seulement appliquée aux valeurs à enregistrer. Les méthodes cohérentes reçoivent les clés de parcours non modifiées, sans transformation utilisant `compress`.

`options`

Définit un ensemble de paramètres visibles aux utilisateurs qui contrôlent le comportement d'une classe d'opérateurs.

La déclaration SQL de la fonction doit ressembler à ceci :

```
CREATE OR REPLACE FUNCTION my_options(internal)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

La fonction `options` se voit donné un pointeur vers une structure `local_relopts` qui doit être remplie avec un ensemble d'options spécifiques à la classe d'opérateur. Les options peuvent être accédées à partir des autres fonctions de support en utilisant les macros `PG_HAS_OPCLASS_OPTIONS()` et `PG_GET_OPCLASS_OPTIONS()`.

Étant donné que la représentation de la clé dans SP-GiST est flexible, elle peut dépendre de paramètres spécifiés par l'utilisateur.

Toutes les méthodes permettant d'utiliser SP-GiST sont normalement exécutées dans un contexte mémoire de courte durée, c'est-à-dire que `CurrentMemoryContext` sera remis à zéro après le traitement de chaque ligne. Il n'est cependant pas réellement important de se soucier de désallouer la mémoire allouée avec palloc (la méthode `config` est une exception : elle essaiera d'éviter les fuites mémoire. Mais généralement, la méthode `config` ne nécessite rien si ce n'est assigner des constantes aux structures passées en paramètre).

Si la colonne indexée a un type de donnée collationnable, l'index de collationnement sera passé à toutes les méthodes, en utilisant le mécanisme standard `PG_GET_COLLATION()`.

69.4. Implémentation

Cette section traite des détails d'implémentation et d'autres astuces qui sont utiles à connaître pour implémenter des opérateurs de classe SP-GiST.

69.4.1. Limites de SP-GiST

Les lignes de feuille individuelles et les lignes intermédiaires doivent tenir dans une unique page d'index (8 ko par défaut). Cependant, lorsque des données de taille variable sont indexées, les longues valeurs ne sont uniquement supportées que par les arbres suffixés, dans lesquels chaque niveau de l'arbre contient un préfixe qui est suffisamment petit pour tenir dans une page. La classe d'opérateurs doit uniquement définir `longValuesOK` à TRUE si elle supporte ce cas de figure. Dans le cas contraire, le cœur de SP-GiST rejettera l'indexation d'une valeur plus large qu'une page.

De la même manière, il est de la responsabilité de l'opérateur de classe de s'assurer que la taille des lignes intermédiaires soit plus petite qu'une page ; cela limite le nombre de nœuds enfants qui peuvent être utilisés dans une ligne intermédiaire, ainsi que la taille maximum d'un préfixe.

Une autre limite est que lorsqu'un nœud de ligne intermédiaire pointe vers un ensemble de lignes de feuille, ces lignes doivent toutes être dans la même page d'index (il s'agit d'une décision d'architecture pour réduire le temps de recherche et utiliser moins de mémoire dans les liens qui lient de telles lignes ensemble). Si l'ensemble de lignes de feuille grandit plus qu'une page, un découpage est réalisé et un nœud intermédiaire est inséré. Pour que ce mécanisme résolve le problème, le nouveau nœud intermédiaire *doit* diviser l'ensemble de valeurs de feuilles en plus d'un groupe de nœuds. Si la fonction `picksplit` de la classe d'opérateurs n'y parvient pas, le cœur de SP-GiST met en œuvre des mesures extraordinaires telles que décrites dans Section 69.4.3.

Quand `longValuesOK` vaut true, il est attendu que les niveaux successifs de l'arbre SP-GiST absorbera de plus en plus d'informations dans les prefixes et labels de noeuds des enregistrements internes, rendant la donnée feuille de plus en plus petite, pour qu'à la fin, elle tienne sur un bloc. Pour empêcher les bugs dans les classes d'opérateurs, du style boucle d'insertions infinies, le code principal de SP-GiST lèvera une erreur si la donnée feuille ne devient pas plus petite au bout de dix cycles d'appel à la méthode `choose`.

Quand `longValuesOK` vaut true, il est attendu que les niveaux successifs de l'arbre SP-GiST absorberont de plus en plus d'informations dans les préfixes et labels de nœuds des lignes internes, rendant la donnée requise pour la feuille de plus en plus petite, jusqu'à ce qu'elle tienne sur un bloc. Pour empêcher que des bugs dans les classes d'opérateurs causent des boucles d'insertion infinies, le noyau de SP-GiST lèvera une erreur si la donnée de la feuille ne devient pas plus petite dans les dix cycles d'appel à la méthode `choose`.

69.4.2. SP-GiST sans label de nœud

Certains algorithmes d'arbres utilisent un ensemble de nœuds figé pour chaque ligne intermédiaire ; par exemple, l'arbre quad-tree impose exactement quatre nœuds correspondant aux quatre coins autour du centroïde de la ligne intermédiaire. Dans ce cas, le code travaille généralement avec les nœuds au moyen de leur identifiant, et le besoin de label de nœud ne se fait pas ressentir. Pour supprimer les labels de nœud (et ainsi gagner de l'espace), la fonction `picksplit` peut retourner NULL pour le tableau `nodeLabels`, et de même, la fonction `choose` peut retourner NULL pour le tableau `prefixNodeLabels` lors de l'action `spgSplitTuple` Cela aura pour effet d'obtenir une valeur NULL pour `nodeLabels` lors des appels aux fonctions `choose` et `inner_consistent`. En principe, les labels de nœuds peuvent être utilisés par certaines lignes intermédiaires, et ignorés pour les autres de même index.

Lorsqu'une ligne intermédaire sans label est concerné, la fonction `choose` ne peut pas retourner `spgAddNode` car l'ensemble des nœuds est supposé être fixé dans de tels cas.

69.4.3. Lignes intermédiaires « All-the-same »

Le cœur de SP-GiST peut surcharger les résultats de la fonction `picksplit` de l'opérateur de classe lorsque `picksplit` ne réussit pas à diviser la valeur de la feuille fournie en au moins un nœud. Dans ce cas, la nouvelle ligne intermédiaire est créée avec de multiples nœuds qui ont tous le même label (si un label est défini) qui est celui attribué au nœud utilisé par `picksplit` et les valeurs des feuilles sont divisées aléatoirement entre les nœuds équivalents. Le drapeau `allTheSame` est activé sur la ligne intermédiaire pour signifier aux fonctions `choose` et `inner_consistent` que la ligne n'a pas l'ensemble de nœud attendu.

Lorsque le cas d'une ligne `allTheSame` est rencontré, le résultat de la fonction `choose` sous la forme `spgMatchNode` est interprété de manière à ce que la nouvelle valeur puisse être assignée à chacun des nœuds équivalents ; le code du cœur de SP-GiST ignorera la valeur `nodeN` fournie et descendra dans l'un des nœuds enfants au hasard (pour conserver l'équilibre de l'arbre). Il s'agirait d'une erreur si la fonction `choose` retournait `spgAddNode` car tous les nœuds ne seraient pas

équivalent ; l'action `spgSplitTuple` doit être utilisée si la valeur à insérer ne correspond pas aux nœuds existants.

Lorsque le cas d'une ligne `allTheSame` est rencontré, la fonction `inner_consistent` peut tout autant retourner tous les nœuds ou aucun des nœuds ciblés pour continuer la recherche indexée car ils sont tous équivalents. Cela peut éventuellement nécessiter du code spécifique, suivant le support réalisé par la fonction `inner_consistent` concernant la signification des nœuds.

69.5. Exemples

Les sources de PostgreSQL incluent plusieurs exemples de classes d'opérateurs d'index pour SP-GiST comme décrit dans Tableau 69.1. Lire le code dans `src/backend/access/spgist/` et `src/backend/utils/adt/`.

Chapitre 70. Index GIN

70.1. Introduction

GIN est l'acronyme de *Generalized Inverted Index* (ou index générique inversé). GIN est prévu pour traiter les cas où les éléments à indexer sont des valeurs composites, et où les requêtes devant utiliser l'index doivent rechercher des valeurs d'éléments apparaissant dans ces éléments composites. Par exemple, les éléments pourraient être des documents, et les requêtes pourraient être des recherches de documents contenant des mots spécifiques.

Nous utilisons le mot *élément* (*item* en version originale) pour désigner une valeur composite qui doit être indexée, et le mot *clé* (*clé* en version originale) pour désigner une des valeurs d'un élément. GIN stocke et recherche toujours des clés, jamais des éléments eux-mêmes.

Un index GIN stocke un jeu de paires de (clé, liste), où *liste* (*posting list* en version originale) est un ensemble d'identifiants de ligne (*row ID* en version originale) où la clé existe. Le même identifiant peut apparaître dans plusieurs listes, puisqu'un élément peut contenir plus d'une clé. Chaque clé est stockée une seule fois, ce qui fait qu'un index GIN est très compact dans le cas où une clé apparaît de nombreuses fois.

GIN est générique dans le sens où la méthode d'accès GIN n'a pas besoin de connaître les opérations spécifiques qu'elle accélère. À la place, elle utilise les stratégies spécifiques définies pour des types de données. La stratégie définit comment extraire les clés des éléments à indexer et des conditions des requêtes, et comment déterminer si une ligne qui contient des valeurs de clés d'une requête répond réellement à la requête.

Un des avantages de GIN est la possibilité qu'il offre que des types de données personnalisés avec des méthodes d'accès appropriées soient développés par un expert du domaine du type de données, plutôt que par un expert en bases de données. La méthode GiST offre le même avantage.

L'implantation de GIN dans PostgreSQL est principalement l'œuvre de Teodor Sigaev et Oleg Bartunov. Plus d'informations sur GIN sont disponibles sur leur site web[1].

70.2. Classes d'opérateur natives

La distribution PostgreSQL inclut les classes d'opérateur GIN affichées dans Tableau 70.1. (Certains des modules optionnels décrits dans Annexe F fournissent des classes d'opérateurs GIN supplémentaires.)

Tableau 70.1. Classes d'opérateur GIN natives

Nom	Opérateurs indexables
array_ops	&& (anyarray,anyarray)
	@> (anyarray,anyarray)
	<@ (anyarray,anyarray)
	= (anyarray,anyarray)
jsonb_ops	@> (jsonb,jsonb)
	@? (jsonb,jsonpath)
	@@ (jsonb,jsonpath)
	? (jsonb,text)
	?| (jsonb,text[])
	?& (jsonb,text[])

[1] http://www.sai.msu.su/~megera/wiki/Gin

Nom	Opérateurs indexables
jsonb_path_ops	@> (jsonb,jsonb)
	@? (jsonb,jsonpath)
	@@ (jsonb,jsonpath)
tsvector_ops	@@ (tsvector,tsquery)
	@@@ (tsvector,tsquery)

Des deux classes d'opérateurs pour le type jsonb, jsonb_ops est l'opérateur par défaut. jsonb_path_ops supporte moins d'opérateurs mais offre de meilleures performances pour ces opérateurs. Voir Section 8.14.4 pour plus de détails.

70.3. Extensibilité

L'interface GIN a un haut niveau d'abstraction. De ce fait, la personne qui code la méthode d'accès n'a besoin d'implanter que les sémantiques du type de données accédé. La couche GIN prend en charge la gestion de la concurrence, des traces et des recherches dans la structure de l'arbre.

Pour développer une méthode d'accès GIN fonctionnelle, il suffit d'implanter quelques méthodes utilisateur. Celles-ci définissent le comportement des clés dans l'arbre et les relations entre clés, valeurs d'éléments indexées et requêtes indexables. En résumé, GIN combine extensibilité, généricité, réutilisation du code, et une interface claire.

Voici les deux méthodes qu'une classe d'opérateurs GIN doit fournir sont :

```
Datum *extractValue(Datum inputValue, int32 *nkeys, bool **nullFlags)
```

Retourne un tableau de clés alloué par palloc en fonction d'un élément à indexer. Le nombre de clés renvoyées doit être stocké dans *nkeys. Si une des clés peut être nulle, allouez aussi par palloc un tableau de *nkeys champs de type bool, stockez son adresse dans *nullFlags, et positionnez les drapeaux null où ils doivent l'être. *nullFlags peut être laissé à NULL (sa valeur initiale) si aucune clé n'est nulle. La valeur retournée peut être NULL si l'élément ne contient aucune clé.

```
Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool
**pmatch, Pointer **extra_data, bool **nullFlags, int32 *searchMode)
```

Renvoie un tableau de clés en fonction de la valeur à rechercher ; c'est-à-dire que query est la valeur du côté droit d'un opérateur indexable dont le côté gauche est la colonne indexée. n est le numéro de stratégie de l'opérateur dans la classe d'opérateurs (voir Section 38.16.2). Souvent, extractQuery doit consulter n pour déterminer le type de données de query et la méthode à utiliser pour extraire les valeurs des clés. Le nombre de clés renvoyées doit être stocké dans *nkeys. Si une des clés peut être nulle, allouez aussi par palloc un tableau de *nkeys champs de type bool, stockez son adresse dans *nullFlags, et positionnez les drapeaux NULL où ils doivent l'être. *nullFlags peut être laissé à NULL (sa valeur initiale) si aucune clé n'est nulle. La valeur de retour peut être NULL si query ne contient aucune clé.

searchMode est un argument de sortie qui permet à extractQuery de spécifier des détails sur comment la recherche sera effectuée. Si *searchMode est positionné à GIN_SEARCH_MODE_DEFAULT (qui est la valeur à laquelle il est initialisé avant l'appel), seuls les éléments qui correspondent à au moins une des clés retournées sont considérés comme des candidats de correspondance. Si *searchMode est positionné à GIN_SEARCH_MODE_INCLUDE_EMPTY, alors en plus des éléments qui contiennent au moins une clé correspondante, les éléments qui ne contiennent aucune clé sont aussi considérées comme des candidats de correspondance. (Ce mode est utile pour implémenter un opérateur « est le sous-ensemble de », par exemple.) Si *searchMode est positionné à GIN_SEARCH_MODE_ALL, alors tous les éléments non nuls de l'index sont candidats de correspondance, qu'ils aient une clé qui corresponde à celles retournées ou non. (Ce mode est beaucoup plus lent que les deux autres,

mais il peut être nécessaire pour implanter des cas exceptionnels correctement. Un opérateur qui a besoin de ce mode dans la plupart des cas n'est probablement pas un bon candidat pour une classe d'opérateurs GIN.) Les symboles à utiliser pour configurer ce mode sont définis dans `access/gin.h`.

`pmatch` est un argument de sortie à utiliser quand une correspondance partielle est permise. Pour l'utiliser, `extractQuery` doit allouer un tableau de booléens `*nkeys` et stocker son adresse dans `*pmatch`. Chaque élément du tableau devrait être positionné à true si la clé correspondante nécessite une correspondance partielle, et à false sinon. Si `*pmatch` est configuré à `NULL`, alors GIN suppose qu'une correspondance partielle n'est pas nécessaire. La variable est initialisée à `NULL` avant l'appel, et peut donc être simplement ignorée par les classes d'opérateurs qui ne gèrent pas les correspondances partielles.

`extra_data` est un argument de sortie qui autorise `extractQuery` à passer des données supplémentaires aux méthodes `consistent` et `comparePartial`. Pour l'utiliser, `extractQuery` doit allouer un tableau de pointeurs `*nkeys` et stocker son adresse dans `*extra_data`, puis stocker ce qu'il souhaite dans les pointeurs individuels. La variable est initialisée à `NULL` avant l'appel, afin que ce paramètre soit simplement ignoré par une classe d'opérateurs qui n'a pas besoin de données supplémentaires. Si `*extra_data` est positionné, le tableau dans son ensemble est passé à la méthode `consistent`, et l'élément approprié à la méthode `comparePartial`.

Une classe d'opérateurs doit aussi fournir une fonction pour vérifier si un élément indexé correspond à la requête. Elle vient en deux versions, une fonction booléenne `consistent` et une fonction ternaire `triConsistent`. Cette dernière couvre les fonctionnalités des deux, donc fournir uniquement `triConsistent` est suffisant. Cependant, si la variante booléenne est bien moins coûteuse à calculer, il peut être avantageux de fournir les deux. Si seule la variante booléenne est fournie, certaines optimisations dépendant de la réfutation d'éléments d'index avant de récupérer toutes les clés sont désactivées.

```
bool consistent(bool check[], StrategyNumber n, Datum query, int32
nkeys, Pointer extra_data[], bool *recheck, Datum queryKeys[], bool
nullFlags[])
```

Retourne true si un élément indexé répond à l'opérateur de requête possédant le numéro de stratégie `n` (ou pourrait le satisfaire si l'indication recheck est retournée). Cette fonction n'a pas d'accès direct aux valeurs des éléments indexés. Au lieu de cela, ce qui est disponible, c'est la connaissance de quelles valeurs de clés extraites de la requête apparaissent dans un élément indexé donné. Le tableau `check` a une longueur de `nkeys`, qui est identique au nombre de clés retourné précédemment par `extractQuery` pour cette donnée `query`. Chaque élément du tableau `check` est true si l'élément indexé contient la clé de requête correspondante, c'est à dire, si (check[i] == true), la i-ème clé du tableau résultat de `extractQuery` est présente dans l'élément indexé. La donnée `query` originale est passée au cas où la méthode `consistent` aurait besoin de le consulter, de même que les tableaux `queryKeys[]` et `nullFlags []` retournés précédemment par `extractQuery`. `extra_data` est le tableau de données supplémentaires renvoyé par `extractQuery`, ou `NULL` si aucun.

Quand `extractQuery` renvoie une clé nulle dans `queryKeys[]`, l'élément correspondant de `check[]` est true si l'élément indexé contient une clé nulle ; c'est-à-dire que la sémantique de `check []` est comme celle de `IS NOT DISTINCT FROM`. La fonction `consistent` peut examiner l'élément correspondant de `nullFlags[]` si elle a besoin de faire la différence entre une correspondance de valeur « normale » et une correspondance nulle.

En cas de réussite, `*recheck` devrait être positionné à true si les enregistrements de la table doivent être revérifiées par rapport à l'opérateur de la requête, ou à false si le test d'index est exact. Autrement dit, une valeur de retour à false garantit que l'enregistrement de la table ne correspond pas ; une valeur de retour à true avec `*recheck` à false garantit que l'enregistrement de la table correspond à la requête ; et une valeur de retour à true avec `*recheck` à true signifie que l'enregistrement de la table pourrait correspondre à la requête, et qu'il doit être récupéré et re-vérifié en évaluant l'opérateur de la requête directement sur l'élément initialement indexé.

```
GinTernaryValue triConsistent(GinTernaryValue check[],
StrategyNumber n, Datum query, int32 nkeys, Pointer extra_data[],
Datum queryKeys[], bool nullFlags[])
```

`triConsistent` est similaire à `consistent`, mais en lieu de booléens dans le vecteur `check`, il y a trois valeurs possibles pour chaque clé : `GIN_TRUE`, `GIN_FALSE` et `GIN_MAYBE`. `GIN_FALSE` et `GIN_TRUE` ont la même signification que des valeurs booléennes standards alors que `GIN_MAYBE` signifie que la présence de cette clé est inconnue. Quand des valeurs `GIN_MAYBE` sont présentes, la fonction devrait seulement renvoyer `GIN_TRUE` si l'élément correspond à coup sûr que l'élément de l'index contient ou non les clés de la requête correspondante. De la même façon, la fonction doit renvoyer `GIN_FALSE` seulement si l'élément ne correspond pas, qu'il contienne ou non des clés `GIN_MAYBE`. Si le résultat dépend des entrées `GIN_MAYBE`, autrement dit si la correspondance ne peut pas être confirmée ou réfutée en se basant sur les clés connues de la requête, la fonction doit renvoyer `GIN_MAYBE`.

Quand il n'y a pas de valeurs `GIN_MAYBE` dans le vecteur `check`, la valeur de retour `GIN_MAYBE` est équivalent à configurer l'indicateur `recheck` dans la fonction booléenne `consistent`.

De plus, GIN doit avoir un moyen de trier les valeurs des clés stockées dans l'index. La classe d'opérateurs peut définir l'ordre de tri en spécifiant une méthode de comparaison :

```
int compare(Datum a, Datum b)
```

Compare deux clés (pas des éléments indexés) et renvoie un entier inférieur à zéro, égal à zéro ou supérieur à zéro, indiquant si la première clé est inférieure à, égale à ou supérieure à la seconde. Les clés nulles ne sont jamais fournies en argument à cette fonction.

Sinon, si la classe d'opérateurs ne fournit pas de méthode `compare`, GIN cherchera la classe d'opérateurs B-Tree par défaut pour le type de donnée de la clé d'index, et utilisera sa fonction de comparaison. Il est recommandé de spécifier la fonction de comparaison dans une classe d'opérateurs GIN destinée à un seul type de donnée, car rechercher la classe d'opérateurs B-Tree coûte quelques cycles CPU. Cependant, les classes d'opérateurs GIN polymorphiques (telle que `array_ops`) ne peuvent typiquement pas spécifier une seule fonction de comparaison.

Une classe d'opérateurs pour GIN peut fournir en option les méthodes suivantes :

```
int comparePartial(Datum partial_key, Datum key, StrategyNumber n,
Pointer extra_data)
```

Compare une clé d'une requête de correspondance partielle à une clé d'index. Renvoie un entier dont le signe indique le résultat : inférieur à zéro signifie que la clé d'index ne correspond pas à la requête mais que le parcours d'index va continuer ; zéro signifie que la clé d'index ne correspond pas à la requête ; supérieur à zéro indique que le parcours d'index doit s'arrêter car il n'existe pas d'autres correspondances. Le numéro de stratégie `n` de l'opérateur qui a engendré la requête de correspondance partielle est fourni au cas où sa sémantique serait nécessaire pour déterminer la fin du parcours. De plus, `extra_data` est l'élément correspondant du tableau renvoyé par `extractQuery` (`NULL` sinon). Les clés nulles ne sont jamais passées à cette fonction.

```
void options(local_relopts *relopts)
```

Définit un ensemble de paramètres visibles aux utilisateurs qui contrôle le comportement d'une classe d'opérateurs.

La fonction `options` se voit donner un pointeur vers une structure `local_relopts` qui doit être remplie avec un ensemble d'options spécifiques à la classe d'opérateurs. Les options peuvent être accédées à partir des autres fonctions de support en utilisant les macros `PG_HAS_OPCLASS_OPTIONS()` et `PG_GET_OPCLASS_OPTIONS()`.

Étant donné que l'extraction des clés des valeurs indexées et la représentation de la clé dans GIN sont flexibles, elles peuvent dépendre de paramètres spécifiés par l'utilisateur.

Pour supporter des requêtes à « correspondance partielle », une classe d'opérateurs doit fournir la méthode `comparePartial`, et sa méthode `extractQuery` doit positionner le paramètre `pmatch` quand une requête à correspondance partielle est rencontrée. Voir Section 70.4.2 pour les détails.

Le type de données réel des différentes valeurs `Datum` mentionnées ci-dessus varient en fonction de la classe d'opérateurs. Les valeurs d'élément passées à `extractValue` sont toujours du type en entrée de la classe d'opérateurs, et toutes les valeurs de clé doivent être du type `STORAGE` de la classe. Le type de l'argument `query` passé aux fonctions `extractQuery`, `consistent` et `triConsistent` est le type de l'argument côté droit de l'opérateur du membre de la classe identifié par le numéro de stratégie. Ce n'est pas nécessairement le même que l'élément indexé, tant que des valeurs de clés d'un type correct peuvent en être extraites. Néanmoins, il est recommandé que les déclarations SQL de ces trois fonctions de support utilisent le type de données indexé de la classe d'opérateurs pour l'argument `query`, même si le type réel pourrait être différent suivant l'opérateur.

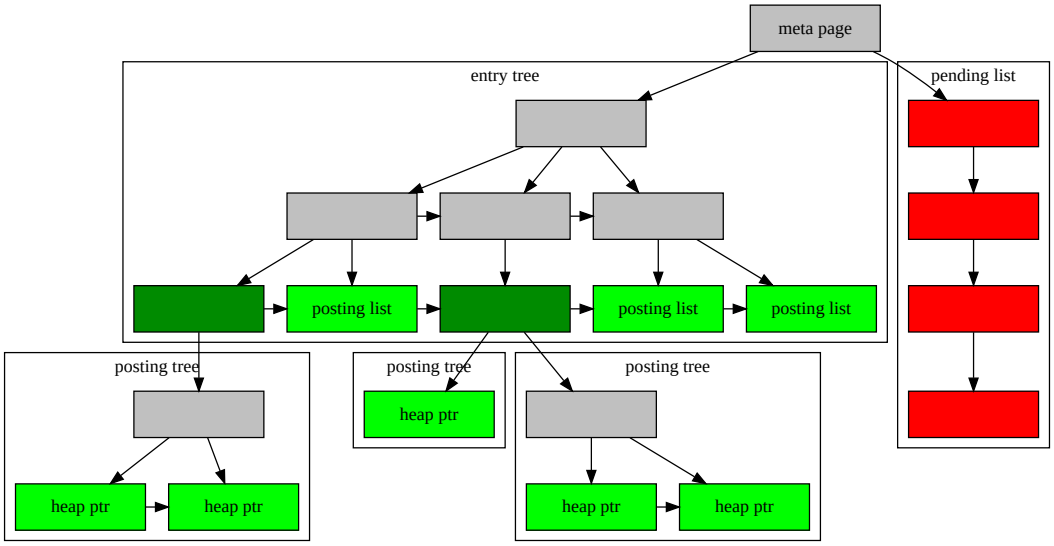
70.4. Implantation

En interne, un index GIN contient un index B-tree construit sur des clés, où chaque clé est une partie d'un ou plusieurs éléments indexé (un membre d'un tableau, par exemple) et où chaque enregistrement d'une page feuille contient soit un pointeur vers un B-tree de pointeurs vers la table (un « posting tree »), ou une liste simple de pointeurs vers les enregistrements (une « posting list ») quand la liste est suffisamment courte pour tenir dans un seul enregistrement d'index avec la valeur de la clé. Figure 70.1 illustre ces composants d'un index GIN.

À partir de PostgreSQL 9.1, des valeurs NULL de clé peuvent être incluses dans l'index. Par ailleurs, des NULLs fictifs sont inclus dans l'index pour des éléments indexés qui sont NULL ou ne contiennent aucune clé d'après `extractValue`. Cela permet des recherches retournant des éléments vides.

Les index multi-colonnes GIN sont implantés en construisant un seul B-tree sur des valeurs composites (numéro de colonne, valeur de clé). Les valeurs de clés pour les différentes colonnes peuvent être de types différents.

Figure 70.1. Cœur de GIN



70.4.1. Technique GIN de mise à jour rapide

Mettre à jour un index GIN a tendance à être lent en raison de la nature intrinsèque des index inversés : insérer ou mettre à jour un ligne de la table peut causer de nombreuses insertions dans l'index(une pour chaque clé extraite de l'élément indexé). GIN est capable de reporter à plus tard

la plupart de ce travail en insérant les nouveaux enregistrements dans une liste temporaire et non triée des entrées en attente. Quand un vacuum ou autoanalyze est déclenché sur la table, ou quand la fonction `gin_clean_pending_list` est appelée, ou si la liste en attente devient plus importante que gin_pending_list_limit, les entrées sont déplacées vers la structure de données GIN principale en utilisant la même technique d'insertion de masse que durant la création de l'index. Ceci améliore grandement la vitesse de mise à jour de l'index GIN, même en prenant en compte le surcoût engendré au niveau du vacuum. De plus, ce travail supplémentaire peut être attribué à un processus d'arrière-plan plutôt qu'à la requête en avant-plan.

Le principal défaut de cette approche est que les recherches doivent parcourir la liste d'entrées en attente en plus de l'index habituel, et que, par conséquent, une grande liste d'entrées en attente ralentira les recherches de manière significative. Un autre défaut est que, bien que la majorité des mises à jour sont rapides, une mise à jour qui rend la liste d'attente « trop grande » déclenchera un cycle de nettoyage immédiat et sera donc bien plus lente que les autres mises à jour. Une utilisation appropriée d'autovacuum peut minimiser ces deux problèmes.

Si la cohérence des temps de réponse est plus importante que la vitesse de mise à jour, l'utilisation de liste d'entrées en attente peut être désactivée en désactivant le paramètre de stockage `fastupdate` pour un index GIN. Voir CREATE INDEX pour plus de détails.

70.4.2. Algorithme de mise en correspondance partielle

GIN peut accepter des requêtes de « correspondances partielles », dans lesquelles la requête ne détermine pas une correspondance parfaite pour une ou plusieurs clés, mais que la correspondance tombe à une distance suffisamment proche des valeurs de clé (dans l'ordre de tri des clés déterminé par la méthode d'appui `compare`). Au lieu de retourner une valeur de clé à mettre en correspondance de façon exacte, la méthode `extractQuery` retourne une valeur de clé qui est la limite inférieure de la plage à rechercher, et retourne l'indicateur `pmatch` positionné à true. La plage de clé est alors parcourue en utilisant la méthode `comparePartial`. `comparePartial` doit renvoyer 0 pour une clé d'index correspondante, une valeur négative pour une non-correspondance qui est toujours dans la plage de recherche, et une valeur positive si la clé d'index est en dehors de la plage de correspondance.

70.5. Conseils et astuces sur GIN

Création vs insertion

L'insertion dans un index GIN peut être lente du fait de la probabilité d'insertion de nombreuses clés pour chaque élément. C'est pourquoi, pour les chargements massifs dans une table, il est conseillé de supprimer l'index GIN et de le re-créer après le chargement.

Quand `fastupdate` est activé pour GIN(voir Section 70.4.1 pour les détails), la pénalité est moindre que quand il n'est pas activé. Mais pour les très grosses mises à jour, il peut toujours être plus efficace de détruire et recréer l'index.

maintenance_work_mem

Le temps de construction d'un index GIN dépend grandement du paramètre `maintenance_work_mem` ; il est contre-productif de limiter la mémoire de travail lors de la création d'un index.

gin_pending_list_limit

Durant une série d'insertions dans un index GIN existant pour lequel l'option `fastupdate` est activé, le système nettoiera la liste d'entrées en attente dès qu'elle deviendra plus grosse que la limite indiquée par `gin_pending_list_limit`. Afin d'éviter des fluctuations mesurables de temps de réponse, il est souhaitable d'avoir un nettoyage de la liste d'attente en arrière-plan (c'est-à-dire via autovacuum). Les opérations de nettoyage en avant-plan peuvent être évitées

en augmentant `gin_pending_list_limit` ou en rendant le processus autovacuum plus aggressif. Toutefois, augmenter la limite de l'opération de nettoyage implique que si un nettoyage en avant-plan se produit, il prendra encore plus de temps.

`gin_pending_list_limit` peut être surchargé sur certains index en modifiant les paramètres de stockage, ce qui permet à chaque index d'avoir sa propre limite de nettoyage. Par exemple, il est possible d'augmenter la limite uniquement pour un index GIN fortement mis à jour ou de la diminuer dans le cas contraire.

gin_fuzzy_search_limit

La raison principale qui a poussé le développement des index GIN a été la volonté d'ajouter les recherches plein texte dans PostgreSQL et il arrive fréquemment qu'une recherche de ce type renvoie un ensemble volumineux de résultats. Cela arrive d'autant plus fréquemment que la requête contient des mots très fréquents, auquel cas l'ensemble de résultats n'est même pas utile. Puisque la lecture des lignes sur disque et leur tri prend beaucoup de temps, cette situation est inacceptable en production. (La recherche dans l'index est, elle, très rapide.)

Pour faciliter l'exécution contrôlée de telles requêtes, GIN dispose d'une limite supérieure souple configurable du nombre de lignes renvoyées, le paramètre de configuration `gin_fuzzy_search_limit`. Par défaut, il est positionné à 0 (c'est-à-dire sans limite). Si une limite différente de 0 est choisie, alors l'ensemble renvoyé est un sous-ensemble du résultat complet, choisi aléatoirement.

« Souple » signifie que le nombre réel de résultats renvoyés peut différer légèrement de la limite indiquée, en fonction de la requête et de la qualité du générateur de nombres aléatoires du système.

D'expérience, des valeurs de l'ordre de quelques milliers (5000 -- 20000) fonctionnent bien.

70.6. Limitations

GIN part de l'hypothèse que les opérateurs indexables sont stricts. Cela signifie que la fonction `extractValue` ne sera pas appelée du tout sur une valeur d'élément nul (à la place, une entrée d'index factice sera créée automatiquement), et que la fonction `extractQuery` ne sera pas appelée non plus pour une valeur de requête nulle (à la place, la requête est considérée comme impossible à satisfaire). Notez toutefois que des valeurs de clé nulles contenues dans un élément composite ou une valeur de requête non nul sont supportées.

70.7. Exemples

Le noyau de la distribution PostgreSQL inclut la classe d'opérateurs GIN précédemment montrée dans Tableau 70.1. Les modules `contrib` suivants contiennent aussi des classes d'opérateurs GIN :

`btree-gin`

Fonctionnalité équivalente à B-Tree pour plusieurs types de données

`hstore`

Module pour le stockage des paires (clé, valeur)

intarray

Support amélioré pour le type `int[]`

`pg_trgm`

Similarité de texte par correspondance de trigramme

Chapitre 71. Index BRIN

71.1. Introduction

BRIN signifie Block Range Index, soit index par intervalles de bloc. BRIN est conçu pour gérer de grosses tables dont certaines ont des colonnes ayant une corrélation naturelle avec leur stockage physique.

BRIN fonctionne en terme d'*intervalle de blocs* (ou « intervalle de pages »). Un *intervalle de bloc* est un groupe de blocs physiquement adjacents dans la table ; Pour chaque gamme de bloc, un résumé des informations est stocké par l'index. Un exemple courant est une table avec une colonne date, contenant les références des ventes d'un magasin. Chaque commande y serait enregistrée chronologiquement. Dans la plupart des cas, les données seront donc insérées dans le même ordre où elles apparaîtront par la suite. De la même manière, une table, avec une colonne code postal, pourrait avoir tous les codes d'une même ville rassemblés naturellement au même endroit.

Les index BRIN peuvent répondre à des requêtes via un parcours d'index bitmap classique, et retourneront toutes les lignes de toutes les pages dans chaque intervalle si le résumé des informations contenues dans l'index est cohérent avec les conditions de la requête. L'exécuteur de la requête doit revérifier ces lignes et annuler celles qui ne répondent pas aux conditions initiales de la requête. En d'autres termes, on parle d'index à perte (*lossy*). Comme l'index BRIN est un petit index, parcourir cet index ajoute une légère surcharge par rapport à un parcours séquentiel mais permet d'éviter de parcourir des grandes parties de la table où on sait qu'on ne trouvera pas de lignes à remonter.

Les données spécifiques qu'un index BRIN va stocker, de même que les requêtes spécifiques auquel l'index va pouvoir répondre dépendent de la classe d'opérateurs choisie pour chaque colonne de l'index. Les types de données possédant un ordre de tri linéaire peuvent utiliser une classe d'opérateurs qui ne conserve que la valeur minimale et la valeur maximale dans chaque intervalle de bloc. Par exemple, un type géométrique peut stocker une *bounding box* pour tous les objets de l'intervalle de bloc.

La taille de l'intervalle de bloc est déterminée à la création de l'index par le paramètre `pages_per_range` Le nombre des entrées de l'index sera égal à la taille de la relation en page, divisée par la valeur sélectionnée dans `pages_per_range`. De ce fait, plus ce nombre est bas, plus l'index sera volumineux (il y a plus d'entrées d'index à stocker) mais, en même temps, le résumé des informations stockées pourra être plus précis, et un nombre plus important de blocs de données pourront être ignorés pendant le parcours d'index.

71.1.1. Maintenance de l'index

Lors de la création de l'index, toutes les pages de la table sont parcourues et un résumé des lignes de l'index est créé pour chaque intervalle, incluant certainement aussi un intervalle incomplet à la fin. Lors de l'ajout de nouvelles données dans des pages déjà incluses dans des résumés, cela va entraîner la mise à jour du résumé, avec les informations sur les nouvelles lignes insérées. Lorsqu'une nouvelle page est créée et qu'elle ne correspond à aucun des derniers intervalles résumés, l'intervalle auquel appartient la nouvelle page n'acquiert pas automatiquement un résumé ; ces lignes restent sans résumé jusqu'à ce qu'une exécution de résumé soit demandée plus tard, créant ainsi le résumé initial pour cet intervalle.

Il existe plusieurs façons de déclencher le résumé initial d'un intervalle de blocs. Si la table est traitée par VACUUM, soit manuellement soit par l'autovacuum, tous les intervalles de blocs existants non résumés sont résumés. De plus, si le paramètre autosummarize de l'index est activé, ce qui n'est pas le cas par défaut, à chaque fois que l'autovacuum est exécuté dans cette base, le résumé surviendra pour tous les intervalles de blocs non résumés qui ont été remplis, que la table elle-même soit traitée par l'autovacuum ou pas ; voir ci-dessous.

Dernièrement, les fonctions suivantes peuvent être utilisées :

`brin_summarize_new_values(regclass)` qui résume tous les blocs sans résumé ;

`brin_summarize_range(regclass, bigint)` qui résume seulement l'intervalle contenant le bloc donné s'il n'est pas résumé.

Quand le résumé automatique est activé, une requête est envoyée à l'`autovacuum` pour exécuter un résumé cible pour un intervalle de bloc quand une insertion est détecté pour le premier élément du premier bloc du prochain intervalle de bloc, opération à réaliser la prochaine fois qu'un autovacuum worker finit son exécution dans la même base. Si la queue des demandes est remplie, la demande n'est pas enregistrée et un message est enregistré dans les traces du serveur :

```
LOG:  request for BRIN range summarization for index "brin_wi_idx"
page 128 was not recorded
```

Quand cela arrive, l'intervalle restera non résumé jusqu'à la prochaine exécution d'un vacuum standard sur la table ou jusqu'à ce qu'une des fonctions mentionnées ci-dessus soit appelée.

Inversement, un intervalle peut se voir supprimer son résumé avec la fonction `brin_desummarize_range(regclass, bigint)`, qui est utile quand l'enregistrement de l'index n'est plus une très bonne représentation car les valeurs existantes ont changées. Voir Section 9.27.8 pour les détails.

71.2. Classes d'opérateur intégrées

La distribution du noyau PostgreSQL inclut la classe d'opérateur BRIN montrée dans Tableau 71.1.

La classe d'opérateur *minmax* stocke les valeurs minimale et maximale apparaissant dans l'intervalle de la colonne indexée. L'opérateur de classe *inclusion* stocke une valeur qui est incluse dans les valeurs contenues dans l'intervalle de la colonne indexée. Les classes d'opérateur *bloom* génèrent un filtre Bloom pour toutes les valeurs dans l'intervalle. Les classes d'opérateur *minmax-multi* conservent les multiples valeurs minimale et maximale, représentant les valeurs apparaissant dans la colonne indexée dans l'intervalle.

Tableau 71.1. Classe d'opérateur BRIN intégrée

Nom	Opérateurs indexables
bit_minmax_ops	= (bit,bit)
	< (bit,bit)
	> (bit,bit)
	<= (bit,bit)
	>= (bit,bit)
box_inclusion_ops	@> (box,point)
	<< (box,box)
	&< (box,box)
	&> (box,box)
	>> (box,box)
	<@ (box,box)
	@> (box,box)
	~= (box,box)
	&& (box,box)
	<<| (box,box)
	&<| (box,box)
	|&> (box,box)

Nom	Opérateurs indexables
	|>> (box,box)
bpchar_bloom_ops	= (character,character)
bpchar_minmax_ops	= (character,character)
	< (character,character)
	<= (character,character)
	> (character,character)
	>= (character,character)
bytea_bloom_ops	= (bytea,bytea)
bytea_minmax_ops	= (bytea,bytea)
	< (bytea,bytea)
	<= (bytea,bytea)
	> (bytea,bytea)
	>= (bytea,bytea)
char_bloom_ops	= ("char","char")
char_minmax_ops	= ("char","char")
	< ("char","char")
	<= ("char","char")
	> ("char","char")
	>= ("char","char")
date_bloom_ops	= (date,date)
date_minmax_ops	= (date,date)
	< (date,date)
	<= (date,date)
	> (date,date)
	>= (date,date)
date_minmax_multi_ops	= (date,date)
	< (date,date)
	<= (date,date)
	> (date,date)
	>= (date,date)
float4_bloom_ops	= (float4,float4)
float4_minmax_ops	= (float4,float4)
	< (float4,float4)
	> (float4,float4)
	<= (float4,float4)
	>= (float4,float4)
float4_minmax_multi_ops	= (float4,float4)
	< (float4,float4)
	> (float4,float4)
	<= (float4,float4)
	>= (float4,float4)

Nom	Opérateurs indexables
float8_bloom_ops	= (float8,float8)
float8_minmax_ops	= (float8,float8)
	< (float8,float8)
	<= (float8,float8)
	> (float8,float8)
	>= (float8,float8)
float8_minmax_multi_ops	= (float8,float8)
	< (float8,float8)
	<= (float8,float8)
	> (float8,float8)
	>= (float8,float8)
inet_inclusion_ops	<< (inet,inet)
	<<= (inet,inet)
	>> (inet,inet)
	>>= (inet,inet)
	= (inet,inet)
	&& (inet,inet)
inet_bloom_ops	= (inet,inet)
inet_minmax_ops	= (inet,inet)
	< (inet,inet)
	<= (inet,inet)
	> (inet,inet)
	>= (inet,inet)
inet_minmax_multi_ops	= (inet,inet)
	< (inet,inet)
	<= (inet,inet)
	> (inet,inet)
	>= (inet,inet)
int2_bloom_ops	= (int2,int2)
int2_minmax_ops	= (int2,int2)
	< (int2,int2)
	> (int2,int2)
	<= (int2,int2)
	>= (int2,int2)
int2_minmax_multi_ops	= (int2,int2)
	< (int2,int2)
	> (int2,int2)
	<= (int2,int2)
	>= (int2,int2)
int4_bloom_ops	= (int4,int4)
int4_minmax_ops	= (int4,int4)

Nom	Opérateurs indexables
	< (int4,int4)
	> (int4,int4)
	<= (int4,int4)
	>= (int4,int4)
int4_minmax_multi_ops	= (int4,int4)
	< (int4,int4)
	> (int4,int4)
	<= (int4,int4)
	>= (int4,int4)
int8_bloom_ops	= (bigint,bigint)
int8_minmax_ops	= (bigint,bigint)
	< (bigint,bigint)
	> (bigint,bigint)
	<= (bigint,bigint)
	>= (bigint,bigint)
int8_minmax_multi_ops	= (bigint,bigint)
	< (bigint,bigint)
	> (bigint,bigint)
	<= (bigint,bigint)
	>= (bigint,bigint)
interval_bloom_ops	= (interval,interval)
interval_minmax_ops	= (interval,interval)
	< (interval,interval)
	<= (interval,interval)
	> (interval,interval)
	>= (interval,interval)
interval_minmax_multi_ops	= (interval,interval)
	< (interval,interval)
	<= (interval,interval)
	> (interval,interval)
	>= (interval,interval)
macaddr_bloom_ops	= (macaddr,macaddr)
macaddr_minmax_ops	= (macaddr,macaddr)
	< (macaddr,macaddr)
	<= (macaddr,macaddr)
	> (macaddr,macaddr)
	>= (macaddr,macaddr)
macaddr_minmax_multi_ops	= (macaddr,macaddr)
	< (macaddr,macaddr)
	<= (macaddr,macaddr)
	> (macaddr,macaddr)

Nom	Opérateurs indexables
	>= (macaddr,macaddr)
macaddr8_bloom_ops	= (macaddr8,macaddr8)
macaddr8_minmax_ops	= (macaddr8,macaddr8)
	< (macaddr8,macaddr8)
	<= (macaddr8,macaddr8)
	> (macaddr8,macaddr8)
	>= (macaddr8,macaddr8)
macaddr8_minmax_multi_ops	= (macaddr8,macaddr8)
	< (macaddr8,macaddr8)
	<= (macaddr8,macaddr8)
	> (macaddr8,macaddr8)
	>= (macaddr8,macaddr8)
name_bloom_ops	= (name,name)
name_minmax_ops	= (name,name)
	< (name,name)
	<= (name,name)
	> (name,name)
	>= (name,name)
numeric_bloom_ops	= (numeric,numeric)
numeric_minmax_ops	= (numeric,numeric)
	< (numeric,numeric)
	<= (numeric,numeric)
	> (numeric,numeric)
	>= (numeric,numeric)
numeric_minmax_multi_ops	= (numeric,numeric)
	< (numeric,numeric)
	<= (numeric,numeric)
	> (numeric,numeric)
	>= (numeric,numeric)
oid_bloom_ops	= (oid,oid)
oid_minmax_ops	= (oid,oid)
	< (oid,oid)
	> (oid,oid)
	<= (oid,oid)
	>= (oid,oid)
oid_minmax_multi_ops	= (oid,oid)
	< (oid,oid)
	> (oid,oid)
	<= (oid,oid)
	>= (oid,oid)
pg_lsn_bloom_ops	= (pg_lsn,pg_lsn)

Nom	Opérateurs indexables
pg_lsn_minmax_ops	= (pg_lsn,pg_lsn)
	< (pg_lsn,pg_lsn)
	> (pg_lsn,pg_lsn)
	<= (pg_lsn,pg_lsn)
	>= (pg_lsn,pg_lsn)
pg_lsn_minmax_multi_ops	= (pg_lsn,pg_lsn)
	< (pg_lsn,pg_lsn)
	> (pg_lsn,pg_lsn)
	<= (pg_lsn,pg_lsn)
	>= (pg_lsn,pg_lsn)
range_inclusion_ops	= (anyrange,anyrange)
	< (anyrange,anyrange)
	<= (anyrange,anyrange)
	>= (anyrange,anyrange)
	> (anyrange,anyrange)
	&& (anyrange,anyrange)
	@> (anyrange,anyelement)
	@> (anyrange,anyrange)
	<@ (anyrange,anyrange)
	<< (anyrange,anyrange)
	>> (anyrange,anyrange)
	&< (anyrange,anyrange)
	&> (anyrange,anyrange)
	-|- (anyrange,anyrange)
text_bloom_ops	= (text,text)
text_minmax_ops	= (text,text)
	< (text,text)
	<= (text,text)
	> (text,text)
	>= (text,text)
tid_bloom_ops	= (tid,tid)
tid_minmax_ops	= (tid,tid)
	< (tid,tid)
	> (tid,tid)
	<= (tid,tid)
	>= (tid,tid)
tid_minmax_multi_ops	= (tid,tid)
	< (tid,tid)
	> (tid,tid)
	<= (tid,tid)
	>= (tid,tid)

Nom	Opérateurs indexables
timestamp_bloom_ops	= (timestamp,timestamp)
timestamp_minmax_ops	= (timestamp,timestamp)
	< (timestamp,timestamp)
	<= (timestamp,timestamp)
	> (timestamp,timestamp)
	>= (timestamp,timestamp)
timestamp_minmax_multi_ops	= (timestamp,timestamp)
	< (timestamp,timestamp)
	<= (timestamp,timestamp)
	> (timestamp,timestamp)
	>= (timestamp,timestamp)
timestamptz_bloom_ops	= (timestamptz,timestamptz)
timestamptz_minmax_ops	= (timestamptz,timestamptz)
	< (timestamptz,timestamptz)
	<= (timestamptz,timestamptz)
	> (timestamptz,timestamptz)
	>= (timestamptz,timestamptz)
timestamptz_minmax_multi_ops	= (timestamptz,timestamptz)
	< (timestamptz,timestamptz)
	<= (timestamptz,timestamptz)
	> (timestamptz,timestamptz)
	>= (timestamptz,timestamptz)
time_bloom_ops	= (time,time)
time_minmax_ops	= (time,time)
	< (time,time)
	<= (time,time)
	> (time,time)
	>= (time,time)
time_minmax_multi_ops	= (time,time)
	< (time,time)
	<= (time,time)
	> (time,time)
	>= (time,time)
timetz_bloom_ops	= (timetz,timetz)
timetz_minmax_ops	= (timetz,timetz)
	< (timetz,timetz)
	<= (timetz,timetz)
	> (timetz,timetz)
	>= (timetz,timetz)
timetz_minmax_multi_ops	= (timetz,timetz)
	< (timetz,timetz)

Nom	Opérateurs indexables
	<= (timetz,timetz)
	> (timetz,timetz)
	>= (timetz,timetz)
uuid_bloom_ops	= (uuid,uuid)
uuid_minmax_ops	= (uuid,uuid)
	< (uuid,uuid)
	> (uuid,uuid)
	<= (uuid,uuid)
	>= (uuid,uuid)
uuid_minmax_multi_ops	= (uuid,uuid)
	< (uuid,uuid)
	> (uuid,uuid)
	<= (uuid,uuid)
	>= (uuid,uuid)
varbit_minmax_ops	= (varbit,varbit)
	< (varbit,varbit)
	> (varbit,varbit)
	<= (varbit,varbit)
	>= (varbit,varbit)

71.2.1. Paramètres de classe d'opérateur

Certaines des classes d'opérateur intégrées permettent aux paramétres spécifiés d'affecter leurs comportements. Chaque opérateur de classe a son propre ensemble de paramètres autorisés. Seules les classes d'opérateur `bloom` et `minmax-multi` permettent de spécifier ces paramètres :

Les classes d'opérateur bloom acceptent ces paramètres :

`n_distinct_per_range`

Définit le nombre estimé de valeurs non NULL distinctes dans un intervalle de blocs, utilisé par les index bloom BRIN pour dimensionner le filtre Bloom. Il se comporte similairement à l'option `n_distinct` pour ALTER TABLE. Quand la valeur affectée est positive, il est supposé que chaque intervalle de blocs contient ce nombre de valeurs non NULL distinctes. Quand la valeur affectée est négative mais supérieure ou égale à -1, il est supposé que le nombre de valeurs non NULL distinctes croît linéairement avec le nombre maximum possible de lignes dans l'intervalle de blocs(environ 290 lignes par bloc). La valeur par défaut est `-0,1`, et le nombre minimum de valeurs distinctes non NULL est `16`.

`false_positive_rate`

Définit le taux désiré de faux positifs utilisé par les index bloom BRIN pour dimensionner le filtre Bloom. La valeur doit être comprise entre 0,0001 et 0,25. La valeur par défaut est 0,01, ce qui correspond à 1% pour le taux de faux positifs.

Les classes d'opérateurs minmax-multi accepte ces paramètres :

`values_per_range`

Définit le nombre maximum de valeurs conservées par les index minmax BRIN pour résumer un intervalle de blocs. Chaque valeur peut représenter un point, ou une limite d'un intervalle. La valeur doit être comprise entre 8 et 256, et la valeur par défaut est 32.

71.3. Extensibilité

L'interface BRIN possède un niveau élevé d'abstraction, qui nécessite l'implémentation de la méthode d'accès rien que pour l'implémentation de la sémantique des types de données accédées. La couche BRIN s'occupera par contre elle-même de la concurrence, l'accès et la recherche dans la structure de l'index.

Tout ce qu'il faut pour faire fonctionner la méthode d'accès BRIN est d'implémenter quelques méthodes utilisateurs, déterminant pour l'index les genre de valeurs stockées dans le résumé et la manière dont elles interagissent avec les nœuds du parcours. En bref, BRIN combine l'extensibilité avec la généralité, la réutilisation du code et une interface claire.

Il y a quatre méthodes qu'un opérateur de classe pour BRIN doit fournir :

`BrinOpcInfo *opcInfo(Oid type_oid)`

Retourne les informations internes au sujet du résumé de données de la colonne indexée. Cette valeur doit pointer vers une structure `BrinOpcInfo` (allouée avec la fonction palloc), qui a cette définition :

```
typedef struct BrinOpcInfo
{
    /* Nombre de colonnes stockées dans une colonne indexée de
 cette classe d'opérateurs */
    uint16      oi_nstored;

    /* Pointeur opaque pour l'utilisation privée de la classe
 d'opérateurs */
    void       *oi_opaque;

    /* Type des entrées cachées de la colonne stockées */
    TypeCacheEntry *oi_typcache[FLEXIBLE_ARRAY_MEMBER];
} BrinOpcInfo;
```

`BrinOpcInfo.oi_opaque` peut être utilisé par les routines d'opérateur de classe pour transmettre des informations entre les procédures de support pendant le parcours de l'index.

`bool consistent(BrinDesc *bdesc, BrinValues *column, ScanKey *keys, int nkeys)`

Retourne si toutes les entrées ScanKey sont cohérentes avec les valeurs indexées données pour cet intervalle. Le nombre d'attributs à utiliser est passé en tant que partie de la clé de parcours. De multiples clés de parcours pour un même attribut peuvent être passées en une fois ; le nombre d'entrées est déterminé par le paramètre `nkeys`.

`bool consistent(BrinDesc *bdesc, BrinValues *column, ScanKey key)`

Retourne la clé de parcours si elle est cohérente avec les valeurs indexées données pour cet intervalle. Le nombre attribué à utiliser est passé en tant que partie de la clé de parcours. Cette version est une variante pour compatibilité descendante plus ancienne de la fonction de cohérence.

`bool addValue(BrinDesc *bdesc, BrinValues *column, Datum newval, bool isnull)`

Renvoie à une ligne indexée et une valeur indexée, modifie les attributs indiqués de cette ligne, de manière à ce que le cumul représente la nouvelle valeur. Si une modification a été apportée à la ligne, la valeur `true` est retournée.

`bool unionTuples(BrinDesc *bdesc, BrinValues *a, BrinValues *b)`

Consolidation de deux lignes d'index. Ceci en prenant deux lignes d'index et en modifiant l'attribut indiqué de la première des deux, de manière à ce qu'elle représente les deux lignes. La seconde ligne n'est pas modifiée.

Une classe d'opérateurs pour BRIN peut indiquer en option la méthode suivante :

`void options(local_relopts *relopts)`

Définit un ensemble de paramètres visibles aux utilisateurs qui contrôlent le comportement d'une classe d'opérateurs.

La fonction `options` se voit donnée un pointeur vers une structure `local_relopts` qui doit être remplie avec un ensemble d'options spécifiques à la classe d'opérateur. Les options peuvent être accédées à partir des autres fonctions de support en utilisant les macros `PG_HAS_OPCLASS_OPTIONS()` et `PG_GET_OPCLASS_OPTIONS()`.

Étant donné que l'extraction des clés des valeurs indexées et la représentation de la clé dans BRIN sont flexibles, elles peuvent dépendre de paramètres spécifiés par l'utilisateur.

La distribution du noyau inclut du support pour les quatre types de classe d'opérateur : minmax, minmax-multi, inclusion et bloom. Les définitions de classes d'opérateur qui les utilisent sont envoyées en types de données basiques appropriées. Des classes d'opérateurs appropriées peuvent être définies par l'utilisateur pour d'autres types de données utilisant des définitions équivalentes, et ceci sans avoir besoin d'écrire du code source. La déclaration des entrées appropriées dans le catalogue est suffisante. Notez que les hypothèses sur les sémantiques de stratégie d'opérateurs sont embarquées dans les fonctions de support du code source.

Les classes d'opérateurs qui implémentent des sémantiques complètement différentes sont utilisables. Les implémentations fournies par les quatre principales fonctions de support présentées ci-dessous sont écrites. Notez que la compatibilité ascendante entre les versions majeures n'est pas garantie : par exemple, les fonctions de support additionnelles peuvent être requises dans des versions ultérieures.

Pour écrire une classe d'opérateurs pour un type de données qui implémente un résultat complètement ordonné, il est possible d'utiliser les fonctions de support "minmax" avec les opérateurs correspondant tel que décrit dans Tableau 71.2. Tous les membres de classe d'opérateurs (fonctions et opérateurs) sont obligatoires.

Tableau 71.2. Fonctions et numéros de support pour les classes d'opérateurs Minmax

Membre de classe d'opérateurs	Objet
Fonction de support 1	Fonction interne `brin_minmax_opcinfo()`
Fonction de support 2	Fonction interne `brin_minmax_add_value()`
Fonction de support 3	Fonction interne `brin_minmax_consistent()`
Fonction de support 4	Fonction interne `brin_minmax_union()`
Stratégie d'opérateur 1	Opérateur strictement inférieur
Stratégie d'opérateur 2	Opérateur inférieur
Stratégie d'opérateur 3	Opérateur d'égalité
Stratégie d'opérateur 4	Opérateur supérieur
Stratégie d'opérateur 5	Opérateur strictement supérieur

Pour écrire un opérateur de classe pour un type de données complexe, qui aurait des valeurs incluses dans un autre type, il est possible d'utiliser la fonction de support d'inclusion avec l'opérateur

correspondant, tel que décrit dans Tableau 71.3. Cela nécessite uniquement une simple fonction d'addition, qui peut être écrite dans n'importe quel langage. Des fonctions supplémentaires peuvent être définies pour obtenir des fonctionnalités additionnelles. Tous les opérateurs sont optionnels. Certains opérateurs requièrent d'autres opérateurs, affichés en tant que dépendances de la table.

Tableau 71.3. Fonctions et numéros de support pour les classes d'opérateurs d'inclusion

Membre de classe d'opérateurs	Objet	Dépendance
Fonction de support 1	Fonction interne `brin_inclusion_opcinfo()`	
Fonction de support 2	Fonction interne `brin_inclusion_add_value()`	
Fonction de support 3	Fonction interne `brin_inclusion_consistent()`	
Fonction de support 4	Fonction interne `brin_inclusion_union()`	
Fonction de support 11	Fonction de fusion de deux élements	
Fonction de support 12	Fonction optionnelle de vérification si les deux éléments peuvent être fusionnés	
Fonction de support 13	Fonction optionnelle de vérification si un élément est contenu dans un autre	
Fonction de support 14	Fonction optionnelle de vérification si un élement est vide	
Stratégie d'opérateur 1	Opérateur A-gauche-de	Stratégie d'opérateur 4
Stratégie d'opérateur 2	Opérateur Ne-s-etend-pas-à-la-droite-de	Stratégie d'opérateur 5
Stratégie d'opérateur 3	Opérateur chevauchement	
Stratégie d'opérateur 4	Opérateur Ne-s-etend-pas-à-la-gauche-de	Stratégie d'opérateur 1
Stratégie d'opérateur 5	Opérateur A-droite-de	Stratégie d'opérateur 2
Stratégie d'opérateur 6, 18	Opérateur Equivalent-ou-identique-à	Stratégie d'opérateur 7
Stratégie d'opérateur 7, 16, 24, 25	Opérateur Contient-ou-identique-à	
Stratégie d'opérateur 8, 26, 27	Opérateur Contient-ou-identique-à	Stratégie d'opérateur 3
Stratégie d'opérateur 9	Opérateur Ne-s-étend-pas-plus-loin	Stratégie d'opérateur 11
Stratégie d'opérateur 10	Opérateur Est-le-suivant	Stratégie d'opérateur 12
Stratégie d'opérateur 11	Opérateur Est-le-précédent	Stratégie d'opérateur 9
Stratégie d'opérateur 12	Opérateur Ne-s-étend-pas-plus-loin	Stratégie d'opérateur 10
Stratégie d'opérateur 20	Opérateur Strictement-inférieur-à	Stratégie d'opérateur 5

Membre de classe d'opérateurs	Objet	Dépendance
Stratégie d'opérateur 21	Opérateur Inférieur-à	Stratégie d'opérateur 5
Stratégie d'opérateur 22	Opérateur Strictement-supérieur-à	Stratégie d'opérateur 1
Stratégie d'opérateur 23	Opérateur supérieur-à	Stratégie d'opérateur 1

Les numéros 1 à 10 des fonctions support sont reservés pour les fonctions internes BRIN, de ce fait le niveau des fonctions SQL commence à 11. La fonction de support 11 est la principale fonction utilisée pour construire l'index. Elle doit accepter deux arguments, avec le même type de données que la la classe d'opérateurs, et renvoyer l'union des deux. La classe d'opérateur inclusion peut stocker des valeurs unies de types différents si elles sont définies avec le paramètre `STORAGE` La valeur renvoyée par la fonction union doit correspondre au type de données `STORAGE`

Les numéros 12 et 14 des fonctions de support sont fournies pour supporter les irrégularités des types de données internes. La fonction 12 est utilisée pour supporter les adresses réseaux de différentes familles qui ne sont pas fusionnables. La fonction 14 est utilisée pour supporter les intervalles vides. La fonction 13 est une fonction optionnelle mais recommandée. Elle permet à une nouvelle valeur d'être vérifiée avant d'être passée à la fonction d'union. Puisque BRIN peut raccourcir certaines opérations lorsque l'union n'est pas modifiée, utiliser cette fonction peut améliorer les performances de l'index.

Pour écrire une classe d'opérateurs pour un type de données qui implémente seulement l'opérateur égalité et supporte le hachage, il est possible d'utiliser les procédures d'appui bloom aux côtés des opérateurs correspondants, comme montré dans Tableau 71.4. Tout membre de classe d'opérateur (procédures et opérateurs) sont obligatoires.

Tableau 71.4. Procédures et numéros de support pour les classes d'opérateurs Bloom

Membre de la classe d'opérateurs	Objet
Procédure support 1	Fonction interne `brin_bloom_opcinfo()`
Procédure support 2	Fonction interne `brin_bloom_add_value()`
Procédure support 3	Fonction interne `brin_bloom_consistent()`
Procédure support 4	Fonction interne `brin_bloom_union()`
Fonction de support 5	fonction interne `brin_bloom_options()`
Procédure support 11	Fonction pour calculer le hachage d'un élément
Stratégie d'opérateur 1	Opérateur égal-à

Les procédures support numéros 1-10 sont réservées pour les fonctions internes BRIN, ainsi les fonctions au niveau SQL débutent avec le numéro 11. La fonction support numéro 11 est la principale fonction requise pour construire un index. Elle doit accepter un argument avec le même type de données que la classe d'opérateurs et retourner un hachage de la valeur.

La classe d'opérateurs minmax-multi est aussi prévue pour les types de données implémentant un ensemble totalement ordonné, et peut être vue comme une simple extension de la classe d'opérateurs minmax. Tandis que la classe d'opérateurs minmax résume les valeurs pour chaque intervalle de bloc dans un intervalle unique et contigu, minmax-multi permet de résumer dans de multiples intervalles plus petits pour améliorer la gestion des valeurs aberrantes. Il est possible d'utiliser les procédures support minmax-multi aux côtés des opérateurs correspondant, comme indiqué dans Tableau 71.5. Tous les membres de classe d'opérateurs (procédures et opérateurs) sont obligatoires.

Tableau 71.5. Procédure et numéros de support pour les classes d'opérateurs minmax-multi

Membre de classe d'opérateurs	Objet
Procédure support 1	Fonction interne `brin_minmax_multi_opcinfo()`
Procédure support 2	Fonction interne `brin_minmax_multi_add_value()`
Procédure support 3	Fonction interne `brin_minmax_multi_consistent()`
Procédure support 4	Fonction interne `brin_minmax_multi_union()`
Fonction de support 5	Fonction interne `brin_minmax_multi_options()`
Procédure support 11	Fonction de calcul de distance entre deux valeurs (longueur d'un intervalle)
Stratégie d'opérateur 1	Opérateur inférieur-à
Stratégie d'opérateur 2	Opérateur inférieur-ou-égal-à
Stratégie d'opérateur 3	Opérateur égal-à
Stratégie d'opérateur 4	Opérateur supérieur-ou-égal-à
Stratégie d'opérateur 5	Opérateur supérieur-à

Les classes d'opérateurs minmax et inclusion supportent les opérateurs utilisables sur des types de données croisés, même si cela complexifie la gestion des dépendances. La classe d'opérateurs minmax a besoin d'un ensemble complet d'opérateurs pour être définie avec deux arguments qui auraient le même type de données. Cela permet aux types de données additionnels d'être supportés en définissant un ensemble d'opérateurs supplémentaires. Les opérateurs de la classe d'opérateurs inclusion sont dépendants d'autres stratégies d'opérateur tel que décrit dans le Tableau 71.3, ou des même stratégie d'opérateur qu'eux-mêmes. Cela nécessite que l'opérateur dépendant soit défini avec le type de données `STORAGE` pour l'argument du côté gauche, et que l'autre type de données supporté se trouve du côté droit de l'opérateur de support. Vous pouvez consulter `float4_minmax_ops` comme exemple pour minmax et `box_inclusion_ops` comme exemple pour inclusion.

Chapitre 72. Index Hash

72.1. Aperçu

PostgreSQL propose une implémentation d'index hash sur disque, qui est résistant aux crashs. Tout type de données peut être indexé par un index hash, y compris les types de données qui n'ont pas un ordre linéaire bien défini. Les index hash stockent seulement la valeur hachée de la donnée en cours d'indexation. De ce fait, il n'y a pas de restrictions sur la taille de la colonne en cours d'indexation.

Les index hash supportent seulement les index à une colonne et ne gèrent pas l'unicité des valeurs.

Les index hash acceptent uniquement l'opérateur =, donc les clauses WHERE qui spécifient des opérations sur des intervalles ne seront pas capables de tirer avantages des index hash.

Chaque ligne d'un index hash stocke la valeur hachée sur 4 octets, pas la valeur réelle de la colonne. Ceci a pour conséquence qu'un index hash peut être bien plus petit que le même index en B-tree lors de l'indexation de données volumineuses, telles que des UUID, des URL, etc. L'absence de la valeur dans la colonne rend aussi tous les parcours d'index à perte. Les index hash peuvent prendre part à des parcours d'index bitmap et à des parcours inverses.

Les index hash sont plus optimisés pour des charges de travail fortes en SELECT et UPDATE qui font des recherches d'égalité sur des tables volumineuses. Dans un index B-tree, les recherches doivent descendre dans l'arbre jusqu'à trouver le bloc feuille. Dans les tables avec des millions de lignes, cette descente peut augmenter le temps d'accès aux données. L'équivalent du bloc feuille dans un index hash est appelé un bloc bucket. Dans le cas d'un index hash, l'accès à ce bloc bucket est direct, réduisant ainsi le temps d'accès dans les tables volumineuses. Cette réduction des I/O logiques est encore plus prononcée sur les index/données qui sont plus volumineuses que le cache (shared_buffers) et la RAM.

Les index hash ont été conçus pour faire face à des distributions inégales des valeurs hachées. L'accès direct aux blocs bucket fonctionne bien si les valeurs hachées sont distribuées de façon égale. Quand des insertions remplissent le bloc bucket, des blocs overflow supplémentaires sont chaînés à ce bloc bucket, étendant localement le stockage des lignes d'index qui correspondent à cette valeur hachée. Lors du parcours d'un bucket pour l'exécution des requêtes, nous avons besoin de parcourir tous les blocs overflow. De ce fait, un index hash non balancé pourrait se révéler pire qu'un B-Tree en terme de nombre d'accès aux blocs requis pour certaines données.

En résultat des cas d'overflow, nous pouvons dire que les index hash sont préférables dans le cas de données uniques, ou tout du moins pratiquement uniques, ou de données avec un petit nombre de lignes par bucket. Une façon d'éviter les problèmes est d'exclure les valeurs très fréquentes de l'index en utilisant une condition d'index partiel mais ceci n'est pas réalisable dans beaucoup de cas.

Tout comme les B-Trees, les index hash réalisent de simples suppressions de lignes d'index. Une opération de maintenance supprime les lignes d'index connues pour pouvoir être supprimées sans risque (ceux dont le bit LP_DEAD de l'identifiant de l'élément est déjà initialisé). Si une insertion ne trouve pas d'espace disponible sur un bloc, nous essayons d'éviter de créer un nouveau bloc overflow en tentant de supprimer les lignes d'index mortes. La suppression ne peut survenir si le bloc est verrouillé à ce moment. La suppression des pointeurs d'index morts survient aussi lors du VACUUM.

S'il peut, VACUUM essaiera aussi de faire tenir les lignes d'index dans aussi peu de blocs overflow que possible, minimisant ainsi la chaîne d'overflow. Si un bloc overflow devient vide, les blocs d'overflow peuvent être recyclés pour réutilisation dans les autres buckets, bien que nous ne les renvoyons jamais au système d'exploitation. Il n'y a actuellement aucune fonctionnalité pour réduire un index hash, autrement qu'en le reconstruisant avec REINDEX. Il n'existe pas non plus de fonctionnalité pour réduire le nombre de buckets.

Les index hash peuvent étendre le nombre de blocs bucket au fur et à mesure de l'augmentation du nombre de lignes indexées. La correspondance clé de hachage - numéro de bucket est choisie pour

que l'index puisse croître de façon incrémentale. Quand un nouveau bucket est à ajouter à l'index, un seul bucket existant devra être divisé, avec certains de ses enregistrements transférés dans le nouveau bucket suivant la correspondance mise à jour clé - numéro de bucket.

Cet agrandissement survient en avant-plan, ce qui pourrait augmenter la durée d'exécution des insertions par les utilisateurs. De ce fait, les index hash pourraient ne pas convenir pour des tables ayant un nombre de lignes augmentant rapidement.

72.2. Implémentation

Il existe quatre types de blocs dans un index hash : le bloc de méta-données (bloc zéro), qui contient des informations de contrôle allouées statiquement ; les blocs des buckets principaux les blocs overflow ; et les blocs bitmap qui conservent la trace des blocs overflow libérés et disponibles pour réutilisation. Dans un but d'adressage, les blocs bitmap sont vus comme un sous-ensemble des blocs overflow.

À la fois le parcours de l'index et l'insertion de lignes nécessitent de localiser le bucket où une ligne donnée doit être située. Pour faire cela, nous avons besoin du nombre de buckets, de la valeur haute (*highmask*) et de la valeur basse (*lowmask*) partir de le bloc de méta-données ; néanmoins, il n'est pas souhaitable pour des raisons de performance d'avoir à verrouiller le bloc de méta-données à chaque fois qu'il est nécessaire de réaliser cette opération.

Les blocs buckets primaires et les blocs overflow sont alloués indépendamment car n'importe quel index pourrait avoir plus ou moins de blocs overflow suivant son nombre de buckets. Le code des index hash utilise un ensemble intéressant de règles d'adressage pour accepter un nombre variable de blocs overflow sans avoir à déplacer des blocs buckets primaires après leur création.

Chaque ligne dans la table indexée est représentée par un seul enregistrement dans l'index hash. Les enregistrements de l'index hash sont stockés dans des blocs buckets et, s'ils existent, dans des blocs overflow. Nous accélérons les recherches en conservant les entrées d'index de tout bloc d'index triées par son code de hachage, permettant ainsi l'utilisation de recherche binaire dans un bloc d'index. Notez néanmoins qu'il n'y a pas de garantie d'un ordre des codes de hachage sur plusieurs blocs d'index d'un bucket.

Les algorithmes de division de bucket pour étendre un index hash sont trop complexes pour être mentionnés ici, mais ils sont décrits dans le fichier `src/backend/access/hash/README`. L'algorithme de division est garanti contre les crashs et peut être relancé s'il ne s'est pas terminé correctement.

Chapitre 73. Stockage physique de la base de données

Ce chapitre fournit un aperçu du format de stockage physique utilisé par les bases de données PostgreSQL.

73.1. Emplacement des fichiers de la base de données

Cette section décrit le format de stockage au niveau des fichiers et répertoires.

Traditionnellement, les fichiers de configuration et les fichiers de données utilisés par une instance du serveur sont stockés ensemble dans le répertoire des données, habituellement référencé en tant que PGDATA (d'après le nom de la variable d'environnement qui peut être utilisé pour le définir). Un emplacement courant pour PGDATA est /var/lib/pgsql/data. Plusieurs groupes, gérés par différentes instances du serveur, peuvent exister sur la même machine.

Le répertoire PGDATA contient plusieurs sous-répertoires et fichiers de contrôle, comme indiqué dans le Tableau 73.1. En plus de ces éléments requis, les fichiers de configuration du groupe, postgresql.conf, pg_hba.conf et pg_ident.conf sont traditionnellement stockés dans PGDATA (bien qu'il soit possible de les placer ailleurs).

Tableau 73.1. Contenu de PGDATA

Élément	Description
PG_VERSION	Un fichier contenant le numéro de version majeur de PostgreSQL
base	Sous-répertoire contenant les sous-répertoires par base de données
current_logfiles	Fichier contenant le ou les fichiers de trace en cours d'écriture par le gestionnaire de traces.
global	Sous-répertoire contenant les tables communes au groupe, telles que pg_database
pg_commit_ts	Sous-répertoire contenant des données d'horodatage des validations de transactions
pg_dynshmem	Sous-répertoire contenant les fichiers utilisés par le système de gestion de la mémoire partagée dynamique
pg_logical	Sous-répertoire contenant les données de statut pour le décodage logique
pg_multixact	Sous-répertoire contenant des données sur l'état des multi-transactions (utilisé pour les verrous de lignes partagées)
pg_notify	Sous-répertoire contenant les données de statut de LISTEN/NOTIFY
pg_replslot	Sous-répertoire contenant les données des slots de réplication
pg_serial	Sous-répertoire contenant des informations sur les transactions sérialisables validées
pg_snapshots	Sous-répertoire contenant les snapshots (images) exportés
pg_stat	Sous-répertoire contenant les fichiers permanents pour le sous-système de statistiques
pg_stat_tmp	Sous-répertoire contenant les fichiers temporaires pour le sous-système des statistiques
pg_subtrans	Sous-répertoire contenant les données d'états des sous-transactions

Élément	Description
pg_tblspc	Sous-répertoire contenant les liens symboliques vers les espaces logiques
pg_twophase	Sous-répertoire contenant les fichiers d'état pour les transactions préparées
pg_wal	Sous-répertoire contenant les fichiers WAL (Write Ahead Log)
pg_xact	Sous-répertoire contenant les données d'état de validation des transactions
postgresql.auto.conf	Fichier utilisé pour les paramètres configurés avec la commande ALTER SYSTEM
postmaster.opts	Un fichier enregistrant les options en ligne de commande avec lesquelles le serveur a été lancé la dernière fois
postmaster.pid	Un fichier verrou contenant l'identifiant du processus postmaster en cours d'exécution (PID), le chemin du répertoire de données, la date et l'heure du lancement de postmaster, le numéro de port, le chemin du répertoire du socket de domaine Unix (pourrait être vide), la première adresse valide dans listen_address (adresse IP ou *, ou vide s'il n'y a pas d'écoute TCP) et l'identifiant du segment de mémoire partagé (ce fichier est supprimé à l'arrêt du serveur)

Pour chaque base de données dans le groupe, il existe un sous-répertoire dans PGDATA/base, nommé d'après l'OID de la base de données dans pg_database. Ce sous-répertoire est l'emplacement par défaut pour les fichiers de la base de données; en particulier, ses catalogues système sont stockés ici.

Notez que les sections suivantes décrivent le comportement de la méthode d'accès table interne nommée heap et des méthodes d'accès index internes. Du fait de la nature extensible de PostgreSQL, d'autres méthodes d'accès pourraient fonctionner différemment.

Chaque table et index est stocké dans un fichier séparé. Pour les relations ordinaires, ces fichiers sont nommés d'après le numéro *filenode* de la table ou de l'index. Ce numéro est stocké dans pg_class.relfilenode. Pour les relations temporaires, le nom du fichier est de la forme t*BBB*_*FFF*, où *BBB* est l'identifiant du processus serveur qui a créé le fichier, et *FFF* et le numéro *filenode*. Dans tous les cas, en plus du fichier principal (aussi appelé *main fork*), chaque table et index a une *carte des espaces libres* (voir Section 73.3), qui enregistre des informations sur l'espace libre disponible dans la relation. La carte des espaces libres est stockée dans un fichier dont le nom est le numéro filenode suivi du suffixe _fsm. Les tables ont aussi une *carte des visibilités*, stockée dans un fichier de suffixe _vm, pour tracer les pages connues comme n'ayant pas de lignes mortes. La carte des visibilités est décrite dans Section 73.4. Les tables non tracées et les index disposent d'un troisième fichier, connu sous le nom de fichier d'initialisation. Son nom a pour suffixe _init (voir Section 73.5).

> **Attention**
>
> Notez que, bien que le filenode de la table correspond souvent à son OID, cela n'est *pas* nécessairement le cas ; certaines opérations, comme TRUNCATE, REINDEX, CLUSTER et quelques formes d'ALTER TABLE peuvent modifier le filenode tout en préservant l'OID. Évitez de supposer que filenode et OID sont identiques. De plus, pour certains catalogues système incluant pg_class lui-même, pg_class.relfilenode contient zéro. Le numéro filenode en cours est stocké dans une structure de données de bas niveau, et peut être obtenu avec la fonction pg_relation_filenode().

Quand une table ou un index dépasse 1 Go, il est divisé en *segments* d'un Go. Le nom du fichier du premier segment est identique au filenode ; les segments suivants sont nommés filenode.1, filenode.2, etc. Cette disposition évite des problèmes sur les plateformes qui ont des limitations sur les tailles des fichiers. (Actuellement, 1 Go est la taille du segment par défaut. Cette taille est ajustable en

utilisant l'option `--with-segsize` pour configurer avant de construire PostgreSQL.) En principe, les fichiers de la carte des espaces libres et de la carte de visibilité pourraient aussi nécessiter plusieurs segments, bien qu'il y ait peu de chance que cela arrive réellement.

Une table contenant des colonnes avec des entrées potentiellement volumineuses aura une table *TOAST* associée, qui est utilisée pour le stockage de valeurs de champs trop importantes pour conserver des lignes adéquates. `pg_class.reltoastrelid` établit un lien entre une table et sa table TOAST, si elle existe. Voir Section 73.2 pour plus d'informations.

Le contenu des tables et des index est discuté plus en détails dans Section 73.6.

Les tablespaces rendent ce scénario plus compliqué. Chaque espace logique défini par l'utilisateur contient un lien symbolique dans le répertoire `PGDATA/pg_tblspc`, pointant vers le répertoire physique du tablespace (celui spécifié dans la commande `CREATE TABLESPACE`). Ce lien symbolique est nommé d'après l'OID du tablespace. À l'intérieur du répertoire du tablespace, il existe un sous-répertoire avec un nom qui dépend de la version du serveur PostgreSQL, comme par exemple `PG_9.0_201008051`. (La raison de l'utilisation de ce sous-répertoire est que des versions successives de la base de données puissent utiliser le même emplacement indiqué par `CREATE TABLESPACE` sans que cela provoque des conflits.) À l'intérieur de ce répertoire spécifique à la version, il existe un sous-répertoire pour chacune des bases de données contenant des éléments dans ce tablespace. Ce sous-répertoire est nommé d'après l'OID de la base. Les tables et les index sont enregistrés dans ce répertoire et suivent le schéma de nommage des filenodes. Le tablespace `pg_default` n'est pas accédé via `pg_tblspc` mais correspond à `PGDATA/base`. De façon similaire, le tablespace `pg_global` n'est pas accédé via `pg_tblspc` mais correspond à `PGDATA/global`.

La fonction `pg_relation_filepath()` affiche le chemin entier (relatif à `PGDATA`) de toute relation. Il est souvent utile pour ne pas avoir à se rappeler toutes les différentes règles ci-dessus. Gardez néanmoins en tête que cette fonction donne seulement le nom du premier segment du fichier principal de la relation -- vous pourriez avoir besoin d'ajouter le numéro de segment et/ou les extensions `_fsm`, `_vm` ou `_init` pour trouver tous les fichiers associés avec la relation.

Les fichiers temporaires (pour des opérations comme le tri de plus de données que ce que la mémoire peut contenir) sont créés à l'intérieur de `PGDATA/base/pgsql_tmp`, ou dans un sous-répertoire `pgsql_tmp` du répertoire du tablespace si un tablespace autre que `pg_default` est indiqué pour eux. Le nom du fichier temporaire est de la forme `pgsql_tmp`*`PPP`*`.`*`NNN`*, où *`PPP`* est le PID du serveur propriétaire et *`NNN`* distingue les différents fichiers temporaires de ce serveur.

73.2. TOAST

Cette section fournit un aperçu de TOAST (*The Oversized-Attribute Storage Technique*, la technique de stockage des attributs trop grands).

Puisque PostgreSQL utilise une taille de page fixe (habituellement 8 ko) et n'autorise pas qu'une ligne s'étende sur plusieurs pages. Du coup, il n'est pas possible de stocker de grandes valeurs directement dans les champs. Pour dépasser cette limitation, les valeurs de champ volumineuses sont compressées et/ou divisées en plusieurs lignes physiques. Ceci survient de façon transparente pour l'utilisateur, avec seulement un petit impact sur le code du serveur. Cette technique est connue sous l'acronyme affectueux de TOAST (ou « the best thing since sliced bread »). L'infrastructure TOAST est aussi utilisé pour améliorer la gestion des valeurs de grande taille en mémoire.

Seuls certains types de données supportent TOAST -- il n'est pas nécessaire d'imposer cette surcharge sur les types de données qui ne produisent pas de gros volumes. Pour supporter TOAST, un type de données doit avoir une représentation (*varlena*) à longueur variable, dans laquelle, généralement, le premier mot de quatre octets contient la longueur totale de la valeur en octets (en incluant ce mot). TOAST ne restreint pas le reste de la représentation de la donnée. Les représentations spéciales appelées collectivement *valeurs TOASTées* fonctionnent en modifiant et en ré-interprétant ce mot de longueur initial. De ce fait, les fonctions C supportant un type de données TOAST-able doivent faire

attention à la façon dont elles gèrent les valeurs en entrées potentiellement TOASTées : une entrée pourrait ne pas consister en un mot longueur de quatre octets et son contenu situé après tant qu'elle n'a pas été *dé-toastée*. (Ceci se fait habituellement en appelant `PG_DETOAST_DATUM` avant toute action sur une valeur en entrée, mais dans certains cas, des approches plus efficaces sont possibles. Voir Section 38.13.1 pour plus de détails.)

TOAST récupère deux bits du mot contenant la longueur d'un varlena (ceux de poids fort sur les machines big-endian, ceux de poids faible sur les machines little endian), limitant du coup la taille logique de toute valeur d'un type de données TOAST à 1 Go (2^{30} - 1 octets). Quand les deux bits sont à zéro, la valeur est une valeur non TOASTé du type de données et les bits restants dans le mot contenant la longueur indiquent la taille totale du datum (incluant ce mot) en octets. Quand le bit de poids fort (ou de poids faible) est à un, la valeur a un en-tête de seulement un octet alors qu'un en-tête normal en fait quatre. Les bits restants donnent la taille totale du datum (incluant ce mot) en octets. Cette alternative supporte un stockage efficace en espace de valeurs plus petites que 127 octets, tout en permettant au type de données de grossir jusqu'à 1 Go si besoin. Les valeurs avec un en-tête sur un octet ne sont pas alignées par rapport à une limite particulière, alors que les valeurs avec des en-têtes à quatre octets sont au moins alignées sur une limite de quatre octets ; la suppression de cet alignement permet de gagner encore un peu d'espace supplémentaire qui est significatif quand on le compare au stockage d'une petite valeur. Voici un cas particulier. Si les bits restants d'un en-tête sur un octet sont tous à zéro (ce qui serait impossible pour une longueur auto-inclue), la valeur est un pointeur vers la donnée sur disque, avec d'autres alternatives décrites ci-dessous. Le type et la taille d'un tel *pointeur TOAST* sont déterminés par le code enregistré dans le deuxième octet du datum. Enfin, quand le premier ou dernier bit vaut 0 mais que le bit adjacent vaut 1, le contenu du datum a été compressé et doit être décompressé avant de pouvoir être utilisé. Dans ce cas, les bits restants du mot longueur de quatre octets donnent une taille totale du datum compressé, pas celles des données au départ. Notez que la compression est aussi possible pour les données de la table TOAST mais l'en-tête varlena n'indique pas si c'est le cas -- le contenu du pointeur TOAST le précise.

La technique de compression utilisée soit pour les données en ligne ou hors ligne est sélectionnable pour chaque colonne en configurant l'option de colonne `COMPRESSION` avec les commandes `CREATE TABLE` et `ALTER TABLE`. La valeur par défaut pour les colonnes sans paramétrage spécifique correspond à la valeur du paramètre default_toast_compression au moment où la donnée est insérée.

Comme mentionné, il existe plusieurs types de pointeurs TOAST. Le type le plus ancien et le plus commun est un pointeur vers des données disques stockées dans une *table TOAST* qui est séparée, bien qu'associée, de la table contenant le pointeur TOAST. Ces pointeurs *sur disque* sont créés par le code de gestion des TOAST (dans `access/common/toast_internals.c`) quand un enregistrement à stocker sur disque est trop gros pour être stocké comme d'habitude. Plus de détails sont disponibles dans Section 73.2.1. Alternativement, un pointeur TOAST peut contenir un pointeur vers des données hors-ligne qui apparaissent ailleurs en mémoire. De tels datums ont une vie courte, et n'iront jamais sur disque. Elles sont cependant utiles pour éviter de copier et de traiter plusieurs fois de grosses données. Section 73.2.2 fournit plus de détails.

73.2.1. Stockage TOAST sur disque

Si une des colonnes d'une table est TOAST-able, la table aura une table TOAST associée, dont l'OID est enregistré dans la colonne `pg_class.reltoastrelid` pour cette table. Les valeurs TOASTées sur disque sont conservées dans la table TOAST, comme décrit en détails ci-dessous.

Les valeurs hors-ligne sont divisées (après compression si nécessaire) en morceaux d'au plus `TOAST_MAX_CHUNK_SIZE` octets (par défaut, cette valeur est choisie pour que quatre morceaux de ligne tiennent sur une page, d'où les 2000 octets). Chaque morceau est stocké comme une ligne séparée dans la table TOAST de la table propriétaire. Chaque table TOAST contient les colonnes `chunk_id` (un OID identifiant la valeur TOASTée particulière), `chunk_seq` (un numéro de séquence pour le morceau de la valeur) et `chunk_data` (la donnée réelle du morceau). Un index unique sur `chunk_id` et `chunk_seq` offre une récupération rapide des valeurs. Un pointeur datum représentant une valeur TOASTée hors-ligne a par conséquent besoin de stocker l'OID de la table TOAST dans

laquelle chercher et l'OID de la valeur spécifique (son `chunk_id`). Par commodité, les pointeurs datums stockent aussi la taille logique du datum (taille de la donnée originale non compressée), la taille stockée réelle (différente si la compression a été appliquée), et la méthode de compression utilisée, si nécessaire. À partir des octets d'en-tête varlena, la taille totale d'un pointeur datum TOAST est par conséquent de 18 octets quelle que soit la taille réelle de la valeur représentée.

Le code TOAST est déclenché seulement quand une valeur de ligne à stocker dans une table est plus grande que `TOAST_TUPLE_THRESHOLD` octets (habituellement 2 ko). Le code TOAST compressera et/ou déplacera les valeurs de champ hors la ligne jusqu'à ce que la valeur de la ligne soit plus petite que `TOAST_TUPLE_TARGET` octets (habituellement là-aussi 2 ko) ou que plus aucun gain ne puisse être réalisé. Lors d'une opération UPDATE, les valeurs des champs non modifiées sont habituellement préservées telles quelles ; donc un UPDATE sur une ligne avec des valeurs hors ligne n'induit pas de coûts à cause de TOAST si aucune des valeurs hors-ligne n'est modifiée.

Le code TOAST connaît quatre stratégies différentes pour stocker les colonnes TOAST-ables :

- `PLAIN` empêche soit la compression soit le stockage hors-ligne. Ceci est la seule stratégie possible pour les colonnes des types de données non TOAST-ables.

- `EXTENDED` permet à la fois la compression et le stockage hors-ligne. Ceci est la valeur par défaut de la plupart des types de données TOAST-ables. La compression sera tentée en premier, ensuite le stockage hors-ligne si la ligne est toujours trop grande.

- `EXTERNAL` autorise le stockage hors-ligne mais pas la compression. L'utilisation d'`EXTERNAL` rendra plus rapides les opérations sur des sous-chaînes d'importantes colonnes de type `text` et `bytea` (au dépens d'un espace de stockage accrus) car ces opérations sont optimisées pour récupérer seulement les parties requises de la valeur hors-ligne lorsqu'elle n'est pas compressée.

- `MAIN` autorise la compression mais pas le stockage hors-ligne. (En réalité le stockage hors-ligne sera toujours réalisé pour de telles colonnes mais seulement en dernier ressort s'il n'existe aucune autre solution pour diminuer suffisamment la taille de la ligne pour qu'elle tienne sur une page.)

Chaque type de données TOAST-able spécifie une stratégie par défaut pour les colonnes de ce type de donnée, mais la stratégie pour une colonne d'une table donnée peut être modifiée avec `ALTER TABLE ... SET STORAGE`.

`TOAST_TUPLE_TARGET` peut être ajusté pour chaque table en utilisant `ALTER TABLE ... SET (toast_tuple_target = N)`

Cette combinaison a de nombreux avantages comparés à une approche plus directe comme autoriser le stockage des valeurs de lignes sur plusieurs pages. En supposant que les requêtes sont habituellement qualifiées par comparaison avec des valeurs de clé relativement petites, la grosse partie du travail de l'exécuteur sera réalisée en utilisant l'entrée principale de la ligne. Les grandes valeurs des attributs TOASTés seront seulement récupérées (si elles sont sélectionnées) au moment où l'ensemble de résultats est envoyé au client. Ainsi, la table principale est bien plus petite et un plus grand nombre de ses lignes tiennent dans le cache du tampon partagé, ce qui ne serait pas le cas sans aucun stockage hors-ligne. Le tri l'utilise aussi, et les tris seront plus souvent réalisés entièrement en mémoire. Un petit test a montré qu'une table contenant des pages HTML typiques ainsi que leurs URL étaient stockées en à peu près la moitié de la taille des données brutes en incluant la table TOAST et que la table principale contenait moins de 10 % de la totalité des données (les URL et quelques petites pages HTML). Il n'y avait pas de différence à l'exécution en comparaison avec une table non TOASTée, dans laquelle toutes les pages HTLM avaient été coupées à 7 ko pour tenir.

73.2.2. Stockage TOAST en mémoire, hors-ligne

Les pointeurs TOAST peuvent pointer vers des données qui ne sont pas sur disque, mais ailleurs, dans la mémoire du processus serveur en cours d'exécution. De toute évidence, de tels pointeurs ont une durée de vie courte, mais ils n'en restent pas moins utiles. Il existe actuellement deux cas : les pointeurs vers des données *indirectes* et les pointeurs vers des données *étendues*.

Les pointeurs TOAST indirectes pointent simplement vers une valeur varlena dite *non-indirect* en mémoire. Ce cas a été créé à la base comme un PoC (*Proof of Concept*), mais il est actuellement utilisé lors du décodage logique pour éviter d'avoir potentiellement à créer des enregistrements physiques dépassant 1 Go (ce que le déplacement des valeurs hors-ligne du champ dans l'enregistrement pourrait faire). L'intérêt est limité car le création du datum pointeur est totalement responsable de la survie de la donnée référencée tant que le pointeur existe, et aucune infrastructure n'a été mise en place pour aider à ça.

Les pointeurs TOAST étendus sont utiles pour les types de données complexes dont la représentation sur disque n'est pas particulièrement adaptée pour un traitement. Par exemple, la représentation varlena standard d'un tableau PostgreSQL inclut des informations sur les dimensions, un champ de bits pour les éléments NULL s'il y en a, et enfin les valeurs de tous les éléments dans l'ordre. Quand l'élément est lui-même de longueur variable, la seule façon de trouver l'élément `N` est de parcourir tous les éléments précédents. Cette représentation est appropriée pour le stockage sur disque car elle prend peu de place mais pour le traitement du tableau, il est mieux d'avoir une représentation « étendue » ou « déconstruite » pour laquelle l'emplacement de chaque élément est identifié. Le mécanisme du pointeur TOAST supporte ce besoin en autorisant un Datum passé par référencer à pointer vers soit une valeur varlena standard (la représentation sur disque) soit un pointeur TOAST vers une représentation étendue quelque part en mémoire. Les détails de cette représentation étendue sont à la discrétion du type de données, bien qu'elle doive avoir un en-tête standard et accepter les autres prérequis de l'API indiqués dans `src/include/utils/expandeddatum.h`. Les fonctions C travaillant avec le type de données doivent choisir de gérer une ou l'autre représentation. Les fonctions qui ne connaissant pas la représentation étendue, et qui de ce fait appliquent `PG_DETOAST_DATUM` à leurs données en entrée, recevront automatiquement la représentation varlena traditionnelle. De ce fait, le support d'une représentation étendue peut se faire petit à petit, une fonction à la fois.

Les pointeurs TOAST vers des valeurs étendues sont encore divisés en pointeurs *read-write* (lecture/écriture) et *read-only* (lecture seule). La représentation pointée est la même dans les deux cas, mais une fonction qui reçoit un pointeur read-write est autorisée à modifier directement la valeur référencée alors qu'une fonction qui reçoit un pointeur read-only ne l'est pas ; elle doit tout d'abord créer une copie si elle veut avoir une version modifiée de la valeur. Cette distinction et certaines conventions associées rendent possible d'éviter des copies inutiles de valeurs étendues pendant l'exécution de la requête.

Pour tous les types de pointeurs TOAST en mémoire, le code de gestion des TOAST s'assurer qu'aucun datum pointeur ne puisse être enregistré par erreur sur disque. Les pointeurs TOAST en mémoire sont automatiquement étendus en des valeurs varlena en ligne tout à fait standards avant leur enregistrement -- puis potentiellement convertis en pointeurs TOAST sur disque si l'enregistrement devient trop gros.

73.3. Carte des espaces libres

Chaque table et index, en dehors des index hash, a une carte des espaces libres (appelée aussi FSM, acronyme de *Free Space Map*) pour conserver la trace des emplacements disponibles dans la relation. Elle est stockée dans un fichier séparé du fichier des données. Le nom de fichier est le numéro relfilenode suivi du suffixe `_fsm`. Par exemple, si le relfilenode d'une relation est 12345, la FSM est stockée dans un fichier appelé `12345_fsm`, dans le même répertoire que celui utilisé pour le fichier des données.

La carte des espaces libres est organisée comme un arbre de pages FSM. Les pages FSM de niveau bas stockent l'espace libre disponible dans chaque page de la relation. Les niveaux supérieurs agrégent l'information des niveaux bas.

À l'intérieur de chaque page FSM se trouve un arbre binaire stocké dans un tableau avec un octet par nœud. Chaque nœud final représente une page de la relation, ou une page FSM de niveau bas. Dans chaque nœud non final, la valeur la plus haute des valeurs enfants est stockée. Du coup, la valeur maximum de tous les nœuds se trouve à la racine.

Voir `src/backend/storage/freespace/README` pour plus de détails sur la façon dont la FSM est structurée, et comment elle est mise à jour et recherchée. Le module pg_freespacemap peut être utilisé pour examiner l'information stockée dans les cartes d'espace libre.

73.4. Carte de visibilité

Chaque relation a une carte de visibilité (VM acronyme de *Visibility Map*) pour garder trace des pages contenant seulement des lignes connues pour être visibles par toutes les transactions actives ; elle conserve aussi la liste des blocs contenant uniquement des lignes gelées. Elle est stockée en dehors du fichier de données dans un fichier séparé nommé suivant le numéro relfilenode de la relation, auquel est ajouté le suffixe `_vm`. Par exemple, si le relfilenode de la relation est 12345, la VM est stockée dans un fichier appelé `12345_vm`, dans le même répertoire que celui du fichier de données. Notez que les index n'ont pas de VM.

La carte de visibilité enregistre deux bits pour chaque bloc de la table. Le premier bit, s'il vaut 1, indique si le bloc associé ne contient que des enregistrements visibles ou, pour le dire autrement, si le bloc ne contient aucune ligne devait être nettoyée par un VACUUM. Cette information peut aussi être utilisée par les *parcours d'index seul* pour répondre à des requêtes n'utilisant que les informations stockées dans les entrées de l'index. Le deuxième bit, s'il vaut 1, signifie que toutes les lignes du bloc associé ont été gelées. Cela signifie que même un vacuum anti-wraparound n'a pas besoin de traiter ce bloc.

Chaque fois qu'un bit est à 1, la condition est vraie à coup sûr. Par contre, dans le cas contraire, la condition peut être vraie comme fausse. Les bits de la carte de visibilité ne sont initialisés que par le VACUUM, mais sont désinitialisés par toutes opérations de modification des données sur une page.

Le module pg_visibility peut être utilisé pour examiner les informations enregistrées dans la carte de visibilité.

73.5. Fichier d'initialisation

Chaque table non journalisée et chaque index d'une table non journalisée disposent d'un fichier d'initialisation. Il s'agit d'une table ou d'un index vide du type approprié. Quand une table non journalisée doit être réinitialisée à cause d'un crash, le fichier d'initialisation est copié sur le fichier principal, et les autres fichiers de cette table sont supprimés (ils seront de nouveau créés automatiquement si nécessaire).

73.6. Emplacement des pages de la base de données

Cette section fournit un aperçu du format des pages utilisées par les tables et index de PostgreSQL.[1] Les séquences et les tables TOAST sont formatées comme des tables standards.

Dans l'explication qui suit, un *octet* contient huit bits. De plus, le terme *élément* fait référence à une valeur de données individuelle qui est stockée dans une page. Dans une table, un élément est une ligne ; dans un index, un élément est une entrée d'index.

Chaque table et index est stocké comme un tableau de *pages* d'une taille fixe (habituellement 8 ko, bien qu'une taille de page différente peut être sélectionnée lors de la compilation du serveur). Dans une table, toutes les pages sont logiquement équivalentes pour qu'un élément (ligne) particulier puisse être stocké dans n'importe quelle page. Dans les index, la première page est généralement réservée comme *métapage* contenant des informations de contrôle, et il peut exister différents types de pages à l'intérieur de l'index, suivant la méthode d'accès à l'index. Les tables ont aussi une carte de visibilité dans un fichier de suffixe `_vm`, pour tracer les pages dont on sait qu'elles ne contiennent pas de lignes mortes et qui n'ont pas du coup besoin de VACUUM.

Tableau 73.2 affiche le contenu complet d'une page. Il existe cinq parties pour chaque page.

[1] Actuellement, l'utilisation de ce format de page n'est pas requis, que ce soit pour les méthodes d'accès aux tables comme aux index. La méthode d'accès aux tables `heap` utilise toujours ce format. Toutes les méthodes existantes d'accès aux index utilisent le format basique mais les données conservées dans les pages de métadonnées ne suivent généralement pas les règles de placement des éléments.

Tableau 73.2. Disposition générale d'une page

Élément	Description
PageHeaderData	Longueur de 24 octets. Contient des informations générales sur la page y compris des pointeurs sur les espaces libres.
ItemIdData	Tableau d'identifiants d'élément pointant vers les éléments réels. Chaque enregistrement est une paire (décalage, longueur). 4 octets par élément.
Free space	L'espace non alloué. Les pointeurs de nouveaux éléments sont alloués à partir du début de cette région, les nouveaux éléments à partir de la fin.
Items	Les éléments eux-mêmes.
Special space	Données spécifiques des méthodes d'accès aux index. Différentes méthodes stockent différentes données. Vide pour les tables ordinaires.

Les 24 premiers octets de chaque page consistent en un en-tête de page (`PageHeaderData`). Son format est détaillé dans Tableau 73.3. Le premier champ trace l'entrée la plus récente dans les journaux de transactions pour cette page. Le deuxième champ contient la somme de contrôle de la page si data checksums est activé. Ensuite se trouve un champ sur deux octets contenant des drapeaux. Il est suivi de champs entiers sur deux octets (`pd_lower`, `pd_upper` et `pd_special`). Ils contiennent les décalages en octets du début de page vers le début de l'espace non alloué, vers la fin de l'espace non alloué et vers le début de l'espace spécial. Les deux octets suivants de l'en-tête de page, `pd_pagesize_version`, enregistrent la taille de la page et un indicateur de version. À partir de la version 8.3 de PostgreSQL, le numéro de version est 4 ; PostgreSQL 8.1 et 8.2 ont utilisé le numéro de version 3 ; PostgreSQL 8.0 a utilisé le numéro de version 2 ; PostgreSQL 7.3 et 7.4 ont utilisé le numéro de version 1 ; les versions précédentes utilisaient le numéro de version 0. (La disposition fondamentale de la page et le format de l'en-tête n'ont pas changé dans la plupart de ces versions mais la disposition de l'en-tête des lignes de tête a changé.) La taille de la page est seulement présente comme vérification croisée ; il n'existe pas de support pour avoir plus d'une taille de page dans une installation. Le dernier champ est une aide indiquant si traiter la page serait profitable : il garde l'information sur le plus vieux XMAX non traité de la page.

Tableau 73.3. Disposition de PageHeaderData

Champ	Type	Longueur	Description
pd_lsn	PageXLogRecPtr	8 octets	LSN : octet suivant le dernier octet de l'enregistrement WAL pour la dernière modification de cette page
pd_checksum	uint16	2 octets	Somme de contrôle de la page
pd_flags	uint16	2 octets	Bits d'état
pd_lower	LocationIndex	2 octets	Décalage jusqu'au début de l'espace libre
pd_upper	LocationIndex	2 octets	Décalage jusqu'à la fin de l'espace libre
pd_special	LocationIndex	2 octets	Décalage jusqu'au début de l'espace spécial
pd_pagesize_version	uint16	2 octets	Taille de la page et disposition de l'information du numéro de version
pd_prune_xid	TransactionId	4 bytes	Plus vieux XMAX non traité sur la page, ou zéro si aucun

Tous les détails se trouvent dans `src/include/storage/bufpage.h`.

Après l'en-tête de la page se trouvent les identificateurs d'élément (`ItemIdData`), chacun nécessitant quatre octets. Un identificateur d'élément contient un décalage d'octet vers le début d'un élément, sa longueur en octets, et quelques bits d'attributs qui affectent son interprétation. Les nouveaux identificateurs d'éléments sont alloués si nécessaire à partir du début de l'espace non alloué. Le nombre d'identificateurs d'éléments présents peut être déterminé en regardant `pd_lower`, qui est augmenté

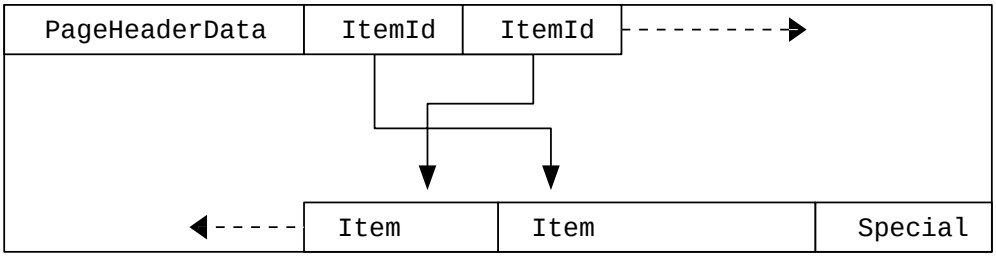
pour allouer un nouvel identificateur. Comme un identificateur d'élément n'est jamais déplacé tant qu'il n'est pas libéré, son index pourrait être utilisé sur une base à long terme pour référencer un élément, même quand l'élément lui-même est déplacé le long de la page pour compresser l'espace libre. En fait, chaque pointeur vers un élément (`ItemPointer`, aussi connu sous le nom de `CTID`), créé par PostgreSQL consiste en un numéro de page et l'index de l'identificateur d'élément.

Les éléments eux-mêmes sont stockés dans l'espace alloué en marche arrière, à partir de la fin de l'espace non alloué. La structure exacte varie suivant le contenu de la table. Les tables et les séquences utilisent toutes les deux une structure nommée `HeapTupleHeaderData`, décrite ci-dessous.

La section finale est la « section spéciale » qui pourrait contenir tout ce que les méthodes d'accès souhaitent stocker. Par exemple, les index b-tree stockent des liens vers les enfants gauche et droit de la page ainsi que quelques autres données sur la structure de l'index. Les tables ordinaires n'utilisent pas du tout de section spéciale (indiquée en configurant `pd_special` à la taille de la page).

Figure 73.1 illustre comment ces parties sont placées dans un bloc.

Figure 73.1. Disposition d'une page



73.6.1. Disposition d'une ligne de table

Toutes les lignes de la table sont structurées de la même façon. Il existe un en-tête à taille fixe (occupant 23 octets sur la plupart des machines), suivi par un bitmap NULL optionnel, un champ ID de l'objet optionnel et les données de l'utilisateur. L'en-tête est détaillé dans Tableau 73.4. Les données réelles de l'utilisateur (les colonnes de la ligne) commencent au décalage indiqué par `t_hoff`, qui doit toujours être un multiple de la distance MAXALIGN pour la plateforme. Le bitmap NULL est seulement présent si le bit *HEAP_HASNULL* est initialisé dans `t_infomask`. S'il est présent, il commence juste après l'en-tête fixe et occupe suffisamment d'octets pour avoir un bit par colonne de données (c'est-à-dire le nombre de bits identique au nombre d'attributs dans `t_infomask2`). Dans cette liste de bits, un bit 1 indique une valeur non NULL, un bit 0 une valeur NULL. Quand le bitmap n'est pas présent, toutes les colonnes sont supposées non NULL. L'ID de l'objet est seulement présent si le bit *HEAP_HASOID_OLD* est initialisé dans `t_infomask`. S'il est présent, il apparaît juste avant la limite `t_hoff`. Tout ajout nécessaire pour faire de `t_hoff` un multiple de MAXALIGN apparaîtra entre le bitmap NULL et l'ID de l'objet. (Ceci nous assure en retour que l'ID de l'objet est convenablement aligné.)

Tableau 73.4. Disposition de HeapTupleHeaderData

Champ	Type	Longueur	Description
t_xmin	TransactionId	4 octets	XID d'insertion
t_xmax	TransactionId	4 octets	XID de suppression
t_cid	CommandId	4 octets	CID d'insertion et de suppression (surcharge avec t_xvac)
t_xvac	TransactionId	4 octets	XID pour l'opération VACUUM déplaçant une version de ligne

Champ	Type	Longueur	Description
t_ctid	ItemPointerData	6 octets	TID en cours pour cette version de ligne ou pour une version plus récente
t_infomask2	uint16	2 octets	nombre d'attributs et quelques bits d'état
t_infomask	uint16	2 octets	différents bits d'options (flag bits)
t_hoff	uint8	1 octet	décalage vers les données utilisateur

Tous les détails sont disponibles dans `src/include/access/htup_details.h`.

Interpréter les données réelles peut seulement se faire avec des informations obtenues à partir d'autres tables, principalement `pg_attribute`. Les valeurs clés nécessaires pour identifier les emplacements des champs sont `attlen` et `attalign`. Il n'existe aucun moyen pour obtenir directement un attribut particulier, sauf quand il n'y a que des champs de largeur fixe et aucune colonne NULL. Tout ceci est emballé dans les fonctions *heap_getattr*, *fastgetattr* et *heap_getsysattr*.

Pour lire les données, vous avez besoin d'examinez chaque attribut à son tour. Commencez par vérifier si le champ est NULL en fonction du bitmap NULL. S'il l'est, allez au suivant. Puis, assurez-vous que vous avez le bon alignement. Si le champ est un champ à taille fixe, alors tous les octets sont placés simplement. S'il s'agit d'un champ à taille variable (attlen = -1), alors c'est un peu plus compliqué. Tous les types de données à longueur variable partagent la même structure commune d'en-tête, `struct varlena`, qui inclut la longueur totale de la valeur stockée et quelques bits d'option. Suivant les options, les données pourraient être soit dans la table de base soit dans une table TOAST ; elles pourraient aussi être compressées (voir Section 73.2).

73.7. Heap-Only Tuples (HOT)

Pour permettre une haute concurrence, PostgreSQL utilise un système connu sous l'acronyme MVCC pour enregistrer les lignes. Néanmoins, MVCC a quelques inconvénients pour les requêtes de mise à jour. Notamment, les mises à jour nécessitent que des nouvelles versions de lignes soient ajoutées aux tables. Ceci peut aussi nécessiter l'ajout de nouveaux enregistrements dans les index pour chaque ligne mise à jour, et la suppression des anciennes versions de lignes et de leurs enregistrements dans les index peut être coûteuse.

Pour aider à réduire la surcharge des mises à jour, PostgreSQL dispose d'une optimisation appelée heap-only tuples (HOT). Cette optimisation est possible quand :

- La mise à jour ne modifie aucune colonne références par les index de la table, ceci incluant les index fonctionnels et partiels.
- Il y a suffisamment d'espace libre dans le bloc contenant l'ancienne ligne pour y enregistrer la nouvelle ligne.

Dans de tels cas, *heap-only tuples* fournit deux optimisations :

- Les nouveaux enregistrements dans les index ne sont pas nécessaires pour représenter les lignes mises à jour.
- Les anciennes versions des lignes mises à jour peuvent être complètement supprimées lors d'opérations normales, celles-ci incluant les `SELECT`, au lieu de nécessiter des opérations périodiques de nettoyage. (Ceci est possible parce que les index ne référencent par leurs identifiants d'éléments de bloc.)

En résumé, les mises à jour des *heap-only tuple* peuvent seulement être créées si les colonnes utilisées par les index ne sont pas mises à jour. Vous pouvez augmenter la possibilité d'avoir suffisamment d'espace dans les blocs pour les mises à jour HOT en diminuant la valeur du paramètre `fillfactor` d'une table. Si vous ne le faites pas, les mises à jour HOT se feront tout de même parce que les nouvelles lignes iront naturellement vers de nouveaux blocs et des blocs existants avec suffisamment d'espace

ligne pour les nouvelles versions de lignes. La vue système pg_stat_all_tables permet de superviser l'occurrence des mises à jour HOT et non HOT.

Chapitre 74. Déclaration du catalogue système et contenu initial

PostgreSQL utilise de nombreux catalogues systèmes différents pour garder la trace de l'existence et les propriétés des objets des bases de données, tels que les tables et les fonctions. Il n'y a aucune différence physique entre un catalogue système et une table utilisateur standard, mais le code C des processus clients connaît la structure et les propriétés de chaque catalogue, et peut les manipuler directement à un bas niveau. Ainsi, par exemple, il est déconseillé de tenter de modifier la structure d'un catalogue à la volée ; cela casserait de nombreuses suppositions inscrites dans le code C sur comment les lignes du catalogues sont arrangées. Mais les structures des catalogues peuvent changer entre plusieurs versions majeures.

Les structures des catalogues sont déclarées dans des en-têtes de fichiers C spécialement formatées dans le répertoire `src/include/catalog/` du code source. Il existe pour chaque catalogue un fichier d'en-tête nommé d'après le catalogue (par exemple, `pg_class.h` pour `pg_class`), qui définit l'ensemble des colonnes que le catalogue a, ainsi que certaines autres propriétés basiques telles que son OID.

Beaucoup des catalogues ont des données initiales qui doivent être chargées à l'intérieur durant la phase de « bootstrap » d'initdb, pour amener le système à un point où il est capable d'exécuter des ordres SQL. (Par exemple, `pg_class.h` doit contenir une entrée pour lui-même, ainsi qu'autant d'entrées pour chacun des autres catalogues système et index.) Ces données initiales sont conservées dans un format éditable dans des fichiers de données qui sont également stockés dans le répertoire `src/include/catalog/`. Par exemple, `pg_proc.dat` décrit toutes les lignes initiales qui doivent être insérées dans le catalogue `pg_proc`.

Pour créer les fichiers de catalogue et y charger ces données initiales, un processus client fonctionnant en mode « bootstrap » lit un fichier BKI (Backend Interface) contenant les commandes et les données initiales. Le fichier `postgres.bki` utilisé dans ce mode est préparé à partir des en-têtes et fichiers de données susmentionnés, en même temps que la création d'une distribution PostgreSQL, par un script Perl nommé `genbki.pl`. Bien qu'il soit spécifique à une version précise de PostgreSQL, `postgres.bki` ne dépend pas de la plateforme et est installé dans le sous-répertoire `share` de l'arborescence installée.

`genbki.pl` produit également des fichiers d'en-tête dérivés pour chaque catalogue, par exemple `pg_class_d.h` pour le catalogue `pg_class`. Ce fichier contient des définitions de macro automatiquement générées, et peut contenir d'autres macros, déclarations d'énumérations, etc qui peuvent être utiles pour du code C client qui lit un catalogue en particulier.

La plupart des développeurs de PostgreSQL n'ont pas besoin de se préoccuper directement du fichier BKI, mais presque toutes les fonctionnalités non triviales ajoutées dans les processus clients nécessiteront de modifier les fichiers d'en-tête de catalogue et/ou les fichiers de données initiales. Le reste de ce chapitre donne des informations sur ce sujet, et par souci de complétude décrit le format de fichier BKI.

74.1. Règles de déclaration de catalogue système

La partie cruciale d'un fichier d'en-tête de catalogue est une définition de structure C décrivant l'agencement de chaque ligne dans le catalogue. Cela commence avec une macro `CATALOG`, qui, pour autant que le compilateur C est concerné, est juste un raccourci pour `typedef struct FormData_`*`catalogname`*. Chaque champ dans cette structure donne naissance à une colonne de catalogue. Les champs peuvent être annotés en utilisant les macros de propriété BKI décrites dans `genbki.h`, par exemple pour définir une valeur par défaut pour un champ pour le marquer comme potentiellement NULL ou non. La ligne `CATALOG` peut également être annotée, avec d'autres

macros de propriété décrites dans `genbki.h`, pour définir d'autres propriétés du catalogue dans son ensemble, par exemple s'il s'agit d'une relation partagée.

Le code de cache du catalogue système (et la plupart du code concernant le catalogue en général) part du principe que la partie de taille fixe de toutes les lignes de tous les catalogues système sont vraiment présentes, car il associe cette déclaration de structure C sur elles. Ainsi, tous les champs de longueur variable et tous les champs potentiellement NULL doivent être placés à la fin, et ils ne peuvent pas être accédés comme des champs de structure. Par exemple, si vous essayez de positionner `pg_type.typrelid` à NULL, cela échouerait quand certaines parties du code essaient de référencer `typetup->typrelid` (ou pire, `typetup->typelem`, car cela suit `typrelid`). Cela aurait pour conséquence des erreurs aléatoires ou même des erreurs de segmentation.

Comme protection contre ce type d'erreurs, les champs de longueur variable ou potentiellement NULL ne devraient pas être fait directement visibles pour le compilateur C. Cela se fait en les entourant de `#ifdef CATALOG_VARLEN` ... `#endif` (où `CATALOG_VARLEN` est un symbole qui n'est jamais défini). Cela empêche le code C d'imprudemment essayer d'accéder à des champs qui pourraient ne pas être là ou pourraient être à des décalage différents. Comme protection contre la création de lignes incorrectes, nous exigeons que toutes les colonnes qui devraient être non NULL soient marquées comme telles dans `pg_attribute`. Le code de « bootstrap » marquera automatiquement les colonnes comme `NOT NULL` si elles sont de taille fixe et ne sont précédées d'aucune colonne potentiellement NULL ou à taille variable. Quand cette règle ne convient pas, vous pouvez forcer un marquage correct en utilisant les annotations `BKI_FORCE_NOT_NULL` et `BKI_FORCE_NULL` selon les besoins.

Le code client ne devrait pas inclure de fichier d'en-tête de catalogue `pg_xxx.h`, car ces fichiers peuvent contenir du code C qui ne compilerait pas en dehors des processus clients. (Typiquement, cela arrive car ces fichiers contiennent également des déclarations pour des fonctions dans des fichiers de `src/backend/catalog/`.) À la place, le client peut inclure les en-têtes correspondantes `pg_xxx_d.h` générées, qui contiendront les OID définis par des `#define` et toute autre donnée qui peut être utile pour le code client. Si vous voulez que des macros ou d'autre code soient visibles par le code client, écrivez `#ifdef EXPOSE_TO_CLIENT_CODE` ... `#endif` autour de cette section pour demander à `genbki.pl` de copier cette section dans l'en-tête `pg_xxx_d.h`.

Une petite partie des catalogues est tellement fondamentale qu'ils ne peuvent même pas être créés par la commande BKI `create` qui est utilisée pour la plupart des catalogues, car cette commande a besoin d'écrire des informations dans ces catalogues pour décrire les nouveaux catalogues. Ceux-ci sont appelés les catalogues *bootstrap*, et en définir un nécessite beaucoup de travail supplémentaire : vous devez manuellement préparer les entrées appropriées pour eux dans le contenu pré-chargé de `pg_class` et `pg_type`, et ces entrées auront besoin d'être modifiées pour les futures changements de la structure du catalogue. (Les catalogues « bootstrap » nécessitent également des entrées pré-chargées dans `pg_attribute`, mais heureusement, `genbki.pl` gère maintenant cette corvée.) Évitez de faire des nouveaux catalogues comme catalogue « bootstrap » si cela est possible.

74.2. Données initiales du catalogue système

Chaque catalogue qui a des données initiales créées manuellement (certains n'en ont pas) a un fichier `.dat` correspondant qui contient ses données initiales dans un format éditable.

74.2.1. Format de fichier de données

Chaque fichier `.dat` contient des structures de données Perl littérales qui sont simplement évaluées pour produire une structure de données en mémoire qui consiste en un tableau de références de hash, un par ligne de catalogue. Un extrait de `pg_database.dat` légèrement modifié va vous décrire les fonctionnalités principales :

```
[
```

```
# A comment could appear here.
{ oid => '1', oid_symbol => 'Template1DbOid',
  descr => 'database\'s default template',
  datname => 'template1', encoding => 'ENCODING',
  datlocprovider => 'LOCALE_PROVIDER', datistemplate => 't',
  datallowconn => 't', datconnlimit => '-1', datfrozenxid => '0',
  datminmxid => '1', dattablespace => 'pg_default', datcollate =>
 'LC_COLLATE',
  datctype => 'LC_CTYPE', daticulocale => 'ICU_LOCALE', datacl =>
 '_null_' },

]
```

Les points à noter :

- La structure générale du fichier est : crochet ouvrant, un ensemble ou plus d'accolades qui chacune représentent une ligne de catalogue, crochet fermant. Il faut mettre une virgule après chaque accolade fermante.

- Au sein de chaque ligne de catalogue, écrivez des paires de *`clé => valeur`* séparées par des virgules. Les *`clé`*s autorisées sont les noms des colonnes du catalogue, ainsi que les clés de métadonnées `oid`, `oid_symbol`, `array_type_oid` et `descr`. (L'utilisation de `oid` et `oid_symbol` est décrite dans Section 74.2.2 ci-dessous, alors que `array_type_oid` est décrite dans Section 74.2.4. `descr` fournit une chaîne de texte de description pour l'objet, qui sera insérée dans `pg_description` ou `pg_shdescription` selon le cas.) Bien que les clés de métadonnées soient facultatives, les colonnes définies pour le catalogue doivent toutes être fournies, sauf pour le cas où le fichier `.h` du catalogue définit une valeur par défaut pour la colonne. (Dans l'exemple ci-dessus, le champ `datdba` a été volontairement omis car `pg_database.h` en fournit une valeur appropriée par défaut.)

- Toutes les valeurs doivent être entourées de guillemets simples. Il faut échapper les guillemets simples utilisés au sein d'une valeur avec un antislash. Les antislash qui doivent être utilisés comme une donnée peuvent être doublés, mais cela n'est pas nécessaire ; cela correspond aux règles Perl pour les littéraux entourés d'un guillemet simple. Veuillez noter que les antislash apparaissant comme données seront traités comme des échappements par le scanner du « bootstrap », d'après les mêmes règles que pour les échappements de chaînes de texte constantes (voir Section 4.1.2.2) ; par exemple `\t` est converti en un caractère tabulation. Si vous voulez un antislash dans la valeur finale, il vous faudra en écrire quatre : Perl en retire deux, laissant `\\` pour le scanner « bootstrap ».

- Les valeurs NULL sont représentées par `_null_`. (Veuillez noter qu'il n'y a aucun moyen de créer une valeur qui est simplement cette chaîne de texte.)

- Les commentaires sont précédés d'un #, et doivent être sur leur propre ligne.

- Les valeurs de champs référençant des OID d'autres entrées de catalogue devraient être représentées par des macros plutôt que par de vrais nombres OID. (Dans l'exemple ci-dessus, `dattablespace` contient une telle référence.) Cela est décrit dans Section 74.2.3 ci-dessous.

- Puisque les hash sont des structures de données non triées, l'ordre des champs et des lignes ne sont pas sémantiquement significatifs. Cependant, pour maintenir un aspect cohérent, nous définissions quelques règles qui sont appliquées par le script de formatage `reformat_dat_file.pl` :

  - Au sein de chaque paire d'accolades, les champs de métadonnées `oid`, `oid_symbol`, `array_type_oid` et `descr` (si présent) apparaissent en premier, dans cet ordre, puis les champs propres au catalogue apparaissent dans leur ordre défini.

  - Des retours à la ligne sont insérés entre les champs selon le besoin pour limiter la longueur de ligne à 80 caractères, si cela est possible. Un retour à la ligne est également inséré entre les champs de métadonnées et les champs normaux.

- Si le fichier de catalogue `.h` spécifie une valeur par défaut pour la colonne, et qu'une entrée de donnée a la même valeur, `reformat_dat_file.pl` omettra cette valeur du fichier de données. Cela conserve la représentation de données compacte.

- `reformat_dat_file.pl` conserve les lignes vides et les commentaires en l'état.

  Il est recommandé d'exécuter `reformat_dat_file.pl` avant de soumettre des patchs pour les données de catalogue. Par commodité, vous pouvez simplement effectuer des changements dans `src/include/catalog/` et exécuter `make reformat-dat-files`.

- Si vous voulez ajouter une nouvelle méthode pour diminuer la taille de de la représentation des données, vous devez l'implémenter dans `reformat_dat_file.pl` et également apprendre à `Catalog::ParseData()` comment remettre les données dans leur représentation complète.

74.2.2. Affectation d'OID

Il est possible de donner un OID manuellement assigné à une ligne de catalogue apparaissant dans les données initiales en écrivant un champ de métadonnées `oid => nnnn`. De plus, si un OID est assigné, une macro C pour cet OID peut être créée en écrivant un champ de métadonnée `oid_symbol => nom`.

Les lignes de catalogues préchargées doivent avoir des OID pré-assignés s'il y a des références d'OID pointant vers elles dans d'autres lignes pré-chargées. Un OID pré-assigné est également nécessaire si l'OID de la ligne doit être référencé depuis le code C. Si aucun de ces cas ne s'applique, le champ de métadonnée `oid` peut être omis, auquel cas le code de « bootstrap » assignera un OID automatiquement. En pratique, nous pré-assignons généralement des OID pour soit toutes soit aucune des lignes d'un catalogue donné, même si seulement une partie des lignes sont vraiment référencées dans d'autres catalogues.

Écrire la vraie valeur numérique d'un OID dans le code C est considéré comme une très mauvaise pratique ; il faut toujours utiliser une macro à la place. Des références directes à des OID de `pg_proc` sont suffisamment communes pour qu'il y ait un mécanisme spécial afin de créer les macros nécessaires automatiquement ; voir `src/backend/utils/Gen_fmgrtab.pl`. De même -- mais, pour raisons historiques, fait d'une autre manière -- il y a une méthode automatique pour créer les macros pour les OID de `pg_type`. Les entrées de `oid_symbol` ne sont donc pas forcément dans ces deux catalogues. De la même manière, les macros pour les OID de catalogue système et index `pg_class` sont positionnés automatiquement. Pour tous les autres catalogues systèmes, vous devez spécifier manuellement toute macro dont vous avez besoin avec les entrées `oid_symbol`.

Pour trouver un OID disponible pour une nouvelle ligne préchargée, exécutez le script `src/include/catalog/unused_oids`. Il affiche l'intervalle inclusif d'OID inutilisés (par exemple, la ligne en sortie `45-900` signifie que les OID 45 jusqu'à 900 n'ont pas encore été alloués). Pour le moment, les OID 1–9999 sont réservés pour des allocations manuelles ; le script `unused_oids` regarde simplement dans les en-têtes de catalogue et les fichiers `.dat` pour voir lesquels n'apparaissent pas. Vous pouvez également utiliser le script `duplicate_oids` pour trouver des erreurs. (`genbki.pl` va allouer des OID à toutes les lignes qui n'en auraient pas reçu manuellement, et va aussi détecter les OID dupliqués au moment de la compilation.)

Lors du choix d'OID pour l'écriture d'un patch qui n'est pas supposé être validé immédiatement, la bonne pratique est d'utiliser un groupe d'OID plus-ou-moins consécutifs en commençant par un choix aléatoire dans l'intervalle 8000 -- 9999. Cela minimise le risque de collisions d'OID avec d'autres patchs développés en parallèle. Pour garder l'intervalle 8000 -- 9999 libre pour le développement, après qu'un patch ait été validé sur le dépôt git principal, ces OID doivent être renumérotés dans un intervalle inférieur libre. Typiquement, cela sera fait à l'approche de la fin de chaque cycle de développement, déplaçant tous les OID consommés par des patchs validés lors de ce cycle en même temps. Le script `renumber_oids.pl` peut être utilisé pour faire cela. Si un patch non validé contient des OID en conflit avec un patch récemment validé, `renumber_oids.pl` peut aussi être utile pour rétablir une telle situation.

À cause de cette convention de renumérotation des OID consommés par les patchs, les OID attribués à un patch ne doivent pas être considérés comme stables jusqu'à ce que celui-ci ait été inclus dans une version officielle. Nous ne changeons toutefois pas les OID assignés manuellement une fois la version sortie, car cela pourrait engendrer des problèmes de compatibilité.

Si `genbki.pl` a besoin d'assigner un OID à un enregistrement du catalogue ne pouvant donc être un OID assigné manuellement, il utilisera une valeur de l'intervalle 10000 -- 11999. Le compteur des OID serveur est défini à 10000 au début d'une phase de « bootstrap », pour que tout objet créé à la volée lors du traitement du « bootstrap » reçoive aussi un OID dans cet intervalle. (Le mécanisme habituel d'attribution des OID fait attention à prévenir tout conflit.).

Les objets dont l'OID est en-dessous de `FirstUnpinnedObjectId` (12000) sont considérés comme « fixes », les empêchant d'être supprimés. (Il existe un petit nombre d'exceptions, qui sont gérées en dur par la fonction `IsPinnedObject()`.) initdb force le compteur d'OID à `FirstUnpinnedObjectId` dès qu'il est prêt à créer des objets non fixes. De ce fait, les objets créés dans les phases ultérieures de initdb, comme des objets créés lors de l'exécution du script `information_schema.sql`, ne seront pas fixes alors que tous les objets connus de `genbki.pl` le seront.

Les OID assignés durant les opérations normales de la base de données sont contraints entre 16384 et plus. Cela assure que l'intervalle 10000 -- 16383 soit libre pour l'assignation automatique d'OID par `genbki.pl` ou durant l'« initdb ». Ces OID assignés automatiquement ne sont pas considérés comme stables, et peuvent changer d'une installation à l'autre.

74.2.3. Recherche de référence d'OID

En principe, la référence d'une ligne de catalogue vers une autre pourrait être écrite juste en se servant de l'OID pré-assigné de la ligne à référencer dans le champ la référençant. Cependant, cela va à l'encontre de la politique du projet, car sujet aux erreurs, difficile à lire et source d'incohérence si un OID nouvellement assigné est renuméroté. C'est pourquoi `genbki.pl` fournit un mécanisme de génération de macros à utiliser à la place. Les règles sont les suivantes :

- L'utilisation de références symboliques est activée pour une colonne en particulier en attachant `BKI_LOOKUP(`*`lookuprule`*`)` à la définition de la colonne, où *`lookuprule`* est le nom du catalogue référencé, par exemple `pg_proc`. `BKI_LOOKUP` peut être attaché aux colonnes de type `Oid`, `regproc`, `oidvector`, ou `Oid[]` ; dans les deux derniers cas, cela implique d'effectuer une recherche pour chaque élément du tableau.

- Dans certaines colonnes du catalogue, des entrées à zéro sont permises à la place d'une référence valide. Si cela est permis, écrivez `BKI_LOOKUP_OPT` au lieu de `BKI_LOOKUP`. Ainsi, vous pouvez écrire `0` dans une entrée. (Si la colonne est déclarée `regproc`, vous pouvez optionnellement écrire `-` au lieu de `0`.) En dehors de ce cas spécial, toutes les entrées dans une colonne `BKI_LOOKUP` doivent être une référence symbolique. `genbki.pl` vous avertira à propos des noms non reconnus.

- Il est également permis d'attacher `BKI_LOOKUP(encoding)` aux colonnes de type entier pour référencer les encodages de caractères, ce qui n'est actuellement pas représenté par des OID de catalogue, mais par un ensemble de valeurs connues de `genbki.pl`.

- La plupart des types d'objets du catalogue sont référencées par leurs noms. Notez que ces noms doivent être strictement identiques aux entrées `typname` de `pg_type`; vous ne pouvez pas utiliser d'alias comme `integer` à la place de `int4`.

- Une fonction peut être représentée par son `proname`, s'il est unique parmi les entrées de `pg_proc.dat` (cela fonctionne comme les entrées regproc). Sinon, écrivez-les sous la forme *`proname(argtypename,argtypename,...)`*, comme pour `regprocedure`. Les noms de type des arguments doivent être écrits exactement comme ils le sont dans les champs `proargtypes` des entrées de `pg_proc.dat`. N'insérez aucun espace.

- Les opérateurs sont représentés par *`oprname(lefttype,righttype)`*, en écrivant les noms de type exactement comme ils apparaissent dans les `oprleft` et `oprright` des entrées de `pg_operator.dat`. (Écrivez `0` pour les opérandes omises d'un opérateur unaire.)

- Les noms des classes et familles d'opérateur ne sont uniques qu'au sein d'une méthode d'accès, elles sont donc représentées avec *nom_methode_acces/nom_objet*.

- Il n'est prévu de qualification par le schéma pour aucun de ces cas ; tous les objets créés durant le « bootstrap » sont prévus pour être dans le schéma pg_catalog.

genbki.pl résout toutes les références symboliques pendant son exécution, et inscrit de simples OID numériques dans les fichiers BKI émis. Le processus client de « bootstrap » n'a donc pas besoin de gérer les références symboliques.

Il est recommandé de marquer les colonnes de référence OID avec BKI_LOOKUP ou BKI_LOOKUP_OPT même si le catalogue n'a pas de données initiales nécessitant une recherche. Ceci permet à genbki.pl d'enregistrer les clés étrangères des relations qui existent dans le catalogue système. Cette information est utilisée dans les tests de régression pour vérifier les entrées incorrectes. Voir aussi les macros DECLARE_FOREIGN_KEY, DECLARE_FOREIGN_KEY_OPT, DECLARE_ARRAY_FOREIGN_KEY, et DECLARE_ARRAY_FOREIGN_KEY_OPT, qui sont utilisées pour déclarer les clés étrangères des relations qui sont trop complexes pour BKI_LOOKUP (typiquement, des clés étrangères multi-colonnes).

74.2.4. Création automatique des types de tableau

La plupart des types de données scalaires devraient avoir un type tableau correspondant (en fait, un type de tableau « varlena » standard, dont le type d'élément est un scalaire référencé par le champ typarray de l'entrée de type scalaire de pg_type). genbki.pl est capable de générer l'entrée de pg_type pour le type de tableau automatiquement dans la plupart des cas.

Pour se servir de cette fonctionnalité, il suffit d'inscrire un champ de méta-données array_type_oid=> *nnnn* dans l'entrée de type scalaire de pg_type, spécifiant l'OID à utiliser pour le type de tableau. Vous pouvez ensuite omettre le champ typarray, puisque l'OID y sera automatiquement renseigné.

Le nom du type de tableau généré est le nom du type scalaire préfixé du tiret bas. Les autres champs du type de tableau sont remplis par les annotations BKI_ARRAY_DEFAULT(*value*) de pg_type.h, ou s'il n'y en a pas, copiés du type scalaire. (Il y a aussi un cas particulier pour typalign.) Ensuite, les champs typelem et typarray des deux enregistrements sont positionnés afin de se référencer l'un et l'autre.

74.2.5. Recettes pour éditer les fichiers de données

Voici quelques suggestions pour les moyens les plus simples d'effectuer des tâches communes lors de la mise à jour de fichiers de données du catalogue.

Ajouter une nouvelle colonne avec valeur par défaut à un catalogue : Ajoutez la colonne au fichier d'en-tête avec une annotation BKI_DEFAULT(*valeur*) Le fichier de données ne doit être ajusté en ajoutant le champ dans les lignes existantes que quand il est nécessaire d'avoir autre chose que la valeur par défaut.

Ajouter une valeur par défaut à une colonne existante qui n'en a pas : Ajoutez une annotation BKI_DEFAULT au fichier d'en-tête, puis exécutez make reformat-dat-files pour supprimer les entrées de champ qui sont maintenant redondantes.

Ajouter une colonne, qu'elle ait une valeur par défaut ou non : Supprimez la colonne de l'en-tête, puis exécutez make reformat-dat-files pour supprimer les entrées du champ maintenant inutiles.

Changer ou supprimer une valeur par défaut existante : Vous ne pouvez pas simplement changer le fichier d'en-tête, puisque cela aurait pour conséquence une mauvaise interprétation des données actuelles. Tout d'abord, exécutez make expand-dat-files pour réécrire les fichiers

de données avec toutes les valeurs par défaut insérées explicitement, puis modifiez ou supprimez l'annotation `BKI_DEFAULT`, puis exécutez `make reformat-dat-files` pour supprimer à nouveau les champs superflus.

Édition en masse ad hoc : `reformat_dat_file.pl` peut être modifié pour effectuer différents types de changements en masse. Cherchez les commentaires de blocs montrant où du code unique peut être inséré. Dans l'exemple suivant, nous allons consolider deux champs booléens de `pg_proc` en un champ de type char :

1. Ajout de la nouvelle colonne, avec une valeur par défaut, à `pg_proc.h`:

```
+    /* see PROKIND_ categories below */
+    char        prokind BKI_DEFAULT(f);
```

2. Création d'un nouveau script basé sur `reformat_dat_file.pl` pour insérer les valeurs appropriées à la volée :

```
-           # At this point we have the full row in memory as a
 hash
-           # and can do any operations we want. As written, it
 only
-           # removes default values, but this script can be
 adapted to
-           # do one-off bulk-editing.
+           # One-off change to migrate to prokind
+           # Default has already been filled in by now, so
 change to other
+           # values as appropriate
+           if ($values{proisagg} eq 't')
+           {
+               $values{prokind} = 'a';
+           }
+           elsif ($values{proiswindow} eq 't')
+           {
+               $values{prokind} = 'w';
+           }
```

3. Lancement du nouveau script :

```
$ cd src/include/catalog
$ perl  rewrite_dat_with_prokind.pl  pg_proc.dat
```

   À cette étape, `pg_proc.dat` a la totalité des trois colonnes, `prokind`, `proisagg` et `proiswindow`, bien qu'elles n'apparaîtront que dans les lignes où elles ont des valeurs qui ne sont pas la valeur par défaut.

4. Suppression de l'ancienne colonne de `pg_proc.h` :

```
-    /* is it an aggregate? */
-    bool        proisagg BKI_DEFAULT(f);
-
-    /* is it a window function? */
-    bool        proiswindow BKI_DEFAULT(f);
```

5. Finalement, exécution de `make reformat-dat-files` pour supprimer les anciennes entrées inutiles de `pg_proc.dat`.

Pour plus d'exemples de scripts utilisés pour l'édition en masse, voir `convert_oid2name.pl` et `remove_pg_type_oid_symbols.pl` joints au message suivant : https://www.postgresql.org/message-id/CAJVSVGVX8gXnPm+Xa=DxR7kFYprcQ1tNcCT5D0O3ShfnM6jehA@mail.gmail.com

74.3. Format des fichiers BKI

Cette section décrit l'interprétation des fichiers BKI par le moteur de PostgreSQL. Cette description est plus facile à comprendre si le fichier `postgres.bki` est utilisé comme exemple.

L'entrée de BKI représente une séquence de commandes. Les commandes sont constituées de lexèmes (*tokens*) dont le nombre dépend de la syntaxe de la commande. Les lexèmes sont habituellement séparés par des espaces fines, mais en l'absence d'ambiguïté ce n'est pas nécessaire. Il n'y a pas de séparateur spécial pour les commandes ; le prochain lexème qui ne peut syntaxiquement pas appartenir à la commande qui précède en lance une autre. (En général, il est préférable, pour des raisons de clarté, de placer toute nouvelle commande sur une nouvelle ligne.) Les lexèmes peuvent être des mots clés, des caractères spéciaux (parenthèses, virgules, etc.), identifiants, nombres ou chaînes de caractères entre guillemets simples. Tous sont sensibles à la casse.

Les lignes qui débutent par # sont ignorées.

74.4. Commandes BKI

`create` *`tablename tableoid`* [`bootstrap`] [`shared_relation`] [`rowtype_oid` *`oid`*] (*`name1`* = *`type1`* [FORCE NOT NULL | FORCE NULL] [, *`name2`* = *`type2`* [FORCE NOT NULL | FORCE NULL], ...])

Crée une table nommée *`nomtable`*, possédant l'OID *`tableoid`* et composée des colonnes données entre parenthèses.

Les types de colonnes suivants sont supportés directement par `bootstrap.c`: `bool`, `bytea`, `char` (1 byte), `name`, `int2`, `int4`, `regproc`, `regclass`, `regtype`, `text`, `oid`, `tid`, `xid`, `cid`, `int2vector`, `oidvector`, `_int4` (array), `_text` (array), `_oid` (array), `_char` (array), `_aclitem` (array). Bien qu'il soit possible de créer des tables contenant des colonnes d'autres types, cela ne peut pas être réalisé avant que `pg_type` ne soit créé et rempli avec les entrées appropriées. (Ce qui signifie en fait que seuls ces types de colonnes peuvent être utilisés dans les tables utilisant le « bootstrap » mais que les catalogues ne l'utilisant pas peuvent contenir tout type interne.)

Quand `bootstrap` est précisé, la table est uniquement construite sur disque ; rien n'est entré dans `pg_class`, `pg_attribute`, etc, pour cette table. Du coup, la table n'est pas accessible par les opérations SQL standard tant que ces entrées ne sont pas réalisées en dur (à l'aide de commandes `insert`). Cette option est utilisée pour créer `pg_class`, etc.

La table est créée partagée si `shared_relation` est indiqué. L'OID du type de ligne de la table (OID de `pg_type`) peut en option être indiquée via la clause `rowtype_oid` ; dans le cas contraire, un OID est automatiquement généré pour lui. (La clause `rowtype_oid` est inutile si `bootstrap` est spécifié, mais il peut néanmoins être fourni pour documentation.)

`open` *`nomtable`*

Ouvre la table nommée *`nomtable`* pour l'ajout de données. Toute table alors ouverte est fermée.

`close` *`nomtable`*

Ferme la table ouverte. Le nom de la table peut être indiqué pour vérification mais ce n'est pas nécessaire.

`insert` ([*`valeur_oid`*] *`valeur1 valeur2 ...`*)

Insère une nouvelle ligne dans la table ouverte en utilisant *`valeur1`*, *`valeur2`*, etc., comme valeurs de colonnes.

La valeur NULL peut être indiquée en utilisant le mot clé spécial `_null_`. Les valeurs qui ne ressemblent pas à des identifiants ou à des chaînes de nombres doivent être placées entre guillemets simples. (Pour inclure un guillemet simple dans une valeur, écrivez la deux fois. Les caractères d'échappement de chaîne (les antislash) sont aussi autorisés dans une chaîne.)

`declare` [`unique`] `index` *`nomindex oidindex`* `on` *`nomtable`* `using` *`nomam`* (*`classeop1 nom1`* [, ...])

Crée un index nommé *`nomindex`*, d'OID *`indexoid`*, sur la table nommée *`nomtable`* en utilisant la méthode d'accès nommée *`nomam`*. Les champs à indexer sont appelés *`nom1`*, *`nom2`* etc., et les classes d'opérateurs à utiliser sont respectivement *`classeop1`*, *`classeop2`* etc. Le fichier index est créé et les entrées appropriées du catalogue sont ajoutées pour lui, mais le contenu de l'index n'est pas initialisé par cette commande.

`declare toast` *`toasttableoid toastindexoid`* `on` *`nomtable`*

Crée une table TOAST pour la table nommée *`nomtable`*. La table TOAST se voit affecter l'OID *`toasttableoid`* et son index l'OID *`toastindexoid`*. Comme avec `declare index`, le remplissage de l'index est reporté.

`build indices`

Remplit les index précédemment déclarés.

74.5. Structure du fichier BKI de « bootstrap »

La commande `open` ne peut pas être utilisée avant que les tables qu'elle utilise n'existent et n'aient des entrées pour la table à ouvrir. (Ces tables minimales sont `pg_class`, `pg_attribute`, `pg_proc` et `pg_type`.) Pour permettre le remplissage de ces tables elles-mêmes, `create` utilisé avec l'option `bootstrap` ouvre implicitement la table créée pour l'insertion de données.

De la même façon, les commandes `declare index` et `declare toast` ne peuvent pas être utilisées tant que les catalogues systèmes dont elles ont besoin n'ont pas été créés et remplis.

Du coup, la structure du fichier `postgres.bki` doit être :

1. `create bootstrap` une des tables critiques
2. `insert` les données décrivant au moins les tables critiques
3. `close`
4. À répéter pour les autres tables critiques.
5. `create` (sans `bootstrap`) une table non critique
6. `open`
7. `insert` les données souhaitées
8. `close`

9. À répéter pour les autres tables non critiques.

10. Définir les index et les tables TOAST.

11. `build indices`

Il existe, sans doute, d'autres dépendances d'ordre non documentées.

74.6. Exemple BKI

La séquence de commandes suivante crée la table `test_table` avec l'OID 420, trois colonnes `oid`, `cola` et `colb` de types respectifs `oid`, `int4` et `text` et insère deux lignes dans la table :

```
create test_table 420 (oid = oid, cola = int4, colb = text)
open test_table
insert ( 421 1 'valeur 1' )
insert ( 422 2 _null_ )
close test_table
```

Chapitre 75. Comment le planificateur utilise les statistiques

Ce chapitre est construit sur les informations fournies dans Section 14.1 et Section 14.2 pour montrer certains détails supplémentaires sur la façon dont le planificateur utilise les statistiques système pour estimer le nombre de lignes que chaque partie d'une requête pourrait renvoyer. C'est une partie importante du processus de planification, fournissant une bonne partie des informations pour le calcul des coûts.

Le but de ce chapitre n'est pas de documenter le code en détail mais plutôt de présenter un aperçu du fonctionnement. Ceci aidera peut-être la phase d'apprentissage pour quelqu'un souhaitant lire le code.

75.1. Exemples d'estimation des lignes

Les exemples montrés ci-dessous utilisent les tables de la base de tests de régression de PostgreSQL. Les affichages indiqués sont pris depuis la version 8.3. Le comportement des versions précédentes (ou ultérieures) pourrait varier. Notez aussi que, comme ANALYZE utilise un échantillonage statistique lors de la réalisation des statistiques, les résultats peuvent changer légèrement après toute exécution d'ANALYZE.

Commençons avec une requête simple :

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Comment le planificateur détermine la cardinalité de `tenk1` est couvert dans Section 14.2 mais est répété ici pour être complet. Le nombre de pages et de lignes est trouvé dans `pg_class` :

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';


 relpages | reltuples
----------+-----------
      358 |     10000
```

Ces nombres sont corrects à partir du dernier VACUUM ou ANALYZE sur la table. Le planificateur récupère ensuite le nombre de pages actuel dans la table (c'est une opération peu coûteuse, ne nécessitant pas un parcours de table). Si c'est différent de `relpages`, alors `reltuples` est modifié en accord pour arriver à une estimation actuelle du nombre de lignes. Dans cet exemple, la valeur de `relpages` est mise à jour, donc l'estimation du nombre de lignes est identique à `reltuples`.

Passons à un exemple avec une condition dans sa clause WHERE :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                    QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007
 width=244)
   Recheck Cond: (unique1 < 1000)
```

```
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80
 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

Le planificateur examine la condition de la clause `WHERE` et cherche la fonction de sélectivité à partir de l'opérateur `<` dans `pg_operator`. C'est contenu dans la colonne `oprrest` et le résultat, dans ce cas, est `scalarltsel`. La fonction `scalarltsel` récupère l'histogramme pour `unique1` à partir de `pg_statistic`. Pour les requêtes manuelles, il est plus simple de regarder dans la vue `pg_stats` :

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

                    histogram_bounds
------------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

Ensuite, la fraction de l'histogramme occupée par « < 1000 » est traitée. C'est la sélectivité. L'histogramme divise l'ensemble en plus petites parties d'égales fréquences, donc tout ce que nous devons faire est de localiser la partie où se trouve notre valeur et compter une *partie* d'elle et *toutes* celles qui la précèdent. La valeur 1000 est clairement dans la seconde partie (993–1997), donc en supposant une distribution linéaire des valeurs à l'intérieur de chaque partie, nous pouvons calculer la sélectivité comme étant :

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max -
 bucket[2].min))/num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

c'est-à-dire une partie complète plus une fraction linéaire de la seconde, divisée par le nombre de parties. Le nombre de lignes estimées peut maintenant être calculé comme le produit de la sélectivité et de la cardinalité de `tenk1` :

```
rows = rel_cardinality * selectivity
     = 10000 * 0.100697
     = 1007  (rounding off)
```

Maintenant, considérons un exemple avec une condition d'égalité dans sa clause `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

De nouveau, le planificateur examine la condition de la clause `WHERE` et cherche la fonction de sélectivité pour `=`, qui est `eqsel`. Pour une estimation d'égalité, l'histogramme n'est pas utile ; à la place, la liste des valeurs les plus communes (*most common values*, d'où l'acronyme MCV fréquemment utilisé) est utilisé pour déterminer la sélectivité. Regardons-les avec quelques colonnes supplémentaires qui nous seront utiles plus tard :

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs
 FROM pg_stats
```

```
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
n_distinct        | 676
most_common_vals  |
 {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs |
 {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

Comme `CRAAAA` apparaît dans la liste des MCV, la sélectivité est tout simplement l'entrée correspondante dans la liste des fréquences les plus courantes (MCF, acronyme de *Most Common Frequencies*) :

```
selectivity = mcf[3]
            = 0.003
```

Comme auparavant, le nombre estimé de lignes est seulement le produit de ceci avec la cardinalité de `tenk1` comme précédemment :

```
rows = 10000 * 0.003
     = 30
```

Maintenant, considérez la même requête mais avec une constante qui n'est pas dans la liste MCV :

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

C'est un problème assez différent, comment estimer la sélectivité quand la valeur n'est *pas* dans la liste MCV. L'approche est d'utiliser le fait que la valeur n'est pas dans la liste, combinée avec la connaissance des fréquences pour tout les MCV :

```
selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 +
 0.003 +
                    0.003 + 0.003 + 0.003 + 0.003))/(676 - 10)
            = 0.0014559
```

C'est-à-dire ajouter toutes les fréquences pour les MCV et les soustraire d'un, puis les diviser par le nombre des *autres* valeurs distinctes. Notez qu'il n'y a pas de valeurs NULL, donc vous n'avez pas à vous en inquiéter (sinon nous pourrions soustraire la fraction NULL à partir du numérateur). Le nombre estimé de lignes est ensuite calculé comme d'habitude :

```
rows = 10000 * 0.0014559
     = 15  (rounding off)
```

L'exemple précédent avec `unique1 < 1000` était une sursimplification de ce que `scalarltsel` faisait réellement ; maintenant que nous avons vu un exemple de l'utilisation des MCV, nous pouvons

ajouter quelques détails supplémentaires. L'exemple était correct aussi loin qu'il a été car, comme `unique1` est une colonne unique, elle n'a pas de MCV (évidemment, n'avoir aucune valeur n'est pas plus courant que toute autre valeur). Pour une colonne non unique, il y a normalement un histogramme et une liste MCV, et *l'histogramme n'inclut pas la portion de la population de colonne représentée par les MCV*. Nous le faisons ainsi parce que cela permet une estimation plus précise. Dans cette situation, `scalarltsel` s'applique directement à la condition (c'est-à-dire « < 1000 ») pour chaque valeur de la liste MCV, et ajoute les fréquence des MCV pour lesquelles la condition est vérifiée. Ceci donne une estimation exacte de la sélectivité dans la portion de la table qui est MCV. L'histogramme est ensuite utilisée de la même façon que ci-dessus pour estimer la sélectivité dans la portion de la table qui n'est pas MCV, et ensuite les deux nombres sont combinés pour estimer la sélectivité. Par exemple, considérez

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 < 'IAAAAA';

                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=3077 width=244)
   Filter: (stringu1 < 'IAAAAA'::name)
```

Nous voyons déjà l'information MCV pour `stringu1`, et voici son histogramme :

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

                                histogram_bounds
------------------------------------------------------------------------------
 {AAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,VAAAAA,XLAAAA,ZZAAAA}
```

En vérifiant la liste MCV, nous trouvons que la condition `stringu1 < 'IAAAAA'` est satisfaite par les six premières entrées et non pas les quatre dernières, donc la sélectivité dans la partie MCV de la population est :

```
selectivity = sum(relevant mvfs)
            = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
            = 0.01833333
```

Additionner toutes les MFC nous indique aussi que la fraction totale de la population représentée par les MCV est de 0.03033333, et du coup la fraction représentée par l'histogramme est de 0.96966667 (encore une fois, il n'y a pas de NULL, sinon nous devrions les exclure ici). Nous pouvons voir que la valeur `IAAAAA` tombe près de la fin du troisième jeton d'histogramme. En utilisant un peu de suggestions sur la fréquence des caractères différents, le planificateur arrive à l'estimation 0.298387 pour la portion de la population de l'histogramme qui est moindre que `IAAAAA`. Ensuite nous combinons les estimations pour les populations MCV et non MCV :

```
selectivity = mcv_selectivity + histogram_selectivity *
 histogram_fraction
            = 0.01833333 + 0.298387 * 0.96966667
            = 0.307669

rows        = 10000 * 0.307669
            = 3077  (rounding off)
```

Dans cet exemple particulier, la correction à partir de la liste MCV est très petit car la distribution de la colonne est réellement assez plat (les statistiques affichant ces valeurs particulières comme étant plus communes que les autres sont principalement dûes à une erreur d'échantillonnage). Dans un cas plus typique où certaines valeurs sont significativement plus communes que les autres, ce processus compliqué donne une amélioration utile dans la précision car la sélectivité pour les valeurs les plus communes est trouvée exactement.

Maintenant, considérons un cas avec plus d'une condition dans la clause `WHERE` :

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 =
 'xxx';

                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=23.80..396.91 rows=1 width=244)
   Recheck Cond: (unique1 < 1000)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80
 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

Le planificateur suppose que les deux conditions sont indépendantes, pour que les sélectivités individuelles des clauses puissent être multipliées ensemble :

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 =
 'xxx')
            = 0.100697 * 0.0014559
            = 0.0001466

rows        = 10000 * 0.0001466
            = 1  (rounding off)
```

Notez que l'estimation du nombre de lignes renvoyées à partir du bitmap index scan reflète seulement la condition utilisée avec l'index ; c'est important car cela affecte l'estimation du coût pour les récupérations suivantes sur la table.

Enfin, nous examinerons une requête qui implique une jointure :

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
------------------------------------------------------------------------------
------------------------------------------------------------------------------
 Nested Loop  (cost=4.64..456.23 rows=50 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.64..142.17 rows=50
 width=244)
         Recheck Cond: (unique1 < 50)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.63
 rows=50 width=0)
               Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..6.27
 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

La restriction sur `tenk1`, `unique1 < 50`, est évaluée avant la jointure de boucle imbriquée. Ceci est géré de façon analogue à l'exemple précédent. Cette fois, la valeur 50 est dans la première partie de l'histogramme `unique1` :

```
selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max -
 bucket[1].min))/num_buckets
            = (0 + (50 - 0)/(993 - 0))/10
            = 0.005035

rows        = 10000 * 0.005035
            = 50  (rounding off)
```

La restriction pour la jointure est `t2.unique2 = t1.unique2`. L'opérateur est tout simplement le `=`, néanmoins la fonction de sélectivité est obtenue à partir de la colonne `oprjoin` de `pg_operator`, et est `eqjoinsel`. `eqjoinsel` recherche l'information statistique de `tenk2` et `tenk1` :

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM
 pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';

tablename  | null_frac | n_distinct | most_common_vals
-----------+-----------+------------+------------------
 tenk1     |         0 |         -1 |
 tenk2     |         0 |         -1 |
```

Dans ce cas, il n'y a pas d'information MCV pour `unique2` parce que toutes les valeurs semblent être unique, donc nous utilisons un algorithme qui relie seulement le nombre de valeurs distinctes pour les deux relations ensembles avec leur fractions NULL :

```
selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/
num_distinct1, 1/num_distinct2)
            = (1 - 0) * (1 - 0) / max(10000, 10000)
            = 0.0001
```

C'est-à-dire, soustraire la fraction NULL pour chacune des relations, et divisez par le maximum du nombre de valeurs distinctes. Le nombre de lignes que la jointure pourrait émettre est calculé comme la cardinalité du produit cartésien de deux inputs, multiplié par la sélectivité :

```
rows = (outer_cardinality * inner_cardinality) * selectivity
     = (50 * 10000) * 0.0001
     = 50
```

S'il y avait eu des listes MCV pour les deux colonnes, `eqjoinsel` aurait utilisé une comparaison directe des listes MCV pour déterminer la sélectivité de jointure à l'intérieur de la aprtie des populations de colonne représentées par les MCV. L'estimation pour le reste des populations suit la même approche affichée ici.

Notez que nous affichons `inner_cardinality` à 10000, c'est-à-dire la taille non modifiée de `tenk2`. Il pourrait apparaître en inspectant l'affichage `EXPLAIN` que l'estimation des lignes jointes vient de 50 * 1, c'est-à-dire que le nombre de lignes externes multiplié par le nombre estimé de lignes obtenu par chaque parcours d'index interne sur `tenk2`. Mais ce n'est pas le cas : la taille de la relation jointe est estimée avant tout plan de jointure particulier considéré. Si tout fonctionne si bien, alors

les deux façons d'estimer la taille de la jointure produiront la même réponse mais, à cause de l'erreur d'arrondi et d'autres facteurs, ils divergent quelque fois significativement.

Pour les personnes intéressées par plus de détails, l'estimation de la taille d'une table (avant toute clause `WHERE`) se fait dans `src/backend/optimizer/util/plancat.c`. La logique générique pour les sélectivités de clause est dans `src/backend/optimizer/path/clausesel.c`. Les fonctions de sélectivité spécifiques aux opérateurs se trouvent principalement dans `src/backend/utils/adt/selfuncs.c`.

75.2. Exemples de statistiques multivariées

75.2.1. Dépendances fonctionnelles

La corrélation multivariée peut être démontrée avec un jeu de test très simple -- une table avec deux colonnes, chacune contenant les mêmes valeurs :

```
CREATE TABLE t (a INT, b INT);
INSERT INTO t SELECT i % 100, i % 100 FROM generate_series(1,
 10000) s(i);
ANALYZE t;
```

Comme expliqué dans Section 14.2, l'optimiseur peut déterminer la cardinalité de `t` en utilisant le nombre de pages et de lignes obtenues dans `pg_class` :

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 't';

 relpages | reltuples
----------+-----------
       45 |     10000
```

La distribution des données est très simple; il n'y a que 100 valeurs différentes dans chaque colonne, distribuées de manière uniforme.

L'exemple suivant montre le résultat de l'estimation d'une condition `WHERE` sur la colonne `a` :

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..170.00 rows=100 width=8) (actual
 rows=100 loops=1)
   Filter: (a = 1)
   Rows Removed by Filter: 9900
```

L'optimiseur examine la condition et détermine que la sélectivité de cette clause est de 1%. En comparant cette estimation avec le nombre de lignes réel, on voit que l'estimation est très précise (elle est en fait exacte car la table est très petite). En changeant la clause `WHERE` pour utiliser la colonne `b`, un plan identique est généré. Mais observons ce qui arrive si nous appliquons la même condition sur chacune des colonnes, en les combinant avec `AND` :

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b =
 1;
                                 QUERY PLAN
```

```
-------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=100
 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

L'optimiseur estime la sélectivité pour chaque condition individuellement, en arrivant à la même estimation d'1% comme au dessus. Puis il part du principe que les conditions sont indépendantes, et multiplie donc leurs sélectivité, produisant une estimation de sélectivité finale d'uniquement 0.01%. C'est une sous estimation importante, puisque le nombre réel de lignes correspondant aux conditions (100) est d'un ordre de grandeur deux fois plus haut.

Ce problème peut être corrigé en créant un objet statistiques qui demandera à ANALYZE de calculer des statistiques multivariées de dépendances fonctionnelles sur les deux colonnes :

```
CREATE STATISTICS stts (dependencies) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b =
 1;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=100 width=8) (actual
 rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

75.2.2. Nombre N-Distinct Multivarié

Un problème similaire apparaît avec l'estimation de la cardinalité d'un ensemble de plusieurs colonnes, tel que le nombre de groupes qu'une clause GROUP BY générerait. Quand GROUP BY liste une seule colonne, l'estimation n-distinct (qui est visible comme le nombre de lignes estimé par le nœud HashAggregate) est très précis :

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a;
                                      QUERY PLAN
-------------------------------------------------------------------------------
 HashAggregate  (cost=195.00..196.00 rows=100 width=12) (actual
 rows=100 loops=1)
   Group Key: a
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=4)
 (actual rows=10000 loops=1)
```

Mais sans statistiques multivariées, l'estimation du nombre de groupe dans une requête ayant deux colonnes dans le GROUP BY, comme dans l'exemple suivant, est faux d'un ordre de grandeur :

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                      QUERY PLAN
-------------------------------------------------------------------------------
 HashAggregate  (cost=220.00..230.00 rows=1000 width=16) (actual
 rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8)
 (actual rows=10000 loops=1)
```

En redéfinissant l'objet statistiques pour inclure un nombre n-distinct pour les deux colonnes, l'estimation est bien améliorée :

```
DROP STATISTICS stts;
CREATE STATISTICS stts (dependencies, ndistinct) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM t GROUP BY a, b;
                                        QUERY PLAN
-------------------------------------------------------------------------------
 HashAggregate  (cost=220.00..221.00 rows=100 width=16) (actual
 rows=100 loops=1)
   Group Key: a, b
   ->  Seq Scan on t  (cost=0.00..145.00 rows=10000 width=8)
 (actual rows=10000 loops=1)
```

75.2.3. Listes MCV

Comme expliqué dans Section 75.2.1, les dépendances fonctionnelles sont un type de statistiques peu coûteux et très efficace, mais leur limitation principale est leur nature globale (traquer les dépendances uniquement au niveau de la colonne, pas entre les valeurs des colonnes individuelles).

Cette section introduit la variante des listes MCV (valeurs les plus communes), une extension directe de la statistique par colonne décrite dans Section 75.1. Ces statistiques adressent la limitation du stockage de valeurs individuelles mais elles sont naturellement plus coûteuses, à la fois pour la construction des statistiques lors du ANALYZE, pour le stockage et pour le temps de planification.

Étudions cette requête à partir de Section 75.2.1, mais cette fois avec une liste MCV crée à partir du même ensemble de colonnes (assurez-vous de supprimer les dépendances fonctionnelles, pour s'assurer que le planificateur utilise les statistiques nouvellement créées).

```
DROP STATISTICS stts;
CREATE STATISTICS stts2 (mcv) ON a, b FROM t;
ANALYZE t;
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b =
 1;
                                     QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=100 width=8) (actual
 rows=100 loops=1)
   Filter: ((a = 1) AND (b = 1))
   Rows Removed by Filter: 9900
```

L'estimation est aussi précise qu'avec les dépendances fonctionnelles grâce à la petite volumétrie de la table et à une distribution simple avec un petit nombre de valeurs distinctes. Avant de regarder les deuxième requête, qui n'était pas géré particulièrement bien par les dépendances fonctionnelles, inspectons un peu la liste MCV.

Inspecter la liste MCV est possible en utilisant la fonction pg_mcv_list_items.

```
SELECT m.* FROM pg_statistic_ext join pg_statistic_ext_data on (oid
 = stxoid),
                pg_mcv_list_items(stxdmcv) m WHERE stxname =
 'stts2';
```

```
 index |  values  | nulls | frequency | base_frequency
-------+----------+-------+-----------+----------------
     0 | {0, 0}   | {f,f} |      0.01 |         0.0001
     1 | {1, 1}   | {f,f} |      0.01 |         0.0001
   ...
    49 | {49, 49} | {f,f} |      0.01 |         0.0001
    50 | {50, 50} | {f,f} |      0.01 |         0.0001
   ...
    97 | {97, 97} | {f,f} |      0.01 |         0.0001
    98 | {98, 98} | {f,f} |      0.01 |         0.0001
    99 | {99, 99} | {f,f} |      0.01 |         0.0001
(100 rows)
```

Ceci confirme qu'il y a 100 combinaisons distinctes dans les deux colonnes, et que leur fréquence est pratiquement identique (fréquence de 1% frequency pour les deux). La fréquence de base est la fréquence calculée par les statistiques par colonne, comme si il n'y avait pas de statistiques multi-colonnes. S'il y avait des valeurs NULL dans une des colonnes, cela se serait vu dans la colonne `nulls`.

Lors de l'estimation de la sélectivité, le planificateur applique toutes les conditions sur les éléments de la liste MCV, puis additionne les fréquences de celles qui correspondent. Voir `mcv_clauselist_selectivity` dans `src/backend/statistics/mcv.c` pour les détails.

Comparé aux dépendances fonctionnelles, les listes MCV ont deux avantages majeurs. Tout d'abord, la liste enregistre les valeurs réelles, rendant possible la décision des combinaisons compatibles.

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a = 1 AND b =
 10;
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=0
 loops=1)
   Filter: ((a = 1) AND (b = 10))
   Rows Removed by Filter: 10000
```

Ensuite, les listes MCV gèrent un plus grand nombre de types de clause, pas uniquement les clauses d'égalité comme les dépendances fonctionnelles. Voir par exemple la requête d'intervalle, présentée précédemment :

```
EXPLAIN (ANALYZE, TIMING OFF) SELECT * FROM t WHERE a <= 49 AND b >
 49;
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on t  (cost=0.00..195.00 rows=1 width=8) (actual rows=0
 loops=1)
   Filter: ((a <= 49) AND (b > 49))
   Rows Removed by Filter: 10000
```

75.3. Statistiques de l'optimiseur et sécurité

L'accès à la table `pg_statistic` est restreint aux superutilisateurs pour que les autres utilisateurs ne puissent apprendre le contenu des tables des autres utilisateurs. Certaines fonctions d'estimation

de la sélectivité utiliseront un opérateur fourni par l'utilisateur (soit l'opérateur apparaissant dans la requête, soit un opérateur lié) pour analyser les statistiques enregistrées. Par exemple, pour déterminer si la valeur la plus commune est applicable, l'estimateur de sélectivité devra exécuter l'opérateur `=` approprié pour comparer la constante de la requête avec la valeur enregistrée. De ce fait, la donnée dans `pg_statistic` est potentielement fournie aux opérateurs définis par l'utilisateur. Un opérateur créé de façon approprié peut intentionnellement donner les opérandes fournis (par exemple en les enregistrant ou en les écrivant dans une table différente) ou en les exposant par erreur en affichant leur valeurs dans des messages d'erreur, auxquels cas il pourrait exposer les données provenant de `pg_statistic` à un utilisateur qui ne devrait pas être capable de les voir.

Pour empêcher cela, ce qui suit s'applique à toute fonction interne d'estimation de la sélectivité. Lors de la planification d'une requête, pour pouvoir utiliser les statistiques enregistrées, soit l'utilisateur actuel doit avoir le droit `SELECT` sur la table ou les colonnes impliquées, columns, soit l'opérateur utilisé doit être `LEAKPROOF` (plus exactement, la fonction utilisée par cet opérateur). Dans le cas contraire, l'estimateur de la sélectivité se comportera comme si aucune statistique n'était disponible, et le planificateur procédera avec les informations par défaut.

Si un utilisateur n'a pas le droit requis pour la table ou les colonnes, alors dans de nombreux cas, la requête renverra une erreur pour refus de droit, auquel cas ce mécanisme est invisible en pratique. Mais si l'utilisateur est en train de lire une vue avec une barrière de sécurité, alors le planificateur pourrait souhaiter de vérifier les statistiques de la table sous-jacente qui n'est normalement pas accessible par l'utilisateur. Dans ce cas, l'opérateur devra être sans fuite. Dans le cas contraire, les statistiques ne seront pas utilisées. Il n'y a pas de retour direct sur cela, en dehors du fait que le plan pourrait être non optimal. Si un utilisateur suspecte que cela lui arrive, il pourrait exécuter la requête avec un utilisateur disposant de plus de droits pour voir si cela cause la génération d'un autre plan.

Cette restriction s'applique seulement aux cas où le planificateur aurait besoin d'exécuter un opérateur défini par un utilisateur sur une ou plusieurs valeurs de `pg_statistic`. De ce fait, le planificateur a l'autorisation d'utiliser des informations statistiques génériques, telles que la fraction de valeurs nulles ou le nombre de valeurs distinctes dans une colonne, quelque soit les droits d'accès.

Les fonctions d'estimation de la sélectivité contenues dans des extensions de tierces parties qui opèrent potentiellement sur des statistiques avec des opérateurs définis par les utilisateurs devraient suivre les mêmes règles de sécurité. Consultez le code source de PostgreSQL pour des exemples.

Chapitre 76. Format de manifeste de sauvegarde

Le manifeste de sauvegarde généré par pg_basebackup est conçu initialement pour vérifier la sauvegarde avec pg_verifybackup. Cependant, d'autres outils peuvent lire le manifeste de sauvegarde et utiliser les informations contenues pour leurs propres besoins. À cette fin, ce chapitre décrit le format du fichier manifeste de sauvegarde.

Un manifeste de sauvegarde est un document JSON encodé en UTF-8. (Même si en général, il est recommandé que les documents JSON soient en Unicode, PostgreSQL permet l'utilisation des types de données `json` et `jsonb` avec tout type d'encodage serveur supporté. Il n'y a, cependant, pas ce type d'exception pour le manifeste de sauvegarde.) Le document JSON est toujours un objet ; les clés présentes dans cet objet sont décrites dans la section suivante.

76.1. Objet de niveau supérieur du manifeste de sauvegarde

Le document JSON du manifeste de sauvegarde contient les clés suivantes.

`PostgreSQL-Backup-Manifest-Version`

La valeur associée est toujours l'entier 1.

`Files`

La valeur associée est toujours une liste d'objets, chacun décrivant un fichier présent dans la sauvegarde. Aucune entrée n'est présente dans cette liste pour les fichiers de journaux de transactions nécessaires pour utiliser la sauvegarde ou pour le manifeste de sauvegarde lui même. La structure de chaque objet de la liste est décrite dans Section 76.2.

`WAL-Ranges`

La valeur associée est toujours une liste d'objets, chacun décrivant un intervalle d'enregistrements de journaux de transaction chronologiques pour utiliser la sauvegarde. La structure de ces objets est décrite dans Section 76.3.

`Manifest-Checksum`

Cette clé est toujours présente sur la dernière ligne du fichier manifeste de sauvegarde. La valeur associée est une somme de contrôle SHA256 de toutes les lignes précédentes. La méthode de somme de contrôle est utilisée ici pour permettre aux clients d'effectuer l'analyse incrémentale du manifeste. Même si le contrôle de somme SHA256 est significativement plus coûteux qu'une somme de contrôle CRC32C, le manifeste doit normalement être suffisamment petit pour que le calcul supplémentaire ne soit pas significatif.

76.2. Objet fichier du manifeste de sauvegarde

Cet objet, décrivant un seul fichier, contient soit une clé `Path`, soit une clé `Encoded-Path`. Habituellement, la clé `Path` est présente. La chaîne de caractères associée est le chemin relatif du fichier à partir de la racine du dossier de la sauvegarde. Les fichiers localisés dans un tablespace défini par l'utilisateur auront un chemin dont les deux premiers composant sont `pg_tblspc` et l'OID du tablespace. Si le chemin n'est pas une chaîne de caractères encodée correctement en UTF-8, ou si

l'utilisateur a demandé que tous les fichiers aient un chemin encodé, alors la clé `Encoded-Path` sera présente à la place. La chaîne de caractères associée sera la même pour l'autre clé, mais encodée comme une chaîne de chiffres hexadécimaux. Chaque paire de chiffres hexadécimaux dans la chaîne représente un unique octet.

Les deux clés suivantes sont toujours présentes :

`Size`

La taille attendue du fichier, au format entier.

`Last-Modified`

La date de dernière modification du fichier vue par le serveur à la date de sauvegarde. Contrairement aux autres attributs présents dans la sauvegarde, cette clé n'est pas utilisée par pg_verifybackup. Elle est incluse uniquement à titre informatif.

Si la sauvegarde a été générée avec le somme de contrôle des fichiers, la clé suivante est présente :

`Checksum-Algorithm`

L'algorithme de somme de contrôle utilisé pour calculer la somme de contrôle de ce fichier. Actuellement, il est le même pour chaque fichier dans le manifeste de sauvegarde, mais cela pourra changer dans une version future. Maintenant, les algorithmes de somme de contrôle supportés sont `CRC32C`, `SHA224`, `SHA256`, `SHA384` et `SHA512`.

`Checksum`

La somme de contrôle pour ce fichier, stockée comme une série de caractères hexadécimaux, deux pour chaque octet de la somme de contrôle.

76.3. Objet intervalle des journaux de transaction du manifeste de sauvegarde

L'objet décrivant l'intervalle de journaux de transaction a trois clés :

`Timeline`

La chronologie pour cet intervalle d'enregistrements de journaux de transaction, sous format entier.

`Start-LSN`

Le LSN à partir duquel la relecture doit démarrer sur la timeline indiquée pour utiliser cette sauvegarde. Le LSN est stocké sous un format habituellement utilisé par PostgreSQL ; une chaîne de caractères constituée de deux chaînes de caractères hexadécimaux séparées par un slash.

`End-LSN`

Le plus récent LSN à partir duquel la relecture sur la timeline indiquée peut terminer quand la sauvegarde est utilisée. Cette clé est au même format que `Start-LSN`.

Normalement, il n'y aura qu'un seul intervalle de journaux de transaction. Cependant si une sauvegarde est générée à partir d'un standby en cours de changement de timeline suite à une promotion, il est possible d'avoir plusieurs intervalles, chacun sur une différente chronologie. Il n'y aura jamais plusieurs intervalles de journaux de transactions présents sur la même chronologie.

Partie VIII. Annexes

Table des matières

A. Codes d'erreur de PostgreSQL 2690
B. Support de date/heure 2699
B.1. Interprétation des Date/Heure saisies 2699
B.2. Gestion des horodatages ambigus ou invalides 2700
B.3. Mots-clés Date/Heure 2701
B.4. Fichiers de configuration date/heure 2702
B.5. Spécification POSIX des fuseaux horaires 2703
B.6. Histoire des unités 2705
B.7. Dates Julien 2706
C. Mots-clés SQL 2708
D. Conformité SQL 2733
D.1. Fonctionnalités supportées 2734
D.2. Fonctionnalités non supportées 2745
D.3. Limitations XML et conformité à SQL/XML 2753
D.3.1. Les requêtes sont limitées à XPath 1.0 2754
D.3.2. Limites accidentelles de l'implémentation 2755
E. Notes de version 2757
E.1. Release 15.19 2757
E.1.1. Migration to Version 15.19 2757
E.1.2. Changes 2757
E.2. Release 15.18 2767
E.2.1. Migration to Version 15.18 2767
E.2.2. Changes 2767
E.3. Release 15.17 2772
E.3.1. Migration to Version 15.17 2772
E.3.2. Changes 2772
E.4. Release 15.16 2773
E.4.1. Migration to Version 15.16 2773
E.4.2. Changes 2773
E.5. Release 15.15 2777
E.5.1. Migration to Version 15.15 2777
E.5.2. Changes 2777
E.6. Release 15.14 2782
E.6.1. Migration to Version 15.14 2782
E.6.2. Changes 2782
E.7. Release 15.13 2788
E.7.1. Migration to Version 15.13 2788
E.7.2. Changes 2788
E.8. Release 15.12 2791
E.8.1. Migration to Version 15.12 2791
E.8.2. Changes 2791
E.9. Release 15.11 2792
E.9.1. Migration to Version 15.11 2792
E.9.2. Changes 2792
E.10. Release 15.10 2796
E.10.1. Migration to Version 15.10 2796
E.10.2. Changes 2796
E.11. Release 15.9 2797
E.11.1. Migration to Version 15.9 2797
E.11.2. Changes 2798
E.12. Release 15.8 2803
E.12.1. Migration to Version 15.8 2803
E.12.2. Changes 2803
E.13. Release 15.7 2808
E.13.1. Migration to Version 15.7 2808

E.13.2. Changes 2808
E.14. Release 15.6 2813
E.14.1. Migration to Version 15.6 2813
E.14.2. Changes 2813
E.15. Release 15.5 2818
E.15.1. Migration to Version 15.5 2819
E.15.2. Changes 2819
E.16. Release 15.4 2824
E.16.1. Migration to Version 15.4 2824
E.16.2. Changes 2824
E.17. Release 15.3 2828
E.17.1. Migration to Version 15.3 2828
E.17.2. Changes 2828
E.18. Release 15.2 2835
E.18.1. Migration to Version 15.2 2836
E.18.2. Changes 2836
E.19. Release 15.1 2841
E.19.1. Migration to Version 15.1 2841
E.19.2. Changes 2841
E.20. Release 15 2843
E.20.1. Overview 2844
E.20.2. Migration to Version 15 2844
E.20.3. Changes 2847
E.20.4. Acknowledgments 2858
E.21. Versions précédentes 2865
F. Modules supplémentaires fournis 2866
F.1. adminpack 2867
F.2. amcheck 2868
F.2.1. Fonctions 2868
F.2.2. Vérification optionnelle *heapallindexed* 2871
F.2.3. Utiliser amcheck efficacement 2872
F.2.4. Réparer une corruption 2873
F.3. auth_delay 2873
F.3.1. Paramètres de configuration 2874
F.3.2. Auteur 2874
F.4. auto_explain 2874
F.4.1. Paramètres de configuration 2874
F.4.2. Exemple 2876
F.4.3. Auteur 2877
F.5. basebackup_to_shell 2877
F.5.1. Paramètres de configuration 2877
F.5.2. Auteur 2877
F.6. basic_archive 2877
F.6.1. Paramètres de configuration 2877
F.6.2. Notes 2878
F.6.3. Auteur 2878
F.7. bloom 2878
F.7.1. Paramètres 2878
F.7.2. Exemples 2878
F.7.3. Interface de la classe d'opérateur 2881
F.7.4. Limitations 2881
F.7.5. Auteurs 2881
F.8. btree_gin 2881
F.8.1. Exemple d'utilisation 2882
F.8.2. Auteurs 2882
F.9. btree_gist 2882
F.9.1. Exemple d'utilisation 2882
F.9.2. Auteurs 2883

F.10. citext 2883
F.10.1. Intérêt 2884
F.10.2. Comment l'utiliser 2884
F.10.3. Comportement des comparaisons de chaînes 2884
F.10.4. Limitations 2885
F.10.5. Auteur 2886
F.11. cube 2886
F.11.1. Syntaxe 2886
F.11.2. Précision 2887
F.11.3. Utilisation 2887
F.11.4. Par défaut 2889
F.11.5. Notes 2890
F.11.6. Crédits 2890
F.11.7. Note de l'auteur 2890
F.12. dblink 2891
F.13. dict_int 2923
F.13.1. Configuration 2923
F.13.2. Utilisation 2923
F.14. dict_xsyn 2924
F.14.1. Configuration 2924
F.14.2. Utilisation 2925
F.15. earthdistance 2926
F.15.1. Distances sur Terre à partir de cubes 2926
F.15.2. Distances sur Terre à partir de points 2927
F.16. file_fdw 2927
F.17. fuzzystrmatch 2930
F.17.1. Soundex 2930
F.17.2. Levenshtein 2931
F.17.3. Metaphone 2932
F.17.4. Double Metaphone 2932
F.18. hstore 2933
F.18.1. Représentation externe de `hstore` 2933
F.18.2. Opérateurs et fonctions `hstore` 2934
F.18.3. Index 2938
F.18.4. Exemples 2938
F.18.5. Statistiques 2939
F.18.6. Compatibilité 2940
F.18.7. Transformations 2940
F.18.8. Auteurs 2941
F.19. intagg 2941
F.19.1. Fonctions 2941
F.19.2. Exemples d'utilisation 2941
F.20. intarray 2942
F.20.1. Fonctions et opérateurs d'`intarray` 2943
F.20.2. Support des index 2944
F.20.3. Exemple 2945
F.20.4. Tests de performance 2945
F.20.5. Auteurs 2946
F.21. isn 2946
F.21.1. Types de données 2946
F.21.2. Conversions 2947
F.21.3. Fonctions et opérateurs 2947
F.21.4. Exemples 2948
F.21.5. Bibliographie 2949
F.21.6. Auteur 2949
F.22. lo 2949
F.22.1. Aperçu 2950
F.22.2. Comment l'utiliser 2950

F.22.3. Limites 2950
F.22.4. Auteur 2951
F.23. ltree 2951
F.23.1. Définitions 2951
F.23.2. Opérateurs et fonctions 2953
F.23.3. Index 2955
F.23.4. Exemple 2956
F.23.5. Transformations 2958
F.23.6. Auteurs 2958
F.24. old_snapshot 2958
F.24.1. Fonctions 2959
F.25. pageinspect 2959
F.25.1. Fonctions générales 2959
F.25.2. Fonctions Heap 2960
F.25.3. Fonctions B-tree 2961
F.25.4. Fonctions BRIN 2964
F.25.5. Fonctions GIN 2965
F.25.6. Fonctions GiST 2966
F.25.7. Fonctions Hash 2968
F.26. passwordcheck 2969
F.27. pg_buffercache 2970
F.27.1. La vue `pg_buffercache` 2970
F.27.2. Affichage en sortie 2971
F.27.3. Auteurs 2972
F.28. pgcrypto 2972
F.28.1. Fonctions de hachage généralistes 2972
F.28.2. Fonctions de hachage de mot de passe 2972
F.28.3. Fonctions de chiffrement PGP 2975
F.28.4. Fonctions de chiffrement brut (Raw) 2981
F.28.5. Fonctions d'octets au hasard 2982
F.28.6. Notes 2982
F.28.7. Auteur 2983
F.29. pg_freespacemap 2983
F.29.1. Fonctions 2983
F.29.2. Exemple de sortie 2984
F.29.3. Auteur 2984
F.30. pg_prewarm 2984
F.30.1. Fonctions 2984
F.30.2. Paramètres de configuration 2985
F.30.3. Auteur 2986
F.31. pgrowlocks 2986
F.31.1. Aperçu 2986
F.31.2. Exemple d'affichage 2987
F.31.3. Auteur 2987
F.32. pg_stat_statements 2987
F.32.1. La vue `pg_stat_statements` 2987
F.32.2. La vue `pg_stat_statements_info` 2991
F.32.3. Fonctions 2991
F.32.4. Paramètres de configuration 2992
F.32.5. Exemple de sortie 2993
F.32.6. Auteurs 2995
F.33. pgstattuple 2995
F.33.1. Fonctions 2996
F.33.2. Auteurs 3000
F.34. pg_surgery 3000
F.34.1. Fonctions 3000
F.34.2. Auteurs 3001
F.35. pg_trgm 3001

F.35.1. Concepts du trigramme (ou trigraphe) 3001
F.35.2. Fonctions et opérateurs 3002
F.35.3. Paramètres GUC 3004
F.35.4. Support des index 3004
F.35.5. Intégration à la recherche plein texte 3006
F.35.6. Références 3007
F.35.7. Auteurs 3007
F.36. pg_visibility 3007
F.36.1. Fonctions 3008
F.36.2. Auteur 3008
F.37. pg_walinspect 3009
F.37.1. Fonctions générales 3009
F.37.2. Auteur 3010
F.38. postgres_fdw 3011
F.38.1. Options FDW de postgres_fdw 3011
F.38.2. Fonctions 3017
F.38.3. Gestion des connexions 3017
F.38.4. Gestion des transactions 3018
F.38.5. Optimisation des requêtes distantes 3018
F.38.6. Environnement d'exécution de requêtes distantes 3019
F.38.7. Compatibilité entre versions 3019
F.38.8. Paramètres de configuration 3019
F.38.9. Exemples 3020
F.38.10. Auteur 3021
F.39. seg 3021
F.39.1. Explications 3021
F.39.2. Syntaxe 3022
F.39.3. Précision 3023
F.39.4. Utilisation 3023
F.39.5. Notes 3023
F.39.6. Crédits 3024
F.40. sepgsql 3024
F.40.1. Aperçu 3024
F.40.2. Installation 3025
F.40.3. Tests de régression 3026
F.40.4. Paramètres GUC 3027
F.40.5. Fonctionnalités 3027
F.40.6. Fonctions Sepgsql 3031
F.40.7. Limitations 3031
F.40.8. Ressources externes 3032
F.40.9. Auteur 3032
F.41. spi 3032
F.41.1. refint -- fonctions de codage de l'intégrité référentielle 3032
F.41.2. autoinc -- fonctions pour l'incrémentation automatique d'un champ 3034
F.41.3. insert_username -- fonctions pour tracer les utilisateurs qui ont modifié une table 3034
F.41.4. moddatetime -- fonctions pour tracer la date et l'heure de la dernière modification 3034
F.42. sslinfo 3035
F.42.1. Fonctions 3035
F.42.2. Auteur 3036
F.43. tablefunc 3037
F.43.1. Fonctions 3037
F.43.2. Auteur 3047
F.44. tcn 3047
F.45. test_decoding 3048
F.46. tsm_system_rows 3049
F.46.1. Exemples 3049

F.47. tsm_system_time 3050
F.47.1. Exemples 3050
F.48. unaccent 3050
F.48.1. Configuration 3050
F.48.2. Utilisation 3051
F.48.3. Fonctions 3052
F.49. uuid-ossp 3052
F.49.1. Fonctions de `uuid-ossp` 3053
F.49.2. Construire `uuid-ossp` 3054
F.49.3. Auteur 3054
F.50. xml2 3054
F.50.1. Notice d'obsolescence 3054
F.50.2. Description des fonctions 3054
F.50.3. `xpath_table` 3055
F.50.4. Fonctions XSLT 3058
F.50.5. Auteur 3058
G. Programmes supplémentaires fournis 3059
G.1. Applications clients 3059
G.2. Applications serveurs 3066
H. Projets externes 3068
H.1. Interfaces client 3068
H.2. Outils d'administration 3068
H.3. Langages procéduraux 3068
H.4. Extensions 3068
I. Dépôt du code source 3069
I.1. Récupérer les sources via Git 3069
J. Documentation 3070
J.1. DocBook 3070
J.2. Ensemble d'outils 3070
J.2.1. Installation sur Fedora, RHEL et dérivés 3071
J.2.2. Installation sur FreeBSD 3071
J.2.3. Paquetages Debian 3071
J.2.4. macOS 3071
J.2.5. Installation manuelle à partir des sources 3072
J.2.6. Détection par `configure` 3073
J.3. Construire la documentation 3073
J.3.1. HTML 3074
J.3.2. Pages man (de manuel) 3074
J.3.3. PDF 3074
J.3.4. Fichiers texte 3075
J.3.5. Vérification syntaxique 3075
J.4. Écriture de la documentation 3075
J.4.1. Emacs 3075
J.5. Guide des styles 3075
J.5.1. Pages de références 3075
K. Limites PostgreSQL 3078
L. Acronymes 3080
M. Glossaire 3087
N. Support des couleurs 3102
N.1. Quand les couleurs sont utilisées 3102
N.2. Configurer les couleurs 3102
O. Fonctionnalités obsolètes ou renommées 3103
O.1. fichier `recovery.conf` intégré à `postgresql.conf` 3103
O.2. Les rôles par défaut sont renommés en rôles prédéfinis 3103
O.3. `pg_xlogdump` renommé en `pg_waldump` 3103
O.4. `pg_resetxlog` renommé en `pg_resetwal` 3103
O.5. `pg_receivexlog` renommé en `pg_receivewal` 3104
P. Traduction française 3105

Annexe A. Codes d'erreur de PostgreSQL

Tous les messages émis par le serveur PostgreSQL se voient affectés des codes d'erreur sur cinq caractères. Ces codes suivent les conventions du standard SQL pour les codes « SQLSTATE ».

Les applications qui souhaitent connaître la condition d'erreur survenue peuvent tester le code d'erreur plutôt que récupérer le message d'erreur textuel. Les codes d'erreur sont moins sujets à changement au fil des versions de PostgreSQL et ne dépendent pas de la localisation des messages d'erreur. Seuls certains codes d'erreur produits par PostgreSQL sont définis par le standard SQL ; divers codes d'erreur supplémentaires, pour des conditions non définies par le standard, ont été inventés ou empruntés à d'autres bases de données.

Comme le préconise le standard, les deux premiers caractères d'un code d'erreur définissent la classe d'erreurs, les trois derniers indiquent la condition spécifique à l'intérieur de cette classe. Ainsi, une application qui ne reconnaît pas le code d'erreur spécifique peut toujours agir en fonction de la classe de l'erreur.

Tableau A.1 liste tous les codes d'erreur définis dans PostgreSQL 15.19. (Certains ne sont pas réellement utilisés mais sont définis par le standard SQL.) Les classes d'erreurs sont aussi affichées. Pour chaque classe d'erreur, il y a un code d'erreur « standard » dont les trois derniers caractères sont `000`. Ce code n'est utilisé que pour les conditions d'erreurs de cette classe qui ne possèdent pas de code plus spécifique.

Les symboles affichés dans la colonne « Nom de condition » sont aussi le nom de la condition à utiliser dans PL/pgSQL. Les noms de conditions peuvent être écrits en minuscule ou en majuscule. Notez que PL/pgSQL ne fait pas la distinction entre avertissement et erreur au niveau des noms des conditions ; il s'agit des classes 00, 01 et 02.

Pour certains types d'erreurs, le serveur rapporte le nom d'un objet de la base (une table, la colonne d'une table, le type d'une donnée ou une contrainte) associé à l'erreur ; par exemple, le nombre de la contrainte unique qui a causé une erreur de type `unique_violation`. Ces noms sont fournis dans des champs séparés du message d'erreur pour que les applications n'aient pas besoin de les extraire d'un texte prévu pour un humain et potentiellement traduit dans sa langue. À partir de PostgreSQL 9.3, cette fonctionnalité est complète pour les erreurs de la classe SQLSTATE 23 (violation d'une contrainte d'intégrité). Elle sera étendue lors des prochaines versions.

Tableau A.1. Codes d'erreur de PostgreSQL

Code erreur	Nom de condition
Classe 00 -- Succès	
00000	successful_completion
Classe 01 -- Avertissement	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature

Code erreur	Nom de condition
Classe 02 -- Aucune donnée (c'est aussi une classe avertissement d'après le standard SQL)	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
Classe 03 -- Requête SQL en cours	
03000	sql_statement_not_yet_complete
Classe 08 -- Exception de connexion	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
Classe 09 -- Exception d'action trigger	
09000	triggered_action_exception
Classe 0A -- Fonctionnalité non supportée	
0A000	feature_not_supported
Classe 0B -- Début de transaction invalide	
0B000	invalid_transaction_initiation
Classe 0F -- Exception de localisation	
0F000	locator_exception
0F001	invalid_locator_specification
Classe 0L -- Droit invalide	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Classe 0P -- Spécification invalide de rôle	
0P000	invalid_role_specification
Classe 0Z -- Exception de diagnostique	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
Classe 20 -- Case non trouvé	
20000	case_not_found
Classe 21 -- Violation de cardinalité	
21000	cardinality_violation
Classe 22 -- Exception de données	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero

Code erreur	Nom de condition
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
22013	invalid_preceding_or_following_size
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
2200H	sequence_generator_limit_exceeded
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format

Code erreur	Nom de condition
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
22030	duplicate_json_object_key_value
22031	invalid_argument_for_sql_json_datetime_function
22032	invalid_json_text
22033	invalid_sql_json_subscript
22034	more_than_one_sql_json_item
22035	no_sql_json_item
22036	non_numeric_sql_json_item
22037	non_unique_keys_in_a_json_object
22038	singleton_sql_json_item_required
22039	sql_json_array_not_found
2203A	sql_json_member_not_found
2203B	sql_json_number_not_found
2203C	sql_json_object_not_found
2203D	too_many_json_array_elements
2203E	too_many_json_object_members
2203F	sql_json_scalar_required
Classe 23 -- Violation de contrainte d'intégrité	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Classe 24 -- État invalide du curseur	
24000	invalid_cursor_state
Classe 25 -- État invalide de transaction	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction
25004	inappropriate_isolation_level_for_branch_transaction
25005	no_active_sql_transaction_for_branch_transaction

Code erreur	Nom de condition
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
25P03	idle_in_transaction_session_timeout
Classe 26 -- Nom invalide de requête SQL	
26000	invalid_sql_statement_name
Classe 27 -- Violation de modification de données par trigger	
27000	triggered_data_change_violation
Classe 28 -- Spécification invalide d'autorisation	
28000	invalid_authorization_specification
28P01	invalid_password
Classe 2B -- Les descripteurs de droits dépendants existent toujours	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Classe 2D -- Fin invalide de transaction	
2D000	invalid_transaction_termination
Classe 2F -- Exception de routine SQL	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Classe 34 -- Nom invalide du curseur	
34000	invalid_cursor_name
Classe 38 -- Exception de routine externe	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Classe 39 -- Exception d'appel de routine externe	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
Classe 3B -- Exception de savepoint	
3B000	savepoint_exception

Code erreur	Nom de condition
3B001	invalid_savepoint_specification
Classe 3D -- Nom invalide de catalogue	
3D000	invalid_catalog_name
Classe 3F -- Nom invalide de schéma	
3F000	invalid_schema_name
Classe 40 -- Annulation de transaction	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Classe 42 -- Erreur de syntaxe ou violation de règle d'accès	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
428C9	generated_always
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema

Code erreur	Nom de condition
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Classe 44 -- Violation de l'option WITH CHECK	
44000	with_check_option_violation
Classe 53 -- Ressources insuffisantes	
53000	insufficient_resources
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
Classe 54 -- Limite du programme dépassée	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
Classe 55 -- Objet pas dans l'état requis	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
55P04	unsafe_new_enum_value_usage
Classe 57 -- Intervention opérateur	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown

Code erreur	Nom de condition
57P03	cannot_connect_now
57P04	database_dropped
Classe 58 -- Erreur système (erreurs externes à PostgreSQL lui-même)	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
Classe 72 -- Échec de snapshot	
72000	snapshot_too_old
Classe F0 -- Erreur de fichier de configuration	
F0000	config_file_error
F0001	lock_file_exists
Classe HV -- Erreur du wrapper de données distantes (SQL/MED)	
HV000	fdw_error
HV005	fdw_column_name_not_found
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection

Code erreur	Nom de condition
Classe P0 -- Erreur PL/pgSQL	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
Classe XX -- Erreur interne	
XX000	internal_error
XX001	data_corrupted
XX002	index_corrupted

Annexe B. Support de date/heure

PostgreSQL utilise un analyseur heuristique interne pour le support des dates/heures saisies. Les dates et heures, saisies sous la forme de chaînes de caractères, sont découpées en champs distincts après détermination du type d'information contenue dans chaque champ. Chaque champ est interprété ; une valeur peut lui être affectée, il peut être ignoré ou encore être rejeté. Le parseur contient des tables de recherche internes pour tous les champs textuels y compris les mois, les jours de la semaine et les fuseaux horaires.

Cette annexe décrit le contenu des tables de correspondance et les méthodes utilisées par le parseur pour décoder les dates et heures.

B.1. Interprétation des Date/Heure saisies

Les chaînes en entrée de type date/heure sont décodées en utilisant le processus suivant.

1. Diviser la chaîne saisie en lexèmes et catégoriser les lexèmes en chaînes, heures, fuseaux horaires et nombres.

    a. Si le lexème numérique contient un double-point (`:`), c'est une chaîne de type heure. On inclut tous les chiffres et double-points qui suivent.

    b. Si le lexème numérique contient un tiret (`-`), une barre oblique (`/`) ou au moins deux points (`.`), c'est une chaîne de type date qui contient peut-être un mois sous forme textuelle. Si un lexème de date a déjà été reconnu, il est alors interprété comme un nom de fuseau horaire (par exemple `America/New_York`).

    c. Si le lexème n'est que numérique alors il s'agit soit d'un champ simple soit d'une date concaténée ISO 8601 (`19990113` pour le 13 janvier 1999, par exemple) ou d'une heure concaténée ISO 8601 (`141516` pour 14:15:16, par exemple).

    d. Si le lexème débute par le signe plus (`+`) ou le signe moins (`-`), alors il s'agit soit d'un fuseau horaire numérique, soit d'un champ spécial.

2. Si le lexème est une chaîne texte alphabétique, le comparer avec les différentes chaînes possibles :

    a. Vérifier si le jeton correspond à une abréviation connue d'un fuseau horaire. Ces abréviations sont fournies par le fichier de configuration décrit dans Section B.4.

    b. S'il n'est pas trouvé, rechercher dans la table interne pour vérifier si le lexème est une chaîne spéciale (`today`, par exemple), un jour (`Thursday`, par exemple), un mois (`January`, par exemple), ou du bruit (`at`, `on`, par exemple).

    c. Si le lexème n'est toujours pas trouvé, une erreur est levée.

3. Lorsque le lexème est un nombre ou un champ de nombre :

    a. S'il y a huit ou six chiffres, et qu'aucun autre champ date n'a été lu, alors il est interprété comme une « date concaténée » (`19990118` ou `990118`, par exemple). L'interprétation est `AAAAMMJJ` ou `AAMMJJ`.

    b. Si le lexème est composé de trois chiffres et qu'une année est déjà lue, alors il est interprété comme un jour de l'année.

    c. Si quatre ou six chiffres et une année sont déjà lus, alors il est interprété comme une heure (`HHMM` ou `HHMMSS`).

    d. Si le lexème est composé de trois chiffres ou plus et qu'aucun champ date n'a été trouvé, il est interprété comme une année (cela impose l'ordre aa-mm-jj des champs dates restants).

e. Dans tous les autres cas, le champ date est supposé suivre l'ordre imposé par le paramètre `datestyle` : mm-jj-aa, jj-mm-aa, ou aa-mm-jj. Si un champ jour ou mois est en dehors des limites, une erreur est levée.

4. Si BC est indiqué, le signe de l'année est inversé et un est ajouté pour le stockage interne. (Il n'y a pas d'année zéro dans le calendrier Grégorien, alors numériquement 1 BC devient l'année zéro.)

5. Si BC n'est pas indiqué et que le champ année est composé de deux chiffres, alors l'année est ajustée à quatre chiffres. Si le champ vaut moins que 70, alors on ajoute 2000, sinon 1900.

> **Astuce**
>
> Les années du calendrier Grégorien AD 1–99 peuvent être saisies avec 4 chiffres, deux zéros en tête (`0099` pour AD 99, par exemple).

B.2. Gestion des horodatages ambigus ou invalides

D'ordinaire, si une chaîne date/heure est syntaxiquement valide mais contient des valeurs de champs hors de l'intervalle, une erreur sera renvoyée. Par exemple, une entrée indiquant le 31 février sera rejetée.

Lors d'un changement d'heure, il est possible qu'une chaîne apparemment valide représente un horodatage inexistant ou ambigu. Ce genre de cas n'est pas rejeté. L'ambiguité est résolue en déterminant le décalage UTC à appliquer. Par exemple, supposons que le paramètre TimeZone est configuré à `America/New_York` :

```
=> SELECT '2018-03-11 02:30'::timestamptz;
      timestamptz
------------------------
 2018-03-11 03:30:00-04
(1 row)
```

Comme ce jour était une transition vers l'avant pour ce fuseau horaire, l'heure 2:30AM n'existe pas ; les horloges passent directement de 2h à 3h EDT. PostgreSQL interprète l'heure donnée comme s'il s'agissait de l'heure standard (UTC-5), qui se décline donc en 3:30 EDT (UTC-4).

De la même façon, prenons en considération ce comportement lors d'une transition en arrière :

```
=> SELECT '2018-11-04 01:30'::timestamptz;
      timestamptz
------------------------
 2018-11-04 01:30:00-05
(1 row)
```

À cette date, il existe deux interprétations possibles de 1:30AM ; soit 1:30AM EDT, soit une heure après la transition, 1:30AM EST. De nouveau, PostgreSQL interprète l'heure donnée comme s'il s'agissait de l'heure standard (UTC-5). Nous pouvons forcer l'autre interprétation en spécifiant le temps et sa règle de conversion :

```
=> SELECT '2018-11-04 01:30 EDT'::timestamptz;
      timestamptz
------------------------
 2018-11-04 01:30:00-04
(1 row)
```

La règle précise qui se trouve appliquée dans de tels cas est qu'un horodatage invalide qui semble survenir pendant une transition vers l'avant est affecté au décalage UTC qui prévaut dans le fuseau horaire juste avant la transition alors qu'un horodatage ambigu qui semble survenir pendant une transition vers l'arrière se voit affecté le décalage UTC qui prévaut juste après la transition. Dans la plupart des fuseaux horaires, ceci est équivalent à dire que « l'interprétation du temps standard est préféré lorsqu'il y a un doute ».

Dans tous les cas, le décalage UTC associé à un horodatage peut être spécifié explicitement, en utilisant soit un décalage numérique UTC ou une abréviation de fuseau horaire correspondant au décalage UTC fixé. La règle donnée s'applique seulement si nécessaire pour convertir un décalage UTC pour un fuseau horaire pour lequel le décalage varie.

B.3. Mots-clés Date/Heure

Tableau B.1 présente les lexèmes reconnus comme des noms de mois.

Tableau B.1. Noms de mois

Mois	Abréviations
January (Janvier)	Jan
February (Février)	Feb
March (Mars)	Mar
April (Avril)	Apr
May (Mai)	
June (Juin)	Jun
July (Juillet)	Jul
August (Août)	Aug
September (Septembre)	Sep, Sept
October (Octobre)	Oct
November (Novembre)	Nov
December (Décembre)	Dec

Tableau B.2 présente les lexèmes reconnus comme des noms de jours de la semaine.

Tableau B.2. Noms des jours de la semaine

Jour	Abréviation
Sunday (Dimanche)	Sun
Monday (Lundi)	Mon
Tuesday (Mardi)	Tue, Tues
Wednesday (Mercredi)	Wed, Weds
Thursday (Jeudi)	Thu, Thur, Thurs
Friday (Vendredi)	Fri

Jour	Abréviation
Saturday (Samedi)	Sat

Tableau B.3 présente les lexèmes utilisés par divers modificateurs.

Tableau B.3. Modificateurs de Champs Date/Heure

Identifiant	Description
`AM`	L'heure précède 12:00
`AT`	Ignoré
`JULIAN`, `JD`, `J`	Le champ suivant est une date du calendrier Julien
`ON`	Ignoré
`PM`	L'heure suit 12:00
`T`	Le champ suivant est une heure

B.4. Fichiers de configuration date/heure

Comme il n'existe pas de réel standard des abréviations de fuseaux horaire, PostgreSQL permet de personnaliser l'ensemble des abréviations acceptées par le serveur. Le paramètre d'exécution timezone_abbreviations détermine l'ensemble des abréviations actives. Bien que tout utilisateur de la base puisse modifier ce paramètre, les valeurs possibles sont sous le contrôle de l'administrateur de bases de données -- ce sont en fait les noms des fichiers de configuration stockés dans `.../share/timezonesets/` du répertoire d'installation. En ajoutant ou en modifiant les fichiers de ce répertoire, l'administrateur peut définir les règles d'abréviation des fuseaux horaires.

`timezone_abbreviations` peut prendre tout nom de fichier situé dans `.../share/timezonesets/`, sous réserve que ce nom soit purement alphabétique. (L'interdiction de caractères non alphabétique dans `timezone_abbreviations` empêche la lecture de fichiers en dehors du répertoire prévu et celle de fichiers de sauvegarde ou autre.)

Un fichier d'abréviation de zones horaires peut contenir des lignes blanches et des commentaires (commençant avec un #). Les autres lignes doivent suivre l'un des formats suivants :

```
abréviation_fuseau_horaire décalage
abréviation_fuseau_horaire décalage D
abréviation_fuseau_horaire nom_fuseau_horaire
@INCLUDE nom_fichier
@OVERRIDE
```

Un *`abréviation_fuseau_horaire`* n'est que l'abréviation définie. Le *`décalage`* est un entier donnant le décalage en secondes à partir d'UTC, une valeur positive signifiant à l'est de Greenwich, une valeur négative à l'ouest. Ainsi, -18000 représente cinq heures à l'ouest de Greenwich, soit l'heure standard de la côte ouest nord américaine. `D` indique que le nom du fuseau représente une heure soumise à des règles de changement d'heure plutôt que l'heure standard.

Autrement, un *`nom_fuseau_horaire`* peut être indiqué, référençant un nom de fuseau horaire défini dans la base de données IANA. La définition du fuseau est consultée pour voir si l'abréviation est ou était utilisée pour ce fuseau et, si c'est bien le cas, la signification appropriée est utilisée -- la signification qui était utilisée pour l'horodatage dont la valeur était en cours de détermination ou la signification utilisée immédiatement avant ça si elle n'était pas actuelle à ce moment, ou la signification la plus ancienne si elle était utilisée seulement après ce moment. Ce comportement est essentiel pour gérer les abréviations dont la signification a varié dans l'histoire. Il est aussi permis de définir une

abréviation en terme de nom de fuseau horaire pour lequel cette abréviation n'apparaît pas ; alors utiliser l'abréviation est équivalent à écrire le nom du fuseau horaire.

> **Astuce**
>
> Utiliser un entier simple pour le *`décalage`* est préféré lors de la définition d'une abréviation dont le décalage à partir d'UTC n'a jamais changé, car ce type d'abréviation est bien moins coûteuse à traiter que celles qui réclament de consulter la définition du fuseau horaire.

La syntaxe `@INCLUDE` autorise l'inclusion d'autres fichiers du répertoire `.../share/timezonesets/`. Les inclusions peuvent être imbriquées jusqu'à une certaine profondeur.

La syntaxe `@OVERRIDE` indique que les entrées suivantes du fichier peuvent surcharger les entrées précédentes (c'est-à-dire des entrées obtenues à partir de fichiers inclus). Sans cela, les définitions en conflit au sein d'une même abréviation lèvent une erreur.

Dans une installation non modifiée, le fichier `Default` contient toutes les abréviations de fuseaux horaire, sans conflit, pour la quasi-totalité du monde. Les fichiers supplémentaires `Australia` et `India` sont fournis pour ces régions : ces fichiers incluent le fichier `Default` puis ajoutent ou modifient les fuseaux horaires si nécessaire.

Pour des raisons de référence, une installation standard contient aussi des fichiers `Africa.txt`, `America.txt`, etc. qui contiennent des informations sur les abréviations connues et utilisées en accord avec la base de données de fuseaux horaires IANA. Les définitions des noms de zone trouvées dans ces fichiers peuvent être copiées et collées dans un fichier de configuration personnalisé si nécessaire. Il ne peut pas être fait directement référence à ces fichiers dans le paramètre `timezone_abbreviations` à cause du point dans leur nom.

> **Note**
>
> Si une erreur survient lors de la lecture des abréviations de fuseaux horaires, aucune nouvelle valeur n'est acceptée mais les anciennes sont conservées. Si l'erreur survient au démarrage de la base, celui-ci échoue.

> **Attention**
>
> Les abréviations de fuseau horaire définies dans le fichier de configuration surchargent les informations sans fuseau définies nativement dans PostgreSQL. Par exemple, le fichier de configuration `Australia` définit `SAT` (*South Australian Standard Time*, soit l'heure standard pour l'Australie du sud). Si ce fichier est actif, `SAT` n'est plus reconnu comme abréviation de samedi (*Saturday*).

> **Attention**
>
> Si les fichiers de `.../share/timezonesets/` sont modifiés, il revient à l'utilisateur de procéder à leur sauvegarde -- une sauvegarde normale de base n'inclut pas ce répertoire.

B.5. Spécification POSIX des fuseaux horaires

PostgreSQL accepte les fuseaux horaires écrits suivant les règles du standard POSIX pour la variable d'environnement TZ. Les spécifications de fuseau horaire POSIX sont inadéquates pour gérer la complexité des fuseaux horaires du monde, mais il existe parfois des raisons pour les utiliser.

Une spécification POSIX de fuseau horaire a la forme suivante :

```
STD decalage [ DST [ decalage_dst ] [ , regle ] ]
```

(Pour des raisons de lisibilité, nous affichons des espaces entre les champs mais les espaces ne doivent pas être utilisés.) Les champs correspondent à :

- *STD* est l'abréviation de fuseau horaire à utiliser.
- *decalage* est le décalage de l'heure standard par rapport à UTC.
- *DST* est l'abréviation de fuseau horaire à utiliser pour les changements d'heure. Si ce champ et les suivants sont soumis, le fuseau d'horaire utilise un décalage UTC fixé sans règle de changement d'heure.
- *dstoffset* est le décalage du changement d'heure à partir d'UTC. Ce champ est typiquement omis parce qu'il vaut par défaut une heure de moins que *decalage* par rapport à l'heure standard, ce qui est généralement la bonne valeur.
- *rule* définit la règle pour quand le changement d'heure est en effet, comme décrit ci-dessous.

Dans cette syntaxe, une abréviation de fuseau horaire peut être une chaîne de lettres, tel que EST, ou une chaîne arbitraire entourée par des crochets, tel que <UTC-05>. Notez que les abréviations de fuseau horaire sont seulement utilisées pour l'affichage, et même seulement pour certains formats de sortie. Les abréviations de fuseaux horaires reconnus dans une entrée d'un champ de type timestamp sont expliquées dans Section B.4.

Les champs de décalage spécifient les heures et, en option, les minutes et secondes, de différence par rapport à UTC. Ils ont comme format *hh*[:*mm*[:*ss*]] avec un option un signe au début (+ ou -). Le signe positif est utilisé pour les fuseaux horaires à l'*ouest* de Greenwich. (Notez que c'est l'inverse de la convention prise par l'ISO-8601 utilisée ailleurs dans PostgreSQL.) *hh* peut avoir un ou deux chiffres ; *mm* et *ss* (s'ils sont utilisées) doivent en avoir deux.

La *règle* de transition de changement d'heure doit avoir le format

```
dstdate [ / dsttime ] , stddate [ / stdtime ]
```

(Comme précédemment, les espaces ne doivent pas être inclus en pratique.) Les champs *dstdate* et *dsttime* définissent quand le changement d'heure commence alors que *stddate* et *stdtime* définissent quand l'heure standard commence. (Dans certains cas, notamment dans les régions au sud de l'équateur, le premier peut être plus tard dans l'année que le deuxième.) Les champs date doivent avoir un des formats suivants :

n

Un entier dénote un jour de l'année, en comptant à partir de zéro et jusqu'à 364 ou 365 (ce dernier pour les années bissextiles).

J*n*

Dans ce format, *n* va de 1 à 365, et le 29 février n'est pas compté même dans le cas d'une année bissextile. (Donc, une transition survenant le 29 février ne peut pas être décrite de cette façon.

Néanmoins, les jours après février ont le même numéro qu'il s'agisse d'une année bissextile ou pas, donc cette forme est généralement plus utile que la forme entière standard pour les transitions sur des dates fixées.)

`M`*`m`*`.`*`n`*`.`*`d`*

Ce format spécifie une transition qui se produit toujours au cours du même mois et le même jour de la semaine. *`m`* identifie le mois, de 1 à 12. *`n`* précise la *`n`*ième occurrence du jour de la semaine identifié par *`d`*. *`n`* est un nombre entre 1 et 4, ou 5 signifiant la dernière occurrence de ce jour de la semaine dans le mois (qui peut être le 4ème ou 5ème). *`d`* est un nombre entre 0 et 6, avec 0 indiquant dimanche. Par exemple, `M3.2.0` signifie « le deuxième dimanche de mars ».

> **Note**
>
> Le format `M` est suffisant pour décrire les lois de transition de changement d'heure les plus communes. Mais notez qu'aucune de ces variantes ne peut gérer les changements d'heure, donc en pratique, les données historiques stockées pour les fuseaux horaires nommés (dans la base de données IANA des fuseaux horaires) est nécessaire pour interpréter correctement les anciennes dates et heures.

Les champs heure dans une règle de transition ont le même format que les champs de décalage décrits précédemment, sauf qu'elles ne peuvent pas contenir de signes. Ils définissent l'heure locale actuelle à laquelle le changement survient. En cas d'omission, la valeur par défaut est `02:00:00`.

Si une abréviation de changement d'heure est donné par que le champ *`rule`* de la transition est omis, le comportement de remplacement est d'utiliser la règle `M3.2.0,M11.1.0`, qui correspond à la pratique des États-Unis de 2020 (c'est-à-dire practice as of 2020 (that is, avancer au deuxième dimanche de Mars, retour au premier dimanche de novembre, les deux transitions se produisant à 2 heures du matin, heure courante). Notez que cette règle ne donne pas les bonnes dates de transition pour les États-Unis pour les années antérieures à 2007.

Comme exemple, `CET-1CEST,M3.5.0,M10.5.0/3` décrit la pratique de changement d'heure actuel (en 2020) à Paris. Cette spécification indique que l'heure standard a l'abréviation `CET` et est une heure avant (est) UTC ; le changement d'heure a pour abréviation `CEST` et est implicitement deux heures avant TC ; le DST commence le dernier dimanche de mars à 2 heures du matin, fuseau CET, et termine le dernier dimanche d'octobre à 3 heures du matin, fuseau CEST.

Les quatre noms de fuseau horaire `EST5EDT`, `CST6CDT`, `MST7MDT` et `PST8PDT` ressemblent beaucoup à des spécifications POSIX de fuseaux. Néanmoins, ils sont en fait traités comme des fuseaux horaires nommés parce que, pour des raisons historiques, il existe des fichiers avec ces noms dans la base de données IANA des fuseaux horaires. L'implication réelle de ceci est que ces noms de fuseaux horaires produiront des transactions valides historiques pour les changements d'heure, même quand une spécification POSIX pure ne le ferait pas.

Il est nécessaire de faire attention au fait qu'il est facile de mal orthographier une spécification POSIX de fuseau horaire car il n'y a pas de vérification sur le côté raisonnable des abréviations. Par exemple, `SET TIMEZONE TO FOOBAR0` fonctionnera, laissant le système utiliser réellement une abréviation spéciale pour UTC.

B.6. Histoire des unités

Le standard SQL précise que à l'intérieur de la définition d'un « litéral datetime », les « valeurs datetime » sont contraintes par les règles naturelles des dates et heures suivant le calendrier Grégorien. PostgreSQL suit le standard SQL en comptant les dates exclusivement dans le calendrier Grégorien, même pour les années datant d'avant l'apparition de ce calendrier. La règle est connue sous le nom (anglais) de *proleptic Gregorian calendar*.

Le calendrier Julien a été introduit par Julius Caesar en -45. Il était couramment utilisé dans le monde occidental jusqu'en l'an 1582, date à laquelle des pays ont commencé à se convertir au calendrier Grégorien. Dans le calendrier Julien, l'année tropicale est arrondie à 365 jours 1/4, soit 365,25 jours. Cela conduit à une erreur de l'ordre d'un jour tous les 128 ans.

L'erreur grandissante du calendrier poussa le Pape Grégoire XIII a réformé le calendrier en accord avec les instructions du Concile de Trent. Dans le calendrier Grégorien, l'année tropicale est arrondie à 365 + 97/400 jours, soit 365,2425 jours. Il faut donc à peu près 3300 ans pour que l'année tropicale subissent un décalage d'un an dans le calendrier Grégorien.

L'arrondi 365+97/400 est obtenu à l'aide de 97 années bissextiles tous les 400 ans. Les règles suivantes sont utilisées :

toute année divisible par 4 est bissextile ;
cependant, toute année divisible par 100 n'est pas bissextile ;
cependant, toute années divisible par 400 est bissextile.

1700, 1800, 1900, 2100 et 2200 ne sont donc pas des années bissextiles. 1600, 2000 et 2400 si. Par opposition, dans l'ancien calendrier Julien, toutes les années divisibles par 4 sont bissextiles.

En février 1582, le pape décrèta que 10 jours devaient être supprimés du mois d'octobre 1582, le 15 octobre devant ainsi arriver après le 4 octobre. Cela a été appliqué en Italie, Pologne, Portugal et Espagne. Les autres pays catholiques ont suivi peu après, mais les pays protestants ont été plus rétifs et les contrées orthodoxes grèques n'ont pas effectué le changement avant le début du 20ème siècle. La réforme a été appliquée par la Grande Bretagne et ses colonies (y compris les actuels Etats-Unis) en 1752. Donc le 2 septembre 1752 a été suivi du 14 septembre 1752. C'est pour cela que la commande `cal` produit la sortie suivante :

```
$ cal 9 1752
   septembre 1752
di lu ma me je ve sa
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Bien sûr, ce calendrier est seulement valide pour la Grande-Bretagne et ses colonies. Comme il serait difficile d'essayer de tracer les calendriers réels utilisés dans les différents endroits géographiques à différentes époques, PostgreSQL n'essaie pas de le faire, et suit les règles du calendrier Grégorien pour toutes les dates, même si cette méthode n'est pas vraie historiquement.

Divers calendriers ont été développés dans différentes parties du monde, la plupart précède le système Grégorien. Par exemple, les débuts du calendrier chinois peuvent être évalués aux alentours du 14ème siècle avant J.-C. La légende veut que l'empereur Huangdi inventa le calendrier en 2637 avant J.-C. La République de Chine utilise le calendrier Grégorien pour les besoins civils. Le calendrier chinois est utilisé pour déterminer les festivals.

B.7. Dates Julien

Le système de *dates Julien* est une méthode pour numéroter les jours. Il n'a pas de relation avec le calendrier Julien, malgré la similarité du nom. Le système de date Julien a été inventé par le précepteur français Joseph Justus Scaliger (1540–1609) et tient probablement son nom du père de Scaliger, le précepteur italien Julius Caesar Scaliger (1484–1558).

Dans le système de date Julien, chaque jour est un nombre séquentiel, commençant à partir de JD 0, appelé quelque fois *la* date Julien. JD 0 correspond au 1er janvier 4713 avant JC dans le calendrier Julien, ou au 24 novembre 4714 avant JC dans le calendrier grégorien. Le comptage de la date Julien est le plus souvent utilisé par les astronomes pour donner un nom à leurs observations, et du coup une date part de midi UTC jusqu'au prochain midi UTC, plutôt que de minuit à minuit : JD 0 désigne les 24 heures de midi UTC le 24 novembre 4714 avant JC au 25 novembre 4714 avant JC à minuit.

Bien que PostgreSQL accepte la saisie et l'affichage des dates en notation de date Julien (et les utilise aussi pour quelques calculs internes de date et heure), il n'utilise pas le compte des dates de midi à midi. PostgreSQL traite une date Julien comme allant de minuit heure locale à minuit heure locale, de la même façon que pour une date normale.

Néanmoins, cette définition fournit une méthode pour obtenir la définition astronomique quand vous en avez besoin : faites le calcul dans le fuseau horaire `UTC+12`. Par exemple,

```
=> SELECT extract(julian from '2021-06-23 7:00:00-04'::timestamptz
 at time zone 'UTC+12');
          extract
------------------------------
 2459388.9583333333333333333
(1 row)
=> SELECT extract(julian from '2021-06-23 8:00:00-04'::timestamptz
 at time zone 'UTC+12');
              extract
-------------------------------------
 2459389.0000000000000000000000000000
(1 row)
=> SELECT extract(julian from date '2021-06-23');
 extract
---------
 2459389
(1 row)
```

Annexe C. Mots-clés SQL

La Tableau C.1 liste tous les éléments qui sont des mots-clés dans le standard SQL et dans PostgreSQL 15.19. Des informations sous-jacentes peuvent être trouvées dans Section 4.1.1. (Par souci d'économie d'espace, seules les deux dernières versions du standard SQL, et de SQL-92 par comparaison, sont incluses. Les différences entre ces deux versions et les versions intermédiaires du standard SQL sont minimes.)

SQL distingue les mots-clés *réservés* et *non réservés*. Selon le standard, les mots-clés réservés sont réellement les seuls mots-clés ; ils ne sont jamais autorisés comme identifiants. Les mots-clés non réservés ont seulement un sens spécial dans certains contextes et peuvent être utilisés comme identifiants dans d'autres contextes. La plupart des mots-clés non réservés sont en fait les noms des tables et des fonctions prédéfinies spécifiés par SQL. Le concept de mots-clés non réservés existe seulement pour indiquer que certains sens prédéfinis sont attachés à un mot dans certains contextes.

Dans l'analyseur de PostgreSQL, la vie est un peu plus compliquée. Il y a différentes classes d'éléments allant de ceux que l'on ne peut jamais utiliser comme identifiants à ceux qui n'ont absolument aucun statut spécial dans l'analyseur, mais sont considérés comme des identifiants ordinaires (c'est généralement le cas pour les fonctions spécifiées par SQL). Même les mots-clés réservés ne sont pas complètement réservés dans PostgreSQL et peuvent être utilisés comme noms des colonnes (par exemple, `SELECT 55 AS CHECK`, même si `CHECK` est un mot-clé).

Dans Tableau C.1, dans la colonne pour PostgreSQL, nous classons comme « non réservé » les mots-clés qui sont explicitement connus par l'analyseur mais qui sont autorisés en tant que noms de colonnes ou de tables. Certains mots-clés qui sont non réservés et qui ne peuvent pas être utilisés comme un nom de fonction ou un type de données sont marqués en conséquence. (La plupart des mots représentent des fonctions prédéfinies ou des types de données avec une syntaxe spéciale. La fonction ou le type est toujours disponible mais il ne peut pas être redéfini par un utilisateur.) Les « réservés » sont des éléments qui ne sont pas autorisés en tant que noms de colonne ou de table. Certains mots-clés réservés sont autorisés comme noms pour les fonctions et les types de données ; cela est également montré dans le tableau. Dans le cas contraire, un mot clé réservé est seulement autorisé dans un label de colonne. Une entrée vide dans cette colonne signifie que le mot est traité comme un identifiant ordinaire par PostgreSQL.

De plus, alors que la plupart des mots clés peut être utilisée comme labels de colonne « bruts » sans écrire `AS` avant eux (comme décrit dans Section 7.3.2), il en existe quelques-uns qui nécessitent un `AS` avant pour éviter toute ambiguité. Ils sont marqués dans la table comme « nécessite `AS` ».

En règle générale, si vous avez des erreurs de la part de l'analyseur pour des commandes qui utilisent un des mots-clés listés comme identifiants, vous devriez essayer de mettre entre guillemets l'identifiant pour voir si le problème disparait.

Il est important de comprendre avant d'étudier la Tableau C.1 que le fait qu'un mot-clé ne soit pas réservé dans PostgreSQL ne signifie pas que la fonctionnalité en rapport avec ce mot n'est pas implémentée. Réciproquement, la présence d'un mot-clé n'indique pas l'existence d'une fonctionnalité.

Tableau C.1. Mots clés SQL

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
A		non réservé	non réservé	
ABORT	non réservé			
ABS		réservé	réservé	
ABSENT	non réservé	réservé	non réservé	
ABSOLUTE	non réservé	non réservé	non réservé	réservé
ACCESS	non réservé			
ACCORDING		non réservé	non réservé	

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
ACOS		réservé		
ACTION	non réservé	non réservé	non réservé	réservé
ADA		non réservé	non réservé	non réservé
ADD	non réservé	non réservé	non réservé	réservé
ADMIN	non réservé	non réservé	non réservé	
AFTER	non réservé	non réservé	non réservé	
AGGREGATE	non réservé			
ALL	réservé	réservé	réservé	réservé
ALLOCATE		réservé	réservé	réservé
ALSO	non réservé			
ALTER	non réservé	réservé	réservé	réservé
ALWAYS	non réservé	non réservé	non réservé	
ANALYSE	réservé			
ANALYZE	réservé			
AND	réservé	réservé	réservé	réservé
ANY	réservé	réservé	réservé	réservé
ARE		réservé	réservé	réservé
ARRAY	réservé, requiert AS	réservé	réservé	
ARRAY_AGG		réservé	réservé	
ARRAY_MAX_CARDINALITY		réservé	réservé	
AS	réservé, requiert AS	réservé	réservé	réservé
ASC	réservé	non réservé	non réservé	réservé
ASENSITIVE	non réservé	réservé	réservé	
ASIN		réservé		
ASSERTION	non réservé	non réservé	non réservé	réservé
ASSIGNMENT	non réservé	non réservé	non réservé	
ASYMMETRIC	réservé	réservé	réservé	
AT	non réservé	réservé	réservé	réservé
ATAN		réservé		
ATOMIC	non réservé	réservé	réservé	
ATTACH	non réservé			
ATTRIBUTE	non réservé	non réservé	non réservé	
ATTRIBUTES		non réservé	non réservé	
AUTHORIZATION	réservé (peut être une fonction ou un type)	réservé	réservé	réservé
AVG		réservé	réservé	réservé
BACKWARD	non réservé			
BASE64		non réservé	non réservé	
BEFORE	non réservé	non réservé	non réservé	
BEGIN	non réservé	réservé	réservé	réservé

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
BEGIN_FRAME		réservé	réservé	
BEGIN_PARTITION		réservé	réservé	
BERNOULLI		non réservé	non réservé	
BETWEEN	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
BIGINT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
BINARY	réservé (peut être une fonction ou un type)	réservé	réservé	
BIT	non réservé (ne peut pas être une fonction ou un type)			réservé
BIT_LENGTH				réservé
BLOB		réservé	réservé	
BLOCKED		non réservé	non réservé	
BOM		non réservé	non réservé	
BOOLEAN	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
BOTH	réservé	réservé	réservé	réservé
BREADTH	non réservé	non réservé	non réservé	
BY	non réservé	réservé	réservé	réservé
C		non réservé	non réservé	non réservé
CACHE	non réservé			
CALL	non réservé	réservé	réservé	
CALLED	non réservé	réservé	réservé	
CARDINALITY		réservé	réservé	
CASCADE	non réservé	non réservé	non réservé	réservé
CASCADED	non réservé	réservé	réservé	réservé
CASE	réservé	réservé	réservé	réservé
CAST	réservé	réservé	réservé	réservé
CATALOG	non réservé	non réservé	non réservé	réservé
CATALOG_NAME		non réservé	non réservé	non réservé
CEIL		réservé	réservé	
CEILING		réservé	réservé	
CHAIN	non réservé	non réservé	non réservé	
CHAINING		non réservé		
CHAR	non réservé (ne peut pas être une fonction ou un type), requiert AS	réservé	réservé	réservé

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
CHARACTER	non réservé (ne peut pas être une fonction ou un type), requiert AS	réservé	réservé	réservé
CHARACTERISTICS	non réservé	non réservé	non réservé	
CHARACTERS		non réservé	non réservé	
CHARACTER_LENGTH		réservé	réservé	réservé
CHARACTER_SET_CATALOG		non réservé	non réservé	non réservé
CHARACTER_SET_NAME		non réservé	non réservé	non réservé
CHARACTER_SET_SCHEMA		non réservé	non réservé	non réservé
CHAR_LENGTH		réservé	réservé	réservé
CHECK	réservé	réservé	réservé	réservé
CHECKPOINT	non réservé			
CLASS	non réservé			
CLASSIFIER		réservé		
CLASS_ORIGIN		non réservé	non réservé	non réservé
CLOB		réservé	réservé	
CLOSE	non réservé	réservé	réservé	réservé
CLUSTER	non réservé			
COALESCE	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
COBOL		non réservé	non réservé	non réservé
COLLATE	réservé	réservé	réservé	réservé
COLLATION	réservé (peut être une fonction ou un type)	non réservé	non réservé	réservé
COLLATION_CATALOG		non réservé	non réservé	non réservé
COLLATION_NAME		non réservé	non réservé	non réservé
COLLATION_SCHEMA		non réservé	non réservé	non réservé
COLLECT		réservé	réservé	
COLUMN	réservé	réservé	réservé	réservé
COLUMNS	non réservé	non réservé	non réservé	
COLUMN_NAME		non réservé	non réservé	non réservé
COMMAND_FUNCTION		non réservé	non réservé	non réservé
COMMAND_FUNCTION_CODE		non réservé	non réservé	
COMMENT	non réservé			
COMMENTS	non réservé			
COMMIT	non réservé	réservé	réservé	réservé
COMMITTED	non réservé	non réservé	non réservé	non réservé
COMPRESSION	non réservé			

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
CONCURRENTLY	réservé (peut être une fonction ou un type)			
CONDITION		réservé	réservé	
CONDITIONAL	non réservé	non réservé		
CONDITION_NUMBER		non réservé	non réservé	non réservé
CONFIGURATION	non réservé			
CONFLICT	non réservé			
CONNECT		réservé	réservé	réservé
CONNECTION	non réservé	non réservé	non réservé	réservé
CONNECTION_NAME		non réservé	non réservé	non réservé
CONSTRAINT	réservé	réservé	réservé	réservé
CONSTRAINTS	non réservé	non réservé	non réservé	réservé
CONSTRAINT_CATALOG		non réservé	non réservé	non réservé
CONSTRAINT_NAME		non réservé	non réservé	non réservé
CONSTRAINT_SCHEMA		non réservé	non réservé	non réservé
CONSTRUCTOR		non réservé	non réservé	
CONTAINS		réservé	réservé	
CONTENT	non réservé	non réservé	non réservé	
CONTINUE	non réservé	non réservé	non réservé	réservé
CONTROL		non réservé	non réservé	
CONVERSION	non réservé			
CONVERT		réservé	réservé	réservé
COPY	non réservé	réservé		
CORR		réservé	réservé	
CORRESPONDING		réservé	réservé	réservé
COS		réservé		
COSH		réservé		
COST	non réservé			
COUNT		réservé	réservé	réservé
COVAR_POP		réservé	réservé	
COVAR_SAMP		réservé	réservé	
CREATE	réservé, requiert AS	réservé	réservé	réservé
CROSS	réservé (peut être une fonction ou un type)	réservé	réservé	réservé
CSV	non réservé			
CUBE	non réservé	réservé	réservé	
CUME_DIST		réservé	réservé	
CURRENT	non réservé	réservé	réservé	réservé
CURRENT_CATALOG	réservé	réservé	réservé	
CURRENT_DATE	réservé	réservé	réservé	réservé

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
CURRENT_DEFAULT_TRANSFORM_GROUP		réservé	réservé	
CURRENT_PATH		réservé	réservé	
CURRENT_ROLE	réservé	réservé	réservé	
CURRENT_ROW		réservé	réservé	
CURRENT_SCHEMA	réservé (peut être une fonction ou un type)	réservé	réservé	
CURRENT_TIME	réservé	réservé	réservé	réservé
CURRENT_TIMESTAMP	réservé	réservé	réservé	réservé
CURRENT_TRANSFORM_GROUP_FOR_TYPE		réservé	réservé	
CURRENT_USER	réservé	réservé	réservé	réservé
CURSOR	non réservé	réservé	réservé	réservé
CURSOR_NAME		non réservé	non réservé	non réservé
CYCLE	non réservé	réservé	réservé	
DATA	non réservé	non réservé	non réservé	non réservé
DATABASE	non réservé			
DATALINK		réservé	réservé	
DATE		réservé	réservé	réservé
DATETIME_INTERVAL_CODE		non réservé	non réservé	non réservé
DATETIME_INTERVAL_PRECISION		non réservé	non réservé	non réservé
DAY	non réservé, requiert AS	réservé	réservé	réservé
DB		non réservé	non réservé	
DEALLOCATE	non réservé	réservé	réservé	réservé
DEC	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
DECFLOAT		réservé		
DECIMAL	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
DECLARE	non réservé	réservé	réservé	réservé
DEFAULT	réservé	réservé	réservé	réservé
DEFAULTS	non réservé	non réservé	non réservé	
DEFERRABLE	réservé	non réservé	non réservé	réservé
DEFERRED	non réservé	non réservé	non réservé	réservé
DEFINE		réservé		
DEFINED		non réservé	non réservé	
DEFINER	non réservé	non réservé	non réservé	
DEGREE		non réservé	non réservé	
DELETE	non réservé	réservé	réservé	réservé

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
DELIMITER	non réservé			
DELIMITERS	non réservé			
DENSE_RANK		réservé	réservé	
DEPENDS	non réservé			
DEPTH	non réservé	non réservé	non réservé	
DEREF		réservé	réservé	
DERIVED		non réservé	non réservé	
DESC	réservé	non réservé	non réservé	réservé
DESCRIBE		réservé	réservé	réservé
DESCRIPTOR		non réservé	non réservé	réservé
DETACH	non réservé			
DETERMINISTIC		réservé	réservé	
DIAGNOSTICS		non réservé	non réservé	réservé
DICTIONARY	non réservé			
DISABLE	non réservé			
DISCARD	non réservé			
DISCONNECT		réservé	réservé	réservé
DISPATCH		non réservé	non réservé	
DISTINCT	réservé	réservé	réservé	réservé
DLNEWCOPY		réservé	réservé	
DLPREVIOUSCOPY		réservé	réservé	
DLURLCOMPLETE		réservé	réservé	
DLURLCOMPLETEONLY		réservé	réservé	
DLURLCOMPLETEWRITE		réservé	réservé	
DLURLPATH		réservé	réservé	
DLURLPATHONLY		réservé	réservé	
DLURLPATHWRITE		réservé	réservé	
DLURLSCHEME		réservé	réservé	
DLURLSERVER		réservé	réservé	
DLVALUE		réservé	réservé	
DO	réservé			
DOCUMENT	non réservé	non réservé	non réservé	
DOMAIN	non réservé	non réservé	non réservé	réservé
DOUBLE	non réservé	réservé	réservé	réservé
DROP	non réservé	réservé	réservé	réservé
DYNAMIC		réservé	réservé	
DYNAMIC_FUNCTION		non réservé	non réservé	non réservé
DYNAMIC_FUNCTION_CODE		non réservé	non réservé	
EACH	non réservé	réservé	réservé	
ELEMENT		réservé	réservé	
ELSE	réservé	réservé	réservé	réservé

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
EMPTY	non réservé	réservé	non réservé	
ENABLE	non réservé			
ENCODING	non réservé	non réservé	non réservé	
ENCRYPTED	non réservé			
END	réservé	réservé	réservé	réservé
END-EXEC		réservé	réservé	réservé
END_FRAME		réservé	réservé	
END_PARTITION		réservé	réservé	
ENFORCED		non réservé	non réservé	
ENUM	non réservé			
EQUALS		réservé	réservé	
ERROR	non réservé	non réservé		
ESCAPE	non réservé	réservé	réservé	réservé
EVENT	non réservé			
EVERY		réservé	réservé	
EXCEPT	réservé, requiert AS	réservé	réservé	réservé
EXCEPTION				réservé
EXCLUDE	non réservé	non réservé	non réservé	
EXCLUDING	non réservé	non réservé	non réservé	
EXCLUSIVE	non réservé			
EXEC		réservé	réservé	réservé
EXECUTE	non réservé	réservé	réservé	réservé
EXISTS	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
EXP		réservé	réservé	
EXPLAIN	non réservé			
EXPRESSION	non réservé	non réservé	non réservé	
EXTENSION	non réservé			
EXTERNAL	non réservé	réservé	réservé	réservé
EXTRACT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
FALSE	réservé	réservé	réservé	réservé
FAMILY	non réservé			
FETCH	réservé, requiert AS	réservé	réservé	réservé
FILE		non réservé	non réservé	
FILTER	non réservé, requiert AS	réservé	réservé	
FINAL		non réservé	non réservé	
FINALIZE	non réservé			
FINISH		non réservé		

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
FIRST	non réservé	non réservé	non réservé	réservé
FIRST_VALUE		réservé	réservé	
FLAG		non réservé	non réservé	
FLOAT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
FLOOR		réservé	réservé	
FOLLOWING	non réservé	non réservé	non réservé	
FOR	réservé, requiert AS	réservé	réservé	réservé
FORCE	non réservé			
FOREIGN	réservé	réservé	réservé	réservé
FORMAT	non réservé	non réservé		
FORTRAN		non réservé	non réservé	non réservé
FORWARD	non réservé			
FOUND		non réservé	non réservé	réservé
FRAME_ROW		réservé	réservé	
FREE		réservé	réservé	
FREEZE	réservé (peut être une fonction ou un type)			
FROM	réservé, requiert AS	réservé	réservé	réservé
FS		non réservé	non réservé	
FULFILL		non réservé		
FULL	réservé (peut être une fonction ou un type)	réservé	réservé	réservé
FUNCTION	non réservé	réservé	réservé	
FUNCTIONS	non réservé			
FUSION		réservé	réservé	
G		non réservé	non réservé	
GENERAL		non réservé	non réservé	
GENERATED	non réservé	non réservé	non réservé	
GET		réservé	réservé	réservé
GLOBAL	non réservé	réservé	réservé	réservé
GO		non réservé	non réservé	réservé
GOTO		non réservé	non réservé	réservé
GRANT	réservé, requiert AS	réservé	réservé	réservé
GRANTED	non réservé	non réservé	non réservé	
GREATEST	non réservé (ne peut pas être une fonction ou un type)			
GROUP	réservé, requiert AS	réservé	réservé	réservé

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
GROUPING	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
GROUPS	non réservé	réservé	réservé	
HANDLER	non réservé			
HAVING	réservé, requiert AS	réservé	réservé	réservé
HEADER	non réservé			
HEX		non réservé	non réservé	
HIERARCHY		non réservé	non réservé	
HOLD	non réservé	réservé	réservé	
HOUR	non réservé, requiert AS	réservé	réservé	réservé
ID		non réservé	non réservé	
IDENTITY	non réservé	réservé	réservé	réservé
IF	non réservé			
IGNORE		non réservé	non réservé	
ILIKE	réservé (peut être une fonction ou un type)			
IMMEDIATE	non réservé	non réservé	non réservé	réservé
IMMEDIATELY		non réservé	non réservé	
IMMUTABLE	non réservé			
IMPLEMENTATION		non réservé	non réservé	
IMPLICIT	non réservé			
IMPORT	non réservé	réservé	réservé	
IN	réservé	réservé	réservé	réservé
INCLUDE	non réservé			
INCLUDING	non réservé	non réservé	non réservé	
INCREMENT	non réservé	non réservé	non réservé	
INDENT		non réservé	non réservé	
INDEX	non réservé			
INDEXES	non réservé			
INDICATOR		réservé	réservé	réservé
INHERIT	non réservé			
INHERITS	non réservé			
INITIAL		réservé		
INITIALLY	réservé	non réservé	non réservé	réservé
INLINE	non réservé			
INNER	réservé (peut être une fonction ou un type)	réservé	réservé	réservé

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
INOUT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
INPUT	non réservé	non réservé	non réservé	réservé
INSENSITIVE	non réservé	réservé	réservé	réservé
INSERT	non réservé	réservé	réservé	réservé
INSTANCE		non réservé	non réservé	
INSTANTIABLE		non réservé	non réservé	
INSTEAD	non réservé	non réservé	non réservé	
INT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
INTEGER	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
INTEGRITY		non réservé	non réservé	
INTERSECT	réservé, requiert AS	réservé	réservé	réservé
INTERSECTION		réservé	réservé	
INTERVAL	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
INTO	réservé, requiert AS	réservé	réservé	réservé
INVOKER	non réservé	non réservé	non réservé	
IS	réservé (peut être une fonction ou un type)	réservé	réservé	réservé
ISNULL	réservé (peut être une fonction ou un type), requiert AS			
ISOLATION	non réservé	non réservé	non réservé	réservé
JOIN	réservé (peut être une fonction ou un type)	réservé	réservé	réservé
JSON	non réservé (ne peut pas être une fonction ou un type)	réservé		
JSON_ARRAY	non réservé (ne peut pas être une fonction ou un type)	réservé		
JSON_ARRAYAGG	non réservé (ne peut pas être une fonction ou un type)	réservé		
JSON_EXISTS	non réservé (ne peut pas être une fonction ou un type)	réservé		

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
JSON_OBJECT	non réservé (ne peut pas être une fonction ou un type)	réservé		
JSON_OBJECTAGG	non réservé (ne peut pas être une fonction ou un type)	réservé		
JSON_QUERY	non réservé (ne peut pas être une fonction ou un type)	réservé		
JSON_SCALAR	non réservé (ne peut pas être une fonction ou un type)			
JSON_SERIALIZE	non réservé (ne peut pas être une fonction ou un type)			
JSON_TABLE	non réservé (ne peut pas être une fonction ou un type)	réservé		
JSON_TABLE_PRIMITIVE		réservé		
JSON_VALUE	non réservé (ne peut pas être une fonction ou un type)	réservé		
K		non réservé	non réservé	
KEEP	non réservé	non réservé		
KEY	non réservé	non réservé	non réservé	réservé
KEYS	non réservé	non réservé		
KEY_MEMBER		non réservé	non réservé	
KEY_TYPE		non réservé	non réservé	
LABEL	non réservé			
LAG		réservé	réservé	
LANGUAGE	non réservé	réservé	réservé	réservé
LARGE	non réservé	réservé	réservé	
LAST	non réservé	non réservé	non réservé	réservé
LAST_VALUE		réservé	réservé	
LATERAL	réservé	réservé	réservé	
LEAD		réservé	réservé	
LEADING	réservé	réservé	réservé	réservé
LEAKPROOF	non réservé			
LEAST	non réservé (ne peut pas être une fonction ou un type)			
LEFT	réservé (peut être une fonction ou un type)	réservé	réservé	réservé
LENGTH		non réservé	non réservé	non réservé
LEVEL	non réservé	non réservé	non réservé	réservé

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
LIBRARY		non réservé	non réservé	
LIKE	réservé (peut être une fonction ou un type)	réservé	réservé	réservé
LIKE_REGEX		réservé	réservé	
LIMIT	réservé, requiert AS	non réservé	non réservé	
LINK		non réservé	non réservé	
LISTAGG		réservé		
LISTEN	non réservé			
LN		réservé	réservé	
LOAD	non réservé			
LOCAL	non réservé	réservé	réservé	réservé
LOCALTIME	réservé	réservé	réservé	
LOCALTIMESTAMP	réservé	réservé	réservé	
LOCATION	non réservé	non réservé	non réservé	
LOCATOR		non réservé	non réservé	
LOCK	non réservé			
LOCKED	non réservé			
LOG		réservé		
LOG10		réservé		
LOGGED	non réservé			
LOWER		réservé	réservé	réservé
M		non réservé	non réservé	
MAP		non réservé	non réservé	
MAPPING	non réservé	non réservé	non réservé	
MATCH	non réservé	réservé	réservé	réservé
MATCHED	non réservé	non réservé	non réservé	
MATCHES		réservé		
MATCH_NUMBER		réservé		
MATCH_RECOGNIZE		réservé		
MATERIALIZED	non réservé			
MAX		réservé	réservé	réservé
MAXVALUE	non réservé	non réservé	non réservé	
MEASURES		réservé		
MEMBER		réservé	réservé	
MERGE	non réservé	réservé	réservé	
MESSAGE_LENGTH		non réservé	non réservé	non réservé
MESSAGE_OCTET_LENGTH		non réservé	non réservé	non réservé
MESSAGE_TEXT		non réservé	non réservé	non réservé
METHOD	non réservé	réservé	réservé	
MIN		réservé	réservé	réservé

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
MINUTE	non réservé, requiert AS	réservé	réservé	réservé
MINVALUE	non réservé	non réservé	non réservé	
MOD		réservé	réservé	
MODE	non réservé			
MODIFIES		réservé	réservé	
MODULE		réservé	réservé	réservé
MONTH	non réservé, requiert AS	réservé	réservé	réservé
MORE		non réservé	non réservé	non réservé
MOVE	non réservé			
MULTISET		réservé	réservé	
MUMPS		non réservé	non réservé	non réservé
NAME	non réservé	non réservé	non réservé	non réservé
NAMES	non réservé	non réservé	non réservé	réservé
NAMESPACE		non réservé	non réservé	
NATIONAL	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
NATURAL	réservé (peut être une fonction ou un type)	réservé	réservé	réservé
NCHAR	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
NCLOB		réservé	réservé	
NESTED	non réservé	non réservé		
NESTING		non réservé	non réservé	
NEW	non réservé	réservé	réservé	
NEXT	non réservé	non réservé	non réservé	réservé
NFC	non réservé	non réservé	non réservé	
NFD	non réservé	non réservé	non réservé	
NFKC	non réservé	non réservé	non réservé	
NFKD	non réservé	non réservé	non réservé	
NIL		non réservé	non réservé	
NO	non réservé	réservé	réservé	réservé
NONE	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
NORMALIZE	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
NORMALIZED	non réservé	non réservé	non réservé	
NOT	réservé	réservé	réservé	réservé

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
NOTHING	non réservé			
NOTIFY	non réservé			
NOTNULL	réservé (peut être une fonction ou un type), requiert AS			
NOWAIT	non réservé			
NTH_VALUE		réservé	réservé	
NTILE		réservé	réservé	
NULL	réservé	réservé	réservé	réservé
NULLABLE		non réservé	non réservé	non réservé
NULLIF	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
NULLS	non réservé	non réservé	non réservé	
NULL_ORDERING		non réservé		
NUMBER		non réservé	non réservé	non réservé
NUMERIC	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
OBJECT	non réservé	non réservé	non réservé	
OCCURRENCE		non réservé		
OCCURRENCES_REGEX		réservé	réservé	
OCTETS		non réservé	non réservé	
OCTET_LENGTH		réservé	réservé	réservé
OF	non réservé	réservé	réservé	réservé
OFF	non réservé	non réservé	non réservé	
OFFSET	réservé, requiert AS	réservé	réservé	
OIDS	non réservé			
OLD	non réservé	réservé	réservé	
OMIT	non réservé	réservé		
ON	réservé, requiert AS	réservé	réservé	réservé
ONE		réservé		
ONLY	réservé	réservé	réservé	réservé
OPEN		réservé	réservé	réservé
OPERATOR	non réservé			
OPTION	non réservé	non réservé	non réservé	réservé
OPTIONS	non réservé	non réservé	non réservé	
OR	réservé	réservé	réservé	réservé
ORDER	réservé, requiert AS	réservé	réservé	réservé
ORDERING		non réservé	non réservé	
ORDINALITY	non réservé	non réservé	non réservé	
OTHERS	non réservé	non réservé	non réservé	

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
OUT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
OUTER	réservé (peut être une fonction ou un type)	réservé	réservé	réservé
OUTPUT		non réservé	non réservé	réservé
OVER	non réservé, requiert AS	réservé	réservé	
OVERFLOW		non réservé		
OVERLAPS	réservé (peut être une fonction ou un type), requiert AS	réservé	réservé	réservé
OVERLAY	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
OVERRIDING	non réservé	non réservé	non réservé	
OWNED	non réservé			
OWNER	non réservé			
P		non réservé	non réservé	
PAD		non réservé	non réservé	réservé
PARALLEL	non réservé			
PARAMETER	non réservé	réservé	réservé	
PARAMETER_MODE		non réservé	non réservé	
PARAMETER_NAME		non réservé	non réservé	
PARAMETER_ORDINAL_POSITION		non réservé	non réservé	
PARAMETER_SPECIFIC_CATALOG		non réservé	non réservé	
PARAMETER_SPECIFIC_NAME		non réservé	non réservé	
PARAMETER_SPECIFIC_SCHEMA		non réservé	non réservé	
PARSER	non réservé			
PARTIAL	non réservé	non réservé	non réservé	réservé
PARTITION	non réservé	réservé	réservé	
PASCAL		non réservé	non réservé	non réservé
PASS		non réservé		
PASSING	non réservé	non réservé	non réservé	
PASSTHROUGH		non réservé	non réservé	
PASSWORD	non réservé			
PAST		non réservé		
PATH	non réservé	non réservé	non réservé	
PATTERN		réservé		
PER		réservé		

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
PERCENT		réservé	réservé	
PERCENTILE_CONT		réservé	réservé	
PERCENTILE_DISC		réservé	réservé	
PERCENT_RANK		réservé	réservé	
PERIOD		réservé	réservé	
PERMISSION		non réservé	non réservé	
PERMUTE		réservé		
PIPE		non réservé		
PLACING	réservé	non réservé	non réservé	
PLAN	non réservé	non réservé		
PLANS	non réservé			
PLI		non réservé	non réservé	non réservé
POLICY	non réservé			
PORTION		réservé	réservé	
POSITION	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
POSITION_REGEX		réservé	réservé	
POWER		réservé	réservé	
PRECEDES		réservé	réservé	
PRECEDING	non réservé	non réservé	non réservé	
PRECISION	non réservé (ne peut pas être une fonction ou un type), requiert AS	réservé	réservé	réservé
PREPARE	non réservé	réservé	réservé	réservé
PREPARED	non réservé			
PRESERVE	non réservé	non réservé	non réservé	réservé
PREV		non réservé		
PRIMARY	réservé	réservé	réservé	réservé
PRIOR	non réservé	non réservé	non réservé	réservé
PRIVATE		non réservé		
PRIVILEGES	non réservé	non réservé	non réservé	réservé
PROCEDURAL	non réservé			
PROCEDURE	non réservé	réservé	réservé	réservé
PROCEDURES	non réservé			
PROGRAM	non réservé			
PRUNE		non réservé		
PTF		réservé		
PUBLIC		non réservé	non réservé	réservé
PUBLICATION	non réservé			
QUOTE	non réservé			

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
QUOTES	non réservé	non réservé		
RANGE	non réservé	réservé	réservé	
RANK		réservé	réservé	
READ	non réservé	non réservé	non réservé	réservé
READS		réservé	réservé	
REAL	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
REASSIGN	non réservé			
RECHECK	non réservé			
RECOVERY		non réservé	non réservé	
RECURSIVE	non réservé	réservé	réservé	
REF	non réservé	réservé	réservé	
REFERENCES	réservé	réservé	réservé	réservé
REFERENCING	non réservé	réservé	réservé	
REFRESH	non réservé			
REGR_AVGX		réservé	réservé	
REGR_AVGY		réservé	réservé	
REGR_COUNT		réservé	réservé	
REGR_INTERCEPT		réservé	réservé	
REGR_R2		réservé	réservé	
REGR_SLOPE		réservé	réservé	
REGR_SXX		réservé	réservé	
REGR_SXY		réservé	réservé	
REGR_SYY		réservé	réservé	
REINDEX	non réservé			
RELATIVE	non réservé	non réservé	non réservé	réservé
RELEASE	non réservé	réservé	réservé	
RENAME	non réservé			
REPEATABLE	non réservé	non réservé	non réservé	non réservé
REPLACE	non réservé			
REPLICA	non réservé			
REQUIRING		non réservé	non réservé	
RESET	non réservé			
RESPECT		non réservé	non réservé	
RESTART	non réservé	non réservé	non réservé	
RESTORE		non réservé	non réservé	
RESTRICT	non réservé	non réservé	non réservé	réservé
RESULT		réservé	réservé	
RETURN	non réservé	réservé	réservé	
RETURNED_CARDINALITY		non réservé	non réservé	

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
RETURNED_LENGTH		non réservé	non réservé	non réservé
RETURNED_OCTET_LENGTH		non réservé	non réservé	non réservé
RETURNED_SQLSTATE		non réservé	non réservé	non réservé
RETURNING	réservé, requiert AS	non réservé	non réservé	
RETURNS	non réservé	réservé	réservé	
REVOKE	non réservé	réservé	réservé	réservé
RIGHT	réservé (peut être une fonction ou un type)	réservé	réservé	réservé
ROLE	non réservé	non réservé	non réservé	
ROLLBACK	non réservé	réservé	réservé	réservé
ROLLUP	non réservé	réservé	réservé	
ROUTINE	non réservé	non réservé	non réservé	
ROUTINES	non réservé			
ROUTINE_CATALOG		non réservé	non réservé	
ROUTINE_NAME		non réservé	non réservé	
ROUTINE_SCHEMA		non réservé	non réservé	
ROW	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
ROWS	non réservé	réservé	réservé	réservé
ROW_COUNT		non réservé	non réservé	non réservé
ROW_NUMBER		réservé	réservé	
RULE	non réservé			
RUNNING		réservé		
SAVEPOINT	non réservé	réservé	réservé	
SCALAR	non réservé	non réservé		
SCALE		non réservé	non réservé	non réservé
SCHEMA	non réservé	non réservé	non réservé	réservé
SCHEMAS	non réservé			
SCHEMA_NAME		non réservé	non réservé	non réservé
SCOPE		réservé	réservé	
SCOPE_CATALOG		non réservé	non réservé	
SCOPE_NAME		non réservé	non réservé	
SCOPE_SCHEMA		non réservé	non réservé	
SCROLL	non réservé	réservé	réservé	réservé
SEARCH	non réservé	réservé	réservé	
SECOND	non réservé, requiert AS	réservé	réservé	réservé
SECTION		non réservé	non réservé	réservé
SECURITY	non réservé	non réservé	non réservé	
SEEK		réservé		

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
SELECT	réservé	réservé	réservé	réservé
SELECTIVE		non réservé	non réservé	
SELF		non réservé	non réservé	
SEMANTICS		non réservé		
SENSITIVE		réservé	réservé	
SEQUENCE	non réservé	non réservé	non réservé	
SEQUENCES	non réservé			
SERIALIZABLE	non réservé	non réservé	non réservé	non réservé
SERVER	non réservé	non réservé	non réservé	
SERVER_NAME		non réservé	non réservé	non réservé
SESSION	non réservé	non réservé	non réservé	réservé
SESSION_USER	réservé	réservé	réservé	réservé
SET	non réservé	réservé	réservé	réservé
SETOF	non réservé (ne peut pas être une fonction ou un type)			
SETS	non réservé	non réservé	non réservé	
SHARE	non réservé			
SHOW	non réservé	réservé		
SIMILAR	réservé (peut être une fonction ou un type)	réservé	réservé	
SIMPLE	non réservé	non réservé	non réservé	
SIN		réservé		
SINH		réservé		
SIZE		non réservé	non réservé	réservé
SKIP	non réservé	réservé		
SMALLINT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
SNAPSHOT	non réservé			
SOME	réservé	réservé	réservé	réservé
SORT_DIRECTION		non réservé		
SOURCE		non réservé	non réservé	
SPACE		non réservé	non réservé	réservé
SPECIFIC		réservé	réservé	
SPECIFICTYPE		réservé	réservé	
SPECIFIC_NAME		non réservé	non réservé	
SQL	non réservé	réservé	réservé	réservé
SQLCODE				réservé
SQLERROR				réservé
SQLEXCEPTION		réservé	réservé	

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
SQLSTATE		réservé	réservé	réservé
SQLWARNING		réservé	réservé	
SQRT		réservé	réservé	
STABLE	non réservé			
STANDALONE	non réservé	non réservé	non réservé	
START	non réservé	réservé	réservé	
STATE		non réservé	non réservé	
STATEMENT	non réservé	non réservé	non réservé	
STATIC		réservé	réservé	
STATISTICS	non réservé			
STDDEV_POP		réservé	réservé	
STDDEV_SAMP		réservé	réservé	
STDIN	non réservé			
STDOUT	non réservé			
STORAGE	non réservé			
STORED	non réservé			
STRICT	non réservé			
STRING	réservé (peut être une fonction ou un type)	non réservé		
STRIP	non réservé	non réservé	non réservé	
STRUCTURE		non réservé	non réservé	
STYLE		non réservé	non réservé	
SUBCLASS_ORIGIN		non réservé	non réservé	non réservé
SUBMULTISET		réservé	réservé	
SUBSCRIPTION	non réservé			
SUBSET		réservé		
SUBSTRING	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
SUBSTRING_REGEX		réservé	réservé	
SUCCEEDS		réservé	réservé	
SUM		réservé	réservé	réservé
SUPPORT	non réservé			
SYMMETRIC	réservé	réservé	réservé	
SYSID	non réservé			
SYSTEM	non réservé	réservé	réservé	
SYSTEM_TIME		réservé	réservé	
SYSTEM_USER		réservé	réservé	réservé
T		non réservé	non réservé	
TABLE	réservé	réservé	réservé	réservé
TABLES	non réservé			

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
TABLESAMPLE	réservé (peut être une fonction ou un type)	réservé	réservé	
TABLESPACE	non réservé			
TABLE_NAME		non réservé	non réservé	non réservé
TAN		réservé		
TANH		réservé		
TEMP	non réservé			
TEMPLATE	non réservé			
TEMPORARY	non réservé	non réservé	non réservé	réservé
TEXT	non réservé			
THEN	réservé	réservé	réservé	réservé
THROUGH		non réservé		
TIES	non réservé	non réservé	non réservé	
TIME	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
TIMESTAMP	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
TIMEZONE_HOUR		réservé	réservé	réservé
TIMEZONE_MINUTE		réservé	réservé	réservé
TO	réservé, requiert AS	réservé	réservé	réservé
TOKEN		non réservé	non réservé	
TOP_LEVEL_COUNT		non réservé	non réservé	
TRAILING	réservé	réservé	réservé	réservé
TRANSACTION	non réservé	non réservé	non réservé	réservé
TRANSACTIONS_COMMITTED		non réservé	non réservé	
TRANSACTIONS_ROLLED_BACK		non réservé	non réservé	
TRANSACTION_ACTIVE		non réservé	non réservé	
TRANSFORM	non réservé	non réservé	non réservé	
TRANSFORMS		non réservé	non réservé	
TRANSLATE		réservé	réservé	réservé
TRANSLATE_REGEX		réservé	réservé	
TRANSLATION		réservé	réservé	réservé
TREAT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
TRIGGER	non réservé	réservé	réservé	
TRIGGER_CATALOG		non réservé	non réservé	
TRIGGER_NAME		non réservé	non réservé	
TRIGGER_SCHEMA		non réservé	non réservé	

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
TRIM	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
TRIM_ARRAY		réservé	réservé	
TRUE	réservé	réservé	réservé	réservé
TRUNCATE	non réservé	réservé	réservé	
TRUSTED	non réservé			
TYPE	non réservé	non réservé	non réservé	non réservé
TYPES	non réservé			
UESCAPE	non réservé	réservé	réservé	
UNBOUNDED	non réservé	non réservé	non réservé	
UNCOMMITTED	non réservé	non réservé	non réservé	non réservé
UNCONDITIONAL	non réservé	non réservé		
UNDER		non réservé	non réservé	
UNENCRYPTED	non réservé			
UNION	réservé, requiert AS	réservé	réservé	réservé
UNIQUE	réservé	réservé	réservé	réservé
UNKNOWN	non réservé	réservé	réservé	réservé
UNLINK		non réservé	non réservé	
UNLISTEN	non réservé			
UNLOGGED	non réservé			
UNMATCHED		réservé		
UNNAMED		non réservé	non réservé	non réservé
UNNEST		réservé	réservé	
UNTIL	non réservé			
UNTYPED		non réservé	non réservé	
UPDATE	non réservé	réservé	réservé	réservé
UPPER		réservé	réservé	réservé
URI		non réservé	non réservé	
USAGE		non réservé	non réservé	réservé
USER	réservé	réservé	réservé	réservé
USER_DEFINED_TYPE_CATALOG		non réservé	non réservé	
USER_DEFINED_TYPE_CODE		non réservé	non réservé	
USER_DEFINED_TYPE_NAME		non réservé	non réservé	
USER_DEFINED_TYPE_SCHEMA		non réservé	non réservé	
USING	réservé	réservé	réservé	réservé
UTF16		non réservé		
UTF32		non réservé		
UTF8		non réservé		
VACUUM	non réservé			

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
VALID	non réservé	non réservé	non réservé	
VALIDATE	non réservé			
VALIDATOR	non réservé			
VALUE	non réservé	réservé	réservé	réservé
VALUES	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
VALUE_OF		réservé	réservé	
VARBINARY		réservé	réservé	
VARCHAR	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	réservé
VARIADIC	réservé			
VARYING	non réservé, requiert AS	réservé	réservé	réservé
VAR_POP		réservé	réservé	
VAR_SAMP		réservé	réservé	
VERBOSE	réservé (peut être une fonction ou un type)			
VERSION	non réservé	non réservé	non réservé	
VERSIONING		réservé	réservé	
VIEW	non réservé	non réservé	non réservé	réservé
VIEWS	non réservé			
VOLATILE	non réservé			
WHEN	réservé	réservé	réservé	réservé
WHENEVER		réservé	réservé	réservé
WHERE	réservé, requiert AS	réservé	réservé	réservé
WHITESPACE	non réservé	non réservé	non réservé	
WIDTH_BUCKET		réservé	réservé	
WINDOW	réservé, requiert AS	réservé	réservé	
WITH	réservé, requiert AS	réservé	réservé	réservé
WITHIN	non réservé, requiert AS	réservé	réservé	
WITHOUT	non réservé, requiert AS	réservé	réservé	
WORK	non réservé	non réservé	non réservé	réservé
WRAPPER	non réservé	non réservé	non réservé	
WRITE	non réservé	non réservé	non réservé	réservé
XML	non réservé	réservé	réservé	
XMLAGG		réservé	réservé	
XMLATTRIBUTES	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	

Mot clé	PostgreSQL	SQL:2016	SQL:2011	SQL-92
XMLBINARY		réservé	réservé	
XMLCAST		réservé	réservé	
XMLCOMMENT		réservé	réservé	
XMLCONCAT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
XMLDECLARATION		non réservé	non réservé	
XMLDOCUMENT		réservé	réservé	
XMLELEMENT	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
XMLEXISTS	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
XMLFOREST	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
XMLITERATE		réservé	réservé	
XMLNAMESPACES	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
XMLPARSE	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
XMLPI	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
XMLQUERY		réservé	réservé	
XMLROOT	non réservé (ne peut pas être une fonction ou un type)			
XMLSCHEMA		non réservé	non réservé	
XMLSERIALIZE	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
XMLTABLE	non réservé (ne peut pas être une fonction ou un type)	réservé	réservé	
XMLTEXT		réservé	réservé	
XMLVALIDATE		réservé	réservé	
YEAR	non réservé, requiert AS	réservé	réservé	réservé
YES	non réservé	non réservé	non réservé	
ZONE	non réservé	non réservé	non réservé	réservé

Annexe D. Conformité SQL

Cette section explique dans quelle mesure PostgreSQL se conforme à la norme SQL en vigueur. Les informations qui suivent ne représentent pas une liste exhaustive de conformance, mais présentent les thèmes principaux utilement et raisonnablement détaillés.

Le nom complet du standard SQL est ISO/IEC 9075 « Database Language SQL ». Le standard est modifié de temps en temps. La mise à jour la plus récente est apparue en 2016. La version 2016 version a la référence ISO/IEC 9075:2016, ou plus simplement SQL:2016. Les versions antérieures sont SQL:2011, SQL:2008, SQL:2006, SQL:2003, SQL:1999 et SQL-92. Chaque version remplace la précédente. Il n'y a donc aucun mérite à revendiquer une compatibilité avec une version antérieure du standard.

Le développement de PostgreSQL respecte le standard en vigueur, tant que celui-ci ne s'oppose pas aux fonctionnalités traditionnelles ou au bon sens. Un grand nombre des fonctionnalités requises par le standard SQL sont déjà supportées. Parfois avec une syntaxe ou un fonctionnement légèrement différents. Une meilleure compatibilité est attendue pour les prochaines versions.

SQL-92 définit trois niveaux de conformité : basique (*Entry*), intermédiaire (*Intermediate*) et complète (*Full*). La majorité des systèmes de gestion de bases de données se prétendaient compatibles au standard SQL dès lors qu'ils se conformaient au niveau Entry ; l'ensemble des fonctionnalités des niveaux Intermediate et Full étaient, soit trop volumineux, soit en conflit avec les fonctionnalités implémentées.

À partir de SQL99, le standard SQL définit un vaste ensemble de fonctionnalités individuelles à la place des trois niveaux de fonctionnalités définis dans SQL-92. Une grande partie représente les fonctionnalités « centrales » que chaque implantation conforme de SQL doit fournir. Les fonctionnalités restantes sont purement optionnelles.

Les versions standards commençant avec SQL:2003 sont également divisé en parties. Chacune est connue par un pseudonyme. Leur numérotation n'est pas consécutive :

- ISO/IEC 9075-1 Framework (SQL/Framework)
- ISO/IEC 9075-2 Foundation (SQL/Foundation)
- ISO/IEC 9075-3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9 Management of External Data (SQL/MED)
- ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)
- ISO/IEC 9075-14 XML-related specifications (SQL/XML)
- ISO/IEC 9075-15 Multi-dimensional arrays (SQL/MDA)

PostgreSQL couvre les parties 1, 2, 9, 11 et 14. La partie 3 est couverte par l'interface ODBC, et la partie 13 est couverte par le plugin PL/Java, mais une conformance exacte n'est pas actuellement vérifiée par ses composants. Il n'y a pas actuellement d'implantations des parties 4, 10 et 15 pour PostgreSQL.

PostgreSQL supporte la plupart des fonctionnalités majeures de SQL:2016. Sur les 177 fonctionnalités requises pour une conformité « centrale » complète (*full Core conformance*), PostgreSQL se conforme à plus de 170. De plus, il existe une longue liste de fonctionnalités optionelles supportées. À la date de rédaction de ce document, aucune version de quelque système de gestion de bases de données que ce soit n'affiche une totale conformité au cœur de SQL:2016.

Les deux sections qui suivent présentent la liste des fonctionnalités supportées par PostgreSQL et celle des fonctionnalités définies dans SQL:2016 qui ne sont pas encore prises en compte. Ces deux listes sont approximatives : certains détails d'une fonctionnalité présentée comme supportée peuvent ne pas être conformes, alors que de grandes parties d'une fonctionnalité non supportée peuvent être implantées. La documentation principale fournit les informations précises sur ce qui est, ou non, supporté.

> **Note**
>
> Les codes de fonctionnalité contenant un tiret sont des sous-fonctionnalités. Si une sous-fonctionnalité n'est pas supportée, la fonctionnalité elle-même sera déclarée non supportée, alors même que d'autres de ses sous-fonctionnalités le sont.

D.1. Fonctionnalités supportées

Identifiant	Cœur ?	Description	Commentaire
B012		Embedded C	
B021		Direct SQL	
B128		Routine language SQL	
E011	Core	Numeric data types	
E011-01	Core	INTEGER and SMALLINT data types	
E011-02	Core	REAL, DOUBLE PRECISION, and FLOAT data types	
E011-03	Core	DECIMAL and NUMERIC data types	
E011-04	Core	Arithmetic operators	
E011-05	Core	Numeric comparison	
E011-06	Core	Implicit casting among the numeric data types	
E021	Core	Character data types	
E021-01	Core	CHARACTER data type	
E021-02	Core	CHARACTER VARYING data type	
E021-03	Core	Character literals	
E021-04	Core	CHARACTER_LENGTH function	trims trailing spaces from CHARACTER values before counting
E021-05	Core	OCTET_LENGTH function	
E021-06	Core	SUBSTRING function	
E021-07	Core	Character concatenation	
E021-08	Core	UPPER and LOWER functions	

Identifiant	Cœur ?	Description	Commentaire
E021-09	Core	TRIM function	
E021-10	Core	Implicit casting among the character string types	
E021-11	Core	POSITION function	
E021-12	Core	Character comparison	
E031	Core	Identifiers	
E031-01	Core	Delimited identifiers	
E031-02	Core	Lower case identifiers	
E031-03	Core	Trailing underscore	
E051	Core	Basic query specification	
E051-01	Core	SELECT DISTINCT	
E051-02	Core	GROUP BY clause	
E051-04	Core	GROUP BY can contain columns not in <select list>	
E051-05	Core	Select list items can be renamed	
E051-06	Core	HAVING clause	
E051-07	Core	Qualified * in select list	
E051-08	Core	Correlation names in the FROM clause	
E051-09	Core	Rename columns in the FROM clause	
E061	Core	Basic predicates and search conditions	
E061-01	Core	Comparison predicate	
E061-02	Core	BETWEEN predicate	
E061-03	Core	IN predicate with list of values	
E061-04	Core	LIKE predicate	
E061-05	Core	LIKE predicate ESCAPE clause	
E061-06	Core	NULL predicate	
E061-07	Core	Quantified comparison predicate	
E061-08	Core	EXISTS predicate	
E061-09	Core	Subqueries in comparison predicate	
E061-11	Core	Subqueries in IN predicate	
E061-12	Core	Subqueries in quantified comparison predicate	
E061-13	Core	Correlated subqueries	
E061-14	Core	Search condition	
E071	Core	Basic query expressions	
E071-01	Core	UNION DISTINCT table operator	
E071-02	Core	UNION ALL table operator	
E071-03	Core	EXCEPT DISTINCT table operator	
E071-05	Core	Columns combined via table operators need not have exactly the same data type	
E071-06	Core	Table operators in subqueries	
E081	Core	Basic Privileges	
E081-01	Core	SELECT privilege	
E081-02	Core	DELETE privilege	

Identifiant	Cœur ?	Description	Commentaire
E081-03	Core	INSERT privilege at the table level	
E081-04	Core	UPDATE privilege at the table level	
E081-05	Core	UPDATE privilege at the column level	
E081-06	Core	REFERENCES privilege at the table level	
E081-07	Core	REFERENCES privilege at the column level	
E081-08	Core	WITH GRANT OPTION	
E081-09	Core	USAGE privilege	
E081-10	Core	EXECUTE privilege	
E091	Core	Set functions	
E091-01	Core	AVG	
E091-02	Core	COUNT	
E091-03	Core	MAX	
E091-04	Core	MIN	
E091-05	Core	SUM	
E091-06	Core	ALL quantifier	
E091-07	Core	DISTINCT quantifier	
E101	Core	Basic data manipulation	
E101-01	Core	INSERT statement	
E101-03	Core	Searched UPDATE statement	
E101-04	Core	Searched DELETE statement	
E111	Core	Single row SELECT statement	
E121	Core	Basic cursor support	
E121-01	Core	DECLARE CURSOR	
E121-02	Core	ORDER BY columns need not be in select list	
E121-03	Core	Value expressions in ORDER BY clause	
E121-04	Core	OPEN statement	
E121-06	Core	Positioned UPDATE statement	
E121-07	Core	Positioned DELETE statement	
E121-08	Core	CLOSE statement	
E121-10	Core	FETCH statement implicit NEXT	
E121-17	Core	WITH HOLD cursors	
E131	Core	Null value support (nulls in lieu of values)	
E141	Core	Basic integrity constraints	
E141-01	Core	NOT NULL constraints	
E141-02	Core	UNIQUE constraints of NOT NULL columns	
E141-03	Core	PRIMARY KEY constraints	
E141-04	Core	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	
E141-06	Core	CHECK constraints	
E141-07	Core	Column defaults	

Identifiant	Cœur ?	Description	Commentaire
E141-08	Core	NOT NULL inferred on PRIMARY KEY	
E141-10	Core	Names in a foreign key can be specified in any order	
E151	Core	Transaction support	
E151-01	Core	COMMIT statement	
E151-02	Core	ROLLBACK statement	
E152	Core	Basic SET TRANSACTION statement	
E152-01	Core	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	
E152-02	Core	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	
E153	Core	Updatable queries with subqueries	
E161	Core	SQL comments using leading double minus	
E171	Core	SQLSTATE support	
E182	Core	Host language binding	
F021	Core	Basic information schema	
F021-01	Core	COLUMNS view	
F021-02	Core	TABLES view	
F021-03	Core	VIEWS view	
F021-04	Core	TABLE_CONSTRAINTS view	
F021-05	Core	REFERENTIAL_CONSTRAINTS view	
F021-06	Core	CHECK_CONSTRAINTS view	
F031	Core	Basic schema manipulation	
F031-01	Core	CREATE TABLE statement to create persistent base tables	
F031-02	Core	CREATE VIEW statement	
F031-03	Core	GRANT statement	
F031-04	Core	ALTER TABLE statement: ADD COLUMN clause	
F031-13	Core	DROP TABLE statement: RESTRICT clause	
F031-16	Core	DROP VIEW statement: RESTRICT clause	
F031-19	Core	REVOKE statement: RESTRICT clause	
F032		CASCADE drop behavior	
F033		ALTER TABLE statement: DROP COLUMN clause	
F034		Extended REVOKE statement	
F034-01		REVOKE statement performed by other than the owner of a schema object	
F034-02		REVOKE statement: GRANT OPTION FOR clause	
F034-03		REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	
F041	Core	Basic joined table	
F041-01	Core	Inner join (but not necessarily the INNER keyword)	
F041-02	Core	INNER keyword	
F041-03	Core	LEFT OUTER JOIN	

Identifiant	Cœur ?	Description	Commentaire
F041-04	Core	RIGHT OUTER JOIN	
F041-05	Core	Outer joins can be nested	
F041-07	Core	The inner table in a left or right outer join can also be used in an inner join	
F041-08	Core	All comparison operators are supported (rather than just =)	
F051	Core	Basic date and time	
F051-01	Core	DATE data type (including support of DATE literal)	
F051-02	Core	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0	
F051-03	Core	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	
F051-04	Core	Comparison predicate on DATE, TIME, and TIMESTAMP data types	
F051-05	Core	Explicit CAST between datetime types and character string types	
F051-06	Core	CURRENT_DATE	
F051-07	Core	LOCALTIME	
F051-08	Core	LOCALTIMESTAMP	
F052		Intervals and datetime arithmetic	
F053		OVERLAPS predicate	
F081	Core	UNION and EXCEPT in views	
F111		Isolation levels other than SERIALIZABLE	
F111-01		READ UNCOMMITTED isolation level	
F111-02		READ COMMITTED isolation level	
F111-03		REPEATABLE READ isolation level	
F131	Core	Grouped operations	
F131-01	Core	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	
F131-02	Core	Multiple tables supported in queries with grouped views	
F131-03	Core	Set functions supported in queries with grouped views	
F131-04	Core	Subqueries with GROUP BY and HAVING clauses and grouped views	
F131-05	Core	Single row SELECT with GROUP BY and HAVING clauses and grouped views	
F171		Multiple schemas per user	
F181	Core	Multiple module support	
F191		Referential delete actions	
F200		TRUNCATE TABLE statement	
F201	Core	CAST function	
F202		TRUNCATE TABLE: identity column restart option	
F221	Core	Explicit defaults	

Identifiant	Cœur ?	Description	Commentaire
F222		INSERT statement: DEFAULT VALUES clause	
F231		Privilege tables	
F231-01		TABLE_PRIVILEGES view	
F231-02		COLUMN_PRIVILEGES view	
F231-03		USAGE_PRIVILEGES view	
F251		Domain support	
F261	Core	CASE expression	
F261-01	Core	Simple CASE	
F261-02	Core	Searched CASE	
F261-03	Core	NULLIF	
F261-04	Core	COALESCE	
F262		Extended CASE expression	
F271		Compound character literals	
F281		LIKE enhancements	
F292		UNIQUE null treatment	SQL:202x draft
F302		INTERSECT table operator	
F302-01		INTERSECT DISTINCT table operator	
F302-02		INTERSECT ALL table operator	
F304		EXCEPT ALL table operator	
F311	Core	Schema definition statement	
F311-01	Core	CREATE SCHEMA	
F311-02	Core	CREATE TABLE for persistent base tables	
F311-03	Core	CREATE VIEW	
F311-04	Core	CREATE VIEW: WITH CHECK OPTION	
F311-05	Core	GRANT statement	
F312		MERGE statement	
F313		Enhanced MERGE statement	
F314		MERGE statement with DELETE branch	
F321		User authorization	
F341		Usage tables	
F361		Subprogram support	
F381		Extended schema manipulation	
F381-01		ALTER TABLE statement: ALTER COLUMN clause	
F381-02		ALTER TABLE statement: ADD CONSTRAINT clause	
F381-03		ALTER TABLE statement: DROP CONSTRAINT clause	
F382		Alter column data type	
F383		Set column not null clause	
F384		Drop identity property clause	
F385		Drop column generation expression clause	

Identifiant	Cœur ?	Description	Commentaire
F386		Set identity column generation clause	
F391		Long identifiers	
F392		Unicode escapes in identifiers	
F393		Unicode escapes in literals	
F394		Optional normal form specification	
F401		Extended joined table	
F401-01		NATURAL JOIN	
F401-02		FULL OUTER JOIN	
F401-04		CROSS JOIN	
F402		Named column joins for LOBs, arrays, and multisets	
F404		Range variable for common column names	
F411		Time zone specification	differences regarding literal interpretation
F421		National character	
F431		Read-only scrollable cursors	
F431-01		FETCH with explicit NEXT	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	
F431-06		FETCH RELATIVE	
F441		Extended set function support	
F442		Mixed column references in set functions	
F471	Core	Scalar subquery values	
F481	Core	Expanded NULL predicate	
F491		Constraint management	
F501	Core	Features and conformance views	
F501-01	Core	SQL_FEATURES view	
F501-02	Core	SQL_SIZING view	
F502		Enhanced documentation tables	
F531		Temporary tables	
F555		Enhanced seconds precision	
F561		Full value expressions	
F571		Truth value tests	
F591		Derived tables	
F611		Indicator data types	
F641		Row and table constructors	
F651		Catalog name qualifiers	
F661		Simple tables	
F672		Retrospective check constraints	

Identifiant	Cœur ?	Description	Commentaire
F690		Collation support	but no character set support
F692		Extended collation support	
F701		Referential update actions	
F711		ALTER domain	
F731		INSERT column privileges	
F751		View CHECK enhancements	
F761		Session management	
F762		CURRENT_CATALOG	
F763		CURRENT_SCHEMA	
F771		Connection management	
F781		Self-referencing operations	
F791		Insensitive cursors	
F801		Full set function	
F850		Top-level <order by clause> in <query expression>	
F851		<order by clause> in subqueries	
F852		Top-level <order by clause> in views	
F855		Nested <order by clause> in <query expression>	
F856		Nested <fetch first clause> in <query expression>	
F857		Top-level <fetch first clause> in <query expression>	
F858		<fetch first clause> in subqueries	
F859		Top-level <fetch first clause> in views	
F860		<fetch first row count> in <fetch first clause>	
F861		Top-level <result offset clause> in <query expression>	
F862		<result offset clause> in subqueries	
F863		Nested <result offset clause> in <query expression>	
F864		Top-level <result offset clause> in views	
F865		<offset row count> in <result offset clause>	
F867		FETCH FIRST clause: WITH TIES option	
S071		SQL paths in function and type name resolution	
S091-01		Arrays of built-in data types	
S091-03		Array expressions	
S092		Arrays of user-defined types	
S095		Array constructors by query	
S096		Optional array bounds	
S098		ARRAY_AGG	
S111		ONLY in query expressions	
S201		SQL-invoked routines on arrays	
S201-01		Array parameters	
S201-02		Array as result type of functions	
S211		User-defined cast functions	

Identifiant	Cœur ?	Description	Commentaire
S301		Enhanced UNNEST	
S404		TRIM_ARRAY	
T031		BOOLEAN data type	
T071		BIGINT data type	
T121		WITH (excluding RECURSIVE) in query expression	
T122		WITH (excluding RECURSIVE) in subquery	
T131		Recursive query	
T132		Recursive query in subquery	
T133		Enhanced cycle mark values	SQL:202x draft
T141		SIMILAR predicate	
T151		DISTINCT predicate	
T152		DISTINCT predicate with negation	
T171		LIKE clause in table definition	
T172		AS subquery clause in table definition	
T173		Extended LIKE clause in table definition	
T174		Identity columns	
T177		Sequence generator support: simple restart option	
T178		Identity columns: simple restart option	
T191		Referential action RESTRICT	
T201		Comparable data types for referential constraints	
T211-01		Triggers activated on UPDATE, INSERT, or DELETE of one base table	
T211-02		BEFORE triggers	
T211-03		AFTER triggers	
T211-04		FOR EACH ROW triggers	
T211-05		Ability to specify a search condition that must be true before the trigger is invoked	
T211-07		TRIGGER privilege	
T212		Enhanced trigger capability	
T213		INSTEAD OF triggers	
T241		START TRANSACTION statement	
T261		Chained transactions	
T271		Savepoints	
T281		SELECT privilege with column granularity	
T285		Enhanced derived column names	
T312		OVERLAY function	
T321-01	Core	User-defined functions with no overloading	
T321-02	Core	User-defined stored procedures with no overloading	
T321-03	Core	Function invocation	
T321-04	Core	CALL statement	
T321-05	Core	RETURN statement	

Identifiant	Cœur ?	Description	Commentaire
T321-06	Core	ROUTINES view	
T321-07	Core	PARAMETERS view	
T323		Explicit security for external routines	
T325		Qualified SQL parameter references	
T331		Basic roles	
T332		Extended roles	
T341		Overloading of SQL-invoked functions and procedures	
T351		Bracketed SQL comments (/*...*/ comments)	
T431		Extended grouping capabilities	
T432		Nested and concatenated GROUPING SETS	
T433		Multiargument GROUPING function	
T434		GROUP BY DISTINCT	
T441		ABS and MOD functions	
T461		Symmetric BETWEEN predicate	
T491		LATERAL derived table	
T501		Enhanced EXISTS predicate	
T521		Named arguments in CALL statement	
T523		Default values for INOUT parameters of SQL-invoked procedures	
T524		Named arguments in routine invocations other than a CALL statement	
T525		Default values for parameters of SQL-invoked functions	
T551		Optional key words for default syntax	
T581		Regular expression substring function	
T591		UNIQUE constraints of possibly null columns	
T611		Elementary OLAP operations	
T612		Advanced OLAP operations	
T613		Sampling	
T614		NTILE function	
T615		LEAD and LAG functions	
T617		FIRST_VALUE and LAST_VALUE function	
T620		WINDOW clause: GROUPS option	
T621		Enhanced numeric functions	
T622		Trigonometric functions	
T623		General logarithm functions	
T624		Common logarithm functions	
T631	Core	IN predicate with one list element	
T651		SQL-schema statements in SQL routines	
T653		SQL-schema statements in external routines	
T655		Cyclically dependent routines	
T811		Basic SQL/JSON constructor functions	

Identifiant	Cœur ?	Description	Commentaire
T812		SQL/JSON: JSON_OBJECTAGG	
T813		SQL/JSON: JSON_ARRAYAGG with ORDER BY	
T814		Colon in JSON_OBJECT or JSON_OBJECTAGG	
T821		Basic SQL/JSON query operators	
T822		SQL/JSON: IS JSON WITH UNIQUE KEYS predicate	
T823		SQL/JSON: PASSING clause	
T824		JSON_TABLE: specific PLAN clause	
T825		SQL/JSON: ON EMPTY and ON ERROR clauses	
T826		General value expression in ON ERROR or ON EMPTY clauses	
T827		JSON_TABLE: sibling NESTED COLUMNS clauses	
T828		JSON_QUERY	
T829		JSON_QUERY: array wrapper options	
T830		Enforcing unique keys in SQL/JSON constructor functions	
T831		SQL/JSON path language: strict mode	
T832		SQL/JSON path language: item method	
T833		SQL/JSON path language: multiple subscripts	
T834		SQL/JSON path language: wildcard member accessor	
T835		SQL/JSON path language: filter expressions	
T836		SQL/JSON path language: starts with predicate	
T837		SQL/JSON path language: regex_like predicate	
T838		JSON_TABLE: PLAN DEFAULT clause	
X010		XML type	
X011		Arrays of XML type	
X014		Attributes of XML type	
X016		Persistent XML values	
X020		XMLConcat	
X031		XMLElement	
X032		XMLForest	
X034		XMLAgg	
X035		XMLAgg: ORDER BY option	
X036		XMLComment	
X037		XMLPI	
X040		Basic table mapping	
X041		Basic table mapping: nulls absent	
X042		Basic table mapping: null as nil	
X043		Basic table mapping: table as forest	
X044		Basic table mapping: table as element	
X045		Basic table mapping: with target namespace	
X046		Basic table mapping: data mapping	

Identifiant	Cœur ?	Description	Commentaire
X047		Basic table mapping: metadata mapping	
X048		Basic table mapping: base64 encoding of binary strings	
X049		Basic table mapping: hex encoding of binary strings	
X050		Advanced table mapping	
X051		Advanced table mapping: nulls absent	
X052		Advanced table mapping: null as nil	
X053		Advanced table mapping: table as forest	
X054		Advanced table mapping: table as element	
X055		Advanced table mapping: with target namespace	
X056		Advanced table mapping: data mapping	
X057		Advanced table mapping: metadata mapping	
X058		Advanced table mapping: base64 encoding of binary strings	
X059		Advanced table mapping: hex encoding of binary strings	
X060		XMLParse: character string input and CONTENT option	
X061		XMLParse: character string input and DOCUMENT option	
X070		XMLSerialize: character string serialization and CONTENT option	
X071		XMLSerialize: character string serialization and DOCUMENT option	
X072		XMLSerialize: character string serialization	
X090		XML document predicate	
X120		XML parameters in SQL routines	
X121		XML parameters in external routines	
X221		XML passing mechanism BY VALUE	
X301		XMLTable: derived column list option	
X302		XMLTable: ordinality column option	
X303		XMLTable: column default option	
X304		XMLTable: passing a context item	must be XML DOCUMENT
X400		Name and identifier mapping	
X410		Alter column data type: XML type	

D.2. Fonctionnalités non supportées

Les fonctionnalités suivantes définies dans SQL:2011 ne sont pas implantées dans cette version de PostgreSQL. Dans certains cas, des fonctionnalités similaires sont disponibles.

Identifiant	Cœur ?	Description	Commentaire
B011		Embedded Ada	
B013		Embedded COBOL	

Identifiant	Cœur ?	Description	Commentaire
B014		Embedded Fortran	
B015		Embedded MUMPS	
B016		Embedded Pascal	
B017		Embedded PL/I	
B031		Basic dynamic SQL	
B032		Extended dynamic SQL	
B032-01		<describe input statement>	
B033		Untyped SQL-invoked function arguments	
B034		Dynamic specification of cursor attributes	
B035		Non-extended descriptor names	
B041		Extensions to embedded SQL exception declarations	
B051		Enhanced execution rights	
B111		Module language Ada	
B112		Module language C	
B113		Module language COBOL	
B114		Module language Fortran	
B115		Module language MUMPS	
B116		Module language Pascal	
B117		Module language PL/I	
B121		Routine language Ada	
B122		Routine language C	
B123		Routine language COBOL	
B124		Routine language Fortran	
B125		Routine language MUMPS	
B126		Routine language Pascal	
B127		Routine language PL/I	
B200		Polymorphic table functions	
B201		More than one PTF generic table parameter	
B202		PTF Copartitioning	
B203		More than one copartition specification	
B204		PRUNE WHEN EMPTY	
B205		Pass-through columns	
B206		PTF descriptor parameters	
B207		Cross products of partitionings	
B208		PTF component procedure interface	
B209		PTF extended names	
B211		Module language Ada: VARCHAR and NUMERIC support	
B221		Routine language Ada: VARCHAR and NUMERIC support	
F054		TIMESTAMP in DATE type precedence list	

Identifiant	Cœur ?	Description	Commentaire
F121		Basic diagnostics management	
F121-01		GET DIAGNOSTICS statement	
F121-02		SET TRANSACTION statement: DIAGNOSTICS SIZE clause	
F122		Enhanced diagnostics management	
F123		All diagnostics	
F263		Comma-separated predicates in simple CASE expression	
F291		UNIQUE predicate	
F301		CORRESPONDING in query expressions	
F403		Partitioned joined tables	
F451		Character set definition	
F461		Named character sets	
F492		Optional table constraint enforcement	
F521		Assertions	
F671		Subqueries in CHECK	intentionally omitted
F673		Reads SQL-data routine invocations in CHECK constraints	
F693		SQL-session and client module collations	
F695		Translation support	
F696		Additional translation documentation	
F721		Deferrable constraints	foreign and unique keys only
F741		Referential MATCH types	no partial match yet
F812	Core	Basic flagging	
F813		Extended flagging	
F821		Local table references	
F831		Full cursor update	
F831-01		Updatable scrollable cursors	
F831-02		Updatable ordered cursors	
F841		LIKE_REGEX predicate	consider regexp_like()
F842		OCCURRENCES_REGEX function	consider regexp_matches()
F843		POSITION_REGEX function	consider regexp_instr()
F844		SUBSTRING_REGEX function	consider regexp_substr()
F845		TRANSLATE_REGEX function	consider regexp_replace()
F846		Octet support in regular expression operators	
F847		Nonconstant regular expressions	
F866		FETCH FIRST clause: PERCENT option	
R010		Row pattern recognition: FROM clause	

Identifiant	Cœur ?	Description	Commentaire
R020		Row pattern recognition: WINDOW clause	
R030		Row pattern recognition: full aggregate support	
S011	Core	Distinct data types	
S011-01	Core	USER_DEFINED_TYPES view	
S023		Basic structured types	
S024		Enhanced structured types	
S025		Final structured types	
S026		Self-referencing structured types	
S027		Create method by specific method name	
S028		Permutable UDT options list	
S041		Basic reference types	
S043		Enhanced reference types	
S051		Create table of type	partially supported
S081		Subtables	
S091		Basic array support	partially supported
S091-02		Arrays of distinct types	
S094		Arrays of reference types	
S097		Array element assignment	
S151		Type predicate	see pg_typeof()
S161		Subtype treatment	
S162		Subtype treatment for references	
S202		SQL-invoked routines on multisets	
S231		Structured type locators	
S232		Array locators	
S233		Multiset locators	
S241		Transform functions	
S242		Alter transform statement	
S251		User-defined orderings	
S261		Specific type method	
S271		Basic multiset support	
S272		Multisets of user-defined types	
S274		Multisets of reference types	
S275		Advanced multiset support	
S281		Nested collection types	
S291		Unique constraint on entire row	
S401		Distinct types based on array types	
S402		Distinct types based on distinct types	
S403		ARRAY_MAX_CARDINALITY	
T011		Timestamp in Information Schema	
T021		BINARY and VARBINARY data types	

Identifiant	Cœur ?	Description	Commentaire
T022		Advanced support for BINARY and VARBINARY data types	
T023		Compound binary literal	
T024		Spaces in binary literals	
T041		Basic LOB data type support	
T041-01		BLOB data type	
T041-02		CLOB data type	
T041-03		POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types	
T041-04		Concatenation of LOB data types	
T041-05		LOB locator: non-holdable	
T042		Extended LOB data type support	
T043		Multiplier T	
T044		Multiplier P	
T051		Row types	
T053		Explicit aliases for all-fields reference	
T061		UCS support	
T076		DECFLOAT data type	
T101		Enhanced nullability determination	
T111		Updatable joins, unions, and columns	
T175		Generated columns	mostly supported
T176		Sequence generator support	supported except for NEXT VALUE FOR
T180		System-versioned tables	
T181		Application-time period tables	
T211		Basic trigger capability	
T211-06		Support for run-time rules for the interaction of triggers and constraints	
T211-08		Multiple triggers for the same event are executed in the order in which they were created in the catalog	intentionally omitted
T231		Sensitive cursors	
T251		SET TRANSACTION statement: LOCAL option	
T272		Enhanced savepoint management	
T301		Functional dependencies	partially supported
T321	Core	Basic SQL-invoked routines	partially supported
T322		Declared data type attributes	
T324		Explicit security for SQL routines	
T326		Table functions	
T471		Result sets return value	
T472		DESCRIBE CURSOR	
T495		Combined data change and retrieval	different syntax
T502		Period predicates	

Identifiant	Cœur ?	Description	Commentaire
T511		Transaction counts	
T522		Default values for IN parameters of SQL-invoked procedures	supported except DEFAULT key word in invocation
T561		Holdable locators	
T571		Array-returning external SQL-invoked functions	
T572		Multiset-returning external SQL-invoked functions	
T601		Local cursor references	
T616		Null treatment option for LEAD and LAG functions	
T618		NTH_VALUE function	function exists, but some options missing
T619		Nested window functions	
T625		LISTAGG	
T641		Multiple column assignment	only some syntax variants supported
T652		SQL-dynamic statements in SQL routines	
T654		SQL-dynamic statements in external routines	
T839		Formatted cast of datetimes to/from character strings	
M001		Datalinks	
M002		Datalinks via SQL/CLI	
M003		Datalinks via Embedded SQL	
M004		Foreign data support	partially supported
M005		Foreign schema support	
M006		GetSQLString routine	
M007		TransmitRequest	
M009		GetOpts and GetStatistics routines	
M010		Foreign data wrapper support	different API
M011		Datalinks via Ada	
M012		Datalinks via C	
M013		Datalinks via COBOL	
M014		Datalinks via Fortran	
M015		Datalinks via M	
M016		Datalinks via Pascal	
M017		Datalinks via PL/I	
M018		Foreign data wrapper interface routines in Ada	
M019		Foreign data wrapper interface routines in C	different API
M020		Foreign data wrapper interface routines in COBOL	
M021		Foreign data wrapper interface routines in Fortran	
M022		Foreign data wrapper interface routines in MUMPS	
M023		Foreign data wrapper interface routines in Pascal	
M024		Foreign data wrapper interface routines in PL/I	
M030		SQL-server foreign data support	

Identifiant	Cœur ?	Description	Commentaire
M031		Foreign data wrapper general routines	
X012		Multisets of XML type	
X013		Distinct types of XML type	
X015		Fields of XML type	
X025		XMLCast	
X030		XMLDocument	
X038		XMLText	
X065		XMLParse: BLOB input and CONTENT option	
X066		XMLParse: BLOB input and DOCUMENT option	
X068		XMLSerialize: BOM	
X069		XMLSerialize: INDENT	
X073		XMLSerialize: BLOB serialization and CONTENT option	
X074		XMLSerialize: BLOB serialization and DOCUMENT option	
X075		XMLSerialize: BLOB serialization	
X076		XMLSerialize: VERSION	
X077		XMLSerialize: explicit ENCODING option	
X078		XMLSerialize: explicit XML declaration	
X080		Namespaces in XML publishing	
X081		Query-level XML namespace declarations	
X082		XML namespace declarations in DML	
X083		XML namespace declarations in DDL	
X084		XML namespace declarations in compound statements	
X085		Predefined namespace prefixes	
X086		XML namespace declarations in XMLTable	
X091		XML content predicate	
X096		XMLExists	XPath 1.0 only
X100		Host language support for XML: CONTENT option	
X101		Host language support for XML: DOCUMENT option	
X110		Host language support for XML: VARCHAR mapping	
X111		Host language support for XML: CLOB mapping	
X112		Host language support for XML: BLOB mapping	
X113		Host language support for XML: STRIP WHITESPACE option	
X114		Host language support for XML: PRESERVE WHITESPACE option	
X131		Query-level XMLBINARY clause	
X132		XMLBINARY clause in DML	
X133		XMLBINARY clause in DDL	
X134		XMLBINARY clause in compound statements	
X135		XMLBINARY clause in subqueries	

Identifiant	Cœur ?	Description	Commentaire
X141		IS VALID predicate: data-driven case	
X142		IS VALID predicate: ACCORDING TO clause	
X143		IS VALID predicate: ELEMENT clause	
X144		IS VALID predicate: schema location	
X145		IS VALID predicate outside check constraints	
X151		IS VALID predicate with DOCUMENT option	
X152		IS VALID predicate with CONTENT option	
X153		IS VALID predicate with SEQUENCE option	
X155		IS VALID predicate: NAMESPACE without ELEMENT clause	
X157		IS VALID predicate: NO NAMESPACE with ELEMENT clause	
X160		Basic Information Schema for registered XML Schemas	
X161		Advanced Information Schema for registered XML Schemas	
X170		XML null handling options	
X171		NIL ON NO CONTENT option	
X181		XML(DOCUMENT(UNTYPED)) type	
X182		XML(DOCUMENT(ANY)) type	
X190		XML(SEQUENCE) type	
X191		XML(DOCUMENT(XMLSCHEMA)) type	
X192		XML(CONTENT(XMLSCHEMA)) type	
X200		XMLQuery	
X201		XMLQuery: RETURNING CONTENT	
X202		XMLQuery: RETURNING SEQUENCE	
X203		XMLQuery: passing a context item	
X204		XMLQuery: initializing an XQuery variable	
X205		XMLQuery: EMPTY ON EMPTY option	
X206		XMLQuery: NULL ON EMPTY option	
X211		XML 1.1 support	
X222		XML passing mechanism BY REF	parser accepts BY REF but ignores it; passing is always BY VALUE
X231		XML(CONTENT(UNTYPED)) type	
X232		XML(CONTENT(ANY)) type	
X241		RETURNING CONTENT in XML publishing	
X242		RETURNING SEQUENCE in XML publishing	
X251		Persistent XML values of XML(DOCUMENT(UNTYPED)) type	
X252		Persistent XML values of XML(DOCUMENT(ANY)) type	

Identifiant	Cœur ?	Description	Commentaire
X253		Persistent XML values of XML(CONTENT(UNTYPED)) type	
X254		Persistent XML values of XML(CONTENT(ANY)) type	
X255		Persistent XML values of XML(SEQUENCE) type	
X256		Persistent XML values of XML(DOCUMENT(XMLSCHEMA)) type	
X257		Persistent XML values of XML(CONTENT(XMLSCHEMA)) type	
X260		XML type: ELEMENT clause	
X261		XML type: NAMESPACE without ELEMENT clause	
X263		XML type: NO NAMESPACE with ELEMENT clause	
X264		XML type: schema location	
X271		XMLValidate: data-driven case	
X272		XMLValidate: ACCORDING TO clause	
X273		XMLValidate: ELEMENT clause	
X274		XMLValidate: schema location	
X281		XMLValidate with DOCUMENT option	
X282		XMLValidate with CONTENT option	
X283		XMLValidate with SEQUENCE option	
X284		XMLValidate: NAMESPACE without ELEMENT clause	
X286		XMLValidate: NO NAMESPACE with ELEMENT clause	
X300		XMLTable	XPath 1.0 only
X305		XMLTable: initializing an XQuery variable	

D.3. Limitations XML et conformité à SQL/XML

Des révisions significatives aux spécifications liées à XML dans ISO/IEC 9075-14 (SQL/XML) ont été introduites par SQL:2006. L'implémentation PostgreSQL du type de données XML et des fonctions associées suivent en grande partie l'édition précédente de 2003, avec quelques emprunts dans les éditions suivantes. En particulier :

- Où le standard actuel fournit une famille de types de données XML pour contenir un « document » ou « content » dans des variantes non typées ou typées par schéma XML, et un type `XML(SEQUENCE)` pour contenir des parties arbitraires de contenu XML, PostgreSQL fournit le seul type `xml`, qui peut contenir un « document » ou « content ». Il n'y a aucun équivalent au type « sequence »standard.

- PostgreSQL fournit deux fonctions introduites par SQL:2006, mais dans des variantes qui utilisent le langage XPath 1.0, plutôt que XML Query comme spécifié pour elles dans le standard.

Cette section présente certaines des différences de résultat que vous pourriez rencontrer.

D.3.1. Les requêtes sont limitées à XPath 1.0

Les fonctions spécifiques à PostgreSQL `xpath()` et `xpath_exists()` requêtent des documents XML en utilisant le langage XPath. PostgreSQL fournit également des variantes pour XPath uniquement des fonctions du standard `XMLEXISTS` et `XMLTABLE`, qui officiellement utilisent le langage XQuery. Pour toutes ces fonctions, PostgreSQL se repose sur la bibliothèque libxml2, qui ne fournit que XPath 1.0.

Il y a une forte connexion entre le langage XQuery et XPath version 2.0 et supérieure : toute expression qui est syntaxiquement valide et s'exécute avec succès avec les deux produisent le même résultat (avec une exception mineure pour les expressions contenant des références de caractères numériques ou des références d'entités prédéfinies, que XQuery remplace par le caractère correspondant alors que XPath n'y touche pas). Mais il n'y a pas une telle connexion entre ces langages et XPath 1.0 ; c'était un précédent langage qui diffère sur de nombreux aspects.

Il y a deux catégories de limitations à garder à l'esprit : la restriction de XQuery à XPath pour les fonctions spécifiées dans le standard SQL ainsi que la restriction de XPath pour la version 1.0 à la fois pour le standard et pour les fonctions PostgreSQL spécifiques.

D.3.1.1. Restriction de XQuery à XPath

Les fonctionnalités de XQuery allant au delà de XPath incluent :

- Les expressions XQuery peuvent construire et retourner de nouveaux nœuds XML, en plus de toutes les valeurs XPath possibles. XPath peut créer et retourner des valeurs des types atomiques (nombres, chaînes et ainsi de suite), mais peuvent uniquement retourner des nœuds XML qui étaient déjà présents dans les documents fournis en entrées de l'expression.

- XQuery dispose de concepts pour l'itération, tri et regroupement.

- XQuery permet la déclaration et l'utilisation de fonctions locales.

Les versions récentes de XPath commencent à offrir des possibilités qui se chevauchent avec celle-ci (comme un `for-each` de style fonctionnel, des fonctions anonymes et `parse-xml` pour créer un nœud depuis une chaîne), mais ces fonctionnalités ne sont pas disponibles avant XPath 3.0.

D.3.1.2. Restriction de XPath pour la version 1.0

Pour les développeurs familiers avec XQuery et XPath 2.0 ou plus, XPath 1.0 présente un certain nombre de différences :

- Le type fondamental d'une expression XQuery/XPath, la `sequence`, qui peut contenir des nœuds XML, des valeurs atomiques ou les deux, n'existe pas dans XPath 1.0. Une expression 1.0 peut uniquement produire des nœuds-ensemble (contenant zéro ou plus nœuds XML), ou une seule valeur atomique.

- Contrairement à une séquence XQuery/XPath, qui peut contenir n'importe quel élément voulu dans n'importe quel ordre voulu, un nœud-ensemble XPath 1.0 n'a aucune garantie sur l'ordre et, comme n'importe quel ensemble, n'autorise pas de multiples occurrences d'un même élément.

> **Note**
>
> La bibliothèque libxml2 semble toujours retourner des nœud-ensembles à PostgreSQL avec leurs membres dans le même ordre relatif qu'ils avaient dans le document en entrée. Sa documentation ne valide pas ce comportement, et une expression XPath 1.0 ne peut pas le contrôler.

- Bien que XQuery/XPath fournisse tous les types définis dans XML Schema et de nombreux opérateurs et fonctions au dessus de ces types, XPath 1.0 n'a que le nœud-ensemble et les trois types atomiques `boolean`, `double`, et `string`.

- XPath 1.0 n'a pas d'opérateur conditionnel. Une expression XQuery/XPath telle que `if ( hat ) then hat/@size else "no hat"` n'a pas d'équivalent avec XPath 1.0.

- XPath 1.0 n'a pas d'opérateur de comparaison de tri pour les chaînes de caractère. `"cat" < "dog"` et `"cat" > "dog"` sont tous les deux faux, car chacun est une comparaison numérique de deux `NaN`s. En comparaison, `=` et `!=` comparent les chaînes en tant que chaînes.

- XPath 1.0 brouille la distinction entre *value comparisons* et *general comparisons* tels que XQuery/XPath les définis. `sale/@hatsize = 7` et `sale/@customer = "alice"` sont tous les deux existentiellement des comparaisons quantifiées, vraies s'il y a au moins un `sale` avec la valeur donnée pour l'attribut, mais `sale/@taxable = false()` est une comparaison de valeur de la *valeur booléenne effective* d'un nœud-ensemble entier. C'est uniquement vrai si aucun `sale` n'a un attribut `taxable`.

- Dans le modèle de données XQuery/XPath, un *nœud document* peut avoir soit une forme de document (c'est-à-dire exactement un élément de premier niveau, avec uniquement des commentaires et des instructions de traitement en dehors de celui-ci) ou une forme de contenu (avec ces contraintes relâchées). Son équivalent dans XPath 1.0, le *nœud racine*, ne peut être que dans une forme de document. C'est en partie la raison pour laquelle une valeur `xml` passée comme objet de contexte à n'importe laquelle des fonctions PostgreSQL basée sur XPath doit être dans une forme de document.

Les différences mises en évidence ici ne représentent pas l'intégralité des différences. Avec XQuery et les versions 2.0 et supérieures de XPath, il y a un mode de compatibilité XPath 1.0, et les listes W3C de changement de bibliothèques de fonction[1] et de changement du langage[2] appliqués dans ce mode offrent un compte rendu plus complet (mais toujours pas exhaustif) des différences. Le mode de compatibilité ne peut pas rendre les langages supérieurs exactement équivalents à XPath 1.0.

D.3.1.3. Correspondances entre types de données et valeurs SQL et XML

Avec SQL:2006 et ultérieur, les deux sens de conversion entre les types de données standard SQL et les types XML Schema sont précisément spécifiés. Cependant, les règles sont exprimées en utilisant les types et sémantiques de XQuery/XPath, et n'ont pas d'application directe pour les différents modèles de XPath 1.0.

Quand PostgreSQL fait correspondre des valeurs de données SQL vers XML (par exemple avec `xmlelement`), ou XML vers SQL (par exemple comme les colonnes en sortie de `xmltable`), à l'exception de quelques cas traités spécifiquement, PostgreSQL part simplement du principe que le type de données XML chaîne de caractère XPath 1.0 sera valide en tant que forme d'entrée texte du type de données SQL, et réciproquement. Cette règle a l'avantage d'être simple tout en produisant, pour de nombreux types de données, des résultats similaires aux correspondances spécifiées dans le standard.

Quand l'interopérabilité avec les autres systèmes est une préoccupation, pour certains types de données, il peut ne pas être nécessaire d'utiliser explicitement des fonctions de formatage de type de données (telles que celles dans Section 9.8) pour produire les correspondances standard.

D.3.2. Limites accidentelles de l'implémentation

Cette section concerne les limites qui ne sont pas inhérentes à la bibliothèque libxml2, mais qui s'appliquent à l'implémentation actuelle dans PostgreSQL.

[1] https://www.w3.org/TR/2010/REC-xpath-functions-20101214/#xpath1-compatibility
[2] https://www.w3.org/TR/xpath20/#id-backwards-compatibility

D.3.2.1. Seul le mécanisme de passage BY VALUE est supporté

Le standard SQL définit deux *mécanismes de passage* qui s'appliquent quand un argument XML est passé de SQL à une fonction XML ou en recevant un résultat : BY REF, pour lequel une valeur XML particulière retient son identité de nœud, et BY VALUE, pour lequel le contenu du XML est passé mais l'identité du nœud n'est pas préservée. Un mécanisme peut être spécifié avant une liste de paramètres, comme le mécanisme par défaut pour tous ceux-ci, ou après n'importe lequel des paramètres, pour surcharger le mécanisme par défaut.

Pour illustrer la différence, si *x* est une valeur XML, ces deux requêtes dans un environnement SQL:2006 produiraient respectivement vrai et faux :

```
SELECT XMLQUERY('$a is $b' PASSING BY REF x AS a, x AS b NULL ON
 EMPTY);
SELECT XMLQUERY('$a is $b' PASSING BY VALUE x AS a, x AS b NULL ON
 EMPTY);
```

PostgreSQL acceptera BY VALUE ou BY REF dans une construction XMLEXISTS ou XMLTABLE, mais les ignorera. Le type de données xml a une représentation sérialisée en chaîne de caractères, il n'y a donc pas d'identité de nœud à préserver, et le passage est dans les faits toujours BY VALUE.

D.3.2.2. Impossibilité de passer des paramètres nommés aux requêtes

Le support des fonctions basées sur XPath supportent de passer un paramètre afin d'être utilisé comme l'item du contexte de l'expression XPath, mais ne supporte pas de passer des valeurs supplémentaires pour être disponibles dans l'expression comme paramètre nommés.

D.3.2.3. Pas de type XML(SEQUENCE)

Le type de donnéesPostgreSQL xml ne peut contenir qu'une valeur dans une forme DOCUMENT ou CONTENT. Un item de contexte d'expression XQuery/XPath doit être un unique nœud XML ou une valeur atomique, mais XPath 1.0 le restreint en plus à n'être qu'un nœud XML, et n'a aucun type de nœud autorisant CONTENT. Le résultat est qu'un DOCUMENT formé comme il faut est la seule forme de valeur XML que PostgreSQL peut fournir à un item de contexte XPath.

Annexe E. Notes de version

Les notes de version contiennent les modifications significatives apparaissant dans chaque version de PostgreSQL. Elles contiennent aussi les fonctionnalités majeures et les problèmes de migration éventuels. Les notes de version ne contiennent pas les modifications qui n'affectent que peu d'utilisateurs ainsi que les modifications internes, non visibles pour les utilisateurs. Par exemple, l'optimiseur est amélioré dans pratiquement chaque version, mais les améliorations ne sont visibles par les utilisateurs que par la plus grande rapidité des requêtes.

Une liste complète de modifications est récupérable pour chaque version en lisant les validations Git. La liste de diffusion `pgsql-committers`[1] enregistre en plus toutes les modifications du code source. Il existe aussi une interface web[2] montrant les modifications sur chaque fichier.

Le nom apparaissant auprès de chaque élément précise le développeur principal de cet élément. Bien sûr, toutes les modifications impliquent des discussions de la communauté et une relecture des correctifs, donc chaque élément est vraiment un travail de la communauté.

E.1. Release 15.19

Release date: 2026-08-13

This release contains a variety of fixes from 15.18. For information about new features in major release 15, see Section E.20.

E.1.1. Migration to Version 15.19

A dump/restore is not required for those running 15.X.

However, the first three security entries below describe configuration adjustments and data cleanups that you may need to make after updating.

Also, if you use `contrib/btree_gist` or `contrib/ltree`, you may need to reindex indexes made with those extensions; see the relevant entries below.

Also, if you are upgrading from a version earlier than 15.14, see Section E.6.

E.1.2. Changes

- Restrict logical decoding output plugins to the set specified by a new server parameter `output_plugin_libraries` (Jacob Champion)

  Previously, a replication user could select any loadable library for logical decoding, allowing exploits of various sorts. To allow locking this down without breaking setups that worked before, introduce a whitelist of allowed output plugins.

  By default, only the output plugins shipped as part of PostgreSQL (`pgoutput` and `test_decoding`) are included in `output_plugin_libraries`. Installations that rely on other output plugins must add them after updating the server, for example

```
output_plugin_libraries = 'pgoutput, test_decoding,
 my_trusted_decoder'
```

[1] https://www.postgresql.org/list/pgsql-committers/
[2] https://git.postgresql.org/gitweb/?p=postgresql.git;a=summary

Additionally, `pg_upgrade --check` will fail if the `output_plugin_libraries` parameter on the new cluster does not permit the plugins of logical replication slots on the old cluster, when migrating from versions 17 and later. Make necessary additions to the new cluster's setting before performing pg_upgrade.

The PostgreSQL Project thanks Vladimir Tokarev and Yu Kunpeng for reporting this problem. (CVE-2026-6471)

- Fix `contrib/pgcrypto`'s PGP encryption to detect unsupported ciphers (Daniel Gustafsson)

  Previously, if OpenSSL rejected the requested cipher (for example, because it is running in FIPS mode, or the legacy provider hasn't been loaded), pgcrypto failed to notice the failure and simply XOR'd the non-encrypted block with the plaintext, rendering the « encryption » trivially breakable. This will typically occur with deprecated or non-FIPS cipher algorithms (cipher-algo=blowfish/bf, twofish, cast5, or 3des).

  By default, pgcrypto will now fail to decrypt any messages that were affected in this way. To allow retrieval of such data, a new option `ignore-cipher-failure` has been added to `pgp_pub_decrypt()` and `pgp_sym_decrypt()`. Setting `ignore-cipher-failure=1` will restore their previous behavior, allowing the faulty encryption wrapper to be stripped off:

  ```
  pgp_sym_decrypt(encrypted_column, any key, 'ignore-cipher-
  failure=1')
  ```

  Once the affected messages are identified and stripped of their wrappers, they can then be re-encrypted with a modern algorithm. It is important however that the behavior of OpenSSL be the same as it was when the faulty messages were created: if the set of unsupported algorithms is not the same, this approach will not work. See the documentation for `ignore-cipher-failure`.

  The PostgreSQL Project thanks Shishir Sharma for reporting this problem. (CVE-2026-14663)

- Fix psql to skip in-line data following a scripted `COPY ... FROM STDIN` command, even if the `COPY` fails before sending `PGRES_COPY_IN` (Tom Lane)

  Previously, if a `COPY` command failed at startup (for instance, because the target table doesn't exist) psql would not realize that and would proceed to read the following in-line data as SQL commands. In the best case that's wrong and in the worst case it's a SQL-injection hazard. Teach psql to recognize syntactically-valid `COPY ... FROM STDIN` commands and to skip data on its own authority if the server doesn't respond with `PGRES_COPY_IN`.

  While this fix is unlikely to affect any production SQL scripts, test scripts might intentionally exercise failing `COPY ... FROM STDIN` commands. Those will need to gain a `\.` data terminator line after each such command.

  The PostgreSQL Project thanks Alexander Lakhin for reporting this problem. (CVE-2026-6464)

- Cross-check the output row type of a portal running `EXECUTE` or `FETCH` (Robert Haas)

  `EXECUTE` and `FETCH` use two portals: an outer one for the statement itself, and an inner one running the query being executed on its behalf. It was previously possible to make the declared row types of the two portals diverge, leading to server memory disclosure and arbitrary code execution.

  The PostgreSQL Project thanks Ben Morris (in collaboration with Claude and Anthropic Research) and Peter Geoghegan for reporting this problem. (CVE-2026-16239)

- Fix buffer overrun with long time zone abbreviation in `to_char()` (Tom Lane)

  This can easily crash the server, and exploits leading to arbitrary code execution have been reported.

The PostgreSQL Project thanks Hcamael, Amjad Shahzad, Tan Zhen of AntAISecurityLab, Tomer Fichman, Zheng Yu, Amy Burnett (OpenAI Codex Security), Rick de Jager, Heewon Song, Sylvie Mayer, Aleksander Alekseev, and Hillai Ben Sasson for reporting this problem. (CVE-2026-14669)

- Fix buffer overrun in regexp match/split functions (Masahiko Sawada)

  If passed invalidly-encoded data, these functions could write past the end of their conversion buffer.

  The PostgreSQL Project thanks Francesco Verardi for reporting this problem. (CVE-2026-14664)

- Harden the `ascii()` function against invalid input (Michael Paquier)

  By supplying invalidly-encoded input, this function could be coaxed to read and return a few bytes of data that it shouldn't. In assert-enabled builds, its assertions could be triggered too.

  The PostgreSQL Project thanks Hcamael for reporting this problem. (CVE-2026-18024)

- Fix multirange type handling in `pg_restore_attribute_stats()` (OpenAI Security Research Team)

  `pg_restore_attribute_stats()` treated multirange types just like their underlying range type. This works correctly for the bounds histogram, but it was wrong for all the other statistics kinds.

  The PostgreSQL Project thanks Amy Burnett (OpenAI Codex Security) for reporting this problem. (CVE-2026-16238)

- Make `scalarineqsel()` check that a constant it expects to be of type `tid` actually is (Tom Lane)

  This expectation will hold for all the built-in operators that use this estimator, but a maliciously-constructed operator could violate it, leading to a crash or server memory disclosure.

  The PostgreSQL Project thanks Hcamael for reporting this problem. (CVE-2026-14668)

- Harden `tsvector` and `tsquery` code against overly long values (both individual lexemes and total vector/query length) (Tom Lane)

  The documented limits were not enforced in all code paths.

  The PostgreSQL Project thanks Yuhang Wu, Zhenpeng Lin, Zheng Yu, and Hcamael for reporting these problems. (CVE-2026-14662)

- Fix various places that mistakenly assumed they would not have to deal with more than `FUNC_MAX_ARGS` function arguments (Tom Lane)

  Notably, the server's actual limit on the number of arguments to an aggregate function is `FUNC_MAX_ARGS - 1`, but the parser failed to enforce that, creating hazards downstream.

  The PostgreSQL Project thanks Zheng Yu, ylwangtju, and Masahiko Sawada for reporting these problems. (CVE-2026-14679)

- Reject calls from SQL to functions that take or return type `internal` (Tom Lane)

  The existing defenses against doing this have been shown to be insufficient, so add more explicit checks.

  The PostgreSQL Project thanks Amy Burnett (OpenAI Codex Security) for reporting this problem. (CVE-2026-14680)

- Preserve the ownership of extended statistics objects when they are rebuilt by `ALTER TABLE` (Masahiko Sawada)

Previously, the role running `ALTER TABLE` gained ownership of such objects, but that seems inappropriate.

The PostgreSQL Project thanks Noah Misch for reporting this problem. (CVE-2026-6469)

- When deparsing an `EXTRACT()` function call, quote the field name if needed (Nathan Bossart)

  The parser accepts any string literal as a field name in `EXTRACT()`, deferring validation to execution. If the call is stored and deparsed (for example during pg_dump), the string body was regurgitated verbatim, allowing SQL injection.

  The PostgreSQL Project thanks Ben Morris (in collaboration with Claude and Anthropic Research) for reporting this problem. (CVE-2026-15741)

- Check for `USAGE` privilege on data types in places that formerly failed to check that (Nathan Bossart)

  `CREATE TYPE AS RANGE` did not check, nor did `ALTER TABLE OF`, nor did commands that create stored expressions. These omissions allowed roles without `USAGE` privilege to nonetheless create objects depending on the type, possibly blocking the type's owner from changing the type later.

  The PostgreSQL Project thanks Jingzhou Fu for reporting this problem. (CVE-2026-6470)

- Invalidate role-dependent cached plans after role changes (Ilya Staroverov, Shinya Kato, Nathan Bossart)

  Role membership, role attribute, and database ownership changes may impact the expected behavior of row-level security policies, but previously we'd continue to use cached plans that were made according to the old state of affairs.

  The PostgreSQL Project thanks Ilya Staroverov and Shinya Kato for reporting this problem. (CVE-2026-14666)

- Reject GSSEncRequest after direct SSL connection (Michael Paquier)

  After establishing a TLS-encrypted connection, the server would still accept a request for GSSAPI encryption. If that succeeded, the connection would proceed using TLS encryption, but it would look like a GSS connection to the `pg_hba` rules. Thus, a `pg_hba` policy intending to disallow TLS would not be enforced correctly.

  The PostgreSQL Project thanks p4p3r for reporting this problem. (CVE-2026-14681)

- Make mock SCRAM authentication secrets more plausible (Nathan Bossart)

  If a SCRAM login is attempted against a role that doesn't exist or doesn't have a SCRAM secret, we generate a mock secret and carry out the authentication handshake anyway, to avoid revealing these facts to an attacker. But the mock secret was made with a fixed iteration count, which in itself can be an observable response discrepancy. Use the configuration setting `scram_iterations` instead, to make the mock secret look more like the installation's real secrets.

  The PostgreSQL Project thanks Radim Marek for reporting this problem. (CVE-2026-14672)

- Fix out-of-bounds writes in ecpg applications caused by invalid `bytea` data received from the server (Michael Paquier)

  ecpg assumed without checking that any `bytea` value must begin with `\x`. A broken or malicious server might send a string shorter than 2 bytes, resulting in memory clobber in the application.

  The PostgreSQL Project thanks ylwangtju for reporting this problem. (CVE-2026-16241)

- Do not do backquote expansion on the argument of psql's `\unrestrict` command (Nathan Bossart)

  This oversight in the fix for CVE-2025-8714 allows a malicious server to inject shell commands into plain-text dump output that will be run at restore time on the machine running psql, the exact scenario that CVE-2025-8714 intended to prevent.

  The PostgreSQL Project thanks Lucas Velgus, Filip Janus, and Daniel Bakker for reporting this problem. (CVE-2026-18408)

- Remove pg_dump's assumption that `pg_proc.protrftypes` cannot have more than `FUNC_MAX_ARGS` entries (Tom Lane)

  Since there could be entries for both input and output arguments, it's feasible for this array's length to exceed `FUNC_MAX_ARGS` (which constrains only input arguments). Even if that were not so, pg_dump cannot assume that the server was built with the same value of `FUNC_MAX_ARGS` that it has. An overrun would lead to a memory clobber inside pg_dump.

  The PostgreSQL Project thanks Masahiko Sawada for reporting this problem. (CVE-2026-19385)

- Harden PL/Perl against « tied » Perl arrays and hashes (Tom Lane)

  A tied object that doesn't behave like a regular one could lead to memory overwrite, or to constructing a corrupt result array (which would likely cause problems later).

  The PostgreSQL Project thanks Hcamael for reporting this problem. (CVE-2026-14670)

- Fix integer overflows in memory-allocation calculations in PL/Perl and PL/Tcl (Heikki Linnakangas)

  This is the same type of problem as CVE-2026-6473, just in a different part of the code, and is fixed in the same way.

  The PostgreSQL Project thanks the Tulya Project (Team Dhiutsa, Bitecope Technologies Private Ltd) for reporting this problem. (CVE-2026-14677)

- Ensure that `contrib/amcheck` functions restrict `search_path` before executing index expressions (Noah Misch)

  Because amcheck will run such index expressions as the owner of their tables, a caller could potentially hijack `search_path`-dependent functions to run arbitrary code as the table owner. By default this is not a vulnerability because only superusers are allowed to call amcheck functions; but if that privilege was granted out, it created a larger hazard than the documentation suggests.

  The PostgreSQL Project thanks Yuelin Wang and Jacob Brazeal for reporting this problem. (CVE-2026-14673)

- Fix integer overflows in `contrib/fuzzystrmatch`'s `levenshtein()` and `levenshtein_less_equal()` functions (Nathan Bossart)

  Passing large cost values to these functions could cause integer overflows, thereby producing nonsensical results, and even causing out-of-bounds writes in some cases.

  The PostgreSQL Project thanks Ben Morris (in collaboration with Claude and Anthropic Research) for reporting this problem. (CVE-2026-15742)

- Fix buffer overrun in `contrib/pg_stat_statements` (Álvaro Herrera)

  Query normalization didn't accurately account for the amount of space the normalized string would require.

The PostgreSQL Project thanks Sajeeb Lohani (with TrendAI Zero Day Initiative) and Yuelin Wang for reporting this problem. (CVE-2026-14676)

- Fix datatype error in `contrib/pg_trgm`'s GiST picksplit function (Heikki Linnakangas)

  This mistake resulted in reading past the end of the buffer, typically causing bad split decisions; but a crash could ensue if you're very unlucky.

  The PostgreSQL Project thanks Mehmet D. Ince for reporting this problem. (CVE-2026-14678)

- Remove the plan cache in `contrib/refint` (Ayush Tiwari)

  This caching behavior has several serious bugs, notably that `check_foreign_key()` embeds the new key values in its cascade-UPDATE queries, so a cached plan reuses the originally-needed values rather than the key values that should be used. The simplest solution is to remove it.

  The PostgreSQL Project thanks Hcamael for reporting this problem. (CVE-2026-14671)

- Fix self-deadlock when replaying WAL generated by an older minor version (Andrey Borodin)

  This error was introduced in the previous set of minor releases. It caused standby servers that were following a primary of an older minor release version to get stuck in some scenarios.

- Fix mis-handling of asynchronous reads when rescanning an asynchronous Append plan node (Alexander Korotkov, Gleb Kashkin, Etsuro Fujita)

  When an upper plan node rescans an Append before having read the entire Append output, we need to discard any in-flight requests sent to external servers (by `postgres_fdw` for example). This was not done correctly in cases where a subplan has parameter changes or is discarded by partition pruning in the next scan. The outcome could be incorrect query results, an infinite loop, or an assertion failure.

- Fix error in partition pruning for RANGE-partitioned tables (David Rowley)

  In some cases the DEFAULT partition would be skipped when it should not be, which could lead to rows missing from query results.

- Fix planner's nullability and strictness checks for *`value`* `IN (`*`array`*`)` expressions (Ayush Tiwari)

  These checks should only succeed if the array operand is known to be non-empty, but that consideration was missed, allowing optimizations to be applied that should not be. This could result in wrong query answers if the array actually was empty.

- Add missed checks for hashability of equality comparisons on container datatypes (arrays, composites, ranges) (Andrei Lepikhov, Tom Lane)

  The planner must verify hashability of the container's component type(s) before deciding it can use a hash-based plan type. This step was missed in some places, leading to « could not identify a hash function » failures at execution.

- Fix mis-optimization of `COUNT` window functions that have an `EXCLUDE` clause or lack `ORDER BY` (Chengpeng Yan, David Rowley)

  These window functions were treated as monotonic when they should not be, allowing wrong answers to be computed.

- Fix `ALTER COLUMN ... DROP EXPRESSION` to work when there are multiple levels of partitions (Alberto Piai)

- Disallow renaming a rule to `_RETURN` (Tom Lane)

That name is reserved for a view's `ON SELECT` rule, but `ALTER RULE` allowed renaming other rules to `_RETURN`, causing trouble later.

- Fix use of `REINDEX CONCURRENTLY` with a deferred uniqueness constraint (Nitin Motiani)

  The transient index copy created during `REINDEX CONCURRENTLY` was incorrectly marked as enforcing immediate uniqueness, causing spurious reports of constraint violation.

- Fix matching of localized month/day names in `to_date()` (Heikki Linnakangas)

  The matching logic misbehaved in cases where case-folding changes the byte length of the string.

- Fix incorrect NFC recomposition for Hangul U+11A7 (TBASE) (Diego Frias, Michael Paquier)

  This character was treated as a valid T syllable, which it is not, and hence silently swallowed during normalization.

- Avoid possible truncation of output lexemes in case-insensitive `synonym` dictionaries (Jeff Davis)

  If folding to lower case increased the byte length of a lexeme, it was incorrectly truncated to its original byte length when emitted.

- Fix typo in `hash_record_extended()` (Man Zeng)

  The code failed to initialize the second isnull argument passed to FunctionCallInvoke(). This is harmless for existing in-core extended hash support functions, which will not examine that value. However, extension-provided hash functions could be affected if they inspect `PG_ARGISNULL(1)`.

- Prevent `satisfies_hash_partition()` from crashing with `VARIADIC NULL` (Robert Haas)

- Report invalid-weight errors more cleanly and consistently in `tsvector_filter()` and allied functions (Ewan Young)

  In particular, report weight characters that are not printable ASCII in octal form (`\`*nnn*), as `charout()` would render them. This avoids possibly producing an invalidly-encoded error message.

- Fix mishandling of namespace nodes in `xpath()` (Michael Paquier)

  This fix avoids an unexpected « could not copy node » error.

- Treat an undefined jsonpath variable as an error even when no variables are supplied (Andrey Rachitskiy)

  The `jsonb @?` and `@@` operators cannot supply any variables to be used in their jsonpath expressions. This code path erroneously treated an unknown jsonpath variable as a JSON null, rather than raising an error as expected. Aside from not being the expected behavior, this mistake could result in unbounded memory consumption.

- Avoid machine-dependent behavior when dividing the smallest possible `money` value by -1 (Andrey Rachitskiy)

- Fix crash after out-of-memory failure partway through creation of a cache entry for a text search dictionary (Tom Lane)

- Fix memory-safety bugs in processing of incorrect ispell/hunspell dictionary files (Andrey Rachitskiy)

- Honor query cancel and vacuum delay during GIN index posting-tree cleanup (Paul Kim, Alexander Korotkov)

  The posting tree for a common value can be large, so that this missed check could allow vacuum to run for a long time before noticing an interrupt.

- Fix possible mis-decoding of index tuples during GiST and SP-GiST index-only scans (Peter Geoghegan)

  This error could lead to emitting corrupted data from an index-only scan plan. The only affected core opclass is GiST's range_ops, and it could only fail if the range column were not the first index column.

- When creating directories, tolerate concurrent creation of the same directory (Andrew Dunstan, Tom Lane)

- Prevent creation of dangling object dependencies by acquiring a shared lock on any object being depended on (Bertrand Drouvot)

  The shared lock will conflict with any attempt to drop the depended-on object, eliminating the race condition that formerly existed. For example, if one session drops a schema (that appears empty to it) concurrently with some other session creating a function in that schema, previously both transactions could commit, leaving an invalid function definition behind. Now, one transaction or the other will fail.

- Fix race condition in conflict detection for `SERIALIZABLE` isolation mode (Peter Geoghegan)

  A conflict could be missed when examining an initially-empty btree index, allowing failure of serializability due to improperly allowing conflicting transactions to commit.

- Fix race condition in ProcSignalBarrier code (Masahiko Sawada)

  This error could result in processes getting stuck, typically after reporting « still waiting for backend with PID *nnnn* to accept ProcSignalBarrier ».

- Fix race conditions when a set of processes that belong to the same lock group exit at the same time (Vlad Lesin)

  These errors could lead to PANIC aborts, with messages such as « latch already owned ». The issue does not normally arise in regular parallel query, since the leader won't exit before seeing its workers finish; but some extensions reach the problem.

- Avoid exposing a WAL receiver's full connection string during timeline jumps (Chao Li)

  The `pg_stat_wal_receiver` view should show a sanitized version of the connection string, without sensitive data. But it transiently showed the full string when we re-use an existing WAL receiver.

- Use run-time checks, not just Asserts, to verify the correct number of columns in tuples received during logical replication (Varik Matevosyan)

  A malicious or buggy publisher could send inconsistent numbers of columns. While we could not find a scenario in which this would have serious ill effects, extra caution seems warranted.

- Clean up quoting of string parameters within constructed replication commands (Tom Lane)

  Various places that generate replication commands were not being adequately careful about quoting replication slot names and other parameters that need to be inserted into those commands. This could result in unexpected syntax errors in those commands. In principle, a crafted replication slot name could result in SQL injection; but such a scenario seems very unlikely to occur in practice,

since replication operations can only be invoked by highly-privileged users and there is no reason for them to use a slot name coming from an untrustworthy source.

- Fix logical decoding of empty prepared transactions (Masahiko Sawada)

  A prepared transaction that did not cause any decodable updates could result in sending `COMMIT/ROLLBACK PREPARED` to the output plugin with no preceding `PREPARE`. For the built-in subscriber this breaks replication, and other plugins will probably not like it either.

- Fix corruption of unlogged sequences after standby promotion (Fujii Masao)

  Previously, if an unlogged sequence was created on the primary and replicated to a standby, accessing the sequence after promoting the standby could fail with « bad magic number in sequence » or related errors.

- Fix cascading standby reconnect failure after archive fallback (Marco Nenciarini)

  A cascading standby could fail to reconnect to its upstream standby with « requested starting point ... is ahead of the WAL flush position » after falling back to archive recovery.

- Avoid race condition while dropping ephemeral replication slots (Zhijie Hou)

  The slot-releasing code performed some additional updates to the replication slot's shared-memory entry after releasing the slot. This is unsafe since another session could immediately re-use the dropped slot's shared-memory entry. Skip those updates in the case of an ephemeral slot.

- Fix stale progress reports during logical replication table synchronization (Shinya Kato)

  Previously, the `pg_stat_progress_copy` view in the subscriber would continue to show the initial `COPY` operation as active even after the data copy had finished. The stale entry remained visible until synchronization caught up with the publisher.

- Clear base backup progress on backup failure (Chao Li)

  Previously the `pg_stat_progress_basebackup` view would continue to show a stale progress entry after a failure, until the replication client disconnected. pg_basebackup normally disconnects immediately, but other clients might not.

- Fix possible PANIC due to concurrent drop of pgstats entries when `track_functions` is enabled (Sami Imseih, Michael Paquier)

- Clean up broken local pgstats entry after failing to obtain space for the corresponding shared hashtable entry (Niall Newman)

  Failure to do this led to a null-pointer dereference the next time the local entry was used.

- In PL/Perl, avoid NULL pointer dereference crash when working with an invalid `PostgreSQL::InServer::ARRAY` object (Xing Guo)

- In PL/Python, properly check for errors when working with sequence and mapping objects (Richard Guo)

  Previously, a broken object or an unhandled exception could result in a NULL pointer dereference crash.

- In libpq, always drain all pending bytes from the SSL or GSS decryption buffer during `pqReadData()` (Jacob Champion)

  This avoids edge cases where libpq or its calling application waits for more data to arrive on the socket, but actually all the data has already arrived.

- Allow libpq to accept ParameterDescription messages exceeding 30000 bytes (Ning Sun)

Previously, this message type was not among those that libpq's validity heuristics believed could be long. The limit resulted in failure for prepared queries having more than 7498 parameters, which is unlikely but supported.

- Reject multiple descriptor header items in ecpg's `GET/SET DESCRIPTOR` statements (Masashi Kamura)

  Previously the grammar allowed this syntax, but broken C code was generated. Adjust the grammar and the documentation to allow only one header item.

- Make line widths match in psql's expanded aligned output format (Pavel Stehule)

  When the table's data rows are narrower than the record header lines, widen the data rows to match the headers, avoiding unsightly output.

- Fix psql's privilege check for showing database size in `\l+` (Christoph Berg)

  The underlying server function permits users who have `pg_read_all_stats` privileges to see the sizes of all databases, even if they lack `CONNECT` privilege. But psql was unaware of that provision and would not call the function unless the user has `CONNECT` privilege.

- Fix psql's tab completion for `\df` to consider procedures too (Erik Wienhold)

- Fix thread-safety bug in pgbench (Fujii Masao)

  When pgbench runs with multiple threads and the `--verbose-errors` option, different threads could attempt to use the same buffer to construct error messages, leading to corrupted log output.

- Use the source cluster's group-read file permissions for pg_recvlogical output files (Fujii Masao)

  pg_recvlogical was documented to behave this way, but it never actually enabled group-read.

- In `contrib/amcheck`, handle short-header varlena datums correctly (Andrey Borodin)

  This error could result in doing excess work while verifying a btree index, but seems not to have had any worse consequences.

- In `contrib/btree_gist`, fix `NaN` handling in the `float4` and `float8` opclasses (Bill Kim, Tom Lane)

  Comparisons, as well as the GiST penalty and distance functions, did not account for `NaN` and would give the wrong answer when handed one. It is recommended to reindex `btree_gist` indexes on float columns after installing this update, if there is any possibility that there are `NaN` entries in those columns.

- In `contrib/btree_gist`, fix searches using a not-equal operator (Ayush Tiwari)

  For variable-length data types, the code for scanning non-leaf index pages applied the wrong comparison function, leading to wrong results and potentially crashes.

- Fix unguarded recursion and loops in `contrib/hstore_plperl`, `contrib/jsonb_plperl`, and `contrib/jsonb_plpython` (Aleksander Alekseev)

  Prevent stack overflow when dealing with deeply nested `jsonb` values, and allow interruption of the infinite loop caused when attempting to dereference circular chains of Perl object references.

- Fix missed release of statistics catcache entry in `contrib/intarray` (Man Zeng)

  This oversight led to warnings like « resource was not closed: cache pg_statistic ».

- In `contrib/ltree`, fix integer overflow in comparisons (Ayush Tiwari)

`ltree` values containing more than about 14,653 labels resulted in wrong comparison answers due to overflow. If a btree index contains such values, it is probably corrupt and should be reindexed after installing this update.

- Fix array overrun in `contrib/pg_surgery`'s `heap_force_kill` and `heap_force_freeze` functions (Michael Paquier)

  Attempting to change a TID whose offset number equals MaxHeapTuplesPerPage wrote one byte past the end of the allocated array, potentially crashing the server.

- In `contrib/pg_surgery`, avoid infinite loop with TID arrays having more than 64K elements (Andrey Rachitskiy)

- Avoid NULL-pointer dereference in `contrib/refint`'s `check_foreign_key()` (Ayush Tiwari)

  In the on-update-cascade case, a null value of a referenced column led to a crash. This is an oversight in the fix for CVE-2026-6637, but the code that was there before that wasn't really right either.

- Fix `contrib/seg` to print segments with ~ certainty indicators correctly (Ewan Young)

  Due to a typo, `seg_out()` did not print a ~ certainty indicator attached to a segment's upper boundary. Worse, if the lower boundary had ~ while the upper boundary had no indicator, the upper boundary was not printed at all, incorrectly converting the value into an open interval.

- Fix crash with namespace nodes in `contrib/xml2`'s `xpath_nodeset()` function (Andrey Chernyy, Michael Paquier)

- Support building PostgreSQL with Visual Studio 2026 (Andrew Dunstan)

- Support building PostgreSQL with OpenSSL 4 (Daniel Gustafsson)

- Update time zone data files to tzdata release 2026c (Tom Lane)

  Alberta (America/Edmonton) will be on year-round UTC-06 (effectively, permanent DST) beginning in November 2026. This release assumes that their TZ abbreviation will be `CST` from that time forward. That seems likely to change, but it's unclear what new abbreviation will be used.

  Morocco (Africa/Casablanca) will move to permanent UTC+00, without daylight saving time transitions, on 2026-09-20.

E.2. Release 15.18

Release date: 2026-05-14

This release contains a variety of fixes from 15.17. For information about new features in major release 15, see Section E.20.

E.2.1. Migration to Version 15.18

A dump/restore is not required for those running 15.X.

However, if you are upgrading from a version earlier than 15.14, see Section E.6.

E.2.2. Changes

- Prevent unbounded recursion while processing startup packets (Michael Paquier)

  A malicious client could crash the connected backend by alternating rejected SSL and GSS encryption requests indefinitely.

The PostgreSQL Project thanks Calif.io (in collaboration with Claude and Anthropic Research) for reporting this problem. (CVE-2026-6479)

- Fix assorted integer overflows in memory-allocation calculations (Tom Lane, Nathan Bossart, Heikki Linnakangas)

  Various places were incautious about the possibility of integer overflow in calculations of how much memory to allocate. Overflow would lead to allocating a too-small buffer which the caller would then write past the end of. This would at least trigger server crashes, and probably could be exploited for arbitrary code execution. In many but by no means all cases, the hazard exists only in 32-bit builds.

  The PostgreSQL Project thanks Xint Code, Bruce Dang, Sven Klemm, and Pavel Kohout for reporting these problems. (CVE-2026-6473)

- Reject over-length options in `ts_headline()` (Michael Paquier)

  The `StartSel`, `StopSel` and `FragmentDelimiter` strings must not exceed 32Kb in length, but this was not checked for. An over-length value would typically crash the server.

  The PostgreSQL Project thanks Xint Code for reporting this problem. (CVE-2026-6473)

- Guard against malicious time zone names in `timeofday()` and `pg_strftime()` (Tom Lane)

  A crafted time zone setting could pass `%` sequences to `snprintf()`, potentially causing crashes or disclosure of server memory. Another path to similar results was to overflow the limited-size output buffer used by `pg_strftime()`.

  The PostgreSQL Project thanks Xint Code for reporting this problem. (CVE-2026-6474)

- When creating a multirange type, ensure the user has `CREATE` privilege on the schema specified for the multirange type (Jelte Fennema-Nio)

  The multirange type can be put into a different schema than its parent range type, but we neglected to apply the required privilege check when doing so.

  The PostgreSQL Project thanks Jelte Fennema-Nio for reporting this problem. (CVE-2026-6472)

- Use timing-safe string comparisons in authentication code (Michael Paquier)

  Use `timingsafe_bcmp()` instead of `memcmp()` or `strcmp()` when checking passwords, hashes, etc. It is not known whether the data dependency of those functions is usefully exploitable in any of these places, but in the interests of safety, replace them.

  The PostgreSQL Project thanks Joe Conway for reporting this problem. (CVE-2026-6478)

- Mark `PQfn()` as unsafe, and avoid using it within libpq (Nathan Bossart)

  For a non-integral result type, `PQfn()` is not passed the size of the output buffer, so it cannot check that the data returned by the server will fit. A malicious server could therefore overwrite client memory. This is unfixable without an API change, so mark the function as deprecated. Internally to libpq, use a variant version that can apply the missing check.

  The PostgreSQL Project thanks Yu Kunpeng and Martin Heistermann for reporting this problem. (CVE-2026-6477)

- Prevent path traversal in pg_basebackup and pg_rewind (Michael Paquier)

  These applications failed to validate output file paths read from their input, so that a malicious source could overwrite any file writable by these applications. Constrain where data can be written by rejecting paths that are absolute or contain parent-directory references.

The PostgreSQL Project thanks XlabAI Team of Tencent Xuanwu Lab and Valery Gubanov for reporting this problem. (CVE-2026-6475)

- Guard against field overflow within `contrib/intarray`'s `query_int` type and `contrib/ltree`'s `ltxtquery` type (Tom Lane)

  Parsing of these query structures did not check for overflow of 16-bit fields, so that construction of an invalid query tree was possible. This can crash the server when executing the query.

  The PostgreSQL Project thanks Xint Code for reporting this problem. (CVE-2026-6473)

- Guard against overly long values of `contrib/ltree`'s `lquery` type (Michael Paquier)

  Values with more than 64K items caused internal overflows, potentially resulting in stack smashes or wrong answers.

  The PostgreSQL Project thanks Vergissmeinnicht, A1ex, and Jihe Wang for reporting this problem. (CVE-2026-6473)

- Prevent SQL injection and buffer overruns in `contrib/spi` (Nathan Bossart)

  `check_foreign_key()` was insufficiently careful about quoting key values, and also used fixed-length buffers for constructing queries. While this module is only meant as example code, it still shouldn't contain such dangerous errors.

  The PostgreSQL Project thanks Nikolay Samokhvalov for reporting this problem. (CVE-2026-6637)

- Check for nondeterministic collations before assuming that an equality condition on a collatable type implies uniqueness (Richard Guo)

  Numerous planner optimizations assume that, for example, at most one table row can satisfy `WHERE x = 'abc'` if there is a unique index on `x`. However this conclusion is unsafe in general if the index and the `WHERE` clause have different collations attached. It is safe when both collations are deterministic, because that property essentially requires that equality of two strings means bitwise equality. But nondeterministic collations don't act that way, so that optimizing on the assumption of unique matches can give wrong query answers if either the `WHERE` clause or the index has a nondeterministic collation.

- Fix incorrect handling of `NEW` generated columns in rule actions and rule qualifications (Richard Guo, Dean Rasheed)

  Previously, such column references would produce NULL in `INSERT` cases, or be equivalent to the `OLD` value in `UPDATE` cases.

- Fix spurious « generated columns are not supported in COPY FROM WHERE conditions » errors (Tom Lane)

  Use of a system column in a `COPY FROM WHERE` condition could sometimes incorrectly report this error.

- Correctly report a serialization failure when `MERGE` encounters a concurrently-updated tuple in repeatable-read or serializable mode (Tender Wang)

  Previously, such cases behaved the same as in lower isolation levels.

- Fix `CREATE TABLE ... LIKE ... INCLUDING STATISTICS` for cases where the source table has dropped column(s) (Julien Tachoires)

  In such cases, extended statistics objects could be copied incorrectly, or the command could give an incorrect error.

- Allow `ALTER INDEX ... ATTACH PARTITION` to mark the parent index valid if appropriate (Sami Imseih)

  There are edge cases in which a partitioned index might remain marked as invalid even when all its leaf indexes are valid. This change provides a mechanism whereby a user can correct such a situation without resorting to manual catalog updates.

- Fix `ALTER FOREIGN DATA WRAPPER` to not drop the wrapper object's dependency on its handler function (Jeff Davis)

- Disallow making a composite type be a member of itself via a multirange (Heikki Linnakangas)

  We already forbade such cases when the intermediate type is a domain, array, composite type, or range; but multiranges were overlooked.

- Fix datum-image comparisons to be insensitive to sign-extension variations (David Rowley)

  This fixes some situations that previously led to « could not find memoization table entry » errors or wrong query results.

- Fix incorrect logic for hashed `IN`/`NOT IN` with non-strict equality operator (Chengpeng Yan)

  The previous coding could crash or give wrong answers. All built-in data types have strict equality operators, so that this issue could only arise with an extension data type.

- Truncate overly-long locale-specific numeric symbols in `to_char()` (Tom Lane)

  If a locale specified a currency symbol, thousands separator, or decimal or sign symbol more than 8 bytes long, a buffer overrun was possible. No such locales exist in the real world, and it's impractical for an unprivileged attacker to install a malicious locale definition underneath a Postgres server; but for safety's sake check for overlength symbols and truncate if needed.

- Prevent buffer overruns when parsing an affix file for an `Ispell` dictionary (Tom Lane)

  A corrupt or malicious affix file could crash the server. This is not considered a security issue because text search configuration files are presumed trustworthy, but it still seems worth fixing.

- Guard against integer overflow in calculations of frame start and end positions for window aggregates (Richard Guo)

  Very large user-specified offsets (close to INT64_MAX) could result in errors or incorrect query results.

- Fix incorrect behavior of `pg_stat_reset_single_table_counters()` on a shared catalog (Chao Li)

  Such cases had a side-effect of resetting the current database's `stat_reset_timestamp`, which was unintended.

- Fix buffer overread when `pglz_decompress()` receives corrupt input (Andrew Dunstan)

  It was possible to read a few bytes past the end of the input, which in very unlucky cases might cause a crash.

- Ensure that tuplestore data structures are internally consistent even after an error (Tom Lane)

  The code was previously careless about this, which is fine most of the time but is problematic for the tuplestore backing a `WITH HOLD` cursor. In v15 and before this leads to easily-reproducible crashes; later branches are not known to be vulnerable, but it seems best to preserve consistency in all.

- Fix premature NULL lag reporting in `pg_stat_replication` (Shinya Kato)

The lag columns frequently read as NULL even while replication activity was happening.

- Avoid rare flush failure when working with non-WAL-logged GiST indexes (Tomas Vondra)

  A non-logged GiST index could nonetheless sometimes produce « xlog flush request *n/nnnn* is not satisfied » errors, due to incorrect selection of a « fake LSN » to represent an insertion point.

- Fix underestimate of required size of DSA page maps for odd-size segments (Paul Bunn)

  This miscalculation led to out-of-bounds accesses and hence server crashes.

- Fix possible server crash when processing extended statistics on expressions of extension data types (Michael Paquier)

  NULL pointer dereferences were possible if the data type's typanalyze function does not compute any useful statistics. No in-core typanalyze function behaves that way, but extensions could.

- If the startup process fails, properly shut down other child processes before exiting the postmaster (Ayush Tiwari)

  The handling of this situation relied on a long-obsolete assumption that no other postmaster children exist while the startup process is running, so that immediate postmaster exit is acceptable. Orphaned children would eventually notice the postmaster's death and exit on their own, but a cleaner shutdown procedure is desirable.

- Fix race condition between WAL replay of checkpoints and multixact ID creations (Heikki Linnakangas)

  A standby server following WAL from a primary of an older minor version could get into a crash-and-restart loop complaining about « could not access status of transaction ».

- Prevent indefinite wait in shutdown of a walsender process (Anthonin Bonnefoy)

  At shutdown of a cluster that is publishing logical replication data, the walsender waits for all pending WAL to be written out. But it did not correctly request that to happen, so that in some cases this could become an indefinite wait.

- Ensure that changes to tables' free space maps are persisted during recovery (Alexey Makhmutov)

  Previously, while WAL replay did update the free space map while replaying operations that should change it, the map page buffer did not get marked dirty if checksums are enabled, so that the changes might never get written out. On a standby server, over time this would result in a map wildly at variance with the table's actual contents. While the map is only used as a hint, this condition could cause significant performance degradation for some period of time after the standby server is promoted to be active, until most of the map has been repaired by updates.

- Fix crashes in some ecpg functions when called without any established connection (Shruthi Gowda)

- Fix assorted bugs in backup decompression and tar-parsing code (Andrew Dunstan, Tom Lane, Chao Li)

  The decompression and tar-file reading code used in pg_basebackup and pg_verifybackup mishandled tar-file padding data, could corrupt LZ4-compressed data in edge cases, failed to check for some unusual error conditions, failed to exit after compression/decompression errors (leading to cascading error reports), and leaked memory.

- In pg_upgrade, take care to use the correct protocol version when connecting to older source servers (Jacob Champion)

This could be problematic when attempting to upgrade from a pre-2018 server.

- In `contrib/basic_archive`, allow the archive directory to be missing at startup (Nathan Bossart)

  Previously, the setting of `basic_archive.archive_directory` was rejected if it didn't point to an existing directory. This is undesirable because archiving will be stuck indefinitely, even if the directory appears later.

- Fix `contrib/ltree` to cope when case-folding changes a string's byte length (Jeff Davis)

  Previously, `lquery` patterns specifying case-insensitive matching might fail to match labels they should match.

- In `contrib/pg_stat_statements`, don't leak memory if an error occurs while parsing the `pgss_query_texts.stat` file (Heikki Linnakangas)

- In `contrib/postgres_fdw`, avoid crash due to premature cleanup of a failed connection (Etsuro Fujita)

  If a remote connection fails abort cleanup, we can't use it any longer. But delay closing the connection object until end of transaction, because there might still be references to it within data structures such as open cursors.

- Update time zone data files to tzdata release 2026b (Tom Lane)

  British Columbia (America/Vancouver) will be on year-round UTC-07 (effectively, permanent DST) beginning in November 2026. This release assumes that their TZ abbreviation will be `MST` from that time forward. That seems likely to change, but it's unclear what new abbreviation will be used. Also a historical correction for Moldova: they have followed EU DST transition times since 2022.

E.3. Release 15.17

Release date: 2026-02-26

This release contains a small number of fixes from 15.16. For information about new features in major release 15, see Section E.20.

E.3.1. Migration to Version 15.17

A dump/restore is not required for those running 15.X.

However, if you are upgrading from a version earlier than 15.14, see Section E.6.

E.3.2. Changes

- Fix failure after replaying a multixid truncation record from WAL that was generated by an older minor version (Heikki Linnakangas)

  Erroneous logic for coping with the way that previous versions handled multixid wraparound led to replay failure, with messages like « could not access status of transaction ». A typical scenario in which this could occur is a standby server of the latest minor version consuming WAL from a primary server of an older version.

- Avoid incorrect complaint of invalid encoding when `substring()` is applied to « toasted » data (Noah Misch)

The fix for CVE-2026-2006 was too aggressive and could raise an error about an incomplete character in cases that are actually valid.

- Fix `pg_stat_get_backend_wait_event()` and `pg_stat_get_backend_wait_event_type()` to report values for auxiliary processes (Heikki Linnakangas)

  Previously these functions returned NULL for auxiliary processes, but that's inconsistent with the `pg_stat_activity` view.

- Fix casting a composite-type variable to a domain type when returning its value from a PL/pgSQL function (Tom Lane)

  If the variable's value is NULL, a « cache lookup failed for type 0 » error resulted.

- Fix potential null pointer dereference in `contrib/hstore`'s binary input function (Michael Paquier)

  `hstore`'s receive function crashed on input containing duplicate keys. `hstore` values generated by Postgres would never contain duplicate keys, so this mistake has gone unnoticed. The crash could be provoked by malicious or corrupted data.

E.4. Release 15.16

Release date: 2026-02-12

This release contains a variety of fixes from 15.15. For information about new features in major release 15, see Section E.20.

E.4.1. Migration to Version 15.16

A dump/restore is not required for those running 15.X.

However, if you are upgrading from a version earlier than 15.14, see Section E.6.

E.4.2. Changes

- Guard against unexpected dimensions of `oidvector`/`int2vector` (Tom Lane)

  These data types are expected to be 1-dimensional arrays containing no nulls, but there are cast pathways that permit violating those expectations. Add checks to some functions that were depending on those expectations without verifying them, and could misbehave in consequence.

  The PostgreSQL Project thanks Altan Birler for reporting this problem. (CVE-2026-2003)

- Harden selectivity estimators against being attached to operators that accept unexpected data types (Tom Lane)

  `contrib/intarray` contained a selectivity estimation function that could be abused for arbitrary code execution, because it did not check that its input was of the expected data type. Third-party extensions should check for similar hazards and add defenses using the technique intarray now uses. Since such extension fixes will take time, we now require superuser privilege to attach a non-built-in selectivity estimator to an operator.

  The PostgreSQL Project thanks Daniel Firer, as part of zeroday.cloud, for reporting this problem. (CVE-2026-2004)

- Fix buffer overrun in `contrib/pgcrypto`'s PGP decryption functions (Michael Paquier)

Decrypting a crafted message with an overlength session key caused a buffer overrun, with consequences as bad as arbitrary code execution.

The PostgreSQL Project thanks Team Xint Code, as part of zeroday.cloud, for reporting this problem. (CVE-2026-2005)

- Fix inadequate validation of multibyte character lengths (Thomas Munro, Noah Misch)

  Assorted bugs allowed an attacker able to issue crafted SQL to overrun string buffers, with consequences as bad as arbitrary code execution. After these fixes, applications may observe « invalid byte sequence for encoding » errors when string functions process invalid text that has been stored in the database.

  The PostgreSQL Project thanks Paul Gerste and Moritz Sanft, as part of zeroday.cloud, for reporting this problem. (CVE-2026-2006)

- Don't allow CTE references in sub-selects to determine semantic levels of aggregate functions (Tom Lane)

  This change undoes a change made two minor releases ago, instead throwing an error if a sub-select references a CTE that's below the semantic level that standard SQL rules would assign to the aggregate based on contained column references and aggregates. The attempted fix turned out to cause problems of its own, and it's unclear what to do instead. Since sub-selects within aggregates are disallowed altogether by the SQL standard, treating such cases as errors seems sufficient.

- Fix trigger transition table capture for `MERGE` in CTE queries (Dean Rasheed)

  When executing a data-modifying CTE query containing both a `MERGE` and another DML operation on a table with statement-level `AFTER` triggers, the transition tables passed to the triggers would not include the rows affected by the `MERGE`, only those affected by the other operation(s).

- Fix failure when all children of a partitioned target table of an update or delete have been pruned (Amit Langote)

  In such cases, the executor could report « could not find junk ctid column » errors, even though nothing needs to be done.

- Allow indexscans on partial hash indexes even when the index's predicate implies the truth of the WHERE clause (Tom Lane)

  Normally we drop a WHERE clause that is implied by the predicate, since it's pointless to test it; it must hold for every index entry. However that can prevent creation of an indexscan plan if the index is one that requires a WHERE clause on the leading index key, as hash indexes do. Don't drop implied clauses when considering such an index.

- Do not emit WAL for unlogged BRIN indexes (Kirill Reshke)

  One seldom-taken code path incorrectly emitted a WAL record relating to a BRIN index even if the index was marked unlogged. Crash recovery would then fail to replay that record, complaining that the file already exists.

- Prevent truncation of CLOG that is still needed by unread `NOTIFY` messages (Joel Jacobson, Heikki Linnakangas)

  This fix prevents « could not access status of transaction » errors when a backend is slow to absorb `NOTIFY` messages.

- Escalate errors occurring during `NOTIFY` message processing to FATAL, i.e. close the connection (Heikki Linnakangas)

Formerly, if a backend got an error while absorbing a `NOTIFY` message, it would advance past that message, report the error to the client, and move on. That behavior was fraught with problems though. One big concern is that the client has no good way to know that a notification was lost, and certainly no way to know what was in it. Depending on the application logic, missing a notification could cause the application to get stuck waiting, for example. Also, any remaining messages would not get processed until someone sent a new `NOTIFY`.

Also, if the connection is idle at the time of receiving a `NOTIFY` signal, any ERROR would be escalated to FATAL anyway, due to unrelated concerns. Therefore, we've chosen to make that happen in all cases, for consistency and to provide a clear signal to the application that it might have missed some notifications.

- Fix bug in following update chain when locking a tuple (Jasper Smit)

  This code path neglected to check the xmin of the first new tuple in the update chain, making it possible to lock an unrelated tuple if the original updater aborted and the space was immediately reclaimed by `VACUUM` and then re-used. That could cause unexpected transaction delays or deadlocks. Errors associated with having identified the wrong tuple have also been observed.

- Fix issues around in-place catalog updates (Noah Misch)

  Send a nontransactional invalidation message for an in-place update, since such an update will survive transaction rollback. Also ensure that the update is WAL-logged before other sessions can see it. These fixes primarily prevent scenarios in which relations' frozen-XID attributes become inconsistent, possibly allowing premature CLOG truncation and subsequent « could not access status of transaction » errors.

- Fix potential backend process crash at process exit due to trying to release a lock in an already-unmapped shared memory segment (Rahila Syed)

- Guard against incorrect truncation of the multixact log after a crash (Heikki Linnakangas)

- Fix possibly mis-encoded result of `pg_stat_get_backend_activity()` (Chao Li)

  The shared-memory buffer holding a session's activity string can end with an incomplete multibyte character. Readers are supposed to truncate off any such incomplete character, but this function failed to do so.

- Guard against recursive memory context logging (Fujii Masao)

  A constant flow of signals requesting memory context logging could cause recursive execution of the logging code, which in theory could lead to stack overflow.

- Fix memory context usage when reinitializing a parallel execution context (Jakub Wartak, Jeevan Chalke)

  This error could result in a crash due to a subsidiary data structure having a shorter lifespan than the parallel context. The problem is not known to be reachable using only core PostgreSQL, but we have reports of trouble in extensions.

- Set next multixid's offset when creating a new multixid, to remove the wait loop that was needed in corner cases (Andrey Borodin)

  The previous logic could get stuck waiting for an update that would never occur.

- Avoid rewriting data-modifying CTEs more than once (Bernice Southey, Dean Rasheed)

  Formerly, when updating an auto-updatable view or a relation with rules, if the original query had any data-modifying CTEs, the rewriter would rewrite those CTEs multiple times due to recursion.

This was inefficient and could produce false errors if a CTE included an update of an always-generated column.

- Fail recovery if WAL does not exist back to the redo point indicated by the checkpoint record (Nitin Jadhav)

  Add an explicit check for this before starting recovery, so that no harm is done and a useful error message is provided. Previously, recovery might crash or corrupt the database in this situation.

- Avoid scribbling on the source query tree during `ALTER PUBLICATION` (Sunil S)

  This error had the visible effect that an event trigger fired for the query would see only the first `publish` option, even if several had been specified. If such a query were set up as a prepared statement, re-executions would misbehave too.

- Pass connection options specified in `CREATE SUBSCRIPTION ... CONNECTION` to the publisher's walsender (Fujii Masao)

  Before this fix, the `options` connection option (if any) was ignored, thus for example preventing setting custom server parameter values in the walsender session. It was intended for that to work, and it did work before refactoring in PostgreSQL version 15 broke it, so restore the previous behavior.

- Prevent invalidation of newly created or newly synced replication slots (Zhijie Hou)

  A race condition with a concurrent checkpoint could allow WAL to be removed that is needed by the replication slot, causing the slot to immediately get marked invalid.

- Fix race condition in computing a replication slot's required xmin (Zhijie Hou)

  This could lead to the error « cannot build an initial slot snapshot as oldest safe xid follows snapshot's xmin ».

- During initial synchronization of a logical replication subscription, commit the addition of a `pg_replication_origin` entry before starting to copy data (Zhijie Hou)

  Previously, if the copy step failed, the new `pg_replication_origin` entry would be lost due to transaction rollback. This led to inconsistent state in shared memory.

- Fix possible failure with « unexpected data beyond EOF » during restart of a streaming replica server (Anthonin Bonnefoy)

- Fix erroneous tracking of column position when parsing partition range bounds (myzhen)

  This could, for example, lead to the wrong column name being cited in error messages about casting partition bound values to the column's data type.

- Fix assorted minor errors in error messages (Man Zeng, Tianchen Zhang)

  For example, an error report about mismatched timeline number in a backup manifest showed the starting timeline number where it meant to show the ending timeline number.

- Fix failure to perform function inlining when doing JIT compilation with LLVM version 17 or later (Anthonin Bonnefoy)

- Adjust our JIT code to work with LLVM 21 (Holger Hoffstätte)

  The previous coding failed to compile on aarch64 machines.

- Support process title changes on GNU/Hurd (Michael Banck)

- Make pg_resetwal print the updated value when changing OldestXID (Heikki Linnakangas)

It already did that for every other variable it can change.

- Make pg_resetwal allow setting next multixact xid to 0 or next multixact offset to UINT32_MAX (Maxim Orlov)

  These are valid values, so rejecting them was incorrect. In the worst case, if a pg_upgrade is attempted when exactly at the point of multixact wraparound, the upgrade would fail.

- In `contrib/amcheck`, use the correct snapshot for btree index parent checks (Mihail Nikalayeu)

  The previous coding caused spurious errors when examining indexes created with `CREATE INDEX CONCURRENTLY`.

- Fix `contrib/amcheck` to handle « half-dead » btree index pages correctly (Heikki Linnakangas)

  `amcheck` expected such a page to have a parent downlink, but it does not, leading to a false error report about « mismatch between parent key and child high key ».

- Fix `contrib/amcheck` to handle incomplete btree root page splits correctly (Heikki Linnakangas)

  `amcheck` could report a false error about « block is not true root ».

- Fix edge-case integer overflow in `contrib/intarray`'s selectivity estimator for `@@` (Chao Li)

  This could cause poor selectivity estimates to be produced for cases involving the maximum integer value.

- Fix multibyte-encoding issue in `contrib/ltree` (Jeff Davis)

  The previous coding could pass an incomplete multibyte character to `lower()`, probably resulting in incorrect behavior.

- Update time zone data files to tzdata release 2025c (Tom Lane)

  The only change is in historical data for pre-1976 timestamps in Baja California.

E.5. Release 15.15

Release date: 2025-11-13

This release contains a variety of fixes from 15.14. For information about new features in major release 15, see Section E.20.

E.5.1. Migration to Version 15.15

A dump/restore is not required for those running 15.X.

However, if you are upgrading from a version earlier than 15.14, see Section E.6.

E.5.2. Changes

- Check for `CREATE` privileges on the schema in `CREATE STATISTICS` (Jelte Fennema-Nio)

  This omission allowed table owners to create statistics in any schema, potentially leading to unexpected naming conflicts.

  The PostgreSQL Project thanks Jelte Fennema-Nio for reporting this problem. (CVE-2025-12817)

- Avoid integer overflow in allocation-size calculations within libpq (Jacob Champion)

Several places in libpq were not sufficiently careful about computing the required size of a memory allocation. Sufficiently large inputs could cause integer overflow, resulting in an undersized buffer, which would then lead to writing past the end of the buffer.

The PostgreSQL Project thanks Aleksey Solovev of Positive Technologies for reporting this problem. (CVE-2025-12818)

- Further fix processing of character classes within `SIMILAR TO` regular expressions (Laurenz Albe)

  The previous fix for translating `SIMILAR TO` pattern matching expressions to POSIX-style regular expressions broke a corner case that formerly worked: if there is an escape character right after the opening bracket and then a closing bracket right after the escape sequence (for example `[\w]`), the closing bracket was no longer seen as terminating the character class.

- Fix parsing of aggregate functions whose arguments contain a sub-select with a `FROM` reference to a CTE outside the aggregate function (Tom Lane)

  Such a CTE reference must act like a outer-level column reference when determining the aggregate's semantic level; but it was not being accounted for, leading to obscure planner or executor errors.

- Fix « no relation entry for relid » errors in corner cases while estimating SubPlan costs (Richard Guo)

- Avoid unlikely use-after-free in planner's expansion of partitioned tables (Bernd Reiß)

  There was a hazard only when the last live partition was concurrently dropped.

- Remove faulty assertion in btree index cleanup (Peter Geoghegan)

- Fix possible infinite loop in GIN index scans with multiple scan conditions (Tom Lane)

  GIN can handle scan conditions that can reject non-matching entries but are not useful for searching for relevant entries, for example a `tsquery` clause like `!term`. But such a condition must not be first in the array of scan conditions. The code failed to ensure that in all cases, with the result that a query having a mix of such conditions with normal conditions might work or not depending on the order in which the conditions were given in the query.

- Ensure that GIN index scans can be canceled (Tom Lane)

  Some code paths were capable of running for a long time without checking for interrupts.

- Ensure that BRIN autosummarization provides a snapshot for index expressions that need one (Álvaro Herrera)

  Previously, autosummarization would fail for such indexes, and then leave placeholder index tuples behind, causing the index to bloat over time.

- Fix integer-overflow hazard in BRIN index scans when the table contains close to 2^{32} pages (Sunil S)

  This oversight could result in an infinite loop or scanning of unneeded table pages.

- Fix incorrect zero-extension of stored values in JIT-generated tuple deforming code (David Rowley)

  When not using JIT, the equivalent code does sign-extension not zero-extension, leading to a different Datum representation of small integer data types. This inconsistency was masked in most cases, but it is known to lead to « could not find memoization table entry » errors when using Memoize plan nodes, and there might be other symptoms.

- Fix incorrect logic for caching result-relation information for triggers (David Rowley, Amit Langote)

In cases where partitions' column sets aren't physically identical to their parent partitioned tables' column sets, this oversight could lead to crashes.

- Add missing EvalPlanQual rechecks for TID Scan and TID Range Scan plan nodes (Sophie Alpert, David Rowley)

  This omission led to possibly not rechecking a condition on `ctid` during concurrent-update situations, causing the update's behavior to vary depending on which plan type had been selected.

- Fix EvalPlanQual handling of foreign or custom joins that do not have an alternative local-join plan prepared for EPQ (Masahiko Sawada, Etsuro Fujita)

  In such cases the foreign or custom access method should be invoked normally, but that did not happen, typically leading to a crash.

- Avoid duplicating hash partition constraints during `DETACH CONCURRENTLY` (Haiyang Li)

  `ALTER TABLE DETACH PARTITION CONCURRENTLY` was written to add a copy of the partitioning constraint to the now-detached partition. This was misguided, partially because non-concurrent `DETACH` doesn't do that, but mostly because in the case of hash partitioning the constraint expression contains references to the parent table's OID. That causes problems during dump/restore, or if the parent table is dropped after `DETACH`. In v19 and later, we'll no longer create any such copied constraints at all. In released branches, to minimize the risk of unforeseen consequences, only skip adding a copied constraint if it is for hash partitioning.

- Disallow generated columns in partition keys (Jian He, Ashutosh Bapat)

  This was already not allowed, but the check missed some cases, such as where the column reference is implicit in a whole-row reference.

- Disallow generated columns in `COPY ... FROM ... WHERE` clauses (Peter Eisentraut, Jian He)

  Previously, incorrect behavior or an obscure error message resulted from attempting to reference such a column, since generated columns have not yet been computed at the point where `WHERE` filtering is done.

- Fix visibility checking for statistics objects in `pg_temp` (Noah Misch)

  A statistics object located in a temporary schema cannot be named without schema qualification, but `pg_statistics_obj_is_visible()` missed that memo and could return « true » regardless. In turn, functions such as `pg_describe_object()` could fail to schema-qualify the object's name as expected.

- Fix `pg_event_trigger_dropped_objects()`'s reporting of temporary status (Antoine Violin, Tom Lane)

  If a dropped column default, trigger, or RLS policy belongs to a temporary table, report it with `is_temporary` true.

- Fix memory leakage in hashed subplans (Haiyang Li)

  Any memory consumed by the hash functions used for hashing tuples constituted a query-lifespan memory leak. One way that could happen is if the values being hashed require de-toasting.

- Fix minor memory leak during WAL replay of database creation (Nathan Bossart)

- Fix corruption of the shared statistics table after out-of-memory failures (Mikhail Kot)

  Previously, an out-of-memory failure partway through creating a new hash table entry left a broken entry behind, potentially causing errors in other sessions later.

- Fix concurrent update issue in `MERGE` (Yugo Nagata)

  When executing a `MERGE UPDATE` action, if there is more than one concurrent update of the target row, the lock-and-retry code would sometimes incorrectly identify the latest version of the target tuple, leading to incorrect results.

- Add missing replica identity checks in `MERGE` and `INSERT ... ON CONFLICT DO UPDATE` (Zhijie Hou)

  If `MERGE` may require update or delete actions, and the target table publishes updates or deletes, insist that it have a `REPLICA IDENTITY` defined. Failing to require this can silently break replication. Likewise, `INSERT` with an `UPDATE` option must require `REPLICA IDENTITY` if the target table publishes either inserts or updates.

- Avoid deadlock during `DROP SUBSCRIPTION` when publisher is on the same server as subscriber (Dilip Kumar)

- Fix incorrect reporting of replication lag in `pg_stat_replication` view (Fujii Masao)

  If any standby server's replay LSN stopped advancing, the `write_lag` and `flush_lag` columns would eventually stop updating.

- Avoid duplicative log messages about invalid `primary_slot_name` settings (Fujii Masao)

- Remove the unfinished slot state file after failing to write a replication slot's state to disk (Michael Paquier)

  Previously, a failure such as out-of-disk-space resulted in leaving a temporary `state.tmp` file behind. That's problematic because it would block all subsequent attempts to write the state, requiring manual intervention to clean up.

- Avoid unwanted WAL receiver shutdown when switching from streaming to archive WAL source (Xuneng Zhou)

  During a timeline change, a standby server's WAL receiver should remain alive, waiting for a new WAL streaming start point. Instead it was repeatedly shutting down and immediately getting restarted, which could confuse status monitoring code.

- Avoid failures in logical replication due to chance collisions of file numbers between regular and temporary tables (Vignesh C)

  This low-probability problem manifested as transient errors like « unexpected duplicate for tablespace *X*, relfilenode *Y* ». `contrib/autoprewarm` was also affected. A side-effect of the fix is that the SQL function `pg_filenode_relation()` will now ignore temporary tables.

- Fix use-after-free issue in the relation synchronization cache maintained by the pgoutput logical decoding plugin (Vignesh C, Masahiko Sawada)

  An error during logical decoding could result in crashes in subsequent logical decoding attempts in the same session. The case is only reachable when pgoutput is invoked via SQL functions.

- Avoid assertion failure when trying to release a replication slot in single-user mode (Hayato Kuroda)

- Fix incorrect printing of messages about failures in checking whether the user has Windows administrator privilege (Bryan Green)

  This code would have crashed or at least printed garbage. No such cases have been reported though, indicating that failure of these system calls is extremely rare.

- Avoid startup failure on macOS and BSD platforms when there is a collision with a pre-existing semaphore set (Tom Lane)

If the pre-existing set has fewer semaphores than we asked for, these platforms return EINVAL not EEXIST as our code expected, resulting in failure to start the database.

- Fix false memory-context-checking warnings in debug builds on 64-bit Windows (David Rowley)
- Correctly handle `GROUP BY DISTINCT` in PL/pgSQL assignment statements (Tom Lane)

  The parser failed to record the `DISTINCT` option in this context, so that the command would act as if it were plain `GROUP BY`.
- Avoid leaking memory when handling a SQL error within PL/Python (Tom Lane)

  This fixes a session-lifespan memory leak introduced in our previous minor releases.
- Fix libpq's trace output of characters with the high bit set (Ran Benita)

  On platforms where `char` is considered signed, the output included unsightly `\xffffff` decoration.
- Fix libpq's handling of socket-related errors on Windows within its GSSAPI logic (Ning Wu, Tom Lane)

  The code for encrypting/decrypting transmitted data using GSSAPI did not correctly recognize error conditions on the connection socket, since Windows reports those differently than other platforms. This led to failure to make such connections on Windows.
- In pg_dump, dump security labels on subscriptions and event triggers (Jian He, Fujii Masao)

  Labels on these types of objects were previously missed.
- Fix pg_dump's sorting of default ACLs and foreign key constraints (Kirill Reshke, Álvaro Herrera)

  Ensure consistent ordering of these database object types, as was already done for other object types.
- In pg_dump, label comments for separately-dumped domain constraints with the proper dependency (Noah Misch)

  This error could lead to parallel pg_restore attempting to create the comment before the constraint itself has been restored.
- In pg_restore, skip comments and security labels for publications and subscriptions that are not being restored (Jian He, Fujii Masao)

  Do not emit `COMMENT` or `SECURITY LABEL` commands for these objects when `--no-publications` or `--no-subscriptions` is specified.
- Fix assorted errors in the data compression logic in pg_dump and pg_restore (Daniel Gustafsson, Tom Lane)

  Error checking was missing or incorrect in several places, and there were also portability issues that would manifest on big-endian hardware. These problems had been missed because this code is only used to read compressed TOC files within directory-format dumps. pg_dump never produces such a dump; the case can be reached only by manually compressing the TOC file after the fact, which is a supported thing to do but very uncommon.
- Fix pgbench to error out cleanly if a `COPY` operation is started (Anthonin Bonnefoy)

  pgbench doesn't intend to support this case, but previously it went into an infinite loop.
- Fix pgbench's reporting of multiple errors (Yugo Nagata)

In cases where two successive `PQgetResult` calls both fail, pgbench might report the wrong error message.

- In pgbench, fix faulty assertion about errors in pipeline mode (Yugo Nagata)
- Ensure that `contrib/pg_buffercache` functions can be canceled (Satyanarayana Narlapuram, Yuhang Qiu)

  Some code paths were capable of running for a long time without checking for interrupts.
- Fix `contrib/pg_prewarm`'s privilege checks for indexes (Ayush Vatsa, Nathan Bossart)

  `pg_prewarm()` requires `SELECT` privilege on relations to be prewarmed. However, since indexes have no SQL privileges of their own, this resulted in non-superusers being unable to prewarm indexes. Instead, check for `SELECT` privilege on the index's table.
- Make `contrib/pgstattuple` more robust about empty or invalid index pages (Nitin Motiani)

  Count all-zero pages as free space, and ignore pages that are invalid according to a check of the page's special-space size. The code for btree indexes already counted all-zero pages as free, but the hash and gist code would error out, which has been found to be much less user-friendly. Similarly, make all three cases agree on ignoring corrupted pages rather than throwing errors.
- Harden our read and write barrier macros to satisfy Clang (Thomas Munro)

  We supposed that `__atomic_thread_fence()` is a sufficient barrier to prevent the C compiler from re-ordering memory accesses around it, but it appears that that's not true for Clang, allowing it to generate incorrect code for at least RISC-V, MIPS, and LoongArch machines. Add explicit compiler barriers to fix that.
- Fix building with LLVM version 21 and later (Holger Hoffstätte)
- Fix PGXS build infrastructure to support building NLS `po` files for extensions (Ryo Matsumura)

E.6. Release 15.14

Release date: 2025-08-14

This release contains a variety of fixes from 15.13. For information about new features in major release 15, see Section E.20.

E.6.1. Migration to Version 15.14

A dump/restore is not required for those running 15.X.

However, if you have any BRIN `numeric_minmax_multi_ops` indexes, it is advisable to reindex them after updating. See the fourth changelog entry below.

Also, if you are upgrading from a version earlier than 15.13, see Section E.7.

E.6.2. Changes

- Tighten security checks in planner estimation functions (Dean Rasheed)

  The fix for CVE-2017-7484, plus followup fixes, intended to prevent leaky functions from being applied to statistics data for columns that the calling user does not have permission to read. Two gaps in that protection have been found. One gap applies to partitioning and inheritance hierarchies where RLS policies on the tables should restrict access to statistics data, but did not.

The other gap applies to cases where the query accesses a table via a view, and the view owner has permissions to read the underlying table but the calling user does not have permissions on the view. The view owner's permissions satisfied the security checks, and the leaky function would get applied to the underlying table's statistics before we check the calling user's permissions on the view. This has been fixed by making security checks on views occur at the start of planning. That might cause permissions failures to occur earlier than before.

The PostgreSQL Project thanks Dean Rasheed for reporting this problem. (CVE-2025-8713)

- Prevent pg_dump scripts from being used to attack the user running the restore (Nathan Bossart)

  Since dump/restore operations typically involve running SQL commands as superuser, the target database installation must trust the source server. However, it does not follow that the operating system user who executes psql to perform the restore should have to trust the source server. The risk here is that an attacker who has gained superuser-level control over the source server might be able to cause it to emit text that would be interpreted as psql meta-commands. That would provide shell-level access to the restoring user's own account, independently of access to the target database.

  To provide a positive guarantee that this can't happen, extend psql with a `\restrict` command that prevents execution of further meta-commands, and teach pg_dump to issue that before any data coming from the source server.

  The PostgreSQL Project thanks Martin Rakhmanov, Matthieu Denais, and RyotaK for reporting this problem. (CVE-2025-8714)

- Convert newlines to spaces in names included in comments in pg_dump output (Noah Misch)

  Object names containing newlines offered the ability to inject arbitrary SQL commands into the output script. (Without the preceding fix, injection of psql meta-commands would also be possible this way.) CVE-2012-0868 fixed this class of problem at the time, but later work reintroduced several cases.

  The PostgreSQL Project thanks Noah Misch for reporting this problem. (CVE-2025-8715)

- Fix incorrect distance calculation in BRIN `numeric_minmax_multi_ops` support function (Peter Eisentraut, Tom Lane)

  The results were sometimes wrong on 64-bit platforms, and wildly wrong on 32-bit platforms. This did not produce obvious failures because the logic is only used to choose how to merge values into ranges; at worst the index would become inefficient and bloated. Nonetheless it's recommended to reindex any BRIN indexes that use the `numeric_minmax_multi_ops` operator class.

- Avoid regression in the size of XML input that we will accept (Michael Paquier, Erik Wienhold)

  Our workaround for a bug in early 2.13.x releases of libxml2 made use of a code path that rejects text chunks exceeding 10MB, whereas the previous coding did not. Those early releases are presumably extinct in the wild by now, so revert to the previous coding.

- Fix `MERGE` into a plain-inheritance parent table (Dean Rasheed)

  Insertions into such a target table could crash or produce incorrect query results due to failing to handle `WITH CHECK OPTION` and `RETURNING` actions.

- Allow tables with statement-level triggers to become partitions or inheritance children (Etsuro Fujita)

  We do not allow partitions or inheritance child tables to have row-level triggers with transition tables, because an operation on the whole inheritance tree would need to maintain a separate transition table for each such child table. But that problem does not apply for statement-level triggers, because only the parent's statement-level triggers will be fired. The code that checks

whether an existing table can become a partition or inheritance child nonetheless rejected both kinds of trigger.

- Disallow collecting transition tuples from child foreign tables (Etsuro Fujita)

  We do not support triggers with transition tables on foreign tables. However, the case of a partition or inheritance child that is a foreign table was overlooked. If the parent has such a trigger, incorrect transition tuples were collected from the foreign child. Instead throw an error, reporting that the case is not supported.

- Allow resetting unknown custom parameters with reserved prefixes (Nathan Bossart)

  Previously, if a parameter setting had been stored using `ALTER DATABASE/ROLE/SYSTEM`, the stored setting could not be removed if the parameter was unknown but had a reserved prefix. This case could arise if an extension used to have a parameter, but that parameter had been removed in an upgrade.

- Fix a potential deadlock during `ALTER SUBSCRIPTION ... DROP PUBLICATION` (Ajin Cherian)

  Ensure that server processes acquire catalog locks in a consistent order during replication origin drops.

- Shorten the race condition window for creating indexes with conflicting names (Tom Lane)

  When choosing an auto-generated name for an index, avoid conflicting with not-yet-committed `pg_class` rows as well as fully-valid ones. This avoids possibly choosing the same name as some concurrent `CREATE INDEX` did, when that command is still in process of filling its index, or is done but is part of a not-yet-committed transaction. There's still a window for trouble, but it's only as long as the time needed to validate a new index's parameters and insert its `pg_class` row.

- Prevent usage of incorrect `VACUUM` options in some cases where multiple tables are vacuumed in a single command (Nathan Bossart, Michael Paquier)

  The `TRUNCATE` and `INDEX_CLEANUP` options of one table could be applied to others.

- Fix processing of character classes within `SIMILAR TO` regular expressions (Laurenz Albe)

  The code that translates `SIMILAR TO` pattern matching expressions to POSIX-style regular expressions did not consider that square brackets can be nested. For example, in a pattern like `[[:alpha:]%_]`, the code treated the `%` and `_` characters as metacharacters when they should be literals.

- When deparsing queries, always add parentheses around the expression in `FETCH FIRST` *`expression`* `ROWS WITH TIES` clauses (Heikki Linnakangas)

  This avoids some cases where the deparsed result wasn't syntactically valid.

- Limit the checkpointer process's fsync request queue size (Alexander Korotkov, Xuneng Zhou)

  With very large `shared_buffers` settings, it was possible for the checkpointer to attempt to allocate more than 1GB for fsync requests, leading to failure and an infinite loop. Clamp the queue size to prevent this scenario.

- Avoid infinite wait in logical decoding when reading a partially-written WAL record (Vignesh C)

  If the server crashes after writing the first part of a WAL record that would span multiple pages, subsequent logical decoding of the WAL stream would wait for data to arrive on the next WAL page. That might never happen if the server is now idle.

- Fix inconsistent quoting of role names in ACL strings (Tom Lane)

The previous quoting rule was locale-sensitive, which could lead to portability problems when transferring `aclitem` values across installations. (pg_dump does not do that, but other tools might.) To ensure consistency, always quote non-ASCII characters in `aclitem` output; but to preserve backward compatibility, never require that they be quoted during `aclitem` input.

- Reject equal signs (=) in the names of relation options and foreign-data options (Tom Lane)

  There's no evident use-case for option names like this, and allowing them creates ambiguity in the stored representation.

- Fix potentially-incorrect decompression of LZ4-compressed archive data (Mikhail Gribkov)

  This error seems to manifest only with not-very-compressible input data, which may explain why it escaped detection.

- Avoid a rare scenario where a btree index scan could mark the wrong index entries as dead (Peter Geoghegan)

- Avoid re-distributing cache invalidation messages from other transactions during logical replication (vignesh C)

  Our previous round of minor releases included a bug fix to ensure that replication receiver processes would respond to cross-process cache invalidation messages, preventing them from using stale catalog data while performing replication updates. However, the fix unintentionally made them also redistribute those messages again, leading to an exponential increase in the number of invalidation messages, which would often end in a memory allocation failure. Fix by not redistributing received messages.

- Avoid premature removal of old WAL during checkpoints (Vitaly Davydov)

  If a replication slot's restart point is advanced while a checkpoint is in progress, no-longer-needed WAL segments could get removed too soon, leading to recovery failure if the database crashes immediately afterwards. Fix by keeping them for one additional checkpoint cycle.

- Never move a replication slot's confirmed-flush position backwards (Shveta Malik)

  In some cases a replication client could acknowledge an LSN that's past what it has stored persistently, and then perhaps send an older LSN after a restart. We consider this not-a-bug so long as the client did not have anything it needed to do for the WAL between the two points. However, we should not re-send that WAL for fear of data duplication, so make sure we always believe the latest confirmed LSN for a given slot.

- Allow waiting for a transaction on a standby server to be interrupted (Kevin K Biju)

  Creation of a replication slot on a standby server may require waiting for some active transaction(s) to finish on the primary and then be replayed on the standby. Since that could be an indefinite wait, it's desirable to allow the operation to be cancelled, but there was no check for query cancel in the loop.

- Fix per-relation memory leakage in autovacuum (Tom Lane)

- Fix some places that might try to fetch toasted fields of system catalogs without any snapshot (Nathan Bossart)

  This could result in an assertion failure or « cannot fetch toast data without an active snapshot » error.

- Avoid assertion failure during cross-table constraint updates (Tom Lane, Jian He)

- Remove faulty assertion that a command tag must have been determined by the end of `PortalRunMulti()` (Álvaro Herrera)

This failed in edge cases such as an empty prepared statement.

- Fix assertion failure in `XMLTABLE` parsing (Richard Guo)

- Restore the ability to run PL/pgSQL expressions in parallel (Dipesh Dhameliya)

  PL/pgSQL's notion of an « expression » is very broad, encompassing any SQL `SELECT` query that returns a single column and no more than one row. So there are cases, for example evaluation of an aggregate function, where the query involves significant work and it'd be useful to run it with parallel workers. This used to be possible, but a previous bug fix unintentionally disabled it.

- Fix edge-case resource leaks in PL/Python error reporting (Tom Lane)

  An out-of-memory failure while reporting an error from Python could result in failure to drop reference counts on Python objects, leading to session-lifespan memory leakage.

- Fix libpq's `PQport()` function to never return NULL unless the passed connection is NULL (Daniele Varrazzo)

  This is the documented behavior, but recent libpq versions would return NULL in some cases where the user had not provided a port specification. Revert to our historical behavior of returning an empty string in such cases. (v18 and later will return the compiled-in default port number, typically `"5432"`, instead.)

- Avoid failure when GSSAPI authentication requires packets larger than 16kB (Jacob Champion, Tom Lane)

  Larger authentication packets are needed for Active Directory users who belong to many AD groups. This limitation manifested in connection failures with unintelligible error messages, typically « GSSAPI context establishment error: The routine must be called again to complete its function: Unknown error ».

- Fix timing-dependent failures in SSL and GSSAPI data transmission (Tom Lane)

  When using SSL or GSSAPI encryption in non-blocking mode, libpq sometimes failed with « SSL error: bad length » or « GSSAPI caller failed to retransmit all data needing to be retried ».

- Avoid null-pointer dereference during connection lookup in ecpg applications (Aleksander Alekseev)

  The case could occur only if the application has some connections that are named and some that are not.

- Improve psql's tab completion for `COPY` and `\copy` options (Atsushi Torikoshi)

  The same completions were offered for both `COPY FROM` and `COPY TO`, although some options are only valid for one case or the other. Distinguish these cases to provide more accurate suggestions.

- Avoid assertion failure in pgbench when multiple pipeline sync messages are received (Fujii Masao)

- Ensure that pg_dump dumps comments on domain constraints in a valid order (Jian He)

  In some cases the comment command could appear before creation of the constraint.

- Ensure stable sort ordering in pg_dump for all types of database objects (Noah Misch, Andreas Karlsson)

  pg_dump sorts objects by their logical names before performing dependency-driven reordering. This sort did not account for the full unique key identifying certain object types such as rules and constraints, and thus it could produce dissimilar sort orders for logically-identical databases. That

made it difficult to compare databases by diff'ing pg_dump output, so improve the logic to ensure stable sort ordering in all cases.

- In pg_upgrade, check for inconsistent inherited not-null constraints (Ali Akbar)

  PostgreSQL versions before 18 allow an inherited column not-null constraint to be dropped. However, this results in a schema that cannot be restored, leading to failure in pg_upgrade. Detect such cases during pg_upgrade's preflight checks to allow users to fix them before initiating the upgrade.

- Avoid assertion failure if `track_commit_timestamp` is enabled during initdb (Hayato Kuroda, Andy Fan)

- Fix pg_waldump to show information about dropped statistics in `PREPARE TRANSACTION` WAL records (Daniil Davydov)

- Avoid possible leak of the open connection during `contrib/dblink` connection establishment (Tom Lane)

  In the rare scenario where we hit out-of-memory while inserting the new connection object into dblink's hashtable, the open connection would be leaked until end of session, leaving an idle session sitting on the remote server.

- Make `contrib/pg_prewarm` cope with very large `shared_buffers` settings (Daria Shanina)

  Autoprewarm failed with a memory allocation error if `shared_buffers` was larger than about 50 million buffers (400GB).

- In `contrib/pg_stat_statements`, avoid leaving gaps in the set of parameter numbers used in a normalized query (Sami Imseih)

- Fix memory leakage in `contrib/postgres_fdw`'s DirectModify methods (Tom Lane)

  The `PGresult` holding the results of the remote modify command would be leaked for the rest of the session if the query fails between invocations of the DirectModify methods, which could happen when there's `RETURNING` data to process.

- Ensure that directories listed in configure's `--with-includes` and `--with-libraries` options are searched before system-supplied directories (Tom Lane)

  A common reason for using these options is to allow a user-built version of some library to override the system-supplied version. However, that failed to work in some environments because of careless ordering of switches in the commands issued by the makefiles.

- Fix configure's checks for `__cpuid()` and `__cpuidex()` (Lukas Fittl, Michael Paquier)

  configure failed to detect these Windows-specific functions, so that they would not be used, leading to slower-than-necessary CRC computations since the availability of hardware instructions could not be verified. The practical impact of this error was limited, because production builds for Windows typically do not use the Autoconf toolchain.

- Fix build failure with `--with-pam` option on Solaris-based platforms (Tom Lane)

  Solaris is inconsistent with other Unix platforms about the API for PAM authentication. This manifested as an « inconsistent pointer » compiler warning, which we never did anything about. But as of GCC 14 it's an error not warning by default, so fix it.

- Make our code portable to GNU Hurd (Michael Banck, Christoph Berg, Samuel Thibault)

  Fix assumptions about `IOV_MAX` and `O_RDONLY` that don't hold on Hurd.

- Make our usage of `memset_s()` conform strictly to the C11 standard (Tom Lane)

  This avoids compile failures on some platforms.

- Prevent uninitialized-value compiler warnings in JSONB comparison code (Tom Lane)

- Avoid deprecation warnings when building with libxml2 2.14 and later (Michael Paquier)

- Avoid problems when compiling `pg_locale.h` under C++ (John Naylor)

  PostgreSQL header files generally need to be wrapped in `extern "C" { ... }` in order to be included in extensions written in C++. This failed for `pg_locale.h` because of its use of libicu headers, but we can work around that by suppressing C++-only declarations in those headers. C++ extensions that want to use libicu's C++ APIs can do so by including the libicu headers ahead of `pg_locale.h`.

E.7. Release 15.13

Release date: 2025-05-08

This release contains a variety of fixes from 15.12. For information about new features in major release 15, see Section E.20.

E.7.1. Migration to Version 15.13

A dump/restore is not required for those running 15.X.

However, if you have any self-referential foreign key constraints on partitioned tables, it may be necessary to recreate those constraints to ensure that they are being enforced correctly. See the second changelog entry below.

Also, if you have any BRIN bloom indexes, it may be advisable to reindex them after updating. See the third changelog entry below.

Also, if you are upgrading from a version earlier than 15.9, see Section E.11.

E.7.2. Changes

- Avoid one-byte buffer overread when examining invalidly-encoded strings that are claimed to be in GB18030 encoding (Noah Misch, Andres Freund)

  While unlikely, a SIGSEGV crash could occur if an incomplete multibyte character appeared at the end of memory. This was possible both in the server and in libpq-using applications. (CVE-2025-4207)

- Handle self-referential foreign keys on partitioned tables correctly (Álvaro Herrera)

  Creating or attaching partitions failed to make the required catalog entries for a foreign-key constraint, if the table referenced by the constraint was the same partitioned table. This resulted in failure to enforce the constraint fully.

  To fix this, you should drop and recreate any self-referential foreign keys on partitioned tables, if partitions have been created or attached since the constraint was created. Bear in mind that violating rows might already be present, in which case recreating the constraint will fail, and you'll need to fix up those rows before trying again.

- Avoid data loss when merging compressed BRIN summaries in `brin_bloom_union()` (Tomas Vondra)

The code failed to account for decompression results not being identical to the input objects, which would result in failure to add some of the data to the merged summary, leading to missed rows in index searches.

This mistake was present back to v14 where BRIN bloom indexes were introduced, but this code path was only rarely reached then. It's substantially more likely to be hit in v17 because parallel index builds now use the code.

- Fix unexpected « attribute has wrong type » errors in `UPDATE`, `DELETE`, and `MERGE` queries that use whole-row table references to views or functions in `FROM` (Tom Lane)

- Fix `MERGE` into a partitioned table with `DO NOTHING` actions (Tender Wang)

  Some cases failed with « unknown action in MERGE WHEN clause » errors.

- Prevent failure in `INSERT` commands when the table has a `GENERATED` column of a domain data type and the domain's constraints disallow null values (Jian He)

  Constraint failure was reported even if the generation expression produced a perfectly okay result.

- Correctly process references to outer CTE names that appear within a `WITH` clause attached to an `INSERT`/`UPDATE`/`DELETE`/`MERGE` command that's inside `WITH` (Tom Lane)

  The parser failed to detect disallowed recursion cases, nor did it account for such references when sorting CTEs into a usable order.

- Fix `ARRAY(`*`subquery`*`)` and `ARRAY[`*`expression, ...`*`]` constructs to produce sane results when the input is of type `int2vector` or `oidvector` (Tom Lane)

  This patch restores the behavior that existed before PostgreSQL 9.5: the result is of type `int2vector[]` or `oidvector[]`.

- Fix possible erroneous reports of invalid affixes while parsing Ispell dictionaries (Jacob Brazeal)

- Fix `ALTER TABLE ADD COLUMN` to correctly handle the case of a domain type that has a default (Jian He, Tom Lane, Tender Wang)

  If a domain type has a default, adding a column of that type (without any explicit `DEFAULT` clause) failed to install the domain's default value in existing rows, instead leaving the new column null.

- Repair misbehavior when there are duplicate column names in a foreign key constraint's `ON DELETE SET DEFAULT` or `SET NULL` action (Tom Lane)

- Improve the error message for disallowed attempts to alter the properties of a foreign key constraint (Álvaro Herrera)

- Avoid error when resetting the `relhassubclass` flag of a temporary table that's marked `ON COMMIT DELETE ROWS` (Noah Misch)

- Fix planner's failure to identify more than one hashable ScalarArrayOpExpr subexpression within a top-level expression (David Geier)

  This resulted in unnecessarily-inefficient execution of any additional subexpressions that could have been processed with a hash table (that is, `IN`, `NOT IN`, or `= ANY` clauses with all-constant right-hand sides).

- Disable « skip fetch » optimization in bitmap heap scan (Matthias van de Meent)

  It turns out that this optimization can result in returning dead tuples when a concurrent vacuum marks a page all-visible.

- Fix performance issues in GIN index search startup when there are many search keys (Tom Lane, Vinod Sridharan)

  An indexable clause with many keys (for example, `jsonbcol ?| array[...]` with tens of thousands of array elements) took $O(N^2)$ time to start up, and was uncancelable for that interval too.

- Detect missing support procedures in a BRIN index operator class, and report an error instead of crashing (Álvaro Herrera)

- Respond to interrupts (such as query cancel) while waiting for asynchronous subplans of an Append plan node (Heikki Linnakangas)

  Previously, nothing would happen until one of the subplans becomes ready.

- Fix race condition in handling of `synchronous_standby_names` immediately after startup (Melnikov Maksim, Michael Paquier)

  For a short period after system startup, backends might fail to wait for synchronous commit even though `synchronous_standby_names` is enabled.

- Fix `pg_strtof()` to not crash with null endptr (Alexander Lakhin, Tom Lane)

- Avoid crash when a Snowball stemmer encounters an out-of-memory condition (Maksim Korotkov)

- Prevent over-advancement of catalog xmin in « fast forward » mode of logical decoding (Zhijie Hou)

  This mistake could allow deleted catalog entries to be vacuumed away even though they were still potentially needed by the WAL-reading process.

- Avoid data loss when DDL operations that don't take a strong lock affect tables that are being logically replicated (Shlok Kyal, Hayato Kuroda)

  The catalog changes caused by the DDL command were not reflected into WAL-decoding processes, allowing them to decode subsequent changes using stale catalog data, probably resulting in data corruption.

- Avoid duplicate snapshot creation in logical replication index lookups (Heikki Linnakangas)

- Fix wrong checkpoint details in error message about incorrect recovery timeline choice (David Steele)

  If the requested recovery timeline is not reachable, the reported checkpoint and timeline should be the values read from the backup_label, if there is one. This message previously reported values from the control file, which is correct when recovering from the control file without a backup_label, but not when there is a backup_label.

- Fix assertion failure in snapshot building (Masahiko Sawada)

- Remove incorrect assertion in `pgstat_report_stat()` (Michael Paquier)

- Fix overly-strict assertion in `gistFindCorrectParent()` (Heikki Linnakangas)

- Fix rare assertion failure in standby servers when the primary is restarted (Heikki Linnakangas)

- In PL/pgSQL, avoid « unexpected plan node type » error when a scrollable cursor is defined on a simple `SELECT` *`expression`* query (Andrei Lepikhov)

- Don't try to drop individual index partitions in pg_dump's `--clean` mode (Jian He)

  The server rejects such `DROP` commands. That has no real consequences, since the partitions will go away anyway in the subsequent `DROP`s of either their parent tables or their partitioned index.

However, the error reported for the attempted drop causes problems when restoring in `--single-transaction` mode.

- In pg_dumpall, avoid emitting invalid role `GRANT` commands if `pg_auth_members` contains invalid role OIDs (Tom Lane)

  Instead, print a warning and skip the entry. This copes better with catalog corruption that has been seen to occur in back branches as a result of race conditions between `GRANT` and `DROP ROLE`.

- In pg_amcheck and pg_upgrade, use the correct function to free allocations made by libpq (Michael Paquier, Ranier Vilela)

  These oversights could result in crashes in certain Windows build configurations, such as a debug build of libpq used by a non-debug build of the calling application.

- Allow `contrib/dblink` queries to be interrupted by query cancel (Noah Misch)

  This change back-patches a v17-era fix. It prevents possible hangs in `CREATE DATABASE` and `DROP DATABASE` due to failure to detect deadlocks.

- Avoid crashing with corrupt input data in `contrib/pageinspect`'s `heap_page_items()` (Dmitry Kovalenko)

- Prevent assertion failure in `contrib/pg_freespacemap`'s `pg_freespacemap()` (Tender Wang)

  Applying `pg_freespacemap()` to a relation lacking storage (such as a view) caused an assertion failure, although there was no ill effect in non-assert builds. Add an error check to reject that case.

- Fix build failure on macOS 15.4 (Tom Lane, Peter Eisentraut)

  This macOS update broke our configuration probe for `strchrnul()`.

- Update time zone data files to tzdata release 2025b for DST law changes in Chile, plus historical corrections for Iran (Tom Lane)

  There is a new time zone America/Coyhaique for Chile's Aysén Region, to account for it changing to UTC-03 year-round and thus diverging from America/Santiago.

E.8. Release 15.12

Release date: 2025-02-20

This release contains a few fixes from 15.11. For information about new features in major release 15, see Section E.20.

E.8.1. Migration to Version 15.12

A dump/restore is not required for those running 15.X.

However, if you are upgrading from a version earlier than 15.9, see Section E.11.

E.8.2. Changes

- Improve behavior of libpq's quoting functions (Andres Freund, Tom Lane)

  The changes made for CVE-2025-1094 had one serious oversight: `PQescapeLiteral()` and `PQescapeIdentifier()` failed to honor their string length parameter, instead always reading

to the input string's trailing null. This resulted in including unwanted text in the output, if the caller intended to truncate the string via the length parameter. With very bad luck it could cause a crash due to reading off the end of memory.

In addition, modify all these quoting functions so that when invalid encoding is detected, an invalid sequence is substituted for just the first byte of the presumed character, not all of it. This reduces the risk of problems if a calling application performs additional processing on the quoted string.

E.9. Release 15.11

Release date: 2025-02-13

This release contains a variety of fixes from 15.10. For information about new features in major release 15, see Section E.20.

E.9.1. Migration to Version 15.11

A dump/restore is not required for those running 15.X.

However, if you are upgrading from a version earlier than 15.9, see Section E.11.

E.9.2. Changes

- Harden `PQescapeString` and allied functions against invalidly-encoded input strings (Andres Freund, Noah Misch)

  Data-quoting functions supplied by libpq now fully check the encoding validity of their input. If invalid characters are detected, they report an error if possible. For the ones that lack an error return convention, the output string is adjusted to ensure that the server will report invalid encoding and no intervening processing will be fooled by bytes that might happen to match single quote, backslash, etc.

  The purpose of this change is to guard against SQL-injection attacks that are possible if one of these functions is used to quote crafted input. There is no hazard when the resulting string is sent directly to a PostgreSQL server (which would check its encoding anyway), but there is a risk when it is passed through psql or other client-side code. Historically such code has not carefully vetted encoding, and in many cases it's not clear what it should do if it did detect such a problem.

  This fix is effective only if the data-quoting function, the server, and any intermediate processing agree on the character encoding that's being used. Applications that insert untrusted input into SQL commands should take special care to ensure that that's true.

  Applications and drivers that quote untrusted input without using these libpq functions may be at risk of similar problems. They should first confirm the data is valid in the encoding expected by the server.

  The PostgreSQL Project thanks Stephen Fewer for reporting this problem. (CVE-2025-1094)

- Exclude parallel workers from connection privilege checks and limits (Tom Lane)

  Do not check `datallowconn`, `rolcanlogin`, and `ACL_CONNECT` privileges when starting a parallel worker, instead assuming that it's enough for the leader process to have passed similar checks originally. This avoids, for example, unexpected failures of parallelized queries when the leader is running as a role that lacks login privilege. In the same vein, enforce `ReservedConnections`, `datconnlimit`, and `rolconnlimit` limits only against regular backends, and count only regular backends while checking if the limits were already reached. Those limits are meant to prevent excessive consumption of process slots for regular backends --- but

parallel workers and other special processes have their own pools of process slots with their own limit checks.

- Fix possible re-use of stale results in window aggregates (David Rowley)

  A window aggregate with a « run condition » optimization and a pass-by-reference result type might incorrectly return the result from the previous partition instead of performing a fresh calculation.

- Keep `TransactionXmin` in sync with `MyProc->xmin` (Heikki Linnakangas)

  This oversight could permit a process to try to access data that had already been vacuumed away. One known consequence is transient « could not access status of transaction » errors.

- Fix race condition that could cause failure to add a newly-inserted catalog entry to a catalog cache list (Heikki Linnakangas)

  This could result, for example, in failure to use a newly-created function within an existing session.

- Prevent possible catalog corruption when a system catalog is vacuumed concurrently with an update (Noah Misch)

- Fix data corruption when relation truncation fails (Thomas Munro)

  The filesystem calls needed to perform relation truncation could fail, leaving inconsistent state on disk (for example, effectively reviving deleted data). We can't really prevent that, but we can recover by dint of making such failures into PANICs, so that consistency is restored by replaying from WAL up to just before the attempted truncation. This isn't a hugely desirable behavior, but such failures are rare enough that it seems an acceptable solution.

- Prevent checkpoints from starting during relation truncation (Robert Haas)

  This avoids a race condition wherein the modified file might not get fsync'd before completing the checkpoint, creating a risk of data corruption if the operating system crashes soon after.

- Use `rename()` not `link()`/`unlink()` to rename files (Nathan Bossart)

  The previous coding was intended to assure that the operation could not accidentally overwrite an existing file. However a failure could leave two links to the same file in existence, confusing subsequent operations and creating a risk of data corruption. In practice we do not use this functionality in places where the target filename could already exist, so it seems better to give up the no-overwrite guarantee to remove the multiple-link hazard.

- Avoid possibly losing an update of `pg_database.datfrozenxid` when `VACUUM` runs concurrently with a `REASSIGN OWNED` that changes that database's owner (Kirill Reshke)

- Fix incorrect `tg_updatedcols` values passed to `AFTER UPDATE` triggers (Tom Lane)

  In some cases the `tg_updatedcols` bitmap could describe the set of columns updated by an earlier command in the same transaction, fooling the trigger into doing the wrong thing.

  Also, prevent memory bloat caused by making too many copies of the `tg_updatedcols` bitmap.

- Fix detach of a partition that has its own foreign-key constraint referencing a partitioned table (Amul Sul)

  In common cases, foreign keys are defined on a partitioned table's top level; but if instead one is defined on a partition and references a partitioned table, and the referencing partition is detached, the relevant `pg_constraint` entries were updated incorrectly. This led to errors like « could not find ON INSERT check triggers of foreign key constraint ».

- Fix mis-processing of `to_timestamp`'s `FF`*n* format codes (Tom Lane)

An integer format code immediately preceding FF*n* would consume all available digits, leaving none for FF*n*.

- When deparsing an XMLTABLE() expression, ensure that XML namespace names are double-quoted when necessary (Dean Rasheed)

- Include the ldapscheme option in pg_hba_file_rules() output (Laurenz Albe)

- Don't merge UNION operations if their column collations aren't consistent (Tom Lane)

  Previously we ignored collations when deciding if it's safe to merge UNION steps into a single N-way UNION operation. This was arguably valid before the introduction of nondeterministic collations, but it's not anymore, since the collation in use can affect the definition of uniqueness.

- Fix missed expression processing for partition pruning steps (Tom Lane)

  This oversight could lead to « unrecognized node type » errors, and perhaps other problems, in queries accessing partitioned tables.

- Allow dshash tables to grow past 1GB (Matthias van de Meent)

  This avoids errors like « invalid DSA memory alloc request size ». The case can occur for example in transactions that process several million tables.

- Avoid possible integer overflow in bringetbitmap() (James Hunter, Evgeniy Gorbanyov)

  Since the result is only used for statistical purposes, the effects of this error were mostly cosmetic.

- Prevent streaming standby servers from looping infinitely when reading a WAL record that crosses pages (Kyotaro Horiguchi, Alexander Kukushkin)

  This would happen when the record's continuation is on a page that needs to be read from a different WAL source.

- Fix unintended promotion of FATAL errors to PANIC during early process startup (Noah Misch)

  This fixes some unlikely cases that would result in « PANIC: proc_exit() called in child process ».

- Fix cases where an operator family member operator or support procedure could become a dangling reference (Tom Lane)

  In some cases a data type could be dropped while references to its OID still remain in pg_amop or pg_amproc. While that caused no immediate issues, an attempt to drop the owning operator family would fail, and pg_dump would produce bogus output when dumping the operator family. This fix causes creation and modification of operator families/classes to add needed dependency entries so that dropping a data type will also drop any dependent operator family elements. That does not help vulnerable pre-existing operator families, though, so a band-aid has also been added to DROP OPERATOR FAMILY to prevent failure when dropping a family that has dangling members.

- Fix multiple memory leaks in logical decoding output (Vignesh C, Masahiko Sawada, Boyu Yang)

- Avoid low-probability crash on out-of-memory, due to missing check for failure return from malloc() (Karina Litskevich)

- Avoid integer overflow while testing wal_skip_threshold condition (Tom Lane)

  A transaction that created a very large relation could mistakenly decide to ensure durability by copying the relation into WAL instead of fsync'ing it, thereby negating the point of wal_skip_threshold. (This only matters when wal_level is set to minimal, else a WAL copy is required anyway.)

- Fix unsafe order of operations during cache lookups (Noah Misch)

  The only known consequence was a usually-harmless « you don't own a lock of type ExclusiveLock » warning during `GRANT TABLESPACE`.

- Fix possible « failed to resolve name » failures when using JIT on older ARM platforms (Thomas Munro)

  This could occur as a consequence of inconsistency about the default setting of `-moutline-atomics` between gcc and clang. At least Debian and Ubuntu are known to ship gcc and clang compilers that target armv8-a but differ on the use of outline atomics by default.

- Fix handling of Windows junction points that are not of PostgreSQL origin (Thomas Munro)

  Previously, initdb would fail if the path to the data directory included junction points whose expansion isn't in « drive absolute » format, or whose expansion points to another junction point.

- Fix assertion failure in `WITH RECURSIVE ... UNION` queries (David Rowley)

- Avoid assertion failure in rule deparsing if a set operation leaf query contains set operations (Man Zeng, Tom Lane)

- Avoid edge-case assertion failure in parallel query startup (Tom Lane)

- Fix assertion failure at shutdown when writing out the statistics file (Michael Paquier)

- In `NULLIF()`, avoid passing a read-write expanded object pointer to the data type's equality function (Tom Lane)

  The equality function could modify or delete the object if it's given a read-write pointer, which would be bad if we decide to return it as the `NULLIF()` result. There is probably no problem with any built-in equality function, but it's easy to demonstrate a failure with one coded in PL/pgSQL.

- Ensure that expression preprocessing is applied to a default null value in `INSERT` (Tom Lane)

  If the target column is of a domain type, the planner must insert a coerce-to-domain step not just a null constant, and this expression missed going through some required processing steps. There is no known consequence with domains based on core data types, but in theory an error could occur with domains based on extension types.

- Repair memory leaks in PL/Python (Mat Arye, Tom Lane)

  Repeated use of `PLyPlan.execute` or `plpy.cursor` resulted in memory leakage for the duration of the calling PL/Python function.

- Fix PL/Tcl to compile with Tcl 9 (Peter Eisentraut)

- In the ecpg preprocessor, fix possible misprocessing of cursors that reference out-of-scope variables (Tom Lane)

- In ecpg, fix compile-time warnings about unsupported use of `COPY ... FROM STDIN` (Ryo Kanbayashi)

  Previously, the intended warning was not issued due to a typo.

- Fix psql to safely handle file path names that are encoded in SJIS (Tom Lane)

  Some two-byte characters in SJIS have a second byte that is equal to ASCII backslash (\). These characters were corrupted by path name normalization, preventing access to files whose names include such characters.

- Fix use of wrong version of `pqsignal()` in pgbench and psql (Fujii Masao, Tom Lane)

  This error could lead to misbehavior when using the `-T` option in pgbench or the `\watch` command in psql, due to interrupted system calls not being resumed as expected.

- Fix misexecution of some nested `\if` constructs in pgbench (Michail Nikolaev)

  An `\if` command appearing within a false (not-being-executed) `\if` branch was incorrectly treated the same as `\elif`.

- In pgbench, fix possible misdisplay of progress messages during table initialization (Yushi Ogiwara, Tatsuo Ishii, Fujii Masao)

- Make pg_controldata more robust against corrupted `pg_control` files (Ilyasov Ian, Anton Voloshin)

  Since pg_controldata will attempt to print the contents of `pg_control` even if the CRC check fails, it must take care not to misbehave for invalid field values. This patch fixes some issues triggered by invalid timestamps and apparently-negative WAL segment sizes.

- Fix possible crash in pg_dump with identity sequences attached to tables that are extension members (Tom Lane)

- Fix pg_basebackup to correctly handle `pg_wal.tar` files exceeding 2GB on Windows (Davinder Singh, Thomas Munro)

- Update configuration probes that determine the compiler switches needed to access ARM CRC instructions (Tom Lane)

  On ARM platforms where the baseline CPU target lacks CRC instructions, we need to supply a `-march` switch to persuade the compiler to compile such instructions. Recent versions of gcc reject the value we were trying, leading to silently falling back to software CRC.

- During configure, if a C23 compiler is detected, try asking for C17 (Thomas Munro)

  PostgreSQL versions before v16 will not compile under C23 rules. If the chosen compiler defaults to C23 or later, try adding a `-std=gnu17` switch to change that. (If this won't work for your compiler, manually specify `CFLAGS` with a suitable switch.)

- Update time zone data files to tzdata release 2025a for DST law changes in Paraguay, plus historical corrections for the Philippines (Tom Lane)

E.10. Release 15.10

Release date: 2024-11-21

This release contains a few fixes from 15.9. For information about new features in major release 15, see Section E.20.

E.10.1. Migration to Version 15.10

A dump/restore is not required for those running 15.X.

However, if you are upgrading from a version earlier than 15.9, see Section E.11.

E.10.2. Changes

- Repair ABI break for extensions that work with struct `ResultRelInfo` (Tom Lane)

Last week's minor releases unintentionally broke binary compatibility with timescaledb and several other extensions. Restore the affected structure to its previous size, so that such extensions need not be rebuilt.

- Restore functionality of `ALTER {ROLE|DATABASE} SET role` (Tom Lane, Noah Misch)

  The fix for CVE-2024-10978 accidentally caused settings for `role` to not be applied if they come from non-interactive sources, including previous `ALTER {ROLE|DATABASE}` commands and the `PGOPTIONS` environment variable.

- Fix cases where a logical replication slot's `restart_lsn` could go backwards (Masahiko Sawada)

  Previously, restarting logical replication could sometimes cause the slot's restart point to be recomputed as an older value than had previously been advertised in `pg_replication_slots`. This is bad, since for example WAL files might have been removed on the basis of the later `restart_lsn` value, in which case replication would fail to restart.

- Avoid deleting still-needed WAL files during pg_rewind (Polina Bungina, Alexander Kukushkin)

  Previously, in unlucky cases, it was possible for pg_rewind to remove important WAL files from the rewound demoted primary. In particular this happens if those files have been marked for archival (i.e., their `.ready` files were created) but not yet archived. Then the newly promoted node no longer has such files because of them having been recycled, but likely they are needed for recovery in the demoted node. If pg_rewind removes them, recovery is not possible anymore.

- Fix race conditions associated with dropping shared statistics entries (Kyotaro Horiguchi, Michael Paquier)

  These bugs could lead to loss of statistics data, assertion failures, or « can only drop stats once » errors.

- Count index scans in `contrib/bloom` indexes in the statistics views, such as the `pg_stat_user_indexes.idx_scan` counter (Masahiro Ikeda)

- Fix crash when checking to see if an index's opclass options have changed (Alexander Korotkov)

  Some forms of `ALTER TABLE` would fail if the table has an index with non-default operator class options.

- Avoid assertion failure caused by disconnected NFA sub-graphs in regular expression parsing (Tom Lane)

  This bug does not appear to have any visible consequences in non-assert builds.

E.11. Release 15.9

Release date: 2024-11-14

This release contains a variety of fixes from 15.8. For information about new features in major release 15, see Section E.20.

E.11.1. Migration to Version 15.9

A dump/restore is not required for those running 15.X.

However, if you have ever detached a partition from a partitioned table that has a foreign-key reference to another partitioned table, and not dropped the former partition, then you may have catalog and/or data corruption to repair, as detailed in the fifth changelog entry below.

Also, if you are upgrading from a version earlier than 15.7, see Section E.13.

E.11.2. Changes

- Ensure cached plans are marked as dependent on the calling role when RLS applies to a non-top-level table reference (Nathan Bossart)

  If a CTE, subquery, sublink, security invoker view, or coercion projection in a query references a table with row-level security policies, we neglected to mark the resulting plan as potentially dependent on which role is executing it. This could lead to later query executions in the same session using the wrong plan, and then returning or hiding rows that should have been hidden or returned instead.

  The PostgreSQL Project thanks Wolfgang Walther for reporting this problem. (CVE-2024-10976)

- Make libpq discard error messages received during SSL or GSS protocol negotiation (Jacob Champion)

  An error message received before encryption negotiation is completed might have been injected by a man-in-the-middle, rather than being real server output. Reporting it opens the door to various security hazards; for example, the message might spoof a query result that a careless user could mistake for correct output. The best answer seems to be to discard such data and rely only on libpq's own report of the connection failure.

  The PostgreSQL Project thanks Jacob Champion for reporting this problem. (CVE-2024-10977)

- Fix unintended interactions between `SET SESSION AUTHORIZATION` and `SET ROLE` (Tom Lane)

  The SQL standard mandates that `SET SESSION AUTHORIZATION` have a side-effect of doing `SET ROLE NONE`. Our implementation of that was flawed, creating more interaction between the two settings than intended. Notably, rolling back a transaction that had done `SET SESSION AUTHORIZATION` would revert `ROLE` to `NONE` even if that had not been the previous state, so that the effective user ID might now be different from what it had been before the transaction. Transiently setting `session_authorization` in a function `SET` clause had a similar effect. A related bug was that if a parallel worker inspected `current_setting('role')`, it saw `none` even when it should see something else.

  The PostgreSQL Project thanks Tom Lane for reporting this problem. (CVE-2024-10978)

- Prevent trusted PL/Perl code from changing environment variables (Andrew Dunstan, Noah Misch)

  The ability to manipulate process environment variables such as `PATH` gives an attacker opportunities to execute arbitrary code. Therefore, « trusted » PLs must not offer the ability to do that. To fix `plperl`, replace `%ENV` with a tied hash that rejects any modification attempt with a warning. Untrusted `plperlu` retains the ability to change the environment.

  The PostgreSQL Project thanks Coby Abrams for reporting this problem. (CVE-2024-10979)

- Fix updates of catalog state for foreign-key constraints when attaching or detaching table partitions (Jehan-Guillaume de Rorthais, Tender Wang, Álvaro Herrera)

  If the referenced table is partitioned, then different catalog entries are needed for a referencing table that is stand-alone versus one that is a partition. `ATTACH/DETACH PARTITION` commands failed to perform this conversion correctly. In particular, after `DETACH` the now stand-alone table would be missing foreign-key enforcement triggers, which could result in the table later containing rows that fail the foreign-key constraint. A subsequent re-`ATTACH` could fail with surprising errors, too.

  The way to fix this is to do `ALTER TABLE DROP CONSTRAINT` on the now stand-alone table for each faulty constraint, and then re-add the constraint. If re-adding the constraint fails, then some erroneous data has crept in. You will need to manually re-establish consistency between the referencing and referenced tables, then re-add the constraint.

This query can be used to identify broken constraints and construct the commands needed to recreate them:

```
SELECT conrelid::pg_catalog.regclass AS "constrained table",
       conname AS constraint,
       confrelid::pg_catalog.regclass AS "references",
       pg_catalog.format('ALTER TABLE %s DROP CONSTRAINT %I;',
                         conrelid::pg_catalog.regclass, conname)
 AS "drop",
       pg_catalog.format('ALTER TABLE %s ADD CONSTRAINT %I %s;',
                         conrelid::pg_catalog.regclass, conname,
                         pg_catalog.pg_get_constraintdef(oid)) AS
 "add"
FROM pg_catalog.pg_constraint c
WHERE contype = 'f' AND conparentid = 0 AND
   (SELECT count(*) FROM pg_catalog.pg_constraint c2
    WHERE c2.conparentid = c.oid) <>
   ((SELECT count(*) FROM pg_catalog.pg_inherits i
    WHERE (i.inhparent = c.conrelid OR i.inhparent = c.confrelid)
 AND
      EXISTS (SELECT 1 FROM pg_catalog.pg_partitioned_table
              WHERE partrelid = i.inhparent)) +
    CASE WHEN pg_catalog.pg_partition_root(conrelid) = confrelid
 THEN
              (SELECT count(*) FROM
 pg_catalog.pg_partition_tree(confrelid)
                WHERE level = 1)
         ELSE 0 END);
```

Since it is possible that one or more of the `ADD CONSTRAINT` steps will fail, you should save the query's output in a file and then attempt to perform each step.

- Avoid possible crashes and « could not open relation » errors in queries on a partitioned table occurring concurrently with a `DETACH CONCURRENTLY` and immediate drop of a partition (Álvaro Herrera, Kuntal Gosh)

- Disallow `ALTER TABLE ATTACH PARTITION` if the table to be attached has a foreign key referencing the partitioned table (Álvaro Herrera)

  This arrangement is not supported, and other ways of creating it already fail.

- Don't use partitionwise joins or grouping if the query's collation for the key column doesn't match the partition key's collation (Jian He, Webbo Han)

  Such plans could produce incorrect results.

- Fix possible « could not find pathkey item to sort » error when the output of a `UNION ALL` member query needs to be sorted, and the sort column is an expression (Andrei Lepikhov, Tom Lane)

- Allow cancellation of the second stage of index build for large hash indexes (Pavel Borisov)

- Fix assertion failure or confusing error message for `COPY (`*`query`*`) TO ...`, when the *`query`* is rewritten by a `DO INSTEAD NOTIFY` rule (Tender Wang, Tom Lane)

- Fix detection of skewed data during parallel hash join (Thomas Munro)

  After repartitioning the inner side of a hash join because one partition has accumulated too many tuples, we check to see if all the partition's tuples went into the same child partition, which suggests

that they all have the same hash value and further repartitioning cannot improve matters. This check malfunctioned in some cases, allowing repeated futile repartitioning which would eventually end in a resource-exhaustion error.

- Fix race condition in committing a serializable transaction (Heikki Linnakangas)

  Mis-processing of a recently committed transaction could lead to an assertion failure or a « could not access status of transaction » error.

- Fix race condition in `COMMIT PREPARED` that resulted in orphaned 2PC files (wuchengwen)

  A concurrent `PREPARE TRANSACTION` could cause `COMMIT PREPARED` to not remove the on-disk two-phase state file for the completed transaction. There was no immediate ill effect, but a subsequent crash-and-recovery could fail with « could not access status of transaction », requiring manual removal of the orphaned file to restore service.

- Avoid invalid memory accesses after skipping an invalid toast index during `VACUUM FULL` (Tender Wang)

  A list tracking yet-to-be-rebuilt indexes was not properly updated in this code path, risking assertion failures or crashes later on.

- Fix ways in which an « in place » catalog update could be lost (Noah Misch)

  Normal row updates write a new version of the row to preserve rollback-ability of the transaction. However, certain system catalog updates are intentionally non-transactional and are done with an in-place update of the row. These patches fix race conditions that could cause the effects of an in-place update to be lost. As an example, it was possible to forget having set `pg_class.relhasindex` to true, preventing updates of the new index and thus causing index corruption.

- Reset catalog caches at end of recovery (Noah Misch)

  This prevents scenarios wherein an in-place catalog update could be lost due to using stale data from a catalog cache.

- Avoid using parallel query while holding off interrupts (Francesco Degrassi, Noah Misch, Tom Lane)

  This situation cannot arise normally, but it can be reached with test scenarios such as using a SQL-language function as B-tree support (which would be far too slow for production usage). If it did occur it would result in an indefinite wait.

- Report the active query ID for statistics purposes at the start of processing of Bind and Execute protocol messages (Sami Imseih)

  This allows more of the work done in extended query protocol to be attributed to the correct query.

- Guard against stack overflow in libxml2 with too-deeply-nested XML input (Tom Lane, with hat tip to Nick Wellnhofer)

  Use `xmlXPathCtxtCompile()` rather than `xmlXPathCompile()`, because the latter fails to protect itself against recursion-to-stack-overflow in libxml2 releases before 2.13.4.

- Do not ignore a concurrent `REINDEX CONCURRENTLY` that is working on an index with predicates or expressions (Michail Nikolaev)

  Normally, `REINDEX CONCURRENTLY` does not need to wait for other `REINDEX CONCURRENTLY` operations on other tables. However, this optimization is not applied if the other `REINDEX CONCURRENTLY` is processing an index with predicates or expressions, on the chance that such expressions contain user-defined code that accesses other tables. Careless coding created a race condition such that that rule was not applied uniformly, possibly allowing inconsistent behavior.

- Fix « failed to find plan for subquery/CTE » errors in `EXPLAIN` (Richard Guo, Tom Lane)

  This case arose while trying to print references to fields of a RECORD-type output of a subquery when the subquery has been optimized out of the plan altogether (which is possible at least in the case that it has a constant-false `WHERE` condition). Nothing remains in the plan to identify the original field names, so fall back to printing `f`*`N`* for the *`N`*'th record column. (That's actually the right thing anyway, if the record output arose from a `ROW()` constructor.)

- Disallow a `USING` clause when altering the type of a generated column (Peter Eisentraut)

  A generated column already has an expression specifying the column contents, so including `USING` doesn't make sense.

- Ignore not-yet-defined Portals in the `pg_cursors` view (Tom Lane)

  It is possible for user-defined code that inspects this view to be called while a new cursor is being set up, and if that happens a null pointer dereference would ensue. Avoid the problem by defining the view to exclude incompletely-set-up cursors.

- Prevent mis-encoding of « trailing junk after numeric literal » error messages (Karina Litskevich)

  We do not allow identifiers to appear immediately following numeric literals (there must be some whitespace between). If a multibyte character immediately followed a numeric literal, the syntax error message about it included only the first byte of that character, causing bad-encoding problems both in the report to the client and in the postmaster log file.

- Avoid « unexpected table_index_fetch_tuple call during logical decoding » error while decoding a transaction involving insertion of a column default value (Takeshi Ideriha, Hou Zhijie)

- Reduce memory consumption of logical decoding (Masahiko Sawada)

  Use a smaller default block size to store tuple data received during logical replication. This reduces memory wastage, which has been reported to be severe while processing long-running transactions, even leading to out-of-memory failures.

- Re-disable sending of stateless (TLSv1.2) session tickets (Daniel Gustafsson)

  A previous change to prevent sending of stateful (TLSv1.3) session tickets accidentally re-enabled sending of stateless ones. Thus, while we intended to prevent clients from thinking that TLS session resumption is supported, some still did.

- Avoid « wrong tuple length » failure when dropping a database with many ACL (permission) entries (Ayush Tiwari)

- Allow adjusting the `session_authorization` and `role` settings in parallel workers (Tom Lane)

  Our code intends to allow modifiable server settings to be set by function `SET` clauses, but not otherwise within a parallel worker. `SET` clauses failed for these two settings, though.

- Fix behavior of stable functions called from a `CALL` statement's argument list, when the `CALL` is within a PL/pgSQL `EXCEPTION` block (Tom Lane)

  As with a similar fix in our previous quarterly releases, this case allowed such functions to be passed the wrong snapshot, causing them to see stale values of rows modified since the start of the outer transaction.

- Fix « cache lookup failed for function » errors in edge cases in PL/pgSQL's `CALL` (Tom Lane)

- Fix thread safety of our fallback (non-OpenSSL) MD5 implementation on big-endian hardware (Heikki Linnakangas)

Thread safety is not currently a concern in the server, but it is for libpq.

- Parse libpq's `keepalives` connection option in the same way as other integer-valued options (Yuto Sasaki)

  The coding used here rejected trailing whitespace in the option value, unlike other cases. This turns out to be problematic in ecpg's usage, for example.

- Avoid use of `pnstrdup()` in ecpglib (Jacob Champion)

  That function will call `exit()` on out-of-memory, which is undesirable in a library. The calling code already handles allocation failures properly.

- In ecpglib, fix out-of-bounds read when parsing incorrect datetime input (Bruce Momjian, Pavel Nekrasov)

  It was possible to try to read the location just before the start of a constant array. Real-world consequences seem minimal, though.

- Fix pg_dump's handling of identity sequences that have persistence different from their owning table's persistence (Tom Lane)

  Since v15, it's been possible to set an identity sequence to be LOGGED when its owning table is UNLOGGED or vice versa. However, pg_dump's method for recreating that situation failed in binary-upgrade mode, causing pg_upgrade to fail when such sequences are present. Fix by introducing a new option for `ADD/ALTER COLUMN GENERATED AS IDENTITY` to allow the sequence's persistence to be set correctly at creation. Note that this means a dump from a database containing such a sequence will only load into a server of this minor version or newer.

- Include the source timeline history in pg_rewind's debug output (Heikki Linnakangas)

  This was the intention to begin with, but a coding error caused the source history to always print as empty.

- Fix misbehavior with junction points on Windows, particularly in pg_rewind (Alexandra Wang)

  This entailed back-patching previous fixes by Thomas Munro, Peter Eisentraut, Alexander Lakhin, and Juan José Santamaría Flecha. Those changes were originally not back-patched out of caution, but they have been in use in later branches for long enough to deem them safe.

- Avoid trying to reindex temporary tables and indexes in vacuumdb and in parallel reindexdb (VaibhaveS, Michael Paquier, Fujii Masao, Nathan Bossart)

  Reindexing other sessions' temporary tables cannot work, but the check to skip them was missing in some code paths, leading to unwanted failures.

- Allow inspection of sequence relations in relevant functions of `contrib/pageinspect` and `contrib/pgstattuple` (Nathan Bossart, Ayush Vatsa)

  This had been allowed in the past, but it got broken during the introduction of non-default access methods for tables.

- Fix incorrect LLVM-generated code on ARM64 platforms (Thomas Munro, Anthonin Bonnefoy)

  When using JIT compilation on ARM platforms, the generated code could not support relocation distances exceeding 32 bits, allowing unlucky placement of generated code to cause server crashes on large-memory systems.

- Fix a few places that assumed that process start time (represented as a `time_t`) will fit into a `long` value (Max Johnson, Nathan Bossart)

On platforms where `long` is 32 bits (notably Windows), this coding would fail after Y2038. Most of the failures appear only cosmetic, but notably `pg_ctl start` would hang.

- Fix building with Strawberry Perl on Windows (Andrew Dunstan)

- Prevent « missing declaration for inet_pton » compiler warning or error when building with MinGW (Thomas Munro, Andrew Dunstan)

- Update time zone data files to tzdata release 2024b (Tom Lane)

  This tzdata release changes the old System-V-compatibility zone names to duplicate the corresponding geographic zones; for example `PST8PDT` is now an alias for `America/Los_Angeles`. The main visible consequence is that for timestamps before the introduction of standardized time zones, the zone is considered to represent local mean solar time for the named location. For example, in `PST8PDT`, `timestamptz` input such as `1801-01-01 00:00` would previously have been rendered as `1801-01-01 00:00:00-08`, but now it is rendered as `1801-01-01 00:00:00-07:52:58`.

  Also, historical corrections for Mexico, Mongolia, and Portugal. Notably, `Asia/Choibalsan` is now an alias for `Asia/Ulaanbaatar` rather than being a separate zone, mainly because the differences between those zones were found to be based on untrustworthy data.

E.12. Release 15.8

Release date: 2024-08-08

This release contains a variety of fixes from 15.7. For information about new features in major release 15, see Section E.20.

E.12.1. Migration to Version 15.8

A dump/restore is not required for those running 15.X.

However, if you are upgrading from a version earlier than 15.7, see Section E.13.

E.12.2. Changes

- Prevent unauthorized code execution during pg_dump (Masahiko Sawada)

  An attacker able to create and drop non-temporary objects could inject SQL code that would be executed by a concurrent pg_dump session with the privileges of the role running pg_dump (which is often a superuser). The attack involves replacing a sequence or similar object with a view or foreign table that will execute malicious code. To prevent this, introduce a new server parameter `restrict_nonsystem_relation_kind` that can disable expansion of non-builtin views as well as access to foreign tables, and teach pg_dump to set it when available. Note that the attack is prevented only if both pg_dump and the server it is dumping from are new enough to have this fix.

  The PostgreSQL Project thanks Noah Misch for reporting this problem. (CVE-2024-7348)

- Prevent infinite loop in `VACUUM` (Melanie Plageman)

  After a disconnected standby server with an old running transaction reconnected to the primary, it was possible for `VACUUM` on the primary to get confused about which tuples are removable, resulting in an infinite loop.

- Fix failure after attaching a table as a partition, if the table had previously had inheritance children (Álvaro Herrera)

- Fix `ALTER TABLE DETACH PARTITION` for cases involving inconsistent index-based constraints (Álvaro Herrera, Tender Wang)

  When a partitioned table has an index that is not associated with a constraint, but a partition has an equivalent index that is, then detaching the partition would misbehave, leaving the ex-partition's constraint with an incorrect `coninhcount` value. This would cause trouble during any further manipulations of that constraint.

- Fix partition pruning setup during `ALTER TABLE DETACH PARTITION CONCURRENTLY` (Álvaro Herrera)

  The executor assumed that no partition could be detached between planning and execution of a query on a partitioned table. This is no longer true since the introduction of `DETACH PARTITION`'s `CONCURRENTLY` option, making it possible for query execution to fail transiently when that is used.

- Correctly update a partitioned table's `pg_class.reltuples` field to zero after its last child partition is dropped (Noah Misch)

  The first `ANALYZE` on such a partitioned table must update `relhassubclass` as well, and that caused the `reltuples` update to be lost.

- Fix handling of polymorphic output arguments for procedures (Tom Lane)

  The SQL `CALL` statement did not resolve the correct data types for such arguments, leading to errors such as « cannot display a value of type anyelement », or even outright crashes. (But `CALL` in PL/pgSQL worked correctly.)

- Fix behavior of stable functions called from a `CALL` statement's argument list (Tom Lane)

  If the `CALL` is within an atomic context (e.g. there's an outer transaction block), such functions were passed the wrong snapshot, causing them to see stale values of rows modified since the start of the outer transaction.

- Detect integer overflow in `money` calculations (Joseph Koshakow)

  None of the arithmetic functions for the `money` type checked for overflow before, so they would silently give wrong answers for overflowing cases.

- Fix over-aggressive clamping of the scale argument in `round(numeric)` and `trunc(numeric)` (Dean Rasheed)

  These functions clamped their scale argument to +/-2000, but there are valid use-cases for it to be larger; the functions returned incorrect results in such cases. Instead clamp to the actual allowed range of type `numeric`.

- Fix result for `pg_size_pretty()` when applied to the smallest possible `bigint` value (Joseph Koshakow)

- Prevent `pg_sequence_last_value()` from failing on unlogged sequences on standby servers and on temporary sequences of other sessions (Nathan Bossart)

  Make it return NULL in these cases instead of throwing an error.

- Fix parsing of ignored operators in `websearch_to_tsquery()` (Tom Lane)

  Per the manual, punctuation in the input of `websearch_to_tsquery()` is ignored except for the special cases of dashes and quotes. However, parentheses and a few other characters appearing immediately before an `or` could cause `or` to be treated as a data word, rather than as an `OR` operator as expected.

- Detect another integer overflow case while computing new array dimensions (Joseph Koshakow)

Reject applying array dimensions `[-2147483648:2147483647]` to an empty array. This is closely related to CVE-2023-5869, but appears harmless since the array still ends up empty.

- Detect another case of a new catalog cache entry becoming stale while detoasting its fields (Noah Misch)

  An in-place update occurring while we expand out-of-line fields in a catalog tuple could be missed, leading to a catalog cache entry that lacks the in-place change but is not known to be stale. This is only possible in the `pg_database` catalog, so the effects are narrow, but misbehavior is possible.

- Correctly check updatability of view columns targeted by `INSERT ... DEFAULT` (Tom Lane)

  If such a column is non-updatable, we should give an error reporting that. But the check was missed and then later code would report an unhelpful error such as « attribute number *N* not found in view targetlist ».

- Avoid reporting an unhelpful internal error for incorrect recursive queries (Tom Lane)

  Rearrange the order of error checks so that we throw an on-point error when a `WITH RECURSIVE` query does not have a self-reference within the second arm of the `UNION`, but does have one self-reference in some other place such as `ORDER BY`.

- Lock owned sequences during `ALTER TABLE SET LOGGED|UNLOGGED` (Noah Misch)

  These commands change the persistence of a table's owned sequences along with the table, but they failed to acquire lock on the sequences while doing so. This could result in losing the effects of concurrent `nextval()` calls.

- Don't throw an error if a queued `AFTER` trigger no longer exists (Tom Lane)

  It's possible for a transaction to execute an operation that queues a deferred `AFTER` trigger for later execution, and then to drop the trigger before that happens. Formerly this led to weird errors such as « could not find trigger *NNNN* ». It seems better to silently do nothing if the trigger no longer exists at the time when it would have been executed.

- Fix failure to remove `pg_init_privs` entries for column-level privileges when their table is dropped (Tom Lane)

  If an extension grants some column-level privileges on a table it creates, relevant catalog entries would remain behind after the extension is dropped. This was harmless until/unless the table's OID was re-used for another relation, when it could interfere with what pg_dump dumps for that relation.

- Fix selection of an arbiter index for `ON CONFLICT` when the desired index has expressions or predicates (Tom Lane)

  If a query using `ON CONFLICT` accesses the target table through an updatable view, it could fail with « there is no unique or exclusion constraint matching the ON CONFLICT specification », even though a matching index does exist.

- Refuse to modify a temporary table of another session with `ALTER TABLE` (Tom Lane)

  Permissions checks normally would prevent this case from arising, but it is possible to reach it by altering a parent table whose child is another session's temporary table. Throw an error if we discover that such a child table belongs to another session.

- Fix handling of extended statistics on expressions in `CREATE TABLE LIKE STATISTICS` (Tom Lane)

  The `CREATE` command failed to adjust column references in statistics expressions to the possibly-different column numbering of the new table. This resulted in invalid statistics objects that would

cause problems later. A typical scenario where renumbering columns is needed is when the source table contains some dropped columns.

- Fix failure to recalculate sub-queries generated from `MIN()` or `MAX()` aggregates (Tom Lane)

  In some cases the aggregate result computed at one row of the outer query could be re-used for later rows when it should not be. This has only been seen to happen when the outer query uses `DISTINCT` that is implemented with hash aggregation, but other cases may exist.

- Avoid crashing when a JIT-inlined backend function throws an error (Tom Lane)

  The error state can include pointers into the dynamically loaded module holding the JIT-compiled code (for error location strings). In some code paths the module could get unloaded before the error report is processed, leading to SIGSEGV when the location strings are accessed.

- Cope with behavioral changes in libxml2 version 2.13.x (Erik Wienhold, Tom Lane)

  Notably, we now suppress « chunk is not well balanced » errors from libxml2, unless that is the only reported error. This is to make error reports consistent between 2.13.x and earlier libxml2 versions. In earlier versions, that message was almost always redundant or outright incorrect, so 2.13.x substantially reduced the number of cases in which it's reported.

- Fix handling of subtransactions of prepared transactions when starting a hot standby server (Heikki Linnakangas)

  When starting a standby's replay at a shutdown checkpoint WAL record, transactions that had been prepared but not yet committed on the primary are correctly understood as being still in progress. But subtransactions of a prepared transaction (created by savepoints or PL/pgSQL exception blocks) were not accounted for and would be treated as aborted. That led to inconsistency if the prepared transaction was later committed.

- Prevent incorrect initialization of logical replication slots (Masahiko Sawada)

  In some cases a replication slot's start point within the WAL stream could be set to a point within a transaction, leading to assertion failures or incorrect decoding results.

- Avoid « can only drop stats once » error during replication slot creation and drop (Floris Van Nee)

- Fix resource leakage in logical replication WAL sender (Hou Zhijie)

  The walsender process leaked memory when publishing changes to a partitioned table whose partitions have row types physically different from the partitioned table's.

- Avoid memory leakage after servicing a notify or sinval interrupt (Tom Lane)

  The processing functions for these events could switch the current memory context to TopMemoryContext, resulting in session-lifespan leakage of any data allocated before the incorrect setting gets replaced. There were observable leaks associated with (at least) encoding conversion of incoming queries and parameters attached to Bind messages.

- Prevent leakage of reference counts for the shared memory block used for statistics (Anthonin Bonnefoy)

  A new backend process attaching to the statistics shared memory incremented its reference count, but failed to decrement the count when exiting. After 2^{32} sessions had been created, the reference count would overflow to zero, causing failures in all subsequent backend process starts.

- Prevent deadlocks and assertion failures during truncation of the multixact SLRU log (Heikki Linnakangas)

  A process trying to delete SLRU segments could deadlock with the checkpointer process.

- Avoid possibly missing end-of-input events on Windows sockets (Thomas Munro)

  Windows reports an FD_CLOSE event only once after the remote end of the connection disconnects. With unlucky timing, we could miss that report and wait indefinitely, or at least until a timeout elapsed, expecting more input.

- Fix buffer overread in JSON parse error reports for incomplete byte sequences (Jacob Champion)

  It was possible to walk off the end of the input buffer by a few bytes when the last bytes comprise an incomplete multi-byte character. While usually harmless, in principle this could cause a crash.

- Disable creation of stateful TLS session tickets by OpenSSL (Daniel Gustafsson)

  This avoids possible failures with clients that think receipt of a session ticket means that TLS session resumption is supported.

- When replanning a PL/pgSQL « simple expression », check it's still simple (Tom Lane)

  Certain fairly-artificial cases, such as dropping a referenced function and recreating it as an aggregate, could lead to surprising failures such as « unexpected plan node type ».

- Fix incompatibility between PL/Perl and Perl 5.40 (Andrew Dunstan)

- Fix recursive `RECORD`-returning PL/Python functions (Tom Lane)

  If we recurse to a new call of the same function that passes a different column definition list (`AS` clause), it would fail because the inner call would overwrite the outer call's idea of what rowtype to return.

- Don't corrupt PL/Python's `TD` dictionary during a recursive trigger call (Tom Lane)

  If a PL/Python-language trigger caused another one to be invoked, the `TD` dictionary created for the inner one would overwrite the outer one's `TD` dictionary.

- Fix PL/Tcl's reporting of invalid list syntax in the result of a function returning tuple (Erik Wienhold, Tom Lane)

  Such a case could result in a crash, or in emission of misleading context information that actually refers to the previous Tcl error.

- Avoid non-thread-safe usage of `strerror()` in libpq (Peter Eisentraut)

  Certain error messages returned by OpenSSL could become garbled in multi-threaded applications.

- Avoid memory leak within pg_dump during a binary upgrade (Daniel Gustafsson)

- Ensure that `pg_restore -l` reports dependent TOC entries correctly (Tom Lane)

  If `-l` was specified together with selective-restore options such as `-n` or `-N`, dependent TOC entries such as comments would be omitted from the listing, even when an actual restore would have selected them.

- Avoid « cursor can only scan forward » error in `contrib/postgres_fdw` (Etsuro Fujita)

  This error could occur if the remote server is v15 or later and a foreign table is mapped to a non-trivial remote view.

- In `contrib/postgres_fdw`, do not send `FETCH FIRST WITH TIES` clauses to the remote server (Japin Li)

  The remote server might not implement this clause, or might interpret it differently than we would locally, so don't risk attempting remote execution.

- Avoid clashing with system-provided `<regex.h>` headers (Thomas Munro)

  This fixes a compilation failure on macOS version 15 and up.

- Fix otherwise-harmless assertion failure in Memoize cost estimation (David Rowley)

- Fix otherwise-harmless assertion failures in `REINDEX CONCURRENTLY` applied to an SP-GiST index (Tom Lane)

E.13. Release 15.7

Release date: 2024-05-09

This release contains a variety of fixes from 15.6. For information about new features in major release 15, see Section E.20.

E.13.1. Migration to Version 15.7

A dump/restore is not required for those running 15.X.

However, a security vulnerability was found in the system views `pg_stats_ext` and `pg_stats_ext_exprs`, potentially allowing authenticated database users to see data they shouldn't. If this is of concern in your installation, follow the steps in the first changelog entry below to rectify it.

Also, if you are upgrading from a version earlier than 15.6, see Section E.14.

E.13.2. Changes

- Restrict visibility of `pg_stats_ext` and `pg_stats_ext_exprs` entries to the table owner (Nathan Bossart)

  These views failed to hide statistics for expressions that involve columns the accessing user does not have permission to read. View columns such as `most_common_vals` might expose security-relevant data. The potential interactions here are not fully clear, so in the interest of erring on the side of safety, make rows in these views visible only to the owner of the associated table.

  The PostgreSQL Project thanks Lukas Fittl for reporting this problem. (CVE-2024-4317)

  By itself, this fix will only fix the behavior in newly initdb'd database clusters. If you wish to apply this change in an existing cluster, you will need to do the following:

  1. Find the SQL script `fix-CVE-2024-4317.sql` in the *`share`* directory of the PostgreSQL installation (typically located someplace like `/usr/share/postgresql/`). Be sure to use the script appropriate to your PostgreSQL major version. If you do not see this file, either your version is not vulnerable (only v14–v16 are affected) or your minor version is too old to have the fix.

  2. In *each* database of the cluster, run the `fix-CVE-2024-4317.sql` script as superuser. In psql this would look like

     ```
     \i /usr/share/postgresql/fix-CVE-2024-4317.sql
     ```

     (adjust the file path as appropriate). Any error probably indicates that you've used the wrong script version. It will not hurt to run the script more than once.

  3. Do not forget to include the `template0` and `template1` databases, or the vulnerability will still exist in databases you create later. To fix `template0`, you'll need to temporarily make it accept connections. Do that with

```
ALTER DATABASE template0 WITH ALLOW_CONNECTIONS true;
```

and then after fixing `template0`, undo it with

```
ALTER DATABASE template0 WITH ALLOW_CONNECTIONS false;
```

- Fix `INSERT` from multiple `VALUES` rows into a target column that is a domain over an array or composite type (Tom Lane)

  Such cases would either fail with surprising complaints about mismatched datatypes, or insert unexpected coercions that could lead to odd results.

- Require `SELECT` privilege on the target table for `MERGE` with a `DO NOTHING` clause (Álvaro Herrera)

  `SELECT` privilege would be required in all practical cases anyway, but require it even if the query reads no columns of the target table. This avoids an edge case in which `MERGE` would require no privileges whatever, which seems undesirable even when it's a do-nothing command.

- Fix handling of self-modified tuples in `MERGE` (Dean Rasheed)

  Throw an error if a target row joins to more than one source row, as required by the SQL standard. (The previous coding could silently ignore this condition if a concurrent update was involved.) Also, throw a non-misleading error if a target row is already updated by a later command in the current transaction, thanks to a `BEFORE` trigger or a volatile function used in the query.

- Fix incorrect pruning of NULL partition when a table is partitioned on a boolean column and the query has a boolean `IS NOT` clause (David Rowley)

  A NULL value satisfies a clause such as *`boolcol`* `IS NOT FALSE`, so pruning away a partition containing NULLs yielded incorrect answers.

- Make `ALTER FOREIGN TABLE SET SCHEMA` move any owned sequences into the new schema (Tom Lane)

  Moving a regular table to a new schema causes any sequences owned by the table to be moved to that schema too (along with indexes and constraints). This was overlooked for foreign tables, however.

- Make `ALTER TABLE ... ADD COLUMN` create identity/serial sequences with the same persistence as their owning tables (Peter Eisentraut)

  `CREATE UNLOGGED TABLE` will make any owned sequences be unlogged too. `ALTER TABLE` missed that consideration, so that an added identity column would have a logged sequence, which seems pointless.

- Improve `ALTER TABLE ... ALTER COLUMN TYPE`'s error message when there is a dependent function or publication (Tom Lane)

- In `CREATE DATABASE`, recognize strategy keywords case-insensitively for consistency with other options (Tomas Vondra)

- Fix `EXPLAIN`'s counting of heap pages accessed by a bitmap heap scan (Melanie Plageman)

  Previously, heap pages that contain no visible tuples were not counted; but it seems more consistent to count all pages returned by the bitmap index scan.

- Fix `EXPLAIN`'s output for subplans in `MERGE` (Dean Rasheed)

`EXPLAIN` would sometimes fail to properly display subplan Params referencing variables in other parts of the plan tree.

- Avoid deadlock during removal of orphaned temporary tables (Mikhail Zhilin)

  If the session that creates a temporary table crashes without removing the table, autovacuum will eventually try to remove the orphaned table. However, an incoming session that's been assigned the same temporary namespace will do that too. If a temporary table has a dependency (such as an owned sequence) then a deadlock could result between these two cleanup attempts.

- Avoid race condition while examining per-relation frozen-XID values (Noah Misch)

  `VACUUM`'s computation of per-database frozen-XID values from per-relation values could get confused by a concurrent update of those values by another `VACUUM`.

- Fix buffer usage reporting for parallel vacuuming (Anthonin Bonnefoy)

  Buffer accesses performed by parallel workers were not getting counted in the statistics reported in `VERBOSE` mode.

- Disallow converting a table to a view within an outer SQL command that is using that table (Tom Lane)

  This avoids possible crashes.

- Ensure that join conditions generated from equivalence classes are applied at the correct plan level (Tom Lane)

  In versions before PostgreSQL 16, it was possible for generated conditions to be evaluated below outer joins when they should be evaluated above (after) the outer join, leading to incorrect query results. All versions have a similar hazard when considering joins to `UNION ALL` trees that have constant outputs for the join column in some `SELECT` arms.

- Prevent potentially-incorrect optimization of some window functions (David Rowley)

  Disable « run condition » optimization of `ntile()` and `count()` with non-constant arguments. This avoids possible misbehavior with sub-selects, typically leading to errors like « WindowFunc not found in subplan target lists ».

- Avoid unnecessary use of moving-aggregate mode with a non-moving window frame (Vallimaharajan G)

  When a plain aggregate is used as a window function, and the window frame start is specified as `UNBOUNDED PRECEDING`, the frame's head cannot move so we do not need to use the special (and more expensive) moving-aggregate mode. This optimization was intended all along, but due to a coding error it never triggered.

- Avoid use of already-freed data while planning partition-wise joins under GEQO (Tom Lane)

  This would typically end in a crash or unexpected error message.

- Avoid freeing still-in-use data in Memoize (Tender Wang, Andrei Lepikhov)

  In production builds this error frequently didn't cause any problems, as the freed data would most likely not get overwritten before it was used.

- Fix incorrectly-reported statistics kind codes in « requested statistics kind *X* is not yet built » error messages (David Rowley)

- Be more careful with `RECORD`-returning functions in `FROM` (Tom Lane)

The output columns of such a function call must be defined by an AS clause that specifies the column names and data types. If the actual function output value doesn't match that, an error is supposed to be thrown at runtime. However, some code paths would examine the actual value prematurely, and potentially issue strange errors or suffer assertion failures if it doesn't match expectations.

- Fix confusion about the return rowtype of SQL-language procedures (Tom Lane)

  A procedure implemented in SQL language that returns a single composite-type column would cause an assertion failure or core dump.

- Add protective stack depth checks to some recursive functions (Egor Chindyaskin)

- Fix mis-rounding and overflow hazards in `date_bin()` (Moaaz Assali)

  In the case where the source timestamp is before the origin timestamp and their difference is already an exact multiple of the stride, the code incorrectly subtracted the stride anyway. Also, detect some integer-overflow cases that would have produced incorrect results.

- Detect integer overflow when adding or subtracting an `interval` to/from a `timestamp` (Joseph Koshakow)

  Some cases that should cause an out-of-range error produced an incorrect result instead.

- Avoid race condition in `pg_get_expr()` (Tom Lane)

  If the relation referenced by the argument is dropped concurrently, the function's intention is to return NULL, but sometimes it failed instead.

- Fix detection of old transaction IDs in XID status functions (Karina Litskevich)

  Transaction IDs more than 2^{31} transactions in the past could be misidentified as recent, leading to misbehavior of `pg_xact_status()` or `txid_status()`.

- Ensure that a table's freespace map won't return a page that's past the end of the table (Ronan Dunklau)

  Because the freespace map isn't WAL-logged, this was possible in edge cases involving an OS crash, a replica promote, or a PITR restore. The result would be a « could not read block » error.

- Fix file descriptor leakage when an error is thrown while waiting in `WaitEventSetWait` (Etsuro Fujita)

- Avoid corrupting exception stack if an FDW implements async append but doesn't configure any wait conditions for the Append plan node to wait for (Alexander Pyhalov)

- Throw an error if an index is accessed while it is being reindexed (Tom Lane)

  Previously this was just an assertion check, but promote it into a regular runtime error. This will provide a more on-point error message when reindexing a user-defined index expression that attempts to access its own table.

- Ensure that index-only scans on `name` columns return a fully-padded value (David Rowley)

  The value physically stored in the index is truncated, and previously a pointer to that value was returned to callers. This provoked complaints when testing under valgrind. In theory it could result in crashes, though none have been reported.

- Fix race condition in deciding whether a table sync operation is needed in logical replication (Vignesh C)

An invalidation event arriving while a subscriber identifies which tables need to be synced would be forgotten about, so that any tables newly in need of syncing might not get processed in a timely fashion.

- Fix crash with DSM allocations larger than 4GB (Heikki Linnakangas)

- Disconnect if a new server session's client socket cannot be put into non-blocking mode (Heikki Linnakangas)

  It was once theoretically possible for us to operate with a socket that's in blocking mode; but that hasn't worked fully in a long time, so fail at connection start rather than misbehave later.

- Fix inadequate error reporting with OpenSSL 3.0.0 and later (Heikki Linnakangas, Tom Lane)

  System-reported errors passed through by OpenSSL were reported with a numeric error code rather than anything readable.

- Avoid concurrent calls to `bindtextdomain()` in libpq and ecpglib (Tom Lane)

  Although GNU gettext's implementation seems to be fine with concurrent calls, the version available on Windows is not.

- Fix crash in ecpg's preprocessor if the program tries to redefine a macro that was defined on the preprocessor command line (Tom Lane)

- In ecpg, avoid issuing false « unsupported feature will be passed to server » warnings (Tom Lane)

- Ensure that the string result of ecpg's `intoasc()` function is correctly zero-terminated (Oleg Tselebrovskiy)

- In psql, avoid leaking a query result after the query is cancelled (Tom Lane)

  This happened only when cancelling a non-last query in a query string made with `\;` separators.

- Fix pg_dumpall so that role comments, if present, will be dumped regardless of the setting of `--no-role-passwords` (Daniel Gustafsson, Álvaro Herrera)

- Skip files named `.DS_Store` in pg_basebackup, pg_checksums, and pg_rewind (Daniel Gustafsson)

  This avoids problems on macOS, where the Finder may create such files.

- Fix PL/pgSQL's parsing of single-line comments (`--`-style comments) following expressions (Erik Wienhold, Tom Lane)

  This mistake caused parse errors if such a comment followed a `WHEN` expression in a PL/pgSQL `CASE` statement.

- In `contrib/amcheck`, don't report false match failures due to short- versus long-header values (Andrey Borodin, Michael Zhilin)

  A variable-length datum in a heap tuple or index tuple could have either a short or a long header, depending on compression parameters that applied when it was made. Treat these cases as equivalent rather than complaining if there's a difference.

- Fix bugs in BRIN output functions (Tomas Vondra)

  These output functions are only used for displaying index entries in `contrib/pageinspect`, so the errors are of limited practical concern.

- In `contrib/postgres_fdw`, avoid emitting requests to sort by a constant (David Rowley)

  This could occur in cases involving `UNION ALL` with constant-emitting subqueries. Sorting by a constant is useless of course, but it also risks being misinterpreted by the remote server, leading to « ORDER BY position *N* is not in select list » errors.

- Make `contrib/postgres_fdw` set the remote session's time zone to `GMT` not `UTC` (Tom Lane)

  This should have the same results for practical purposes. However, `GMT` is recognized by hard-wired code in the server, while `UTC` is looked up in the timezone database. So the old code could fail in the unlikely event that the remote server's timezone database is missing entries.

- In `contrib/xml2`, avoid use of library functions that have been deprecated in recent versions of libxml2 (Dmitry Koval)

- Fix incompatibility with LLVM 18 (Thomas Munro, Dmitry Dolgov)

- Allow `make check` to work with the musl C library (Thomas Munro, Bruce Momjian, Tom Lane)

E.14. Release 15.6

Release date: 2024-02-08

This release contains a variety of fixes from 15.5. For information about new features in major release 15, see Section E.20.

E.14.1. Migration to Version 15.6

A dump/restore is not required for those running 15.X.

However, one bug was fixed that could have resulted in corruption of GIN indexes during concurrent updates. If you suspect such corruption, reindex affected indexes after installing this update.

Also, if you are upgrading from a version earlier than 15.5, see Section E.15.

E.14.2. Changes

- Tighten security restrictions within `REFRESH MATERIALIZED VIEW CONCURRENTLY` (Heikki Linnakangas)

  One step of a concurrent refresh command was run under weak security restrictions. If a materialized view's owner could persuade a superuser or other high-privileged user to perform a concurrent refresh on that view, the view's owner could control code executed with the privileges of the user running `REFRESH`. Fix things so that all user-determined code is run as the view's owner, as expected.

  The only known exploit for this error does not work in PostgreSQL 16.0 and later, so it may be that v16 is not vulnerable in practice.

  The PostgreSQL Project thanks Pedro Gallegos for reporting this problem. (CVE-2024-0985)

- Fix memory leak when performing JIT inlining (Andres Freund, Daniel Gustafsson)

  There have been multiple reports of backend processes suffering out-of-memory conditions after sufficiently many JIT compilations. This fix should resolve that.

- When dequeueing from an LWLock, avoid needing to search the list of waiting processes (Andres Freund)

This fixes O(N^2) behavior when the list of waiters is long. In some use-cases this results in substantial throughput improvements.

- Avoid generating incorrect partitioned-join plans (Richard Guo)

  Some uncommon situations involving lateral references could create incorrect plans. Affected queries could produce wrong answers, or odd failures such as « variable not found in subplan target list », or executor crashes.

- Fix incorrect wrapping of subquery output expressions in PlaceHolderVars (Tom Lane)

  This fixes incorrect results when a subquery is underneath an outer join and has an output column that laterally references something outside the outer join's scope. The output column might not appear as NULL when it should do so due to the action of the outer join.

- Fix misprocessing of window function run conditions (Richard Guo)

  This oversight could lead to « WindowFunc not found in subplan target lists » errors.

- Skip inappropriate actions when `MERGE` causes a cross-partition update (Dean Rasheed)

  When executing a `MERGE UPDATE` action on a partitioned table, if the `UPDATE` is turned into a `DELETE` and `INSERT` due to changing a partition key column, skip firing `AFTER UPDATE ROW` triggers, as well as other post-update actions such as RLS checks. These actions would typically fail, which is why a regular `UPDATE` doesn't do them in such cases; `MERGE` shouldn't either.

- Cope with `BEFORE ROW DELETE` triggers in cross-partition `MERGE` updates (Dean Rasheed)

  If such a trigger attempted to prevent the update by returning NULL, `MERGE` would suffer an error or assertion failure.

- Prevent access to a no-longer-pinned buffer in `BEFORE ROW UPDATE` triggers (Alexander Lakhin, Tom Lane)

  If the tuple being updated had just been updated and moved to another page by another session, there was a narrow window where we would attempt to fetch data from the new tuple version without any pin on its buffer. In principle this could result in garbage data appearing in non-updated columns of the proposed new tuple. The odds of problems in practice seem rather low, however.

- Avoid requesting an oversize shared-memory area in parallel hash join (Thomas Munro, Andrei Lepikhov, Alexander Korotkov)

  The limiting value was too large, allowing « invalid DSA memory alloc request size » errors to occur with sufficiently large expected hash table sizes.

- Avoid assertion failures in `heap_update()` and `heap_delete()` when a tuple to be updated by a foreign-key enforcement trigger fails the extra visibility crosscheck (Alexander Lakhin)

  This error had no impact in non-assert builds.

- Fix overly tight assertion about `false_positive_rate` parameter of BRIN bloom operator classes (Alexander Lakhin)

  This error had no impact in non-assert builds, either.

- Fix possible failure during `ALTER TABLE ADD COLUMN` on a complex inheritance tree (Tender Wang)

  If a grandchild table would inherit the new column via multiple intermediate parents, the command failed with « tuple already updated by self ».

- Fix problems with duplicate token names in `ALTER TEXT SEARCH CONFIGURATION ... MAPPING` commands (Tender Wang, Michael Paquier)

- Properly lock the associated table during `DROP STATISTICS` (Tomas Vondra)

  Failure to acquire the lock could result in « tuple concurrently deleted » errors if the `DROP` executes concurrently with `ANALYZE`.

- Fix function volatility checking for `GENERATED` and `DEFAULT` expressions (Tom Lane)

  These places could fail to detect insertion of a volatile function default-argument expression, or decide that a polymorphic function is volatile although it is actually immutable on the datatype of interest. This could lead to improperly rejecting or accepting a `GENERATED` clause, or to mistakenly applying the constant-default-value optimization in `ALTER TABLE ADD COLUMN`.

- Detect that a new catalog cache entry became stale while detoasting its fields (Tom Lane)

  We expand any out-of-line fields in a catalog tuple before inserting it into the catalog caches. That involves database access which might cause invalidation of catalog cache entries -- but the new entry isn't in the cache yet, so we would miss noticing that it should get invalidated. The result is a race condition in which an already-stale cache entry could get made, and then persist indefinitely. This would lead to hard-to-predict misbehavior. Fix by rechecking the tuple's visibility after detoasting.

- Fix edge-case integer overflow detection bug on some platforms (Dean Rasheed)

  Computing `0 - INT64_MIN` should result in an overflow error, and did on most platforms. However, platforms with neither integer overflow builtins nor 128-bit integers would fail to spot the overflow, instead returning `INT64_MIN`.

- Detect Julian-date overflow when adding or subtracting an `interval` to/from a `timestamp` (Tom Lane)

  Some cases that should cause an out-of-range error produced an incorrect result instead.

- Add more checks for overflow in `interval_mul()` and `interval_div()` (Dean Rasheed)

  Some cases that should cause an out-of-range error produced an incorrect result instead.

- Ensure cached statistics are discarded after a change to `stats_fetch_consistency` (Shinya Kato)

  In some code paths, it was possible for stale statistics to be returned.

- Make the `pg_file_settings` view check validity of unapplied values for settings with `backend` or `superuser-backend` context (Tom Lane)

  Invalid values were not noted in the view as intended. This escaped detection because there are very few settings in these groups.

- Match collation too when matching an existing index to a new partitioned index (Peter Eisentraut)

  Previously we could accept an index that has a different collation from the corresponding element of the partition key, possibly leading to misbehavior.

- Avoid failure if a child index is dropped concurrently with `REINDEX INDEX` on a partitioned index (Fei Changhong)

- Fix insufficient locking when cleaning up an incomplete split of a GIN index's internal page (Fei Changhong, Heikki Linnakangas)

  The code tried to do this with shared rather than exclusive lock on the buffer. This could lead to index corruption if two processes attempted the cleanup concurrently.

- Avoid premature release of buffer pin in GIN index insertion (Tom Lane)

  If an index root page split occurs concurrently with our own insertion, the code could fail with « buffer NNNN is not owned by resource owner ».

- Avoid failure with partitioned SP-GiST indexes (Tom Lane)

  Trying to use an index of this kind could lead to « No such file or directory » errors.

- Fix ownership change reporting for large objects (Tom Lane)

  A no-op `ALTER LARGE OBJECT OWNER` command (that is, one selecting the existing owner) passed the wrong class ID to the `PostAlterHook`, probably confusing any extension using that hook.

- Fix reporting of I/O timing data in `EXPLAIN (BUFFERS)` (Michael Paquier)

  The numbers labeled as « shared/local » actually refer only to shared buffers, so change that label to « shared ».

- Ensure durability of `CREATE DATABASE` (Noah Misch)

  If an operating system crash occurred during or shortly after `CREATE DATABASE`, recovery could fail, or subsequent connections to the new database could fail. If a base backup was taken in that window, similar problems could be observed when trying to use the backup. The symptom would be that the database directory, `PG_VERSION` file, or `pg_filenode.map` file was missing or empty.

- Add more `LOG` messages when starting and ending recovery from a backup (Andres Freund)

  This change provides additional information in the postmaster log that may be useful for diagnosing recovery problems.

- Prevent standby servers from incorrectly processing dead index tuples during subtransactions (Fei Changhong)

  The `startedInRecovery` flag was not correctly set for a subtransaction. This affects only processing of dead index tuples. It could allow a query in a subtransaction to ignore index entries that it should return (if they are already dead on the primary server, but not dead to the standby transaction), or to prematurely mark index entries as dead that are not yet dead on the primary. It is not clear that the latter case has any serious consequences, but it's not the intended behavior.

- Fix integer overflow hazard in checking whether a record will fit into the WAL decoding buffer (Thomas Munro)

  This bug appears to be only latent except when running a 32-bit PostgreSQL build on a 64-bit platform.

- Fix deadlock between a logical replication apply worker, its tablesync worker, and a session process trying to alter the subscription (Shlok Kyal)

  One edge of the deadlock loop did not involve a lock wait, so the deadlock went undetected and would persist until manual intervention.

- Ensure that column default values are correctly transmitted by the pgoutput logical replication plugin (Nikhil Benesch)

  `ALTER TABLE ADD COLUMN` with a constant default value for the new column avoids rewriting existing tuples, instead expecting that reading code will insert the correct default into a tuple that lacks that column. If replication was subsequently initiated on the table, pgoutput would transmit NULL instead of the correct default for such a column, causing incorrect replication on the subscriber.

- Fix failure of logical replication's initial sync for a table with no columns (Vignesh C)

  This case generated an improperly-formatted `COPY` command.

- Prevent examining system catalogs with the wrong snapshot during logical decoding (Fei Changhong)

  If decoding begins partway through a transaction that modifies system catalogs, the decoder may not recognize that, causing it to fail to treat that transaction as in-progress for catalog lookups. This fix deals with the case that a top-level transaction is already marked as containing catalog changes, but its subtransaction(s) are not.

- Return the correct status code when a new client disconnects without responding to the server's password challenge (Liu Lang, Tom Lane)

  In some cases we'd treat this as a loggable error, which was not the intention and tends to create log spam, since common clients like psql frequently do this. It may also confuse extensions that use `ClientAuthentication_hook`.

- Fix incompatibility with OpenSSL 3.2 (Tristan Partin, Bo Andreson)

  Use the BIO « app_data » field for our private storage, instead of assuming it's okay to use the « data » field. This mistake didn't cause problems before, but with 3.2 it leads to crashes and complaints about double frees.

- Be more wary about OpenSSL not setting `errno` on error (Tom Lane)

  If `errno` isn't set, assume the cause of the reported failure is read EOF. This fixes rare cases of strange error reports like « could not accept SSL connection: Success ».

- Fix file descriptor leakage when a foreign data wrapper's `ForeignAsyncRequest` function fails (Heikki Linnakangas)

- Report ENOMEM errors from file-related system calls as `ERRCODE_OUT_OF_MEMORY`, not `ERRCODE_INTERNAL_ERROR` (Alexander Kuzmenkov)

- In PL/pgSQL, support SQL commands that are `CREATE FUNCTION`/`CREATE PROCEDURE` with SQL-standard bodies (Tom Lane)

  Previously, such cases failed with parsing errors due to the semicolon(s) appearing in the function body.

- Fix libpq's handling of errors in pipelines (Álvaro Herrera)

  The pipeline state could get out of sync if an error is returned for reasons other than a query problem (for example, if the connection is lost). Potentially this would lead to a busy-loop in the calling application.

- Make libpq's `PQsendFlushRequest()` function flush the client output buffer under the same rules as other `PQsend` functions (Jelte Fennema-Nio)

  In pipeline mode, it may still be necessary to call `PQflush()` as well; but this change removes some inconsistency.

- Avoid race condition when libpq initializes OpenSSL support concurrently in two different threads (Willi Mann, Michael Paquier)

- Fix timing-dependent failure in GSSAPI data transmission (Tom Lane)

  When using GSSAPI encryption in non-blocking mode, libpq sometimes failed with « GSSAPI caller failed to retransmit all data needing to be retried ».

- In pg_dump, don't dump RLS policies or security labels for extension member objects (Tom Lane, Jacob Champion)

  Previously, commands would be included in the dump to set these properties, which is really incorrect since they should be considered as internal affairs of the extension. Moreover, the restoring user might not have adequate privilege to set them, and indeed the dumping user might not have enough privilege to dump them (since dumping RLS policies requires acquiring lock on their table).

- In pg_dump, don't dump an extended statistics object if its underlying table isn't being dumped (Rian McGuire, Tom Lane)

  This conforms to the behavior for other dependent objects such as indexes.

- Make it an error for a pgbench script to end with an open pipeline (Anthonin Bonnefoy)

  Previously, pgbench would behave oddly if a `\startpipeline` command lacked a matching `\endpipeline`. This seems like a scripting mistake rather than a case that pgbench needs to handle nicely, so throw an error.

- Fix crash in `contrib/intarray` if an array with an element equal to `INT_MAX` is inserted into a `gist__int_ops` index (Alexander Lakhin, Tom Lane)

- Report a better error when `contrib/pageinspect`'s `hash_bitmap_info()` function is applied to a partitioned hash index (Alexander Lakhin, Michael Paquier)

- Report a better error when `contrib/pgstattuple`'s `pgstathashindex()` function is applied to a partitioned hash index (Alexander Lakhin)

- On Windows, suppress autorun options when launching subprocesses in pg_ctl and pg_regress (Kyotaro Horiguchi)

  When launching a child process via `cmd.exe`, pass the `/D` flag to prevent executing any autorun commands specified in the registry. This avoids possibly-surprising side effects.

- Move `is_valid_ascii()` from `mb/pg_wchar.h` to `utils/ascii.h` (Jubilee Young)

  This change avoids the need to include `<simd.h>` in `pg_wchar.h`, which was causing problems for some third-party code.

- Fix compilation failures with libxml2 version 2.12.0 and later (Tom Lane)

- Fix compilation failure of `WAL_DEBUG` code on Windows (Bharath Rupireddy)

- Suppress compiler warnings from Python's header files (Peter Eisentraut, Tom Lane)

  Our preferred compiler options provoke warnings about constructs appearing in recent versions of Python's header files. When using gcc, we can suppress these warnings with a pragma.

- Avoid deprecation warning when compiling with LLVM 18 (Thomas Munro)

- Update time zone data files to tzdata release 2024a for DST law changes in Greenland, Kazakhstan, and Palestine, plus corrections for the Antarctic stations Casey and Vostok. Also historical corrections for Vietnam, Toronto, and Miquelon. (Tom Lane)

E.15. Release 15.5

Release date: 2023-11-09

This release contains a variety of fixes from 15.4. For information about new features in major release 15, see Section E.20.

E.15.1. Migration to Version 15.5

A dump/restore is not required for those running 15.X.

However, several mistakes have been discovered that could lead to certain types of indexes yielding wrong search results or being unnecessarily inefficient. It is advisable to `REINDEX` potentially-affected indexes after installing this update. See the fourth through seventh changelog entries below.

Also, if you are upgrading from a version earlier than 15.4, see Section E.16.

E.15.2. Changes

- Fix handling of unknown-type arguments in `DISTINCT "any"` aggregate functions (Tom Lane)

  This error led to a `text`-type value being interpreted as an `unknown`-type value (that is, a zero-terminated string) at runtime. This could result in disclosure of server memory following the `text` value.

  The PostgreSQL Project thanks Jingzhou Fu for reporting this problem. (CVE-2023-5868)

- Detect integer overflow while computing new array dimensions (Tom Lane)

  When assigning new elements to array subscripts that are outside the current array bounds, an undetected integer overflow could occur in edge cases. Memory stomps that are potentially exploitable for arbitrary code execution are possible, and so is disclosure of server memory.

  The PostgreSQL Project thanks Pedro Gallegos for reporting this problem. (CVE-2023-5869)

- Prevent the `pg_signal_backend` role from signalling background workers and autovacuum processes (Noah Misch, Jelte Fennema-Nio)

  The documentation says that `pg_signal_backend` cannot issue signals to superuser-owned processes. It was able to signal these background processes, though, because they advertise a role OID of zero. Treat that as indicating superuser ownership. The security implications of cancelling one of these process types are fairly small so far as the core code goes (we'll just start another one), but extensions might add background workers that are more vulnerable.

  Also ensure that the `is_superuser` parameter is set correctly in such processes. No specific security consequences are known for that oversight, but it might be significant for some extensions.

  The PostgreSQL Project thanks Hemanth Sandrana and Mahendrakar Srinivasarao for reporting this problem. (CVE-2023-5870)

- Fix misbehavior during recursive page split in GiST index build (Heikki Linnakangas)

  Fix a case where the location of a page downlink was incorrectly tracked, and introduce some logic to allow recovering from such situations rather than silently doing the wrong thing. This error could result in incorrect answers from subsequent index searches. It may be advisable to reindex all GiST indexes after installing this update.

- Prevent de-duplication of btree index entries for `interval` columns (Noah Misch)

  There are `interval` values that are distinguishable but compare equal, for example `24:00:00` and `1 day`. This breaks assumptions made by btree de-duplication, so `interval` columns need to be excluded from de-duplication. This oversight can cause incorrect results from index-only scans. Moreover, after updating amcheck will report an error for almost all such indexes. Users should reindex any btree indexes on `interval` columns.

- Process `date` values more sanely in BRIN `datetime_minmax_multi_ops` indexes (Tomas Vondra)

The distance calculation for dates was backward, causing poor decisions about which entries to merge. The index still produces correct results, but is much less efficient than it should be. Reindexing BRIN `minmax_multi` indexes on `date` columns is advisable.

- Process large `timestamp` and `timestamptz` values more sanely in BRIN `datetime_minmax_multi_ops` indexes (Tomas Vondra)

  Infinities were mistakenly treated as having distance zero rather than a large distance from other values, causing poor decisions about which entries to merge. Also, finite-but-very-large values (near the endpoints of the representable timestamp range) could result in internal overflows, again causing poor decisions. The index still produces correct results, but is much less efficient than it should be. Reindexing BRIN `minmax_multi` indexes on `timestamp` and `timestamptz` columns is advisable if the column contains, or has contained, infinities or large finite values.

- Avoid calculation overflows in BRIN `interval_minmax_multi_ops` indexes with extreme interval values (Tomas Vondra)

  This bug might have caused unexpected failures while trying to insert large interval values into such an index.

- Fix partition step generation and runtime partition pruning for hash-partitioned tables with multiple partition keys (David Rowley)

  Some cases involving an `IS NULL` condition on one of the partition keys could result in a crash.

- Fix inconsistent rechecking of concurrently-updated rows during `MERGE` (Dean Rasheed)

  In `READ COMMITTED` mode, an update that finds that its target row was just updated by a concurrent transaction will recheck the query's `WHERE` conditions on the updated row. `MERGE` failed to ensure that the proper rows of other joined tables were used during this recheck, possibly resulting in incorrect decisions about whether the newly-updated row should be updated again by `MERGE`.

- Correctly identify the target table in an inherited `UPDATE`/`DELETE`/`MERGE` even when the parent table is excluded by constraints (Amit Langote, Tom Lane)

  If the initially-named table is excluded by constraints, but not all its inheritance descendants are, the first non-excluded descendant was identified as the primary target table. This would lead to firing statement-level triggers associated with that table, rather than the initially-named table as should happen. In v16, the same oversight could also lead to « invalid perminfoindex 0 in RTE with relid NNNN » errors.

- Fix edge case in btree mark/restore processing of ScalarArrayOpExpr clauses (Peter Geoghegan)

  When restoring an indexscan to a previously marked position, the code could miss required setup steps if the scan had advanced exactly to the end of the matches for a ScalarArrayOpExpr (that is, an `indexcol = ANY(ARRAY[])`) clause. This could result in missing some rows that should have been fetched.

- Fix intra-query memory leak in Memoize execution (Orlov Aleksej, David Rowley)

- Fix intra-query memory leak when a set-returning function repeatedly returns zero rows (Tom Lane)

- Don't crash if `cursor_to_xmlschema()` is applied to a non-data-returning Portal (Boyu Yang)

- Throw the intended error if `pgrowlocks()` is applied to a partitioned table (David Rowley)

  Previously, a not-on-point complaint « only heap AM is supported » would be raised.

- Handle invalid indexes more cleanly in assorted SQL functions (Noah Misch)

Report an error if `pgstatindex()`, `pgstatginindex()`, `pgstathashindex()`, or `pgstattuple()` is applied to an invalid index. If `brin_desummarize_range()`, `brin_summarize_new_values()`, `brin_summarize_range()`, or `gin_clean_pending_list()` is applied to an invalid index, do nothing except to report a debug-level message. Formerly these functions attempted to process the index, and might fail in strange ways depending on what the failed `CREATE INDEX` had left behind.

- Fix `pg_stat_reset_single_table_counters()` to do the right thing for a shared catalog (Masahiro Ikeda)

  Previously the reset would be ineffective.

- Avoid premature memory allocation failure with long inputs to `to_tsvector()` (Tom Lane)

- Fix over-allocation of the constructed `tsvector` in `tsvectorrecv()` (Denis Erokhin)

  If the incoming vector includes position data, the binary receive function left wasted space (roughly equal to the size of the position data) in the finished `tsvector`. In extreme cases this could lead to « maximum total lexeme length exceeded » failures for vectors that were under the length limit when emitted. In any case it could lead to wasted space on-disk.

- Fix incorrect coding in `gtsvector_picksplit()` (Alexander Lakhin)

  This could lead to poor page-split decisions in GiST indexes on `tsvector` columns.

- Improve checks for corrupt PGLZ compressed data (Flavien Guedez)

- In `COPY FROM`, fail cleanly when an unsupported encoding conversion is needed (Tom Lane)

  Recent refactoring accidentally removed the intended error check for this, such that it ended in « cache lookup failed for function 0 » instead of a useful error message.

- Avoid crash in `EXPLAIN` if a parameter marked to be displayed by `EXPLAIN` has a NULL boot-time value (Xing Guo, Aleksander Alekseev, Tom Lane)

  No built-in parameter fits this description, but an extension could define such a parameter.

- Ensure we have a snapshot while dropping `ON COMMIT DROP` temp tables (Tom Lane)

  This prevents possible misbehavior if any catalog entries for the temp tables have fields wide enough to require toasting (such as a very complex `CHECK` condition).

- Avoid improper response to shutdown signals in child processes just forked by `system()` (Nathan Bossart)

  This fix avoids a race condition in which a child process that has been forked off by `system()`, but hasn't yet exec'd the intended child program, might receive and act on a signal intended for the parent server process. That would lead to duplicate cleanup actions being performed, which will not end well.

- Cope with torn reads of `pg_control` in frontend programs (Thomas Munro)

  On some file systems, reading `pg_control` may not be an atomic action when the server concurrently writes that file. This is detectable via a bad CRC. Retry a few times to see if the file becomes valid before we report error.

- Avoid torn reads of `pg_control` in relevant SQL functions (Thomas Munro)

  Acquire the appropriate lock before reading `pg_control`, to ensure we get a consistent view of that file.

- Avoid integer overflow when computing size of backend activity string array (Jakub Wartak)

  On 64-bit machines we will allow values of `track_activity_query_size` large enough to cause 32-bit overflow when multiplied by the allowed number of connections. The code actually allocating the per-backend local array was careless about this though, and allocated the array incorrectly.

- Fix briefly showing inconsistent progress statistics for `ANALYZE` on inherited tables (Heikki Linnakangas)

  The block-level counters should be reset to zero at the same time we update the current-relation field.

- Fix the background writer to report any WAL writes it makes to the statistics counters (Nazir Bilal Yavuz)

- Fix confusion about forced-flush behavior in `pgstat_report_wal()` (Ryoga Yoshida, Michael Paquier)

  This could result in some statistics about WAL I/O being forgotten in a shutdown.

- Track the dependencies of cached `CALL` statements, and re-plan them when needed (Tom Lane)

  DDL commands, such as replacement of a function that has been inlined into a `CALL` argument, can create the need to re-plan a `CALL` that has been cached by PL/pgSQL. That was not happening, leading to misbehavior or strange errors such as « cache lookup failed ».

- Avoid a possible pfree-a-NULL-pointer crash after an error in OpenSSL connection setup (Sergey Shinderuk)

- Track nesting depth correctly when inspecting `RECORD`-type Vars from outer query levels (Richard Guo)

  This oversight could lead to assertion failures, core dumps, or « bogus varno » errors.

- Track hash function and negator function dependencies of ScalarArrayOpExpr plan nodes (David Rowley)

  In most cases this oversight was harmless, since these functions would be unlikely to disappear while the node's original operator remains present.

- Fix error-handling bug in `RECORD` type cache management (Thomas Munro)

  An out-of-memory error occurring at just the wrong point could leave behind inconsistent state that would lead to an infinite loop.

- Fix assertion failure when logical decoding is retried in the same session after an error (Hou Zhijie)

- Treat out-of-memory failures as fatal while reading WAL (Michael Paquier)

  Previously this would be treated as a bogus-data condition, leading to the conclusion that we'd reached the end of WAL, which is incorrect and could lead to inconsistent WAL replay.

- Fix possible recovery failure due to trying to allocate memory based on a bogus WAL record length field (Thomas Munro, Michael Paquier)

- Fix race condition in database dropping that could lead to the autovacuum launcher getting stuck (Andres Freund, Will Mortensen, Jacob Speidel)

  The race could lead to a statistics entry for the removed database remaining present, confusing the launcher's selection of which database to process.

- Fix datatype size confusion in logical tape management (Ranier Vilela)

Integer overflow was possible on platforms where long is wider than int, although it would take a multiple-terabyte temporary file to cause a problem.

- Avoid unintended close of syslogger process's stdin (Heikki Linnakangas)
- Avoid doing plan cache revalidation of utility statements that do not receive interesting processing during parse analysis (Tom Lane)

  Aside from saving a few cycles, this prevents failure after a cache invalidation for statements that must not set a snapshot, such as `SET TRANSACTION ISOLATION LEVEL`.
- Keep by-reference `attmissingval` values in a long-lived context while they are being used (Andrew Dunstan)

  This avoids possible use of dangling pointers when a tuple slot outlives the tuple descriptor with which its value was constructed.
- Recalculate the effective value of `search_path` after `ALTER ROLE` (Jeff Davis)

  This ensures that after renaming a role, the meaning of the special string `$user` is re-determined.
- Fix « could not duplicate handle » error occurring on Windows when `min_dynamic_shared_memory` is set above zero (Thomas Munro)
- Fix order of operations in `GenericXLogFinish` (Jeff Davis)

  This code violated the conditions required for crash safety by writing WAL before marking changed buffers dirty. No core code uses this function, but extensions do (`contrib/bloom` does, for example).
- Remove incorrect assertion in PL/Python exception handling (Alexander Lakhin)
- Fix assertion failure in pg_dump when it's asked to dump the `pg_catalog` schema (Peter Eisentraut)
- Fix pg_restore so that selective restores will include both table-level and column-level ACLs for selected tables (Euler Taveira, Tom Lane)

  Formerly, only the table-level ACL would get restored if both types were present.
- Add logic to pg_upgrade to check for use of `abstime`, `reltime`, and `tinterval` data types (Álvaro Herrera)

  These obsolete data types were removed in PostgreSQL version 12, so check to make sure they aren't present in an older database before claiming it can be upgraded.
- Avoid generating invalid temporary slot names in pg_basebackup (Jelte Fennema)

  This has only been seen to occur when the server connection runs through pgbouncer.
- Avoid false « too many client connections » errors in pgbench on Windows (Noah Misch)
- In `contrib/amcheck`, do not report interrupted page deletion as corruption (Noah Misch)

  This fix prevents false-positive reports of « the first child of leftmost target page is not leftmost of its level », « block NNNN is not leftmost » or « left link/right link pair in index XXXX not in agreement ». They appeared if amcheck ran after an unfinished btree index page deletion and before `VACUUM` had cleaned things up.
- Fix failure of `contrib/btree_gin` indexes on `interval` columns, when an indexscan using the `<` or `<=` operator is performed (Dean Rasheed)

Such an indexscan failed to return all the entries it should.

- Add support for LLVM 16 and 17 (Thomas Munro, Dmitry Dolgov)

- Suppress assorted build-time warnings on recent macOS (Tom Lane)

  Xcode 15 (released with macOS Sonoma) changed the linker's behavior in a way that causes many duplicate-library warnings while building PostgreSQL. These were harmless, but they're annoying so avoid citing the same libraries twice. Also remove use of the `-multiply_defined suppress` linker switch, which apparently has been a no-op for a long time, and is now actively complained of.

- When building `contrib/unaccent`'s rules file, fall back to using `python` if `--with-python` was not given and make variable `PYTHON` was not set (Japin Li)

- Remove `PHOT` (Phoenix Islands Time) from the default timezone abbreviations list (Tom Lane)

  Presence of this abbreviation in the default list can cause failures on recent Debian and Ubuntu releases, as they no longer install the underlying tzdb entry by default. Since this is a made-up abbreviation for a zone with a total human population of about two dozen, it seems unlikely that anyone will miss it. If someone does, they can put it back via a custom abbreviations file.

E.16. Release 15.4

Release date: 2023-08-10

This release contains a variety of fixes from 15.3. For information about new features in major release 15, see Section E.20.

E.16.1. Migration to Version 15.4

A dump/restore is not required for those running 15.X.

However, if you use BRIN indexes, it may be advisable to reindex them; see the third changelog entry below.

Also, if you are upgrading from a version earlier than 15.1, see Section E.19.

E.16.2. Changes

- Disallow substituting a schema or owner name into an extension script if the name contains a quote, backslash, or dollar sign (Noah Misch)

  This restriction guards against SQL-injection hazards for trusted extensions.

  The PostgreSQL Project thanks Micah Gates, Valerie Woolard, Tim Carey-Smith, and Christoph Berg for reporting this problem. (CVE-2023-39417)

- Fix `MERGE` to enforce row security policies properly (Dean Rasheed)

  When `MERGE` performs an `UPDATE` action, it should enforce any `UPDATE` or `SELECT` RLS policies defined on the target table, to be consistent with the way that a plain `UPDATE` with a `WHERE` clause works. Instead it was enforcing `INSERT` RLS policies for both `INSERT` and `UPDATE` actions.

  In addition, when `MERGE` performs a `DO NOTHING` action, it applied the target table's `DELETE` RLS policies to existing rows, even though those rows are not being deleted. While it's not a security problem, this could result in unwanted errors.

The PostgreSQL Project thanks Dean Rasheed for reporting this problem. (CVE-2023-39418)

- Fix confusion between empty (no rows) ranges and all-NULL ranges in BRIN indexes, as well as incorrect merging of all-NULL summaries (Tomas Vondra)

  Each of these oversights could result in forgetting that a BRIN index range contains any NULL values, potentially allowing subsequent queries that should return NULL values to miss doing so.

  This fix will not in itself correct faulty BRIN entries. It's recommended to `REINDEX` any BRIN indexes that may be used to search for nulls.

- Avoid leaving a corrupted database behind when `DROP DATABASE` is interrupted (Andres Freund)

  If `DROP DATABASE` was interrupted after it had already begun taking irreversible steps, the target database remained accessible (because the removal of its `pg_database` row would roll back), but it would have corrupt contents. Fix by marking the database as inaccessible before we begin to perform irreversible operations. A failure after that will leave the database still partially present, but nothing can be done with it except to issue another `DROP DATABASE`.

- Ensure that partitioned indexes are correctly marked as valid or not at creation (Michael Paquier)

  If a new partitioned index matches an existing but invalid index on one of the partitions, the partitioned index could end up being marked valid prematurely. This could lead to misbehavior or assertion failures in subsequent queries on the partitioned table.

- Ignore invalid child indexes when matching partitioned indexes to child indexes during `ALTER TABLE ATTACH PARTITION` (Michael Paquier)

  Such an index will now be ignored, and a new child index created instead.

- Fix possible failure when marking a partitioned index valid after all of its partitions have been attached (Michael Paquier)

  The update of the index's `pg_index` entry could use stale data for other columns. One reported symptom is an « attempted to update invisible tuple » error.

- Fix `ALTER EXTENSION SET SCHEMA` to complain if the extension contains any objects outside the extension's schema (Michael Paquier, Heikki Linnakangas)

  Erroring out if the extension contains objects in multiple schemas was always intended; but the check was mis-coded so that it would fail to detect some cases, leading to surprising behavior.

- Fix tracking of tables' access method dependencies (Michael Paquier)

  `ALTER TABLE ... SET ACCESS METHOD` failed to update relevant `pg_depend` entries when changing a table's access method. When using non-built-in access methods, this creates a risk that an access method could be dropped even though tables still depend on it. This fix corrects the logic in `ALTER TABLE`, but it will not adjust any already-missing `pg_depend` entries.

- Don't use partial unique indexes for uniqueness proofs in the planner (David Rowley)

  This could give rise to incorrect plans, since the presumed uniqueness of rows read from a table might not hold if the index in question isn't used to scan the table.

- Don't Memoize lateral joins with volatile join conditions (Richard Guo)

  Applying Memoize to a sub-plan that contains volatile filter conditions is likely to lead to wrong answers. The check to avoid doing this missed some cases that can arise when using `LATERAL`.

- Avoid producing incorrect plans for foreign joins with pseudoconstant join clauses (Etsuro Fujita)

The planner currently lacks support for attaching pseudoconstant join clauses to a pushed-down remote join, so disable generation of remote joins in such cases. (A better solution will require ABI-breaking changes of planner data structures, so it will have to wait for a future major release.)

- Correctly handle sub-SELECTs in RLS policy expressions and security-barrier views when expanding rule actions (Tom Lane)
- Fix race conditions in conflict detection for `SERIALIZABLE` isolation mode (Thomas Munro)

  Conflicts could be missed when using bitmap heap scans, when using GIN indexes, and when examining an initially-empty btree index. All these cases could lead to serializability failures due to improperly allowing conflicting transactions to commit.
- Fix misbehavior of EvalPlanQual checks with inherited or partitioned target tables (Tom Lane)

  This oversight could lead to update or delete actions in `READ COMMITTED` isolation mode getting performed when they should have been skipped because of a conflicting concurrent update.
- Fix hash join with an inner-side hash key that contains Params coming from an outer nested loop (Tom Lane)

  When rescanning the join after the values of such Params have changed, we must rebuild the hash table, but neglected to do so. This could result in missing join output rows.
- Fix intermittent failures when trying to update a field of a composite column (Tom Lane)

  If the overall value of the composite column is wide enough to require out-of-line toasting, then an unluckily-timed cache flush could cause errors or server crashes.
- Prevent query-lifespan memory leaks in some `UPDATE` queries with triggers (Tomas Vondra)
- Prevent query-lifespan memory leaks when an Incremental Sort plan node is rescanned (James Coleman, Laurenz Albe, Tom Lane)
- Accept fractional seconds in the input to `jsonpath`'s `datetime()` method (Tom Lane)
- Prevent stack-overflow crashes with very complex text search patterns (Tom Lane)
- Allow tokens up to 10240 bytes long in `pg_hba.conf` and `pg_ident.conf` (Tom Lane)

  The previous limit of 256 bytes has been found insufficient for some use-cases.
- Ensure that all existing placeholders are checked for matches when an extension declares its GUC prefix to be reserved (Karina Litskevich, Ekaterina Sokolova)

  Faulty loop logic could cause some entries to be skipped.
- Fix mishandling of C++ out-of-memory conditions (Heikki Linnakangas)

  If JIT is in use, running out of memory in a C++ `new` call would lead to a PostgreSQL FATAL error, instead of the expected C++ exception.
- Fix rare null-pointer crash in `plancache.c` (Tom Lane)
- Avoid leaking a stats entry for a subscription when it is dropped (Masahiko Sawada)
- Avoid losing track of possibly-useful shared memory segments when a page free results in coalescing ranges of free space (Dongming Liu)

  Ensure that the segment is moved into the appropriate « bin » for its new amount of free space, so that it will be found by subsequent searches.

- Allow `VACUUM` to continue after detecting certain types of b-tree index corruption (Peter Geoghegan)

  If an invalid sibling-page link is detected, log the issue and press on, rather than throwing an error as before. Nothing short of `REINDEX` will fix the broken index, but preventing `VACUUM` from completing until that is done risks making matters far worse.

- Ensure that `WrapLimitsVacuumLock` is released after `VACUUM` detects invalid data in `pg_database.datfrozenxid` or `pg_database.datminmxid` (Andres Freund)

  Failure to release this lock could lead to a deadlock later, although the lock would be cleaned up if the session exits or encounters some other error.

- Avoid double replay of prepared transactions during crash recovery (suyu.cmj, Michael Paquier)

  After a crash partway through a checkpoint with some two-phase transaction state data already flushed to disk by this checkpoint, crash recovery could attempt to replay the prepared transaction(s) twice, leading to a fatal error such as « lock is already held » in the startup process.

- Ensure that a newly created, but still empty table is `fsync`'ed at the next checkpoint (Heikki Linnakangas)

  Without this, if there is an operating system crash causing the empty file to disappear, subsequent operations on the table might fail with « could not open file » errors.

- Ensure that creation of the init fork of an unlogged index is WAL-logged (Heikki Linnakangas)

  While an unlogged index's main data fork is not WAL-logged, its init fork should be, to ensure that we have a consistent state to restore the index to after a crash. This step was missed if the init fork contains no data, which is a case not used by any standard index AM; but perhaps some extension behaves that way.

- Silence bogus « missing contrecord » errors (Thomas Munro)

  Treat this case as plain end-of-WAL to avoid logging inaccurate complaints from pg_waldump and walsender.

- Fix overly strict assertion in `jsonpath` code (David Rowley)

  This assertion failed if a query applied the `.type()` operator to a `like_regex` result. There was no bug in non-assert builds.

- Avoid assertion failure when processing an empty statement via the extended query protocol in an already-aborted transaction (Tom Lane)

- Avoid assertion failure when the `stats_fetch_consistency` setting is changed intra-transaction (Kyotaro Horiguchi)

- Fix `contrib/fuzzystrmatch`'s Soundex `difference()` function to handle empty input sanely (Alexander Lakhin, Tom Lane)

  An input string containing no alphabetic characters resulted in unpredictable output.

- Tighten whitespace checks in `contrib/hstore` input (Evan Jones)

  In some cases, characters would be falsely recognized as whitespace and hence discarded.

- Disallow oversize input arrays with `contrib/intarray`'s `gist__int_ops` index opclass (Ankit Kumar Pandey, Alexander Lakhin)

  Previously this code would report a `NOTICE` but press on anyway, creating an invalid index entry that presents a risk of crashes when the index is read.

- Avoid useless double decompression of GiST index entries in `contrib/intarray` (Konstantin Knizhnik, Matthias van de Meent, Tom Lane)

- Fix `contrib/pageinspect`'s `gist_page_items()` function to work when there are included index columns (Alexander Lakhin, Michael Paquier)

  Previously, if the index has included columns, `gist_page_items()` would fail to display those values on index leaf pages, or crash outright on non-leaf pages.

- In psql, ignore the `PSQL_WATCH_PAGER` environment variable when stdin/stdout are not a terminal (Tom Lane)

  This corresponds to the treatment of `PSQL_PAGER` in commands besides `\watch`.

- Fix pg_dump to correctly handle new-style SQL-language functions whose bodies require parse-time dependencies on unique indexes (Tom Lane)

  Such cases can arise from `GROUP BY` and `ON CONFLICT` clauses, for example. The function must then be postponed until after the unique index in the dump output, but pg_dump did not do that and instead printed a warning about « could not resolve dependency loop ».

- Improve pg_dump's display of details about dependency-loop problems (Tom Lane)

- Avoid crash in pgbench with an empty pipeline and prepared mode (Álvaro Herrera)

- Ensure that `pg_index.indisreplident` is kept up-to-date in relation cache entries (Shruthi Gowda)

  This value could be stale in some cases. There is no core code that relies on the relation cache's copy, so this is only a latent bug as far as Postgres itself is concerned; but there may be extensions for which it is a live bug.

- Fix make_etags script to work with non-Exuberant ctags (Masahiko Sawada)

E.17. Release 15.3

Release date: 2023-05-11

This release contains a variety of fixes from 15.2. For information about new features in major release 15, see Section E.20.

E.17.1. Migration to Version 15.3

A dump/restore is not required for those running 15.X.

However, if you are upgrading from a version earlier than 15.1, see Section E.19.

E.17.2. Changes

- Prevent `CREATE SCHEMA` from defeating changes in `search_path` (Alexander Lakhin)

  Within a `CREATE SCHEMA` command, objects in the prevailing `search_path`, as well as those in the newly-created schema, would be visible even within a called function or script that attempted to set a secure `search_path`. This could allow any user having permission to create a schema to hijack the privileges of a security definer function or extension script.

  The PostgreSQL Project thanks Alexander Lakhin for reporting this problem. (CVE-2023-2454)

- Enforce row-level security policies correctly after inlining a set-returning function (Stephen Frost, Tom Lane)

  If a set-returning SQL-language function refers to a table having row-level security policies, and it can be inlined into a calling query, those RLS policies would not get enforced properly in some cases involving re-using a cached plan under a different role. This could allow a user to see or modify rows that should have been invisible.

  The PostgreSQL Project thanks Wolfgang Walther for reporting this problem. (CVE-2023-2455)

- Fix potential corruption of the template (source) database after `CREATE DATABASE` with the `STRATEGY WAL_LOG` option (Nathan Bossart, Ryo Matsumura)

  Improper buffer handling created a risk that any later modification of the template's `pg_class` catalog would be lost.

- Fix memory leakage and unnecessary disk reads during `CREATE DATABASE` with the `STRATEGY WAL_LOG` option (Andres Freund)

- Avoid crash when the new schema name is omitted in `CREATE SCHEMA` (Michael Paquier)

  The SQL standard allows writing `CREATE SCHEMA AUTHORIZATION` *`owner_name`*, with the schema name defaulting to *`owner_name`*. However some code paths expected the schema name to be present and would fail.

- Fix various planner failures with `MERGE` commands (Tom Lane)

  Planning could fail with errors like « variable not found in subplan target list » or « PlaceHolderVar found where not expected ».

- Fix the row count reported by `MERGE` for some corner cases (Dean Rasheed)

  The row count reported in the command tag counted rows that actually hadn't been modified due to a `BEFORE ROW` trigger returning NULL. This is inconsistent with what happens in plain `UPDATE` or `DELETE`, so change it to not count such rows. Also, avoid counting a row twice when `MERGE` moves it into a different partition of a partitioned table.

- Fix `MERGE` problems with concurrent updates (Dean Rasheed, Álvaro Herrera)

  Some cases misbehaved if a row to be updated or deleted by `MERGE` had just been updated by a concurrent transaction. This could lead to a crash, or the wrong merge action being executed, or no action at all.

- Add support for decompiling `MERGE` commands (Álvaro Herrera)

  This was overlooked when `MERGE` was added, but it's essential support for `MERGE` in new-style SQL functions.

- Fix enabling/disabling of foreign-key triggers in partitioned tables (Tom Lane)

  `ALTER TABLE ... ENABLE/DISABLE TRIGGER` failed if applied to a partitioned table's foreign-key enforcement triggers, because it tried to locate the clone triggers for the partitions by name, and they do not have the same name. Locate them by parent-trigger OID instead.

- Disallow altering composite types that are stored in indexes (Tom Lane)

  `ALTER TYPE` disallows non-binary-compatible modifications of composite types if they are stored in any table columns. (Perhaps that will be allowed someday, but it hasn't happened yet; the locking implications of rewriting many tables are daunting.) We overlooked the possibility that an index might contain a composite type that doesn't also appear in its table.

- Disallow system columns as elements of foreign keys (Tom Lane)

Since the removal of OID as a system column, there is no plausible use-case for this, and various bits of code no longer support it. Disallow it rather than trying to fix all the cases.

- Ensure that `COPY TO` from an RLS-enabled parent table does not copy any rows from child tables (Antonin Houska)

  The documentation is quite clear that `COPY TO` copies rows from only the named table, not any inheritance children it may have. However, if row-level security was enabled on the table then this stopped being true.

- Avoid possible crash when `array_position()` or `array_positions()` is passed an empty array (Tom Lane)

- Fix possible out-of-bounds fetch in `to_char()` (Tom Lane)

  With bad luck this could have resulted in a server crash.

- Avoid buffer overread in `translate()` function (Daniil Anisimov)

  When using the deletion feature, the function might fetch the byte just after the input string, creating a small risk of crash.

- Adjust text-search-related character classification logic to correctly detect whether the prevailing locale is `C` (Jeff Davis)

  This code got confused if the database's default collation uses ICU.

- Avoid possible crash on empty input for type `interval` (Tom Lane)

- Re-allow exponential notation in ISO-8601 interval fields (Tom Lane)

  Interval input like `P0.1e10D` isn't officially sanctioned by ISO-8601, but we accepted it for a long time before version 15, so re-allow it.

- Fix error cursor setting for parse errors in JSON string literals (Tom Lane)

  Most cases in which a syntax error is detected in a string literal within a JSON value failed to set the error cursor appropriately. This led at least to an unhelpful error message (pointing to the token before the string, rather than the actual trouble spot), and could even result in a crash in v14 and later.

- Fix data corruption due to `vacuum_defer_cleanup_age` being larger than the current 64-bit xid (Andres Freund)

  In v14 and later with non-default settings of `vacuum_defer_cleanup_age`, it was possible to compute a very large vacuum cleanup horizon xid, leading to vacuum removing rows that are still live. v12 and v13 have a lesser form of the same problem affecting only GiST indexes, which could lead to index pages getting recycled too early.

- Fix parser's failure to detect some cases of improperly-nested aggregates (Tom Lane)

  This oversight could lead to executor failures for queries that should have been rejected as invalid.

- Fix data structure corruption during parsing of serial `SEQUENCE NAME` options (David Rowley)

  This can lead to trouble if an event trigger captures the corrupted parse tree.

- Correctly update plan nodes' parallel-safety markings when moving initplans from one node to another (Tom Lane)

  This planner oversight could lead to « subplan was not initialized » errors at runtime.

- Avoid failure with PlaceHolderVars in extended-statistics code (Tom Lane)

  Use of dependency-type extended statistics could fail with « PlaceHolderVar found where not expected ».

- Fix incorrect tests for whether a qual clause applied to a subquery can be transformed into a window aggregate « run condition » within the subquery (David Rowley)

  A SubPlan within such a clause would cause assertion failures or incorrect answers, as would some other unusual cases.

- Disable the inverse-transition optimization for window aggregates when the call contains sub-SELECTs (David Rowley)

  This optimization requires that the aggregate's argument expressions have repeatable results, which might not hold for a sub-SELECT.

- Fix oversights in execution of nested `ARRAY[]` constructs (Alexander Lakhin, Tom Lane)

  Correctly detect overflow of the total space needed for the result array, avoiding a possible crash due to undersized output allocation. Also ensure that any trailing padding space in the result array is zeroed; while leaving garbage there is harmless for most purposes, it can result in odd behavior later.

- Prevent crash when updating a field within an array-of-domain-over-composite-type column (Dmitry Dolgov)

- Fix partition pruning logic for partitioning on boolean columns (David Rowley)

  Pruning with a condition like `boolcol IS NOT TRUE` was done incorrectly, leading to possibly not returning rows in which `boolcol` is NULL. Also, the rather unlikely case of partitioning on `NOT boolcol` was handled incorrectly.

- Fix race condition in per-batch cleanup during parallel hash join (Thomas Munro, Melanie Plageman)

  A crash was possible given unlucky timing and `parallel_leader_participation` = `off` (which is not the default).

- Recalculate `GENERATED` columns after an EvalPlanQual check (Tom Lane)

  In `READ COMMITTED` isolation mode, the effects of a row update might need to get reapplied to a newer version of the row than the query found originally. If so, we need to recompute any `GENERATED` columns, in case they depend on columns that were changed by the concurrent update.

- Fix memory leak in Memoize plan execution (David Rowley)

- Fix buffer refcount leak when using batched inserts for a foreign table included in a partitioned tree (Alexander Pyhalov)

- Restore support for sub-millisecond `vacuum_cost_delay` settings (Thomas Munro)

- Don't balance vacuum cost delay when a table has a per-relation `vacuum_cost_delay` setting of zero (Masahiko Sawada)

  Delay balancing is supposed to be disabled whenever autovacuum is processing a table with a per-relation `vacuum_cost_delay` setting, but this was done only for positive settings, not zero.

- Fix corner-case crashes when columns have been added to the end of a view (Tom Lane)

- Repair rare failure of MULTIEXPR_SUBLINK subplans in partitioned updates (Andres Freund, Tom Lane)

Use of the syntax `INSERT ... ON CONFLICT DO UPDATE SET (c1, ...) = (SELECT ...)` with a partitioned target table could result in failure if any child table is dissimilar from the parent (for example, different physical column order). This typically manifested as failure of consistency checks in the executor; but a crash or incorrect data updates are also possible.

- Fix handling of `DEFAULT` markers within a multi-row `INSERT ... VALUES` query on a view that has a `DO ALSO INSERT ... SELECT` rule (Dean Rasheed)

  Such cases typically failed with « unrecognized node type » errors or assertion failures.

- Support references to `OLD` and `NEW` within subqueries in rule actions (Dean Rasheed, Tom Lane)

  Such references are really lateral references, but the server could crash if the subquery wasn't explicitly marked with `LATERAL`. Arrange to do that implicitly when necessary.

- When decompiling a rule or SQL function body containing `INSERT`/`UPDATE`/`DELETE` within `WITH`, take care to print the correct alias for the target table (Tom Lane)

- Fix glitches in `SERIALIZABLE READ ONLY` optimization (Thomas Munro)

  Transactions already marked as « doomed » confused the safe-snapshot optimization for `SERIALIZABLE READ ONLY` transactions. The optimization was unnecessarily skipped in some cases. In other cases an assertion failure occurred (but there was no problem in non-assert builds).

- Avoid leaking cache callback slots in the `pgoutput` logical decoding plugin (Shi Yu)

  Multiple cycles of starting up and shutting down the plugin within a single session would eventually lead to an « out of relcache_callback_list slots » error.

- Avoid unnecessary calls to custom validators for index operator class options (Alexander Korotkov)

  This change fixes some cases where an unexpected error was thrown.

- Avoid useless work while scanning a multi-column BRIN index with multiple scan keys (Tomas Vondra)

  The existing code effectively considered only the last scan key while deciding whether a range matched, thus usually scanning more of the index than it needed to.

- Fix netmask handling in BRIN inet_minmax_multi_ops opclass (Tomas Vondra)

  This error triggered an assertion failure in assert-enabled builds, but is mostly harmless in production builds.

- Fix dereference of dangling pointer during buffering build of a GiST index (Alexander Lakhin)

  This error seems to usually be harmless in production builds, as the fetched value is noncritical; but in principle it could cause a server crash.

- Ignore dropped columns and generated columns during logical replication of an update or delete action (Onder Kalaci, Shi Yu)

  Replication with the `REPLICA IDENTITY FULL` option failed if the table contained such columns.

- Correct the name of the wait event for SLRU buffer I/O for commit timestamps (Alexander Lakhin)

  This wait event is named `CommitTsBuffer` according to the documentation, but the code had it as `CommitTSBuffer`. Change the code to match the documentation, as that way is more consistent with the naming of related wait events.

- Re-activate reporting of wait event `SLRUFlushSync` (Thomas Munro)

  Reporting of this type of wait was accidentally removed in code refactoring.

- Avoid possible underflow when calculating how many WAL segments to keep (Kyotaro Horiguchi)

  This could result in not honoring `wal_keep_size` accurately.

- Disable startup progress reporting overhead in standby mode (Bharath Rupireddy)

  In standby mode, we don't actually report progress of recovery, but we were doing work to track it anyway.

- Support RSA-PSS certificates with SCRAM-SHA-256 channel binding (Jacob Champion, Heikki Linnakangas)

  This feature requires building with OpenSSL 1.1.1 or newer. Both the server and libpq are affected.

- Avoid race condition with process ID tracking on Windows (Thomas Munro)

  The operating system could recycle a PID before the postmaster observed that that child process was gone. This could lead to tracking more than one child with the same PID, resulting in confusion.

- Fix `list_copy_head()` to work correctly on an empty List (David Rowley)

  This case is not known to be reached by any core PostgreSQL code, but extensions might rely on it working.

- Add missing cases to `SPI_result_code_string()` (Dean Rasheed)

- Fix erroneous Valgrind markings in `AllocSetRealloc()` (Karina Litskevich)

  In the unusual case where the size of a large (>8kB) palloc chunk is decreased, a Valgrind-aware build would mismark the defined-ness state of the memory released from the chunk, possibly causing incorrect results during Valgrind testing.

- Fix assertion failure for `MERGE` into a partitioned table with row-level security enabled (Dean Rasheed)

- Avoid assertion failure when decoding a transactional logical replication message (Tomas Vondra)

- Avoid locale sensitivity when processing regular expression escapes (Jeff Davis)

  A backslash followed by a non-ASCII character could sometimes cause an assertion failure, depending on the prevailing locale.

- Avoid trying to write an empty WAL record in `log_newpage_range()` when the last few pages in the specified range are empty (Matthias van de Meent)

  It is not entirely clear whether this case is reachable in released branches, but if it is then an assertion failure could occur.

- Fix session-lifespan memory leakage in plpgsql `DO` blocks that use cast expressions (Ajit Awekar, Tom Lane)

- Tighten array dimensionality checks when converting Perl list structures to multi-dimensional SQL arrays (Tom Lane)

  plperl could misbehave when the nesting of sub-lists is inconsistent so that the data does not represent a rectangular array of values. Such cases now produce errors, but previously they could result in a crash or garbage output.

- Tighten array dimensionality checks when converting Python list structures to multi-dimensional SQL arrays (Tom Lane)

  plpython could misbehave when dealing with empty sub-lists, or when the nesting of sub-lists is inconsistent so that the data does not represent a rectangular array of values. The former should result in an empty output array, and the latter in an error. But some cases resulted in a crash, and others in unexpected output.

- Fix unwinding of exception stack in plpython (Xing Guo)

  Some rare failure cases could return without cleaning up the PG_TRY exception stack, risking a crash if another error was raised before the next stack level was unwound.

- Fix inconsistent GSS-encryption error handling in libpq's `PQconnectPoll()` (Michael Paquier)

  With `gssencmode` set to `require`, the connection was not marked dead after a GSS initialization failure. Make it fail immediately, as the equivalent case for TLS encryption has long done.

- Fix possible data corruption in ecpg programs built with the `-C ORACLE` option (Kyotaro Horiguchi)

  When `ecpg_get_data()` is called with `varcharsize` set to zero, it could write a terminating zero character into the last byte of the preceding field, truncating the data in that field.

- Fix pg_dump so that partitioned tables that are hash-partitioned on an enum-type column can be restored successfully (Tom Lane)

  Since the hash codes for enum values depend on the OIDs assigned to the enum, they are typically different after a dump and restore, meaning that rows often need to go into a different partition than they were in originally. Users can work around that by specifying the `--load-via-partition-root` option; but since there is very little chance of success without that, teach pg_dump to apply it automatically to such tables.

  Also, fix pg_restore to not try to `TRUNCATE` target tables before restoring into them when `--load-via-partition-root` mode is used. This avoids a hazard of deadlocks and lost data.

- Correctly detect non-seekable files on Windows (Juan José Santamaría Flecha, Michael Paquier, Daniel Watzinger)

  This bug led to misbehavior when pg_dump writes to a pipe or pg_restore reads from one.

- In pgbench's « prepared » mode, prepare all the commands in a pipeline before starting the pipeline (Álvaro Herrera)

  This avoids a failure when a pgbench script tries to start a serializable transaction inside a pipeline.

- In `contrib/amcheck`'s heap checking code, deal correctly with tuples having zero xmin or xmax (Robert Haas)

- In `contrib/amcheck`, deal sanely with xids that appear to be before epoch zero (Andres Freund)

  In cases of corruption we might see a wrapped-around 32-bit xid that appears to be before the first xid epoch. Promoting such a value to 64-bit form produced a value far in the future, resulting in wrong reports. Return FirstNormalFullTransactionId in such cases so that things work reasonably sanely.

- In `contrib/basebackup_to_shell`, properly detect failure to open a pipe (Robert Haas)

- In `contrib/hstore_plpython`, avoid crashing if the Python value to be transformed isn't a mapping (Dmitry Dolgov, Tom Lane)

This should give an error, but Python 3 changed some APIs in a way that caused the check to misbehave, allowing a crash to ensue.

- Require the `siglen` option of a GiST index on an `ltree` column, if specified, to be a multiple of 4 (Alexander Korotkov)

  Other values result in misaligned accesses to index content, which is harmless on Intel-compatible hardware but can cause a crash on some other architectures.

- In `contrib/pageinspect`, add defenses against incorrect input for the `gist_page_items()` function (Dmitry Koval)

- Fix misbehavior in `contrib/pg_trgm` with an unsatisfiable regular expression (Tom Lane)

  A regex such as `$foo` is legal but unsatisfiable; the regex compiler recognizes that and produces an empty NFA graph. Attempting to optimize such a graph into a pg_trgm GIN or GiST index qualification resulted in accessing off the end of a work array, possibly leading to crashes.

- Fix handling of escape sequences in `contrib/postgres_fdw`'s `application_name` parameter (Kyotaro Horiguchi, Michael Paquier)

  The code to expand these could fail if executed in a background process, as for example during auto-analyze of a foreign table.

- In `contrib/pg_walinspect`, limit memory usage of `pg_get_wal_records_info()` (Bharath Rupireddy)

- Use the `--strip-unneeded` option when stripping static libraries with GNU-compatible strip (Tom Lane)

  Previously, `make install-strip` used the `-x` option in this case. This change avoids misbehavior of llvm-strip, and gives slightly smaller output as well.

- Stop recommending auto-download of DTD files for building the documentation, and indeed disable it (Aleksander Alekseev, Peter Eisentraut, Tom Lane)

  It appears no longer possible to build the SGML documentation without a local installation of the DocBook DTD files. Formerly xsltproc could download those files on-the-fly from sourceforge.net; but sourceforge.net now permits only HTTPS access, and no common version of xsltproc supports that. Hence, remove the bits of our documentation suggesting that that's possible or useful, and instead add xsltproc's `--nonet` option to the build recipes.

- When running TAP tests in PGXS builds, use a saner location for the temporary `portlock` directory (Peter Eisentraut)

  Place it under `tmp_check` in the build directory. With the previous coding, a PGXS build would try to place it in the installation directory, which is not necessarily writable.

- Update time zone data files to tzdata release 2023c for DST law changes in Egypt, Greenland, Morocco, and Palestine. (Tom Lane)

  When observing Moscow time, Europe/Kirov and Europe/Volgograd now use the abbreviations MSK/MSD instead of numeric abbreviations, for consistency with other timezones observing Moscow time. Also, America/Yellowknife is no longer distinct from America/Edmonton; this affects some pre-1948 timestamps in that area.

E.18. Release 15.2

Release date: 2023-02-09

This release contains a variety of fixes from 15.1. For information about new features in major release 15, see Section E.20.

E.18.1. Migration to Version 15.2

A dump/restore is not required for those running 15.X.

However, if you are upgrading from a version earlier than 15.1, see Section E.19.

E.18.2. Changes

- libpq can leak memory contents after GSSAPI transport encryption initiation fails (Jacob Champion)

  A modified server, or an unauthenticated man-in-the-middle, can send a not-zero-terminated error message during setup of GSSAPI (Kerberos) transport encryption. libpq will then copy that string, as well as following bytes in application memory up to the next zero byte, to its error report. Depending on what the calling application does with the error report, this could result in disclosure of application memory contents. There is also a small probability of a crash due to reading beyond the end of memory. Fix by properly zero-terminating the server message. (CVE-2022-41862)

- Fix calculation of which `GENERATED` columns need to be updated in child tables during an `UPDATE` on a partitioned table or inheritance tree (Amit Langote, Tom Lane)

  This fixes failure to update `GENERATED` columns that do not exist in the parent table, or that have different dependencies than are in the parent column's generation expression.

- Fix possible failure of `MERGE` to compute `GENERATED` columns (Dean Rasheed)

  When the first row-level action of the `MERGE` was an `UPDATE`, any subsequent `INSERT` actions would fail to compute `GENERATED` columns that were deemed unnecessary to compute for the `UPDATE` action (due to not depending on any of the `UPDATE` target columns).

- Fix `MERGE`'s check for unreachable `WHEN` clauses (Dean Rasheed)

  A `WHEN` clause following an unconditional `WHEN` clause should be rejected as unreachable, but this case was not always detected.

- Fix `MERGE`'s rule-detection test (Dean Rasheed)

  `MERGE` is not supported on tables with rules; but it also failed on tables that once had rules but no longer do.

- In `MERGE`, don't count a `DO NOTHING` action as a processed tuple (Álvaro Herrera)

  This makes the code's behavior match the documentation.

- Allow a `WITH RECURSIVE ... CYCLE` CTE to access its output column (Tom Lane)

  A reference to the `SET` column from within the CTE would fail with « cache lookup failed for type 0 ».

- Fix handling of pending inserts when doing a bulk insertion to a foreign table (Etsuro Fujita)

  In some cases pending insertions were not flushed to the FDW soon enough, leading to logical inconsistencies, for example `BEFORE ROW` triggers not seeing rows they should be able to see.

- Allow `REPLICA IDENTITY` to be set on an index that's not (yet) valid (Tom Lane)

  When pg_dump dumps a partitioned index that's marked `REPLICA IDENTITY`, it generates a command sequence that applies `REPLICA IDENTITY` before the partitioned index has been

marked valid, causing restore to fail. There seems no very good reason to prohibit doing it in that order, so allow it. The marking will have no effect anyway until the index becomes valid.

- Fix handling of `DEFAULT` markers in rules that perform an `INSERT` from a multi-row `VALUES` list (Dean Rasheed)

  In some cases a `DEFAULT` marker would not get replaced with the proper default-value expression, leading to an « unrecognized node type » error.

- Reject uses of undefined variables in `jsonpath` existence checks (Alexander Korotkov, David G. Johnston)

  While `jsonpath` match operators threw an error for an undefined variable in the path pattern, the existence operators silently treated it as a match.

- Fix `jsonb` subscripting to cope with toasted subscript values (Tom Lane, David G. Johnston)

  Using a text value fetched directly from a table as a `jsonb` subscript was likely to fail. Fetches would usually not find any matching element. Assignments could store the value with a garbage key, although keys long enough to cause that problem are probably rare in the field.

- Fix edge-case data corruption in parallel hash joins (Dmitry Astapov)

  If the final chunk of a large tuple being written out to a temporary file was exactly 32760 bytes, it would be corrupted due to a fencepost bug. The query would typically fail later with corrupted-data symptoms.

- Honor non-default settings of `checkpoint_completion_target` (Bharath Rupireddy)

  Internal state was not updated after a change in `checkpoint_completion_target`, possibly resulting in performing checkpoint I/O faster or slower than desired, especially if that setting was changed on-the-fly.

- Log the correct ending timestamp in `recovery_target_xid` mode (Tom Lane)

  When ending recovery based on the `recovery_target_xid` setting with `recovery_target_inclusive = off`, we printed an incorrect timestamp (always 2000-01-01) in the « recovery stopping before ... transaction » log message.

- Improve error reporting for some buffered file read failures (Peter Eisentraut)

  Correctly report a short read, giving the numbers of bytes desired and actually read, instead of reporting an irrelevant error code. Most places got this right already, but some recently-written replication logic did not.

- Remove arbitrary limit on number of elements in `int2vector` and `oidvector` (Tom Lane)

  The input functions for these types previously rejected more than 100 elements. With the introduction of the logical replication column list feature, it's necessary to accept `int2vector`s having up to 1600 columns, otherwise long column lists cause logical-replication failures.

- In extended query protocol, avoid an immediate commit after `ANALYZE` if we're running a pipeline (Tom Lane)

  If there's not been an explicit `BEGIN TRANSACTION`, `ANALYZE` would take it on itself to commit, which should not happen within a pipelined series of commands.

- Reject cancel request packets having the wrong length (Andrey Borodin)

  The server would process a cancel request even if its length word was too small. This led to reading beyond the end of the allocated buffer. In theory that could cause a segfault, but it seems quite unlikely to happen in practice, since the buffer would have to be very close to the end of memory.

The more likely outcome was a bogus log message about wrong backend PID or cancel code. Complain about the wrong length, instead.

- Fix planner preprocessing oversights for window function run-condition expressions (Richard Guo, David Rowley)

  This could lead to planner errors such as « WindowFunc not found in subplan target lists ».

- Fix possible dangling-pointer access during execution of window function run-condition expressions (David Rowley)

  In practice, because the run-condition optimization is only applied to certain window functions that happen to all return `int8`, this only manifested as a problem on 32-bit builds.

- Add recursion and looping defenses in subquery pullup (Tom Lane)

  A contrived query can result in deep recursion and unreasonable amounts of time spent trying to flatten subqueries. A proper fix for that seems unduly invasive for a back-patch, but we can at least add stack depth checks and an interrupt check to allow the query to be cancelled.

- Fix planner issues when combining Memoize nodes with partitionwise joins or parameterized nestloops (Richard Guo)

  These errors could lead to not using Memoize in contexts where it would be useful, or possibly to wrong query plans.

- Fix partitionwise-join code to tolerate failure to produce a plan for each partition (Tom Lane)

  This could result in « could not devise a query plan for the given query » errors.

- Limit the amount of cleanup work done by `get_actual_variable_range` (Simon Riggs)

  Planner runs occurring just after deletion of a large number of tuples appearing at the end of an index could expend significant amounts of work setting the « killed » bits for those index entries. Limit the amount of work done in any one query by giving up on this process after examining 100 heap pages. All the cleanup will still happen eventually, but without so large a performance hiccup.

- Prevent the statistics machinery from getting confused when a relation's relkind changes (Andres Freund)

  Converting a table to a view could lead to crashes or assertion failures.

- Fix under-parenthesized display of `AT TIME ZONE` constructs (Tom Lane)

  This could result in dump/restore failures for rules or views in which an argument of `AT TIME ZONE` is itself an expression.

- Prevent clobbering of cached parsetrees for utility statements in SQL functions (Tom Lane, Daniel Gustafsson)

  If a SQL-language function executes the same utility command more than once within a single calling query, it could crash or report strange errors such as « unrecognized node type ».

- Ensure that execution of full-text-search queries can be cancelled while they are performing phrase matches (Tom Lane)

- Fix memory leak in hashing strings with nondeterministic collations (Jeff Davis)

- Fix deadlock between `DROP DATABASE` and logical replication worker process (Hou Zhijie)

  This was caused by an ill-advised choice to block interrupts while creating a logical replication slot in the worker. In version 15 that could lead to an undetected deadlock. In version 14, no deadlock has been observed, but it's still a bad idea to block interrupts while waiting for network I/O.

- Clean up the libpq connection object after a failed replication connection attempt (Andres Freund)

  The previous coding leaked the connection object. In background code paths that's pretty harmless because the calling process will give up and exit. But in commands such as `CREATE SUBSCRIPTION`, such a failure resulted in a small session-lifespan memory leak.

- In hot-standby servers, reduce processing effort for tracking XIDs known to be active on the primary (Simon Riggs, Michail Nikolaev)

  Insufficiently-aggressive cleanup of the KnownAssignedXids array could lead to poor performance, particularly when `max_connections` is set to a large value on the standby.

- Ignore invalidated logical-replication slots while determining oldest catalog xmin (Sirisha Chamarthi)

  A replication slot could prevent cleanup of dead tuples in the system catalogs even after it becomes invalidated due to exceeding `max_slot_wal_keep_size`. Thus, failure of a replication consumer could lead to indefinitely-large catalog bloat.

- In logical decoding, notify the remote node when a transaction is detected to have crashed (Hou Zhijie)

  After a server restart, we'll re-stream the changes for transactions occurring shortly before the restart. Some of these transactions probably never completed; when we realize that one didn't we throw away the relevant decoding state locally, but we neglected to tell the subscriber about it. That led to the subscriber keeping useless streaming files until it's next restarted.

- Fix uninitialized-memory usage in logical decoding (Masahiko Sawada)

  In certain cases, resumption of logical decoding could try to re-use XID data that had already been freed, leading to unpredictable behavior.

- Acquire spinlock while updating shared state during logical decoding context creation (Masahiko Sawada)

  We neglected to acquire the appropriate lock while updating data about two-phase transactions, potentially allowing other processes to see inconsistent data.

- Fix pgoutput replication plug-in to not send columns not listed in a table's replication column list (Hou Zhijie)

  `UPDATE` and `DELETE` events did not pay attention to the configured column list, thus sending more data than expected. This did not cause a problem when the receiver is our built-in logical replication code, but it might confuse other receivers, and in any case it wasted network bandwidth.

- Avoid rare « failed to acquire cleanup lock » panic during WAL replay of hash-index page split operations (Robert Haas)

- Advance a heap page's LSN when setting its all-visible bit during WAL replay (Jeff Davis)

  Failure to do this left the page possibly different on standby servers than the primary, and violated some other expectations about when the LSN changes. This seems only a theoretical hazard so far as PostgreSQL itself is concerned, but it could upset third-party tools.

- Fix `int64_div_fast_to_numeric()` to work for a wider range of inputs (Dean Rasheed)

  This function misbehaved with some values of its second argument. No such usages exist in core PostgreSQL, but it's clearly a hazard for external modules, so repair.

- Fix latent buffer-overrun problem in `WaitEventSet` logic (Thomas Munro)

The `epoll`-based and `kqueue`-based implementations could ask the kernel for too many events if the size of their internal buffer was different from the size of the caller's output buffer. That case is not known to occur in released PostgreSQL versions, but this error is a hazard for external modules and future bug fixes.

- Avoid nominally-undefined behavior when accessing shared memory in 32-bit builds (Andres Freund)

  clang's undefined-behavior sanitizer complained about use of a pointer that was less aligned than it should be. It's very unlikely that this would cause a problem in non-debug builds, but it's worth fixing for testing purposes.

- Fix assertion failure in BRIN minmax-multi opclasses (Tomas Vondra)

  The assertion was overly strict, so this mistake was harmless in non-assert builds.

- Remove faulty assertion in useless-RESULT-RTE optimization logic (Tom Lane)

- Fix copy-and-paste errors in cache-lookup-failure messages for ACL checks (Justin Pryzby)

  In principle these errors should never be reached. But if they are, some of them reported the wrong type of object.

- Fix possible corruption of very large tablespace map files in pg_basebackup (Antonin Houska)

- Avoid harmless warning from pg_dump in `--if-exists` mode (Tom Lane)

  If the `public` schema has a non-default owner then use of pg_dump's `--if-exists` option resulted in a warning message « warning: could not find where to insert IF EXISTS in statement "--*not* dropping schema, since initdb creates it" ». The dump output was okay, though.

- Fix psql's `\sf` and `\ef` commands to handle SQL-language functions that have SQL-standard function bodies (Tom Lane)

  These commands misidentified the start of the function body when it used new-style syntax.

- Fix tab completion of `ALTER FUNCTION/PROCEDURE/ROUTINE ... SET SCHEMA` (Dean Rasheed)

- Update `contrib/pageinspect` to mark its disk-accessing functions as `PARALLEL RESTRICTED` (Tom Lane)

  This avoids possible failure if one of these functions is used to examine a temporary table, since a session's temporary tables are not accessible from parallel workers.

- Fix `contrib/seg` to not crash or print garbage if an input number has more than 127 digits (Tom Lane)

- Fix build on Microsoft Visual Studio 2013 (Tom Lane)

  A previous patch supposed that all platforms of interest have `snprintf()`, but MSVC 2013 isn't quite there yet. Revert to using `sprintf()` on that platform.

- Fix compile failure in building PL/Perl with MSVC when using Strawberry Perl (Andrew Dunstan)

- Fix mismatch of PL/Perl built with MSVC versus a Perl library built with gcc (Andrew Dunstan)

  Such combinations could previously fail with « loadable library and perl binaries are mismatched » errors.

- Suppress compiler warnings from Perl's header files (Andres Freund)

Our preferred compiler options provoke warnings about constructs appearing in recent versions of Perl's header files. When using gcc, we can suppress these warnings with a pragma.

- Fix pg_waldump to build on compilers that don't discard unused static-inline functions (Tom Lane)
- Update time zone data files to tzdata release 2022g for DST law changes in Greenland and Mexico, plus historical corrections for northern Canada, Colombia, and Singapore. (Tom Lane)

  Notably, a new timezone America/Ciudad_Juarez has been split off from America/Ojinaga.

E.19. Release 15.1

Release date: 2022-11-10

This release contains a variety of fixes from 15.0. For information about new features in major release 15, see Section E.20.

E.19.1. Migration to Version 15.1

A dump/restore is not required for those running 15.X.

However, if you regularly create and drop tables exceeding 1GB, see the first changelog entry below.

E.19.2. Changes

- Fix failure to remove non-first segments of large tables (Tom Lane)

  PostgreSQL splits large tables into multiple files (normally with 1GB per file). The logic for dropping a table was broken and would miss removing all but the first such file, in two cases: drops of temporary tables and WAL replay of drops of regular tables. Applications that routinely create multi-gigabyte temporary tables could suffer significant disk space leakage.

  Orphaned temporary-table files are removed during postmaster start, so the mere act of updating to 15.1 is sufficient to clear any leaked temporary-table storage. However, if you suffered any database crashes while using 15.0, and there might have been large tables dropped just before such crashes, it's advisable to check the database directories for files named according to the pattern *`NNNN.NN`*. If there is no matching file named just *`NNNN`* (without the *`.NN`* suffix), these files should be removed manually.

- Fix handling of `DEFAULT` tokens that appear in a multi-row `VALUES` clause of an `INSERT` on an updatable view (Tom Lane)

  This oversight could lead to « cache lookup failed for type » errors, or in older branches even to crashes.

- Disallow rules named `_RETURN` that are not `ON SELECT` (Tom Lane)

  This avoids confusion between a view's `ON SELECT` rule and any other rules it may have.

- Avoid failure in `EXPLAIN VERBOSE` for a query using `SEARCH BREADTH FIRST` with constant initial values (Tom Lane)

- Prevent use of `MERGE` on a partitioned table with foreign-table partitions (Álvaro Herrera)

  The case isn't supported, and previously threw an incomprehensible error.

- Fix construction of per-partition foreign key constraints while doing `ALTER TABLE ATTACH PARTITION` (Jehan-Guillaume de Rorthais, Álvaro Herrera)

Previously, incorrect or duplicate constraints could be constructed for the newly-added partition.

- Fix planner failure with extended statistics on partitioned or inherited tables (Richard Guo, Justin Pryzby)

  Some cases failed with « cache lookup failed for statistics object ».

- Fix mis-ordering of WAL operations in fast insert path for GIN indexes (Matthias van de Meent, Zhang Mingli)

  This mistake is not known to have any negative consequences within core PostgreSQL, but it did cause issues for some extensions.

- Fix bugs in logical decoding when replay starts from a point between the beginning of a transaction and the beginning of its subtransaction (Masahiko Sawada, Kuroda Hayato)

  These errors could lead to assertion failures in debug builds, and otherwise to memory leaks.

- Accept interrupts in more places during logical decoding (Amit Kapila, Masahiko Sawada)

  This ameliorates problems with slow shutdown of replication workers.

- Prevent attempts to replicate into a foreign-table partition in replication workers (Shi Yu, Tom Lane)

  Although partitioned tables can have foreign tables as partitions, replicating into such a partition isn't currently supported. The logical replication worker process would crash if it was attempted. Now, an error is thrown.

- Avoid crash after function syntax error in replication workers (Maxim Orlov, Anton Melnikov, Masahiko Sawada, Tom Lane)

  If a syntax error occurred in a SQL-language or PL/pgSQL-language `CREATE FUNCTION` or `DO` command executed in a logical replication worker, the worker process would crash with a null pointer dereference or assertion failure.

- Avoid double call of the shutdown callback of an archiver module (Nathan Bossart, Bharath Rupireddy)

- Add plan-time check for attempted access to a table that has no table access method (Tom Lane)

  This prevents a crash in some catalog-corruption scenarios, for example use of a view whose `ON SELECT` rule is missing.

- Prevent postmaster crash when shared-memory state is corrupted (Tom Lane)

  The postmaster process is supposed to survive and initiate a database restart if shared memory becomes corrupted, but one bit of code was being insufficiently cautious about that.

- In libpq, handle single-row mode correctly when pipelining (Denis Laxalde)

  The single-row flag was not reset at the correct time if pipeline mode was also active.

- Fix psql's exit status when a command-line query is canceled (Peter Eisentraut)

  `psql -c` *`query`* would exit successfully if the query was canceled. Fix it to exit with nonzero status, as in other error cases.

- Allow cross-platform tablespace relocation in pg_basebackup (Robert Haas)

  Allow the remote path in `--tablespace-mapping` to be either a Unix-style or Windows-style absolute path, since the source server could be on a different OS than the local system.

- Fix pg_dump's failure to dump comments attached to some `CHECK` constraints (Tom Lane)

- Fix `CREATE DATABASE` to allow its `oid` parameter to exceed 2^{31} (Tom Lane)

  This oversight prevented pg_upgrade from succeeding when the source installation contained databases with OIDs larger than that.

- In pg_stat_statements, fix access to already-freed memory (zhaoqigui)

  This occurred if pg_stat_statements tracked a `ROLLBACK` command issued via extended query protocol. In debug builds it consistently led to an assertion failure. In production builds there would often be no visible ill effect; but if the freed memory had already been reused, the likely result would be to store garbage for the query string.

- Fix incompatibilities with LLVM 15 (Thomas Munro, Andres Freund)

- Allow use of `__sync_lock_test_and_set()` for spinlocks on any machine (Tom Lane)

  This eases porting to new machine architectures, at least if you're using a compiler that supports this GCC builtin function.

- Rename symbol `REF` to `REF_P` to avoid compile failure on recent macOS (Tom Lane)

- Avoid using `sprintf`, to avoid compile-time deprecation warnings (Tom Lane)

- Update time zone data files to tzdata release 2022f for DST law changes in Chile, Fiji, Iran, Jordan, Mexico, Palestine, and Syria, plus historical corrections for Chile, Crimea, Iran, and Mexico. (Tom Lane)

  Also, the Europe/Kiev zone has been renamed to Europe/Kyiv. Also, the following zones have been merged into nearby, more-populous zones whose clocks have agreed with them since 1970: Antarctica/Vostok, Asia/Brunei, Asia/Kuala_Lumpur, Atlantic/Reykjavik, Europe/Amsterdam, Europe/Copenhagen, Europe/Luxembourg, Europe/Monaco, Europe/Oslo, Europe/Stockholm, Indian/Christmas, Indian/Cocos, Indian/Kerguelen, Indian/Mahe, Indian/Reunion, Pacific/Chuuk, Pacific/Funafuti, Pacific/Majuro, Pacific/Pohnpei, Pacific/Wake and Pacific/Wallis. (This indirectly affects zones that were already links to one of these: Arctic/Longyearbyen, Atlantic/Jan_Mayen, Iceland, Pacific/Ponape, Pacific/Truk, and Pacific/Yap.) America/Nipigon, America/Rainy_River, America/Thunder_Bay, Europe/Uzhgorod, and Europe/Zaporozhye were also merged into nearby zones after discovering that their claimed post-1970 differences from those zones seem to have been errors. In all these cases, the previous zone name remains as an alias; but the actual data is that of the zone that was merged into.

  These zone mergers result in loss of pre-1970 timezone history for the merged zones, which may be troublesome for applications expecting consistency of `timestamptz` display. As an example, the stored value `1944-06-01 12:00 UTC` would previously display as `1944-06-01 13:00:00+01` if the Europe/Stockholm zone is selected, but now it will read out as `1944-06-01 14:00:00+02`.

  It is possible to build the time zone data files with options that will restore the older zone data, but that choice also inserts a lot of other old (and typically poorly-attested) zone data, resulting in more total changes from the previous release than accepting these upstream changes does. PostgreSQL has chosen to ship the tzdb data as-recommended, and so far as we are aware most major operating system distributions are doing likewise. However, if these changes cause significant problems for your application, a possible solution is to install a local build of the time zone data files using tzdb's backwards-compatibility options (see their `PACKRATDATA` and `PACKRATLIST` options).

E.20. Release 15

Release date: 2022-10-13

E.20.1. Overview

PostgreSQL 15 contains many new features and enhancements, including:

- Support for the SQL `MERGE` command.
- Selective publication of tables' contents within logical replication publications, through the ability to specify column lists and row filter conditions.
- More options for compression, including support for Zstandard (zstd) compression. This includes support for performing compression on the server side during pg_basebackup.
- Support for structured server log output using the JSON format.
- Performance improvements, particularly for in-memory and on-disk sorting.

The above items and other new features of PostgreSQL 15 are explained in more detail in the sections below.

E.20.2. Migration to Version 15

A dump/restore using pg_dumpall or use of pg_upgrade or logical replication is required for those wishing to migrate data from any previous release. See Section 19.6 for general information on migrating to new major releases.

Version 15 contains a number of changes that may affect compatibility with previous releases. Observe the following incompatibilities:

- Remove `PUBLIC` creation permission on the `public` schema (Noah Misch)

  The new default is one of the secure schema usage patterns that Section 5.9.6 has recommended since the security release for CVE-2018-1058. The change applies to new database clusters and to newly-created databases in existing clusters. Upgrading a cluster or restoring a database dump will preserve `public`'s existing permissions.

  For existing databases, especially those having multiple users, consider revoking `CREATE` permission on the `public` schema to adopt this new default. For new databases having no need to defend against insider threats, granting `CREATE` permission will yield the behavior of prior releases.

- Change the owner of the `public` schema to be the new `pg_database_owner` role (Noah Misch)

  This allows each database's owner to have ownership privileges on the `public` schema within their database. Previously it was owned by the bootstrap superuser, so that non-superuser database owners could not do anything with it.

  This change applies to new database clusters and to newly-created databases in existing clusters. Upgrading a cluster or restoring a database dump will preserve `public`'s existing ownership specification.

- Remove long-deprecated exclusive backup mode (David Steele, Nathan Bossart)

  If the database server stops abruptly while in this mode, the server could fail to start. The non-exclusive backup mode is considered superior for all purposes. Functions `pg_start_backup()`/`pg_stop_backup()` have been renamed to `pg_backup_start()`/`pg_backup_stop()`, and the functions `pg_backup_start_time()` and `pg_is_in_backup()` have been removed.

- Increase `hash_mem_multiplier` default to 2.0 (Peter Geoghegan)

This allows query hash operations to use more `work_mem` memory than other operations.

- Remove server-side language `plpython2u` and generic Python language `plpythonu` (Andres Freund)

  Python 2.x is no longer supported. While the original intent of `plpythonu` was that it could eventually refer to `plpython3u`, changing it now seems more likely to cause problems than solve them, so it's just been removed.

- Generate an error if `array_to_tsvector()` is passed an empty-string array element (Jean-Christophe Arnu)

  This is prohibited because lexemes should never be empty. Users of previous Postgres releases should verify that no empty lexemes are stored because they can lead to dump/restore failures and inconsistent results.

- Generate an error when `chr()` is supplied with a negative argument (Peter Eisentraut)

- Prevent `CREATE OR REPLACE VIEW` from changing the collation of an output column (Tom Lane)

- Disallow zero-length Unicode identifiers, e.g., `U&""` (Peter Eisentraut)

  Non-Unicode zero-length identifiers were already disallowed.

- Prevent numeric literals from having non-numeric trailing characters (Peter Eisentraut)

  Previously, query text like `123abc` would be interpreted as `123` followed by a separate token `abc`.

- Adjust JSON numeric literal processing to match the SQL/JSON-standard (Peter Eisentraut)

  This accepts numeric formats like `.1` and `1.`, and disallows trailing junk after numeric literals, like `1.type()`.

- When `interval` input provides a fractional value for a unit greater than months, round to the nearest month (Bruce Momjian)

  For example, convert `1.99 years` to `2 years`, not `1 year 11 months` as before.

- Improve consistency of `interval` parsing with trailing periods (Tom Lane)

  Numbers with trailing periods were rejected on some platforms.

- Mark the `interval` output function as stable, not immutable, since it depends on `IntervalStyle` (Tom Lane)

  This will, for example, cause creation of indexes relying on the text output of `interval` values to fail.

- Detect integer overflow in interval justification functions (Joe Koshakow)

  The affected functions are `justify_interval()`, `justify_hours()`, and `justify_days()`.

- Change the I/O format of type `"char"` for non-ASCII characters (Tom Lane)

  Bytes with the high bit set are now output as a backslash and three octal digits, to avoid encoding issues.

- Remove the default `ADMIN OPTION` privilege a login role has on its own role membership (Robert Haas)

Previously, a login role could add/remove members of its own role, even without `ADMIN OPTION` privilege.

- Allow logical replication to run as the owner of the subscription (Mark Dilger)

  Because row-level security policies are not checked, only superusers, roles with `bypassrls`, and table owners can replicate into tables with row-level security policies.

- Prevent `UPDATE` and `DELETE` logical replication operations on tables where the subscription owner does not have `SELECT` permission on the table (Jeff Davis)

  `UPDATE` and `DELETE` commands typically involve reading the table as well, so require the subscription owner to have table `SELECT` permission.

- When `EXPLAIN` references the session's temporary object schema, refer to it as `pg_temp` (Amul Sul)

  Previously the actual schema name was reported, leading to inconsistencies across sessions.

- Fix `pg_statio_all_tables` to sum values for the rare case of TOAST tables with multiple indexes (Andrei Zubkov)

  Previously such cases would show one row for each index.

- Disallow setting custom options that match the name of an installed extension, but are not one of the extension's declared variables (Florin Irion, Tom Lane)

  This change causes any such pre-existing variables to be deleted during extension load, and then prevents new ones from being created later in the session. The intent is to prevent confusion about whether a variable is associated with an extension or not.

- Remove obsolete server variable `stats_temp_directory` (Andres Freund, Kyotaro Horiguchi)

- Improve the algorithm used to compute `random()` (Fabien Coelho)

  This will cause `random()`'s results to differ from what was emitted by prior versions, even for the same seed value.

- libpq's `PQsendQuery()` function is no longer supported in pipeline mode (Álvaro Herrera)

  Applications that are using that combination will need to be modified to use `PQsendQueryParams()` instead.

- On non-Windows platforms, consult the `HOME` environment variable to find the user's home directory (Anders Kaseorg)

  If `HOME` is empty or unset, fall back to the previous method of checking the `<pwd.h>` database. This change affects libpq (for example, while looking up `~/.pgpass`) as well as various client application programs.

- Remove pg_dump's `--no-synchronized-snapshots` option (Tom Lane)

  All still-supported server versions support synchronized snapshots, so there's no longer a need for this option.

- After an error is detected in psql's `--single-transaction` mode, change the final `COMMIT` command to `ROLLBACK` only if `ON_ERROR_STOP` is set (Michael Paquier)

- Avoid unnecessary casting of constants in queries sent by postgres_fdw (Dian Fay)

When column types are intentionally different between local and remote databases, such casts could cause errors.

- Remove xml2's `xml_is_well_formed()` function (Tom Lane)

  This function has been implemented in the core backend since Postgres 9.1.

- Allow custom scan providers to indicate if they support projections (Sven Klemm)

  The default is now that custom scan providers are assumed to not support projections; those that do will need to be updated for this release.

E.20.3. Changes

Below you will find a detailed account of the changes between PostgreSQL 15 and the previous major release.

E.20.3.1. Server

- Record and check the collation version of each database (Peter Eisentraut)

  This feature is designed to detect collation version changes to avoid index corruption. Function `pg_database_collation_actual_version()` reports the underlying operating system collation version, and `ALTER DATABASE ... REFRESH` sets the recorded database collation version to match the operating system collation version.

- Allow ICU collations to be set as the default for clusters and databases (Peter Eisentraut)

  Previously, only libc-based collations could be selected at the cluster and database levels. ICU collations could only be used via explicit `COLLATE` clauses.

- Add system view `pg_ident_file_mappings` to report `pg_ident.conf` information (Julien Rouhaud)

E.20.3.1.1. Partitioning

- Improve planning time for queries referencing partitioned tables (David Rowley)

  This change helps when only a few of many partitions are relevant.

- Allow ordered scans of partitions to avoid sorting in more cases (David Rowley)

  Previously, a partitioned table with a `DEFAULT` partition or a `LIST` partition containing multiple values could not be used for ordered partition scans. Now they can be used if such partitions are pruned during planning.

- Improve foreign key behavior of updates on partitioned tables that move rows between partitions (Amit Langote)

  Previously, such updates ran a delete action on the source partition and an insert action on the target partition. PostgreSQL will now run an update action on the partition root, providing cleaner semantics.

- Allow `CLUSTER` on partitioned tables (Justin Pryzby)

- Fix `ALTER TRIGGER RENAME` on partitioned tables to properly rename triggers on all partitions (Arne Roland, Álvaro Herrera)

  Also prohibit cloned triggers from being renamed.

E.20.3.1.2. Indexes

- Allow btree indexes on system and TOAST tables to efficiently store duplicates (Peter Geoghegan)

  Previously de-duplication was disabled for these types of indexes.

- Improve lookup performance of GiST indexes that were built using sorting (Aliaksandr Kalenik, Sergei Shoulbakov, Andrey Borodin)

- Allow unique constraints and indexes to treat `NULL` values as not distinct (Peter Eisentraut)

  Previously `NULL` entries were always treated as distinct values, but this can now be changed by creating constraints and indexes using `UNIQUE NULLS NOT DISTINCT`.

- Allow the `^@` starts-with operator and the `starts_with()` function to use btree indexes if using the C collation (Tom Lane)

  Previously these could only use SP-GiST indexes.

E.20.3.1.3. Optimizer

- Allow extended statistics to record statistics for a parent with all its children (Tomas Vondra, Justin Pryzby)

  Regular statistics already tracked parent and parent-plus-all-children statistics separately.

- Add server variable `recursive_worktable_factor` to allow the user to specify the expected size of the working table of a recursive query (Simon Riggs)

E.20.3.1.4. General Performance

- Allow hash lookup for `NOT IN` clauses with many constants (David Rowley, James Coleman)

  Previously the code always sequentially scanned the list of values.

- Allow `SELECT DISTINCT` to be parallelized (David Rowley)

- Speed up encoding validation of UTF-8 text by processing 16 bytes at a time (John Naylor, Heikki Linnakangas)

  This will improve text-heavy operations like `COPY FROM`.

- Improve performance for sorts that exceed `work_mem` (Heikki Linnakangas)

  When the sort data no longer fits in `work_mem`, switch to a batch sorting algorithm that uses more output streams than before.

- Improve performance and reduce memory consumption of in-memory sorts (Ronan Dunklau, David Rowley, Thomas Munro, John Naylor)

- Allow WAL full page writes to use LZ4 and Zstandard compression (Andrey Borodin, Justin Pryzby)

  This is controlled by the `wal_compression` server setting.

- Add support for writing WAL using direct I/O on macOS (Thomas Munro)

  This only works if `max_wal_senders = 0` and `wal_level = minimal`.

- Allow vacuum to be more aggressive in setting the oldest frozen and multi transaction id (Peter Geoghegan)

- Allow a query referencing multiple foreign tables to perform parallel foreign table scans in more cases (Andrey Lepikhov, Etsuro Fujita)

- Improve the performance of window functions that use `row_number()`, `rank()`, `dense_rank()` and `count()` (David Rowley)

- Improve the performance of spinlocks on high-core-count ARM64 systems (Geoffrey Blake)

E.20.3.1.5. Monitoring

- Enable default logging of checkpoints and slow autovacuum operations (Bharath Rupireddy)

  This changes the default of `log_checkpoints` to `on` and that of `log_autovacuum_min_duration` to 10 minutes. This will cause even an idle server to generate some log output, which might cause problems on resource-constrained servers without log file rotation. These defaults should be changed in such cases.

- Generate progress messages in the server log during slow server starts (Nitin Jadhav, Robert Haas)

  The messages report the cause of the delay. The time interval for notification is controlled by the new server variable `log_startup_progress_interval`.

- Store cumulative statistics system data in shared memory (Kyotaro Horiguchi, Andres Freund, Melanie Plageman)

  Previously this data was sent to a statistics collector process via UDP packets, and could only be read by sessions after transferring it via the file system. There is no longer a separate statistics collector process.

- Add additional information to `VACUUM VERBOSE` and autovacuum logging messages (Peter Geoghegan)

- Add `EXPLAIN (BUFFERS)` output for temporary file block I/O (Masahiko Sawada)

- Allow log output in JSON format (Sehrope Sarkuni, Michael Paquier)

  The new setting is `log_destination = jsonlog`.

- Allow `pg_stat_reset_single_table_counters()` to reset the counters of relations shared across all databases (Sadhuprasad Patro)

- Add wait events for local shell commands (Fujii Masao)

  The new wait events are used when calling `archive_command`, `archive_cleanup_command`, `restore_command` and `recovery_end_command`.

E.20.3.1.6. Privileges

- Allow table accesses done by a view to optionally be controlled by privileges of the view's caller (Christoph Heiss)

  Previously, view accesses were always treated as being done by the view's owner. That's still the default.

- Allow members of the `pg_write_server_files` predefined role to perform server-side base backups (Dagfinn Ilmari Mannsåker)

  Previously only superusers could perform such backups.

- Allow `GRANT` to grant permissions to change individual server variables via `SET` and `ALTER SYSTEM` (Mark Dilger)

The new function `has_parameter_privilege()` reports on this privilege.

- Add predefined role `pg_checkpoint` that allows members to run `CHECKPOINT` (Jeff Davis)

  Previously checkpoints could only be run by superusers.

- Allow members of the `pg_read_all_stats` predefined role to access the views `pg_backend_memory_contexts` and `pg_shmem_allocations` (Bharath Rupireddy)

  Previously these views could only be accessed by superusers.

- Allow `GRANT` to grant permissions on `pg_log_backend_memory_contexts()` (Jeff Davis)

  Previously this function could only be run by superusers.

E.20.3.1.7. Server Configuration

- Add server variable `shared_memory_size` to report the size of allocated shared memory (Nathan Bossart)

- Add server variable `shared_memory_size_in_huge_pages` to report the number of huge memory pages required (Nathan Bossart)

  This is only supported on Linux.

- Honor server variable `shared_preload_libraries` in single-user mode (Jeff Davis)

  This change supports use of `shared_preload_libraries` to load custom access methods and WAL resource managers, which would be essential for database access even in single-user mode.

- On Solaris, make the default setting of `dynamic_shared_memory_type` be `sysv` (Thomas Munro)

  The previous default choice, `posix`, can result in spurious failures on this platform.

- Allow `postgres -C` to properly report runtime-computed values (Nathan Bossart)

  Previously runtime-computed values `data_checksums`, `wal_segment_size`, and `data_directory_mode` would report values that would not be accurate on the running server. However, this does not work on a running server.

E.20.3.2. Streaming Replication and Recovery

- Add support for LZ4 and Zstandard compression of server-side base backups (Jeevan Ladhe, Robert Haas)

- Run the checkpointer and bgwriter processes during crash recovery (Thomas Munro)

  This helps to speed up long crash recoveries.

- Allow WAL processing to pre-fetch needed file contents (Thomas Munro)

  This is controlled by the server variable `recovery_prefetch`.

- Allow archiving via loadable modules (Nathan Bossart)

  Previously, archiving was only done by calling shell commands. The new server variable `archive_library` can be set to specify a library to be called for archiving.

- No longer require `IDENTIFY_SYSTEM` to be run before `START_REPLICATION` (Jeff Davis)

E.20.3.2.1. Logical Replication

- Allow publication of all tables in a schema (Vignesh C, Hou Zhijie, Amit Kapila)

  For example, this syntax is now supported: `CREATE PUBLICATION pub1 FOR TABLES IN SCHEMA s1,s2`. `ALTER PUBLICATION` supports a similar syntax. Tables added later to the listed schemas will also be replicated.

- Allow publication content to be filtered using a `WHERE` clause (Hou Zhijie, Euler Taveira, Peter Smith, Ajin Cherian, Tomas Vondra, Amit Kapila)

  Rows not satisfying the `WHERE` clause are not published.

- Allow publication content to be restricted to specific columns (Tomas Vondra, Álvaro Herrera, Rahila Syed)

- Allow skipping of transactions on a subscriber using `ALTER SUBSCRIPTION ... SKIP` (Masahiko Sawada)

- Add support for prepared (two-phase) transactions to logical replication (Peter Smith, Ajin Cherian, Amit Kapila, Nikhil Sontakke, Stas Kelvich)

  The new `CREATE_REPLICATION_SLOT` option is called `TWO_PHASE`. pg_recvlogical now supports a new `--two-phase` option during slot creation.

- Prevent logical replication of empty transactions (Ajin Cherian, Hou Zhijie, Euler Taveira)

  Previously, publishers would send empty transactions to subscribers if subscribed tables were not modified.

- Add SQL functions to monitor the directory contents of logical replication slots (Bharath Rupireddy)

  The new functions are `pg_ls_logicalsnapdir()`, `pg_ls_logicalmapdir()`, and `pg_ls_replslotdir()`. They can be run by members of the predefined `pg_monitor` role.

- Allow subscribers to stop the application of logical replication changes on error (Osumi Takamichi, Mark Dilger)

  This is enabled with the subscriber option `disable_on_error` and avoids possible infinite error loops during stream application.

- Adjust subscriber server variables to match the publisher so datetime and float8 values are interpreted consistently (Japin Li)

  Some publishers might be relying on inconsistent behavior.

- Add system view `pg_stat_subscription_stats` to report on subscriber activity (Masahiko Sawada)

  The new function `pg_stat_reset_subscription_stats()` allows resetting these statistics counters.

- Suppress duplicate entries in the `pg_publication_tables` system view (Hou Zhijie)

  In some cases a partition could appear more than once.

E.20.3.3. Utility Commands

- Add SQL `MERGE` command to adjust one table to match another (Simon Riggs, Pavan Deolasee, Álvaro Herrera, Amit Langote)

This is similar to `INSERT ... ON CONFLICT` but more batch-oriented.

- Add support for `HEADER` option in `COPY` text format (Rémi Lapeyre)

  The new option causes the column names to be output, and optionally verified on input.

- Add new WAL-logged method for database creation (Dilip Kumar)

  This is the new default method for copying the template database, as it avoids the need for checkpoints during database creation. However, it might be slow if the template database is large, so the old method is still available.

- Allow `CREATE DATABASE` to set the database OID (Shruthi Gowda, Antonin Houska)

- Prevent `DROP DATABASE`, `DROP TABLESPACE`, and `ALTER DATABASE SET TABLESPACE` from occasionally failing during concurrent use on Windows (Thomas Munro)

- Allow foreign key `ON DELETE SET` actions to affect only specified columns (Paul Martinez)

  Previously, all of the columns in the foreign key were always affected.

- Allow `ALTER TABLE` to modify a table's `ACCESS METHOD` (Justin Pryzby, Jeff Davis)

- Properly call object access hooks when `ALTER TABLE` causes table rewrites (Michael Paquier)

- Allow creation of unlogged sequences (Peter Eisentraut)

- Track dependencies on individual columns in the results of functions returning composite types (Tom Lane)

  Previously, if a view or rule contained a reference to a specific column within the result of a composite-returning function, that was not noted as a dependency; the view or rule was only considered to depend on the composite type as a whole. This meant that dropping the individual column would be allowed, causing problems in later use of the view or rule. The column-level dependency is now also noted, so that dropping such a column will be rejected unless the view is changed or dropped.

E.20.3.4. Data Types

- Allow the scale of a `numeric` value to be negative, or greater than its precision (Dean Rasheed, Tom Lane)

  This allows rounding of values to the left of the decimal point, e.g., `'1234'::numeric(4, -2)` returns 1200.

- Improve overflow detection when casting values to interval (Joe Koshakow)

- Change the I/O format of type `"char"` for non-ASCII characters (Tom Lane)

- Update the display width information of modern Unicode characters, like emojis (Jacob Champion)

  Also update from Unicode 5.0 to 14.0.0. There is now an automated way to keep Postgres updated with Unicode releases.

E.20.3.5. Functions

- Add multirange input to `range_agg()` (Paul Jungwirth)

- Add `MIN()` and `MAX()` aggregates for the `xid8` data type (Ken Kato)

- Add regular expression functions for compatibility with other relational systems (Gilles Darold, Tom Lane)

  The new functions are `regexp_count()`, `regexp_instr()`, `regexp_like()`, and `regexp_substr()`. Some new optional arguments were also added to `regexp_replace()`.

- Add the ability to compute the distance between `polygons` (Tom Lane)

- Add `to_char()` format codes `of`, `tzh`, and `tzm` (Nitin Jadhav)

  The upper-case equivalents of these were already supported.

- When applying `AT TIME ZONE` to a `time with time zone` value, use the transaction start time rather than wall clock time to determine whether DST applies (Aleksander Alekseev, Tom Lane)

  This allows the conversion to be considered stable rather than volatile, and it saves a kernel call per invocation.

- Ignore NULL array elements in `ts_delete()` and `setweight()` functions with array arguments (Jean-Christophe Arnu)

  These functions effectively ignore empty-string array elements (since those could never match a valid lexeme). It seems consistent to let them ignore NULL elements too, instead of failing.

- Add support for petabyte units to `pg_size_pretty()` and `pg_size_bytes()` (David Christensen)

- Change `pg_event_trigger_ddl_commands()` to output references to other sessions' temporary schemas using the actual schema name (Tom Lane)

  Previously this function reported all temporary schemas as `pg_temp`, but it's misleading to use that for any but the current session's temporary schema.

E.20.3.6. PL/pgSQL

- Fix enforcement of PL/pgSQL variable `CONSTANT` markings (Tom Lane)

  Previously, a variable could be used as a `CALL` output parameter or refcursor `OPEN` variable despite being marked `CONSTANT`.

E.20.3.7. libpq

- Allow IP address matching against a server certificate's Subject Alternative Name (Jacob Champion)

- Allow `PQsslAttribute()` to report the SSL library type without requiring a libpq connection (Jacob Champion)

- Change query cancellations sent by the client to use the same TCP settings as normal client connections (Jelte Fennema)

  This allows configured TCP timeouts to apply to query cancel connections.

- Prevent libpq event callback failures from forcing an error result (Tom Lane)

E.20.3.8. Client Applications

- Allow pgbench to retry after serialization and deadlock failures (Yugo Nagata, Marina Polyakova)

E.20.3.8.1. psql

- Improve performance of psql's `\copy` command, by sending data in larger chunks (Heikki Linnakangas)
- Add `\dconfig` command to report server variables (Mark Dilger, Tom Lane)

  This is similar to the server-side `SHOW` command, but it can process patterns to show multiple variables conveniently.
- Add `\getenv` command to assign the value of an environment variable to a psql variable (Tom Lane)
- Add `+` option to the `\lo_list` and `\dl` commands to show large-object privileges (Pavel Luzanov)
- Add a pager option for the `\watch` command (Pavel Stehule, Thomas Munro)

  This is only supported on Unix and is controlled by the `PSQL_WATCH_PAGER` environment variable.
- Make psql include intra-query double-hyphen comments in queries sent to the server (Tom Lane, Greg Nancarrow)

  Previously such comments were removed from the query before being sent. Double-hyphen comments that are before any query text are not sent, and are not recorded as separate psql history entries.
- Adjust psql so that Readline's meta-# command will insert a double-hyphen comment marker (Tom Lane)

  Previously a pound marker was inserted, unless the user had taken the trouble to configure a non-default comment marker.
- Make psql output all results when multiple queries are passed to the server at once (Fabien Coelho)

  Previously, only the last query result was displayed. The old behavior can be restored by setting the `SHOW_ALL_RESULTS` psql variable to `off`.
- After an error is detected in `--single-transaction` mode, change the final `COMMIT` command to `ROLLBACK` only if `ON_ERROR_STOP` is set (Michael Paquier)

  Previously, detection of an error in a `-c` command or `-f` script file would lead to issuing `ROLLBACK` at the end, regardless of the value of `ON_ERROR_STOP`.
- Improve psql's tab completion (Shinya Kato, Dagfinn Ilmari Mannsåker, Peter Smith, Koyu Tanigawa, Ken Kato, David Fetter, Haiying Tang, Peter Eisentraut, Álvaro Herrera, Tom Lane, Masahiko Sawada)
- Limit support of psql's backslash commands to servers running PostgreSQL 9.2 or later (Tom Lane)

  Remove code that was only used when running with an older server. Commands that do not require any version-specific adjustments compared to 9.2 will still work.

E.20.3.8.2. pg_dump

- Make pg_dump dump `public` schema ownership changes and security labels (Noah Misch)
- Improve performance of dumping databases with many objects (Tom Lane)

  This will also improve the performance of pg_upgrade.

- Improve parallel pg_dump's performance for tables with large TOAST tables (Tom Lane)
- Add dump/restore option `--no-table-access-method` to force restore to only use the default table access method (Justin Pryzby)
- Limit support of pg_dump and pg_dumpall to servers running PostgreSQL 9.2 or later (Tom Lane)

E.20.3.9. Server Applications

- Add new pg_basebackup option `--target` to control the base backup location (Robert Haas)

  The new options are `server` to write the backup locally and `blackhole` to discard the backup (for testing).

- Allow pg_basebackup to do server-side gzip, LZ4, and Zstandard compression and client-side LZ4 and Zstandard compression of base backup files (Dipesh Pandit, Jeevan Ladhe)

  Client-side `gzip` compression was already supported.

- Allow pg_basebackup to compress on the server side and decompress on the client side before storage (Dipesh Pandit)

  This is accomplished by specifying compression on the server side and plain output format.

- Allow pg_basebackup's `--compress` option to control the compression location (server or client), compression method, and compression options (Michael Paquier, Robert Haas)
- Add the LZ4 compression method to pg_receivewal (Georgios Kokolatos)

  This is enabled via `--compress=lz4` and requires binaries to be built using `--with-lz4`.

- Add additional capabilities to pg_receivewal's `--compress` option (Georgios Kokolatos)
- Improve pg_receivewal's ability to restart at the proper WAL location (Ronan Dunklau)

  Previously, pg_receivewal would start based on the WAL file stored in the local archive directory, or at the sending server's current WAL flush location. With this change, if the sending server is running Postgres 15 or later, the local archive directory is empty, and a replication slot is specified, the replication slot's restart point will be used.

- Add pg_rewind option `--config-file` to simplify use when server configuration files are stored outside the data directory (Gunnar Bluth)

E.20.3.9.1. pg_upgrade

- Store pg_upgrade's log and temporary files in a subdirectory of the new cluster called `pg_upgrade_output.d` (Justin Pryzby)

  Previously such files were left in the current directory, requiring manual cleanup. Now they are automatically removed on successful completion of pg_upgrade.

- Disable default status reporting during pg_upgrade operation if the output is not a terminal (Andres Freund)

  The status reporting output can be enabled for non-tty usage by using `--verbose`.

- Make pg_upgrade report all databases with invalid connection settings (Jeevan Ladhe)

  Previously only the first database with an invalid connection setting was reported.

- Make pg_upgrade preserve tablespace and database OIDs, as well as relation relfilenode numbers (Shruthi Gowda, Antonin Houska)

- Add a `--no-sync` option to pg_upgrade (Michael Paquier)

  This is recommended only for testing.

- Limit support of pg_upgrade to old servers running PostgreSQL 9.2 or later (Tom Lane)

E.20.3.9.2. pg_waldump

- Allow pg_waldump output to be filtered by relation file node, block number, fork number, and full page images (David Christensen, Thomas Munro)

- Make pg_waldump report statistics before an interrupted exit (Bharath Rupireddy)

  For example, issuing a control-C in a terminal running `pg_waldump --stats --follow` will report the current statistics before exiting. This does not work on Windows.

- Improve descriptions of some transaction WAL records reported by pg_waldump (Masahiko Sawada, Michael Paquier)

- Allow pg_waldump to dump information about multiple resource managers (Heikki Linnakangas)

  This is enabled by specifying the `--rmgr` option multiple times.

E.20.3.10. Documentation

- Add documentation for `pg_encoding_to_char()` and `pg_char_to_encoding()` (Ian Lawrence Barwick)

- Document the ^@ starts-with operator (Tom Lane)

E.20.3.11. Source Code

- Add support for continuous integration testing using cirrus-ci (Andres Freund, Thomas Munro, Melanie Plageman)

- Add configure option `--with-zstd` to enable Zstandard builds (Jeevan Ladhe, Robert Haas, Michael Paquier)

- Add an ABI identifier field to the magic block in loadable libraries, allowing non-community PostgreSQL distributions to identify libraries that are not compatible with other builds (Peter Eisentraut)

  An ABI field mismatch will generate an error at load time.

- Create a new `pg_type.typcategory` value for `"char"` (Tom Lane)

  Some other internal-use-only types have also been assigned to this category.

- Add new protocol message `TARGET` to specify a new `COPY` method to be used for base backups (Robert Haas)

  pg_basebackup now uses this method.

- Add new protocol message `COMPRESSION` and `COMPRESSION_DETAIL` to specify the compression method and options (Robert Haas)

- Remove server support for old `BASE_BACKUP` command syntax and base backup protocol (Robert Haas)

- Add support for extensions to set custom backup targets (Robert Haas)

- Allow extensions to define custom WAL resource managers (Jeff Davis)
- Add function `pg_settings_get_flags()` to get the flags of server variables (Justin Pryzby)
- On Windows, export all the server's global variables using `PGDLLIMPORT` markers (Robert Haas)

  Previously, only specific variables were accessible to extensions on Windows.
- Require GNU make version 3.81 or later to build PostgreSQL (Tom Lane)
- Require OpenSSL to build the pgcrypto extension (Peter Eisentraut)
- Require Perl version 5.8.3 or later (Dagfinn Ilmari Mannsåker)
- Require Python version 3.2 or later (Andres Freund)

E.20.3.12. Additional Modules

- Allow amcheck to check sequences (Mark Dilger)
- Improve amcheck sanity checks for TOAST tables (Mark Dilger)
- Add new module basebackup_to_shell as an example of a custom backup target (Robert Haas)
- Add new module basic_archive as an example of performing archiving via a library (Nathan Bossart)
- Allow btree_gist indexes on boolean columns (Emre Hasegeli)

  These can be used for exclusion constraints.
- Fix pageinspect's `page_header()` to handle 32-kilobyte page sizes (Quan Zongliang)

  Previously, improper negative values could be returned in certain cases.
- Add counters for temporary file block I/O to pg_stat_statements (Masahiko Sawada)
- Add JIT counters to pg_stat_statements (Magnus Hagander)
- Add new module pg_walinspect (Bharath Rupireddy)

  This gives SQL-level output similar to pg_waldump.
- Indicate the permissive/enforcing state in sepgsql log messages (Dave Page)

E.20.3.12.1. postgres_fdw

- Allow postgres_fdw to push down `CASE` expressions (Alexander Pyhalov)
- Add server variable `postgres_fdw.application_name` to control the application name of postgres_fdw connections (Hayato Kuroda)

  Previously the remote session's `application_name` could only be set on the remote server or via a postgres_fdw connection specification. `postgres_fdw.application_name` supports some escape sequences for customization, making it easier to tell such connections apart on the remote server.
- Allow parallel commit on postgres_fdw servers (Etsuro Fujita)

  This is enabled with the `CREATE SERVER` option `parallel_commit`.

E.20.4. Acknowledgments

The following individuals (in alphabetical order) have contributed to this release as patch authors, committers, reviewers, testers, or reporters of issues.

Abhijit Menon-Sen
Adam Brusselback
Adam Mackler
Adrian Ho
Ahsan Hadi
Ajin Cherian
Alastair McKinley
Aleksander Alekseev
Ales Zeleny
Alex Kingsborough
Alex Kozhemyakin
Alexander Korotkov
Alexander Kukushkin
Alexander Lakhin
Alexander Nawratil
Alexander Pyhalov
Alexey Borzov
Alexey Ermakov
Aliaksandr Kalenik
Álvaro Herrera
Amit Kapila
Amit Khandekar
Amit Langote
Amul Sul
Anastasia Lubennikova
Anders Kaseorg
Andreas Dijkman
Andreas Grob
Andreas Seltenreich
Andrei Zubkov
Andres Freund
Andrew Alsup
Andrew Bille
Andrew Dunstan
Andrew Gierth
Andrew Kesper
Andrey Borodin
Andrey Lepikhov
Andrey Sokolov
Andy Fan
Anton Melnikov
Anton Voloshin
Antonin Houska
Arjan van de Ven
Arne Roland
Arthur Zakirov
Ashutosh Bapat
Ashutosh Sharma
Ashwin Agrawal
Asif Rehman
Asim Praveen
Atsushi Torikoshi
Aya Iwata

Bauyrzhan Sakhariyev
Benoit Lobréau
Bernd Dorn
Bertrand Drouvot
Bharath Rupireddy
Björn Harrtell
Boris Kolpackov
Boris Korzun
Brad Nicholson
Brar Piening
Bruce Momjian
Bruno da Silva
Bryn Llewellyn
Carl Sopchak
Cary Huang
Chapman Flack
Chen Jiaoqian
Chris Bandy
Chris Lowder
Christian Quest
Christoph Berg
Christoph Heiss
Christophe Pettus
Christopher Painter-Wakefield
Claudio Freire
Clemens Zeidler
Corey Huinker
Dag Lem
Dagfinn Ilmari Mannsåker
Dan Kubb
Daniel Cherniy
Daniel Gustafsson
Daniel Polski
Daniel Vérité
Daniel Westermann
Daniele Varrazzo
Daniil Anisimov
Danny Shemesh
Darafei Praliaskouski
Daria Lepikhova
Dave Cramer
Dave Page
David Christensen
David Fetter
David G. Johnston
David Rowley
David Steele
David Zhang
Dean Rasheed
Dian Fay
Dilip Kumar
Dipesh Pandit
Dmitry Dolgov
Dmitry Koval
Dmitry Marakasov
Dominique Devienne
Dong Wook
Drew DeVault

Eduard Català
Egor Chindyaskin
Egor Rogov
Ekaterina Kiryanova
Elena Indrupskaya
Elvis Pranskevichus
Emmanuel Quincerot
Emre Hasegeli
Eric Mutta
Erica Zhang
Erik Rijkers
Erki Eessaar
Etsuro Fujita
Euler Taveira
Fabien Coelho
Fabrice Chapuis
Fabrice Fontaine
Fabrízio de Royes Mello
Feike Steenbergen
Filip Gospodinov
Florin Irion
Floris Van Nee
Frédéric Yhuel
Gabriela Serventi
Gaurab Dey
Geoff Winkless
Geoffrey Blake
Georgios Kokolatos
Gilles Darold
Greg Nancarrow
Greg Rychlewski
Greg Sabino Mullane
Greg Stark
Gregory Smith
Guillaume Lelarge
Gunnar Bluth
Gurjeet Singh
Haiyang Wang
Haiying Tang
Hannu Krosing
Hans Buschmann
Hayato Kuroda
Heath Lord
Heikki Linnakangas
Herwig Goemans
Himanshu Upadhyaya
Holly Roberts
Hou Zhijie
Hubert Lubaczewski
Ian Barwick
Ian Campbell
Ibrar Ahmed
Ildus Kurbangaliev
Ilya Anfimov
Itamar Gafni
Jacob Champion
Jaime Casanova
Jakub Wartak

James Coleman
James Hilliard
James Inform
Jan Piotrowski
Japin Li
Jason Harvey
Jason Kim
Jean-Christophe Arnu
Jeevan Ladhe
Jeff Davis
Jeff Janes
Jehan-Guillaume de Rorthais
Jelte Fennema
Jeremy Evans
Jeremy Schneider
Jian Guo
Jian He
Jimmy Yih
Jiri Fejfar
Jitka Plesníková
Joe Conway
Joe Wildish
Joel Jacobson
Joey Bodoia
John Naylor
Jonathan Katz
Josef Simanek
Joseph Koshakow
Josh Soref
Joshua Brindle
Juan José Santamaría Flecha
Julien Rouhaud
Julien Roze
Junwang Zhao
Jürgen Purtz
Justin Pryzby
Ken Kato
Kevin Burke
Kevin Grittner
Kevin Humphreys
Kevin McKibbin
Kevin Sweet
Kevin Zheng
Klaudie Willis
Konstantin Knizhnik
Konstantina Skovola
Kosei Masumura
Kotaro Kawamoto
Koyu Tanigawa
Kuntal Ghosh
Kyotaro Horiguchi
Lars Kanis
Lauren Fliksteen
Laurent Hasson
Laurenz Albe
Leslie Lemaire
Liam Bowen
Lingjie Qiang

Liu Huailing
Louis Jachiet
Lukas Fittl
Ma Liangzhu
Maciek Sakrejda
Magnus Hagander
Mahendra Singh Thalor
Maksim Milyutin
Marc Bachmann
Marcin Krupowicz
Marcus Gartner
Marek Szuba
Marina Polyakova
Mario Emmenlauer
Mark Dilger
Mark Murawski
Mark Wong
Markus Wanner
Markus Winand
Martijn van Oosterhout
Martin Jurca
Martin Kalcher
Martín Marqués
Masahiko Sawada
Masahiro Ikeda
Masao Fujii
Masaya Kawamoto
Masayuki Hirose
Matthias van de Meent
Matthijs van der Vleuten
Maxim Orlov
Maxim Yablokov
Melanie Plageman
Michael Banck
Michael Harris
Michael J. Sullivan
Michael Meskes
Michael Mühlbeyer
Michael Paquier
Michael Powers
Mike Fiedler
Mike Oh
Mikhail Kulagin
Miles Delahunty
Naoki Okano
Nathan Bossart
Nathan Long
Nazir Bilal Yavuz
Neha Sharma
Neil Chen
Nicola Contu
Nicolas Lutic
Nikhil Benesch
Nikhil Shetty
Nikhil Sontakke
Nikita Glukhov
Nikolai Berkoff
Nikolay Samokhvalov

Nikolay Shaplov
Nitin Jadhav
Noah Misch
Noboru Saito
Noriyoshi Shinoda
Olaf Bohlen
Olly Betts
Onder Kalaci
Oskar Stenberg
Otto Kekalainen
Paul Guo
Paul Jungwirth
Paul Martinez
Pavan Deolasee
Pavel Borisov
Pavel Luzanov
Pavel Stehule
Peter Eisentraut
Peter Geoghegan
Peter Slavov
Peter Smith
Petr Jelínek
Phil Florent
Phil Krylov
Pierre-Aurélien Georges
Prabhat Sahu
Quan Zongliang
Rachel Heaton
Rahila Syed
Rajakavitha Kodhandapani
Rajkumar Raghuwanshi
Ranier Vilela
Rei Kamigishi
Reid Thompson
Rémi Lapeyre
Renan Soares Lopes
Richard Guo
Richard Wesley
RKN Sai Krishna
Robert Haas
Robert Treat
Roberto Mello
Robins Tharakan
Roger Mason
Roman Zharkov
Ronan Dunklau
Rui Zhao
Ryan Kelly
Ryo Matsumura
Ryohei Takahashi
Sadhuprasad Patro
Sait Talha Nisanci
Sami Imseih
Sandeep Thakkar
Sebastian Kemper
Sehrope Sarkuni
Sergei Kornilov
Sergei Shoulbakov

Sergey Shinderuk
Shay Rojansky
Shenhao Wang
Shi Yu
Shinya Kato
Shruthi Gowda
Simon Perepelitsa
Simon Riggs
Sirisha Chamarthi
Soumyadeep Chakraborty
Stan Hu
Stas Kelvich
Stefen Hillman
Stephen Frost
Steve Chavez
Sumanta Mukherjee
Suraj Khamkar
Suraj Kharage
Sven Klemm
Takamichi Osumi
Takayuki Tsunakawa
Takeshi Ideriha
Tatsuhiro Nakamori
Tatsuhito Kasahara
Tatsuo Ishii
Tatsuro Yamada
Teja Mupparti
Teodor Sigaev
Thibaud Walkowiak
Thom Brown
Thomas McKay
Thomas Munro
Tim McNamara
Timo Stolz
Timur Khanjanov
Tom Lane
Tomas Barton
Tomas Vondra
Tony Reix
Troy Frericks
Tushar Ahuja
Victor Wagner
Victor Yegorov
Vignesh C
Vik Fearing
Vincas Dargis
Vitaly Burovoy
Vitaly Voronov
Vladimir Sitnikov
Wang Ke
Wei Sun
Wei Wang
Whale Song
Will Mortensen
Wolfgang Walther
Yanliang Lei
Yaoguang Chen
Yogendra Suralkar

YoungHwan Joo
Yugo Nagata
Yukun Wang
Yura Sokolov
Yusuke Egashira
Yuzuko Hosoya
Zhang Mingli
Zhang Wenjie
Zhihong Yu
Zhiyong Wu

E.21. Versions précédentes

Les notes de versions des branches précédentes sont disponibles sur le site web https://www.postgresql.org/docs/release/.

Annexe F. Modules supplémentaires fournis

Cette annexe et la suivante contiennent des informations concernant les modules disponibles dans le répertoire `contrib` de la distribution PostgreSQL. Ce sont des outils de portage, des outils d'analyse, des fonctionnalités supplémentaires qui ne font pas partie du système PostgreSQL de base, principalement parce qu'ils s'adressent à une audience limitée ou sont trop expérimentaux pour faire partie de la distribution de base. Cela ne concerne en rien leur utilité.

Cette annexe couvre les extensions et quelques autres modules plug-in du serveur disponibles dans le répertoire `contrib` du répertoire des sources.Annexe G couvre les programmes outils.

Lors de la construction à partir des sources de la distribution, ces extensions ne sont pas construites automatiquement, sauf si vous utilisez la cible « world » (voir Étape 2). Ils peuvent être construits et installés en exécutant :

```
make
make install
```

dans le répertoire `contrib` d'un répertoire des sources configuré ; ou pour ne construire et installer qu'un seul module sélectionné, on exécute ces commandes dans le sous-répertoire du module. Beaucoup de ces modules ont des tests de régression qui peuvent être exécutés en lançant la commande :

```
make check
```

avant l'installation ou

```
make installcheck
```

une fois que le serveur PostgreSQL est démarré.

Lorsqu'une version packagée de PostgreSQL est utilisée, ces modules sont typiquement disponibles dans un package séparé, comme par exemple `postgresql-contrib`.

Beaucoup de ces modules fournissent de nouvelles fonctions, de nouveaux opérateurs ou types utilisateurs. Pour pouvoir utiliser un de ces modules, après avoir installé le code, il faut enregistrer les nouveaux objets SQL dans la base de données. Cela se fait en exécutant la commande CREATE EXTENSION. Dans une base de données neuve, vous pouvez simplement faire :

```
CREATE EXTENSION nom_module;
```

Cette commande enregistre de nouveaux objets SQL dans la base de données courante, donc vous avez besoin d'exécuter cette commande dans chaque base de données où vous souhaitez l'utiliser. Autrement, exécutez-la dans la base de données `template1` pour que l'extension soit copiée dans les bases de données créées après.

Pour tous ces modules, `CREATE EXTENSION` doit être exécuté par un superutilisateur, sauf si le module est considéré « trusted » (de confiance), auquel cas l'instruction peut être exécutée par un

utilisateur qui a l'attribut CREATE sur la base de données courante. Les modules de confiance sont identifiés comme tels dans les sections qui suivent. Généralement, les modules de confiance sont des modules qui ne peuvent pas fournir d'accès à des fonctionnalité hors de la base.

Beaucoup de modules vous permettent d'installer leurs objets dans le schéma de votre choix. Pour cela, ajoutez SCHEMA *nom_schéma* à la commande CREATE EXTENSION. Par défaut, les objets seront placés dans le schéma de création par défaut, qui est par défaut public.

Néanmoins, notez que certains de ces modules ne sont pas des « extensions » dans ce sens, mais sont chargés sur le serveur d'une autre façon, par le biais de shared_preload_libraries. Voir la documentation de chaque module pour les détails.

F.1. adminpack

L'adminpack fournit un certain nombre de fonctions de support que pgAdmin ou d'autres outils de gestion et d'administration peuvent utiliser pour fournir des fonctionnalités supplémentaires, comme la gestion à distance de journaux applicatifs. L'utilisation de toutes ces fonctions est seulement autorisée aux superutilisateurs par défaut mais peut être autorisée à d'autres utilisateurs en utilisant la commande GRANT.

Les fonctions affichées dans Tableau F.1 fournissent des accès en écriture aux fichiers de la machine hébergeant le serveur. (Voir aussi les fonctions dans Tableau 9.99, qui fournissent des accès en lecture seule.) Seuls les fichiers du répertoire principal de l'instance sont accessibles, sauf si l'utilisateur a l'attribut SUPERUSER ou fait partie des rôles pg_read_server_files ou pg_write_server_files, suivant la fonction. Les chemins relatifs et absolus sont permis.

Tableau F.1. Fonctions de adminpack

Fonction Description
pg_catalog.pg_file_write (*filename* text, *data* text, *append* boolean) → bigint Écrit dans un fichier
pg_catalog.pg_file_sync (*filename* text) → void Vide un fichier ou un répertoire sur disque.
pg_catalog.pg_file_rename (*oldname* text, *newname* text [, *archivename* text]) → boolean Renomme un fichier
pg_catalog.pg_file_unlink (*filename* text) → boolean Supprime un fichier.
pg_catalog.pg_logdir_ls () → setof record Liste les fichiers de trace du répertoire précisé par log_directory.

pg_file_write écrit les données indiquées par le paramètre *data* dans le fichier indiqué par le paramètre *filename*. Si le paramètre *append* vaut false, le fichier ne doit pas déjà exister. S'il vaut true, le fichier peut déjà exister et les données y seront ajoutées. Renvoit le nombre d'octets écrits.

pg_file_sync synchronise sur disque le fichier ou le répertoire indiqué d'après son *filename*. Une erreur est renvoyée en cas d'échec (par exemple si le fichier indiqué n'est pas présent). Notez que data_sync_retry n'a pas d'effet sur cette fonction, et de ce fait, une erreur de niveau PANIC ne sera pas renvoyée même en cas d'échec pour synchroniser les fichiers de la base.

`pg_file_rename` renomme un fichier. Si *`archivename`* est omis ou vaut NULL, il renomme simplement *`oldname`* en *`newname`* (qui ne doit pas déjà exister). Si *`archivename`* est fourni, il renomme tout d'abord *`newname`* en *`archivename`* (qui ne doit pas déjà exister), puis il renomme *`oldname`* en *`newname`*. En cas d'échec à la deuxième étape, il essaiera de renommer *`archivename`* en *`newname`* avant de renvoyer l'erreur. Renvoit true en cas de succès, false si les fichiers sources ne sont pas présents ou modifiables. Dans tous les autres cas, elle renvoit une erreur.

`pg_file_unlink` supprime le fichier indiqué. Renvoit true en cas de succès, false si le fichier spécifié n'est pas présent ou si l'appel à `unlink()` échoue. Dans tous les autres cas, elle renvoit une erreur.

`pg_logdir_ls` renvoit l'horodatage et le chemin de tous les journaux applicatifs stockés dans le répertoire indiqué par le paramètre log_directory. Le paramètre log_filename doit avoir sa configuration par défaut (`postgresql-%Y-%m-%d_%H%M%S.log`) pour utiliser cette fonction.

F.2. amcheck

Le module `amcheck` fournit des fonctions vous permettant de vérifier la cohérence logique de la structure des relations.

Les fonctions spécifiques aux B-Tree vérifient diverses *propriétés invariantes* dans la structure de la représentation de certaines relations. La justesse des fonctions d'accès aux données durant les parcours d'index et d'autres opérations importantes reposent sur le fait que ces propriétés invariantes sont toujours vraies. Par exemple, certaines fonctions vérifient, entre autres choses, que tous les blocs d'index B-Tree ont leurs éléments dans un ordre « logique » (par exemple, pour des index B-Tree sur un type `text`, les lignes de l'index devraient être triées dans l'ordre alphabétique défini par la collation). Si, pour une raison ou une autre, cette propriété invariante spécifique n'est plus vérifiée, il faut s'attendre à ce que des recherches binaires sur la page affectée renseignent de manière erronée les parcours d'index, avec pour conséquence des résultats de requêtes SQL erronés. Si la structure semble valide, aucune erreur n'est levée.

La vérification est réalisée en utilisant les mêmes procédures que celles utilisées par les parcours d'index eux-mêmes, qui peuvent très bien être du code pour une classe d'opérateurs utilisateur. Par exemple, la vérification d'index B-Tree s'appuie sur les comparaisons faites avec une ou plusieurs routines pour la fonction de support 1. Voir Section 38.16.3 pour plus de détails sur les fonctions de support des classes d'opérateurs.

À la différence des fonctions de vérification B-Tree qui rapportent la corruption en remontant des erreurs, la fonction de vérification de table `verify_heapam` vérifie une table et tente de renvoyer un ensemble de lignes, une par corruption détectée. Malgré tout, si des installations, se reposant sur la fonction `verify_heapam`, sont elles-mêmes corrompues, la fonction pourrait ne pas être capable de continuer et leverait une erreur.

Le droit d'exécuter les fonctions `amcheck` peut être accordé aux utilisateurs normaux, sans attribut `SUPERUSER`, mais avant de donner cette permission, de prudentes considérations doivent être prises concernant la sécurité des données et les problèmes de confidentialité. Bien que les rapports de corruption générés par ces fonctions ne se focalisent pas tant sur le contenu des données corrompues que sur la structure de ces données et la nature de la corruption détectée, un attaquant qui gagnerait le droit d'exécuter ces fonctions, tout particulièrement si l'attaquant peut aussi causer de la corruption, pourrait être capable de déduire quelque chose des données elles-mêmes dans les messages.

F.2.1. Fonctions

`bt_index_check(index regclass, heapallindexed boolean) returns void`

`bt_index_check` vérifie que sa cible, un index B-Tree, respecte un éventail de propriétés invariables. Exemple d'utilisation :

```
test=# SELECT bt_index_check(index => c.oid, heapallindexed =>
 i.indisunique),
              c.relname,
              c.relpages
FROM pg_index i
JOIN pg_opclass op ON i.indclass[0] = op.oid
JOIN pg_am am ON op.opcmethod = am.oid
JOIN pg_class c ON i.indexrelid = c.oid
JOIN pg_namespace n ON c.relnamespace = n.oid
WHERE am.amname = 'btree' AND n.nspname = 'pg_catalog'
-- Ignore les tables temporaires, qui peuvent appartenir à
 d'autres sessions
AND c.relpersistence != 't'
-- La fonction peut renvoyer une erreur si cela est omis :
AND c.relkind = 'i' AND i.indisready AND i.indisvalid
ORDER BY c.relpages DESC LIMIT 10;

 bt_index_check |             relname             | relpages
----------------+---------------------------------+----------
                | pg_depend_reference_index       |       43
                | pg_depend_depender_index        |       40
                | pg_proc_proname_args_nsp_index  |       31
                | pg_description_o_c_o_index      |       21
                | pg_attribute_relid_attnam_index |       14
                | pg_proc_oid_index               |       10
                | pg_attribute_relid_attnum_index |        9
                | pg_amproc_fam_proc_index        |        5
                | pg_amop_opr_fam_index           |        5
                | pg_amop_fam_strat_index         |        5
(10 rows)
```

Cet exemple montre une session qui vérifie les 10 plus gros index du catalogue de la base de données « test ». La vérification de la présence des lignes de la table dans l'index est demandée pour le sous-ensemble des index uniques. Puisqu'aucune erreur n'est retournée, tous les index testés semblent être cohérents au niveau logique. Bien évidemment, cette requête pourrait être facilement modifiée pour appeler `bt_index_check` sur chacun des index présents dans la base de données pour lesquels la vérification est supportée.

`bt_index_check` acquiert un `AccessShareLock` sur l'index cible ainsi que sur la relation à laquelle il appartient. Ce niveau de verrouillage est le même que celui acquis sur les relations lors d'une simple requête `SELECT`. `bt_index_check` ne vérifie pas les propriétés invariantes qui englobent les relations enfant/parent, mais vérifiera la présence de toutes les lignes de la table et des lignes d'index à l'intérieur de l'index quand *`heapallindexed`* vaut `true`. Quand il faut lancer un test de recherche de corruption, de routine et pas trop lourd, sur un environnement de production, l'utilisation de `bt_index_check` offre généralement le meilleur compromis entre vérification minutieuse et impact limité sur les performances et la disponibilité de l'application.

`bt_index_parent_check(index regclass, heapallindexed boolean, rootdescend boolean) returns void`

`bt_index_parent_check` teste que sa cible, un index B-Tree, respecte un certain nombre de propriétés invariantes. En option, quand l'argument *`heapallindexed`* vaut `true`, la fonction vérifie la présence de toutes les lignes dans la table. Quand l'argument optionnel *`rootdescend`* vaut `true`, la vérification recherche de nouveau les lignes au niveau des feuilles en réalisant une nouvelle recherche à partir de la racine pour chaque ligne. Les vérifications réalisables par `bt_index_parent_check` sont un surensemble des vérifications réalisées par `bt_index_check`. On peut voir `bt_index_parent_check` comme une version plus minutieuse de `bt_index_check` : contrairement à `bt_index_check`,

`bt_index_parent_check` vérifie également les propriétés invariantes qui englobent les relations parent/enfant, et la présence de tous les liens dans la structure de l'index. `bt_index_parent_check` respecte la convention habituelle qui consiste à renvoyer une erreur si une incohérence ou tout autre problème est détecté.

Un verrou de type `ShareLock` est requis sur l'index ciblé par `bt_index_parent_check` (un `ShareLock` est également acquis sur la relation associée). Ces verrous empêchent des modifications concurrentes par des commandes `INSERT`, `UPDATE` et `DELETE`. Les verrous empêchent également la relation associée d'être traitée de manière concurrente par `VACUUM`, ainsi que toute autre commande d'administration. Il est à noter que cette fonction ne conserve les verrous que durant son exécution et non pas durant l'intégralité de la transaction.

La vérification supplémentaire qu'opère `bt_index_parent_check` est plus apte à détecter différents cas pathologiques. Ces cas peuvent impliquer une classe d'opérateur B-Tree implémentée de manière incorrecte utilisée pour l'index vérifié, ou, hypothétiquement, des bugs non encore découverts dans le code de la méthode d'accès associée à cet index B-Tree. Il est à noter que `bt_index_parent_check` ne peut pas être utilisé lorsque le mode Hot Standby est activé (c'est-à-dire, sur un serveur secondaire disponible en lecture seule), contrairement à `bt_index_check`.

> **Astuce**
>
> Les fonctions `bt_index_check` et `bt_index_parent_check` journalisent toutes les deux des messages sur la vérification, avec le niveau de sévérité `DEBUG1` et `DEBUG2`. Ces messages fournissent des informations détaillées sur la vérification, exploitables par les développeurs PostgreSQL. Les utilisateurs avancés peuvent aussi trouver ces informations utiles, car elles fournissent du contexte additionnel si jamais la vérification détecte une incohérence. Exécuter
>
> ```
> SET client_min_messages = DEBUG1;
> ```
>
> dans une session interactive, psql, avant d'exécuter une requête de vérification, va afficher les messages sur sa progression avec un niveau de détail raisonnable.

`verify_heapam(relation regclass, on_error_stop boolean, check_toast boolean, skip text, startblock bigint, endblock bigint, blkno OUT bigint, offnum OUT integer, attnum OUT integer, msg OUT text) returns setof record`

Vérifie une table, une séquence ou une vue matérialisée sur des corruptions structurelles, où les pages de relation contiennent des données au format invalide, et sur des corruptions logiques, où les pages sont valides structurellement mais incohérentes avec le reste de l'instance de base de données.

Les arguments optionnels suivants sont reconnus :

`on_error_stop`

Si true, la vérification de corruption s'arrête à la fin du premier bloc dans lequel de la corruption est trouvée.

La valeur par défaut est false.

`check_toast`

Si true, les valeurs TOAST sont vérifiées sur la table TOAST de la relation cible.

Cette option est connue pour être lente. Aussi, si une table TOAST ou un de ses index est corrompu, vérifier les valeurs TOAST peut, en théorie, faire tomber le serveur. Cependant dans la majorité des cas, cela produira juste une erreur.

La valeur par défaut est false.

`skip`

Si ce paramètre n'est pas mis à `none`, la vérification de corruption saute les blocs qui sont marqués tous visibles ou tous gelés, comme définie par la valeur de l'option. Les options valides sont `all-visible`, `all-frozen` et `none`.

La valeur par défaut est `none`.

`startblock`

Si indiquée, la vérification de corruption commence au bloc spécifié, sautant tous les blocs précédents. Si *`startblock`* est spécifiée à une valeur en dehors de la plage des blocs de la table cible, une erreur est renvoyée.

Par défaut, la vérification débute au premier bloc.

`endblock`

Si indiquée, la vérification de corruption termine à ce bloc, ignorant tous les blocs restants. Spécifier un *`endblock`* en dehors de la plage des blocs d'une table cible renvoie une erreur.

Par défaut, tous les blocs sont vérifiés.

Pour chaque corruption détectée, `verify_heapam` renvoie une ligne avec les colonnes suivantes :

`blkno`

Le numéro de bloc contenant le bloc corrompu.

`offnum`

Le décalage pour la ligne corrompue.

`attnum`

Le numéro d'attribut de la colonne corrompue dans la ligne, si la corruption est spécifique à la colonne et non pas à la ligne en entier.

`msg`

Un message décrivant le problème détecté.

F.2.2. Vérification optionnelle *`heapallindexed`*

Quand l'argument *`heapallindexed`* des fonctions de vérification des B-Tree est à `true`, une phase de vérification supplémentaire est opérée sur la table associée à l'index cible. Elle consiste en une opération `CREATE INDEX CONCURRENTLY` « bidon » qui vérifie la présence de tous les nouveaux enregistrements hypothétiques d'index dans une structure de récapitulation temporaire et en mémoire (elle est construite au besoin durant la première phase de la vérification). Cette structure de récapitulation prend l'« empreinte digitale » de chaque enregistrement rencontré dans l'index cible. Le principe directeur derrière la vérification *`heapallindexed`* est qu'un nouvel index équivalent à l'index cible existant ne doit avoir que des entrées que l'on peut trouver dans la structure existante.

La phase *`heapallindexed`* supplémentaire a un coût significatif : typiquement, la vérification peut être plusieurs fois plus longue. Cependant, il n'y a pas de changement quant aux verrous acquis au niveau de la table quand on opère une vérification *`heapallindexed`*.

La structure de récapitulation est limitée en taille par `maintenance_work_mem`. Pour s'assurer que la probabilité de rater une incohérence ne dépasse 2 % pour chaque enregistrement qui devrait être dans l'index, il faut environ 2 octets de mémoire par enregistrement. Quand moins de mémoire est disponible par enregistrement, la probabilité de manquer une incohérence augmente lentement. Cette approche limite significativement le coût de la vérification, tout en réduisant légèrement la probabilité de détecter un problème, particulièrement sur les installations où la vérification est traitée comme une opération de maintenance de routine. Tout enregistrement absent ou déformé a une nouvelle chance d'être détecté avec chaque lancement de la vérification.

F.2.3. Utiliser amcheck efficacement

`amcheck` peut être efficace pour détecter différents types de modes d'échec que les sommes de contrôle n'arriveront jamais à détecter. Cela inclut :

- Les incohérences dans la structure causées par des implémentations incorrectes de classe d'opérateurs.

  Cela inclut également des problèmes causés par le changement des règles de comparaison des collations du système d'exploitation. Les comparaisons des données d'un type ayant une collation comme `text` doivent être immuables (tout comme toutes les autres comparaisons utilisées pour les parcours d'index B-Tree doivent être immuables), ce qui implique que les règles de collation du système d'exploitation ne doivent jamais changer. Bien que cela soit rare, des mises à jour des règles des collations du système d'exploitation peuvent causer ces problèmes. Le plus souvent, c'est une incohérence dans l'ordre de collation entre le serveur principal et son serveur de secours qui est en cause, généralement due au fait que les versions majeures du système d'exploitation utilisées ne sont pas les mêmes. Plus généralement, une incohérence dans l'ordre de collation entre un serveur primaire et son réplica est impliquée, peut-être parce que la version *majeure* du système d'exploitation utilisée est différente. De telles incohérences ne surviendront généralement que sur des serveurs secondaires, et ne pourront par conséquent être détectées que là.

  Si un tel problème survient, il se peut que cela n'affecte pas chaque index qui utilise le tri d'une collation affectée, tout simplement parce que les valeurs *indexées* pourraient avoir le même ordre de tri absolu indépendamment des incohérences comportementales. Voir Section 24.1 et Section 24.2 pour plus de détails sur comment PostgreSQL utilise les locales et collations du système d'exploitation.

- Les incohérences dans la structure entre les index et les tables indexées (lorsque la vérification *`heapallindexed`* est réalisée).

  Il n'y a pas de vérification avec la table initiale en temps normal. Les symptômes d'une corruption de la table peuvent être subtils.

- La corruption causée par un hypothétique bug non encore découvert dans le code de la méthode d'accès dans PostgreSQL, dans le code effectuant le tri ou le code de gestion transactionnelle.

  La vérification automatique de l'intégrité structurelle des index joue un rôle dans les tests généraux des fonctionnalités de PostgreSQL, nouvelles ou proposées, qui pourraient possiblement permettre l'introduction d'incohérences logiques. La vérification de la structure de la table et des informations de visibilité et de statut des transactions associées joue un rôle similaire. Une stratégie de test évidente est d'appeler les fonctions d'`amcheck` de manière continue en même temps que les tests de régression standards sont lancés. Voir Section 33.1 pour plus de détails sur comment lancer les tests.

- Les failles dans le système de fichiers ou dans le sous-système de stockage quand les sommes de contrôles ne sont pas activées.

  Il est à noter que `amcheck` n'examine une page que telle qu'elle se présente dans un tampon en mémoire partagée lors de la vérification, et qu'en accédant au bloc dans la mémoire partagée. Par conséquent, `amcheck` n'examine pas forcément les données lues depuis le système de fichiers au moment de la vérification. Notez que si les sommes de contrôles sont activées, `amcheck` peut lever une erreur de somme de contrôle incorrecte quand un bloc corrompu est lu vers un tampon.

- Les corruptions causées par une RAM défaillante, et plus largement le sous-système mémoire et le système d'exploitation.

  PostgreSQL ne protège pas contre les erreurs mémoire corrigibles, et il est supposé que vous utilisez de la mémoire RAM de standard industriel ECC (*Error Correcting Codes*) ou avec une meilleure protection. Cependant, la mémoire ECC n'est typiquement immunisée que contre les erreurs d'un seul bit, et il ne faut pas partir du principe que ce type de mémoire fournit une protection *absolue* contre les défaillances provoquant une corruption de la mémoire.

  Quand la vérification *`heapallindexed`* est réalisée, il y a une chance fortement accrue de détecter des erreurs d'un bit car l'égalité binaire stricte est testée et les attributs indexés de la table sont testés.

La corruption structurelle peut survenir par un stockage matériel défectueux, ou par des fichiers de relation réécrits ou modifiés par un logiciel sans rapport. Ce type de corruption peut aussi être détecté avec les sommes de contrôle des fichiers de données.

Des pages de relation qui sont correctement formatées, internalement cohérentes, et correctes par rapport à leurs propres sommes de contrôle internes, peuvent encore contenir de la corruption logique. Par conséquent, ce type de corruption ne peut pas être détecté avec checksums. Des exemples incluent les valeurs TOAST dans la table principale pour lesquelles il manque l'entrée correspondante dans la table TOAST, et les lignes dans la table principale avec un identifiant de transaction plus ancien que le plus ancien identifiant de transaction valide dans la base de données ou l'instance.

De multiples causes de corruption logique ont été observées sur des systèmes en production, incluant des bugs dans le logiciel serveur PostgreSQL, des erreurs ou de mauvaises conceptions dans des outils de sauvegarde et restauration, et des erreurs utilisateur.

Les relations corrompues sont plus préoccupantes sur des environnements de production en cours, plus particulièrement ceux où les risques d'activités fortes sont les moins tolérés. Pour cette raison, `verify_heapam` a été conçue pour diagnostiquer la corruption sans risque. Elle ne peut pas protéger contre toutes les causes de crash de processus serveur, car même exécuter la requête appelante pourrait ne pas être sûr sur un système notablement corrompu. L'accès au tables du catalogue est effectué et peut poser problème si le catalogue lui-même est corrompu.

De manière générale, `amcheck` ne peut que prouver la présence d'une corruption ; il ne peut pas en prouver l'absence.

F.2.4. Réparer une corruption

Aucune erreur concernant une corruption remontée par `amcheck` ne devrait être un faux positif. `amcheck` remonte des erreurs dans le cas où des conditions, par définition, ne devraient jamais arriver, et par conséquent une analyse minutieuse des erreurs remontées par `amcheck` est souvent nécessaire.

Il n'y a pas de méthode générale pour réparer les problèmes que `amcheck` détecte. Une explication de la cause principale menant à ce que la propriété invariante soit violée devrait être étudiée. pageinspect peut jouer un rôle utile dans le diagnostic de la corruption qu'`amcheck` détecte. Un `REINDEX` peut échouer à réparer la corruption.

F.3. auth_delay

`auth_delay` demande au serveur d'observer une brève pause avant de rapporter une erreur d'authentification. Cela rend les attaques par force brute plus difficiles. Notez que cela n'empêche en rien les attaques par déni de service et pourrait même les exacerber car les processus qui attendent de rapporter l'échec d'authentification consomment toujours des connexions.

Pour fonctionner, ce module doit être chargé via le paramètre shared_preload_libraries dans `postgresql.conf`.

F.3.1. Paramètres de configuration

`auth_delay.milliseconds` (`integer`)

Le nombre de millisecondes à attendre avant de rapporter une erreur d'authentification. La valeur par défaut est 0.

Ces paramètres doivent être configurés dans le fichier `postgresql.conf`. Voici un exemple typique d'utilisation :

```
# postgresql.conf
shared_preload_libraries = 'auth_delay'

auth_delay.milliseconds = '500'
```

F.3.2. Auteur

KaiGai Kohei <kaigai@ak.jp.nec.com>

F.4. auto_explain

Le module `auto_explain` fournit un moyen de tracer les plans d'exécution des requêtes lentes automatiquement, sans qu'il soit nécessaire de lancer EXPLAIN manuellement. C'est particulièrement utile pour repérer les requêtes non optimisées sur de grosses applications.

Le module ne fournit pas de fonctions accessibles par SQL. Pour l'utiliser, il suffit de le charger sur le serveur. Il peut être chargé dans une session individuelle :

```
LOAD 'auto_explain';
```

(Seul le superutilisateur peut le faire.) Une utilisation plus commune est de le précharger dans certaines ou toutes les sessions, en incluant `auto_explain` dans session_preload_libraries ou dans shared_preload_libraries dans le fichier `postgresql.conf`. Il est alors possible de récupérer les requêtes lentes non prévues, quel que soit le moment où elles se produisent. Évidemment, il y a un prix à payer pour cela.

F.4.1. Paramètres de configuration

Plusieurs paramètres de configuration contrôlent le comportement d'`auto_explain`. Le comportement par défaut est de ne rien faire. Il est donc nécessaire de préciser au minimum `auto_explain.log_min_duration` pour obtenir un résultat.

`auto_explain.log_min_duration` (`integer`)

`auto_explain.log_min_duration` est la durée minimale d'exécution d'une requête (en millisecondes) à partir de laquelle le plan d'exécution sera tracé. La positionner à `0` trace tous les plans. `-1` (la valeur par défaut) désactive l'écriture des plans. Par exemple, si vous la positionnez à `250ms`, tous les ordres qui durent 250 ms ou plus seront tracés. Seuls les superutilisateurs peuvent modifier ce paramétrage.

`auto_explain.log_analyze` (`boolean`)

`auto_explain.log_analyze` entraîne l'écriture du résultat de `EXPLAIN ANALYZE`, à la place du résultat de `EXPLAIN`, lorsqu'un plan d'exécution est tracé. Ce paramètre est désactivé par défaut. Seuls les superutilisateurs peuvent modifier ce paramètre.

> **Note**
>
> Lorsque ce paramètre est activé, un chronométrage par nœud du plan est calculé pour tous les ordres exécutés, qu'ils durent suffisamment longtemps pour être réellement tracés, ou non. Ceci peut avoir des conséquences très négatives sur les performances. Désactiver `auto_explain.log_timing` améliore les performances au prix d'une diminution des informations.

`auto_explain.log_buffers` (`boolean`)

`auto_explain.log_buffers` contrôle l'affichage des statistiques d'utilisation du cache disque de PostgreSQL dans la trace d'un plan d'exécution ; il s'agit de l'équivalent de l'option `BUFFERS` de la commande `EXPLAIN`. Ce paramètre n'a pas d'effet tant que `auto_explain.log_analyze` n'est pas activé. Il est désactivé par défaut.

`auto_explain.log_wal` (`boolean`)

`auto_explain.log_wal` contrôle l'affichage des statistiques d'utilisation des journaux de transaction lorsqu'un plan d'exécution est tracé ; il s'agit de l'équivalent de l'option `WAL` pour la commande `EXPLAIN`. Ce paramètre n'a pas d'effet tant que `auto_explain.log_analyze` n'est pas activé. Ce paramètre est désactivé par défaut. Seuls les superutilisateurs peuvent modifier la valeur de ce paramètre.

`auto_explain.log_timing` (`boolean`)

`auto_explain.log_timing` contrôle l'affichage du chronométrage de chaque nœud lorsqu'un plan d'exécution est tracé ; il s'agit de l'équivalent de l'option `TIMING` pour la commande `EXPLAIN`. La surcharge occasionnée par la lecture répétée de l'horloge système peut ralentir significativement l'exécution des requêtes sur certains systèmes. De ce fait, il peut être utile de désactiver ce paramètre quand seul le nombre de lignes exacts importe. Ce paramètre n'a pas d'effet tant que `auto_explain.log_analyze` n'est pas activé. Il est désactivé par défaut. Seuls les superutilisateurs peuvent modifier la valeur de ce paramètre.

`auto_explain.log_triggers` (`boolean`)

`auto_explain.log_triggers` entraîne la prise en compte des statistiques d'exécution des triggers quand un plan d'exécution est tracé. Ce paramètre n'a pas d'effet tant que `auto_explain.log_analyze` n'est pas activé. Il est désactivé par défaut. Seuls les superutilisateurs peuvent modifier la valeur de ce paramètre.

`auto_explain.log_verbose` (`enum`)

`auto_explain.log_verbose` contrôle l'affichage des détails quand un plan d'exécution est tracé ; il s'agit de l'équivalent de l'option `VERBOSE` pour la commande `EXPLAIN`. Ce paramètre est désactivé par défaut.

`auto_explain.log_settings` (`boolean`)

`auto_explain.log_settings` contrôle quelles informations sont affichées à propos des options de configuration modifiées au moment où le plan est tracé. Ne sont incluses dans la sortie que les options impactant la planification de requêtes avec des valeurs différentes des défauts intégrés à PostgreSQL. Ce paramètre est désactivé par défaut. Seuls les superutilisateurs peuvent le changer.

`auto_explain.log_format` (`boolean`)

`auto_explain.log_format` sélectionne le format de sortie utilisé par la commande `EXPLAIN`. Les valeurs autorisées sont `text`, `xml`, `json` et `yaml`. Le format par défaut est le texte brut.

`auto_explain.log_level` (`enum`)

`auto_explain.log_level` sélectionne le niveau de trace à partir duquel auto_explain va tracer le plan. Les valeurs valides sont `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, et `LOG`. Le défaut est `LOG`. Seuls les superutilisateurs peuvent changer ce paramètre.

`auto_explain.log_nested_statements` (`boolean`)

`auto_explain.log_nested_statements` entraîne la prise en compte des ordres imbriqués (les requêtes exécutées dans une fonction) dans la trace. Quand il est désactivé, seuls les plans d'exécution de plus haut niveau sont tracés. Ce paramètre est désactivé par défaut. Seuls les superutilisateurs peuvent modifier ce paramètre.

`auto_explain.sample_rate` (`real`)

`auto_explain.sample_rate` force auto_explain à tracer le plan d'exécution que d'une fraction des requêtes de chaque session. La valeur par défaut est de 1, autrement dit toutes les requêtes. Dans le cas de requêtes imbriquées, soit toutes se voient tracées leur plan, soit aucune. Seuls les superutilisateurs peuvent modifier ce paramètre.

D'ordinaire, ces paramètres sont configurés dans le fichier `postgresql.conf` mais les superutilisateurs peuvent les modifier en ligne pour leur propres sessions. Voici un exemple typique d'utilisation :

```
# postgresql.conf
session_preload_libraries = 'auto_explain'

auto_explain.log_min_duration = '3s'
```

F.4.2. Exemple

```
postgres=# LOAD 'auto_explain';
postgres=# SET auto_explain.log_min_duration = 0;
postgres=# SET auto_explain.log_analyze = true;
postgres=# SELECT count(*)
    FROM pg_class, pg_index
    WHERE oid = indrelid AND indisunique;
```

Ceci devrait produire un résultat de ce style dans les journaux applicatifs :

```
  LOG:  duration: 3.651 ms  plan:
    Query Text: SELECT count(*)
                FROM pg_class, pg_index
               WHERE oid = indrelid AND indisunique;
    Aggregate  (cost=16.79..16.80 rows=1 width=0) (actual
 time=3.626..3.627 rows=1 loops=1)
      ->  Hash Join  (cost=4.17..16.55 rows=92 width=0) (actual
 time=3.349..3.594 rows=92 loops=1)
            Hash Cond: (pg_class.oid = pg_index.indrelid)
            ->  Seq Scan on pg_class  (cost=0.00..9.55 rows=255
 width=4) (actual time=0.016..0.140 rows=255 loops=1)
            ->  Hash  (cost=3.02..3.02 rows=92 width=4) (actual
 time=3.238..3.238 rows=92 loops=1)
                  Buckets: 1024  Batches: 1  Memory Usage: 4kB
                  ->  Seq Scan on pg_index  (cost=0.00..3.02
 rows=92 width=4) (actual time=0.008..3.187 rows=92 loops=1)
                        Filter: indisunique
```

F.4.3. Auteur

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>

F.5. basebackup_to_shell

`basebackup_to_shell` ajoute une méthode de sauvegarde personnalisée nommée `shell`. Elle permet de lancer `pg_basebackup --target=shell` ou, selon la configuration du module, `pg_basebackup --target=shell:`*`CHAINE_DETAILS`*, ce qui provoquera l'exécution sur le serveur, pour chaque archive tar générée par la sauvegarde, d'une commande choisie par l'administrateur. La commande recevra le contenu de l'archive sur son entrée standard.

Ce module est d'abord un exemple de la manière de créer de nouvelles cibles de sauvegarde par un module d'extension ; mais, dans certains scénarios, il peut avoir son utilité propre. Pour fonctionner, ce module doit être chargé via shared_preload_libraries ou local_preload_libraries.

F.5.1. Paramètres de configuration

`basebackup_to_shell.command` (`string`)

La commande à exécuter par le serveur, pour chaque archive générée par le processus de sauvegarde. Si `%f` apparaît dans la chaîne de la commande, il sera remplacé par le nom de l'archive (par exemple `base.tar`). Si `%d` apparaît, il sera remplacé par la cible précisée par l'utilisateur. Une cible est requise si `%d` apparaît dans la chaîne, et interdite sinon. Pour des raisons de sécurité, elle ne peut contenir que des caractères alphanumériques. Si `%%` apparaît dans la chaîne, il sera remplacé par un seul `%`. Une erreur est levée si un `%` apparaît suivi d'un autre caractère que ceux ci-dessus, ou en bout de chaîne.

`basebackup_to_shell.required_role` (`string`)

Le rôle nécessaire pour utiliser la cible `shell`. S'il n'est pas précisé, n'importe quel utilisateur de réplication peut utiliser cette cible de sauvegarde.

F.5.2. Auteur

Robert Haas <rhaas@postgresql.org>

F.6. basic_archive

`basic_archive` est un exemple de module d'archivage. Ce module copie les fichiers de segment WAL complets vers le répertoire spécifié. Ce n'est pas spécialement utile, mais peut servir de point de départ au développement de votre propre module d'archivage. Pour plus d'informations à propos des modules d'archivage, voir Chapitre 51.

Pour fonctionner, ce module doit être chargé via archive_library, et archive_mode doit être activé.

F.6.1. Paramètres de configuration

`basic_archive.archive_directory` (`string`)

Répertoire où le serveur devra copier les fichiers de segment WAL. Ce répertoire doit déjà exister. La valeur par défaut est une chaîne de caractères vide, ce qui stoppe l'archivage des WAL ; mais si archive_mode est activé, le serveur va se mettre à accumuler les fichiers en attendant qu'on lui fournisse une nouvelle valeur.

Les paramètres suivants doivent être configurés dans le fichier `postgresql.conf`. Une utilisation typique serait :

```
# postgresql.conf
archive_mode = 'on'
archive_library = 'basic_archive'
basic_archive.archive_directory = '/chemin/vers/repertoire_archive'
```

F.6.2. Notes

Des arrêts brutaux du serveur peuvent laisser des fichiers temporaires avec le préfixe `archtemp` dans le répertoire d'archivage. Il est recommandé de supprimer de tels fichiers avant de redémarrer le serveur après un arrêt brutal. Il est tout à fait possible de les supprimer pendant le fonctionnement du serveur, tant qu'ils ne sont plus liés à aucun processus d'archivage encore en cours, bien qu'il faille tout de même procéder avec prudence avec ce type d'opérations.

F.6.3. Auteur

Nathan Bossart

F.7. bloom

`bloom` fournit une méthode d'accès aux index basée sur les filtres Bloom[1].

Un filtre Bloom est une structure de données efficace en termes d'espace disque, utilisée pour tester si un élément fait partie d'un ensemble. Dans le cas d'une méthode d'accès aux index, il permet une exclusion rapide des lignes ne correspondant pas à la recherche via des signatures dont la taille est déterminée lors de la création de l'index.

Une signature est une représentation avec perte des attributs indexés et, de ce fait, est susceptible de renvoyer des faux positifs ; c'est-à-dire qu'il peut indiquer à tort qu'un élément fait partie d'un ensemble. De ce fait, les résultats d'une recherche doivent toujours être vérifiés en utilisant les valeurs réelles des attributs de la ligne dans la table. Des signatures plus larges réduisent les risques de faux positifs et réduisent donc le nombre de visites inutiles à la table. Bien sûr, l'index est plus volumineux et donc plus lent à parcourir.

Ce type d'index est principalement utile quand une table a de nombreux attributs et que les requêtes en testent des combinaisons arbitraires. Un index B-tree traditionnel est plus rapide qu'un index bloom mais il en faut généralement plusieurs pour supporter toutes les requêtes que gèrerait un seul index bloom. Notez que les index bloom ne supportent que les recherches par égalité, là où les index B-tree peuvent aussi réaliser des recherches d'inégalité et d'intervalles.

F.7.1. Paramètres

Un index `bloom` accepte les paramètres suivants dans sa clause `WITH` :

`length`

Longueur de chaque signature (enregistrement dans l'index) en bits, arrondi au multiple de `16` le plus proche. La valeur par défaut est de `80` et le maximum est `4096`.

`col1 -- col32`

Nombre de bits générés pour chaque colonne d'index. Le nom de chaque paramètre fait référence au numéro de la colonne d'index qu'il contrôle. La valeur par défaut est `2` bits et le maximum `4095`. Les paramètres pour les colonnes d'index non utilisées sont ignorés.

F.7.2. Exemples

Voici un exemple de création d'un index bloom :

[1] https://en.wikipedia.org/wiki/Bloom_filter

```
CREATE INDEX bloomidx ON tbloom USING bloom (i1,i2,i3)
       WITH (length=80, col1=2, col2=2, col3=4);
```

L'index est créé avec une longueur de signature de 80 bits, avec les attributs i1 et i2 correspondant à 2 bits, et l'attribut i3 à 4 bits. Nous pourrions avoir omis les informations `length`, `col1`, et `col2` car elles ont les valeurs par défaut.

Voici un exemple plus complet de définition et d'utilisation d'un index bloom, ainsi qu'une comparaison avec les index B-tree équivalents. L'index bloom est considérablement plus petit que l'index B-tree et offre de meilleures performances.

```
=# CREATE TABLE tbloom AS
   SELECT
     (random() * 1000000)::int as i1,
     (random() * 1000000)::int as i2,
     (random() * 1000000)::int as i3,
     (random() * 1000000)::int as i4,
     (random() * 1000000)::int as i5,
     (random() * 1000000)::int as i6
   FROM
  generate_series(1,10000000);
SELECT 10000000
```

Un parcours séquentiel sur cette grande table prend beaucoup de temps :

```
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 =
 123451;
                                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Seq Scan on tbloom  (cost=0.00..213744.00 rows=250 width=24)
 (actual time=357.059..357.059 rows=0 loops=1)
   Filter: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Filter: 10000000
 Planning Time: 0.346 ms
 Execution Time: 357.076 ms
(5 rows)
```

Même avec l'index B-tree défini, le résultat sera toujours un parcours séquentiel :

```
=# CREATE INDEX btreeidx ON tbloom (i1, i2, i3, i4, i5, i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('btreeidx'));
 pg_size_pretty
----------------
 386 MB
(1 row)
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 =
 123451;
                                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Seq Scan on tbloom  (cost=0.00..213744.00 rows=2 width=24) (actual
 time=351.016..351.017 rows=0 loops=1)
```

```
   Filter: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Filter: 10000000
 Planning time: 0.138 ms
 Execution Time: 351.035 ms
```

Avoir un index bloom défini sur la table est préférable à un B-tree pour gérer ce type de recherche :

```
=# CREATE INDEX bloomidx ON tbloom USING bloom (i1, i2, i3, i4, i5,
 i6);
CREATE INDEX
=# SELECT pg_size_pretty(pg_relation_size('bloomidx'));
 pg_size_pretty
----------------
 153 MB
(1 row)
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 =
 123451;
                                                          QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tbloom  (cost=1792.00..1799.69 rows=2
 width=24) (actual time=22.605..22.606 rows=0 loops=1)
   Recheck Cond: ((i2 = 898732) AND (i5 = 123451))
   Rows Removed by Index Recheck: 2300
   Heap Blocks: exact=2256
   ->  Bitmap Index Scan on bloomidx  (cost=0.00..178436.00 rows=1
 width=0) (actual time=20.005..20.005 rows=2300 loops=1)
         Index Cond: ((i2 = 898732) AND (i5 = 123451))
 Planning time: 0.099 ms
 Execution Time: 22.632 ms
(8 rows)
```

Le problème principal avec la recherche B-tree est que B-tree est inefficace quand les critères de recherche ne portent pas sur la ou les premières colonne(s) de l'index. Une meilleure stratégie avec les B-tree est de créer un index séparé pour chaque colonne. À ce moment-là, l'optimiseur pourra choisir quelque chose comme :

```
=# CREATE INDEX btreeidx1 ON tbloom (i1);
CREATE INDEX
=# CREATE INDEX btreeidx2 ON tbloom (i2);
CREATE INDEX
=# CREATE INDEX btreeidx3 ON tbloom (i3);
CREATE INDEX
=# CREATE INDEX btreeidx4 ON tbloom (i4);
CREATE INDEX
=# CREATE INDEX btreeidx5 ON tbloom (i5);
CREATE INDEX
=# CREATE INDEX btreeidx6 ON tbloom (i6);
CREATE INDEX
=# EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 =
 123451;
                                                          QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tbloom  (cost=9.29..13.30 rows=1 width=24)
 (actual time=0.032..0.033 rows=0 loops=1)
   Recheck Cond: ((i5 = 123451) AND (i2 = 898732))
```

```
   ->  BitmapAnd  (cost=9.29..9.29 rows=1 width=0) (actual
 time=0.047..0.047 rows=0 loops=1)
         ->  Bitmap Index Scan on btreeidx5  (cost=0.00..4.52
 rows=11 width=0) (actual time=0.026..0.026 rows=7 loops=1)
               Index Cond: (i5 = 123451)
         ->  Bitmap Index Scan on btreeidx2  (cost=0.00..4.52
 rows=11 width=0) (actual time=0.007..0.007 rows=8 loops=1)
               Index Cond: (i2 = 898732)
 Planning Time: 0.264 ms
 Execution Time: 0.047 ms
(9 rows)
```

Bien que cette requête soit bien plus rapide qu'avec n'importe lequel des index à une colonne, nous payons une pénalité en taille d'index. Chacun des index btree mono-colonne occupe 88,5 Mo, soit un espace total de plus de 531 Mo, autrement dit trois fois la place utilisée par l'index bloom.

F.7.3. Interface de la classe d'opérateur

Une classe d'opérateur pour les index bloom ne requiert qu'une fonction de hachage pour le type de données indexé et un opérateur d'égalité pour la recherche. Cet exemple définit la classe d'opérateur pour le type de données `text` :

```
CREATE OPERATOR CLASS text_ops
DEFAULT FOR TYPE text USING bloom AS
    OPERATOR    1   =(text, text),
    FUNCTION    1   hashtext(text);
```

F.7.4. Limitations

- Seules les classes d'opérateurs pour `int4` et `text` sont incluses avec le module.
- Seul l'opérateur `=` est supporté pour la recherche. Mais il est possible d'ajouter le support des tableaux avec les opérations union et intersection dans le futur.
- La méthode d'accès `bloom` ne permet pas d'avoir des index `UNIQUE`.
- La méthode d'accès `bloom` ne permet pas de rechercher des valeurs `NULL`.
- La méthode d'accès `bloom` n'accepte pas les index `UNIQUE`.
- La méthode d'accès `bloom` ne supporte pas la recherche des valeurs `NULL`.

F.7.5. Auteurs

Teodor Sigaev <teodor@postgrespro.ru>, Postgres Professional, Moscou, Russie

Alexander Korotkov <a.korotkov@postgrespro.ru>, Postgres Professional, Moscou, Russie

Oleg Bartunov <obartunov@postgrespro.ru>, Postgres Professional, Moscou, Russie

F.8. btree_gin

`btree_gin` fournit des échantillons de classes d'opérateurs GIN qui codent un comportement équivalent à un B-tree pour les types `int2`, `int4`, `int8`, `float4`, `float8`, `timestamp with time zone`, `timestamp without time zone`, `time with time zone`, `time without time zone`, `date`, `interval`, `oid`, `money`, `"char"`, `varchar`, `text`, `bytea`, `bit`, `varbit`, `macaddr`, `macaddr8`, `inet`, `cidr`, `uuid`, `name`, `bool`, `bpchar` et tous les types `enum`.

En général, ces classes d'opérateurs ne sont pas plus rapides que les méthodes standard d'indexation B-tree équivalentes, et il leur manque une fonctionnalité majeure du code B-tree standard : la capacité à forcer l'unicité. Toutefois, elles sont utiles pour tester GIN et comme base pour développer d'autres classes d'opérateurs GIN. Par ailleurs, pour des requêtes qui testent à la fois une colonne indexable via GIN et une colonne indexable par B-tree, il peut être plus efficace de créer un index GIN multi-colonnes qui utilise l'une de ces classes d'opérateurs que de créer deux index séparés qui devront être combinés par une opération de bitmap ET.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

F.8.1. Exemple d'utilisation

```
CREATE TABLE test (a int4);
-- crée l'index
CREATE INDEX testidx ON test USING GIN (a);
-- requête
SELECT * FROM test WHERE a < 10;
```

F.8.2. Auteurs

Teodor Sigaev (<teodor@stack.net>) et Oleg Bartunov (<oleg@sai.msu.su>). Voir http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gin pour plus d'informations.

F.9. btree_gist

`btree_gist` fournit des classes d'opérateur GiST qui codent un comportement équivalent à celui du B-tree pour les types de données `int2`, `int4`, `int8`, `float4`, `float8`, `numeric`, `timestamp with time zone`, `timestamp without time zone`, `time with time zone`, `time without time zone`, `date`, `interval`, `oid`, `money`, `char`, `varchar`, `text`, `bytea`, `bit`, `varbit`, `macaddr`, `macaddr8`, `inet`, `cidr`, `uuid`, `bool` et tous les types `enum`.

En règle générale, ces classes d'opérateur ne sont pas plus performantes que les méthodes d'indexage standard équivalentes du B-tree. Et il leur manque une fonctionnalité majeure : la possibilité d'assurer l'unicité. Néanmoins, elles fournissent d'autres fonctionnalités qui ne sont pas disponibles avec un index B-tree, comme décrit ci-dessous. De plus, ces classes d'opérateur sont utiles quand un index GiST multi-colonnes est nécessaire, quand certaines colonnes sont d'un type de données seulement indexable avec GiST. Enfin, ces classes d'opérateur sont utiles pour tester GiST et comme base de développement pour d'autres classes d'opérateur GiST.

En plus des opérateurs de recherche B-tree typiques, `btree_gist` fournit aussi un support pour `<>` (« non égale »). C'est utile en combinaison avec une contrainte d'exclusion, comme décrit ci-dessous.

De plus, pour les types de données disposant d'une métrique naturelle pour la distance, `btree_gist` définit un opérateur de distance, `<->`, et fournit un support par index GiST pour les recherches du type voisin-le-plus-proche en utilisant cet opérateur. Les opérateurs de distance sont fournis pour `int2`, `int4`, `int8`, `float4`, `float8`, `timestamp with time zone`, `timestamp without time zone`, `time without time zone`, `date`, `interval`, `oid` et `money`.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

F.9.1. Exemple d'utilisation

Exemple simple d'utilisation de `btree_gist` au lieu d'un index `btree` :

```
CREATE TABLE test (a int4);
-- création de l'index
CREATE INDEX testidx ON test USING GIST (a);
-- requête
SELECT * FROM test WHERE a < 10;
-- recherche du plus proche voisin: trouver les dix entrées les
 plus
-- proches de "42"
SELECT *, a &lt;-&gt; 42 AS dist FROM test ORDER BY a &lt;-&gt; 42
 LIMIT 10;
```

Utiliser une contrainte d'exclusion pour imposer qu'une cage dans un zoo ne contienne qu'un seul type d'animal :

```
=&gt; CREATE TABLE zoo (
  cage   INTEGER,
  animal TEXT,
  EXCLUDE USING GIST (cage WITH =, animal WITH &lt;&gt;)
);

=&gt; INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=&gt; INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=&gt; INSERT INTO zoo VALUES(123, 'lion');
ERROR:  conflicting key value violates exclusion constraint
 "zoo_cage_animal_excl"
DETAIL:  Key (cage, animal)=(123, lion) conflicts with existing key
 (cage, animal)=(123, zebra).
=&gt; INSERT INTO zoo VALUES(124, 'lion');
INSERT 0 1
```

F.9.2. Auteurs

Teodor Sigaev (<teodor@stack.net>), Oleg Bartunov (<oleg@sai.msu.su>), Janko Richter (<jankorichter@yahoo.de>) et Paul Jungwirth (<pj@illuminatedcomputing.com>). Voir le site sur GiST[2] pour plus d'informations.

F.10. citext

Le module `citext` fournit un type chaîne de caractères insensible à la casse, `citext`. En réalité, il appelle en interne la fonction `lower` lorsqu'il compare des valeurs. Dans les autres cas, il se comporte presque exactement comme le type `text`.

> **Astuce**
>
> Utilisez des *collations non déterministes* (voir Section 24.2.2.4) plutôt que ce module. Elles peuvent être utilisées pour les comparaisons insensibles à la casse, insensibles aux accents, ainsi que d'autres combinaisons, et elles gèrent aussi correctement un plus grand nombre de spécificités d'Unicode.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

[2] http://www.sai.msu.su/~megera/postgres/gist

F.10.1. Intérêt

L'approche standard pour effectuer des rapprochements insensibles à la casse avec PostgreSQL était d'utiliser la fonction `lower` pour comparer des valeurs. Par exemple :

```
SELECT * FROM tab WHERE lower(col) = LOWER(?);
```

Ceci fonctionne plutôt bien, mais présente quelques inconvénients :

- Cela rend les ordres SQL bavards, et vous devez sans arrêt vous souvenir d'utiliser la fonction `lower` à la fois sur la colonne et la valeur de la requête.
- Cela n'utilise pas les index, à moins que vous ne créiez un index fonctionnel avec la fonction `lower`.
- Si vous déclarez une colonne `UNIQUE` ou `PRIMARY KEY`, l'index généré implicitement est sensible à la casse. Il est donc inutile pour des recherches insensibles à la casse, et il ne va pas garantir l'unicité de manière insensible à la casse.

Le type de données `citext` vous permet d'éviter les appels à `lower` dans les requêtes SQL, et peut rendre une clé primaire insensible à la casse. `citext` tient compte de la locale, comme `text`, ce qui signifie que la comparaison entre caractères majuscules et minuscules dépend des règles de la locale paramétrée par `LC_CTYPE` de la base de données. Ici également, le comportement est identique à l'utilisation de la fonction `lower` dans les requêtes. Mais comme cela est fait de manière transparente par le type de données, vous n'avez pas à vous souvenir de faire quelque chose de particulier dans vos requêtes.

F.10.2. Comment l'utiliser

Voici un exemple simple d'utilisation :

```
CREATE TABLE users (
    nick CITEXT PRIMARY KEY,
    pass TEXT   NOT NULL
);

INSERT INTO users VALUES ( 'larry',
sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Tom',
sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Damian',
sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'NEAL',
sha256(random()::text::bytea) );
INSERT INTO users VALUES ( 'Bjørn',
sha256(random()::text::bytea) );

SELECT * FROM users WHERE nick = 'Larry';
```

L'ordre `SELECT` va renvoyer un enregistrement, bien que la colonne `nick` ait été positionnée à `larry` et que la requête soit pour `Larry`.

F.10.3. Comportement des comparaisons de chaînes

`citext` réalise des comparaisons en convertissant chaque chaîne en minuscule (comme si `lower` avait été appelé) puis en comparant normalement les résultats. Du coup, deux chaînes sont considérées égales si `lower` donne un résultat identique pour elles.

Afin d'émuler un tri insensible à la casse de la manière la plus fidèle possible, il existe des versions spécifiques à `citext` de plusieurs opérateurs et fonctions de traitement de chaînes. Ainsi, par exemple, les opérateurs pour les expressions rationnelles `~` et `~*` ont le même comportement quand ils sont appliqués au type `citext` : ils comparent tous les deux de manière insensible à la casse. Cela est aussi vrai pour `!~` et `!~*`, et également pour les opérateurs `LIKE`, `~~`, `~~*`, et `!~~` et `!~~*`. Si vous voulez faire une comparaison sensible à la casse, vous pouvez convertir dans un `text`.

De la même façon, toutes les fonctions ci-dessous font une comparaison insensible à la casse si leurs arguments sont de type `citext` :

- `regexp_match()`
- `regexp_matches()`
- `regexp_replace()`
- `regexp_split_to_array()`
- `regexp_split_to_table()`
- `replace()`
- `split_part()`
- `strpos()`
- `translate()`

Pour les fonctions regexp, si vous voulez effectuer des comparaisons sensibles à la casse, vous pouvez positionner l'indicateur « c » pour forcer une comparaison sensible à la casse. Sinon, si vous souhaitez un comportement sensible à la casse, vous devez convertir dans un type `text` avant d'utiliser une de ces fonctions.

F.10.4. Limitations

- Le comportement de `citext` sur la sensibilité à la casse dépend du paramètre `LC_CTYPE` de votre base de données. Par conséquent, la manière dont il compare les valeurs est fixée lorsque la base de données est créée. Il n'est pas réellement insensible à la casse dans les termes définis par le standard Unicode. En pratique, ce que cela signifie est que, tant que vous êtes satisfait de votre tri, vous devriez être satisfait des comparaisons de `citext`. Mais si vous avez des données stockées dans différentes langues dans votre base, des utilisateurs de certains langages pourraient trouver que les résultats de leurs requêtes sont inattendus si le tri est déterminé pour un autre langage.

- À partir de PostgreSQL, vous pouvez attacher une clause `COLLATE` aux colonnes `citext` ou à une valeur. Actuellement, les opérateurs `citext` honoreront une valeur `COLLATE` différente de la valeur par défaut lors de la comparaison de chaînes. Par contre, la mise en minuscule se fait toujours à partir de la configuration `LC_CTYPE` de la base de données (c'est-à-dire comme si `COLLATE "default"` avait été donné). Cela pourrait changer dans une prochaine version pour que les deux étapes suivent la clause `COLLATE`.

- `citext` n'est pas aussi performant que `text` parce que les fonctions opérateurs et les fonctions de comparaison B-tree doivent faire des copies des données et les convertir en minuscules pour les comparaisons. De plus, seul `text` accepte la déduplication dans un B-Tree. Néanmoins, `citext` est légèrement plus efficace qu'utiliser `lower` pour obtenir des comparaisons insensibles à la casse.

- `citext` n'aide pas réellement dans un certain contexte. Vos données doivent être comparées de manière sensible à la casse dans certains contextes, et de manière insensible à la casse dans d'autres contextes. La réponse habituelle à cette question est d'utiliser le type `text` et d'utiliser manuellement la fonction `lower` lorsque vous avez besoin d'une comparaison insensible à la casse ;

ceci fonctionne très bien si vous avez besoin peu fréquemment de comparaisons insensibles à la casse. Si vous avez besoin de comparaisons insensibles à la casse la plupart du temps, pensez à stocker les données en `citext` et à convertir explicitement les colonnes en `text` quand vous voulez une comparaison sensible à la casse. Dans les deux situations, vous aurez besoin de deux index si vous voulez que les deux types de recherche soient rapides.

- Le schéma contenant les opérateurs `citext` doit être dans le `search_path` (généralement `public`) ; dans le cas contraire, les opérateurs `text` sensibles à la casse seront appelés.

- L'approche consistant à mettre en minuscules les chaînes pour les comparer ne gère pas correctement certains cas spéciaux d'Unicode, par exemple quand une lettre majuscule a deux équivalents en lettres minuscules. Unicode distingue la *correspondance de casse* du *pli de casse* pour cette raison. Utilisez les collations non déterministes au lieu de `citext` pour gérer ça correctement.

F.10.5. Auteur

David E. Wheeler `<david@kineticode.com>`

Inspiré par le module original `citext` par Donald Fraser.

F.11. cube

Ce module code le type de données `cube` pour représenter des cubes à plusieurs dimensions.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

F.11.1. Syntaxe

Tableau F.2 affiche les représentations externes valides pour le type `cube`. *x*, *y*, etc. dénotent des nombres flottants.

Tableau F.2. Représentations externes d'un cube

Syntaxe externe	Signification
x	point uni-dimensionnel (ou intervalle unidimensionnel de longueur nulle)
(*x*)	Identique à ci-dessus
x1,*x2*,...,*xn*	Un point dans un espace à n dimensions, représenté en interne comme un cube de volume nul
(*x1*,*x2*,...,*xn*)	Identique à ci-dessus
(*x*),(*y*)	Intervalle uni-dimensionnel débutant à *x* et finissant à *y* ou vice-versa ; l'ordre n'importe pas
[(*x*),(*y*)]	Identique à ci-dessus
(*x1*,...,*xn*),(*y1*,...,*yn*)	Cube à n dimensions représenté par paires de coins diagonalement opposés
[(*x1*,...,*xn*),(*y1*,...,*yn*)]	Identique à ci-dessus

L'ordre de saisie des coins opposés d'un cube n'a aucune importance. Les fonctions `cube` s'occupent de la bascule nécessaire à l'obtention d'une représentation uniforme « bas gauche, haut droit ». Quand les coins coincident, le type `cube` enregistre un coin ainsi que le drapeau « is point » pour éviter de perdre de l'espace.

Les espaces sont ignorées, `[(x),(y)]` est donc identique à `[ ( x ), ( y ) ]`.

F.11.2. Précision

Les valeurs sont enregistrées en interne sous la forme de nombres en virgule flottante. Cela signifie que les nombres avec plus de 16 chiffres significatifs sont tronqués.

F.11.3. Utilisation

Tableau F.3 affiche les opérateurs spécialisés fournis par le type `cube`.

Tableau F.3. Opérateurs pour cube

Opérateur Description
`cube && cube` → `boolean` Est-ce que les cubes se superposent ?
`cube @> cube` → `boolean` Est-ce que le premier cube contient le second ?
`cube <@ cube` → `boolean` Est-ce que le premier cube est contenu dans le second ?
`cube -> integer` → `float8` Extrait les *n*-ième coordonnées du cube (à partir de 1)
`cube ~> integer` → `float8` Extrait les *n*-ième coordonnées du cube, en comptant ainsi : *n* = 2 * *k* - 1 indique la limite inférieure de la *k*-ième dimension, *n* = 2 * *k* indique la limite supérieure de la *k*-ième dimension. Un paramètre négatif *n* désigne la valeur inverse de la coordonnée positive correspondante. Cet opérateur est conçu pour le support KNN-GiST.
`cube <-> cube` → `float8` Calcule la distance Euclidienne entre deux cubes.
`cube <#> cube` → `float8` Calcule la distance taxicab (métrique L-1) entre deux cubes.
`cube <=> cube` → `float8` Calcule la distance Chebyshev (métrique L-inf) entre deux cubes.

En plus des opérateurs ci-dessus, les opérateurs de comparaison usuels indiqués dans Tableau 9.1 sont disponibles pour le type `cube`. Ces opérateurs comparent tout d'abord les premiers coordonnées et, si ces derniers sont égaux, comparent les deuxièmes coordonnées. Ils existent principalement pour supporter la classe d'opérateur d'index b-tree pour `cube`, qui peut seulement être utile par exemple si vous souhaitez une contrainte UNIQUE sur une colonne `cube`. Autrement, l'ordonnancement n'a pas d'usage concret.

Le module `cube` fournit aussi une classe d'opérateurs pour index GiST pour les valeurs `cube`. Un index GiST peut être utilisé sur le type `cube` pour chercher des valeurs en utilisant les opérateurs `=`, `&&`, `@>` et `<@` dans les clauses `WHERE`.

De plus, un index GiST `cube` peut être utilisé pour trouver les plus proches voisins en utilisant les opérateurs de métriques `<->`, `<#>` et `<=>` dans les clauses `ORDER BY`. Par exemple, le plus proche voisin du point 3-D (0.5, 0.5, 0.5) peut être trouvé de façon efficace avec :

```
SELECT c FROM test ORDER BY c <-> cube(array[0.5,0.5,0.5]) LIMIT 1;
```

L'opérateur ~> peut aussi être utilisé de cette façon pour récupérer efficacement les premières valeurs triées par une coordonnée sélectionnée. Par exemple, pour obtenir les quelques premiers cubes triés par la première coordonnée (coin bas gauche) ascendante, il est possible d'utiliser la requête suivante :

```
SELECT c FROM test ORDER BY c ~> 1 LIMIT 5;
```

Et pour obtenir des cubes 2-D triés par la première coordonnée du coin haut droit descendant :

```
SELECT c FROM test ORDER BY c ~> 3 DESC LIMIT 5;
```

Tableau F.4 indique les fonctions disponibles.

Tableau F.4. Fonctions cube

Fonction **Description** **Exemple(s)**
`cube ( float8 ) → cube` Crée un cube uni-dimensionnel de coordonnées identiques. `cube(1) → (1)`
`cube ( float8, float8 ) → cube` Crée un cube uni-dimensionnel. `cube(1, 2) → (1),(2)`
`cube ( float8[] ) → cube` Crée un cube de volume nul en utilisant les coordonnées définies par le tableau. `cube(ARRAY[1,2,3]) → (1, 2, 3)`
`cube ( float8[], float8[] ) → cube` Crée un cube avec les coordonnées haut droit et bas gauche définies par deux tableaux de flottants, obligatoirement de même taille. `cube(ARRAY[1,2], ARRAY[3,4]) → (1, 2),(3, 4)`
`cube ( cube, float8 ) → cube` Crée un nouveau cube en ajoutant une dimension à un cube existant, avec les mêmes valeurs pour les deux points finaux de la nouvelle coordonnée. Ceci est utile pour construire des cubes pièce par pièce à partir de valeurs calculées. `cube('(1,2),(3,4)'::cube, 5) → (1, 2, 5),(3, 4, 5)`
`cube ( cube, float8, float8 ) → cube` Crée un nouveau cube en ajoutant une dimension à un cube existant. Ceci est utile pour construire des cubes pièce par pièce à partir de valeurs calculées. `cube('(1,2),(3,4)'::cube, 5, 6) → (1, 2, 5),(3, 4, 6)`
`cube_dim ( cube ) → integer` Renvoie le nombre de dimensions du cube. `cube_dim('(1,2),(3,4)') → 2`
`cube_ll_coord ( cube, integer ) → float8` Renvoie la *n*-ième valeur de coordonnée pour le coin bas gauche du cube. `cube_ll_coord('(1,2),(3,4)', 2) → 2`

Fonction **Description** **Exemple(s)**
`cube_ur_coord` (`cube`, `integer`) → `float8` Renvoie la *n*-ième valeur de coordonnée pour le coin haut droite du cube. `cube_ur_coord('(1,2),(3,4)', 2)` → `4`
`cube_is_point` (`cube`) → `boolean` Renvoie true si le cube est un point, autrement dit si les deux coins de définition sont identiques. `cube_is_point(cube(1,1))` → `t`
`cube_distance` (`cube`, `cube`) → `float8` Renvoie la distance entre deux cubes. Si les deux cubes sont des points, il s'agit de la fonction de distance habituelle. `cube_distance('(1,2)', '(3,4)')` → `2.8284271247461903`
`cube_subset` (`cube`, `integer[]`) → `cube` Crée un nouveau cube à partir d'un cube existant, en utilisant une liste d'index de dimension à partir d'un tableau. Peut être utilisé pour extraire les points finaux d'une seule dimension ou pour supprimer les dimensions, ou pour les réordonner comme souhaité. `cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[2])` → `(3),(7)` `cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[3,2,1,1])` → `(5, 3, 1, 1),(8, 7, 6, 6)`
`cube_union` (`cube`, `cube`) → `cube` Produit l'union de deux cubes. `cube_union('(1,2)', '(3,4)')` → `(1, 2),(3, 4)`
`cube_inter` (`cube`, `cube`) → `cube` Produit l'intersection de deux cubes. `cube_inter('(1,2)', '(3,4)')` → `(3, 4),(1, 2)`
`cube_enlarge` (*c* `cube`, *r* `double`, *n* `integer`) → `cube` Augmente la taille du cube suivant le radius *r* spécifié sur au moins *n* dimensions. Si le radius est négatif, le cube est réduit. Toutes les dimensions définies sont modifiées par le radius *r*. Les coordonnées bas-gauche sont réduites de *r* et les coordonnées haut-droite sont augmentées de *r*. Si une coordonnée bas- gauche est augmentée suffisamment pour être plus importante que la coordonnée haute-droite (ceci peut seulement survenir quand $r < 0$), alors les deux coordonnées sont configurées avec leur moyenne. Si *n* est supérieur au nombre de dimensions définies et que le cube est grossi ($r > 0$), alors les dimensions supplémentaires sont ajoutées pour tout *n* ; 0 est utilisé comme valeur initiale pour les coordonnées supplémentaires. Cette fonction est utile pour créer les « bounding boxes » autour d'un point permettant de chercher les points les plus proches. `cube_enlarge('(1,2),(3,4)', 0.5, 3)` → `(0.5, 1.5, -0.5),(3.5, 4.5, 0.5)`

F.11.4. Par défaut

Le développeur pense que l'union :

```
select cube_union('(0,5,2),(2,3,1)', '0');
cube_union
-------------------
```

```
(0, 0, 0),(2, 5, 2)
(1 row)
```

n'est pas en contradiction avec le bon sens. Pas plus que l'intersection

```
select cube_inter('(0,-1),(1,1)', '(-2),(2)');
cube_inter
-------------
(0, 0),(1, 0)
(1 row)
```

Dans toutes les opérations binaires sur des boîtes de tailles différentes, l'auteur suppose que la plus petite est une projection cartésienne, c'est-à-dire qu'il y a des zéros à la place des coordonnées omises dans la représentation sous forme de chaîne. Les exemples ci-dessus sont équivalents à :

```
cube_union('(0,5,2),(2,3,1)','(0,0,0),(0,0,0)');
cube_inter('(0,-1),(1,1)','(-2,0),(2,0)');
```

Le prédicat de contenance suivant utilise la syntaxe en points alors qu'en fait, le second argument est représenté en interne par une boîte. Cette syntaxe rend inutile la définition du type point et des fonctions pour les prédicats (boîte,point).

```
select cube_contains('(0,0),(1,1)', '0.5,0.5');
cube_contains
--------------
t
(1 row)
```

F.11.5. Notes

Pour des exemples d'utilisation, voir les tests de régression `sql/cube.sql`.

Pour éviter toute mauvaise utilisation, le nombre de dimensions des cubes est limité à 100. Cela se configure dans `cubedata.h`.

F.11.6. Crédits

Auteur d'origine : Gene Selkov, Jr. `<selkovjr@mcs.anl.gov>`, Mathematics and Computer Science Division, Argonne National Laboratory.

F.11.7. Note de l'auteur

Mes remerciements vont tout particulièrement au professeur Joe Hellerstein (https://dsf.berkeley.edu/jmh/) qui a su extraire l'idée centrale de GiST (http://gist.cs.berkeley.edu/), et à son étudiant précédant, Andy Dong pour son exemple rédigé dans Illustra. Mes remerciements vont également aux développeurs de PostgreSQL qui m'ont permis de créer mon propre monde et de pouvoir y vivre sans être dérangé. Toute ma gratitude aussi à Argonne Lab et au département américain de l'énergie pour les années de support dans mes recherches sur les bases de données.

Des modifications mineures ont été effectuées sur ce module par Bruno Wolff III `<bruno@wolff.to>` en août/septembre 2002. Elles incluent la modification de la précision (de simple à double) et l'ajout de quelques nouvelles fonctions.

Des mises à jour supplémentaires ont été réalisées par Joshua Reich <josh@root.net> en juillet 2006. Elles concernent l'ajout de `cube(float8[], float8[])` et le nettoyage du code pour utiliser le protocole d'appel V1 à la place de la forme V0 maintenant obsolète.

F.12. dblink

`dblink` est un module qui permet de se connecter à d'autres bases de données PostgreSQL depuis une session de base de données.

Voir aussi postgres_fdw, qui fournit à peu près les mêmes fonctionnalités en utilisant une infrastructure plus moderne et respectant les standards.

dblink_connect

dblink_connect — ouvre une connexion persistante vers une base de données distante.

Synopsis

```
dblink_connect(text connstr) returns text
dblink_connect(text connname, text connstr) returns text
```

Description

`dblink_connect()` établit une connexion à une base de données PostgreSQL distante. Le serveur et la base de données à contacter sont identifiées par une chaine de connexion standard de la libpq. Il est possible d'affecter un nom à la connexion. Plusieurs connexions nommées peuvent être ouvertes en une seule fois, mais il ne peut y avoir qu'une seule connexion anonyme à la fois. Toute connexion est maintenue jusqu'à ce qu'elle soit close ou que la session de base de données soit terminée.

La chaîne de connexion peut aussi être le nom d'un serveur distant existant. Il est recommandé d'utiliser le foreign-data wrapper `dblink_fdw` lors de la définition du serveur distant. Voir l'exemple ci-dessous ainsi que CREATE SERVER et CREATE USER MAPPING.

Arguments

`connname`

Le nom à utiliser pour la connexion ; en cas d'omission, une connexion sans nom est ouverte, qui remplace toute autre connexion sans nom.

`connstr`

Chaîne de connexion au format standard de la libpq, par exemple `hostaddr=127.0.0.1 port=5432 dbname=mabase user=postgres password=monmotdepasse options=-csearch_path=`. Pour les détails, voir Section 34.1.1. Sinon, il est possible de donner le nom d'un serveur distant.

Valeur de retour

Renvoie le statut qui est toujours `OK` (puisque toute erreur amène la fonction à lever une erreur, sans retour).

Notes

Si des utilisateurs non dignes de confiance, ont accès à une base de données qui n'a pas adopté une méthode sécurisée d'usage des schémas, commencez chaque session en supprimant les schémas modifiables par tout le monde du paramètre `search_path`. Par exemple, un utilisateur pourrait ajouter `options=-csearch_path=` au *`connstr`*. Cette considération n'est pas spécifique à `dblink` ; elle s'applique à toute interface permettant d'exécuter des commandes SQL arbitraires.

Seuls les superutilisateurs peuvent utiliser `dblink_connect` pour créer des connexions authentifiées sans mot de passe. Si des utilisateurs standard ont ce besoin, il faut utiliser la fonction `dblink_connect_u` à sa place.

Il est déconseillé de choisir des noms de connexion contenant des signes d'égalité car ils peuvent introduire des risques de confusion avec les chaînes de connexion dans les autres fonctions `dblink`.

Exemple

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
  dblink_connect
 ----------------
  OK
 (1 row)

 SELECT dblink_connect('myconn', 'dbname=postgres options=-
csearch_path=');
  dblink_connect
 ----------------
  OK
 (1 row)

 -- Fonctionnalité FOREIGN DATA WRAPPER
 -- Note : la connexion locale nécessite l'authentification
 password pour que
 --        cela fonctionne correctement
 --        Sinon, vous recevrez le message d'erreur suivant
 provenant de
 --        dblink_connect() :
 --       ERROR:  password is required
 --       DETAIL:  Non-superuser cannot connect if the server does
 not request a password.
 --       HINT:  Target server's authentication method must be
 changed.
CREATE SERVER fdtest FOREIGN DATA WRAPPER dblink_fdw OPTIONS
 (hostaddr '127.0.0.1', dbname 'contrib_regression');

CREATE USER regress_dblink_user WITH PASSWORD 'secret';
CREATE USER MAPPING FOR regress_dblink_user SERVER fdtest OPTIONS
 (user 'regress_dblink_user', password 'secret');
GRANT USAGE ON FOREIGN SERVER fdtest TO regress_dblink_user;
GRANT SELECT ON TABLE foo TO regress_dblink_user;

\set ORIGINAL_USER :USER
\c - regress_dblink_user
SELECT dblink_connect('myconn', 'fdtest');
  dblink_connect
 ----------------
  OK
 (1 row)

SELECT * FROM dblink('myconn','SELECT * FROM foo') AS t(a int, b
 text, c text[]);
  a  | b |       c
 ----+---+---------------
   0 | a | {a0,b0,c0}
   1 | b | {a1,b1,c1}
   2 | c | {a2,b2,c2}
   3 | d | {a3,b3,c3}
   4 | e | {a4,b4,c4}
   5 | f | {a5,b5,c5}
   6 | g | {a6,b6,c6}
   7 | h | {a7,b7,c7}
   8 | i | {a8,b8,c8}
```

```
   9 | j | {a9,b9,c9}
  10 | k | {a10,b10,c10}
 (11 rows)

\c - :ORIGINAL_USER
REVOKE USAGE ON FOREIGN SERVER fdtest FROM regress_dblink_user;
REVOKE SELECT ON TABLE foo FROM regress_dblink_user;
DROP USER MAPPING FOR regress_dblink_user SERVER fdtest;
DROP USER regress_dblink_user;
DROP SERVER fdtest;
```

dblink_connect_u

dblink_connect_u — ouvre une connexion distante à une base de données de façon non sécurisée.

Synopsis

```
dblink_connect_u(text connstr) returns text
dblink_connect_u(text connname, text connstr) returns text
```

Description

`dblink_connect_u()` est identique à `dblink_connect()`, à ceci près qu'elle permet à des utilisateurs non-privilégiés de se connecter par toute méthode d'authentification.

Si le serveur distant sélectionne une méthode d'authentification qui n'implique pas de mot de passe, une impersonnalisation et une escalade de droits peut survenir car la session semble émaner de l'utilisateur qui exécute le serveur PostgreSQL local. De plus, même si le serveur distant réclame un mot de passe, il est possible de fournir le mot de passe à partir de l'environnement du serveur, par exemple en utilisant un fichier `~/.pgpass` appartenant à l'utilisateur du serveur. Cela apporte un risque supplémentaire d'impersonnification, sans parler de la possibilité d'exposer un mot de passe sur un serveur distant qui ne mérite pas votre confiance. C'est pourquoi, `dblink_connect_u()` est installé initialement sans aucun droit pour `PUBLIC`, ce qui restreint son utilisation aux seuls superutilisateurs. Dans certaines cas, le droit `EXECUTE` sur `dblink_connect_u()` peut être accordé à quelque utilisateur spécifique digne de confiance, mais cela doit se faire avec une extrême prudence. Il est aussi recommandé que tout fichier `~/.pgpass` appartenant à l'utilisateur du serveur ne contienne *pas* de joker dans le nom de l'hôte.

Pour plus de détails, voir `dblink_connect()`.

dblink_disconnect

dblink_disconnect — ferme une connexion persistante vers une base de données distante.

Synopsis

```
dblink_disconnect() returns text
dblink_disconnect(text connname) returns text
```

Description

`dblink_disconnect()` ferme une connexion ouverte par `dblink_connect()`. La forme sans argument ferme une connexion non nommée.

Arguments

connname

Le nom de la connexion à fermer

Valeur de retour

Renvoie le statut qui est toujours `OK` (puisque toute erreur amène la fonction à lever une erreur, sans retour).

Exemple

```
SELECT dblink_disconnect();
  dblink_disconnect
 -------------------
  OK
 (1 row)

 SELECT dblink_disconnect('myconn');
  dblink_disconnect
 -------------------
  OK
 (1 row)
```

dblink

dblink — exécute une requête sur une base de données distante

Synopsis

```
   dblink(text connname, text sql [, bool fail_on_error]) returns
setof record
   dblink(text connstr, text sql [, bool fail_on_error]) returns
setof record
   dblink(text sql [, bool fail_on_error]) returns setof record
```

Description

`dblink` exécute une requête (habituellement un `SELECT`, mais toute instruction SQL qui renvoie des lignes est valable) sur une base de données distante.

Si deux arguments `text` sont présents, le premier est d'abord considéré comme nom de connexion persistante ; si cette connexion est trouvée, la commande est exécutée sur cette connexion. Dans le cas contraire, le premier argument est considéré être une chaîne de connexion comme dans le cas de `dblink_connect`, et la connexion indiquée n'est conservée que pour la durée d'exécution de cette commande.

Arguments

`connname`

Le nom de la connexion à utiliser ; ce paramètre doit être omis pour utiliser une connexion sans nom.

`connstr`

Une chaîne de connexion similaire à celle décrite précédemment pour `dblink_connect`.

`sql`

L'instruction SQL à exécuter sur l'hôte distant, par exemple `select * from foo`.

`fail_on_error`

Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la fonction ne retourne aucune ligne.

Valeur de retour

La fonction renvoie les lignes produites par la requête. Comme `dblink` peut être utilisée avec toute requête, elle est déclarée comme renvoyant le type `record`, plutôt que de préciser un ensemble particulier de colonnes. Cela signifie que l'ensemble des colonnes attendues doit être précisé dans la requête appelante -- sinon PostgreSQL ne sait pas quoi attendre. Voici un exemple :

```
SELECT *
    FROM dblink('dbname=mydb options=-csearch_path=',
                'select proname, prosrc from pg_proc')
```

```
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

La partie « alias » de la clause `FROM` doit spécifier les noms et types des colonnes retournés par la fonction. (La précision des noms des colonnes dans un alias est une syntaxe du standard SQL mais la précision des types des colonnes est une extension PostgreSQL.) Cela permet au système de savoir comment étendre `*`, et à quoi correspond `proname` dans la clause `WHERE` avant de tenter l'exécution de la fonction. À l'exécution, une erreur est renvoyée si le nombre de colonnes du résultat effectif de la requête sur la base de données distante diffère de celui indiqué dans la clause `FROM`. Les noms de colonnes n'ont pas besoin de correspondre et `dblink` n'impose pas une correspondance exacte des types. L'opération réussit si les chaînes de données renvoyées sont valides pour le type déclaré dans la clause `FROM`.

Notes

Il est souvent plus pratique de créer une vue pour utiliser `dblink` avec des requêtes prédéterminées. Cela permet de laisser la vue gérer le type de la colonne plutôt que d'avoir à le saisir pour chaque requête. Par exemple :

```
CREATE VIEW myremote_pg_proc AS
  SELECT *
    FROM dblink('dbname=postgres options=-csearch_path=',
                'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text);

SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
```

Exemple

```
SELECT * FROM dblink('dbname=postgres options=-csearch_path=',
                     'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('dbname=postgres');
 dblink_connect
----------------
 OK
(1 row)
```

```
SELECT * FROM dblink('select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('myconn', 'dbname=regression options=-
csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'select proname, prosrc from
 pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 bytearecv  | bytearecv
 byteasend  | byteasend
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteain    | byteain
 byteaout   | byteaout
(14 rows)
```

dblink_exec

dblink_exec — exécute une commande sur une base de données distante

Synopsis

```
   dblink_exec(text connname, text sql [, bool fail_on_error])
returns text
   dblink_exec(text connstr, text sql [, bool fail_on_error])
returns text
   dblink_exec(text sql [, bool fail_on_error]) returns text
```

Description

`dblink_exec` exécute une commande (c'est-à-dire toute instruction SQL qui ne renvoie pas de lignes) dans une base de données distante.

Quand deux arguments de type `text` sont fournis, le premier est d'abord considéré comme nom d'une connexion persistante ; si cette connexion est trouvée, la commande est exécutée sur cette connexion. Dans le cas contraire, le premier argument est traitée comme une chaîne de connexion pour `dblink_connect`, et la connexion indiquée n'est maintenue que pour la durée d'exécution de cette commande.

Arguments

connname

Le nom de la connexion à utiliser ; ce paramètre doit être omis pour utiliser une connexion sans nom.

connstr

Une chaîne de connexion similaire à celle décrite précédemment pour `dblink_connect`.

sql

La commande SQL à exécuter sur la base de données distante ; par exemple `INSERT INTO foo VALUES(0, 'a', '{"a0","b0","c0"}')`.

fail_on_error

Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur locale. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la valeur de retour de la fonction est positionné à `ERROR`.

Valeur de retour

Renvoie le statut de la commande ou `ERROR` en cas d'échec.

Exemple

```
SELECT dblink_connect('dbname=dblink_test_standby');
 dblink_connect
----------------
```

```
 OK
(1 row)

SELECT dblink_exec('insert into foo values(21, ''z'',
 ''{"a0","b0","c0"}'');');
   dblink_exec
-----------------
 INSERT 943366 1
(1 row)

SELECT dblink_connect('myconn', 'dbname=regression');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('myconn', 'insert into foo values(21, ''z'',
 ''{"a0","b0","c0"}'');');
   dblink_exec
------------------
 INSERT 6432584 1
(1 row)

SELECT dblink_exec('myconn', 'insert into pg_class values
 (''foo'')',false);
NOTICE:  sql error
DETAIL:  ERROR:  null value in column "relnamespace" violates not-
null constraint

 dblink_exec
-------------
 ERROR
(1 row)
```

dblink_open

dblink_open — ouvre un curseur sur une base de données distante

Synopsis

```
   dblink_open(text cursorname, text sql [, bool fail_on_error])
returns text
   dblink_open(text connname, text cursorname, text sql [, bool
fail_on_error]) returns text
```

Description

`dblink_open()` ouvre un curseur sur une base de données distante. Le curseur peut ensuite être manipulé avec `dblink_fetch()` et `dblink_close()`.

Arguments

connname

Le nom de la connexion à utiliser ; ce paramètre doit être omis pour utiliser une connexion sans nom.

cursorname

Nom à affecter au curseur.

sql

L'instruction `SELECT` à exécuter sur l'hôte distant, par exemple `SELECT * FROM pg_class`.

fail_on_error

Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur locale. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la valeur de retour de la fonction est positionné à `ERROR`.

Valeur de retour

Renvoie le statut, soit `OK` soit `ERROR`.

Notes

Puisqu'un curseur ne peut persister qu'au sein d'une transaction, `dblink_open` lance un bloc de transaction explicite (`BEGIN`) côté distant, si le côté distant n'est pas déjà à l'intérieur d'une transaction. Cette transaction est refermée à l'exécution de l'instruction `dblink_close`. Si `dblink_exec` est utilisée pour modifier les données entre `dblink_open` et `dblink_close`, et qu'une erreur survient ou que `dblink_disconnect` est utilisé avant `dblink_close`, les modifications *sont perdues* car la transaction est annulée.

Exemple

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
```

```
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)
```

dblink_fetch

dblink_fetch — renvoie des lignes à partir d'un curseur ouvert sur une base de données distante

Synopsis

```
   dblink_fetch(text cursorname, int howmany [, bool
fail_on_error]) returns setof record
   dblink_fetch(text connname, text cursorname, int howmany [,
bool fail_on_error]) returns setof record
```

Description

`dblink_fetch` récupère des lignes à partir d'un curseur déjà ouvert par `dblink_open`.

Arguments

connname

Nom de la connexion à utiliser ; ce paramètre doit être omis pour utiliser une connexion sans nom.

cursorname

Le nom du curseur à partir duquel récupérer les lignes.

howmany

Nombre maximum de lignes à récupérer. Les *howmany* lignes suivantes sont récupérées, en commençant à la position actuelle du curseur, vers l'avant. Une fois le curseur arrivé à la fin, aucune ligne supplémentaire n'est renvoyée.

fail_on_error

Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur locale. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la fonction ne retourne aucune ligne.

Valeur de retour

La fonction renvoie les lignes récupérées à partir du curseur. Pour utiliser cette fonction, l'ensemble des colonnes attendues doit être spécifié, comme décrit précédemment pour `dblink`.

Notes

Si le nombre de colonnes de retour spécifiées dans la clause `FROM`, et le nombre réel de colonnes renvoyées par le curseur distant diffèrent, une erreur est remontée. Dans ce cas, le curseur distant est tout de même avancé du nombre de lignes indiqué, comme si l'erreur n'avait pas eu lieu. Il en est de même pour toute autre erreur survenant dans la requête locale après l'exécution du `FETCH` distant.

Exemple

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
```

```
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc
 where proname like ''bytea%''');
 dblink_open
-------------
 OK
(1 row)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source
 text);
 funcname |  source
----------+----------
 byteacat | byteacat
 byteacmp | byteacmp
 byteaeq  | byteaeq
 byteage  | byteage
 byteagt  | byteagt
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source
 text);
 funcname  |  source
-----------+-----------
 byteain   | byteain
 byteale   | byteale
 bytealike | bytealike
 bytealt   | bytealt
 byteane   | byteane
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source
 text);
  funcname  |   source
------------+------------
 byteanlike | byteanlike
 byteaout   | byteaout
(2 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source
 text);
 funcname | source
----------+--------
(0 rows)
```

dblink_close

dblink_close — ferme un curseur sur une base de données distante

Synopsis

```
   dblink_close(text cursorname [, bool fail_on_error]) returns
 text
   dblink_close(text connname, text cursorname [, bool
 fail_on_error]) returns text
```

Description

`dblink_close` ferme un curseur précédemment ouvert avec `dblink_open`.

Arguments

`connname`

Le nom de la connexion à utiliser ; ce paramètre doit être omis pour utiliser une connexion sans nom.

`cursorname`

Nom du curseur à fermer.

`fail_on_error`

Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la valeur de retour est positionnée à `ERROR`.

Valeur de retour

Renvoie le statut, soit `OK` soit `ERROR`.

Notes

Si `dblink_open` a ouvert un bloc de transaction explicite, et que c'est le dernier curseur ouvert restant dans cette connexion, `dblink_close` exécute le `COMMIT` correspondant.

Exemple

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
```

```
(1 row)

SELECT dblink_close('foo');
 dblink_close
--------------
 OK
(1 row)
```

dblink_get_connections

dblink_get_connections — renvoie les noms de toutes les connexions nommées ouvertes

Synopsis

```
dblink_get_connections() returns text[]
```

Description

`dblink_get_connections` renvoie un tableau contenant le nom de toutes les connexions nommées ouvertes de `dblink`.

Valeur de retour

Renvoie un tableau texte des noms des connexions, ou NULL s'il n'y en a pas.

Exemple

```
SELECT dblink_get_connections();
```

dblink_error_message

dblink_error_message — récupère le dernier message d'erreur sur la connexion nommée

Synopsis

```
dblink_error_message(text connname) returns text
```

Description

`dblink_error_message` récupère le dernier message d'erreur sur une connexion donnée.

Arguments

connname

Nom de la connexion à utiliser.

Return Value

Renvoie le dernier message, ou `OK` s'il n'y a pas eu d'erreur sur cette connexion.

Notes

Quand des requêtes asynchrones sont initiées par `dblink_send_query`, le message d'erreur associé avec la connexion pourrait ne pas être mis à jour tant que le message de réponse du serveur n'est pas consommé. Ceci signifie typiquement que `dblink_is_busy` ou `dblink_get_result` doivent être appelés avant `dblink_error_message`, pour que toute erreur générée par la requête asynchrone soit visible.

Exemple

```
SELECT dblink_error_message('dtest1');
```

dblink_send_query

dblink_send_query — envoie une requête asynchrone à une base de données distante

Synopsis

```
dblink_send_query(text connname, text sql) returns int
```

Description

`dblink_send_query` envoie une requête à exécuter de façon asynchrone, c'est-à-dire sans attendre immédiatement le résultat. Il ne doit pas déjà exister de requête asynchrone en exécution sur la connexion.

Après l'envoi réussi d'une requête asynchrone, le statut de fin d'exécution de la requête se vérifie avec `dblink_is_busy`, et les résultats sont finalement récupérés avec `dblink_get_result`. Il est aussi possible de tenter l'annulation d'une requête asynchrone active en utilisant `dblink_cancel_query`.

Arguments

`connname`

Le nom de la connexion à utiliser.

`sql`

L'instruction SQL à exécuter dans la base de données distante, par exemple `select * from pg_class`.

Valeur de retour

Renvoie 1 si la requête a été envoyée avec succès, 0 sinon.

Exemple

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1
< 3');
```

dblink_is_busy

dblink_is_busy — vérifie si la connexion est occupée par le traitement d'une requête asynchrone

Synopsis

```
dblink_is_busy(text connname) returns int
```

Description

`dblink_is_busy` teste si une requête asynchrone est en cours d'exécution.

Arguments

connname

Le nom de la connexion à vérifier.

Valeur de retour

Renvoie 1 si la connexion est occupée, 0 dans le cas contraire. Si cette fonction renvoie 0, il est garanti que l'appel à `dblink_get_result` ne bloque pas.

Exemple

```
SELECT dblink_is_busy('dtest1');
```

dblink_get_notify

dblink_get_notify — récupère les notifications asynchrones sur une connexion

Synopsis

```
   dblink_get_notify() returns setof (notify_name text, be_pid
int, extra text)
   dblink_get_notify(text connname) returns setof (notify_name
text, be_pid int, extra text)
```

Description

`dblink_get_notify` récupère les notifications soit sur une connexion anonyme (sans nom), soit sur une connexion nommée si le nom est précisé. Pour recevoir des notifications via dblink, `LISTEN` doit d'abord être lancé en utilisant `dblink_exec`. Pour les détails, voir LISTEN et NOTIFY.

Arguments

connname

Le nom d'une connexion nommée qui veut récupérer les notifications.

Valeur de retour

Renvoie `setof (notify_name text, be_pid int, extra text)` ou un ensemble vide.

Exemple

```
SELECT dblink_exec('LISTEN virtual');
 dblink_exec
-------------
 LISTEN
(1 row)

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
(0 rows)

NOTIFY virtual;
NOTIFY

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
 virtual     |   1229 |
(1 row)
```

dblink_get_result

dblink_get_result — récupère le résultat d'une requête asynchrone

Synopsis

```
    dblink_get_result(text connname [, bool fail_on_error]) returns
 setof record
```

Description

`dblink_get_result` récupère le résultat d'une requête asynchrone précédemment envoyée avec `dblink_send_query`. Si la requête n'est pas terminée, `dblink_get_result` en attend la fin.

Arguments

`connname`

Le nom de la connexion à utiliser.

`fail_on_error`

Si true (valeur par défaut en cas d'omission), une erreur distante est reportée localement comme une erreur locale. Dans le cas contraire, un message d'erreur distant est traité localement comme un message de type NOTICE, et la fonction ne retourne aucune ligne.

Valeur de retour

Pour une requête asynchrone (c'est-à-dire une instruction SQL renvoyant des lignes), la fonction renvoie les lignes produites par la requête. Pour utiliser cette fonction, il faut spécifier l'ensemble des colonnes attendues, comme indiqué pour `dblink`.

Pour une commande asynchrone (c'est-à-dire une instruction SQL ne renvoyant aucune ligne), la fonction renvoie une seule ligne avec une colonne texte contenant la chaîne de statut de la commande. Il est impératif d'indiquer dans la clause `FROM` appelante que le résultat est constitué d'une unique colonne texte .

Notes

Cette fonction *doit* être appelée si `dblink_send_query` a renvoyé 1. Elle doit l'être une fois pour chaque requête envoyée, et une fois de plus pour obtenir un ensemble vide, avant de pouvoir utiliser à nouveau la connexion.

Lorsqu'on utilise `dblink_send_query` et `dblink_get_result`, dblink récupère l'intégralité de la requête avant de les renvoyer au système local. Si la requête renvoie un grand nombre de lignes, cela peut conduire à une surcharge temporaire de la mémoire dans la session locale. Il peut être préférable d'ouvrir un curseur avec `dblink_open` puis de récupérer un nombre gérable de lignes. Sinon, vous pouvez utiliser un simple `dblink()`, qui évite la surcharge de la mémoire en mettant en attente de gros ensembles de résultats sur disque.

Exemple

```
contrib_regression=# SELECT dblink_connect('dtest1',
 'dbname=contrib_regression');
 dblink_connect
----------------
```

```
 OK
(1 row)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo
 where f1 < 3') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS
 t1(f1 int, f2 text, f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS
 t1(f1 int, f2 text, f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo
 where f1 < 3; select * from foo where f1 > 6') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS
 t1(f1 int, f2 text, f3 text[]);
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS
 t1(f1 int, f2 text, f3 text[]);
 f1 | f2 |      f3
----+----+---------------
  7 | h  | {a7,b7,c7}
  8 | i  | {a8,b8,c8}
  9 | j  | {a9,b9,c9}
 10 | k  | {a10,b10,c10}
(4 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS
 t1(f1 int, f2 text, f3 text[]);
 f1 | f2 | f3
----+----+----
(0 rows)
```

dblink_cancel_query

dblink_cancel_query — annule toute requête en cours d'exécution sur la connexion nommée

Synopsis

```
dblink_cancel_query(text connname) returns text
```

Description

`dblink_cancel_query` tente d'annuler toute requête en cours d'exécution sur la connexion nommée. La réussite de la fonction n'est pas assurée (la requête distante pourrait, par exemple, être déjà terminée). Une demande d'annulation augmente simplement la probabilité que la requête échoue rapidement. Le protocole de requête normal doit toujours être terminé, par exemple en appelant `dblink_get_result`.

Arguments

connname

Le nom de la connexion à utiliser.

Valeur de retour

Renvoie `OK` si la demande d'annulation a été envoyée, ou le texte d'un message d'erreur en cas d'échec.

Exemple

```
SELECT dblink_cancel_query('dtest1');
```

dblink_get_pkey

dblink_get_pkey — renvoie la position et le nom des champs de clé primaire d'une relation

Synopsis

```
dblink_get_pkey(text relname) returns setof dblink_pkey_results
```

Description

`dblink_get_pkey` fournit des informations sur la clé primaire d'une relation de la base de données locale. Il est parfois utile de produire des requêtes à transmettre à des bases distantes.

Arguments

`relname`

Le nom d'une relation locale, par exemple `foo` ou `monschema.matable`. Ajouter des guillemets doubles si le nom a une casse mixte, ou contient des caractères spéciaux, par exemple `"FooBar"` ; sans guillemets, la chaîne est forcée en minuscule.

Valeur de retour

Renvoie une ligne pour chaque champ de clé primaire, ou aucune ligne si la relation n'a pas de clé primaire. Le type de ligne résultante est défini ainsi :

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

La colonne `position` comme à 1 et va jusqu'à *`N`* ; elle correspond au numéro du champ dans la clé primaire, pas au numéro de colonne dans la liste des colonnes de la table.

Exemple

```
CREATE TABLE foobar (
    f1 int,
    f2 int,
    f3 int,
    PRIMARY KEY (f1, f2, f3)
);
CREATE TABLE

SELECT * FROM dblink_get_pkey('foobar');
 position | colname
----------+---------
        1 | f1
        2 | f2
        3 | f3
(3 rows)
```

dblink_build_sql_insert

dblink_build_sql_insert — construit une instruction d'insertion en utilisant un tuple local, remplaçant les valeurs des champs de la clé primaire avec les valeurs fournies

Synopsis

```
    dblink_build_sql_insert(text relname,
                            int2vector primary_key_attnums,
                            integer num_primary_key_atts,
                            text[] src_pk_att_vals_array,
                            text[] tgt_pk_att_vals_array) returns
 text
```

Description

`dblink_build_sql_insert` peut être utile pour réaliser une réplication sélective d'une table locale vers une base distante. Elle sélectionne une ligne de la table locale sur la base de la clé primaire et construit une commande SQL `INSERT` qui duplique cette ligne, mais avec pour valeurs de clé primaire celles du dernier argument. (Pour réaliser une copie exacte de la ligne, il suffit d'indiquer les mêmes valeurs pour les deux derniers arguments.)

Arguments

`relname`

Le nom d'une relation locale, par exemple `foo` ou `monschema.matable`. Ajouter des guillemets doubles si le nom est en casse mixte ou contient des caractères spéciaux, par exemple `"FooBar"` ; sans guillemets, la chaîne est forcée en minuscule.

`primary_key_attnums`

Les numéros des attributs (commençant à 1) des champs de la clé primaire, par exemple `1 2`.

`num_primary_key_atts`

Le nombre de champs de la clé primaire.

`src_pk_att_vals_array`

Les valeurs des champs de la clé primaire à utiliser pour identifier le tuple local. Chaque champ est représenté dans sa forme textuelle. Une erreur est renvoyée s'il n'y a pas de lignes locales avec ces valeurs de clé primaire.

`tgt_pk_att_vals_array`

Les valeurs des champs de la clé primaire à placer dans la commande `INSERT` résultante. Chaque champ est représenté dans sa forme textuelle.

Valeur de retour

Renvoie l'instruction SQL demandée en tant que texte.

Notes

À partir de PostgreSQL 9.0, les numéros des attributs dans *`primary_key_attnums`* sont interprétés comme des numéros logiques de colonnes correspondant à la position de la colonne dans

`SELECT * FROM relation`. Les versions précédentes interprétaient les numéros comme des positions physiques de colonnes. Une différence existe si une des colonnes à gauche de la colonne indiquée a été supprimé de la table.

Exemple

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}',
 '{"1", "b''a"}');
            dblink_build_sql_insert
--------------------------------------------------
 INSERT INTO foo(f1,f2,f3) VALUES('1','b''a','1')
(1 row)
```

dblink_build_sql_delete

dblink_build_sql_delete — construit une instruction de suppression en utilisant les valeurs fournies pour les champs de la clé primaire

Synopsis

```
    dblink_build_sql_delete(text relname,
                            int2vector primary_key_attnums,
                            integer num_primary_key_atts,
                            text[] tgt_pk_att_vals_array) returns
 text
```

Description

`dblink_build_sql_delete` peut être utile pour réaliser une réplication sélective d'une table locale vers une base distante. Elle construit une commande SQL `DELETE` qui supprime la ligne avec les valeurs indiquées de clé primaire.

Arguments

`relname`

Le nom d'une relation locale, par exemple `foo` ou `monschema.matable`. Ajouter des guillemets doubles si le nom est en casse mixte ou contient des caractères spéciaux, par exemple `"FooBar"` ; sans guillemets, la chaîne est forcée en minuscule.

`primary_key_attnums`

Les numéros des attributs (commençant à 1) des champs de la clé primaire, par exemple `1 2`.

`num_primary_key_atts`

Le nombre de champs de la clé primaire.

`tgt_pk_att_vals_array`

Les valeurs de champs de la clé primaire, à utiliser dans la commande `DELETE` résultante. Chaque champ est représenté dans sa forme textuelle.

Valeur de retour

Renvoie l'instruction SQL demandée en tant que texte.

Notes

À partir de PostgreSQL 9.0, les numéros des attributs dans *`primary_key_attnums`* sont interprétés comme des numéros logiques de colonnes correspondant à la position de la colonne dans `SELECT * FROM relation`. Les versions précédentes interprétaient les numéros comme des positions physiques de colonnes. Une différence existe si une des colonnes à gauche de la colonne indiquée a été supprimé de la table.

Exemple

```
SELECT dblink_build_sql_delete('"MyFoo"', '1 2', 2, '{"1", "b"}');
           dblink_build_sql_delete
---------------------------------------------
 DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'
(1 row)
```

dblink_build_sql_update

dblink_build_sql_update — construit une instruction de mise à jour à partir d'un tuple local, en remplaçant les valeurs des champs de la clé primaire par celles fournies

Synopsis

```
    dblink_build_sql_update(text relname,
                            int2vector primary_key_attnums,
                            integer num_primary_key_atts,
                            text[] src_pk_att_vals_array,
                            text[] tgt_pk_att_vals_array) returns
 text
```

Description

`dblink_build_sql_update` peut être utile pour réaliser une réplication sélective d'une table locale vers une base de données distante. Elle sélectionne une ligne à partir de la table locale en se basant sur la clé primaire, puis construit une commande SQL `UPDATE` qui duplique cette ligne, mais avec pour valeurs de clé primaire celles du dernier argument. (Pour faire une copie exacte de la ligne, on indique les mêmes valeurs pour les deux derniers arguments.) La commande `UPDATE` affecte toujours tous les champs de la ligne -- la différence principale entre cette instruction et `dblink_build_sql_insert` est l'hypothèse de l'existence de la ligne cible dans la table distante.

Arguments

`relname`

Le nom d'une relation locale, par exemple `foo` ou `monschema.matable`. Ajouter des guillemets doubles si le nom est en casse mixte ou contient des caractères spéciaux, par exemple `"FooBar"` ; sans guillemets, la chaîne est forcée en minuscule.

`primary_key_attnums`

Les numéros des attributs (commençant à 1) des champs de la clé primaire, par exemple `1 2`.

`num_primary_key_atts`

Le nombre de champs de la clé primaire.

`src_pk_att_vals_array`

Les valeurs des champs de la clé primaire à utiliser pour identifier le tuple local. Chaque champ est représenté dans sa forme textuelle. Une erreur est renvoyée s'il n'y a pas de lignes locales avec ces valeurs de clé primaire.

`tgt_pk_att_vals_array`

Les valeurs des champs de la clé primaire à placer dans la commande `UPDATE` résultante. Chaque champ est représenté dans sa forme textuelle.

Valeur de retour

Renvoie l'instruction SQL demandée en tant que texte.

Notes

À partir de PostgreSQL 9.0, les numéros des attributs dans *primary_key_attnums* sont interprétés comme des numéros logiques de colonnes correspondant à la position de la colonne dans SELECT * FROM relation. Les versions précédentes interprétaient les numéros comme des positions physiques de colonnes. Une différence existe si une des colonnes à gauche de la colonne indiquée a été supprimé de la table.

Exemple

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{"1", "a"}',
 '{"1", "b"}');
                   dblink_build_sql_update
-------------------------------------------------------------
 UPDATE foo SET f1='1',f2='b',f3='1' WHERE f1='1' AND f2='b'
(1 row)
```

F.13. dict_int

dict_int est un exemple de modèle de dictionnaire pour la recherche plein texte. La création de ce dictionnaire a été motivée par la volonté de pouvoir contrôler l'indexage d'entiers (signés et non signés), pour permettre à de tels nombres d'être indexés sans grossissement excessif du nombre de mots uniques, ce qui affecte grandement la performance de la recherche.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

F.13.1. Configuration

Le dictionnaire accepte trois options :

- le paramètre maxlen indique le nombre maximum de chiffres autorisés dans un mot de type entier. La valeur par défaut est 6 ;
- Le paramètre rejectlong précise si un entier trop long doit être tronqué ou ignoré. Si rejectlong vaut false (valeur par défaut), le dictionnaire renvoie les maxlen premiers chiffres de l'entier. Si rejectlong vaut true, le dictionnaire traite l'entier comme un terme courant, l'entier n'est donc pas indexé. Cela signifie aussi qu'un tel nombre ne peut pas être recherché.
- Le paramètre absval spécifie si les signes « + » ou « - » au début doivent être supprimés des entiers. Par défaut, la valeur est false. Quand la valeur vaut true, le signe est supprimé avant que maxlen soit appliquée.

F.13.2. Utilisation

Installer l'extension dict_int crée un modèle de recherche plein texte intdict_template et un dictionnaire intdict basé sur ce dernier avec les paramètres par défaut. Les paramètres peuvent être modifiés, par exemple :

```
mabase# ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4,
 REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

ou créez de nouveaux dictionnaires à partir du modèle.

Pour tester le dictionnaire :

```
mydb# select ts_lexize('intdict', '12345678');
 ts_lexize
-----------
 {123456}
```

mais une utilisation réelle nécessite de l'inclure dans une configuration de recherche plein texte comme celle décrite dans Chapitre 12. Cela peut ressembler à ceci :

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR int, uint WITH intdict;
```

F.14. dict_xsyn

Le module `dict_xsyn` (*Extended Synonym Dictionary*, dictionnaire étendu de synonymes) est un exemple de modèle de dictionnaire pour la recherche plein texte. Ce type de dictionnaire remplace des mots avec un ensemble de synonymes, ce qui rend possible la recherche d'un mot en utilisant un de ses synonymes.

F.14.1. Configuration

Un dictionnaire `dict_xsyn` accepte les options suivantes :

- `matchorig` contrôle si le mot original est accepté par le dictionnaire. Par défaut à `true`.
- `matchsynonyms` contrôle si les synonymes sont acceptés par le dictionnaire. Par défaut à `false`.
- `keeporig` contrôle si le mot original est inclus dans la sortie du dictionnaire. Par défaut à `true`.
- `keepsynonyms` contrôle si les synonymes sont inclus dans la sortie du dictionnaire. Par défaut à `true`.
- `rules` est le nom du fichier contenant la liste des synonymes. Ce fichier doit être stocké dans `$SHAREDIR/tsearch_data/` (où `$SHAREDIR` est le répertoire des données partagées de la distribution PostgreSQL). Son nom doit se terminer par `.rules` (cette extension n'est pas à inclure dans le paramètre `rules`).

Le fichier rules a le format suivant :

- chaque ligne représente un groupe de synonymes pour un mot simple, donné en premier sur la ligne. Les synonymes sont séparés par un espace :

  ```
  mot syn1 syn2 syn3
  ```

- le signe dièse (#) est un délimiteur de commentaires. Il peut apparaître n'importe où dans la ligne. Le reste de la ligne sera ignoré.

Un exemple est donné dans `xsyn_sample.rules` qui est installé dans `$SHAREDIR/tsearch_data/`.

F.14.2. Utilisation

Installer l'extension `dict_xsyn` crée un modèle `xsyn_template` de recherche plein texte et un dictionnaire `xsyn` basé sur le modèle, avec des paramètres par défaut. Il est possible de modifier les paramètres, par exemple :

```
ma_base# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules',
 KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

ou de créer de nouveaux dictionnaires basés sur le modèle.

Pour tester le dictionnaire :

```
ma_base=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

ma_base# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules',
 KEEPORIG=true);
ALTER TEXT SEARCH DICTIONARY

ma_base=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {word,syn1,syn2,syn3}

ma_base# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules',
 KEEPORIG=false, MATCHSYNONYMS=true);
ALTER TEXT SEARCH DICTIONARY

ma_base=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

ma_base# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules',
 KEEPORIG=true, MATCHORIG=false, KEEPSYNONYMS=false);
ALTER TEXT SEARCH DICTIONARY

ma_base=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {word}
```

Une utilisation réelle implique son ajout dans une configuration de recherche plein texte comme décrit dans Chapitre 12. Cela pourrait ressembler à ceci :

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR word, asciiword WITH xsyn, english_stem;
```

F.15. earthdistance

Le module `earthdistance` fournit deux approches différentes pour calculer de grandes distances circulaires à la surface de la Terre. La première dépend du module `cube`. La seconde est basée sur le type de données interne `point` et utilise longitude et latitude pour les coordonnées.

Dans ce module, la Terre est supposée parfaitement sphérique (si cette hypothèse n'est pas acceptable, le projet PostGIS[1] doit être considéré.)

Le module `cube` doit être installé avant que `earthdistance` ne puisse l'être (bien que vous pouvez utiliser la clause `CASCADE` de `CREATE EXTENSION` pour installer les deux en une seule commande).

> **Attention**
>
> Il est fortement recommandé que `earthdistance` et `cube` soient installés dans le même schéma et que ce schéma n'ait pas de droit CREATE donné à un utilisateur auquel on ne ferait pas confiance. Sinon, il existe un risque au moment de l'installation pour que le schéma de `earthdistance` contienne des objets définis par un utilisateur hostile. De plus, lors de l'utilisation des fonctions de `earthdistance` après l'installation, le chemin de recherche entier devrait contenir seulement les schémas de confiance.

F.15.1. Distances sur Terre à partir de cubes

Les données sont stockées dans des cubes qui sont des points (les coins sont identiques), les trois coordonnées représentant la distance x, y et z au centre de la Terre. Un *domaine* `earth` sur le type `cube` est fourni. Il inclut des contraintes de vérification pour que la valeur respecte ces restrictions et reste raisonnablement proche de la surface réelle de la Terre.

Le rayon de la Terre, obtenu à partir de la fonction `earth()`, est donné en mètres. Il est toutefois possible de modifier le module pour changer l'unité, ou pour utiliser une autre valeur de rayon.

Ce paquet peut être appliqué aux bases de données d'astronomie. Les astronomes peuvent modifier `earth()` pour que le rayon renvoyé soit `180/pi()`, de sorte que les distances soient en degrés.

Les fonctions acceptent latitude et longitude en entrée et en sortie (en degrés), calculent la distance circulaire entre deux points et permettent de préciser facilement une boîte utilisable par les recherches par index.

Les fonctions fournies sont montrées dans Tableau F.5.

Tableau F.5. Fonctions earthdistance par cubes

Fonction Description
`earth` () → `float8` Renvoie le rayon estimé de la Terre.
`sec_to_gc` (`float8`) → `float8` Convertit la distance en ligne droite (sécant) entre deux points à la surface de la Terre en distance circulaire.
`gc_to_sec` (`float8`) → `float8`

[1] https://postgis.net/

Fonction Description
Convertit la distance circulaire entre deux points à la surface de la Terre en une distance en ligne droite (sécant).
`ll_to_earth` (`float8`, `float8`) → `earth` Renvoie l'emplacement d'un point à la surface de la Terre étant données sa latitude (argument 1) et sa longitude (argument 2) en degrés.
`latitude` (`earth`) → `float8` Renvoie la latitude en degrés d'un point à la surface de la Terre.
`longitude` (`earth`) → `float8` Renvoie la longitude en degrés d'un point à la surface de la Terre.
`earth_distance` (`earth`, `earth`) → `float8` Renvoie la distance circulaire entre deux points à la surface de la Terre.
`earth_box` (`earth`, `float8`) → `cube` Renvoie une boîte autorisant une recherche par index avec l'opérateur `@>` du type cube pour les points situés au maximum à une distance circulaire donnée d'un emplacement. Certains points de cette boîte sont plus éloignés que la distance circulaire indiquée. Une deuxième vérification utilisant `earth_distance` doit, donc, être incluse dans la requête.

F.15.2. Distances sur Terre à partir de points

La seconde partie du module se fonde sur la représentation des emplacements sur Terre comme valeurs de type `point`, pour lesquelles le premier composant représente la longitude en degrés, et le second la latitude en degrés. Les points ont la forme (longitude, latitude) et non l'inverse, car intuitivement, la longitude se compare à l'axe X, la latitude à l'axe Y.

Un opérateur unique est fourni, il est indiqué dans Tableau F.6.

Tableau F.6. Opérateurs earthdistance par points

Opérateur Description
`point` `<@>` `point` → `float8` Donne la distance en miles entre deux points à la surface de la Terre.

Contrairement à la partie fondée sur `cube`, les unités ne sont pas modifiables : une modification de la fonction `earth()` n'affecte pas les résultats de l'opérateur.

La représentation longitude/latitude a pour inconvénient d'obliger à tenir compte des conditions particulières près des pôles et près des longitudes de +/- 180 degrés. La représentation par `cube` évite ces discontinuités.

F.16. file_fdw

Le module `file_fdw` fournit le wrapper de données distantes `file_fdw`, qui peut être utilisé pour accéder à des fichiers de données situées sur le système de fichiers du serveur, ou pour exécuter des programmes sur le serveur et lire leur sortie. Les fichiers de données ou program output doivent être dans un format qui puisse être lu par `COPY FROM`; voyez COPY pour les détails. L'accès à ce type de fichier se fait uniquement en lecture seule.

Une table distante créée en utilisant ce wrapper peut avoir les options suivantes:

`filename`

Spécifie le fichier devant être lu. Les chemins relatifs sont relatifs au répertoire principal des données. `filename` ou `program` doit être spécifié, mais pas les deux en même temps.

`program`

Spécifie la commande à exécuter. La sortie standard de cette commande sera lue comme si `COPY FROM PROGRAM` était utilisé. Il est nécessaire d'indiquer soit `program` soit `filename` mais pas les deux.

`force_null`

C'est une option booléenne. Si elle vaut vrai, cela signifie que les valeurs de la colonne qui correspondent à la chaîne NULL sont retournées comme `NULL` même si la valeur est entourée de guillemets. Sans cette option, seules les valeurs non entourées de guillemets qui correspondent à la chaîne NULL seront retournées comme `NULL`. Cela a le même effet que de spécifier les colonnes dans l'option `FORCE_NULL` de la commande `COPY`.

`format`

Spécifie le format des données, comme dans l'option `FORMAT` de la commande `COPY`.

`header`

Spécifie si les données ont une ligne d'entête, comme l'option `HEADER` de la commande `COPY`.

`delimiter`

Spécifie le caractère délimiteur des données, comme l'option `DELIMITER` de la commande `COPY`.

`quote`

Spécifie le caractère guillemet, comme l'option `QUOTE` de la commande `COPY`.

`escape`

Spécifie le caractère d'échappement des données, comme l'option `ESCAPE` de la commande `COPY`.

`null`

Spécifie la chaîne null des données, comme l'option `NULL` de la commande `COPY`.

`encoding`

Spécifie l'encodage des données, comme l'option `ENCODING` de la commande `COPY`.

Notez que, bien que `COPY` autorise la spécification d'options comme `HEADER` sans valeur correspondante, la syntaxe des options de la table externe requiert la présence d'une valeur dans tous les cas. Pour activer les options de `COPY` sans valeur, vous pouvez donner la valeur TRUE à la place, since all such options are Booleans.

Une colonne d'une table distante créée en utilisant ce wrapper peut avoir les options suivantes :

`force_not_null`

C'est une option booléenne. Si elle vaut true, cela signifie que les valeurs de la colonne ne doivent pas être comparées à celle de la chaîne NULL (autrement dit, l'option `null` au niveau de la table). Ceci a le même effet que de lister la colonne dans l'option `FORCE_NOT_NULL` de `COPY`.

L'option `FORCE_QUOTE` de `COPY` n'est pas supportée par `file_fdw` pour le moment.

Ces options ne peuvent être spécifiées que pour une table distante ou ses colonnes, pas comme options du wrapper de données distantes `file_fdw`, pas plus que comme des options d'un serveur ou d'un mapping d'utilisateur utilisant le wrapper.

Changer les options au niveau des tables nécessite l'attribut SUPERUSER ou avoir les droits du rôle `pg_read_server_files` (pour utiliser un fichier) ou du rôle `pg_execute_server_program` (pour utiliser un programme), pour des raisons de sécurité : seuls certains utilisateurs devraient pouvoir contrôler quel fichier est lu ou quel programme est exécuté. En principe, des utilisateurs standards devraient pouvoir modifier les autres options, mais ceci n'est pas supporté pour le moment.

Lorsque l'option `program` est spécifiée, gardez à l'esprit que la chaîne de texte est exécutée par le shell. Si vous devez passer des arguments à la commande qui viennent d'une source non approuvée, vous devez prendre soin de supprimer ou échapper des caractères qui pourraient avoir une signification spéciale pour le shell. Pour raisons de sécurité, il est préférable d'utiliser une chaîne de commande fixe comme argument, ou au moins d'éviter d'y fournir des données saisies par des utilisateurs.

Pour une table utilisant `file_fdw`, `EXPLAIN` montre le nom du fichier devant être lu ou le programme à exécuter. Pour un fichier, à moins que `COSTS OFF` soit spécifié, la taille du fichier (en octets) est affichée aussi.

Exemple F.1. Créer une table distante pour les journaux applicatifs PostgreSQL au format CSV

Une des utilisations évidentes de `file_fdw` est de rendre les journaux applicatifs de PostgreSQL disponibles sous la forme d'une table. Pour faire cela, vous devez tout d'abord enregistrer les traces au format CSV. Nous appelerons le fichier de traces `pglog.csv`. Tout d'abord, installez l'extension `file_fdw` :

```
CREATE EXTENSION file_fdw;
```

Ensuite créez un serveur de données distantes :

```
CREATE SERVER pglog FOREIGN DATA WRAPPER file_fdw;
```

Maintenant, vous pouvez créer la table de données distantes. En utilisant la commande `CREATE FOREIGN TABLE`, vous devez définir les colonnes de la table, le nom du fichier CSV, et son format :

```
CREATE FOREIGN TABLE pglog (
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
```

```
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  backend_type text,
  leader_pid integer,
  query_id bigint
) SERVER pglog
OPTIONS ( filename 'log/pglog.csv', format 'csv' );
```

C'est tout -- maintenant, vous pouvez lire le fichier en exécutant une requête sur cette table. Bien sûr, en production, vous aurez besoin de définir un moyen pour tenir compte de la rotation du fichier de traces.

F.17. fuzzystrmatch

Le module `fuzzystrmatch` fournit diverses fonctions qui permettent de déterminer les similarités et la distance entre des chaînes.

> **Attention**
>
> À présent, les `soundex`, `metaphone`, `dmetaphone` et `dmetaphone_alt` ne fonctionnent pas correctement avec les encodages multi-octets (comme l'UTF-8).

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

F.17.1. Soundex

Le système Soundex est une méthode qui permet d'associer des noms (ou des mots) dont la prononciation est proche en les convertissant dans le même code. Elle a été utilisée à l'origine par le « United States Census » en 1880, 1900 et 1910. Soundex n'est pas très utile pour les noms qui ne sont pas anglais.

Le module `fuzzystrmatch` fournit deux fonctions pour travailler avec des codes Soundex :

```
soundex(text) returns text
difference(text, text) returns int
```

La fonction `soundex` convertit une chaîne en son code Soundex. La fonction `difference` convertit deux chaînes en leurs codes Soundex, puis rapporte le nombre de positions de code correspondant. Comme les codes Soundex ont quatre caractères, le résultat va de zéro à quatre. Zéro correspond à aucune correspondance, quatre à une correspondance exacte. (Du coup, la fonction est mal nommée -- `similarity` aurait été un meilleur nom.)

Voici quelques exemples d'utilisation :

```
SELECT soundex('hello world!');

SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
```

```
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne',
 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne',
 'Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');

SELECT * FROM s WHERE soundex(nm) = soundex('john');

SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

F.17.2. Levenshtein

Cette fonction calcule la distance de Levenshtein entre deux chaînes :

```
  levenshtein(text source, text target, int ins_cost, int
 del_cost, int sub_cost) returns int
  levenshtein(text source, text target) returns int
  levenshtein_less_equal(text source, text target, int ins_cost,
 int del_cost, int sub_cost, int max_d) returns int
  levenshtein_less_equal(text source, text target, int max_d)
 returns int
```

La `source` et la cible (`target`) sont des chaînes quelconques non NULL de 255 caractères. Les paramètres de coût indiquent respectivement le coût d'une insertion, suppression ou substitution d'un paramètre. Vous pouvez omettre les paramètres de coût, comme dans la deuxième version. Dans ce cas, elles ont 1 comme valeur par défaut.

`levenshtein_less_equal` est une version accélérée de la fonction Levenshtein à utiliser que lorsque de petites distances sont intéressantes. Si la distance réelle est inférieure ou égale à `max_d`, alors `levenshtein_less_equal` renvoie la bonne distance ; sinon elle renvoie une valeur supérieure à `max_d`. Si `max_d` est négatif, alors le comportement est identique à `levenshtein`.

Exemples :

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
 levenshtein
-------------
           2
(1 row)

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2,1,1);
 levenshtein
-------------
           3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',2);
 levenshtein_less_equal
```

```
------------------------
                       3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',4);
 levenshtein_less_equal
------------------------
                       4
(1 row)
```

F.17.3. Metaphone

Metaphone, comme Soundex, construit un code représentatif de la chaîne en entrée. Deux chaînes sont considérées similaires si elles ont le même code.

Cette fonction calcule le code metaphone d'une chaîne en entrée :

```
metaphone(text source, int max_output_length) returns text
```

`source` doit être une chaîne non NULL de 255 caractères au maximum. `max_output_length` fixe la longueur maximale du code metaphone résultant ; s'il est plus long, la sortie est tronquée à cette taille.

Exemple

```
test=# SELECT metaphone('GUMBO', 4);
 metaphone
-----------
 KM
(1 row)
```

F.17.4. Double Metaphone

Le système « Double Metaphone » calcule deux chaînes « qui se ressemblent » pour une chaîne en entrée -- une « primaire » et une « alternative ». Dans la plupart des cas, elles sont identiques mais, tout spécialement pour les noms autres qu'anglais, elles peuvent être légèrement différentes, selon la prononciation. Ces fonctions calculent le code primaire et le code alternatif :

```
dmetaphone(text source) returns text
dmetaphone_alt(text source) returns text
```

Il n'y a pas de limite de longueur sur les chaînes en entrée.

Exemple :

```
test=# select dmetaphone('gumbo');
 dmetaphone
------------
 KMP
(1 row)
```

F.18. hstore

Ce module code le type de données `hstore`, qui permet de stocker des ensembles de paires clé/valeur dans une simple valeur PostgreSQL. Ce peut être utile dans divers cas, comme des lignes avec de nombreux attributs rarement examinés ou des données semi-structurées. Les clés et les valeurs sont de simples chaînes de texte.

Ce module est considéré comme « trusted », c'est-à-dire qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) ayant le droit CREATE sur la base de données courante.

F.18.1. Représentation externe de `hstore`

La représentation textuelle d'une valeur `hstore`, utilisée en entrée et en sortie, inclut zéro ou plusieurs paires *`clé`* => *`valeur`* séparées par des virgules. Par exemple :

```
k => v
foo => bar, baz => whatever
"1-a" => "anything at all"
```

L'ordre des paires n'est pas significatif (et peut ne pas être reproduit en sortie). Les espaces blancs entre les paires ou autour des signes => sont ignorés. Ajoutez des guillemets pour les clés et valeurs contenant des espaces, virgules, = ou >. Pour inclure un guillemet double ou un antislash dans une clé ou une valeur, échappez-le avec un antislash.

Chaque clé dans un `hstore` est unique. Si vous déclarez un `hstore` avec des clés dupliquées, seule une sera stockée dans `hstore` et il n'y a pas de garantie sur celle qui sera conservée :

```
SELECT 'a=>1,a=>2'::hstore;
  hstore
----------
 "a"=>"1"
```

Une valeur, mais pas une clé, peut être un NULL SQL. Par exemple :

```
key => NULL
```

Le mot-clé NULL est insensible à la casse. Ajoutez des guillemets autour du NULL pour qu'il soit traité comme toute autre chaîne.

> **Note**
>
> Gardez en tête que le format texte `hstore`, lorsqu'il est utilisé en entrée, s'applique *avant* tout guillemet ou échappement nécessaire. Si vous passez une valeur littérale de type `hstore` via un paramètre, aucun traitement supplémentaire n'est nécessaire. Par contre, si vous la passez comme constante littérale entre guillemets, alors les guillemets simples et, suivant la configuration du paramètre `standard_conforming_strings`, les caractères antislash doivent être échappés correctement. Voir Section 4.1.2.1 pour plus d'informations sur la gestion des chaînes constantes.

En sortie, mettez toujours des guillemets doubles autour des clés et valeurs, même quand cela n'est pas strictement nécessaire.

F.18.2. Opérateurs et fonctions `hstore`

Les opérateurs fournis par le module `hstore` sont montrés dans Tableau F.7 et les fonctions dans Tableau F.8.

Tableau F.7. Opérateurs`hstore`

Operator **Description** **Exemple(s)**
`hstore -> text → text` Renvoie la valeur associée à une clé donnée (`NULL` si inexistante). `'a=>x, b=>y'::hstore -> 'a' → x`
`hstore -> text[] → text[]` Retourne la valeur associée à des clés (`NULL` si inexistantes). `'a=>x, b=>y, c=>z'::hstore -> ARRAY['c','a'] → {"z","x"}`
`hstore || hstore → hstore` Concatène deux `hstore`s. `'a=>b, c=>d'::hstore || 'c=>x, d=>q'::hstore → "a"=>"b", "c"=>"x", "d"=>"q"`
`hstore ? text → boolean` Ce `hstore` contient-il une clé donnée ? `'a=>1'::hstore ? 'a' → t`
`hstore ?& text[] → boolean` Ce `hstore` contient-il toutes les clés indiquées ? `'a=>1,b=>2'::hstore ?& ARRAY['a','b'] → t`
`hstore ?| text[] → boolean` Ce `hstore` contient-il une des clés indiquées ? `'a=>1,b=>2'::hstore ?| ARRAY['b','c'] → t`
`hstore @> hstore → boolean` L'opérande gauche contient-il le droit ? `'a=>b, b=>1, c=>NULL'::hstore @> 'b=>1' → t`
`hstore <@ hstore → boolean` L'opérande gauche est-il contenu dans le droit ? `'a=>c'::hstore <@ 'a=>b, b=>1, c=>NULL' → f`
`hstore - text → hstore` Supprime la clé dans l'opérande gauche. `'a=>1, b=>2, c=>3'::hstore - 'b'::text → "a"=>"1", "c"=>"3"`
`hstore - text[] → hstore` Supprime les clés dans l'opérande gauche. `'a=>1, b=>2, c=>3'::hstore - ARRAY['a','b'] → "c"=>"3"`
`hstore - hstore → hstore` Supprime les clés de l'opérande gauche qui correspondent à des paires de l'opérande droit. `'a=>1, b=>2, c=>3'::hstore - 'a=>4, b=>2'::hstore → "a"=>"1", "c"=>"3"`

Operator **Description** **Exemple(s)**
`anyelement #= hstore → anyelement` Remplace les champs de l'opérande gauche (qui doit être un type composite) avec les valeurs correspondantes du `hstore`. `ROW(1,3) #= 'f1=>11'::hstore → (11,3)`
`%% hstore → text[]` Convertit un `hstore` en tableau alternant les clés et les valeurs. `%% 'a=>foo, b=>bar'::hstore → {a,foo,b,bar}`
`%# hstore → text[]` Convertit `hstore` en un tableau à deux dimensions clé/valeur. `%# 'a=>foo, b=>bar'::hstore → {{a,foo},{b,bar}}`

Tableau F.8. `hstore` Functions

Fonction **Description** **Exemple(s)**
`hstore ( record ) → hstore` Construit un `hstore` à partir d'un enregistrement ou d'une ligne. `hstore(ROW(1,2)) → "f1"=>"1", "f2"=>"2"`
`hstore ( text[] ) → hstore` Construit un `hstore` à partir d'un tableau, qui peut être soit un tableau clé/valeur, soit un tableau à deux dimensions. `hstore(ARRAY['a','1','b','2']) → "a"=>"1", "b"=>"2"` `hstore(ARRAY[['c','3'],['d','4']]) → "c"=>"3", "d"=>"4"`
`hstore ( text[], text[] ) → hstore` Construit un `hstore` à partir de tableaux séparés pour les clés et les valeurs. `hstore(ARRAY['a','b'], ARRAY['1','2']) → "a"=>"1", "b"=>"2"`
`hstore ( text, text ) → hstore` Construit un `hstore` à un seul élément. `hstore('a', 'b') → "a"=>"b"`
`akeys ( hstore ) → text[]` Extrait les clés du `hstore` dans un tableau. `akeys('a=>1,b=>2') → {a,b}`
`skeys ( hstore ) → setof text` Extrait les clés du `hstore` dans un ensemble. `skeys('a=>1,b=>2') →` `a` `b`
`avals ( hstore ) → text[]` Extrait les valeurs du `hstore` dans un tableau. `avals('a=>1,b=>2') → {1,2}`

Fonction
Description
Exemple(s)

`svals ( hstore ) → setof text`
Extrait les valeurs du `hstore` dans un ensemble.
`svals('a=>1,b=>2') →`

```
 1
 2
```

`hstore_to_array ( hstore ) → text[]`
Extrait les clés et les valeurs du `hstore` sous la forme d'un tableau de clés et valeurs alternées.
`hstore_to_array('a=>1,b=>2') → {a,1,b,2}`

`hstore_to_matrix ( hstore ) → text[]`
Extrait les clés et valeurs `hstore` sous la forme d'un tableau à deux dimensions.
`hstore_to_matrix('a=>1,b=>2') → {{a,1},{b,2}}`

`hstore_to_json ( hstore ) → json`
Convertit une valeur `hstore` en `json`, en convertissant toutes les valeurs non NULL en chaînes JSON.
Cette fonction est appelée implicitement quand une valeur `hstore` est convertie vers du `json`.
`hstore_to_json('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4') → {"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}`

`hstore_to_jsonb ( hstore ) → jsonb`
Convertit une valeur `hstore` en `jsonb`, en convertissant toutes les valeurs non NULL en chaînes JSON.
Cette fonction est appelée implicitement quand une valeur `hstore` est convertie vers du `jsonb`.
`hstore_to_jsonb('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4') → {"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}`

`hstore_to_json_loose ( hstore ) → json`
Convertit un `hstore` en une valeur `json`, mais en essayant de repérer les valeurs numériques et booléennes pour qu'elles ne soient pas entre guillemets dans le JSON.
`hstore_to_json_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4') → {"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}`

`hstore_to_jsonb_loose ( hstore ) → jsonb`
Convertit un `hstore` en une valeur `jsonb`, mais en essayant de repérer les valeurs numériques et booléennes pour qu'elles ne soient pas entre guillemets dans le JSON.
`hstore_to_jsonb_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4') → {"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}`

Fonction **Description** **Exemple(s)**
`slice ( hstore, text[] ) → hstore` Extrait un sous-ensemble d'un `hstore` contenant seulement les clés indiquées. `slice('a=>1,b=>2,c=>3'::hstore, ARRAY['b','c','x']) → "b"=>"2", "c"=>"3"`
`each ( hstore ) → setof record ( key text, value text )` Extrait les clés et valeurs d'un `hstore` sous la forme d'un ensemble d'enregistrements. `select * from each('a=>1,b=>2') →` `key | value` `-----+-------` `a | 1` `b | 2`
`exist ( hstore, text ) → boolean` Le `hstore` contient-il une clé donnée ? `exist('a=>1', 'a') → t`
`defined ( hstore, text ) → boolean` Le `hstore` contient-il une valeur non `NULL` comme clé ? `defined('a=>NULL', 'a') → f`
`delete ( hstore, text ) → hstore` Supprime la paire correspondant à une clé donnée. `delete('a=>1,b=>2', 'b') → "a"=>"1"`
`delete ( hstore, text[] ) → hstore` Supprime toutes les paires correspondant à ces clés. `delete('a=>1,b=>2,c=>3', ARRAY['a','b']) → "c"=>"3"`
`delete ( hstore, hstore ) → hstore` Supprime les paires correspondant à celles du second argument. `delete('a=>1,b=>2', 'a=>4,b=>2'::hstore) → "a"=>"1"`
`populate_record ( anyelement, hstore ) → anyelement` Remplacer tous les champs de l'opérande gauche (qui doit être un type composite) avec les valeurs correspondantes du `hstore`. `populate_record(ROW(1,2), 'f1=>42'::hstore) → (42,2)`

En plus de ces opérateurs et fonctions, les valeurs du type `hstore` peuvent utiliser des indices, leur permettant ainsi d'agir comme des tableaux associatifs. Seul un indice simple de type `text` peut être indiqué ; il est interprété comme la clé, et la valeur correspondante est récupérée ou stockée. Par exemple,

```
CREATE TABLE mytable (h hstore);
INSERT INTO mytable VALUES ('a=>b, c=>d');
SELECT h['a'] FROM mytable;
 h
---
 b
(1 row)
```

```
UPDATE mytable SET h['c'] = 'new';
SELECT h FROM mytable;
          h
----------------------
 "a"=>"b", "c"=>"new"
(1 row)
```

Une lecture par indice renvoie `NULL` si l'indice est `NULL` ou si la clé n'existe pas dans ce `hstore`. (De ce fait, une lecture par indice n'est pas fortement différente de l'opérateur `->`.) Une mise à jour par indice échoue si l'indice est `NULL` ; sinon elle remplace la valeur pour cette clé, ajoutant une entrée au `hstore` si la clé n'existe pas déjà.

F.18.3. Index

`hstore` est supporté par les index GiST et GIN pour les opérateurs `@>`, `?`, `?&` et `?|`. Par exemple :

```
CREATE INDEX hidx ON testhstore USING GIST (h);

CREATE INDEX hidx ON testhstore USING GIN (h);
```

La classe d'opérateurs GiST `gist_hstore_ops` opère une approximation d'un ensemble de clés/valeurs avec une signature bitmap. Son paramètre optionnel entier, `siglen`, détermine la longueur de la signature en octets. La longueur par défaut est 16 octets. Les valeurs valides de longueur de signature vont de 1 à 2024 octets. Des signatures plus longues mènent à des recherches plus précises (scannant une partie plus courte de l'index et moins de pages de la table), au prix d'un index plus gros.

Exemple de création d'un tel index avec une longueur de signature de 32 octets :

```
CREATE INDEX hidx ON testhstore USING GIST (h
 gist_hstore_ops(siglen=32));
```

`hstore` supporte aussi les index `btree` ou `hash` pour l'opérateur `=`. Cela permet aux colonnes `hstore` d'être déclarées `UNIQUE` et d'être utilisées dans des expressions `GROUP BY`, `ORDER BY` et `DISTINCT`. L'ordre de tri pour les valeurs `hstore` n'est pas particulièrement utile mais ces index peuvent servir à des recherches d'équivalence. Créez des index de comparaisons `=` de la façon suivante :

```
CREATE INDEX hidx ON testhstore USING BTREE (h);

CREATE INDEX hidx ON testhstore USING HASH (h);
```

F.18.4. Exemples

Ajouter une clé, ou mettre à jour une clé existante avec une nouvelle valeur :

```
UPDATE tab SET h['c'] = '3';
```

Voici une autre façon de faire la même chose :

```
UPDATE tab SET h = h || hstore('c', '3');
```

Si plusieurs clés doivent être ajoutées ou modifiées en une seule opération, l'approche par concaténation est plus efficace que par indice :

```
UPDATE tab SET h = h || hstore(array['q', 'w'], array['11', '12']);
```

Supprimer une clé :

```
UPDATE tab SET h = delete(h, 'k1');
```

Convertir un enregistrement (`record`) en un `hstore` :

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT hstore(t) FROM test AS t;
                  hstore
-------------------------------------------
 "col1"=>"123", "col2"=>"foo", "col3"=>"bar"
(1 row)
```

Convertir un type `hstore` en un type `record` prédéfini :

```
CREATE TABLE test (col1 integer, col2 text, col3 text);

SELECT * FROM populate_record(null::test,
                              '"col1"=>"456", "col2"=>"zzz"');
 col1 | col2 | col3
------+------+------
  456 | zzz  |
(1 row)
```

Modifier un enregistrement existant en utilisant les valeurs provenant d'un `hstore` :

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT (r).* FROM (SELECT t #= '"col3"=>"baz"' AS r FROM test t) s;
 col1 | col2 | col3
------+------+------
  123 | foo  | baz
(1 row)
```

F.18.5. Statistiques

Le type `hstore`, du fait de sa libéralité intrinsèque, peut contenir beaucoup de clés différentes. C'est à l'application de vérifier la validité des clés. Les exemples ci-dessous présentent plusieurs techniques pour vérifier les clés et obtenir des statistiques.

Exemple simple :

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

En utilisant une table :

```
CREATE TABLE stat AS SELECT (each(h)).key, (each(h)).value FROM
 testhstore;
```

Statistiques en ligne :

```
SELECT key, count(*) FROM
  (SELECT (each(h)).key FROM testhstore) AS stat
  GROUP BY key
  ORDER BY count DESC, key;
    key    | count
-----------+-------
 line      |   883
 query     |   207
 pos       |   203
 node      |   202
 space     |   197
 status    |   195
 public    |   194
 title     |   190
 org       |   189
.................
```

F.18.6. Compatibilité

À partir de PostgreSQL 9.0, `hstore` utilise une représentation interne différente des anciennes versions. Cela ne présente aucun obstacle pour les mises à jour par sauvegarde/restauration car la représentation textuelle utilisée dans la sauvegarde n'est pas changée.

Dans le cas d'une mise à jour binaire, la compatibilité ascendante est maintenue en faisant en sorte que le nouveau code reconnaisse les données dans l'ancien format. Ceci aura pour conséquence une légère pénalité au niveau des performances lors du traitement de données qui n'auront pas été modifiées par le nouveau code. Il est possible de forcer une mise à jour de toutes les valeurs d'une colonne de la table en réalisant la requête `UPDATE` suivante :

```
UPDATE nom_table SET col_hstore = col_hstore || '';
```

Une autre façon de le faire :

```
ALTER TABLE nom_table ALTER col_hstore TYPE col_hstore USING
 hstorecol || '';
```

La méthode `ALTER TABLE` requiert un verrou de type `ACCESS EXCLUSIVE` sur la table, mais n'a pas pour résultat un gonflement de la table à cause d'anciennes versions des lignes.

F.18.7. Transformations

Des extensions supplémentaires sont disponibles, qui implémentent des transformations pour le type `hstore` pour les langages PL/Perl et PL/Python. Les extensions pour PL/Perl sont appelées `hstore_plperl` et `hstore_plperlu`, pour les deux versions de PL/Perl. Si vous installez ces transformations et si vous les spécifiez lors de la création d'une fonction, les valeurs `hstore` sont converties en hachage Perl. L'extension pour PL/Python s'appelle `hstore_plpython3u`. Si vous l'utilisez, les valeurs `hstore` sont converties en dictionnaires Python.

De ces extensions additionnelles, `hstore_plperl` est considéré « trusted » ; pas les autres.

> **Attention**
>
> Il est fortement recommandé que les extensions de transformation soient installées dans le même schéma que `hstore`. Sinon, il existe un risque de sécurité si le schéma d'une extension de transformation contient des objets définis par un utilisateur hostile.

F.18.8. Auteurs

Oleg Bartunov `<oleg@sai.msu.su>`, Moscou, Université de Moscou, Russie

Teodor Sigaev `<teodor@sigaev.ru>`, Moscou, Delta-Soft Ltd., Russie

Améliorations supplémentaires par Andrew Gierth `<andrew@tao11.riddles.org.uk>`, Royaume Uni

F.19. intagg

Le module `intagg` fournit un agrégateur d'entiers et un énumérateur. `intagg` est maintenant obsolète car il existe des fonctions natives qui fournissent les mêmes fonctionnalités et au-delà. Néanmoins, le module est toujours disponible pour la compatibilité et utilise ces fonctions natives.

F.19.1. Fonctions

L'agrégateur est une fonction d'agrégat `int_array_aggregate(integer)` qui produit un tableau d'entiers contenant exactement les entiers fournis en argument. Cette fonction appelle `array_agg`, qui fait la même chose pour n'importe quel type de tableau.

L'énumérateur est une fonction `int_array_enum(integer[])` qui renvoie `setof integer`. C'est essentiellement une opération inverse de l'agrégateur : elle développe un tableau d'entiers en un ensemble de lignes. Cette fonction utilise `unnest`, qui fait la même chose pour n'importe quel type de tableau.

F.19.2. Exemples d'utilisation

Un grand nombre de bases de données utilisent la notion de table « une vers plusieurs » (*one to many*). Ce type de table se trouve habituellement entre deux tables indexées, par exemple :

```
CREATE TABLE left (id INT PRIMARY KEY, ...);
CREATE TABLE right (id INT PRIMARY KEY, ...);
CREATE TABLE one_to_many(left INT REFERENCES left, right INT
 REFERENCES right);
```

Il est typiquement utilisé de cette façon :

```
  SELECT right.* from right JOIN one_to_many ON (right.id =
 one_to_many.right)
    WHERE one_to_many.left = item;
```

Cela renverra tous les éléments de la table de droite pour un enregistrement donné de la table de gauche. C'est une construction très courante en SQL.

Cette méthode devient complexe lorsqu'il existe de nombreuses entrées dans la table one_to_many. Souvent, une jointure de ce type résulte en un parcours d'index et une récupération de chaque enregistrement de la table de droite pour une entrée de la table de gauche. Sur un système très dynamique, il n'y a pas grand-chose à faire. Au contraire, lorsqu'une partie des données est statique, une table de résumé peut être créée par agrégation.

```
CREATE TABLE summary AS
  SELECT left, int_array_aggregate(right) AS right
  FROM one_to_many
  GROUP BY left;
```

Ceci crée une table avec une ligne par élément gauche et un tableau d'éléments droits. Sans un moyen d'utiliser ce tableau, c'est à peu près inutilisable, d'où l'énumérateur.

Exemple :

```
SELECT left, int_array_enum(right) FROM summary WHERE left = item;
```

La requête ci-dessus, qui utilise int_array_enum, produit les mêmes résultats que celle-ci :

```
SELECT left, right FROM one_to_many WHERE left = item;
```

Ici la requête sur la table de résumé ne récupère qu'une ligne de la table alors que la requête directe à one_to_many doit faire un parcours d'index et récupérer une ligne par enregistrement.

Sur une instance, un EXPLAIN a montré qu'une requête avec un coût de 8488 a été réduite à un coût de 329. La requête originale était une jointure impliquant la table one_to_many, remplacée par :

```
SELECT right, count(right) FROM
  ( SELECT left, int_array_enum(right) AS right
    FROM summary JOIN (SELECT left FROM left_table WHERE left
 = item) AS lefts
         ON (summary.left = lefts.left)
  ) AS list
  GROUP BY right
  ORDER BY count DESC;
```

F.20. intarray

Le module intarray fournit un certain nombre de fonctions et d'opérateurs utiles pour manipuler des tableaux d'entiers sans valeurs NULL. Il y a aussi un support pour les recherches par index en utilisant certains des opérateurs.

Toutes ces opérations rejeteront une erreur si un tableau fourni contient des éléments NULL.

La plupart des opérations sont seulement intéressantes pour des tableaux à une dimension. Bien qu'elles acceptent des tableaux à plusieurs dimensions, les données sont traitées comme s'il y avait un tableau linéaire.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

F.20.1. Fonctions et opérateurs d'`intarray`

Les fonctions fournies par le module `intarray` sont affichées dans Tableau F.9 alors que les opérateurs sont indiqués dans Tableau F.10.

Tableau F.9. Fonctions `intarray`

Function **Description** **Example(s)**
`icount ( integer[] ) → integer` Retourne le nombre d'éléments dans le tableau. `icount('{1,2,3}'::integer[]) → 3`
`sort ( integer[], dir text ) → integer[]` Trie le tableau par ordre croissant ou décroissant. *`dir`* doit valoir `asc` ou `desc`. `sort('{1,3,2}'::integer[], 'desc') → {3,2,1}`
`sort ( integer[] ) → integer[]` `sort_asc ( integer[] ) → integer[]` Trie par ordre croissant. `sort(array[11,77,44]) → {11,44,77}`
`sort_desc ( integer[] ) → integer[]` Trie par ordre décroissant. `sort_desc(array[11,77,44]) → {77,44,11}`
`uniq ( integer[] ) → integer[]` Supprime les doublons adjacents. Souvent utilisé avec `sort` pour supprimer tous les doublons. `uniq('{1,2,2,3,1,1}'::integer[]) → {1,2,3,1}` `uniq(sort('{1,2,3,2,1}'::integer[])) → {1,2,3}`
`idx ( integer[], item integer ) → integer` Retourne l'index du premier élément correspondant à *`item`*, ou 0 si pas de correspondance. `idx(array[11,22,33,22,11], 22) → 2`
`subarray ( integer[], start integer, len integer ) → integer[]` Extrait une partie d'un tableau depuis la position *`start`*, avec *`len`* éléments. `subarray('{1,2,3,2,1}'::integer[], 2, 3) → {2,3,2}`
`subarray ( integer[], start integer ) → integer[]` Extrait une partie de tableau en commençant à la position *`start`*. `subarray('{1,2,3,2,1}'::integer[], 2) → {2,3,2,1}`
`intset ( integer ) → integer[]` Crée un tableau d'un seul élément. `intset(42) → {42}`

Tableau F.10. Opérateurs d'`intarray`

Opérateur **Description**
`integer[] && integer[] → boolean`

Opérateur **Description**
Est-ce que les tableaux se recoupent (ont-ils au moins un élément en commun) ?
`integer[]` `@>` `integer[]` → `boolean` Est-ce que le tableau contient le tableau de droite ?
`integer[]` `<@` `integer[]` → `boolean` Est-ce que le tableau de gauche est contenu dans le tableau de droite ?
`#` `integer[]` → `integer` Retourne le nombre d'éléments dans le tableau.
`integer[]` `#` `integer` → `integer` Retourne l'indice du premier élément du tableau correspondant à l'argument de droite, ou 0 s'il n'y a pas de correspondance. (Identique à la fonction `idx`.)
`integer[]` `+` `integer` → `integer[]` Ajoute un élement en fin de tableau.
`integer[]` `+` `integer[]` → `integer[]` Concatène les tableaux.
`integer[]` `-` `integer` → `integer[]` Supprime les entrées correspondantes à l'argument de droite dans le tableau.
`integer[]` `-` `integer[]` → `integer[]` Supprime les éléments du tableau de droite dans le tableau de gauche.
`integer[]` `|` `integer` → `integer[]` Calcule l'union des arguments.
`integer[]` `|` `integer[]` → `integer[]` Calcule l'union des tableaux en arguments.
`integer[]` `&` `integer[]` → `integer[]` Calcule l'intersection des tableaux en arguments.
`integer[]` `@@` `query_int` → `boolean` Est-ce que le tableau satisfait la requête ? (voir ci-dessous)
`query_int` `~~` `integer[]` → `boolean` Est-ce que le tableau satisfait la requête ? (commuteur pour `@@`)

Les opérateurs `&&`, `@>` et `<@` sont équivalents aux opérateurs internes PostgreSQL de même nom, sauf qu'ils travaillent sur des tableaux d'entiers, sans valeurs NULL, alors que les opérateurs internes travaillent sur des tableaux de tout type. Cette restriction les rend plus rapides que les opérateurs internes dans de nombreux cas.

Les opérateurs `@@` et `~~` testent si un tableau satisfait une *requête*, qui est exprimée comme une valeur d'un type de données spécialisé `query_int`. Une *requête* consiste en des valeurs de type integer qui sont vérifiées avec les éléments du tableau, parfois combinées en utilisant les opérateurs & (AND), `|` (OR) et `!` (NOT). Les parenthèses peuvent être utilisées si nécessaire. Par exemple, la requête `1&(2|3)` établit une correspondance avec les tableaux qui contiennent 1 et aussi soit 2 soit 3.

F.20.2. Support des index

`intarray` fournit un support d'index pour les opérateurs `&&`, `@>` et `@@`, ainsi que pour l'égalité de tableaux.

Deux classes d'opérateurs pour index GiST, avec paramètres, sont fournies : `gist__int_ops` (utilisé par défaut) convient pour des tableaux d'ensembles de données de petites et moyennes tailles

alors que `gist__intbig_ops` utilise une signature plus importante et est donc plus intéressant pour indexer des gros ensembles de données. (c'est-à-dire les colonnes contenant un grand nombre de valeurs de tableaux distinctes). L'implantation utilise une structure de données RD-tree avec une compression interne à perte.

`gist__int_ops` donne une approximation d'un ensemble d'entiers sous forme d'un tableau d'intervalles d'entiers. Son paramètre entier optionnel `numranges` détermine le nombre maximal d'intervalles dans une clé d'index. La valeur par défaut de `numranges` est 100. Les valeurs valides sont entre 1 et 253. Utiliser des tableaux plus grands comme clés d'index GiST entraîne une recherche plus précise (en parcourant une fraction plus petite de l'index et donc moins de pages de la table), au détriment d'un index plus gros.

`gist__intbig_ops` donne une approximation d'un ensemble d'entiers sous forme d'une signature bitmap. Son paramètre entier optionnel `siglen` détermine la taille de la signature en octets. La taille de signature par défaut est 16 octets. Les valeurs valides pour la taille de signature est entre 1 et 2024 octets. Une signature plus longue entraîne une recherche plus précise (en parcourant une fraction plus petite de l'index et donc moins de pages de la table), au détriment d'un index plus gros.

Il y a aussi une classe d'opérateurs GIN,`gin__int_ops` supportant les mêmes opérateurs ainsi que `<@`, mais qui n'est pas disponible par défaut.

Le choix d'un indexage GiST ou IN dépend des caractéristiques relatives de performance qui sont discutées ailleurs.

F.20.3. Exemple

```
-- un message peut être dans un ou plusieurs « sections »
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);

-- crée un index spécialisé avec une signature de longueur 32
 octets
CREATE INDEX message_rdtree_idx ON message USING GIST (sections
 gist__intbig_ops (siglen=32));

-- sélectionne les messages dans la section 1 ou 2 - opérateur
 OVERLAP
SELECT message.mid FROM message WHERE message.sections && '{1,2}';

-- sélectionne les messages dans sections 1 et 2 - opérateur
 CONTAINS
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';

-- idem, en utilisant l'opérateur QUERY
SELECT message.mid FROM message WHERE message.sections @@
 '1&2'::query_int;
```

F.20.4. Tests de performance

Le répertoire des sources (`contrib/intarray/bench`) contient une suite de tests de performance, qui peut être exécutée sur un serveur PostgreSQL déjà installé. (Cela nécessite aussi l'installation de `DBD::Pg`.). Pour l'exécuter :

```
cd .../contrib/intarray/bench
createdb TEST
psql -c "CREATE EXTENSION intarray" TEST
./create_test.pl | psql TEST
```

```
./bench.pl
```

Le script `bench.pl` contient un grand nombre d'options. Elles sont affichées quand il est exécuté sans arguments.

F.20.5. Auteurs

Ce travail a été réalisé par Teodor Sigaev (<teodor@sigaev.ru>) et Oleg Bartunov (<oleg@sai.msu.su>). Voir le site de GiST[2] pour des informations supplémentaires. Andrey Oktyabrski a fait un gros travail en ajoutant de nouvelles fonctions et opérateurs.

F.21. isn

Le module `isn` fournit des types de données pour les standards internationaux de numérotation suivants : EAN13, UPC, ISBN (livres), ISMN (musique) et ISSN (numéro de série). Les nombres sont validés en saisie suivant une liste de préfixes codés en dur ; cette liste de préfixes est aussi utilisée pour placer un trait d'union sur les nombres en sortie. Comme de nouveaux préfixes sont ajoutés de temps en temps, la liste des préfixes pourrait devenir obsolète. Il est probable qu'une prochaine version de ce module utilisera une liste stockée sous la forme d'une ou plusieurs tables qui pourront être modifiées aisément par les utilisateurs quand cela se révélera nécessaire. Néanmoins, actuellement, la liste est modifiable uniquement par changement du code source et recompilation. Il est aussi possible que la validation du préfixe et le support des traits d'union soient supprimés de ce module dans une version future.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

F.21.1. Types de données

Tableau F.11 affiche les types de données fournis par le module `isn`.

Tableau F.11. Types de données `isn`

Type de données	Description
EAN13	Numéro d'article européen (*European Article Numbers*), toujours affiché dans le format de l'EAN13
ISBN13	Numéro standard international pour les livres (*International Standard Book Numbers*) à afficher dans le nouveau format EAN13
ISMN13	Numéro standard international pour la musique (*International Standard Music Numbers*) à afficher dans le nouveau format EAN13
ISSN13	Numéro de série au standard international (*International Standard Serial Numbers*) à afficher dans le nouveau format EAN13
ISBN	Numéro standard international pour les livres (*International Standard Book Numbers*) à afficher dans l'ancien format court
ISMN	Numéro standard international pour la musique (*International Standard Music Numbers*) à afficher dans l'ancien format court
ISSN	Numéro de série au standard international (*International Standard Serial Numbers*) à afficher dans l'ancien format court
UPC	Code produit universel (*Universal Product Codes*)

[2] http://www.sai.msu.su/~megera/postgres/gist

Quelques notes :

1. Les nombres ISBN13, ISMN13, ISSN13 sont tous des nombres EAN13.
2. Les nombres EAN13 ne sont pas toujours des ISBN13, ISMN13 ou ISSN13 (mais certains le sont).
3. Certains nombres ISBN13 peuvent être affichés comme des ISBN.
4. Certains nombres ISMN13 peuvent être affichés comme des ISMN.
5. Certains nombres ISSN13 peuvent être affichés comme des ISSN.
6. Les nombres UPC sont un sous-ensemble des nombres EAN13 (ce sont basiquement des EAN13 sans le premier `0`).
7. Tous les nombres UPC, ISBN, ISMN et ISSN numbers peuvent être représentés sous la forme EAN13.

En interne, tous ces types utilisent la même représentation (un entier sur 64 bits), et tous sont interchangeables. Plusieurs types sont fournis pour contrôler le formatage de l'affichage et pour permettre une vérification très fine de la validité des entrées qui est supposée dénoter un type particulier de nombre.

Les types `ISBN`, `ISMN` et `ISSN` afficheront la version courte du nombre (ISxN 10) quand c'est possible, et afficheront la version au format ISxN 13 pour les nombres qui ne tiennent pas dans la version courte. Les types `EAN13`, `ISBN13`, `ISMN13` et `ISSN13` afficheront toujours la version longue de l'ISxN (EAN13).

F.21.2. Conversions

Le module `isn` fournit les paires suivantes pour les conversions de types :

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13
- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISBN <=> ISBN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

Lors d'une conversion d'`EAN13` vers un autre type, il y a une vérification à l'exécution que la valeur est dans le domaine de l'autre type et une erreur est renvoyée dans le cas contraire. Les autres conversions sont simplement un renommage qui succèdera à chaque fois.

F.21.3. Fonctions et opérateurs

Le module `isn` fournit des opérateurs de comparaison standard, plus un support des index B-Tree et hachés pour tous les types de données. De plus, il existe plusieurs fonctions spécialisées, listées dans Tableau F.12. Dans cette table, `isn` signifie un des types de données de ce module :

Tableau F.12. Fonctions de `isn`

`isn_weak` (`boolean`) → `boolean` Configure le mode de saisie faible et renvoie le nouveau paramétrage.
`isn_weak` () → `boolean` Renvoie le statut actuel du mode faible
`make_valid` (`isn`) → `isn` Valide un nombre invalide (efface le drapeau d'invalidité).
`is_valid` (`isn`) → `boolean` Vérifie la présence du drapeau d'invalidité

Le mode *faible* est utilisé pour insérer des données invalides dans une table. Invalide signifie que le chiffre de vérification est mauvais, pas qu'il manque des numéros.

Pourquoi voudriez-vous utiliser le mode faible ? Tout simplement parce que vous pouvez avoir une grosse collection de nombres ISBN, et que beaucoup d'entre eux, quelque soit la raison, ont un mauvais chiffre de vérification (peut-être que les nombres ont été scannés à partir d'une liste imprimée et que l'OCR s'est trompé sur les numéros, peut-être que les numéros ont été saisis manuellement... qui sait). Bref, le fait est que vous pouvez vouloir corriger ça, mais que vous voulez être capable d'avoir tous les nombres dans votre base de données pour que vous puissiez vérifier l'information et peut-être utiliser un outil externe pour localiser les nombres invalides dans la base de données, puis les vérifier et valider plus facilement ; donc par exemple, vous voudrez sélectionner tous les nombres invalides dans la table.

Quand vous insérez des nombres invalides dans une table en utilisant le mode faible, le nombre sera inséré avec le chiffre de vérification corrigé, mais il sera affiché avec un point d'exclamation (`!`) à la fin, par exemple `0-11-000322-5!`. Ce marqueur d'invalidité peut être vérifié avec la fonction `is_valid` et effacé avec la fonction `make_valid`.

Vous pouvez aussi forcer l'insertion de nombres invalides, même quand vous n'êtes pas dans le mode faible, en ajoutant le caractère `!` à la fin du nombre.

Une autre fonctionnalité spéciale est que, durant la saisie, vous pouvez écrire `?` à la place du chiffre de vérification. Ce dernier sera calculé et inséré automatiquement.

F.21.4. Exemples

```
--Using the types directly:
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');

--Casting types:
-- note that you can only cast from ean13 to another type when the
-- number would be valid in the realm of the target type;
-- thus, the following will NOT work: select
 isbn(ean13('0220356483481'));
-- but these will:
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));

--Create a table with a single column to hold ISBN numbers:
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');

--Automatically calculate check digits (observe the '?'):
```

```
INSERT INTO test VALUES('220500896?');
INSERT INTO test VALUES('978055215372?');

SELECT issn('3251231?');
SELECT ismn('979047213542?');

--Using the weak mode:
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);

SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X!';

SELECT * FROM test;

SELECT isbn13(id) FROM test;
```

F.21.5. Bibliographie

Les informations qui ont permis l'implémentation de ce module ont été récupérées sur plusieurs sites, dont :

- https://www.isbn-international.org/
- https://www.issn.org/
- https://www.ismn-international.org/
- https://www.wikipedia.org/

Les préfixes utilisées pour le formatage ont été récupérés à partir de :

- https://www.gs1.org/standards/id-keys
- https://en.wikipedia.org/wiki/List_of_ISBN_identifier_groups
- https://www.isbn-international.org/content/isbn-users-manual/29
- https://en.wikipedia.org/wiki/International_Standard_Music_Number
- https://www.ismn-international.org/ranges/tools

Nous avons porté une grande attention lors de la création des algorithmes et ils ont été vérifiés méticuleusement par rapport aux algorithmes suggérés dans les manuels utilisateurs officiels ISBN, ISMN et ISSN.

F.21.6. Auteur

Germán Méndez Bravo (Kronuz), 2004–2006

Ce module est inspiré du code `isbn_issn` de Garrett A. Wollman.

F.22. lo

Le module `lo` ajoute un support des « Large Objects » (aussi appelé LO ou BLOB). Il inclut le type de données `lo` et un trigger `lo_manage`.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

F.22.1. Aperçu

Un des problèmes avec le pilote JDBC (mais cela affecte aussi le pilote ODBC) est que la spécification suppose que les références aux BLOB (Binary Large OBject) sont stockées dans une table et que, si une entrée est modifiée, le BLOB associé est supprimé de cette base.

En l'état actuel de PostgreSQL, ceci n'arrive pas. Les « Large Objects » sont traités comme des objets propres ; une entrée de table peut référencer un « Large Object » par son OID, mais plusieurs tables peuvent référencer le même OID. Donc, le système ne supprime pas un « Large Object » simplement parce que vous modifiez ou supprimez une entrée contenant un tel OID.

Ceci n'est pas un problème pour les applications spécifiques à PostgreSQL mais un code standard utilisant JDBC ou ODBC ne supprimera pas ces objets, ceci aboutissant à des « Large Objects » orphelins -- des objets qui ne sont référencés par personne et occupant de la place.

Le module `lo` permet de corriger ceci en attachant un trigger aux tables contenant des colonnes de référence des LO. Le trigger fait essentiellement un `lo_unlink` quand vous supprimez ou modifiez une valeur référencée un « Large Object ». Quand vous utilisez ce trigger, vous supposez que, dans toute la base de données, il n'existe qu'une seule référence d'un « Large Object » référencé dans une colonne contrôlée par un trigger !

Le module fournit aussi un type de données `lo`, qui n'est qu'un *domaine* sur le type `oid`. Il est utile pour différencier les colonnes de la base qui contiennent des références d'objet de ceux qui contiennent des OID sur d'autres choses. Vous n'avez pas besoin d'utiliser le type `lo` pour utiliser le trigger mais cela facilite le travail pour garder la trace des colonnes de votre base qui représentent des « Large Objects » que vous gérez avec le trigger. Une rumeur dit aussi que le pilote ODBC a du mal si vous n'utilisez pas le type `lo` pour les colonnes BLOB.

F.22.2. Comment l'utiliser

Voici un exemple d'utilisation :

```
CREATE TABLE image (title TEXT, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
  FOR EACH ROW EXECUTE FUNCTION lo_manage(raster);
```

Pour chaque colonne qui contiendra des références uniques aux « Large Objects », créez un trigger BEFORE UPDATE OR DELETE trigger, et donnez le nom de la colonne comme argument du trigger. Vous pouvez aussi restreindre le trigger pour ne s'exécuter que sur les mises à jour de la colonne en utilisant BEFORE UPDATE OF *nom_colonne*. Si vous avez plusieurs colonnes `lo` dans la même table, créez un trigger séparé pour chacune en vous souvenant de donner un nom différent à chaque trigger sur la même table.

F.22.3. Limites

- Supprimer une table résultera quand même en des objets orphelins pour tous les objets qu'elle contient, car le trigger n'est pas exécuté. Vous pouvez éviter ceci en faisant précéder le DROP TABLE avec DELETE FROM *table*.

  TRUNCATE présente le même danger.

Si vous avez déjà, ou suspectez avoir, des « Large Objects » orphelins, voir le module vacuumlo (vacuumlo) pour vous aider à les nettoyer. Une bonne idée est d'exécuter vacuumlo occasionnellement pour s'assurer du ménage réalisé par le trigger `lo_manage`.

- Quelques interfaces peuvent créer leurs propres tables et n'ajouteront pas les triggers associés. De plus, les utilisateurs peuvent oublier de créer les triggers, ou ne pas savoir le faire.

F.22.4. Auteur

Peter Mount <peter@retep.org.uk>

F.23. ltree

Ce module implémente le type de données `ltree` pour représenter des labels de données stockés dans une structure hiérarchique de type arbre. Des fonctionnalités étendues de recherche sont fournies.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

F.23.1. Définitions

Un *label* est une séquence de caractères alphanumériques et de tirets bas (par exemple, dans la locale C, les caractères `A-Za-z0-9_` sont autorisés). La longueur d'un label est limité par 256 caractères.

Exemples : `42`, `Personal_Services`

Le *chemin de label* est une séquence de zéro ou plusieurs labels séparés par des points, par exemple `L1.L2.L3`, ce qui représente le chemin de la racine jusqu'à un nœud particulier. La longueur d'un chemin est limité à 65535 labels.

Exemple : `Top.Countries.Europe.Russia`

Le module `ltree` fournit plusieurs types de données :

- `ltree` stocke un chemin de label.
- `lquery` représente un type d'expression rationnelle du chemin pour la correspondance de valeurs de type `ltree`. Un mot simple établit une correspondance avec ce label dans un chemin. Le caractère joker (`*`) est utilisé pour spécifier tout nombre de labels (niveaux). Ils peuvent être liés avec des points pour former des motifs qui doivent correspondre au chemin entier du label. Par exemple :

```
foo         Correspond au chemin exact foo
*.foo.*     Correspond à tout chemin contenant le label foo
*.foo       Correspond à tout chemin dont le dernier label est
foo
```

Les caractères joker et les mots simples peuvent être quantifiés pour restreindre le nombre de labels de la correspondance :

```
*{n}        Correspond à exactement n labels
*{n,}       Correspond à au moins n labels
*{n,m}      Correspond à au moins n labels mais à pas plus de m
*{,m}       Correspond à au plus m labels  --  identiques
 à *{0,m}
```

```
foo{n,m}    Correspond à au moins n mais pas plus que m
 occurrences de foo
foo{,}      Correspond à n'importe quel nombre d'occurrences
 de foo, incluant zéro
```

En absence d'un quantificateur explicite, le défaut pour un symbole étoile est de correspondre à n'importe quel nombre de labels (c'est-à-dire `{,}`) tandis que le défaut pour un élément non-étoile est qu'il doit correspondre exactement une fois (c'est-à-dire `{1}`).

Il existe plusieurs modificateurs qui peuvent être placés à la fin d'un élément `lquery` sans joker pour que la correspondance se fasse sur plus que la correspondance exacte :

```
@           Correspondance sans vérification de casse, par
 exemple a@ établit une correspondance avec A
*           Correspondance d'un préfixe pour un label, par
 exemple foo* établit une correspondance avec foobar
%           Correspondance avec les mots séparés par des tirets
 bas
```

Le comportement de `%` est un peu complexe. Il tente d'établir une correspondance avec des mots plutôt qu'avec un label complet. Par exemple, `foo_bar%` établit une correspondance avec `foo_bar_baz` mais pas avec `foo_barbaz`. S'il est combiné avec `*`, la correspondance du préfixe s'applique à chaque mot séparément. Par exemple, `foo_bar%*` établit une correspondance avec `foo1_bar2_baz`, mais pas avec `foo1_br2_baz`.

Vous pouvez aussi écrire de nombreux éléments non-étoiles, modifiés ou non, séparés avec `|` (OR) pour correspondre à au moins un de ces éléments, et vous pouvez mettre un `!` (NOT) au début d'un groupe non-étoile pour correspondre à tout label qui ne correspond à aucune alternative. Un quantificateur, s'il y a, va toujours à la fin du groupe ; cela signifie un certain nombre de correspondances pour le groupe entier (c'est-à-dire, un certain nombre de labels correspondants ou non correspondants à n'importe quelle alternative.)

Voici un exemple annoté d'une `lquery` :

```
Top.*{0,2}.sport*@.!football|tennis{1,}.Russ*|Spain
a.  b.     c.      d.                   e.
```

Cette requête établira une correspondance avec tout chemin qui :

a. commence avec le label `Top`

b. et suit avec zéro ou deux labels jusqu'à

c. un label commençant avec le préfixe `sport` quelque soit la casse

d. puis un ou plusieurs labels, dont aucun ne correspond ni à `football` ni à `tennis`

e. et se termine enfin avec un label commençant par `Russ` ou correspond strictement à `Spain`.

- `ltxtquery` représente en quelque sorte une recherche plein texte pour la correspondance de valeurs `ltree`. Une valeur `ltxtquery` contient des mots, quelque fois avec les modificateurs `@`, `*`, `%` à la fin ; les modifications ont la même signification que dans un `lquery`. Les mots peuvent être combinés avec `&` (AND), `|` (OR), `!` (NOT) et des parenthèses. La différence clé d'une `lquery` est que `ltxtquery` établit une correspondance avec des mots sans relation avec leur position dans le chemin de labels.

Voici un exemple de `ltxtquery` :

```
Europe & Russia*@ & !Transportation
```

Ceci établira une correspondance avec les chemins contenant le label `Europe` et tout label commençant par `Russia` (quelque soit la casse), mais pas les chemins contenant le label `Transportation`. L'emplacement de ces mots dans le chemin n'est pas important. De plus, quand `%` est utilisé, le mot peut établir une correspondance avec tout mot séparé par un tiret bas dans un label, quelque soit sa position.

Note : `ltxtquery` autorise un espace blanc entre des symboles mais `ltree` et `lquery` ne le permettent pas.

F.23.2. Opérateurs et fonctions

Le type `ltree` dispose des opérateurs de comparaison habituels `=`, `<>`, `<`, `>`, `<=`, `>=`. Les comparaisons trient dans l'ordre du parcours d'un arbre, avec les enfants d'un nœud trié par le texte du label. De plus, les opérateurs spécialisés indiqués dans Tableau F.13 sont disponibles.

Tableau F.13. Opérateurs `ltree`

Opérateur **Description**
`ltree @> ltree → boolean` L'argument gauche est-il un ancêtre de l'argument droit (ou identique) ?
`ltree <@ ltree → boolean` L'argument gauche est-il un descendant de l'argument droit (ou identique) ?
`ltree ~ lquery → boolean` `lquery ~ ltree → boolean` Est-ce que `ltree` établit une correspondance avec `lquery` ??
`ltree ? lquery[] → boolean` `lquery[] ? ltree → boolean` Est-ce que `ltree` établit une correspondance avec au moins un `lquery` dans ce tableau ?
`ltree @ ltxtquery → boolean` `ltxtquery @ ltree → boolean` Est-ce que `ltree` établit une correspondance avec `ltxtquery` ?
`ltree || ltree → ltree` Concatène des chemins `ltree`.
`ltree || text → ltree` `text || ltree → ltree` Convertit du texte en `ltree` et concatène.
`ltree[] @> ltree → boolean` `ltree <@ ltree[] → boolean` Est-ce que le tableau contient un ancêtre de `ltree` ?
`ltree[] <@ ltree → boolean` `ltree @> ltree[] → boolean`

Opérateur **Description**
Est-ce que le tableau contient un descendant de `ltree` ??
`ltree[]` ~ `lquery` → `boolean` `lquery` ~ `ltree[]` → `boolean` Est-ce que le tableau contient au moins un chemin correspondant à `lquery` ??
`ltree[]` ? `lquery[]` → `boolean` `lquery[]` ? `ltree[]` → `boolean` Est-ce que le tableau `ltree` contient au moins un chemin correspondant à un `lquery` ?
`ltree[]` @ `ltxtquery` → `boolean` `ltxtquery` @ `ltree[]` → `boolean` Est-ce que le tableau contient au moins un chemin correspondant à `ltxtquery` ??
`ltree[]` ?@> `ltree` → `ltree` Renvoie la première entrée du tableau ancêtre de `ltree` ou NULL sinon.
`ltree[]` ?<@ `ltree` → `ltree` Renvoie la première entrée du tableau qui est un descendant de `ltree` ou NULL sinon.
`ltree[]` ?~ `lquery` → `ltree` Renvoie la première entrée du tableau établissant une correspondance avec `lquery` et NULL sinon.
`ltree[]` ?@ `ltxtquery` → `ltree` Renvoie la première entrée du tableau établissant une correspondance avec `ltxtquery` et NULL sinon.

Les opérateurs operators <@, @>, @ et ~ ont des versions analogues ^<@, ^@>, ^@, ^~, qui sont identiques sauf qu'elles n'utilisent pas les index. Elles sont utiles pour tester.

Les fonctions disponibles sont indiquées dans Tableau F.14.

Tableau F.14. Fonctions `ltree`

Fonction **Description** **Exemple(s)**
`subltree` (`ltree`, *`start`* `integer`, *`end`* `integer`) → `ltree` Retourne le sous-chemin du `ltree` à partir de la position *`start`* jusqu'à la position *`end`*-1 (comptée à partir de 0). `subltree('Top.Child1.Child2', 1, 2)` → `Child1`
`subpath` (`ltree`, *`offset`* `integer`, *`len`* `integer`) → `ltree` Retourne le sous-ensemble du `ltree` en commençant à la position *`offset`*, avec une longueur *`len`*. Si *`offset`* est négatif, le sous-ensemble débute depuis la fin de chemin et s'étend vers le début. Si *`len`* est négatif, cela élimine ce nombre (en valeur absolue) de labels depuis la fin du chemin. `subpath('Top.Child1.Child2', 0, 2)` → `Top.Child1`
`subpath` (`ltree`, *`offset`* `integer`) → `ltree` Retourne le sous-ensemble du `ltree` en commençant à la position *`offset`*, avec une longueur *`len`*. Si *`offset`* est négatif, le sous-ensemble débute depuis la fin de chemin et s'étend vers le début. `subpath('Top.Child1.Child2', 1)` → `Child1.Child2`

Fonction **Description** **Exemple(s)**
`nlevel` (`ltree`) → `integer` Retourne le nombre de label dans le chemin. `nlevel('Top.Child1.Child2')` → `3`
`index` (*a* `ltree`, *b* `ltree`) → `integer` Retourne la position de la première occurrence de *b* dans *a*, ou -1 si non trouvé. `index('0.1.2.3.5.4.5.6.8.5.6.8', '5.6')` → `6`
`index` (*a* `ltree`, *b* `ltree`, *offset* `integer`) → `integer` Retourne la position de la première occurrence de *b* dans *a*, ou -1 si non trouvé. La recherche débute à la position *offset* ; un *offset* négatif indique un départ à *-offset* label depuis la fin du chemin. `index('0.1.2.3.5.4.5.6.8.5.6.8', '5.6', -4)` → `9`
`text2ltree` (`text`) → `ltree` Convertit un `text` en `ltree`.
`ltree2text` (`ltree`) → `text` Convertit un `ltree` en `text`.
`lca` (`ltree` [, `ltree` [, ...]]) → `ltree` Calcule le plus long ancêtre commun des chemins. (8 arguments sont supportés au maximum). `lca('1.2.3', '1.2.3.4.5.6')` → `1.2`
`lca` (`ltree[]`) → `ltree` Calcule le plus long ancêtre commun des chemins dans un tableau. `lca(array['1.2.3'::ltree, '1.2.3.4'])` → `1.2`

F.23.3. Index

`ltree` accepte différents types d'index pouvant améliorer les performances des opérateurs indiqués :

- Index B-tree sur `ltree` : <, <=, =, >=, >
- Index GiST sur `ltree` (classe d'opérateurs `gist_ltree_ops`) : <, <=, =, >=, >, @>, <@, @, ~, ?

  La classe d'opérateurs GiST `gist_ltree_ops` effectue une approximation sur un ensemble de chemins de labels sous format de signature bitmap. La taille de la signature par défaut est 8 octets. La longueur doit être un multiple positif de l'alignement d'un type `int` (4 octets sur la plupart des machines) jusqu'à 2024. Des tailles de signatures plus longues permettent une recherche plus précise (en parcourant une fraction plus petite de l'index et moins de pages heap), au coût d'un plus large index.

  Exemple de création d'un tel index avec une taille de signature par défaut de 8 octets :

```
CREATE INDEX path_gist_idx ON test USING GIST (path);
```

  Exemple de création d'un tel index avec une taille de signature de 100 octets :

```
CREATE INDEX path_gist_idx ON test USING GIST (path
 gist_ltree_ops(siglen=100));
```

- Index GiST sur `ltree[]` (classe d'opérateurs `gist__ltree_ops` : `ltree[] <@ ltree`, `ltree @> ltree[]`, `@`, `~`, `?`

  La classe d'opérateurs GiST `gist__ltree_ops` fonctionne de façon similaire à `gist_ltree_ops` et aussi prend une taille de signature en paramètre. La valeur par défaut de `siglen` dans `gist__ltree_ops` est 28 octets.

  Exemple de création d'un tel index avec la taille de signature par défaut de 28 octets :

```
CREATE INDEX path_gist_idx ON test USING GIST (array_path);
```

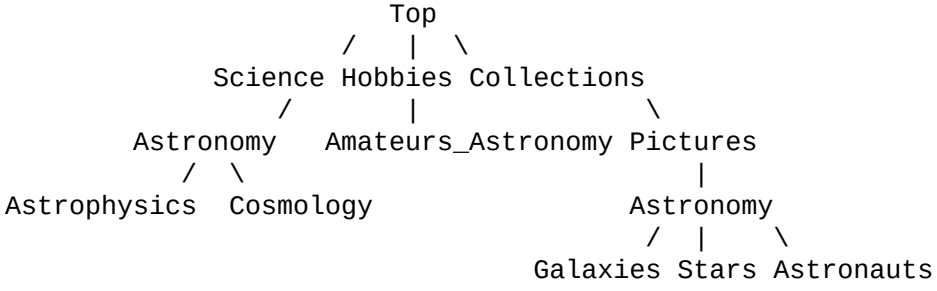
  Note : ce type d'index est à perte.

F.23.4. Exemple

Cet exemple utilise les données suivantes (disponibles dans le fichier `contrib/ltree/ltreetest.sql` des sources) :

```
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES
 ('Top.Collections.Pictures.Astronomy.Stars');
INSERT INTO test VALUES
 ('Top.Collections.Pictures.Astronomy.Galaxies');
INSERT INTO test VALUES
 ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING GIST (path);
CREATE INDEX path_idx ON test USING BTREE (path);
```

Maintenant, nous avons une table `test` peuplée avec des données décrivant la hiérarchie ci-dessous :



Nous pouvons faire de l'héritage :

```
ltreetest=> SELECT path FROM test WHERE path <@ 'Top.Science';
                path
-----------------------------------
 Top.Science
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(4 rows)
```

Voici quelques exemples de correspondance de chemins :

```
ltreetest=> SELECT path FROM test WHERE path ~ '*.Astronomy.*';
                     path
-----------------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Collections.Pictures.Astronomy
 Top.Collections.Pictures.Astronomy.Stars
 Top.Collections.Pictures.Astronomy.Galaxies
 Top.Collections.Pictures.Astronomy.Astronauts
(7 rows)

ltreetest=> SELECT path FROM test WHERE path ~ '*.!
pictures@.Astronomy.*';
                path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

Voici quelques exemples de recherche plein texte :

```
ltreetest=> SELECT path FROM test WHERE path @ 'Astro*% & !
pictures@';
                path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Hobbies.Amateurs_Astronomy
(4 rows)

ltreetest=> SELECT path FROM test WHERE path @ 'Astro* & !
pictures@';
                path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

Construction d'un chemin en utilisant les fonctions :

```
ltreetest=> SELECT subpath(path,0,2)||'Space'||subpath(path,2) FROM
 test WHERE path <@ 'Top.Science.Astronomy';
                ?column?
----------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

Nous pouvons simplifier ceci en créant une fonction SQL qui insère un label à une position spécifiée dans un chemin :

```
CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
    AS 'select subpath($1,0,$2) || $3 || subpath($1,$2);'
    LANGUAGE SQL IMMUTABLE;

ltreetest=> SELECT ins_label(path,2,'Space') FROM test WHERE path
 <@ 'Top.Science.Astronomy';
               ins_label
----------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

F.23.5. Transformations

L'extension `ltree_plpython3u` implémente les transformations du type `ltree` pour PL/Python. Si elle est installée et indiquée lors de la création d'une fonction, les valeurs `ltree` sont converties en listes Python. Il est à noter que l'inverse n'est pas encore supportée.

> **Attention**
>
> Il est fortement recommandé que l'extension de transformation soit installée dans le même schéma que `ltree`. Sinon il existe un risque au moment de l'installation si le schéma de l'extension de transformation contient des objets définis par un utilisateur hostile.

F.23.6. Auteurs

Tout le travail a été réalisé par Teodor Sigaev (`<teodor@stack.net>`) et Oleg Bartunov (`<oleg@sai.msu.su>`). Voir http://www.sai.msu.su/~megera/postgres/gist pour des informations supplémentaires. Les auteurs voudraient remercier Eugeny Rodichev pour son aide. Commentaires et rapports de bogue sont les bienvenus.

F.24. old_snapshot

Le module `old_snapshot` permet la vérification de l'état du serveur qui est utilisé pour implémenter old_snapshot_threshold.

F.24.1. Fonctions

`pg_old_snapshot_time_mapping(array_offset OUT int4, end_timestamp OUT timestamptz, newest_xmin OUT xid) returns setof record`

Renvoie toutes les entrées de correspondance entre l'horodatage du serveur et les XID. Chaque entrée représente le xmin le plus récent de tout snapshot pris dans la minute correspondante.

F.25. pageinspect

Le module `pageinspect` fournit des fonctions qui vous permettent d'inspecter le contenu des pages de la base de données à un bas niveau, ce qui est utile pour le débogage. Toutes ces fonctions ne sont utilisables que par les superutilisateurs.

F.25.1. Fonctions générales

`get_raw_page(relname text, fork text, blkno bigint) returns bytea`

`get_raw_page` lit le bloc spécifié de la relation nommée et renvoie une copie en tant que valeur de type `bytea`. Ceci permet la récupération de la copie cohérente à un instant t d'un bloc spécifique. *`fork`* devrait être `'main'` pour les données, et `'fsm'` pour la carte des espaces libres, `'vm'` pour la carte de visibilité ou `'init'` pour le fichier d'initialisation.

`get_raw_page(relname text, blkno bigint) returns bytea`

Une version raccourcie de `get_raw_page`, pour ne lire que la partie des données. Équivalent à `get_raw_page(relname, 'main', blkno)`.

`page_header(page bytea) returns record`

`page_header` affiche les champs communs à toutes les pages des tables et index PostgreSQL.

L'image d'une page obtenu avec `get_raw_page` doit être passé en argument. Par exemple :

```
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
    lsn    | checksum | flags  | lower | upper | special |
 pagesize | version | prune_xid
-----------+----------+--------+-------+-------+---------
+----------+---------+-----------
 0/24A1B50 |        0 |      1 |   232 |   368 |    8192 |
 8192 |       4 |         0
```

Les colonnes renvoyées correspondent aux champs de la structure `PageHeaderData`. Voir `src/include/storage/bufpage.h` pour les détails.

Le champ `checksum` est la somme de contrôle stockée dans la page, qui peut être incorrecte si la page est d'une manière ou d'une autre corrompue. Si les sommes de contrôle des données ne sont pas activées pour cette instance, alors la valeur stockée n'a aucune signification.

`page_checksum(page bytea, blkno bigint4) returns smallint`

`page_checksum` calcule la somme de contrôle pour la page, comme si elle était citée au numéro de bloc passé en paramètre.

Une image de page obtenue avec `get_raw_page` devrait être passée en argument. Par exemple :

```
test=# SELECT page_checksum(get_raw_page('pg_class', 0), 0);
```

```
 page_checksum
---------------
         13443
```

Veuillez noter que la somme de contrôle dépend du numéro de bloc, ainsi des numéros de blocs qui se correspondent devraient être passé (sauf lors de débugage ésotériques).

Les sommes de contrôle calculées avec cette fonction peuvent être comparées avec le champ `checksum` retournée par la fonction `page_header`. Si les sommes de contrôle des données sont activées sur cette instance, alors les deux valeurs devraient être égales.

`fsm_page_contents(page bytea) returns text`

`fsm_page_contents` montre la structure interne du nœud d'une page FSM. Par exemple :

```
test=# SELECT fsm_page_contents(get_raw_page('pg_class', 'fsm',
 0));
```

La sortie est une chaîne de texte multiligne, avec une ligne par nœud dans l'arbre binaire au sein de la page. Seuls ces nœuds qui ne valent pas zéro sont affichés. Le pointeur appelé « next », qui pointe vers le prochain slot à être retourné depuis la page, est également affiché.

Voir `src/backend/storage/freespace/README` pour plus d'information sur la structure d'une page FSM.

F.25.2. Fonctions Heap

`heap_page_items(page bytea) returns setof record`

`heap_page_items` affiche tous les pointeurs de ligne dans une page de table. Pour les pointeurs de ligne en utilisation, les en-têtes de ligne ainsi que les données des lignes sont aussi affichées. Toutes les lignes sont affichées, qu'elles soient ou non visibles dans l'image MVCC au moment où la page brute a été copiée.

Une image d'une page de table obtenue avec `get_raw_page` doit être fournie en argument. Par exemple :

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class',
 0));
```

Voir `src/include/storage/itemid.h` et `src/include/access/htup_details.h` pour des explications sur les champs renvoyés.

La fonction `heap_tuple_infomask_flags` peut être utilisée pour déchiffrer les drapeaux de `t_infomask` et `t_infomask2` pour les lignes de la table.

`tuple_data_split(rel_oid oid, t_data bytea, t_infomask integer, t_infomask2 integer, t_bits text [, do_detoast bool]) returns bytea[]`

`tuple_data_split` divise les données d'une ligne en attributs de la même façon que le fait le moteur.

```
test=# SELECT tuple_data_split('pg_class'::regclass,
 t_data, t_infomask, t_infomask2, t_bits) FROM
 heap_page_items(get_raw_page('pg_class', 0));
```

Cette fonction doit être appelée avec les mêmes arguments que ce qui est renvoyé par `heap_page_items`.

Si *`do_detoast`* est `true`, la donnée TOAST sera traitée. La valeur par défaut est `false`.

`heap_page_item_attrs(page bytea, rel_oid regclass [, do_detoast bool]) returns setof record`

`heap_page_item_attrs` est équivalent à `heap_page_items` sauf qu'elle renvoie les données brutes de la ligne sous la forme d'un tableau d'attributs pouvant en option être traités par *`do_detoast`* (comportement désactivé par défaut).

Une image de la page HEAP obtenue avec `get_raw_page` doit être fournie comme argument. Par exemple :

```
test=# SELECT * FROM
 heap_page_item_attrs(get_raw_page('pg_class', 0),
 'pg_class'::regclass);
```

`heap_tuple_infomask_flags(t_infomask integer, t_infomask2 integer) returns record`

`heap_tuple_infomask_flags` décode les champs `t_infomask` et `t_infomask2` renvoyés par `heap_page_items` en un ensemble de tableaux composés de noms de drapeau, lisibles par un humain, avec une colonne pour tous les drapeaux et une colonne pour les drapeaux composés. Par exemple :

```
test=# SELECT t_ctid, raw_flags, combined_flags
         FROM heap_page_items(get_raw_page('pg_class', 0)),
           LATERAL heap_tuple_infomask_flags(t_infomask,
 t_infomask2)
         WHERE t_infomask IS NOT NULL OR t_infomask2 IS NOT
 NULL;
```

Cette fonction doit être appelée avec les mêmes arguments que les attributes de retour de `heap_page_items`.

Les drapeaux combinés sont affichés pour les macros de niveau source prenant en compte la valeur de plus d'un bit, comme par exemple `HEAP_XMIN_FROZEN`.

Voir `src/include/access/htup_details.h` pour des explications sur les noms des drapeaux renvoyés.

F.25.3. Fonctions B-tree

`bt_metap(relname text) returns record`

`bt_metap` renvoie des informations sur une méta-page d'un index B-tree. Par exemple :

```
test=# SELECT * FROM bt_metap('pg_cast_oid_index');
-[ RECORD 1 ]-------------+-------
magic                     | 340322
version                   | 4
root                      | 1
level                     | 0
```

```
fastroot                 | 1
fastlevel                | 0
last_cleanup_num_delpages | 0
last_cleanup_num_tuples   | 230
allequalimage            | f
```

`bt_page_stats(relname text, blkno bigint) returns record`

`bt_page_stats` renvoie un résumé des informations sur les pages enfants des index B-tree. Par exemple :

```
test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
-[ RECORD 1 ]-+-----
blkno         | 1
type          | l
live_items    | 224
dead_items    | 0
avg_item_size | 16
page_size     | 8192
free_size     | 3668
btpo_prev     | 0
btpo_next     | 0
btpo_level    | 0
btpo_flags    | 3
```

`bt_page_items(relname text, blkno bigint) returns setof record`

`bt_page_items` renvoie des informations détaillées sur tous les éléments d'une page d'index B-tree. Par exemple :

```
test=# SELECT itemoffset, ctid, itemlen, nulls, vars, data,
 dead, htid, tids[0:2] AS some_tids
        FROM bt_page_items('tenk2_hundred', 5);
 itemoffset |   ctid    | itemlen | nulls | vars |          data
           | dead |  htid  |      some_tids
------------+-----------+---------+-------+------
+-------------------------+------+--------+---------------------
          1 | (16,1)    |      16 | f     | f    | 30 00 00 00
 00 00 00 00 |       |         |
          2 | (16,8292) |     616 | f     | f    | 24 00 00 00
 00 00 00 00 | f     | (1,6)   | {"(1,6)","(10,22)"}
          3 | (16,8292) |     616 | f     | f    | 25 00 00 00
 00 00 00 00 | f     | (1,18)  | {"(1,18)","(4,22)"}
          4 | (16,8292) |     616 | f     | f    | 26 00 00 00
 00 00 00 00 | f     | (4,18)  | {"(4,18)","(6,17)"}
          5 | (16,8292) |     616 | f     | f    | 27 00 00 00
 00 00 00 00 | f     | (1,2)   | {"(1,2)","(1,19)"}
          6 | (16,8292) |     616 | f     | f    | 28 00 00 00
 00 00 00 00 | f     | (2,24)  | {"(2,24)","(4,11)"}
          7 | (16,8292) |     616 | f     | f    | 29 00 00 00
 00 00 00 00 | f     | (2,17)  | {"(2,17)","(11,2)"}
          8 | (16,8292) |     616 | f     | f    | 2a 00 00 00
 00 00 00 00 | f     | (0,25)  | {"(0,25)","(3,20)"}
          9 | (16,8292) |     616 | f     | f    | 2b 00 00 00
 00 00 00 00 | f     | (0,10)  | {"(0,10)","(0,14)"}
```

```
         10 | (16,8292) |     616 | f     | f    | 2c 00 00 00
 00 00 00 00 | f    | (1,3)  | {"(1,3)","(3,9)"}
         11 | (16,8292) |     616 | f     | f    | 2d 00 00 00
 00 00 00 00 | f    | (6,28) | {"(6,28)","(11,1)"}
         12 | (16,8292) |     616 | f     | f    | 2e 00 00 00
 00 00 00 00 | f    | (0,27) | {"(0,27)","(1,13)"}
         13 | (16,8292) |     616 | f     | f    | 2f 00 00 00
 00 00 00 00 | f    | (4,17) | {"(4,17)","(4,21)"}
(13 rows)
```

Ceci est une page feuille B-tree. Tous les enregistrements qui pointent vers la table sont des éléments de la liste d'éléments (qui, chacune, stocke un total de 100 TID de 6 octets). Il existe aussi un enregistrement « clé haute » sur l'`itemoffset` numéro 1. `ctid` est utilisé pour enregistrer des informations codées sur chaque enregistrement dans cet exemple, bien que les enregistrements des pages feuilles enregistrent souvent directement le TID de la table dans le champ `ctid`. `tids` est la liste de TID enregistrée sous la forme d'une liste d'éléments.

Dans un bloc interne (non affiché), la partie du numéro de bloc dans le champ `ctid` esy un lien inférieur (« downlink »), qui est un numéro de bloc vers un autre bloc de l'index lui-même. La partie décalage (*offset*, le deuxième nombre) du champ `ctid` enregistre des informations codées sur l'enregistrement, telle que le nombre de colonnes présentes (le troncage du suffixe pourrait avoir supprimé des colonnes de suffixe inutiles). Les colonnes tronquées sont traitées comme ayant la valeur « moins infini ».

`htid` affiche un TID de table pour l'enregistrement, quelque soit la représentation sous-jacente de l'enregistrement. Cette valeur peut correspondre à un `ctid` ou peut être décodée parmi les représentations alternatives utilisées par les enregistrements des listes d'éléments et par les enregistrements des pages internes. Les enregistrements dans les pages internes ont habituellement la colonne TID de la table tronquée, ce qui est représenté comme une valeur `htid` NULL.

Notez que le premier élément une page (autre que la dernière, toute page avec une valeur différente de zéro dans le champ `btpo_next`) est la « clé haute » du bloc, ce qui signifie que ces données (`data`) serve comme limite haute de tous les éléments apparaissant sur la page, alors que son champ `ctid` ne pointe pas vers un autre bloc. De plus, sur les pages internes, le premier véritable élément de donnée (le premier élément qui n'est pas une clé haute) a chaque colonne tronquée, laissant aucune valeur réelle dans son champ `data`. Néanmoins, un tel élément n'a pas de lien inférieur valide dans son champ `ctid`.

Pour plus de détails sur la structure des index B-tree, voir Section 67.4.1. Pour plus de détails sur la déduplication et les listes d'éléments, voir Section 67.4.3.

`bt_page_items(page bytea) returns setof record`

Il est également possible de passer une page à `bt_page_items` comme une valeur de type `bytea`. Une image de page obtenue avec `get_raw_page` devrait être passé en argument. Ainsi le précédent exemple pourrait également être réécrit ainsi :

```
test=# SELECT itemoffset, ctid, itemlen, nulls, vars, data,
 dead, htid, tids[0:2] AS some_tids
        FROM bt_page_items(get_raw_page('tenk2_hundred', 5));
 itemoffset |   ctid    | itemlen | nulls | vars |          data
           | dead |  htid  |      some_tids
------------+-----------+---------+-------+------
+-------------------------+------+--------+---------------------
          1 | (16,1)    |      16 | f     | f    | 30 00 00 00
 00 00 00 00 |      |        |
```

```
          2 | (16,8292) |     616 | f     | f    | 24 00 00 00
 00 00 00 00 | f    | (1,6)  | {"(1,6)","(10,22)"}
          3 | (16,8292) |     616 | f     | f    | 25 00 00 00
 00 00 00 00 | f    | (1,18) | {"(1,18)","(4,22)"}
          4 | (16,8292) |     616 | f     | f    | 26 00 00 00
 00 00 00 00 | f    | (4,18) | {"(4,18)","(6,17)"}
          5 | (16,8292) |     616 | f     | f    | 27 00 00 00
 00 00 00 00 | f    | (1,2)  | {"(1,2)","(1,19)"}
          6 | (16,8292) |     616 | f     | f    | 28 00 00 00
 00 00 00 00 | f    | (2,24) | {"(2,24)","(4,11)"}
          7 | (16,8292) |     616 | f     | f    | 29 00 00 00
 00 00 00 00 | f    | (2,17) | {"(2,17)","(11,2)"}
          8 | (16,8292) |     616 | f     | f    | 2a 00 00 00
 00 00 00 00 | f    | (0,25) | {"(0,25)","(3,20)"}
          9 | (16,8292) |     616 | f     | f    | 2b 00 00 00
 00 00 00 00 | f    | (0,10) | {"(0,10)","(0,14)"}
         10 | (16,8292) |     616 | f     | f    | 2c 00 00 00
 00 00 00 00 | f    | (1,3)  | {"(1,3)","(3,9)"}
         11 | (16,8292) |     616 | f     | f    | 2d 00 00 00
 00 00 00 00 | f    | (6,28) | {"(6,28)","(11,1)"}
         12 | (16,8292) |     616 | f     | f    | 2e 00 00 00
 00 00 00 00 | f    | (0,27) | {"(0,27)","(1,13)"}
         13 | (16,8292) |     616 | f     | f    | 2f 00 00 00
 00 00 00 00 | f    | (4,17) | {"(4,17)","(4,21)"}
(13 rows)
```

Tous les autres détails sont les même qu'expliqué au point précédent.

F.25.4. Fonctions BRIN

`brin_page_type(page bytea) returns text`

`brin_page_type` renvoie le type de bloc du bloc indiqué pour l'index BRIN donné ou renvoie une erreur si le bloc n'est pas un bloc valide d'un index BRIN. Par exemple :

```
test=# SELECT brin_page_type(get_raw_page('brinidx', 0));
 brin_page_type
----------------
 meta
```

`brin_metapage_info(page bytea) returns record`

`brin_metapage_info` renvoie une information assortie sur la métapage d'un index BRIN. Par exemple :

```
test=# SELECT * FROM brin_metapage_info(get_raw_page('brinidx',
 0));
   magic    | version | pagesperrange | lastrevmappage
------------+---------+---------------+----------------
 0xA8109CFA |       1 |             4 |              2
```

`brin_revmap_data(page bytea) returns setof tid`

`brin_revmap_data` renvoie la liste des identifiants de lignes dans un bloc de type « range map » d'un index BRIN. Par exemple :

```
test=# SELECT * FROM brin_revmap_data(get_raw_page('brinidx',
 2)) LIMIT 5;
  pages
---------
 (6,137)
 (6,138)
 (6,139)
 (6,140)
 (6,141)
```

```
brin_page_items(page bytea, index oid) returns setof record
```

`brin_page_items` renvoie les données enregistrées dans le bloc de données de l'index BRIN. Par exemple :

```
test=# SELECT * FROM brin_page_items(get_raw_page('brinidx', 5),
                                     'brinidx')
       ORDER BY blknum, attnum LIMIT 6;
 itemoffset | blknum | attnum | allnulls | hasnulls |
 placeholder |    value
------------+--------+--------+----------+----------
+-------------+--------------
         137 |      0 |      1 | t        | f        | f
  |
         137 |      0 |      2 | f        | f        | f
  | {1 .. 88}
         138 |      4 |      1 | t        | f        | f
  |
         138 |      4 |      2 | f        | f        | f
  | {89 .. 176}
         139 |      8 |      1 | t        | f        | f
  |
         139 |      8 |      2 | f        | f        | f
  | {177 .. 264}
```

Les colonnes renvoyées correspondent aux champs des structures `BrinMemTuple` et `BrinValues`. Voir `src/include/access/brin_tuple.h` pour plus de détails.

F.25.5. Fonctions GIN

```
gin_metapage_info(page bytea) returns record
```

`gin_metapage_info` renvoie des informations sur la métapage d'un index GIN index metapage. Par exemple :

```
test=# SELECT * FROM gin_metapage_info(get_raw_page('gin_index',
 0));
-[ RECORD 1 ]----+-----------
pending_head     | 4294967295
pending_tail     | 4294967295
tail_free_size   | 0
n_pending_pages  | 0
n_pending_tuples | 0
n_total_pages    | 7
```

```
   n_entry_pages    | 6
   n_data_pages     | 0
   n_entries        | 693
   version          | 2
```

`gin_page_opaque_info(page bytea) returns record`

`gin_page_opaque_info` renvoie des informations sur la partie opaque d'un index GIN, comme le type de bloc. Par exemple :

```
   test=# SELECT * FROM
    gin_page_opaque_info(get_raw_page('gin_index', 2));
    rightlink | maxoff |         flags
   -----------+--------+-----------------------
            5 |      0 | {data,leaf,compressed}
   (1 row)
```

`gin_leafpage_items(page bytea) returns setof record`

`gin_leafpage_items` renvoie des informations sur les données enregistrées dans un bloc feuille d'un index GIN. Par exemple :

```
   test=# SELECT first_tid, nbytes, tids[0:5] AS some_tids
           FROM gin_leafpage_items(get_raw_page('gin_test_idx',
    2));
    first_tid | nbytes |                        some_tids
   -----------+--------
   +-----------------------------------------------------------
    (8,41)    |    244 |
    {"(8,41)","(8,43)","(8,44)","(8,45)","(8,46)"}
    (10,45)   |    248 |
    {"(10,45)","(10,46)","(10,47)","(10,48)","(10,49)"}
    (12,52)   |    248 |
    {"(12,52)","(12,53)","(12,54)","(12,55)","(12,56)"}
    (14,59)   |    320 |
    {"(14,59)","(14,60)","(14,61)","(14,62)","(14,63)"}
    (167,16)  |    376 |
    {"(167,16)","(167,17)","(167,18)","(167,19)","(167,20)"}
    (170,30)  |    376 |
    {"(170,30)","(170,31)","(170,32)","(170,33)","(170,34)"}
    (173,44)  |    197 |
    {"(173,44)","(173,45)","(173,46)","(173,47)","(173,48)"}
   (7 rows)
```

F.25.6. Fonctions GiST

`gist_page_opaque_info(page bytea) returns record`

`gist_page_opaque_info` renvoie des informations à partir de l'aire opaque d'un bloc d'index GiST, tel que le NSN, le lien droit et le type de bloc. Par exemple :

```
   test=# SELECT * FROM
    gist_page_opaque_info(get_raw_page('test_gist_idx', 2));
```

```
 lsn | nsn | rightlink | flags
-----+-----+-----------+--------
 0/1 | 0/0 |         1 | {leaf}
(1 row)
```

```
gist_page_items(page bytea, index_oid regclass) returns setof record
```

`gist_page_items` renvoie des informations sur les données enregistrées dans un bloc d'un index GiST. Par exemple :

```
test=# SELECT * FROM
 gist_page_items(get_raw_page('test_gist_idx', 0),
 'test_gist_idx');
 itemoffset |   ctid    | itemlen | dead |             keys
------------+-----------+---------+------
+------------------------------
          1 | (1,65535) |      40 | f    | (p)=("(166,166),
(1,1)")
          2 | (2,65535) |      40 | f    | (p)=("(332,332),
(167,167)")
          3 | (3,65535) |      40 | f    | (p)=("(498,498),
(333,333)")
          4 | (4,65535) |      40 | f    | (p)=("(664,664),
(499,499)")
          5 | (5,65535) |      40 | f    | (p)=("(830,830),
(665,665)")
          6 | (6,65535) |      40 | f    | (p)=("(996,996),
(831,831)")
          7 | (7,65535) |      40 | f    | (p)=("(1000,1000),
(997,997)")
(7 rows)
```

```
gist_page_items_bytea(page bytea) returns setof record
```

Identique à `gist_page_items`, mais renvoie la donnée clé sous la forme d'un objet brut de type `bytea`. Comme il ne tente pas de décoder la clé, il n'a pas besoin de connaître l'index impliqué. Par exemple :

```
test=# SELECT * FROM
 gist_page_items_bytea(get_raw_page('test_gist_idx', 0));
 itemoffset |   ctid    | itemlen | dead |
               key_data
------------+-----------+---------+------
+----------------------------------------
----------------------------------------
          1 | (1,65535) |      40 | f    |
 \x00000100ffff28000000000000c0644000000000
00c0644000000000000f03f000000000000f03f
          2 | (2,65535) |      40 | f    |
 \x00000200ffff28000000000000c0744000000000
00c0744000000000000e064400000000000e06440
          3 | (3,65535) |      40 | f    |
 \x00000300ffff28000000000000207f4000000000
00207f400000000000d074400000000000d07440
          4 | (4,65535) |      40 | f    |
 \x00000400ffff28000000000000c0844000000000
00c0844000000000307f400000000000307f40
```

```
          5 | (5,65535) |      40 | f    |
 \x00000500ffff28000000000000f0894000000000
00f089400000000000c8844000000000c88440
          6 | (6,65535) |      40 | f    |
 \x00000600ffff28000000000000208f4000000000
00208f400000000000f8894000000000f88940
          7 | (7,65535) |      40 | f    |
 \x00000700ffff28000000000000408f4000000000
00408f400000000000288f400000000000288f40
(7 rows)
```

F.25.7. Fonctions Hash

`hash_page_type(page bytea) returns text`

`hash_page_type` renvoie le type de page de la page d'index HASH donné. Par exemple :

```
test=# SELECT hash_page_type(get_raw_page('con_hash_index', 0));
 hash_page_type
----------------
 metapage
```

`hash_page_stats(page bytea) returns setof record`

`hash_page_stats` retourne des informations sur un bucket ou une page overflow d'un index HASH. Par exemple :

```
test=# SELECT * FROM
 hash_page_stats(get_raw_page('con_hash_index', 1));
-[ RECORD 1 ]---+-----------
live_items      | 407
dead_items      | 0
page_size       | 8192
free_size       | 8
hasho_prevblkno | 4096
hasho_nextblkno | 8474
hasho_bucket    | 0
hasho_flag      | 66
hasho_page_id   | 65408
```

`hash_page_items(page bytea) returns setof record`

`hash_page_items` retourne des informations sur les données stockées dans un bucket ou une page overflow d'une page d'index HASH. Par exemple :

```
test=# SELECT * FROM
 hash_page_items(get_raw_page('con_hash_index', 1)) LIMIT 5;
 itemoffset |   ctid    |    data
------------+-----------+------------
          1 | (899,77)  | 1053474816
          2 | (897,29)  | 1053474816
          3 | (894,207) | 1053474816
          4 | (892,159) | 1053474816
          5 | (890,111) | 1053474816
```

```
hash_bitmap_info(index oid, blkno bigint) returns record
```

`hash_bitmap_info` montre le statut d'un bit dans la page de bitmap pour une page overflow en particulier d'un index HASH. Par exemple :

```
test=# SELECT * FROM hash_bitmap_info('con_hash_index', 2052);
 bitmapblkno | bitmapbit | bitstatus
-------------+-----------+-----------
          65 |         3 | t
```

```
hash_metapage_info(page bytea) returns record
```

`hash_metapage_info` retourne l'information stockée dans la méta-page d'un index HASH index. Par exemple :

```
test=# SELECT magic, version, ntuples, ffactor, bsize, bmsize,
 bmshift,
test-#     maxbucket, highmask, lowmask, ovflpoint, firstfree,
 nmaps, procid,
test-#     regexp_replace(spares::text, '(,0)*}', '}') as
 spares,
test-#     regexp_replace(mapp::text, '(,0)*}', '}') as mapp
test-# FROM hash_metapage_info(get_raw_page('con_hash_index',
 0));
-[ RECORD
 1 ]----------------------------------------------------------------------------
magic     | 105121344
version   | 4
ntuples   | 500500
ffactor   | 40
bsize     | 8152
bmsize    | 4096
bmshift   | 15
maxbucket | 12512
highmask  | 16383
lowmask   | 8191
ovflpoint | 28
firstfree | 1204
nmaps     | 1
procid    | 450
spares    |
 {0,0,0,0,0,0,1,1,1,1,1,1,1,3,4,4,4,45,55,58,59,508,567,628,704,1193,1202,
mapp      | {65}
```

F.26. passwordcheck

Le module `passwordcheck` vérifie les mots de passe des utilisateurs quand ils sont configurés avec CREATE ROLE ou ALTER ROLE. Si un mot de passe est considéré trop faible, il sera rejeté et la commande se terminera avec une erreur.

Pour activer ce module, ajoutez `'$libdir/passwordcheck'` dans le paramètre shared_preload_libraries du fichier `postgresql.conf`, puis redémarrez le serveur.

Vous pouvez adapter ce module à vos besoins en modifiant son code source. Par exemple, vous pouvez utiliser CrackLib[3] pour vérifier les mots de passe -- ceci requiert seulement la suppression des commentaires sur deux lignes du `Makefile` et la reconstruction du module. (Nous ne pouvons pas inclure CrackLib par défaut pour des raisons de licence.) Sans CrackLib, le module impose quelques règles simples sur la force du mot de passe, que vous pouvez modifier ou étendre au besoin.

> **Attention**
>
> Pour empêcher l'envoi en clair des mots de passe sur le réseau, leur écriture dans les journaux applicatifs ou leur récupération par un administrateur de bases de données, PostgreSQL permet à l'utilisateur de fournir des mots de passe déjà chiffrés. Beaucoup de programmes clients utilisent cette fonctionnalité et chiffrent le mot de passe avant de l'envoyer au serveur.
>
> Ceci limite l'utilité du module `passwordcheck` car, dans ce cas, il peut seulement tenter de deviner le mot de passe. Pour cette raison, `passwordcheck` n'est pas recommandé si vos besoins en sécurité sont importants. Il est plus intéressant d'utiliser une méthode d'authentification externe comme GSSAPI (voir Chapitre 21) plutôt que de se fier aux mots de passe internes.
>
> Une alternative est de modifier `passwordcheck` pour rejeter les mots de passe pré-chiffrés, mais forcer ainsi les utilisateurs à entrer leurs mots de passe en clair porte son propre lot de problèmes de sécurité.

F.27. pg_buffercache

Le module `pg_buffercache` fournit un moyen pour examiner ce qui se passe dans le cache partagé en temps réel.

Ce module propose une fonction C, `pg_buffercache_pages`, qui renvoie un ensemble d'enregistrements, ainsi qu'une vue, `pg_buffercache`, qui englobe la fonction pour une utilisation facilitée.

Par défaut, l'utilisation est restreinte aux superutilisateurs et aux rôles disposant des attributs du rôle `pg_monitor`. L'accès peut être accordé à d'autres rôle en utilisant `GRANT`.

F.27.1. La vue `pg_buffercache`

Voici la définition des colonnes exposées par la vue affichée dans Tableau F.15 :

Tableau F.15. Colonnes de `pg_buffercache`

Type de la colonne Description
`bufferid integer` ID, qui va de 1 à `shared_buffers`
`relfilenode oid` (references `pg_class.relfilenode`) Numéro filenode de la relation
`reltablespace oid` (references `pg_tablespace.oid`) OID du tablespace de la relation
`reldatabase oid` (references `pg_database.oid`) OID de la base de données de la relation

[3] https://github.com/cracklib/cracklib

Type de la colonne **Description**
`relforknumber` `smallint` Numéro du fork dans la relation ; voir `common/relpath.h`
`relblocknumber` `bigint` Numéro de page dans la relation
`isdirty` `boolean` Page modifiée ?
`usagecount` `smallint` Compteur d'accès clock-sweep
`pinning_backends` `integer` Nombre de processus serveur en accès sur ce bloc

Il y a une ligne pour chaque tampon dans le cache partagé. Les tampons inutilisés sont affichés avec des champs NULL sauf pour `bufferid`. Les catalogues systèmes partagés sont affichés comme appartenant à la base de données zéro.

Comme le cache est partagé par toutes les bases de données, il y aura des pages de relations n'appartenant pas à la base de données courante. Cela signifie qu'il pourrait y avoir des lignes sans correspondance dans `pg_class`, ou qu'il pourrait y avoir des jointures incorrectes. Si vous essayez une jointure avec `pg_class`, une bonne idée est de restreindre la jointure aux lignes ayant un `reldatabase` égal à l'OID de la base de données actuelle ou à zéro.

Comme des verrous du gestionnaire de tampons ne sont pas acquis pour copier les données d'état du tampon que la vue affichera, accéder à la vue `pg_buffercache` a moins d'impact sur l'activité normale du tampon, mais il ne fournit pas un ensemble cohérent de résultats sur tous les tampons. Néanmoins, nous nous assurons que l'information de chaque tampon est cohérente avec lui-même.

F.27.2. Affichage en sortie

```
 regression=# SELECT n.nspname, c.relname, count(*) AS buffers
              FROM pg_buffercache b JOIN pg_class c
              ON b.relfilenode = pg_relation_filenode(c.oid) AND
                 b.reldatabase IN (0, (SELECT oid FROM pg_database
                                       WHERE datname =
current_database()))
              JOIN pg_namespace n ON n.oid = c.relnamespace
              GROUP BY n.nspname, c.relname
              ORDER BY 3 DESC
              LIMIT 10;

 nspname   |        relname         | buffers
-----------+------------------------+---------
 public    | delete_test_table      |     593
 public    | delete_test_table_pkey |     494
 pg_catalog | pg_attribute           |     472
 public    | quad_poly_tbl          |     353
 public    | tenk2                  |     349
 public    | tenk1                  |     349
 public    | gin_test_idx           |     306
 pg_catalog | pg_largeobject         |     206
 public    | gin_test_tbl           |     188
 public    | spgist_text_tbl        |     182
 (10 rows)
```

F.27.3. Auteurs

Mark Kirkwood <markir@paradise.net.nz>

Suggestions de conception : Neil Conway <neilc@samurai.com>

Conseils pour le débogage : Tom Lane <tgl@sss.pgh.pa.us>

F.28. pgcrypto

Le module pgcrypto propose des fonctions de cryptographie pour PostgreSQL.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

pgcrypto requiert OpenSSL et ne sera pas installé si le support d'OpenSSL n'a pas été sélectionné lors de la construction de PostgreSQL.

F.28.1. Fonctions de hachage généralistes

F.28.1.1. digest()

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

Calcule un hachage binaire de *data*. *type* est l'algorithme utilisé. Les algorithmes standards sont md5, sha1, sha224, sha256, sha384 et sha512. De plus, tout algorithme digest que OpenSSL accepte est automatiquement pris en compte.

Si vous voulez en résultat une chaîne hexadécimale, utilisez encode() sur le résultat. Par exemple :

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
  SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

F.28.1.2. hmac()

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key bytea, type text) returns bytea
```

Calcule un MAC haché sur *data* avec la clé *key*. *type* est identique à digest().

C'est similaire à digest() mais le hachage peut être recalculé en connaissant seulement la clé. Ceci évite le scénario où quelqu'un modifie les données et le hachage en même temps.

Si la clé est plus grosse que le bloc haché, il sera tout d'abord haché puis le résultat sera utilisé comme clé.

F.28.2. Fonctions de hachage de mot de passe

Les fonctions crypt() et gen_salt() sont spécialement conçues pour hacher les mots de passe. crypt() s'occupe du hachage et gen_salt() prépare les paramètres de l'algorithme pour ça.

Les algorithmes de crypt() diffèrent des algorithmes de hachage habituels comme MD5 ou SHA1 :

1. Ils sont lents. Comme la quantité de données est petite, c'est le seul moyen de rendre difficile la découverte par la force des mots de passe.

2. Ils incluent une valeur aléatoire appelée sel (*salt* en anglais) avec le résultat, pour que les utilisateurs qui ont le même mot de passe puissent avoir des mots de passe chiffrés différents. C'est aussi une défense supplémentaire comme l'inversion de l'algorithme.

3. Ils incluent le type de l'algorithme dans le résultat pour que les mots de passe hachés avec différents algorithmes puissent co-exister.

4. Certains s'adaptent. Cela signifie que, une fois que les ordinateurs iront plus vite, vous pourrez configurer l'algorithme pour qu'il soit plus lent, ceci sans introduire d'incompatibilité avec les mots de passe existant.

Tableau F.16 liste les algorithmes supportés par la fonction `crypt()`.

Tableau F.16. Algorithmes supportés par `crypt()`

Algorithme	Longueur maximum du mot de passe	Adaptif ?	Bits sel	Longueur de la sortie	Description
`bf`	72	oui	128	60	Basé sur Blowfish, variante 2a
`md5`	unlimited	non	48	34	crypt() basé sur MD5
`xdes`	8	oui	24	20	DES étendu
`des`	8	non	12	13	crypt original UNIX

F.28.2.1. `crypt()`

```
crypt(password text, salt text) returns text
```

Calcule un hachage de mot de passe (*`password`*) d'après crypt(3) UN*X. Lors du stockage d'un nouveau mot de passe, vous devez utiliser la fonction `gen_salt()` pour générer un nouveau sel (*`salt`*). Lors de la vérification de mot de passe, passez la valeur hachée stockée *`salt`*, et testez si le résultat correspond à la valeur stockée.

Exemple d'ajout d'un nouveau mot de passe :

```
UPDATE ... SET pswhash = crypt('new password',
gen_salt('md5'));
```

Exemple d'authentification :

```
SELECT (pswhash = crypt('entered password', pswhash)) AS
pswmatch FROM ... ;
```

Ceci renvoie `true` si le mot de passe saisi est correct.

F.28.2.2. `gen_salt()`

```
gen_salt(type text [, iter_count integer ]) returns text
```

Génère une nouvelle valeur aléatoire sel pour son utilisation avec `crypt()`. La chaîne sel indique aussi à `crypt()` l'algorithme à utiliser.

Le paramètre *`type`* précise l'algorithme de hachage. Les types acceptés sont : `des`, `xdes`, `md5` et `bf`.

Le paramètre *`iter_count`* laisse l'utilisateur indiquer le nombre d'itération, pour les algorithmes qui en ont. Plus le nombre est important, plus le hachage du mot de passe prendra du temps, et du coup plus le craquage du mot de passe prendre du temps. Cela étant dit, un nombre trop important rend pratiquement impossible le calcul du hachage. Si le paramètre *`iter_count`* est omis, le nombre d'itération par défaut est utilisé. Les valeurs autorisées pour *`iter_count`* dépendent de l'algorithme et sont affichées dans Tableau F.17.

Tableau F.17. Nombre d'itération pour `crypt()`

Algorithme	Par défaut	Min	Max
`xdes`	725	1	16777215
`bf`	6	4	31

Pour `xdes`, il existe une limite supplémentaire qui fait que ce nombre doit être un nombre impair.

Pour utiliser un nombre d'itération approprié, pensez que la fonction crypt DES original a été conçu pour avoir la vitesse de quatre hachages par seconde sur le matériel de l'époque. Plus lent que quatre hachages par secondes casserait probablement la facilité d'utilisation. Plus rapide que cent hachages à la seconde est probablement trop rapide.

Tableau F.18 donne un aperçu de la lenteur relative de différents algorithmes de hachage. La table montre le temps que prendrait le calcul de toutes les combinaisons réalisables pour un mot de passe sur huit caractères, en supposant que le mot de passe contient soit que des lettres minuscules, soit des lettres minuscules et majuscules et des chiffres. Dans les entrées `crypt-bf`, le nombre après un slash est le paramètre *`iter_count`* de `gen_salt`.

Tableau F.18. Vitesse de l'algorithm de hachage

Algorithme	Hachages/sec	Pour `[a-z]`	Pour `[A-Za-z0-9]`	Durée par rapport à `md5 hash`
`crypt-bf/8`	1792	4 années	3927 années	100k
`crypt-bf/7`	3648	2 années	1929 années	50k
`crypt-bf/6`	7168	1 année	982 années	25k
`crypt-bf/5`	13504	188 années	521 années	12.5k
`crypt-md5`	171584	15 jours	41 années	1k
`crypt-des`	23221568	157.5 minutes	108 jours	7
`sha1`	37774272	90 minutes	68 jours	4
`md5` (hash)	150085504	22.5 minutes	17 jours	1

Lorsqu'elle est compilée avec OpenSSL 3.0.0 ou une version ultérieure, le fournisseur hérité doit être activé dans le fichier de configuration `openssl.cnf` pour utiliser les anciens chiffrements tels que DES or Blowfish.

Notes :

- La machine utilisée est un Intel Mobile Core i3.

- Les numéros des algorithmes `crypt-des` et `crypt-md5` sont pris de la sortie du -test de John the Ripper v1.6.38.
- Les nombres hachés `md5` font partie de mdcrack 1.2.
- Les nombres `sha1` font partie de lcrack-20031130-beta.
- Les nombres `crypt-bf` sont pris en utilisant le programme simple qui boucle sur 1000 mots de passe de huit caractères. De cette façon, je peux afficher la vitesse avec les différents nombres de tours. Pour référence : `john -test` affiche 13506 tours/sec pour `crypt-bf/5`. (La petite différence dans les résultats est dû au fait que l'implémentation de `crypt-bf` dans `pgcrypto` est la même que celle utilisée dans John the Ripper.)

Notez que « tenter toutes les combinaisons » n'est pas un exercice réaliste. Habituellement, craquer les mots de passe se fait avec l'aide de dictionnaires contenant les mots standards et différentes variantes. Donc, même des mots de passe qui ressemblent vaguement à des mots peuvent être craqués plus rapidement que les nombres ci-dessus le suggèrent alors qu'un mot de passe sur six caractères qui ne ressemble pas à un mot pourrait ne pas être craqué.

F.28.3. Fonctions de chiffrement PGP

Les fonctions implémentent la partie chiffrement du standard OpenPGP (RFC 4880[4]). Les chiffrements à clés symétriques et publiques sont supportés.

Un message PGP chiffré consiste en deux parties ou *paquets* :

- Un paquet contenant la clé de session -- soit une clé symétrique soit une clé publique chiffrée.
- Paquet contenant les données chiffrées avec la clé de session.

Lors du chiffrement avec une clé symétrique (par exemple, un mot de passe) :

1. Le mot de passe est haché en utilisant l'algorithme String2Key (S2K). C'est assez similaire à l'algorithme `crypt()` -- lenteur voulue et nombre aléatoire pour le sel -- mais il produit une clé binaire de taille complète.
2. Si une clé de session séparée est demandée, une nouvelle clé sera générée au hasard. Sinon une clé S2K sera utilisée directement en tant que clé de session.
3. Si une clé S2K est à utiliser directement, alors seuls les paramètres S2K sont placés dans le paquet de session. Sinon la clé de session sera chiffrée avec la clé S2K et placée dans le paquet de session.

Lors du chiffrement avec une clé publique :

1. Une nouvelle clé de session est générée au hasard.
2. Elle est chiffrée en utilisant la clé public et placée dans le paquet de session.

Dans les deux cas, les données à chiffrer sont traitées ainsi :

1. Manipulation optionnelle des données : compression, conversion vers UTF-8, conversion de retours à la ligne.
2. Les données sont préfixées avec un bloc d'octets pris au hasard. C'est identique à l'utilisation de *random IV*.
3. Un hachage SHA1 d'un préfixe et de données au hasard est ajouté.
4. Tout ceci est chiffré avec la clé de la session et placé dans la paquet de données.

F.28.3.1. `pgp_sym_encrypt()`

[4] https://datatracker.ietf.org/doc/html/rfc4880

```
   pgp_sym_encrypt(data text, psw text [, options text ]) returns
bytea
   pgp_sym_encrypt_bytea(data bytea, psw text [, options text ])
returns bytea
```

Chiffre *data* avec une clé PGP symétrique *psw*. Le paramètre *options* peut contenir des options décrites ci-dessous.

F.28.3.2. pgp_sym_decrypt()

```
   pgp_sym_decrypt(msg bytea, psw text [, options text ]) returns
text
   pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ])
returns bytea
```

Déchiffre un message PGP chiffré avec une clé symétrique.

Déchiffrer des données `bytea` avec `pgp_sym_decrypt` est interdit. Ceci a pour but d'éviter la sortie de données de type caractère invalides. Déchiffrer des données textuelles avec `pgp_sym_decrypt_bytea` ne pose pas de problème.

Le paramètre *options* peut contenir les paramètres décrits ci-dessous.

F.28.3.3. pgp_pub_encrypt()

```
   pgp_pub_encrypt(data text, key bytea [, options text ]) returns
bytea
   pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ])
returns bytea
```

Chiffre *data* avec la clé PGP publique *key*. Avec cette fonction, une clé privée renverra une erreur.

Le paramètre *options* peut contenir des options décrites ci-dessous.

F.28.3.4. pgp_pub_decrypt()

```
   pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options
text ]]) returns text
   pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [,
options text ]]) returns bytea
```

Déchiffre un message chiffré avec une clé publique. *key* doit être la clé secrète correspondant à la clé publique utilisée pour chiffrer. Si la clé secrète est protégée par un mot de passe, vous devez saisir le mot de passe dans *psw*. S'il n'y a pas de mot de passe mais que vous devez indiquer des options, vous devez saisir un mot de passe vide.

Déchiffrer des données `bytea` avec `pgp_pub_decrypt` est interdit. Ceci a pour but d'éviter la sortie de données de type caractère invalides. Déchiffrer des données textuelles avec `pgp_pub_decrypt_bytea` ne pose pas de problème.

Le paramètre *options* peut contenir des options décrites ci-dessous.

F.28.3.5. pgp_key_id()

```
pgp_key_id(bytea) returns text
```

`pgp_key_id` extrait l'identifiant de la clé pour une clé PGP publique ou secrète. Ou il donne l'identifiant de la clé utilisé pour chiffrer les données si un message chiffré est fourni.

Elle peut renvoyer deux identifiants de clés spéciaux :

- `SYMKEY`

  Le message est chiffré avec une clé symétrique.

- `ANYKEY`

  La donnée est chiffrée avec une clé publique mais l'identifiant de la clé est effacé. Cela signifie que vous avez besoin d'essayer toutes les clés secrètes pour voir laquelle la déchiffre. `pgcrypto` ne réalise pas lui-même de tels messages.

Notez que des clés différentes peuvent avoir le même identifiant. C'est rare mais normal. L'application client doit alors essayer de déchiffer avec chacune d'elle pour voir laquelle correspond -- ce qui revient à la gestion de `ANYKEY`.

F.28.3.6. `armor()`, `dearmor()`

```
armor(data bytea [ , keys text[], values text[] ]) returns text
dearmor(data text) returns bytea
```

Ces fonctions enveloppent les données dans une armure ASCII PGP qui est basiquement en Base64 avec CRC et un formatage supplémentaire.

Si les tableaux *`keys`* et *`values`* sont précisées, un *entête d'armure* est ajouté au format standard pour chaque paire clé/valeur. Les deux tableaux doivent avoir une seule dimension, et ils doivent avoir la même longueur. Les clés et valeurs ne peuvent pas contenir de caractères non ASCII.

F.28.3.7. `pgp_armor_headers`

```
pgp_armor_headers(data text, key out text, value out text) returns
 setof record
```

`pgp_armor_headers()` extrait les en-têtes d'armure à partir de *`data`*. La valeur de retour est un ensemble de lignes avec deux colonnes, une clé et une valeur. Si les clés ou valeurs contiennent des caractères non ASCII, ils sont traités avec l'encodage UTF-8.

F.28.3.8. Options pour les fonctions PGP

Les options sont nommées de façon similaires à GnuPG. Les valeurs sont fournies après un signe d'égalité ; les options sont séparées par des virgules. Par exemple :

```
    pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-
algo=aes256')
```

Toutes les options en dehors de `convert-crlf` s'appliquent seulement aux fonctions de chiffrement. Les fonctions de déchiffrement obtiennent des paramètres des données PGP.

Les options les plus intéressantes sont probablement `compression-algo` et `unicode-mode`. Le reste doit avoir des valeurs par défaut raisonnables.

F.28.3.8.1. cipher-algo

Quel algorithme de chiffrement à utiliser.

Valeurs : bf, aes128, aes192, aes256, 3des, cast5
Par défaut : aes128
Applique à : pgp_sym_encrypt, pgp_pub_encrypt

F.28.3.8.2. compress-algo

Algorithme de compression à utiliser. Seulement disponible si PostgreSQL a été construit avec zlib.

Valeurs :
0 - sans compression
1 - compression ZIP
2 - compression ZLIB [=ZIP plus meta-data and block-CRC's]
Par défaut : 0
S'applique à : pgp_sym_encrypt, pgp_pub_encrypt

F.28.3.8.3. compress-level

Niveau de compression. Les grands niveaux compressent mieux mais sont plus lents. 0 désactive la compression.

Valeurs : 0, 1-9
Par défaut : 6
S'applique à : pgp_sym_encrypt, pgp_pub_encrypt

F.28.3.8.4. convert-crlf

Précise si `\n` doit être converti en `\r\n` lors du chiffrement et `\r\n` en `\n` lors du déchiffrement. La RFC 4880 spécifie que les données texte doivent être stockées en utilisant les retours chariot `\r\n`. Utilisez cette option pour obtenir un comportement respectant la RFC.

Valeurs : 0, 1
Par défaut : 0
S'applique à : pgp_sym_encrypt, pgp_pub_encrypt, pgp_sym_decrypt, pgp_pub_decrypt

F.28.3.8.5. disable-mdc

Ne protège pas les données avec SHA-1. La seule bonne raison pour utiliser cette option est d'avoir une compatibilité avec les anciens produits PGP précédant l'ajout de paquets protégés SHA-1 dans la RFC 4880. Les versions récentes des logiciels de gnupg.org et pgp.com le supportent.

Valeurs : 0, 1
Par défaut : 0
S'applique à : pgp_sym_encrypt, pgp_pub_encrypt

F.28.3.8.6. sess-key

Utilise la clé de session séparée. Le chiffrement par clé publique utilise toujours une clé de session séparée, c'est pour le chiffrement de clé symétrique, qui utilise directement par défaut S2K.

Valeurs : 0, 1
Par défaut : 0
S'applique à : pgp_sym_encrypt

F.28.3.8.7. s2k-mode

Algorithme S2K à utiliser.

Valeurs :
0 - Sans sel. Dangereux !
1 - Avec sel mais avec un décompte fixe des itérations.
3 - Décompte variables des itérations.
Par défaut : 3
S'applique à : pgp_sym_encrypt

F.28.3.8.8. s2k-count

Le nombre d'itérations de l'algorithme S2K à utiliser. La valeur doit être comprise entre 1024 et 65011712, valeurs incluses.

Par défaut : Une valeur aléatoire entre 65536 et 253952
S'applique à : pgp_sym_encrypt, seulement avec s2k-mode=3

F.28.3.8.9. s2k-digest-algo

Algorithme digest à utiliser dans le calcul S2K.

Valeurs : md5, sha1
Par défaut : sha1
S'applique à : pgp_sym_encrypt

F.28.3.8.10. s2k-cipher-algo

Chiffrement à utiliser pour le chiffrage de la clé de session séparée.

Valeurs : bf, aes, aes128, aes192, aes256
Par défaut : use cipher-algo
S'applique à : pgp_sym_encrypt

F.28.3.8.11. unicode-mode

Sélection de la conversion des données texte à partir de l'encodage interne de la base vers l'UTF-8 et inversement. Si votre base de données est déjà en UTF-8, aucune conversion ne sera réalisée, seules les données seront marquées comme étant en UTF-8. Sans cette option, cela ne se fera pas.

Valeurs : 0, 1
Par défaut : 0
S'applique à : pgp_sym_encrypt, pgp_pub_encrypt

F.28.3.8.12. ignore-cipher-failure

Dangereux ! Demande à pgcrypto d'utiliser un algorithme de déchiffrage incorrect correspondant au comportement historique avant le correctif pour CVE-2026-14663, en ignorant complètement les échecs du *cipher* d'OpenSSL. C'est uniquement à l'intention des utilisateurs qui ont besoin de récupérer des messages mal chiffrés créés lorsque `cipher-algo` n'était pas disponible avec la configuration en cours d'OpenSSL. Des messages erronés ne requièrent pas la bonne clé de déchiffrement quand `ignore-cipher-failure` est activé, donc il n'y a pas de garantie que le texte déchiffré venait bien d'un déteneur de la clé.

Comparez ce cas à celui d'un message correctement chiffré, mais dont l'algorithme de chiffrement n'est pas disponible avec la configuration OpenSSL actuelle. La récupération de ce texte en clair via `pgcrypto` nécessite de rendre le chiffrement effectif accessible à OpenSSL, par exemple en activant le fournisseur approprié. L'option `ignore-cipher-failure` n'est ni nécessaire ni utile dans ce scénario. Si le « déchiffrement » d'un message correctement chiffré effectué avec cette option venait à passer les contrôles d'intégrité PGP, ce résultat serait fortuit et ne rendrait pas fiable le texte récupéré en clair.

Valeurs : 0, 1
Par défaut : 0
S'applique à : pgp_sym_decrypt, pgp_pub_decrypt

F.28.3.9. Générer des clés PGP avec GnuPG

Pour générer une nouvelle clé :

```
gpg --gen-key
```

Le type de clé préféré est « DSA and Elgamal ».

Pour le chiffrement RSA, vous devez créer soit une clé de signature seulement DSA ou RSA en tant que maître, puis ajouter la sous-clé de chiffrement RSA avec `gpg --edit-key`.

Pour lister les clés :

```
gpg --list-secret-keys
```

Pour exporter une clé publique dans un format armure ASCII :

```
gpg -a --export KEYID > public.key
```

Pour exporter une clé secrète dans un format armure ASCII :

```
gpg -a --export-secret-keys KEYID > secret.key
```

Vous avez besoin d'utiliser la fonction `dearmor()` sur ces clés avant de les passer aux fonctions PGP. Ou si vous gérez des données binaires, vous pouvez supprimer l'option `-a` pour la commande.

Pour plus de détails, voir la `page de référence de gpg`, le livre « GNU Privacy Handbook »[5] et d'autres documents sur le site gnupg.org[6].

F.28.3.10. Limites du code PGP

- Pas de support des signatures. Cela signifie aussi qu'on ne peut pas vérifier si la sous-clé de chiffrage appartient bien à la clé maître.
- Pas de support de la clé de chiffrement en tant que clé maître. Cela ne devrait pas être un problème étant donné que cette pratique n'est pas encouragée.
- Pas de support pour plusieurs sous-clés. Ceci peut être un problème car c'est une pratique courante. D'un autre côté, vous ne devez pas utiliser vos clés GPG/PGP habituelles avec `pgcrypto`, mais en créer de nouvelles car l'utilisation est assez différente.

F.28.4. Fonctions de chiffrement brut (Raw)

Ces fonctions exécutent directement un calcul des données ; ils n'ont pas de fonctionnalités avancées de chiffrement PGP. Du coup, ils ont les problèmes majeurs suivant :

1. Elles utilisent directement la clé de l'utilisateur comme clé de calcul.
2. Elles ne fournissent pas une vérification de l'intégrité pour savoir si les données chiffrées ont été modifiées.
3. Elles s'attendent à ce que les utilisateurs gèrent eux-même tous les paramètres du chiffrement, même IV.
4. Elles ne gèrent pas le texte.

Donc, avec l'introduction du chiffrement PGP, l'utilisation des fonctions de chiffrement brut n'est pas encouragée.

```
    encrypt(data bytea, key bytea, type text) returns bytea
    decrypt(data bytea, key bytea, type text) returns bytea

    encrypt_iv(data bytea, key bytea, iv bytea, type text) returns
 bytea
    decrypt_iv(data bytea, key bytea, iv bytea, type text) returns
 bytea
```

Chiffrer/déchiffrer les données en utilisant la méthode de calcul spécifiée par *`type`*. La syntaxe de la chaîne *`type`* est :

```
   algorithm [ - mode ] [ /pad: padding ]
```

où *`algorithm`* fait partie de :

- `bf` -- Blowfish
- `aes` -- AES (Rijndael-128, -192 ou -256)

[5] https://www.gnupg.org/gph/en/manual.html
[6] https://www.gnupg.org/

et *mode* fait partie de :

- cbc -- le bloc suivant dépend du précédent. (par défaut)
- ecb -- chaque bloc est chiffré séparément. (seulement pour les tests)

et *padding* fait partie de :

- pkcs -- les données peuvent avoir n'importe quelle longueur (par défaut)
- none -- les données doivent être des multiples de la taille du bloc de calcul.

Donc, pour exemple, ces derniers sont équivalents :

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

Dans encrypt_iv et decrypt_iv, le paramètre *iv* est la valeur initiale pour le mode CBC ; elle est ignorée pour ECB. Elle est remplie de zéro pour l'alignement si la taille de données ne correspond à un multiple de la taille du bloc. Elle a pour valeur par défaut que des zéros dans les fonctions sans ce paramètre.

F.28.5. Fonctions d'octets au hasard

```
gen_random_bytes(count integer) returns bytea
```

Renvoie *count*) octets pour un chiffrement fort. Il peut y avoir au maximum 1024 octets extrait à un instant t, ceci pour éviter de vider le contenu du générateur de nombres aléatoires.

```
gen_random_uuid() returns uuid
```

Retourne un UUID de version 4 (aléatoire). (Obsolète, cette fonction appelle en interne la fonction native du même nom.)

F.28.6. Notes

F.28.6.1. Configuration

pgcrypto se configure lui-même suivant les découvertes du script configure principal de PostgreSQL. Les options qui l'affectent sont --with-zlib et --with-ssl=openssl.

Quand il est compilé avec zlib, les fonctions de chiffrement PGP peuvent compresser les données avant chiffrement.

pgcrypto requiert OpenSSL. Sinon, il ne sera ni construit ni installé.

Lorsqu'il est compilé avec la version 3.0.0 ou ultérieure d'OpenSSL, l'ancien fournisseur doit être activé dans le fichier de configuration openssl.cnf pour utiliser les chiffrements les plus anciens comme DES ou Blowfish.

F.28.6.2. Gestion des NULL

Comme le standard SQL le demande, toutes les fonctions renvoient NULL si un des arguments est NULL. Cela peut permettre une faille de sécurité si c'est utilisé sans précaution.

F.28.6.3. Limites de la sécurité

Toutes les fonctions de `pgcrypto` sont exécutées au sein du serveur de bases de données. Cela signifie que toutes les données et les mots de passe sont passés entre `pgcrypto` et l'application client en texte clair. Donc, vous devez :

1. Vous connecter localement ou utiliser des connexions SSL ;
2. Faire confiance à votre administrateur système et de base de données.

Si vous ne le pouvez pas, alors il est préférable de chiffrer directement au sein de l'application client.

L'implémentation ne résiste pas à des attaques par canal auxiliaire[7]. Par exemple, le temps requis pour terminer l'exécution d'une fonction de déchiffrement de `pgcrypto` varie suivant les texts de déchiffrement d'une certaine taille.

F.28.7. Auteur

Marko Kreen `<markokr@gmail.com>`

`pgcrypto` utilise du code provenant des sources suivantes :

Algorithme	Auteur	Origine du source
DES crypt	David Burren and others	FreeBSD libcrypt
MD5 crypt	Poul-Henning Kamp	FreeBSD libcrypt
Blowfish crypt	Solar Designer	www.openwall.com

F.29. pg_freespacemap

Le module `pg_freespacemap` fournit un moyen d'examiner la carte des espaces libres (FSM). Il fournit une fonction appelée `pg_freespace`, ou plus précisément deux fonctions qui se surchargent. Les fonctions indiquent la valeur enregistrée dans la carte des espaces libres pour une page donnée ou pour toutes les pages de la relation.

Par défaut l'utilisation est restreinte aux superutilisateurs et aux rôles disposant des attributs du rôle `pg_stat_scan_tables`. L'accès peut être accordé à d'autres en utilisant `GRANT`.

F.29.1. Fonctions

`pg_freespace(rel regclass IN, blkno bigint IN) returns int2`

Renvoie la quantité d'espace libre dans la page de la relation, spécifiée par `blkno`, d'après la FSM.

`pg_freespace(rel regclass IN, blkno OUT bigint, avail OUT int2)`

Affiche la quantité d'espace libre sur chaque page de la relation suivant la FSM. Un ensemble de lignes du type `(blkno bigint, avail int2)` est renvoyé, une ligne pour chaque page de la relation.

Les valeurs stockées dans la carte des espaces libres ne sont pas exactes. Elles sont arrondies à une précision de 1/256# du `BLCKSZ` (32 octets pour un `BLCKSZ` par défaut), et elles ne sont pas parfaitement mises à jour quand des lignes sont insérées et mises à jour.

Pour les index, sont tracées les pages entièrement inutilisées, plutôt que l'espace vide au sein des pages. En conséquence, les valeurs ne sont pas significatives. Elles indiquent simplement si la page est en cours d'utilisation ou vide.

[7] https://en.wikipedia.org/wiki/Side-channel_attack

F.29.2. Exemple de sortie

```
postgres=# SELECT * FROM pg_freespace('foo');
 blkno | avail
-------+-------
     0 |     0
     1 |     0
     2 |     0
     3 |    32
     4 |   704
     5 |   704
     6 |   704
     7 |  1216
     8 |   704
     9 |   704
    10 |   704
    11 |   704
    12 |   704
    13 |   704
    14 |   704
    15 |   704
    16 |   704
    17 |   704
    18 |   704
    19 |  3648
(20 rows)

postgres=# SELECT * FROM pg_freespace('foo', 7);
 pg_freespace
--------------
         1216
(1 row)
```

F.29.3. Auteur

Version originale par Mark Kirkwood <markir@paradise.net.nz>. Réécrit en version 8.4 pour s'adapter à la nouvelle implémentation de la FSM par Heikki Linnakangas <heikki@enterprisedb.com>

F.30. pg_prewarm

Le module `pg_prewarm` fournit un moyen pratique de charger des données des relations dans le cache de données du système d'exploitation ou dans le cache de données de PostgreSQL. Ce préchargement peut être lancé manuellement avec la fonction `pg_prewarm`, ou automatiquement en incluant `pg_prewarm` dans shared_preload_libraries. Dans ce dernier cas, le système exécutera un processus d'arrière-plan (*background worker*) qui enregistrera périodiquement le contenu des shared buffers dans un fichier nommé `autoprewarm.blocks` puis, après un redémarrage rechargera ces blocs en utilisant deux *background workers*.

F.30.1. Fonctions

```
pg_prewarm(regclass, mode text default 'buffer', fork text default
 'main',
```

```
first_block int8 default null,
last_block int8 default null) RETURNS int8
```

Le premier argument est la relation qui doit être préchargée. Le second est la méthode de préchargement à utiliser, comme décrit plus bas. Le troisième argument correspond au type de fichier à précharger (généralement `main`). Le quatrième argument est le numéro du premier bloc à précharger (`NULL` est accepté comme synonyme de zéro). Le cinquième argument correspond au dernier numéro de bloc à précharger (`NULL` signifie que l'on précharge jusqu'au dernier bloc dans la relation). La valeur retournée correspond au nombre de blocs préchargés.

Il y a trois méthodes de préchargement disponibles. `prefetch` envoie une requête de prélecture asynchrone au système d'exploitation si celui-ci le supporte ou sinon renvoie une erreur. `read` lit l'intervalle de blocs demandé. Contrairement à `prefetch`, toutes les plateformes et options de compilation le supportent, mais cette méthode peut être plus lente. `buffer` lit l'intervalle de blocs demandé pour le charger dans le cache de données de la base.

Il est à noter qu'avec n'importe laquelle de ces méthodes, tenter de précharger plus de blocs qu'il n'est possible de mettre en cache -- par le système d'exploitation en utilisant `prefetch` ou `read`, ou par PostgreSQL en utilisant `buffer` -- aura probablement pour effet d'expulser du cache les blocs des numéros inférieurs au fur et à mesure que les blocs des numéros supérieurs seront lus. De plus, les données préchargées ne bénéficient d'aucune protection spécifique contre l'éviction du cache. Il est donc possible que d'autres activités du système d'exploitation puissent évincer du cache les données fraîchement préchargées peu après leur lecture. Pour toutes ces raisons, le préchargement est typiquement plus utile au démarrage, quand les caches sont majoritairement vides.

```
autoprewarm_start_worker() RETURNS void
```

Lance autoprewarm, le worker principal. Normalement cela est automatique, mais ce peut être utile si le préchauffage automatique n'était pas configuré au démarrage du serveur et que vous voulez démarrer le worker plus tard.

```
autoprewarm_dump_now() RETURNS int8
```

Met à jour `autoprewarm.blocks` immédiatement. Ce peut être utile si le worker autoprewarm worker ne fonctionne pas mais que vous prévoyez de le lancer après le prochain redémarrage. La valeur retournée est le nombre d'enregistrements écrits dans `autoprewarm.blocks`.

F.30.2. Paramètres de configuration

`pg_prewarm.autoprewarm` (`boolean`)

Contrôle si le serveur doit lancer le worker autoprewarm. La valeur par défaut est « on ». Ce paramètre ne peut être positionné qu'au démarrage du serveur.

`pg_prewarm.autoprewarm_interval` (`integer`)

Il s'agit de l'intervalle entre les mises à jour de `autoprewarm.blocks`. La valeur par défaut est de 300 secondes. Si la valeur est 0, le fichier ne sera pas écrit à intervalles réguliers, mais seulement à l'extinction du serveur.

Ces paramètres doivent être configurés dans le fichier `postgresql.conf`. Un cas d'usage typique serait :

```
# postgresql.conf
```

```
shared_preload_libraries = 'pg_prewarm'

pg_prewarm.autoprewarm = true
pg_prewarm.autoprewarm_interval = 300s
```

F.30.3. Auteur

Robert Haas <rhaas@postgresql.org>

F.31. pgrowlocks

Le module `pgrowlocks` fournit une fonction pour afficher les informations de verrouillage de lignes pour une table spécifiée.

Par défaut, l'utilisation est restreinte aux superutilisateurs, aux rôles disposant des droits du rôle `pg_stat_scan_tables`, ainsi que les utilisateurs avec le droit `SELECT` sur la table.

F.31.1. Aperçu

```
pgrowlocks(text) returns setof record
```

Le paramètre est le nom d'une table. Le résultat est un ensemble d'enregistrements, avec une ligne pour chaque ligne verrouillée dans la table. Les colonnes en sortie sont affichées dans Tableau F.19.

Tableau F.19. Colonnes de `pgrowlocks`

Nom	Type	Description
`locked_row`	`tid`	ID de ligne (TID) d'une ligne verrouillée
`locker`	`xid`	ID de transaction de la pose du verrou, ou ID multixact dans le cas d'une multi-transaction
`multi`	`boolean`	True si le verrou est détenu par une multi-transaction
`xids`	`xid[]`	ID de transaction détenant les verrous (plus d'une en cas de multi-transaction)
`modes`	`text[]`	Mode de verrouillage des verrous (plus d'un dans le cas d'une multi-transaction), un tableau de `Key Share`, `Share`, `For No Key Update`, `No Key Update`, `For Update`, `Update`.
`pids`	`integer[]`	ID de processus des serveurs ayant posé les verrous (plus d'une en cas de multi-transaction)

`pgrowlocks` prend un verrou `AccessShareLock` pour la table cible et lit chaque ligne une par une pour récupérer les informations de verrouillage de lignes. Ce n'est pas très rapide pour une grosse table. Notez que :

1. Si un verrou de type ACCESS EXCLUSIVE est posé sur la table, pgrowlocks sera bloqué.

2. pgrowlocks ne garantit pas de produire une image cohérente. Il est possible qu'un nouveau verrou de ligne soit pris ou qu'un ancien verrou soit libéré pendant son exécution.

pgrowlocks ne montre pas le contenu des lignes verrouillées. Si vous voulez jeter un œil au contenu de la ligne en même temps, vous pouvez le faire ainsi :

```
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
  WHERE p.locked_row = a.ctid;
```

Mais soyez conscient qu'une telle requête sera particulièrement inefficace.

F.31.2. Exemple d'affichage

```
=# SELECT * FROM pgrowlocks('t1');
 locked_row | locker | multi | xids  |     modes      |  pids
------------+--------+-------+-------+----------------+--------
 (0,1)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,2)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,3)      |    607 | f     | {607} | {"For Update"} | {3107}
 (0,4)      |    607 | f     | {607} | {"For Update"} | {3107}
(4 rows)
```

F.31.3. Auteur

Tatsuo Ishii

F.32. pg_stat_statements

Le module pg_stat_statements fournit un moyen de surveiller les statistiques d'optimisation et d'exécution de tous les ordres SQL exécutés par un serveur.

Le module doit être chargé par l'ajout de pg_stat_statements à shared_preload_libraries dans le fichier de configuration postgresql.conf parce qu'il a besoin de mémoire partagée supplémentaire. Ceci signifie qu'il faut redémarrer le serveur pour ajouter ou supprimer le module. De plus, le calcul de l'identifiant de requête doit être activé pour que le module soit actif, ce qui est fait automatiquement si compute_query_id est configuré à auto ou on, ou tout autre module tiers chargé qui calcule les identifiants de requête.

Quand pg_stat_statements est actif, il récupère des statistiques sur toutes les bases de données du serveur. Pour y accéder et les manipuler, le module fournit les vues pg_stat_statements et pg_stat_statements_info, et les fonctions pg_stat_statements_reset et pg_stat_statements. Elles ne sont pas disponibles globalement mais peuvent être activées pour une base de données spécifique avec l'instruction CREATE EXTENSION pg_stat_statements.

F.32.1. La vue pg_stat_statements

Les statistiques collectées par le module sont rendues disponibles par une vue nommée pg_stat_statements. Cette vue contient une ligne pour chaque combinaison d'un identifiant de base de données, d'un identifiant utilisateur, d'un identifiant de requête et s'il s'agit d'une requête

principale ou non (jusqu'au nombre maximum d'ordres distincts que le module peut surveiller). Les colonnes de la vue sont affichées dans Tableau F.20.

Tableau F.20. Colonnes de `pg_stat_statements`

Type de colonne **Description**
`userid` `oid` (référence `pg_authid.oid`) OID de l'utilisateur qui a exécuté la requête
`dbid` `oid` (référence `pg_database.oid`) OID de la base dans laquelle a été exécutée la requête
`toplevel` `bool` True si la requête a été exécutée comme instruction de haut niveau (toujours true si `pg_stat_statements.track` est configuré à `top`)
`queryid` `bigint` Code de hachage, calculé à partir de l'arbre d'analyse de la requête
`query` `text` Texte d'une requête représentative
`plans` `bigint` Nombre d'optimisations de la requête (si `pg_stat_statements.track_planning` est activé, sinon zéro)
`total_plan_time` `double precision` Durée totale passée à optimiser la requête, en millisecondes (si `pg_stat_statements.track_planning` est activé, sinon zéro)
`min_plan_time` `double precision` Durée minimale passée à optimiser la requête, en millisecondes (si `pg_stat_statements.track_planning` est activé, sinon zéro)
`max_plan_time` `double precision` Durée maximale passée à optimiser la requête, en millisecondes (si `pg_stat_statements.track_planning` est activé, sinon zéro)
`mean_plan_time` `double precision` Durée moyenne passée à optimiser la requête, en millisecondes (si `pg_stat_statements.track_planning` est activé, sinon zéro)
`stddev_plan_time` `double precision` Déviation standard de la durée passée à optimiser la requête, en millisecondes (si `pg_stat_statements.track_planning` est activé, sinon zéro)
`calls` `bigint` Nombre d'exécutions de la requête
`total_exec_time` `double precision` Durée totale passée à exécuter la requête, en millisecondes
`min_exec_time` `double precision` Durée minimale passée à exécuter la requête, en millisecondes
`max_exec_time` `double precision` Durée maximale passée à exécuter la requête, en millisecondes
`mean_exec_time` `double precision` Durée moyenne passée à exécuter la requête, en millisecondes
`stddev_exec_time` `double precision` Déviation standard de la durée passée à exécuter la requête, en millisecondes
`rows` `bigint` Nombre total de lignes récupérées ou affectées par la requête
`shared_blks_hit` `bigint`

Type de colonne **Description**
Nombre total de blocs lus dans le cache partagé par la requête
`shared_blks_read` `bigint` Nombre total de blocs lus hors cache partagé par la requête
`shared_blks_dirtied` `bigint` Nombre total de blocs modifiés dans le cache partagé par la requête
`shared_blks_written` `bigint` Nombre total de blocs du cache partagé écrit sur disque par la requête
`local_blks_hit` `bigint` Nombre total de blocs lus dans le cache local par la requête
`local_blks_read` `bigint` Nombre total de blocs lus hors du cache local par la requête
`local_blks_dirtied` `bigint` Nombre total de blocs modifiés dans le cache local par la requête
`local_blks_written` `bigint` Nombre total de blocs du cache local écrit sur disque par la requête
`temp_blks_read` `bigint` Nombre total de blocs lus dans les fichiers temporaires par la requête
`temp_blks_written` `bigint` Nombre total de blocs écrits dans les fichiers temporaires par la requête
`blk_read_time` `double precision` Durée totale de lecture des blocs des fichiers de données (hors cache) par la requête, en millisecondes (si track_io_timing est activé, sinon zéro)
`blk_write_time` `double precision` Durée totale de l'écriture des blocs des fichiers de données (hors cache) par la requête, en millisecondes (si track_io_timing est activé, sinon zéro)
`temp_blk_read_time` `double precision` Durée totale des lectures des blocs de fichiers temporaires en millisecondes (si track_io_timing est activé, sinon zéro)
`temp_blk_write_time` `double precision` Durée totale des écritures des blocs de fichiers temporaires en millisecondes (si track_io_timing est activé, sinon zéro)
`wal_records` `bigint` Nombre total d'enregistrements générés dans les WAL par la requête
`wal_fpi` `bigint` Nombre total d'images complètes de blocs (*full page images*) générés dans les WAL par la requête
`wal_bytes` `numeric` Nombre total d'octets générés dans les WAL par la requête
`jit_functions` `bigint` Nombre total de fonctions compilées par JIT pour cette requête
`jit_generation_time` `double precision` Durée totale passée par la requête sur la génération de code JIT, en millisecondes
`jit_inlining_count` `bigint` Nombre de fois où les fonctions ont été incluses
`jit_inlining_time` `double precision` Durée totale passée par la requête sur l'inclusion de fonctions, en millisecondes
`jit_optimization_count` `bigint`

Type de colonne Description
Nombre de fois où la requête a été optimisée
`jit_optimization_time` `double precision` Durée totale passée sur l'optimisation de la requête, en millisecondes
`jit_emission_count` `bigint` Nombre de fois où du code a été émis
`jit_emission_time` `double precision` Durée totale passée par la requête sur de l'émission de code, en millisecondes

Pour raisons de sécurité, seuls les superutilisateurs et les rôles disposant des droits du rôle `pg_read_all_stats` sont autorisé à voir le texte SQL ainsi que le champ `queryid` des requêtes exécutées par d'autres utilisateurs. Les autres utilisateurs peuvent cependant voir les statistiques, si la vue a été installée dans leur base de données.

Les requêtes qui disposent d'un plan d'exécution (c'est-à-dire `SELECT`, `INSERT`, `UPDATE`, `DELETE` et `MERGE`) sont combinées en une entrée unique dans `pg_stat_statements` lorsqu'elles ont un plan d'exécution similaire (d'après leur hachage). En substance, cela signifie que deux requêtes seront considérées comme équivalentes si elles sont sémantiquement les mêmes mais disposent de valeurs littérales différentes dans la requête. Les requêtes utilitaires (c'est-à-dire toutes les autres) ne sont considérées comme unique que lorsqu'elles sont égales au caractère près.

> **Note**
>
> Les détails suivant sur le remplacement des constantes et le `queryid` s'appliquent seulement si compute_query_id est activé. Si vous utilisez un module externe pour calculer `queryid`, vous devez vous référer à sa documentation pour les détails.

Quand la valeur d'une constante a été ignorée pour pouvoir comparer la requête à d'autres requêtes, la constante est remplacée par un symbole de paramètre, tel que `$1`, dans l'affichage de `pg_stat_statements`. Le reste du texte de la requête est tel qu'était la première requête ayant la valeur de hashage `queryid` spécifique associée à l'entrée dans `pg_stat_statements`.

Dans certains cas, les requêtes SQL avec des textes différents peuvent être fusionnés en une seule entrée `pg_stat_statements`. Normalement, cela n'arrive que pour les requêtes dont la sémantique est équivalente, mais il y a une petite chance que des collisions de l'algorithme de hachage aient pour conséquence la fusion de requêtes sans rapport en une entrée. (Cela ne peut cependant pas arriver pour des requêtes appartenant à des utilisateurs différents ou des bases de données différentes).

Puisque la valeur de hachage `queryid` est calculée sur la représentation de la requête après analyse, l'inverse est également possible : des requêtes avec un texte identique peuvent apparaître comme des entrées séparées, si elles ont des significations différentes en fonction de facteurs externes, comme des réglages de `search_path` différents.

Les programmes utilisant `pg_stat_statements` pourraient préférer utiliser `queryid` (peut-être en association avec `dbid` et `userid`) pour disposer d'un identifiant plus stable et plus sûr pour chaque entrée plutôt que le texte de la requête. Cependant, il est important de comprendre qu'il n'y a qu'une garantie limitée sur la stabilité de la valeur de hachage de `queryid`. Puisque l'identifiant est dérivé de l'arbre après analyse, sa valeur est une fonction, entre autres choses, des identifiants d'objet interne apparaissant dans cette représentation. Cela a des implications paradoxales. Par exemple, `pg_stat_statements` considérera deux requêtes apparemment identiques comme distinctes, si elles référencent une table qui a été supprimée et recréée entre l'exécution de ces deux requêtes. Le processus de hachage est également sensible aux différences d'architecture des machines ainsi que d'autres facettes de la plateforme. De plus, il n'est pas sûr de partir du principe que `queryid` restera stable entre des versions majeures de PostgreSQL.

Deux serveurs participant à une réplication basée sur le rejeu physique des journaux de transactions devraient avoir des valeurs de `queryid` identiques pour la même requête. Cependant, les systèmes de réplication logique ne promettent pas de conserver des réplicas identiques surtout les détails intéressants, donc `queryid` ne sera pas un identifiant utile pour accumuler les coûts dans un ensemble de réplicas logiques. En cas de doute, un test direct est recommandé.

En règle général, il peut être supposé que les valeurs `queryid` sont stables entre des versions mineures de PostgreSQL, en supposant que les instances sont exécutées sur la même architecture matérielle et que les détails de métadonnées du catalogue correspondent. La compatibilité entre versions mineures ne sera cassée qu'en cas de dernière extrémité.

Le symbole de paramètre utilité pour remplacer les constantes dans le texte représentatif de la requête démarre après le plus grand paramètre `$`*n* dans le texte de la requête originale, ou `$1` s'il n'y en avait pas. Il est intéressant de noter que dans certains cas il pourrait y avoir un symbole de paramètre caché qui affecte cette numérotation. Par exemple, PL/pgSQL utilise des symbole de paramètre cachés pour insérer des valeurs de variables locales à la fonction dans les requêtes, ainsi un ordre PL/pgSQL comme `SELECT i + 1 INTO j` aurait un texte représentatif tel que `SELECT i + $2`.

Les textes des requêtes sont conservées dans un fichier texte externe et ne consomment pas de mémoire partagée. De ce fait, même les textes très longs de requêtes peuvent être enregistrés avec succès. Néanmoins, si beaucoup de textes très longs de requêtes sont accumulées, le fichier externe peut devenir suffisamment gros pour ne plus être gérable. Si cela survient, comme méthode de restauration, `pg_stat_statements` peut choisir d'ignorer les textes de requêtes. Dans ce cas, le champ `query` apparaitra vide sur les lignes de la vue `pg_stat_statements` mais les statistiques associées seront préservées. Si cela arrive, réfléchissez à réduire la valeur du paramètre `pg_stat_statements.max` pour empêcher que cela ne recommence.

`plans` et `calls` peuvent différer car les statistiques d'optimisation/planification et d'exécution sont mises à jours à leur fin respective, et seulement si elles ont réussi. Par exemple, si une requête est optimisée avec succès mais échoue sur la phase d'exécution, seules les statistiques d'optimisation/planification seront modifiées. Si l'optimisation/planification est ignorée parce qu'un plan en cache est utilisé, seules les statistiques d'exécution seront mises à jour.

F.32.2. La vue `pg_stat_statements_info`

Les statistiques du module `pg_stat_statements` lui-même sont tracées et rendues disponibles via une vue nommée `pg_stat_statements_info`. Cette vue contient une seule ligne. Les colonnes de la vue sont affichées dans Tableau F.21.

Tableau F.21. Colonnes de `pg_stat_statements_info`

Type de colonne Description
`dealloc bigint` Nombre total de fois où les enregistrements de `pg_stat_statements` pour les requêtes les moins exécutées ont été désallouées parce que plus de `pg_stat_statements.max` requêtes distinctes ont été observées
`stats_reset timestamp with time zone` Horodatage de la dernière réinitialisation de toutes les statistiques de `pg_stat_statements`.

F.32.3. Fonctions

```
pg_stat_statements_reset(userid Oid, dbid Oid, queryid bigint)
returns void
```

`pg_stat_statements_reset` annule les statistiques récupérées jusqu'à maintenant par `pg_stat_statements` correspondant aux `userid`, `dbid` et `queryid` indiqués. Si un de

ces paramètres n'est pas spécifié, la valeur par défaut `0` (invalide) est utilisée pour chacun d'entre eux et les statistiques qui correspondent avec les autres paramètres seront réinitialisées. Si aucun paramètre n'est spécifié ou si tous les paramètres spécifiés sont à `0` (invalide), elle annulera toutes les statistiques. Si toutes les statistiques de la vue `pg_stat_statements` sont annulées, cela va aussi réinitialiser les statistiques de la vue `pg_stat_statements_info`. Par défaut, cette fonction peut seulement être exécutée par les superutilisateurs. L'accès peut être donné à d'autres utilisateurs en utilisant la commande `GRANT`.

`pg_stat_statements(showtext boolean) returns setof record`

La vue `pg_stat_statements` est basée sur une fonction également nommée `pg_stat_statements`. Les clients peuvent appeler la fonction `pg_stat_statements` directement, et peuvent en spécifiant `showtext := false` ne pas récupérer le texte de la requête (ce qui veut dire que l'argument `OUT` qui correspond à la colonne `query` de la vue retournera des NULL). Cette fonctionnalité est prévue pour le support d'outils externes qui pourraient vouloir éviter le surcoût de récupérer de manière répétée les textes des requêtes de longueur indéterminées. De tels outils peuvent à la place eux-même mettre le premier texte de requête récupéré pour chaque entrée, puisque c'est déjà ce que fait `pg_stat_statements` lui-même, et ensuite récupérer les textes de requêtes uniquement si nécessaire. Puisque le serveur stocke les textes de requête dans un fichier, cette approche pourrait réduire les entrée/sorties physiques pour des vérifications répétées des données de `pg_stat_statements`.

F.32.4. Paramètres de configuration

`pg_stat_statements.max` (`integer`)

`pg_stat_statements.max` est le nombre maximum d'ordres tracés par le module (c'est-à-dire le nombre maximum de lignes dans la vue `pg_stat_statements`). Si un nombre supérieur d'ordres SQL distincts a été observé, c'est l'information sur les ordres les moins exécutés qui est ignorée. Le nombre de fois où une telle information est ignorée est consultable dans la vu `pg_stat_statements_info`. La valeur par défaut est 5000. Ce paramètre peut uniquement être positionné au démarrage du serveur.

`pg_stat_statements.track` (`enum`)

`pg_stat_statements.track` contrôle quels sont les ordres comptabilisés par le module. Spécifiez `top` pour suivre les ordres de plus haut niveau (ceux qui sont soumis directement par les clients), `all` pour suivre également les ordres imbriqués (tels que les ordres invoqués dans les fonctions) ou `none` pour désactiver la récupération des statistiques sur les requêtes. La valeur par défaut est `top`. Seuls les superutilisateurs peuvent changer ce paramétrage.

`pg_stat_statements.track_utility` (`boolean`)

`pg_stat_statements.track_utility` contrôle si les commandes utilitaires sont tracées par le module. Les commandes utilitaires sont toutes les commandes SQL sauf `SELECT`, `INSERT`, `UPDATE`, `DELETE` et `MERGE`. La valeur par défaut est `on`. Seuls les superutilisateurs peuvent modifier cette configuration.

`pg_stat_statements.track_planning` (`boolean`)

`pg_stat_statements.track_planning` contrôle si les opérations d'optimisation/planification et leur durée sont tracées par ce module. Activer ce paramètre pourrait résulter en une perte visible de performance, spécialement quand les requêtes avec des structures identiques sont exécutées par de nombreuses connexions concurrentes, ce qui pousse à mettre à jour un petit nombre d'entrées de `pg_stat_statements`. La valeur par défaut est `off`. Seuls les superutilisateurs peuvent modifier cette configuration.

`pg_stat_statements.save` (`boolean`)

`pg_stat_statements.save` précise s'il faut sauvegarder les statistiques lors des arrêts du serveur. S'il est `off`, alors les statistiques ne sont pas sauvegardées lors de l'arrêt ni rechargées au

démarrage du serveur. La valeur par défaut est on. Ce paramètre peut uniquement être positionné dans le fichier postgresql.conf ou sur la ligne de commande du serveur.

Le module a besoin de mémoire partagée supplémentaire proportionnelle à pg_stat_statements.max. Notez que cette mémoire est consommée quand le module est chargé, même si pg_stat_statements.track est positionné à none.

Ces paramètres doivent être définis dans postgresql.conf. Un usage courant pourrait être :

```
# postgresql.conf
shared_preload_libraries = 'pg_stat_statements'

compute_query_id = on
pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

F.32.5. Exemple de sortie

```
bench=# SELECT pg_stat_statements_reset();

$ pgbench -i bench
$ pgbench -c10 -t300 bench

bench=# \x
bench=# SELECT query, calls, total_exec_time, rows, 100.0 *
 shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS
 hit_percent
          FROM pg_stat_statements ORDER BY total_exec_time DESC
 LIMIT 5;

-[ RECORD 1 ]---+-------------------------------------------------
------------------
query           | UPDATE pgbench_branches SET bbalance = bbalance +
 $1 WHERE bid = $2
calls           | 3000
total_exec_time | 25565.855387
rows            | 3000
hit_percent     | 100.0000000000000000
-[ RECORD 2 ]---+-------------------------------------------------
------------------
query           | UPDATE pgbench_tellers SET tbalance = tbalance +
 $1 WHERE tid = $2
calls           | 3000
total_exec_time | 20756.669379
rows            | 3000
hit_percent     | 100.0000000000000000
-[ RECORD 3 ]---+-------------------------------------------------
------------------
query           | copy pgbench_accounts from stdin
calls           | 1
total_exec_time | 291.865911
rows            | 100000
hit_percent     | 100.0000000000000000
-[ RECORD 4 ]---+-------------------------------------------------
------------------
```

```
query           | UPDATE pgbench_accounts SET abalance = abalance +
 $1 WHERE aid = $2
calls           | 3000
total_exec_time | 271.232977
rows            | 3000
hit_percent     | 98.8454011741682975
-[ RECORD 5 ]---+-------------------------------------------------
------------------
query           | alter table pgbench_accounts add primary key
 (aid)
calls           | 1
total_exec_time | 160.588563
rows            | 0
hit_percent     | 100.0000000000000000


bench=# SELECT pg_stat_statements_reset(0,0,s.queryid) FROM
 pg_stat_statements AS s
           WHERE s.query = 'UPDATE pgbench_branches SET bbalance =
 bbalance + $1 WHERE bid = $2';

bench=# SELECT query, calls, total_exec_time, rows, 100.0 *
 shared_blks_hit /
              nullif(shared_blks_hit + shared_blks_read, 0) AS
 hit_percent
         FROM pg_stat_statements ORDER BY total_exec_time DESC
 LIMIT 5;
-[ RECORD 1 ]---+-------------------------------------------------
------------------
query           | UPDATE pgbench_tellers SET tbalance = tbalance +
 $1 WHERE tid = $2
calls           | 3000
total_exec_time | 20756.669379
rows            | 3000
hit_percent     | 100.0000000000000000
-[ RECORD 2 ]---+-------------------------------------------------
------------------
query           | copy pgbench_accounts from stdin
calls           | 1
total_exec_time | 291.865911
rows            | 100000
hit_percent     | 100.0000000000000000
-[ RECORD 3 ]---+-------------------------------------------------
------------------
query           | UPDATE pgbench_accounts SET abalance = abalance +
 $1 WHERE aid = $2
calls           | 3000
total_exec_time | 271.232977
rows            | 3000
hit_percent     | 98.8454011741682975
-[ RECORD 4 ]---+-------------------------------------------------
------------------
query           | alter table pgbench_accounts add primary key
 (aid)
calls           | 1
total_exec_time | 160.588563
rows            | 0
hit_percent     | 100.0000000000000000
```

```
-[ RECORD 5 ]---+-------------------------------------------------
-----------------
query           | vacuum analyze pgbench_accounts
calls           | 1
total_exec_time | 136.448116
rows            | 0
hit_percent     | 99.9201915403032721

bench=# SELECT pg_stat_statements_reset(0,0,0);

bench=# SELECT query, calls, total_exec_time, rows, 100.0 *
 shared_blks_hit /
              nullif(shared_blks_hit + shared_blks_read, 0) AS
 hit_percent
         FROM pg_stat_statements ORDER BY total_exec_time DESC
 LIMIT 5;
-[ RECORD 1 ]---+-------------------------------------------------
--------------------------
query           | SELECT pg_stat_statements_reset(0,0,0)
calls           | 1
total_exec_time | 0.189497
rows            | 1
hit_percent     |
-[ RECORD 2 ]---+-------------------------------------------------
--------------------------
query           | SELECT query, calls, total_exec_time, rows, $1 *
 shared_blks_hit /          +
                |               nullif(shared_blks_hit +
 shared_blks_read, $2) AS hit_percent+
                |           FROM pg_stat_statements ORDER BY
 total_exec_time DESC LIMIT $3
calls           | 0
total_exec_time | 0
rows            | 0
hit_percent     |
```

F.32.6. Auteurs

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>. La normalisation des requêtes a été ajoutée par Peter Geoghegan <peter@2ndquadrant.com>.

F.33. pgstattuple

Le module pgstattuple fournit plusieurs fonctions pour obtenir des statistiques au niveau ligne.

Comme ces fonctions renvoient des informations détaillées au niveau page, l'accès est restreint par défaut. Par défaut, seul le rôle pg_stat_scan_tables a le droit EXECUTE. Bien sûr, les superutilisateurs contournent cette restriction. Après l'installation de l'extension, les utilisateurs peuvent exécuter des commandes GRANT pour modifier les droits sur les fonctions, pour permettre à d'autres rôles de les exécuter. Néanmoins, il serait préférable d'ajouter ces utilisateurs au rôle pg_stat_scan_tables.

F.33.1. Fonctions

```
pgstattuple(regclass) returns record
```

`pgstattuple` renvoie la longueur physique d'une relation, le pourcentage des lignes « mortes », et d'autres informations. Ceci peut aider les utilisateurs à déterminer si une opération de VACUUM est nécessaire. L'argument est le nom de la relation cible (qui peut être qualifié par le nom du schéma) ou l'OID. Par exemple :

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]------+-------
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

Les colonnes en sortie sont décrites dans Tableau F.22.

Tableau F.22. Colonnes de `pgstattuple`

Colonne	Type	Description
`table_len`	`bigint`	Longueur physique de la relation en octets
`tuple_count`	`bigint`	Nombre de lignes vivantes
`tuple_len`	`bigint`	Longueur totale des lignes vivantes en octets
`tuple_percent`	`float8`	Pourcentage des lignes vivantes
`dead_tuple_count`	`bigint`	Nombre de lignes mortes
`dead_tuple_len`	`bigint`	Longueur totale des lignes mortes en octets
`dead_tuple_percent`	`float8`	Pourcentage des lignes mortes
`free_space`	`bigint`	Espace libre total en octets
`free_percent`	`float8`	Pourcentage de l'espace libre

> **Note**
>
> La valeur de la colonne `table_len` sera toujours supérieure à la somme des colonnes `tuple_len`, `dead_tuple_len` et `free_space`. La différence correspond aux données systèmes comme la table de pointeurs vers les lignes (une table par bloc) et aux octets d'alignements permettant de s'assurer que les lignes sont correctement alignées.

`pgstattuple` acquiert seulement un verrou en lecture sur la relation. Les résultats ne reflètent donc pas une image instantanée, des mises à jour en parallèle pouvant en effet les affecter.

`pgstattuple` juge qu'une ligne est « morte » si `HeapTupleSatisfiesDirty` renvoie false.

`pgstattuple(text) returns record`

Identique à `pgstattuple(regclass)`, sauf que la relation cible est désignée en tant que TEXT. Cette fonction est conservée pour raison de compatibilité ascendante, et sera dépréciée dans une prochaine version.

`pgstatindex(regclass) returns record`

`pgstatindex` renvoie un enregistrement affichant des informations sur un index B-Tree. Par exemple :

```
test=> SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]------+------
version            | 2
tree_level         | 0
index_size         | 16384
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 54.27
leaf_fragmentation | 0
```

En voici les colonnes :

Colonne	Type	Description
`version`	`integer`	Numéro de version du B-tree
`tree_level`	`integer`	Niveau de l'arbre pour la page racine
`index_size`	`bigint`	Taille totale de l'index en octets
`root_block_no`	`bigint`	Emplacement du bloc racine (0 si aucun)
`internal_pages`	`bigint`	Nombre de pages « internes » (niveau supérieur)
`leaf_pages`	`bigint`	Nombre de pages feuilles
`empty_pages`	`bigint`	Nombre de pages vides
`deleted_pages`	`bigint`	Nombre de pages supprimées
`avg_leaf_density`	`float8`	Densité moyenne des pages feuilles
`leaf_fragmentation`	`float8`	Fragmentation des pages feuilles

L'information `index_size` rapportée correspondra normalement à un bloc de plus que ce qui est indiqué par la formule `internal_pages + leaf_pages + empty_pages + deleted_pages` car elle inclut aussi le bloc de méta-données de l'index.

Comme pour `pgstattuple`, les résultats sont accumulés page par page, et ne représentent pas forcément une image instantanée de l'index complet.

`pgstatindex(text) returns record`

Identique à `pgstatindex(regclass)`, sauf que l'index cible est spécifié en tant que TEXT. Cette fonction est conservée pour raison de compatibilité ascendante, et sera dépréciée dans une prochaine version.

`pgstatginindex(regclass) returns record`

`pgstatginindex` renvoie un enregistrement montrant les informations sur un index GIN. Par exemple :

```
test=> SELECT * FROM pgstatginindex('test_gin_index');
-[ RECORD 1 ]--+--
version        | 1
pending_pages  | 0
pending_tuples | 0
```

Les colonnes en sortie sont :

Colonne	Type	Description
`version`	`integer`	Numéro de version GIN
`pending_pages`	`integer`	Nombre de pages dans la liste en attente
`pending_tuples`	`bigint`	Nombre de lignes dans la liste en attente

`pgstathashindex(regclass) returns record`

`pgstathashindex` retourne un enregistrement montrant des informations à propos d'un index HASH. Par exemple :

```
test=> select * from pgstathashindex('con_hash_index');
-[ RECORD 1 ]--+-----------------
version        | 4
bucket_pages   | 33081
overflow_pages | 0
bitmap_pages   | 1
unused_pages   | 32455
live_items     | 10204006
dead_items     | 0
free_percent   | 61.8005949100872
```

Les colonnes en sortie sont :

Colonne	Type	Description
`version`	`integer`	Numéro de version de HASH
`bucket_pages`	`bigint`	Nombre de pages bucket
`overflow_pages`	`bigint`	Nombre de pages overflow
`bitmap_pages`	`bigint`	Nombre de pages bitmap
`unused_pages`	`bigint`	Nombre de pages inutilisées
`live_items`	`bigint`	Nombre de lignes vivantes
`dead_tuples`	`bigint`	Nombre de lignes mortes
`free_percent`	`float`	Pourcentage d'espace libre

`pg_relpages(regclass) returns bigint`

`pg_relpages` renvoie le nombre de pages dans la relation.

`pg_relpages(text) returns bigint`

Identique à `pg_relpages(regclass)`, sauf que la relation cible est spécifiée en tant que TEXT. Cette fonction est conservée pour raison de compatibilité ascendante, et sera dépréciée dans une prochaine version.

`pgstattuple_approx(regclass) returns record`

`pgstattuple_approx` est une alternative plus rapide à `pgstattuple` qui retourne des estimations. L'argument est le nom ou l'OID de la relation visée. Par exemple :

```
test=> SELECT * FROM
 pgstattuple_approx('pg_catalog.pg_proc'::regclass);
-[ RECORD 1 ]--------+-------
table_len            | 573440
scanned_percent      | 2
approx_tuple_count   | 2740
approx_tuple_len     | 561210
approx_tuple_percent | 97.87
dead_tuple_count     | 0
dead_tuple_len       | 0
dead_tuple_percent   | 0
approx_free_space    | 11996
approx_free_percent  | 2.09
```

Les colonnes en sortie sont décrites dans Tableau F.23.

Alors que `pgstattuple` effectue toujours un parcours séquentiel complet de la table et renvoie un décompte exact des lignes vivantes et supprimées (et de leur taille) ainsi que de l'espace libre, `pgstattuple_approx` essaie d'éviter un parcours séquentiel complet et retourne un décompte exact des lignes supprimées avec une approximation du nombre de lignes et de la taille des lignes vivantes et de l'espace libre.

Il réalise ceci en sautant les pages qui n'ont que des lignes visibles selon la carte de visibilité (si une page a son bit positionné dans la carte de visibilité, elle est considérée comme ne possédant pas de lignes supprimées). Pour ces pages, il déduit l'espace libre correspondant à partir de la carte des espaces libres, et suppose que le reste de la page est occupé par des lignes vivantes.

Pour les pages qui ne sont pas sautées, il balaie chaque ligne, en enregistrant son existence et sa taille dans les compteurs correspondants, et ajoute l'espace libre de la page. À la fin, il estime le nombre de lignes vivantes en se basant sur le nombre de pages et de lignes visitées (de la même manière que VACUUM estime pg_class.reltuples).

Tableau F.23. Colonnes de `pgstattuple_approx`

Colonne	Type	Description
`table_len`	`bigint`	Longueur physique de la relation en octets (exact)
`scanned_percent`	`float8`	Pourcentage parcouru de la table
`approx_tuple_count`	`bigint`	Nombre de lignes vivantes (estimé)

Colonne	Type	Description
approx_tuple_len	bigint	Longueur totale des lignes vivantes en octets (estimé)
approx_tuple_percent	float8	Pourcentage des lignes vivantes
dead_tuple_count	bigint	Nombre de lignes mortes (exact)
dead_tuple_len	bigint	Longueur totale des lignes mortes en octets (exact)
dead_tuple_percent	float8	Pourcentage des lignes mortes
approx_free_space	bigint	Espace libre total en octets (estimé)
approx_free_percent	float8	Pourcentage de l'espace libre

Dans la sortie ci-dessus, l'espace libre indiqué peut ne pas correspondre exactement à la sortie de `pgstattuple` car la carte des espaces libres donne un chiffre exact mais pas à l'octet près.

F.33.2. Auteurs

Tatsuo Ishii, Satoshi Nagayasu et Abhijit Menon-Sen

F.34. pg_surgery

Le module `pg_surgery` fournit différentes fonctions pour réaliser des opérations sur une relation endommagée. Ces fonctions sont dangereuses de par leur concept et les utiliser pourrait corrompre (ou corrompre encore plus) votre base de données. Par exemple, ces fonctions peuvent facilement être utilisées pour rendre une table incohérente avec ses propres index, causant des violations de contraintes `UNIQUE` ou `FOREIGN KEY`, voire même de rendre des lignes visibles qui, lorsqu'elles sont lues, vont causer un crash du serveur de bases de données. Vous devez faire très attention en les utilisant. Leur utilisation doit rester pour les cas désespérés.

F.34.1. Fonctions

`heap_force_kill(regclass, tid[]) returns void`

`heap_force_kill` marque les pointeurs de lignes « utilisées » comme « mortes » sans examiner les lignes. Le but de cette fonction est de forcer la suppression de lignes autrement inaccessibles. Par exemple :

```
test=> select * from t1 where ctid = '(0, 1)';
ERROR:  could not access status of transaction 4007513275
DETAIL:  Could not open file "pg_xact/0EED": No such file or
 directory.

test=# select heap_force_kill('t1'::regclass, ARRAY['(0,
 1)']::tid[]);
 heap_force_kill
-----------------

(1 row)

test=# select * from t1 where ctid = '(0, 1)';
(0 rows)
```

```
heap_force_freeze(regclass, tid[]) returns void
```

heap_force_freeze marque les lignes comme gelées sans examiner les données des lignes. Le but de cette fonction est de rendre accessibles des lignes qui étaient auparavant inaccessibles à cause de la corruption des informations de visibilité, ou qui empêchaient la réussite d'un vacuum sur la table à cause de corruption sur les informations de visibilité. Par exemple :

```
test=> vacuum t1;
ERROR:  found xmin 507 from before relfrozenxid 515
CONTEXT:  while scanning block 0 of relation "public.t1"

test=# select ctid from t1 where xmin = 507;
 ctid
-------
 (0,3)
(1 row)

test=# select heap_force_freeze('t1'::regclass, ARRAY['(0,
 3)']::tid[]);
 heap_force_freeze
-------------------

(1 row)

test=# select ctid from t1 where xmin = 2;
 ctid
-------
 (0,3)
(1 row)
```

F.34.2. Auteurs

Ashutosh Sharma <ashu.coek88@gmail.com>

F.35. pg_trgm

Le module pg_trgm fournit des fonctions et opérateurs qui permettent de déterminer des similarités de textes alphanumériques en fonction de correspondances de trigrammes. Il fournit également des classes d'opérateurs accélérant les recherches de chaînes similaires.

Ce module est considéré comme « trusted », c'est-à-dire qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) possédant l'attribut CREATE sur la base de données courante.

F.35.1. Concepts du trigramme (ou trigraphe)

Un trigramme est un groupe de trois caractères consécutifs pris dans une chaîne. Nous pouvons mesurer la similarité de deux chaînes en comptant le nombre de trigrammes qu'elles partagent. Cette idée simple est très efficace pour mesurer la similarité des mots dans la plupart des langues.

> **Note**
>
> pg_trgm ignore les caractères qui ne forment pas de mots (donc non alphanumériques) lors de l'extraction des trigrammes d'une chaîne de caractères. Chaque mot est considéré avoir deux espaces en préfixe et un espace en suffixe lors de la détermination de l'ensemble de trigrammes contenu dans la chaîne. Par exemple, l'ensemble des trigrammes dans la chaîne « cat » est

« c », « ca », « cat » et « at ». L'ensemble de trigrammes dans la chaîne « foo|bar » est « f », « fo », « foo », « oo », « b », « ba », « bar » et « ar ».

F.35.2. Fonctions et opérateurs

Les fonctions fournies par le module pg_trgm sont affichées dans Tableau F.24 alors que les opérateurs sont indiqués dans Tableau F.25.

Tableau F.24. Fonctions de pg_trgm

Fonction Description
similarity (text, text) → real Renvoie un nombre indiquant la similarité des deux arguments. L'échelle du résultat va de zéro (indiquant que les deux chaînes sont complètement différentes) à un (indiquant que les deux chaînes sont identiques).
show_trgm (text) → text[] Renvoie un tableau de tous les trigrammes d'une chaîne donnée. (En pratique, ceci est peu utile, sauf pour le débogage.)
word_similarity (text, text) → real Renvoie un nombre qui indique la plus grande similarité entre l'ensemble de trigrammes dans la première chaîne et toute étendue continue d'un ensemble trié de trigrammes dans la deuxième chaîne. Pour les détails, voir l'explication ci-dessous.
strict_word_similarity (text, text) → real Identique à word_similarity(text, text), mais force à étendre les limites pour correspondre aux limites du mot. Comme nous n'avons pas de trigrammes sur plusieurs mots, cette fonction renvoie en fait la plus grande similarité entre la première chaîne et toute étendue continue de mots de la deuxième chaîne.
show_limit () → real Renvoie la limite de similarité utilisée par l'opérateur %. Ceci configure la similarité minimale entre deux mots pour qu'ils soient considérés suffisamment proches pour être des fautes d'orthographe l'un de l'autre (*obsolète* ; utilisez à la place SHOW pg_trgm.similarity_threshold.).
set_limit (real) → real Configure la limite de similarité actuelle utilisée par l'opérateur %. La limite doit valoir entre 0 et 1 (le défaut est 0,3). (*obsolète* ; utilisez à la place SET pg_trgm.similarity_threshold.).

Prenons l'exemple suivant :

```
# SELECT word_similarity('word', 'two words');
 word_similarity
-----------------
             0.8
(1 row)
```

Dans la première chaîne, l'ensemble de trigrammes est {" w"," wo","ord","wor","rd "}. Dans la seconde chaîne, l'ensemble trié de trigrammes est {" t"," tw","two","wo "," w"," wo","wor","ord","rds","ds "}. L'étendue la plus similaire d'un ensemble trié de trigrammes dans la seconde chaîne est {" w"," wo","wor","ord"}, et la similarité est 0.8.

Cette fonction renvoie une valeur qui peut être comprise approximativement comme la plus grande similarité entre la première chaîne et toute sous-chaîne de la deuxième chaîne. Néanmoins, cette fonction n'ajoute pas de remplissage aux limites de l'étendue. De ce fait, le nombre de caractères supplémentaires présents dans la deuxième chaîne n'est pas considéré, sauf pour les limites de mots sans correspondance.

En même temps, `strict_word_similarity` sélectionne une étendue de mots dans la deuxième chaîne. Dans l'exemple ci-dessus, `strict_word_similarity` sélectionnerait l'étendue d'un mot seul `'words'`, dont l'ensemble de trigrammes est `{"  w","wo ","wor","ord","rds","ds "}`.

```
# SELECT strict_word_similarity('word', 'two words'),
 similarity('word', 'words');
 strict_word_similarity | similarity
------------------------+------------
               0.571429 |   0.571429
(1 row)
```

De ce fait, la fonction `strict_word_similarity` est utile pour trouver la similarité de mots entiers, alors que `word_similarity` est plus intéressant pour trouver la similarité de parties de mots.

Tableau F.25. Opérateurs de `pg_trgm`

Opérateur **Description**
`text % text → boolean` Renvoie `true` si les arguments ont une similarité supérieure à la limite configurée par `pg_trgm.similarity_threshold`.
`text <% text → boolean` Renvoie `true` si la similarité entre l'ensemble de trigrammes du premier argument et une étendue continue d'un ensemble trié de trigrammes dans le second argument est plus grande que la limite de similarité actuelle, telle qu'elle est configurée avec le paramètre `pg_trgm.word_similarity_threshold`.
`text %> text → boolean` Inverse de l'opérateur <<%.
`text <<% text → boolean` Renvoie `true` si son second argument possède une étendue continue d'un ensemble de trigrammes trié correspondant aux limites de mots, et que sa similarité avec l'ensemble de trigrammes du premier argument est plus grand que la limite de similarité stricte de mot strict courante, telle que configurée par le paramètre `pg_trgm.strict_word_similarity_threshold`
`text %>> text → boolean` Inverse de l'opérateur <<%.
`text <-> text → real` Retourne la « distance » entre les arguments, c'est-à-dire un moins la valeur de `similarity()`.
`text <<-> text → real` Retourne la « distance » entre les arguments, c'est-à-dire un moins la valeur de `word_similarity()`.
`text <->> text → real`

Opérateur Description
Inverse de l'opérateur <<->.
text <<<-> text → real Retourne la « distance » entre les arguments, c'est-à-dire un moins la valeur de strict_word_similarity().
text <->>> text → real Inverse de l'opérateur <<<->.

F.35.3. Paramètres GUC

pg_trgm.similarity_threshold (real)

Configure la limite de similarité utilisée par l'opérateur %. La limite doit se situer entre 0 et 1 (la valeur par défaut est 0,3).

pg_trgm.word_similarity_threshold (real)

Configure la limite de similarité de mot utilisée par les opérateurs <% et %>. La limite doit être comprise entre 0 et 1 (la valeur par défaut est 0,6).

pg_trgm.strict_word_similarity_threshold (real)

Configure la limite de similarité de mot stricte utilisée par les opérateurs <<% et %>>. La limite doit être comprise entre 0 et 1 (la valeur par défaut est 0,5).

F.35.4. Support des index

Le module pg_trgm fournit des classes d'opérateurs pour les index GiST et GIN qui vous permettent de créer un index sur une colonne de type text dans le but d'accélérer les recherches de similarité. Ces types d'index supportent les opérateurs de similarité décrits ci-dessus et supportent de plus les recherches basées sur des trigrammes pour les requêtes LIKE, ILIKE, ~, ~* et =. Les opérateurs d'inégalité ne sont pas supportés. Notez que ces index peuvent ne pas être aussi efficaces que les index B-Tree pour l'opérateur d'égalité.

Exemple :

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING GIST (t gist_trgm_ops);
```

ou

```
CREATE INDEX trgm_idx ON test_trgm USING GIN (t gin_trgm_ops);
```

L'opérateur de classe GiST gist_trgm_ops assimile un ensemble de trigrammes à une signature bitmap. Son paramètre entier optionnel siglen détermine la longueur de la signature en octets. La valeur par défaut est de 12 octets. Les valeurs valides vont de 1 à 2024 octets. Les signatures plus longues mènent à une recherche plus précise (parcourant une plus petite fraction de l'index et moins de pages de la table), au prix d'un index plus gros.

Exemple de création d'un tel index avec une longueur de signature de 32 octets :

```
CREATE INDEX trgm_idx ON test_trgm USING GIST (t
 gist_trgm_ops(siglen=32));
```

À ce point, vous aurez un index sur la colonne `t` que vous pouvez utiliser pour une recherche de similarité. Une requête typique est :

```
SELECT t, similarity(t, 'word') AS sml
  FROM test_trgm
  WHERE t % 'word'
  ORDER BY sml DESC, t;
```

Ceci renverra toutes les valeurs dans la colonne texte suffisamment similaires à *`word`*, triées de la meilleure correspondance à la pire. L'index sera utilisé pour accélérer l'opération même sur un grand ensemble de données.

Une variante de la requête ci-dessus est

```
SELECT t, t <-> 'word' AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

Ceci peut être implémenté assez efficacement par des index GiST, mais pas par des index GIN. Cela battra généralement la première formulation quand on demande juste un petit nombre de correspondances proches.

De plus, vous pouvez utiliser un index sur la colonne `t` pour la similarité, stricte ou pas, entre mots. Des requêtes typiques sont :

```
SELECT t, strict_word_similarity('word', t) AS sml
  FROM test_trgm
  WHERE 'word' <<% t
  ORDER BY sml DESC, t;
```

et

```
SELECT t, word_similarity('word', t) AS sml
  FROM test_trgm
  WHERE 'word' <% t
  ORDER BY sml DESC, t;
```

Ceci renverra toutes les valeurs dans la colonne texte pour lesquelles il existe une étendue continue de l'ensemble ordonné de trigrammes suffisamment similaire à l'ensemble de trigrammes de *`word`*, trié de la meilleure correspondance à la pire. L'index sera utilisé pour accélérer l'opération, y compris sur de très gros ensembles de données.

Les variations possibles des requêtes ci-dessus sont :

```
SELECT t, 'word' <<-> t AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

et

```
SELECT t, 'word' <<<-> t AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

Ceci peut être implémenté assez efficacement par des index GiST, mais pas par des index GIN.

À partir de PostgreSQL 9.1, ces types d'index supportent aussi les recherches d'index pour `LIKE` et `ILIKE`, par exemple

```
SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

La recherche par index fonctionne par extraction des trigrammes à partir de la chaîne recherchée, puis en les recherchant dans l'index. Plus le nombre de trigrammes dans la recherche est important, plus efficace sera la recherche. Contrairement à des recherches basées sur les B-tree, la chaîne de recherche n'a pas besoin d'un signe pourcentage sur le côté gauche.

À partir de PostgreSQL 9.3, ces types d'index supportent aussi les recherches de correspondances d'expressions rationnelles (opérateurs ~ et ~*). Par exemple

```
SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

La recherche dans l'index fonctionne en extrayant les trigrammes de l'expression rationnelle, puis en les recherchant dans l'index. Plus il est possible d'extraire de trigrammes de l'expression rationnelle, plus la recherche dans l'index sera efficace. Contrairement à des recherches basées sur les B-tree, la chaîne de recherche n'a pas besoin d'un signe pourcentage sur le côté gauche.

Pour les recherches `LIKE` comme avec des expressions rationnelles, gardez en tête qu'un motif sans trigramme extractible dégénérera en parcours complet de l'index.

Le choix d'un indexage GiST ou GIN dépend de leurs caractéristiques de performance relatives, qui sont discutées ailleurs.

F.35.5. Intégration à la recherche plein texte

La correspondance de trigrammes est un outil très utile lorsqu'il est utilisé en conjonction avec un index plein texte. En particulier, il peut aider à la reconnaissance des mots mal orthographiés, pour lesquels le mécanisme de recherche plein texte ne trouvera pas de correspondance.

La première étape est la génération d'une table auxiliaire contenant tous les mots uniques dans les documents :

```
CREATE TABLE words AS SELECT word FROM
        ts_stat('SELECT to_tsvector(''simple'', bodytext) FROM
 documents');
```

où `documents` est une table avec un champ texte `bodytext`, où nous voulons faire nos recherches. La raison de l'utilisation de la configuration `simple` dans la fonction `to_tsvector` est que nous voulons une liste des mots originaux (non réduits à leur racine), plutôt qu'une configuration spécifique à la langue.

Ensuite, nous créons un index trigramme sur la colonne word :

```
CREATE INDEX words_idx ON words USING GIN(word gin_trgm_ops);
```

Maintenant, une requête `SELECT`, similaire à l'exemple précédent, peut être utilisée pour suggérer des mots mal orthographiés dans la recherche de l'utilisateur. Un test utile supplémentaire est de demander aussi que les mots sélectionnés soient d'une longueur similaire au mot mal orthographié.

> **Note**
>
> Comme la table `words` a été générée comme une table statique, séparée, il sera nécessaire de la régénérer périodiquement, afin qu'elle reste raisonnablement à jour avec la collection des documents. Il n'est pas nécessaire, généralement, qu'elle soit en permanence totalement à jour.

F.35.6. Références

Site de développement de GiST[8]

Site de développement de TSearch2[9]

F.35.7. Auteurs

Oleg Bartunov <oleg@sai.msu.su>, Moscou, Université de Moscou, Russie

Teodor Sigaev <teodor@sigaev.ru>, Moscou, Delta-Soft Ltd., Russie

Alexander Korotkov <a.korotkov@postgrespro.ru>, Moscou, Postgres Professional, Russie

Documentation : Christopher Kings-Lynne

Ce module est sponsorisé par Delta-Soft Ltd., Moscou, Russie.

F.36. pg_visibility

Le module `pg_visibility` fournit la possibilité d'examiner la visibility map (VM) et les informations de visibilité au niveau bloc d'une table. Il fournit aussi des fonctions permettant de vérifier l'intégrité d'une visibility map et de forcer sa reconstruction.

Trois bits différents sont utilisés pour enregistrer des informations sur la visibilité au niveau des blocs. L'octet totalement-visible (all-visible) de la visibility map indique que chaque ligne d'un bloc donné d'une relation est visible pour toute transaction courante et future. L'octet totalement-figé (all- frozen) de la visibility map indique que chaque ligne du bloc est figée, c'est-à-dire qu'aucun vacuum n'aura besoin de modifier le bloc tant qu'une ligne n'est pas insérée, mise à jour, supprimée ou verrouillée dans ce bloc. Le bit `PD_ALL_VISIBLE` dans l'en-tête de page a la même signification que l'octet totalement-visible de la visibility map, mais il est stocké au sein du bloc plutôt que dans une structure de donnée séparée. Ces deux bits seront normalement identiques, mais le bit de niveau de bloc peut parfois rester défini pendant que la visibility map est purgée lors de la récupération suite à un crash ; ou ils peuvent être différents suite à un changement survenant après que `pg_visibility` ait examiné la visibility map et avant qu'il ait examiné le bloc donnée. Tout événement causant une corruption de données peut aussi un désaccord sur ces trois bits.

Les fonctions qui affichent les informations concernant le bit `PD_ALL_VISIBLE` sont plus beaucoup plus coûteuses que celles qui consultent uniquement la visibility map. En effet, elles doivent lire les blocs de données des relations plutôt que de ne s'intéresser qu'à la visibility map (qui est bien plus petite). Les fonctions qui vérifient les blocs de données de la relation sont aussi très coûteuses.

[8] http://www.sai.msu.su/~megera/postgres/gist/
[9] http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/

F.36.1. Fonctions

`pg_visibility_map(relation regclass, blkno bigint, all_visible OUT boolean, all_frozen OUT boolean) renvoie un enregistrement`

Renvoie tous les octets complétement visibles et complétement figés de la visibility map pour un bloc donné pour une relation donnée.

`pg_visibility(relation regclass, blkno bigint, all_visible OUT boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns setof record`

Renvoie tous les octets complétement visibles et complétement figés de la visibility map pour un bloc donné pour une relation donnée ainsi que l'octet `PD_ALL_VISIBLE` pour le bloc.

`pg_visibility_map(relation regclass, blkno OUT bigint, all_visible OUT boolean, all_frozen OUT boolean) returns setof record`

Renvoie tous les octets complétement visibles et complétement figés de la visibility map pour un bloc donné pour une relation donnée.

`pg_visibility(relation regclass, blkno OUT bigint, all_visible OUT boolean, all_frozen OUT boolean, pd_all_visible OUT boolean) returns setof record`

Renvoie tous les octets complétement visibles et complétement figés de la visibility map pour un bloc donné pour une relation donnée, ainsi que l'octet `PD_ALL_VISIBLE` pour le bloc.

`pg_visibility_map_summary(relation regclass, all_visible OUT bigint, all_frozen OUT bigint) returns setof record`

Renvoie le nombre de pages complétement visibles ainsi que le nombre de pages complétement figées de la relation, en concordance avec la visibility map.

`pg_check_frozen(relation regclass, t_ctid OUT tid) returns setof tid`

Renvoie le TID (identifiant de ligne) des lignes non gelées présentes dans les pages marquées complètement figés dans la visibility map. Si cette fonction renvoie un ensemble non vide de TID, la visibility map est corrompue.

`pg_check_visible(relation regclass, t_ctid OUT tid) returns setof tid`

Renvoie les TID (identifiants de lignes) de tous les enregistrements qui ne sont pas all-visible enregistrés dans des pages marquées all-visible dans la visibility map. Si cette fonction renvoie un ensemble non vide, la visibility map est corrompue.

`pg_truncate_visibility_map(relation regclass) returns void`

Tronque la visibility map de la relation indiquée. Cette fonction est utile si vous pensez que la visibility map de cette relation est corrompue et que vous souhaitez forcer sa reconstruction. Le premier `VACUUM` exécuté sur cette relation après l'exécution de cette fonction parcourera chaque bloc de la relation et reconstruiera la visibility map. (Tant que cela ne sera pas fait, les requêtes traiteront la visibility map comme ne contenant que des zéros.)

Par défaut, ces fonctions ne sont exécutables que par des superutilisateurs et les rôles disposant des droits du rôle `pg_stat_scan_tables`, à l'exception de `pg_truncate_visibility_map(relation regclass)` qui ne peut être exécutée que par des superutilisateurs.

F.36.2. Auteur

Robert Haas `<rhaas@postgresql.org>`

F.37. pg_walinspect

Le module `pg_walinspect` fournit des fonctions SQL qui vous permettent d'inspecter, à un bas niveau, le contenu des journaux de transactions d'une instance PostgreSQL en cours d'exécution, ce qui est utile pour le débogage, l'analyse, la création de rapports ou à des fins pédagogiques. Il est similaire à pg_waldump, mais est accessible via SQL plutôt que via un utilitaire séparé.

Toutes les fonctions de ce module fourniront des informations sur les WAL utilisant l'identifiant de la timeline actuelle du serveur.

Toutes les fonctions de ce module essaieront de trouver le premier enregistrement WAL valide qui est situé à l'endroit ou après le *`in_lsn`* ou le *`start_lsn`* donné et émettra une erreur si un tel enregistrement n'est pas présent. De même, *`end_lsn`* doit être présent, et s'il tombe au milieu d'un enregistrement, l'intégralité de l'enregistrement doit être disponible.

> **Note**
>
> Certaines fonctions, telles que `pg_logical_emit_message`, renvoient le LSN *après* l'enregistrement qui vient d'être inséré. Par conséquent, si vous transmettez ce LSN comme *`in_lsn`* ou *`start_lsn`* à l'une de ces fonctions, elle renverra l'enregistrement *suivant*.

Par défaut, l'utilisation de ces fonctions est réservée aux superutilisateurs et aux membres du rôle `pg_read_server_files`. L'accès peut être accordé par des superutilisateurs à d'autres utilisateurs grâce à la commande `GRANT`.

F.37.1. Fonctions générales

`pg_get_wal_record_info(in_lsn pg_lsn) returns record`

Obtient les informations d'enregistrement WAL pour un LSN donné. Si le LSN fourni n'est pas au début d'un enregistrement WAL, elle retourne les informations du prochain enregistrement WAL valide disponible ; ou une erreur si aucun enregistrement de ce type n'est trouvé. Voici un exemple de l'utilisation de cette fonction :

```
postgres=# SELECT * FROM pg_get_wal_record_info('0/1E826E98');
-[ RECORD 1 ]----
+-----------------------------------------------
start_lsn        | 0/1E826F20
end_lsn          | 0/1E826F60
prev_lsn         | 0/1E826C80
xid              | 0
resource_manager | Heap2
record_type      | PRUNE
record_length    | 58
main_data_length | 8
fpi_length       | 0
description      | snapshotConflictHorizon 33748 nredirected 0
 ndead 2
block_ref        | blkref #0: rel 1663/5/60221 fork main blk 2
```

`pg_get_wal_records_info(start_lsn pg_lsn, end_lsn pg_lsn)`

Obtient des informations sur tous les enregistrements WAL valides entre *`start_lsn`* et *`end_lsn`*. Renvoie une ligne par enregistrement WAL. Si *`start_lsn`* ou *`end_lsn`* ne sont pas encore disponibles, la fonction renvoie une erreur. Par exemple :

```
postgres=# SELECT * FROM pg_get_wal_records_info('0/1E913618',
 '0/1E913740') LIMIT 1;
-[ RECORD 1 ]----
+-------------------------------------------------------------
start_lsn        | 0/1E913618
end_lsn          | 0/1E913650
prev_lsn         | 0/1E9135A0
xid              | 0
resource_manager | Standby
record_type      | RUNNING_XACTS
record_length    | 50
main_data_length | 24
fpi_length       | 0
description      | nextXid 33775 latestCompletedXid 33774
 oldestRunningXid 33775
block_ref        |
```

`pg_get_wal_records_info_till_end_of_wal(start_lsn pg_lsn) returns setof record`

Cette fonction est identique à `pg_get_wal_records_info()` sauf qu'elle obtient des informations sur tous les enregistrements WAL valides depuis *`start_lsn`* jusqu'à la fin du WAL.

`pg_get_wal_stats(start_lsn pg_lsn, end_lsn pg_lsn, per_record boolean DEFAULT false) returns setof record`

Récupère les statistiques de tous les enregistrements WAL valides entre *`start_lsn`* et *`end_lsn`*. Par défaut, elle renvoie une ligne par type *`resource_manager`*. Lorsque *`per_record`* est défini sur `true`, elle renvoie une ligne par *`record_type`*. Si *`start_lsn`* ou *`end_lsn`* ne sont pas encore disponibles, la fonction retourne une erreur. Par exemple :

```
postgres=# SELECT * FROM pg_get_wal_stats('0/1E847D00',
 '0/1E84F500')
            WHERE count > 0 AND
                  "resource_manager/record_type" =
 'Transaction'
             LIMIT 1;
-[ RECORD 1 ]---------------+-------------------
resource_manager/record_type | Transaction
count                        | 2
count_percentage             | 8
record_size                  | 875
record_size_percentage       | 41.23468426013195
fpi_size                     | 0
fpi_size_percentage          | 0
combined_size                | 875
combined_size_percentage     | 2.8634072910530795
```

`pg_get_wal_stats_till_end_of_wal(start_lsn pg_lsn, per_record boolean DEFAULT false) returns setof record`

Cette fonction est identique à `pg_get_wal_stats()` sauf qu'elle obtient les statistiques de tous les enregistrements WAL valides depuis *`start_lsn`* jusqu'à la fin du WAL.

F.37.2. Auteur

Bharath Rupireddy <bharath.rupireddyforpostgres@gmail.com>

F.38. postgres_fdw

Le module `postgres_fdw` fournit le wrapper de données distantes `postgres_fdw`, dont le but est de donner accès à des données enregistrées dans des serveurs PostgreSQL externes.

Les fonctionnalités proposées par ce module sont à peu près les mêmes que celles proposées par le module dblink. Mais `postgres_fdw` fournit une syntaxe plus transparente et respectant les standards pour l'accès à des tables distantes. Elle peut aussi donner de meilleures performances dans beaucoup de cas.

Pour préparer un accès distant en utilisant `postgres_fdw` :

1. Installez l'extension `postgres_fdw` en utilisant CREATE EXTENSION.
2. Créez un objet serveur distant en utilisant CREATE SERVER, pour représenter chaque base distante à laquelle vous souhaitez vous connecter. Indiquez les informations de connexions, sauf `user` et `password`, comme options de l'objet serveur.
3. Créez une correspondance d'utilisateur avec CREATE USER MAPPING pour chaque utilisateur de la base que vous voulez autoriser à accéder à un serveur distant. Indiquez le nom et le mot de passe de l'utilisateur distant avec les options `user` et `password` de la correspondance d'utilisateur.
4. Créez une table distante avec CREATE FOREIGN TABLE ou IMPORT FOREIGN SCHEMA pour chaque table distante que vous voulez utiliser. Les colonnes de la table distante doit correspondre aux colonnes de la table sur le serveur distant. Néanmoins, vous pouvez utiliser un nom de table et des noms de colonne différents de ceux de la table sur le serveur distant si vous indiquez les bons noms de colonne en options de la table distante.

Maintenant, vous avez seulement besoin de `SELECT` sur la table distante pour accéder aux données de la table du serveur distant. Vous pouvez aussi modifier la table sur le serveur distant en utilisant les commandes `INSERT`, `UPDATE`, `DELETE`, `COPY` et `TRUNCATE`. (Bien sûr, l'utilisateur distant que vous avez spécifié dans vos correspondances d'utilisateur doit avoir les droits pour réaliser ces actions.)

Notez que l'option `ONLY` spécifiée dans `SELECT`, `UPDATE`, `DELETE` ou `TRUNCATE` n'a pas d'effet lors des accès ou modifications d'une table distante.

Notez que `postgres_fdw` n'a pour l'instant pas de support pour les instructions `INSERT` avec une clause `ON CONFLICT DO UPDATE`. Néanmoins, la clause `ON CONFLICT DO NOTHING` est supportée, si la spécification de l'index unique est omise. Notez aussi que `postgres_fdw` supporte le déplacement de ligne demandé par des instructions `UPDATE` exécutées sur des tables partitionnées, mais il ne gère pas le cas où une partition distante choisir pour insérer une ligne est aussi une partition cible d'`UPDATE` qui sera mise à jour ailleurs dans la même commande.

Il est généralement recommandé que les colonnes d'une table distante soient déclarées avec exactement les mêmes types de données et le même collationnement que celles utilisées pour les colonnes référencées dans la table du serveur distant. Bien que `postgres_fdw` soit actuellement assez lâche sur les conversions de type de données, des anomalies sémantiques surprenantes peuvent survenir quand les types ou les collationnements ne correspondent pas dans le cas où le serveur distant interprète légèrement différemment les conditions de la requête.

Notez qu'une table distante peut être déclarée avec moins de colonnes ou avec les colonnes dans un ordre différent. La correspondance des colonnes sur la table du serveur distant se fait par nom, et non pas par position.

F.38.1. Options FDW de postgres_fdw

F.38.1.1. Options de connexions

Un serveur distant utilisant le wrapper de données distantes `postgres_fdw` peut avoir les mêmes options que celles acceptées par libpq dans les chaînes de connexion comme décrit dans Section 34.1.2. Cependant, ces options ne sont pas autorisées ou sont gérées d'une façon spéciale :

- `user`, `password` et `sslpassword` (spécifiez-les au niveau de la correspondance d'utilisateur, ou utilisez un fichier service)
- `client_encoding` (ceci est configuré automatiquement à partir de l'encodage du serveur local)
- `application_name` - ceci pourrait apparaître dans *une ou les deux* connexion et postgres_fdw.application_name. Si les deux sont présents, `postgres_fdw.application_name` surcharge le paramètre de connexion. Contrairement à libpq, `postgres_fdw` permet à `application_name` d'inclure des « séquences d'échappement ». Voir postgres_fdw.application_name pour les détails.
- `fallback_application_name` (toujours configuré à `postgres_fdw`)
- `sslkey` et `sslcert` - ils peuvent apparaître soit dans une connexion, soit dans la correspondance d'utilisateur soit dans les deux. Si ce dernier cas est vrai, la configuration de la correspondance d'utilisateur surcharge la configuration de la connexion.

Seuls les superutilisateurs peuvent créer ou modifier des correspondances d'utilisateur avec les paramètres `sslcert` et `sslkey`.

Seuls les superutilisateurs peuvent se connecter à un serveur distant sans authentification par mot de passe. Donc spécifiez toujours l'option `password` pour les correspondances d'utilisateur appartenant aux utilisateurs simples.

Un superutilisateur peut dépasser cette vérification sur une base par-correspondance-utilisateur en configurant l'option `password_required 'false'`, par exemple :

```
ALTER USER MAPPING FOR some_non_superuser SERVER loopback_nopw
OPTIONS (ADD password_required 'false');
```

Pour empêcher des utilisateurs sans droit d'exploiter les droits d'authentification de l'utilisateur unix utilisé par le serveur PostgreSQL pour escalader vers des droits superutilisateur, seul le superutilisateur peut configurer cette option sur une correspondance d'utilisateur.

Une grande attention est nécessaire pour s'assurer que cela n'autorise pas l'utilisateur à se connecter comme un superutilisateur sur la base de données distante d'après les CVE-2007-3278 et CVE-2007-6601. Ne configurez pas `password_required=false` sur le rôle `public`. Gardez en tête que l'utilisateur peut potentiellement utiliser tout certificat client, le fichier `.pgpass`, le fichier `.pg_service.conf`, et les autres fichiers se trouvant dans le répertoire personnel unix de l'utilisateur système qui exécute le serveur PostgreSQL. Ils peuvent aussi utiliser toute relation de confiance autorisée par les modes d'authentification comme `peer` et `ident`.

F.38.1.2. Options pour le nom de l'objet

Ces options peuvent être utilisées pour contrôler les noms utilisés dans les requêtes SQL envoyées au serveur PostgreSQL distant. Ces options sont nécessaires lorsqu'une table distante est créée avec des noms différents de ceux de la table du serveur distant.

`schema_name` (`string`)

Cette option, qui peut être indiquée pour une table distante, donne le nom du schéma à utiliser pour la table du serveur distant. Si cette option est omise, le nom du schéma de la table distante est utilisé.

`table_name` (`string`)

Cette option, qui peut être indiquée pour une table distante, donne le nom de la table à utiliser pour la table du serveur distant. Si cette option est omise, le nom de la table distante est utilisé.

`column_name` (`string`)

Cette option, qui peut être indiquée pour une colonne d'une table distante, donne le nom de la colonne à utiliser pour la colonne de la table du serveur distant. Si cette option est omise, le nom de la colonne de la table distante est utilisé.

F.38.1.3. Options d'estimation du coût

`postgres_fdw` récupère des données distantes en exécutant des requêtes sur des serveurs distants. Idéalement, le coût estimé du parcours d'une table distante devrait être celui occasionné par le parcours de la table sur le serveur distant, et un supplément causé par la communication entre le serveur local et le serveur distant. Le moyen le plus fiable d'obtenir une telle estimation est de demander au serveur distant, puis d'ajouter quelque chose pour le supplément. Pour des requêtes simples, cela ne vaut pas le coût d'une requête supplémentaire vers le serveur distant. Donc `postgres_fdw` propose les options suivantes pour contrôler la façon dont l'estimation de coût est faite :

`use_remote_estimate` (`boolean`)

Cette option, qui peut être indiquée pour une table distante ou pour un serveur distant, contrôle si `postgres_fdw` exécute des commandes `EXPLAIN` distantes pour obtenir les estimations de coût. Une configuration sur la table distante surcharge celle sur le serveur, mais seulement pour cette table. La valeur par défaut est `false`.

`fdw_startup_cost` (`floating point`)

Cette option, qui peut être indiquée pour un serveur distant, est une valeur numérique qui est ajoutée au coût de démarrage estimé de tout parcours de table distante sur ce serveur. Cela représente le coût supplémentaire causé par l'établissement d'une connexion, l'analyse et la planification de la requête du côté du serveur distant, etc. La valeur par défaut est `100`.

`fdw_tuple_cost` (`floating point`)

Cette option, qui peut être indiquée pour un serveur distant, est une valeur en virgule flottante qui est utilisée comme coût supplémentaire par ligne pour les parcours de la table distante sur ce serveur. Cela représente le coût supplémentaire associé au transfert de données entre les serveurs. Vous pouvez augmenter ou réduire ce nombre pour refléter les latences réseau vers le serveur distant. La valeur par défaut est `0.01`.

Quand `use_remote_estimate` est vrai, `postgres_fdw` obtient le nombre de lignes et les estimations de coût à partir du serveur distant. Il ajoute `fdw_startup_cost` et `fdw_tuple_cost` aux estimations de coût. Quand `use_remote_estimate` est faux, `postgres_fdw` réalise le décompte local des lignes ainsi que l'estimation de coût, puis ajoute `fdw_startup_cost` et `fdw_tuple_cost` aux estimations de coût. Cette estimation locale a peu de chances d'être précise sauf si des copies locales des statistiques de la table distante sont disponibles. Exécuter ANALYZE sur la table distante permet de mettre à jour les statistiques locales ; cela exécute un parcours sur la table distante, puis calcule et enregistre les statistiques comme si la table était locale. Garder des statistiques locales peut être utile pour réduire la surcharge de planification par requête pour une table distante mais, si la table distante est fréquemment mise à jour, les statistiques locales seront rapidement obsolètes.

F.38.1.4. Options d'exécution à distance

Par défaut, seules les clauses `WHERE` utilisant des opérateurs et des fonctions intégrés sont considérés pour une exécution sur le serveur distant. Les clauses impliquant des fonctions non intégrées sont vérifiées localement une fois les lignes récupérées. Si ces fonctions sont disponibles sur le serveur distant et peuvent produire les mêmes résultats que localement, les performances peuvent être améliorées en envoyant ces clauses `WHERE` pour une exécution distante. Ce comportement peut être contrôlé en utilisant l'option suivante :

`extensions` (`string`)

Cette option est une liste de noms d'extensions PostgreSQL, séparés par des virgules, installées dans des versions compatibles sur les serveurs local et distant. Les fonctions et opérateurs immutables et appartenant à une extension listée seront considérées pour une exécution sur le serveur distant. Cette option peut seulement être spécifiée sur les serveurs distants, et non pas par table.

Lors de l'utilisation de l'option `extensions`, *il est de la responsabilité de l'utilisateur* que les extensions listées existent bien et se comportent de façon identique sur les serveurs local et distant. Dans le cas contraire, les requêtes pourraient échouer ou se comporter de façon inattendue.

`fetch_size` (`integer`)

Cette option indique le nombre de lignes que `postgres_fdw` doit récupérer à chaque opération de lecture. Cette option est disponible au niveau serveur et table. Une configuration spécifiée sur une table surcharge celle du serveur. La valeur par défaut est `100`.

`batch_size` (`integer`)

Cette option spécifie le nombre de lignes que `postgres_fdw` doit insérer à chaque opération d'insertion. Elle est disponible au niveau serveur et/ou table distante. L'option spécifiée sur une table surcharge celle spécifiée sur le serveur. La valeur par défaut est `1`.

Notez que le nombre réel de lignes qu'insère `postgres_fdw` en une fois dépend du nombre de colonnes et de la valeur fournie pour `batch_size`. Le batch est exécuté comme une seule requête, et le protocole libpq (qu'utilise `postgres_fdw` pour se connecter au serveur distant) limitent le nombre de paramètres dans une seule requête à 65535. Quand le nombre de colonne * `batch_size` dépasse la limite, le paramètre `batch_size` sera ajusté pour éviter une erreur.

F.38.1.5. Options d'exécution asynchrone

`postgres_fdw` supporte les exécutions asynchrones, qui exécutent plusieurs parties d'un nœud `Append` de manière concurrente plutôt qu'en série pour améliorer les performances. L'exécution peut être contrôlée en utilisant l'option suivante :

`async_capable` (`boolean`)

Cette option contrôle si `postgres_fdw` permet de parcourir de manière concurrente les tables distantes pour les exécutions asynchrones. Elle est configurable au niveau du serveur distant et/ou de la table distante. L'option spécifiée au niveau table surcharge l'option au niveau serveur. La valeur par défaut est `false`.

Pour s'assurer que les données retournées depuis un serveur distant sont cohérentes, `postgres_fdw` n'ouvrira qu'une seule connexion pour un serveur distant et exécutera les requêtes sur ce serveur séquentiellement même si de nombreuses tables distantes sont impliquées, sauf si ces tables sont sujettes à différentes correspondances d'utilisateur. Dans ce cas, il peut être plus performant de désactiver cette option pour éliminer la surcharge associée à l'exécution asynchrone des requêtes.

Une exécution asynchrone est réalisée quand un nœud `Append` contient aussi bien un ou des sous-plans exécutés de manière synchrone que un ou des sous-plans exécutés de manière asynchrone. Dans un tel cas, si les sous-plans asynchones sont ceux traités par `postgres_fdw`, les lignes des sous-plans asynchrones ne sont pas retournés avant qu'au moins un sous-plan synchrone retourne toutes ses lignes, car ce sous-plan est exécuté pendant que les sous-plans aynchrones attendent les résultats des requêtes asynchrones envoyées au serveur distant. Ce comportement pourrait changer dans une version future.

F.38.1.6. Options de gestion de la transaction

Comme décrit dans la section sur la gestion des transactions, avec `postgres_fdw`, les transactions sont gérées en créant des transactions distantes correspondantes et les sous-transactions sont gérées par la création de sous-transactions distantes correspondantes. Lorsque plusieurs transactions distantes sont impliquées dans la transaction locale courante, par défaut, `postgres_fdw` valide les transactions distantes en série quand la transaction locale est validée. Lorsque plusieurs sous-transactions distantes sont impliquées dans la sous-transaction locale courante, par défaut, `postgres_fdw` valide ces sous-transactions distantes en série quand la sous-transaction locale est validée. Les performances peuvent être améliorées avec les options suivantes !

`parallel_commit` (`boolean`)

Cette option contrôle si `postgres_fdw` valide en parallèle des transactions distantes ouvertes à partir d'un serveur distant dans une transaction locale quand la transaction locale est validée. Ce paramètre s'applique aussi aux sous-transactions locales et distantes. Cette option peut seulement être données aux serveurs distants, et non pas par table. La valeur par défaut est `false`.

Si plusieurs serveurs distants avec cette option activée sont impliqués dans une transaction locale, plusieurs transactions distantes sur ces serveurs distants sont validées en parallèle au travers des serveurs distants quand la transaction locale est validée.

Quand cette option est activée, un serveur distant avec plein de transactions distantes pourrait avoir un impact négatif sur les performances quand la transaction locale est validée.

F.38.1.7. Options de mise à jour

Par défaut, toutes les tables distantes utilisant `postgres_fdw` sont supposées comme étant modifiables. Cela peut se surcharger en utilisant l'option suivante :

`updatable` (`boolean`)

Cette option contrôle si `postgres_fdw` autorise les tables distantes à être modifiées en utilisant les commandes `INSERT`, `UPDATE` et `DELETE`. Cette option est utilisable sur une table distante ou sur un serveur distant. La configuration de cette option au niveau table surcharge celle au niveau serveur. La valeur par défaut est `true`.

Bien sûr, si la table distante n'est pas modifiable, une erreur surviendra malgré tout. L'utilisation de cette option permet principalement que l'erreur soit renvoyée localement, sans avoir à tenter l'exécution sur le serveur distant. Notez néanmoins que les vues `information_schema` indiqueront que la table distante est modifiable ou pas, suivant la configuration de cette option, et donc sans vérification du serveur distant.

F.38.1.8. Options de troncabilité

Par défaut, toutes les tables distantes utilisant `postgres_fdw` sont considérées comme troncable. Ceci peut être surchargé en utilisant l'option suivante :

`truncatable` (`boolean`)

Cette option contrôle si `postgres_fdw` permet aux tables distantes d'être tronquées en utilisant la commande `TRUNCATE`. Elle est disponible au niveau du server distant ainsi qu'à celui de la table distante. L'option au niveau table surcharge l'option au niveau serveur. La valeur par défaut est `true`.

Bien sûr, si la table distante n'est en fait pas troncable, une erreur se produira de toute façon. L'utilisation principale de cette option permet que l'erreur soit déclenchée localement sans requêter le serveur distant.

F.38.1.9. Options d'import

`postgres_fdw` est capable d'importer les définitions des tables distantes en utilisant IMPORT FOREIGN SCHEMA. Cette commande crée les définitions des tables distantes sur le serveur local, correspondant aux tables et vues présentes sur le serveur distant. Si les tables distantes à importer ont des colonnes de type défini par des utilisateurs, le serveur local doit avoir des types compatibles de même nom.

Le comportement de l'import est personnalisable avec les options suivantes (à fournir à la commande `IMPORT FOREIGN SCHEMA`) :

`import_collate` (`boolean`)

Cette option contrôle si les options `COLLATE` d'une colonne sont inclus dans les définitions des tables distantes importées à partir d'un serveur distant. La valeur par défaut est `true`. Vous pourriez avoir besoin de la désactiver si le serveur distant possède un ensemble de noms de collation différent de celui du serveur local, ce qui risque d'être le cas s'il utilise un autre système d'exploitation. Néanmoins, si vous le faites, il existe un risque réel que les collations importées dans colonnes de la table ne correspondent pas aux données sous-jacentes, résultant en un comportement anormal des requêtes.

Même quand ce paramètre est configuré à `true`, l'import des colonnes dont la collation est la valeur par défaut du serveur distant peut êre risqué. Elles seront importées avec `COLLATE "default"`, ce qui sélectionnera la collation par défaut du serveur local, qui pourrait être différent.

`import_default` (`boolean`)

Cette option contrôle si les expressions `DEFAULT` d'une colonne sont incluses dans les définitions des tables distantes importées d'un serveur distant. La valeur par défaut est `false`. Si vous activez cette option, faites attention au fait que les valeurs par défaut pourraient être calculées différemment sur le serveur local et sur le serveur distant ; par exemple, `nextval()` est une source habituelle de problèmes. La commande `IMPORT` échouera si une expression par défaut importée utilise une fonction ou un opérateur qui n'existe pas localement.

`import_generated` (`boolean`)

Cette option contrôle si les expressions `GENERATED` de colonnes sont incluses dans les définitions des tables distantes importées à partir d'un serveur distant. La valeur par défaut est `true`. La commande `IMPORT` échouera si une expression importée utilise une fonction ou un opérateur qui n'existe pas localement.

`import_not_null` (`boolean`)

Cette option contrôle si les contraintes `NOT NULL` des colonnes sont incluses dans les définitions des tables distantes importées à partir d'un serveur distant. La valeur par défaut est `true`.

Notez que les contraintes autres que `NOT NULL` ne seront jamais importées des tables distantes. Bien que PostgreSQL supporte les contraintes `CHECK` sur les tables distantes, rien n'existe pour les importer automatiquement à cause du risque qu'une expression de contrainte puisse être évaluée différemment entre les serveurs local et distant. Toute incohérence du comportement d'une contrainte `CHECK` pourrait amener des erreurs difficile à détecter dans l'optimisation des requêtes. Donc, si vous souhaitez importer les contraintes `CHECK`, vous devez le faire manuellement et vous devez vérifier la sémantique de chaque contrainte avec attention. Pour plus de détails sur le traitement des contraintes `CHECK` sur les tables distantes, voir CREATE FOREIGN TABLE.

Les tables ou tables distantes qui sont des partitions d'autres tables ne sont importées que lorsqu'elles sont explicitement spécifiées dans la clause `LIMIT TO`. Sinon, elles sont automatiquement exclues de IMPORT FOREIGN SCHEMA. Étant donné que toutes les données sont accessibles via la table partitionnée qui constitue la racine de la hiérarchie de partitionnement, l'importation des seules tables partitionnées devrait permettre d'accéder à l'ensemble des données sans créer d'objets supplémentaires.

F.38.1.10. Options de gestion de connexions

Par défaut, toutes les connexions que `postgres_fdw` établit au serveur distant sont conservées ouvertes en session locale pour une réutilisation.

`keep_connections` (`boolean`)

Cette option contrôle si `postgres_fdw` conserve les connexions au serveur distant ouvertes, permettant aux requêtes ultérieures de les réutiliser. Elle n'est disponible qu'au niveau serveur

distant. La valeur par défaut est `on`. Si elle est configurée à `off`, toutes les connexions à ce serveur distant seront supprimées à la fin de chaque transaction.

F.38.2. Fonctions

`postgres_fdw_get_connections(OUT server_name text, OUT valid boolean) retourne un setof record`

Cette fonction renvoie les noms des serveurs distants de toutes les connexions ouvertes que `postgres_fdw` a établies. Elle renvoie aussi la validité ou non de chaque connexion. `false` est retourné si la connexion au serveur distant est utilisé dans la transaction courante locale sauf si le serveur distant ou les correspondances d'utilisateur sont modifiées ou supprimées(notez que le nom de serveur d'une connexion invalide sera `NULL` si le serveur est supprimé), et ainsi de telles connexions invalides seront fermées à la fin de cette transaction. `true` est retournée dans les autres cas. S'il n'y a pas de connexions ouvertes, aucun enregistrement n'est retourné. Exemple d'utilisation de cette fonction :

```
postgres=# SELECT * FROM postgres_fdw_get_connections() ORDER BY
 1;

 server_name | valid
-------------+-------
 loopback1   | t
 loopback2   | f
```

`postgres_fdw_disconnect(server_name text) retourne un boolean`

Cette fonction supprime les connexions ouvertes qui sont établies par `postgres_fdw` depuis une session locale à un serveur distant avec un nom donné. Notez qu'il peut y avoir de multiples connexions au serveur distant donné en utilisant différentes correspondances d'utilisateur. Si les connexions sont utilisées dans la transaction courante locale, elles ne sont pas déconnectées et des messages d'alerte sont affichés. La fonction retourne `true` si elle déconnecte au moins une connexion, sinon `false`. Si aucun serveur distant avec un nom donné n'est trouvé, une erreur est affichée. Exemple d'utilisation de la fonction :

```
postgres=# SELECT postgres_fdw_disconnect('loopback1');

 postgres_fdw_disconnect
-------------------------
 t
```

`postgres_fdw_disconnect_all() retourne un boolean`

La fonction supprime toutes les connexions ouvertes établies par `postgres_fdw` depuis la session locale au serveur distant. Si les connexions sont utilisées dans la transaction locale courante, elles ne sont pas déconnectées et des messages d'alerte sont affichés. La fonction retourne `true` si elle déconnecte au moins une connexion, sinon `false`. Exemple d'utilisation de la fonction :

```
postgres=# SELECT postgres_fdw_disconnect_all();

 postgres_fdw_disconnect_all
-----------------------------
 t
```

F.38.3. Gestion des connexions

`postgres_fdw` établit une connexion au serveur distant lors de la première requête qui utilise une table distante associée avec le serveur distant. Par défaut cette connexion est conservée et ré-utilisée

pour les requêtes suivantes dans la même session. Ce comportement peut être contrôlé en utilisant l'option `keep_connections` pour un serveur distant. Si plusieurs correspondances d'utilisateur sont utilisées pour accéder au serveur distant, une connexion est établie pour chaque correspondance d'utilisateur.

Lors de changement de la définition ou suppression d'un serveur distant ou de correspondances d'utilisateur, les connexions associées sont fermées. Mais notez que si des connexions sont utilisées dans la transaction locale courante, elles sont conservées jusqu'à la fin de transaction. Les connexions fermées seront ré-établies quand cela sera nécessaire par les futures requêtes utilisant une table distante.

Une fois qu'une connexion à un serveur distant a été établie, elle est par défaut conservée jusqu'à ce que la session locale ou distante correspondante termine. Pour déconnecter une connexion explicitement, l'option `keep_connections` pour un serveur distant peut être désactivée. Il est possible aussi d'utiliser les fonctions `postgres_fdw_disconnect` et `postgres_fdw_disconnect_all`. Par exemple, ces dernières sont utiles pour fermer les connexions qui ne sont plus nécessaires, libérant ainsi les connexions sur le serveur distant.

F.38.4. Gestion des transactions

Lorsqu'une requête référence des tables sur un serveur distant, `postgres_fdw` ouvre une transaction sur le serveur distant si une transaction n'est pas déjà ouverte pour la transaction locale en cours. La transaction distante est validée ou annulée suivant que la transaction locale est validée ou annulée. Les points de sauvegardes sont gérés de la même façon en créant les points de sauvegarde correspondants.

La transaction distante utilise le niveau d'isolation `SERIALIZABLE` quand la transaction locale a le niveau `SERIALIZABLE`. Dans les cas contraires, elle utilise le niveau `REPEATABLE READ`. Ce choix assure que, si une requête réalise plusieurs parcours de table sur le serveur distant, elle obtiendra des résultats cohérents pour tous les parcours. Une conséquence est que les requêtes successives à l'intérieur d'une seule transaction verront les mêmes données provenant du serveur distant, même si des mises à jour sont réalisées en même temps avec l'activité standard du serveur. Ce comportement serait attendue de toute façon si la transaction locale utilise le niveau d'isolation `SERIALIZABLE` ou `REPEATABLE READ` mais elle pourrait surprendre pour une transaction locale en niveau `READ COMMITTED`. Une prochaine version de PostgreSQL pourrait modifier ce comportement.

Notez que `postgres_fdw` ne supporte pas actuellement de préparer la transaction distante pour une validation en deux phases (2PC).

F.38.5. Optimisation des requêtes distantes

`postgres_fdw` tente d'optimiser les requêtes distantes pour réduire la quantité de données transférées depuis les serveurs distants. Cela se fait en envoyant les clauses `WHERE` au serveur distant pour exécution, et en ne récupérant que les colonnes nécessaires pour la requête courante. Pour réduire le risque de mauvaise exécution des requêtes, les clauses `WHERE` ne sont pas envoyées au serveur distant sauf si elles utilisent seulement des types de données, opérateurs et fonctions intégrées ou appartenant à une extension listée dans l'option `extensions` du serveur distant. Les opérateurs et fonctions dans ce type de clause doivent aussi être `IMMUTABLE`. Pour une requête `UPDATE` ou `DELETE`, `postgres_fdw` tente d'optimiser l'exécution de la requête en envoyant la requête complète au serveur distant s'il n'existe pas de clauses `WHERE` pouvant être envoyées au serveur distant, pas de jointures locales pour la requête, pas de triggers `BEFORE` ou `AFTER` au niveau ligne ou de colonnes calculées automatiquement sur la table cible, et pas de contraintes `CHECK OPTION` pour les vues parentes. Dans un `UPDATE`, les expressions à affecter aux colonnes cibles doivent seulement utiliser les types de données intégrées, les opérateurs ou les fonctions `IMMUTABLE` pour réduire le risque de mauvaise exécution de la requête.

Quand `postgres_fdw` rencontre une jointure entre des tables externes sur le même serveur distant, il envoie la jointure entière au serveur distant, sauf s'il pense qu'il sera plus efficace de récupérer les lignes de chaque table individuellement ou si les références de table sont sujet à des correspondances

d'utilisateur différentes. Lors de l'envoi des clauses JOIN, il prend les mêmes précautions que mentionnées ci-dessus pour les clauses WHERE.

La requête envoyée au serveur distant pour exécution peut être examinée en utilisant EXPLAIN VERBOSE.

F.38.6. Environnement d'exécution de requêtes distantes

Dans les sessions distantes ouvertes par postgres_fdw, le paramètre search_path est configuré à pg_catalog, pour que seuls les objets internes soient visibles, sauf utilisant d'un nom de schéma. Ceci n'est pas un problème pour les requêtes générées par postgres_fdw lui-même car il fournit toujours ce type de qualification. Néanmoins, cela peut se révéler problématique pour les fonctions exécutées sur le serveur distant via des triggers ou des règles sur les tables distantes. Par exemple, si une table distante est en fait une vue, toute fonction utilisée dans cette vue sera exécutée avec le chemin de recherche restreint. Il est recommandé de qualifier tous les noms dans ce type de fonctions ou de leur attacher une option SET search_path (voir CREATE FUNCTION) pour établir le chemin de recherche attendu.

De même, postgres_fdw établie une configuration des sessions distantes pour différents paramètres :

- TimeZone est positionné à UTC
- datestyle est positionné à ISO
- IntervalStyle est positionné à postgres
- extra_float_digits est positionné à 3 pour les serveurs distants de version 9.0 et après et est positionné à 2 pour les versions plus anciennes

Ces paramètres sont moins à même d'être problématique que search_path, mais ils peuvent être gérés avec les options de fonction SET si le besoin devait se faire sentir.

Il n'est *pas* recommandé de surcharger ce comportement en modifiant la configuration de la session pour ces paramètres. Cela peut être la cause d'un mauvais fonctionnement de postgres_fdw.

F.38.7. Compatibilité entre versions

postgres_fdw peut être utilisé avec les serveurs distants de version 8.3 et ultérieures. En lecture seule, il est possible d'aller aussi loin que la 8.1. Néanmoins, une limitation est que postgres_fdw assume généralement que les fonctions et opérateurs internes immutables sont sûrs pour être envoyés au serveur distant pour exécution s'ils apparaissent dans une clause WHERE de la table distante. Du coup, une fonction interne ajoutée depuis la sortie du serveur distant pourrait être envoyée pour exécution, résultant en un message d'erreur indiquant que la fonction n'existe pas (« function does not exist ») ou une erreur similaire. Ce type d'échec peut être contourné en réécrivant la requête, par exemple en embarquant la table distance dans un sous-SELECT avec OFFSET 0 comme optimisation, et plaçant la fonction ou l'opérateur problématique en dehors du sous-SELECT.

F.38.8. Paramètres de configuration

postgres_fdw.application_name (string)

Donne une valeur pour le paramètre de configuration application_name utilisé quand postgres_fdw établie une connexion vers un serveur distant. Elle surcharge l'option application_name de l'objet serveur. Notez que la modification de ce paramètre n'affecte pas les connexions existantes jusqu'à ce qu'elles soient ré-établies.

postgres_fdw.application_name peut être toute chaîne, quelque soit sa longueur, et peut même contenir des caractères non ASCII. Néanmoins, quand il est passé et utilisé comme dans un serveur distant, notez qu'il sera tronqué à moins que NAMEDATALEN caractères et tout

caractère autre que les caractères ASCII affichables sera remplacé par des points d'interrogation (?). Voir application_name pour les détails.

Le caractère % commence une « séquence d'échappement » qui est remplacé par une information de statut comme indiqué ci-dessous. Les échappements non reconnus sont ignorés. Les autres caractères sont recopiés directement dans le nom de l'application. Notez qu'il n'est pas autorisé d'indiquer un signe plus/moins ou un littéral numérique après le % et avant l'option, pour de l'alignement ou du remplissage.

Échappement	**Effet**
%a	Nom de l'application sur le serveur local
%c	Identifiant de session sur le serveur local (voir log_line_prefix pour plus de détails)
%C	Nom de l'instance sur le serveur local (voir cluster_name pour plus de détails)
%u	Nom de l'utilisateur sur le serveur local
%d	Nom de la base sur le serveur local
%p	Identifiant du processus sur le serveur local
%%	Caractère %

Par exemple, supposons que l'utilisateur `local_user` établisse une connexion à partir de la base `local_db` vers `foreign_db` en tant qu'utilisateur `foreign_user`, la configuration `'db=%d, user=%u'` est remplacée par `'db=local_db, user=local_user'`.

F.38.9. Exemples

Voici un exemple de création d'une table distante avec `postgres_fdw`. Tout d'abord, il faut installer l'extension :

```
CREATE EXTENSION postgres_fdw;
```

Ensuite, il faut créer un serveur distant avec CREATE SERVER. Dans cet exemple, nous souhaitons nous connecter à un serveur PostgreSQL sur l'hôte `192.83.123.89` écoutant sur le port `5432`. La base de données sur le serveur distant sur laquelle la connexion est faite s'appelle `foreign_db` :

```
CREATE SERVER foreign_server
        FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host '192.83.123.89', port '5432', dbname
 'foreign_db');
```

Une correspondance d'utilisateur, définie avec CREATE USER MAPPING, est également nécessaire pour identifier le rôle qui sera utilisé sur le serveur distant :

```
CREATE USER MAPPING FOR local_user
        SERVER foreign_server
        OPTIONS (user 'foreign_user', password 'password');
```

Il est maintenant possible de créer une table distante avec CREATE FOREIGN TABLE. Dans cet exemple, nous souhaitons accéder à la table nommée `some_schema.some_table` sur le serveur distant. Le nom local pour celle-ci sera `foreign_table` :

```
CREATE FOREIGN TABLE foreign_table (
        id integer NOT NULL,
        data text
)
        SERVER foreign_server
        OPTIONS (schema_name 'some_schema', table_name
 'some_table');
```

Il est essentiel que les types de données et autres propriétés des colonnes déclarées dans `CREATE FOREIGN TABLE` correspondent à la vraie table distante. Les noms des colonnes doivent également correspondre, à moins que des options `column_name` soient attachées aux colonnes individuelles pour montrer comment elles sont nommées sur la table distante. Dans de nombreux cas, l'utilisation de `IMPORT FOREIGN SCHEMA` est préférable à la construction manuelle des tables distantes.

F.38.10. Auteur

Shigeru Hanada `<shigeru.hanada@gmail.com>`

F.39. seg

Ce module code le type de données `seg` pour représenter des segments de ligne ou des intervalles de nombres à virgule flottante. `seg` peut représenter l'incertitude des points extrêmes d'un intervalle, ce qui le rend particulièrement utile pour représenter des mesures de laboratoires.

Ce module est considéré « trusted », ce qui indique qu'il peut être installé par des utilisateurs qui n'ont pas l'attribut `SUPERUSER` mais qui ont l'attribut `CREATE` sur la base de données courante.

F.39.1. Explications

La géométrie des mesures est habituellement plus complexe qu'un point dans un continuum numérique. Une mesure est habituellement un segment de ce continuum avec des limites non définissables. Les mesures apparaissent comme des intervalles à cause de ce côté incertain et du hasard, ainsi qu'à cause du fait que la valeur mesurée peut naturellement être un intervalle indiquant certaines conditions comme une échelle de température pour la stabilité d'une protéine.

En utilisant le bon sens, il apparaît plus agréable de stocker de telles données sous la forme d'intervalle, plutôt que sous la forme d'une paire de nombres. En pratique, c'est même plus efficace dans la plupart des applications.

En allant plus loin, le côté souple des limites suggère que l'utilisation des types de données numériques traditionnels amène en fait une certaine perte d'informations. Pensez à ceci : votre instrument lit 6.50, et vous saisissez cette valeur dans la base de données. Qu'obtenez-vous en la récupérant ? Regardez :

```
test=> select 6.50 :: float8 as "pH";
 pH
---
6.5
(1 row)
```

Dans le monde des mesures, 6.50 n'est pas identique à 6.5. La différence pourrait même être critique. Les personnes ayant réalisé l'expérience écrivent habituellement (et publient) les chiffres qu'ils connaissent. 6.50 est en fait un intervalle incertain compris dans un intervalle plus grand et encore plus incertain, 6.5, le point central étant (probablement) la seule fonctionnalité commune qu'ils partagent. Nous ne voulons pas que de telles différences de données apparaissent de façon identique.

La conclusion ? il est agréable d'avoir un type de données spécial qui peut enregistrer les limites d'un intervalle avec une précision variable arbitraire. Variable dans le sens où chaque élément de données enregistre sa propre précision.

Vérifiez ceci :

```
test=> select '6.25 .. 6.50'::seg as "pH";
          pH
------------
6.25 .. 6.50
(1 row)
```

F.39.2. Syntaxe

La représentation externe d'un intervalle se forme en utilisant un ou deux nombres à virgule flottante joint par l'opérateur d'échelle (`..` ou `...`). Sinon, il peut être spécifié comme un point central plus ou moins une déviation. Des indicateurs optionnels (<, > et ~) peuvent aussi être stockés. (Néanmoins, ces indicateurs sont ignorés par la logique interne.) Tableau F.26 donne un aperçu des représentations autorisées ; Tableau F.27 montre quelques exemples.

Dans Tableau F.26, *x*, *y* et *delta* dénotent des nombres à virgule flottante. *x* et *y*, mais pas *delta*, peuvent être précédés par un indicateur de certitude :

Tableau F.26. Représentations externes de `seg`

x	Valeur seule (intervalle de longueur zéro)
x `..` *y*	Intervalle de *x* à *y*
x `(+-)` *delta*	Intervalle de *x* - *delta* à *x* + *delta*
x `..`	Intervalle ouvert avec une limite inférieure *x*
`..` *x*	Intervalle ouvert avec une limite supérieure *x*

Tableau F.27. Exemples d'entrées valides de type `seg`

`5.0`	Crée un segment de longueur zéro (un point si vous préférez)
`~5.0`	Crée un segment de taille nulle et enregistre ~ dans les données. ~ est ignoré par les opérations `seg` mais conservé en commentaire.
`<5.0`	Crée un point à 5.0. < est ignoré mais conservé en commentaire.
`>5.0`	Crée un point à 5.0. > est ignoré mais conservé en commentaire.
`5(+-)0.3`	Crée un intervalle `4.7 .. 5.3`. Notez que la notation `(+-)` n'est pas conservée.
`50 ..`	Tout ce qui supérieur ou égal à 50
`.. 0`	Tout ce qui est inférieur ou égal à 0
`1.5e-2 .. 2E-2`	Crée un intervalle `0.015 .. 0.02`
`1 ... 2`	Identique à `1...2`, ou `1 .. 2`, ou `1..2` (les espaces autour de l'opérateur d'échelle sont ignorés)

Comme `...` est largement utilisé dans les sources de données, il est autorisé comme autre orthographe possible de `...`. Malheureusement, ceci crée une ambiguité pour l'analyseur : la limite supérieure dans `0...23` est `23` ou `0.23`. Ceci se résout en réclamant au moins un chiffre avant le point décimal dans tous les nombres de type `seg`.

Comme vérification, `seg` rejette les intervalles dont la limite inférieure est supérieure à la limite supérieure, par exemple `5 .. 2`.

F.39.3. Précision

Les valeurs `seg` sont stockés en interne sous la forme de paires de nombres en virgule flottante de 32 bits. Cela signifie que les nombres avec plus de sept chiffres significatifs sont tronqués.

Les nombres avec moins ou avec exactement sept chiffres significatifs conservent leur précision originale. C'est-à-dire que, si votre requête renvoie 0.00, vous serez sûr que les zéros qui suivent ne sont pas des conséquences du formatage : elles reflètent la précision de la donnée originale. Le nombre de zéro au début n'affectent pas la précision : deux chiffres significatifs sont considérés pour la valeur 0.0067.

F.39.4. Utilisation

Le module `seg` inclut une classe d'opérateur pour les index GiST dans le cas des valeurs `seg`. Les opérateurs supportés par la classe d'opérateur GiST are shown in Tableau F.28.

Tableau F.28. Opérateurs GiST du type Seg

Opérateur Description
`seg << seg → boolean` Est-ce que le premier `seg` est entièrement à gauche du second ? [a, b] << [c, d] est vrai si b < c.
`seg >> seg → boolean` Est-ce que le premier `seg` est entièrement à droite du second ? [a, b] >> [c, d] est vrai si a > d.
`seg &< seg → boolean` Est-ce que le premier `seg` ne s'étend pas à droite du second ? [a, b] &< [c, d] est vrai si b <= d.
`seg &> seg → boolean` Est-ce que le premier `seg` ne s'étend pas à gauche du second ? [a, b] &> [c, d] est vrai si a >= c.
`seg = seg → boolean` Est-ce que les deux `seg` sont égaux ?
`seg && seg → boolean` Est-ce que les deux `seg` se chevauchent ?
`seg @> seg → boolean` Est-ce que le premier `seg` contient le second ?
`seg <@ seg → boolean` Est-ce que le premier `seg` est contenu dans le second ?

En plus des opérateurs ci-dessus, les opérateurs de comparaison habituels de Tableau 9.1 sont disponibles pour le type `seg`. Ces opérateurs comparent en premier (a) à (c) et, s'ils sont égaux, comparent (b) à (d). Cela fait un bon tri dans la plupart des cas, ce qui est utile si vous voulez utiliser ORDER BY avec ce type.

F.39.5. Notes

Pour des exemples d'utilisation, voir les tests de régression `sql/seg.sql`.

Le mécanisme qui convertit `(+-)` en échelles standards n'est pas entièrement précis pour déterminer le nombre de chiffres significatifs pour les limites. Par exemple, si vous ajoutez un chiffre supplémentaire à la limite basse si l'intervalle résultat inclut une puissance de dix :

```
postgres=> select '10(+-)1'::seg as seg;
      seg
---------
9.0 .. 11             -- should be: 9 .. 11
```

La performance d'un index R-tree peut dépendre largement de l'ordre des valeurs en entrée. Il pourrait être très utile de trier la table en entrée sur la colonne `seg` ; voir le script `sort-segments.pl` pour un exemple.

F.39.6. Crédits

Auteur original : Gene Selkov, Jr. `<selkovjr@mcs.anl.gov>`, Mathematics and Computer Science Division, Argonne National Laboratory.

Mes remerciements vont principalement au professeur Joe Hellerstein (http://db.cs.berkeley.edu/jmh/) pour avoir élucidé l'idée centrale de GiST (http://gist.cs.berkeley.edu/). Mes remerciements aussi aux développeurs de PostgreSQL pour m'avoir permis de créer mon propre monde et de pouvoir y vivre sans perturbation. Argonne Lab et le département américain de l'énergie ont aussi toute ma gratitude pour les années de support dans ma recherche sur les bases de données.

F.40. sepgsql

`sepgsql` est un module chargeable ajoutant le support des contrôles d'accès par label basé sur la politique de sécurité de SELinux.

> **Avertissement**
>
> L'implémentation actuelle a des limitations importantes et ne force pas le contrôle d'accès pour toutes les actions. Voir Section F.40.7.

F.40.1. Aperçu

Ce module s'intègre avec SELinux pour fournir une couche de vérification de sécurité supplémentaire qui va au-delà de ce qui est déjà fourni par PostgreSQL. De la perspective de SELinux, ce module permet à PostgreSQL de fonctionner comme un gestionnaire d'objet en espace utilisateur. Chaque accès à une table ou à une fonction initié par une requête DML sera vérifié par rapport à la politique de sécurité du système. Cette vérification est en plus des vérifications de droits SQL habituels effectuées par PostgreSQL.

Les décisions de contrôle d'accès de SELinux sont faites en utilisant les labels de sécurité qui sont représentés par des chaînes comme `system_u:object_r:sepgsql_table_t:s0`. Chaque décision de contrôle d'accès implique deux labels : celui de l'utilisateur tentant de réaliser l'action et celui de l'objet sur lequel l'action est réalisée. Comme ces labels peuvent être appliqués sur tout type d'objet, les décisions de contrôle d'accès pour les objets stockés dans la base peuvent être (et avec ce module, sont) sujets au même critère général utilisé pour les objets de tout type (par exemple les fichiers). Ce concept a pour but de permettre la mise en place d'une politique centralisée pour protéger l'information quelle que soit la façon dont l'information est stockée.

L'instruction `SECURITY LABEL` permet d'affecter un label de sécurité à un objet de la base de données.

F.40.2. Installation

`sepgsql` peut seulement être utilisé sur Linux 2.6.28 ou ultérieur, avec SELinux activé. Il n'est pas disponible sur les autres plateformes. Vous aurez aussi besoin de libselinux ou ultérieur et de selinux-policy 2.1.10 ou ultérieur (même si certaines distributions peuvent proposer les règles nécessaires dans des versions antérieures de politique).

La commande `sestatus` vous permet de vérifier le statut de SELinux. Voici un affichage standard :

```
$ sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 24
Policy from config file:        targeted
```

Si SELinux est désactivé ou non installé, vous devez tout d'abord configurer ce produit avant d'utiliser ce module.

Pour construire ce module, ajoutez l'option `--with-selinux` dans votre commande `configure` lors de la compilation de PostgreSQL. Assurez-vous que le RPM `libselinux-devel` est installé au moment de la construction.

Pour utiliser ce module, vous devez ajouter `sepgsql` dans le paramètre shared_preload_libraries du fichier `postgresql.conf`. Le module ne fonctionnera pas correctement s'il est chargé d'une autre façon. Une fois que le module est chargé, vous devez exécuter `sepgsql.sql` dans chaque base de données. Cela installera les fonctions nécessaires à la gestion des labels de sécurité et affectera des labels initiaux de sécurité.

Voici un exemple montrant comment initialiser un répertoire de données avec les fonctions `sepgsql` et les labels de sécurité installés. Ajustez les chemins de façon appropriée pour que cela corresponde à votre installation :

```
$ export PGDATA=/path/to/data/directory
 $ initdb
 $ vi $PGDATA/postgresql.conf
  modifiez
    #shared_preload_libraries = ''               # (change
 requires restart)
  en
    shared_preload_libraries = 'sepgsql'         # (change
 requires restart)
 $ for DBNAME in template0 template1 postgres; do
    postgres --single -F -c exit_on_error=true $DBNAME \
      </usr/local/pgsql/share/contrib/sepgsql.sql >/dev/null
   done
```

Notez que vous pourriez voir les notifications suivantes, suivant la combinaison de versions particulières de libselinux et de selinux-policy.

```
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 33 has
 invalid object type db_blobs
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 36 has
 invalid object type db_language
```

```
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 37 has
 invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 38 has
 invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 39 has
 invalid object type db_language
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 40 has
 invalid object type db_language
```

Ces messages ne sont graves et peuvent être ignorés sans conséquence.

Si le processus d'installation se termine sans erreurs, vous pouvez commencer à lancer le serveur normalement.

F.40.3. Tests de régression

Dû à la nature de SELinux, exécuter les tests de régression pour `sepgsql` nécessite quelques étapes de configuration supplémentaires, certaines se faisant en tant qu'utilisateur root. Les tests de régression ne seront pas exécutés par une commande `make check` ou `make installcheck` ordinaire ; vous devez faire la configuration puis appeler le script de test manuellement. Les tests s'exécuteront dans le répertoire `contrib/sepgsql` du répertoire des sources de PostgreSQL, préalablement configuré. Bien que cela nécessite un arbre de construction, les tests sont conçus pour être exécutés par un serveur déjà installé, donc comparable à `make installcheck`, et non pas `make check`.

Tout d'abord, configurez `sepgsql` dans une base de données fonctionnelle d'après les instructions comprises dans Section F.40.2. Notez que l'utilisateur du système d'exploitation doit être capable de se connecter à la base de données en tant que superutilisateur sans authentification par mot de passe.

Ensuite, construisez et installez le paquet de politique pour les tests de régression. Le fichier `sepgsql-regtest` est un paquet de politique à but spécial. Il fournit un ensemble de règles à autoriser pendant les tests de régression. Il doit être construit à partir du fichier source de politique `sepgsql-regtest.te`, ce qui se fait en utilisant `make` avec un fichier Makefile fourni par SELinux. Vous aurez besoin de localiser le Makefile approprié sur votre système ; le chemin affiché ci-dessous est seulement un exemple. (Ce fichier Makefile est habituellement fourni avec le RPM `selinux-policy-devel` ou `selinux-policy`.) Une fois construit, installez ce paquet de politique en utilisant la commande `semodule`, qui charge les paquets de politique fournis dans le noyau. Si ce paquet est correctement installé, `semodule -l` doit lister `sepgsql-regtest` comme un paquet de politique disponible :

```
$ cd .../contrib/sepgsql
$ make -f /usr/share/selinux/devel/Makefile
$ sudo semodule -u sepgsql-regtest.pp
$ sudo semodule -l | grep sepgsql
sepgsql-regtest 1.07
```

Pour des raisons de sécurité, les règles de `sepgsql-regtest` ne sont pas activés par défaut. Le paramètre `sepgsql_regression_test_mode` active les règles pour le lancement des tests de régression. Il peut être activé en utilisant la commande `setsebool` :

```
$ sudo setsebool sepgsql_regression_test_mode on
$ getsebool sepgsql_regression_test_mode
```

Ensuite, vérifiez que votre shell est exécuté dans le domaine `unconfined_t` :

```
$ ./test_sepgsql
```

Ce script tentera de vérifier que vous avez fait correctement toutes les étapes de configuration, puis il lancera les tests de régression du module `sepgsql`.

Une fois les tests terminés, il est recommandé de désactiver le paramètre `sepgsql_regression_test_mode` :

```
$ sudo setsebool sepgsql_regression_test_mode off
```

Vous pouvez préférer supprimer complètement la politique `sepgsql-regtest` :

```
$ sudo semodule -r sepgsql-regtest
```

F.40.4. Paramètres GUC

`sepgsql.permissive` (`boolean`)

Ce paramètre active sepgsql pour qu'il fonctionne en mode permissif, quelle que soit la configuration du système. La valeur par défaut est off. Ce paramètre est configurable dans le fichier `postgresql.conf` et sur la ligne de commande.

Quand ce paramètre est activé, sepgsql fonctionne en mode permissif, même si SELinux fonctionne en mode forcé. Ce paramètre est utile principalement pour des tests.

`sepgsql.debug_audit` (`boolean`)

Ce paramètre active l'affichage de messages d'audit quelle que soit la configuration de la politique. La valeur par défaut est off, autrement dit les messages seront affichés suivant la configuration du système.

La politique de sécurité de SELinux a aussi des règles pour contrôler la trace des accès. Par défaut, les violations d'accès sont tracées, contrairement aux accès autorisés.

Ce paramètre force l'activation de toutes les traces, quelle que soit la politique du système.

F.40.5. Fonctionnalités

F.40.5.1. Classes d'objet contrôlé

Le modèle de sécurité SELinux décrit toutes les règles de contrôle d'accès comme des relations entre une entité sujet (habituellement le client d'une base) et une entité objet (tel que l'objet base de données). Les deux sont identifiés par un label de sécurité. Si un accès à un objet sans label est tenté, l'objet est traité comme si le label `unlabeled_t` lui est affecté.

Actuellement, `sepgsql` autorise l'affectation de label de sécurité aux schémas, tables, colonnes, séquences, vues et fonctions. Quand `sepgsql` est en cours d'utilisation, des labels de sécurité sont automatiquement affectés aux objets de la base au moment de leur création. Ce label est appelé un label de sécurité par défaut et est configuré par la politique de sécurité du système, qui prend en entrée le label du créateur, le label affecté à l'objet parent du nouvel objet et en option le nom de l'objet construit.

Un nouvel objet base de données hérite en gros du label de sécurité de l'objet parent, sauf quand la politique de sécurité a des règles spéciales, connues sous le nom de règles de transition, auquel cas

un label différent est affecté. Pour les schémas, l'objet parent est la base de données ; pour les tables, séquences, vues et fonctions, il s'agit du schéma ; pour les colonnes, il s'agit de la table.

F.40.5.2. Droits DML

Pour les tables, `db_table:select`, `db_table:insert`, `db_table:update` ou `db_table:delete` sont vérifiés pour toutes les tables cibles référencées, suivant l'ordre de l'instruction. De plus, `db_table:select` est aussi vérifié pour toutes les tables qui contiennent des colonnes référencées dans la clause `WHERE` ou `RETURNING`, comme source de données d'un `UPDATE`, et ainsi de suite.

Les droits au niveau colonne seront aussi vérifiés pour chaque colonne référencée. `column_db:select` est vérifié sur les colonnes lues en utilisant `SELECT`, mais aussi celles référencées dans d'autres instructions DML ; `column_db:update` ou `column_db:insert` sear aussi vérifié pour les colonnes modifiées par `UPDATE` ou `INSERT`.

Bien sûr, il vérifie aussi `db_column:update` ou `db_column:insert` sur la colonne en cours de modification par `UPDATE` ou `INSERT`.

Par exemple :

```
UPDATE t1 SET x = 2, y = func1(y) WHERE z = 100;
```

Ici, `column_db:update` sera vérifié pour `t1.x` car elle est mise à jour, `column_db:{select update}` sera vérifié pour `t1.y` car elle est à la fois mise à jour et référencée, et `column_db:select` sera vérifié pour `t1.z` car elle est référencée. `db_table:{select update}` vérifiera aussi la table.

Pour les séquences, `db_sequence:get_value` est vérifié quand nous référençons un objet séquence en utilisant `SELECT` ; néanmoins, notez que nous ne vérifions pas les droits d'exécution sur les fonctions correspondantes, par exemple `lastval()`.

Pour les vues, `db_view:expand` devrait être vérifié, et ensuite tous les autres droits des objets dus à l'aplatissement de la vue, individuellement.

Pour les fonctions, `db_procedure:{execute}` sera vérifié quand un utilisateur essaie d'exécuter une fonction dans une requête ou en utilisant l'appel « fast-path ». Si cette fonction est déclarée comme étant de confiance, il vérifie aussi le droit `db_procedure:{entrypoint}` pour s'assurer qu'il peut s'exécuter comme un point d'entrée d'une procédure de confiance.

Pour accéder à tout objet d'un schéma, le droit `db_schema:search` est requis sur le schéma contenant l'objet. Quand un objet est référencé sans le nom du schéma, les schémas qui n'ont pas ce droit ne seront pas recherchés (exactement le même comportement que l'absence du droit `USAGE` sur le schéma). Si une qualification explicite du schéma est présent, une erreur surviendra si l'utilisateur n'a pas le droit requis sur le schéma nommé.

Le client doit être autorisé à accéder à toutes les tables et colonnes référencées, même si elles proviennent de vues qui ont été aplaties, pour pouvoir appliquer des règles de contrôles d'accès cohérentes indépendamment de la manière dont le contenu des tables est référencé.

Le système des droits de la base, par défaut, autorise les superutilisateurs de la base à modifier les catalogues systèmes en utilisant des commandes DML, et de référencer ou modifier les tables TOAST. Ces opérations sont interdites quand `sepgsql` est activé.

F.40.5.3. Droits DDL

SELinux définit plusieurs droits pour contrôler les opérations standards pour chaque type d'objet : création, modification, suppression et changement du label de sécurité. De plus, certains types d'objet

ont des droits spéciaux pour contrôler leurs opérations caractéristiques : ajout ou suppression d'entrées dans un schéma particulier.

Créer un objet de bases de données nécessite le droit `create`. SELinux acceptera ou refusera ce droit en se basant sur le label de sécurité du client et sur le label de sécurité proposé pour le nouvel objet. Dans certains cas, des droits supplémentaires sont demandés :

- `CREATE DATABASE` requiert en plus le droit `getattr` pour la base de données source ou modèle.
- Créer un schéma requiert en plus le droit `add_name` sur le schéma parent.
- Créer une table requiert en plus le droit de créer chaque colonne de la table, tout comme si chaque colonne de la table était un objet séparé de haut-niveau.
- Créer une fonction marquée `LEAKPROOF` requiert en plus le droit `install`. (Ce droit est aussi vérifié quand `LEAKPROOF` est configuré pour une fonction existante.)

Quand la commande `DROP` est exécutée, `drop` sera vérifié sur l'objet qui doit être supprimé. Les droits seront aussi vérifiés pour les objets supprimés indirectement via `CASCADE`. La suppression des objets contenus dans un schéma particulier (tables, vues, séquences et fonctions) nécessite habituellement `remove_name` sur le schéma.

Quand la commande `ALTER` est exécutée, `setattr` sera vérifié sur l'objet en cours de modification pour chaque type d'objet, sauf pour les objets sous-jacents comme les index ou les triggers d'une table. Pour ces derniers, les droits sont vérifiés sur l'objet parent. Dans certains cas, des droits supplémentaires sont réclamés :

- Déplacer un objet vers un nouveau schéma réclame en plus le droit `remove_name` sur l'ancien schéma et le droit `add_name` sur le nouveau schéma.
- Configurer l'attribut `LEAKPROOF` sur une fonction requiert le droit `install`.
- Utiliser `SECURITY LABEL` sur un objet requiert en plus le droit `relabelfrom` pour l'objet en conjonction avec son ancien label et le droit `relabelto` pour l'objet en conjonction avec son nouveau label. (Dans les cas où plusieurs fournisseurs de label sont installés et que l'utilisateur essaie de configurer un label de sécurité mais qui est géré par SELinux, seul `setattr` peut être vérifié ici. Cela ne peut pas se faire actuellement à cause des restrictions de l'implémentation.)

F.40.5.4. Procédures de confiance

Les procédures de confiance sont similaires aux fonctions dont la sécurité est définie à la création ou aux commandes set-uid. SELinux propose une fonctionnalité qui permet d'autoriser un code de confiance à s'exécuter en utilisant un label de sécurité différent de celui du client, généralement pour donner un accès hautement contrôlé à des données sensibles (par exemple, des lignes peuvent être omises ou la précision des valeurs stockées peut être réduite). Que la fonction agisse ou pas comme une procédure de confiance est contrôlé par son label de sécurité et la politique de sécurité du système d'exploitation. Par exemple :

```
postgres=# CREATE TABLE customer (
               cid     int primary key,
               cname   text,
               credit  text
           );
CREATE TABLE
postgres=# SECURITY LABEL ON COLUMN customer.credit
               IS 'system_u:object_r:sepgsql_secret_table_t:s0';
SECURITY LABEL
postgres=# CREATE FUNCTION show_credit(int) RETURNS text
```

```
            AS 'SELECT regexp_replace(credit, ''-[0-9]+$'', ''-
xxxx'', ''g'')
                        FROM customer WHERE cid = $1'
          LANGUAGE sql;
CREATE FUNCTION
postgres=# SECURITY LABEL ON FUNCTION show_credit(int)
              IS
 'system_u:object_r:sepgsql_trusted_proc_exec_t:s0';
SECURITY LABEL
```

Les opérations ci-dessus doivent être réalisées par un utilisateur administrateur.

```
postgres=# SELECT * FROM customer;
ERROR:  SELinux: security policy violation
postgres=# SELECT cid, cname, show_credit(cid) FROM customer;
 cid | cname  |     show_credit
-----+--------+---------------------
   1 | taro   | 1111-2222-3333-xxxx
   2 | hanako | 5555-6666-7777-xxxx
(2 rows)
```

Dans ce cas, un utilisateur standard ne peut pas faire référence à `customer.credit` directement mais une procédure de confiance comme `show_credit` lui permet d'afficher le numéro de carte de crédit des clients, avec quelques chiffres masqués.

F.40.5.5. Transitions de domaine dynamique

Il est possible d'utiliser la fonctionnalité de transition de domaine dynamique de SELinux pour basculer le label de sécurité du processus client, le domaine client, vers un nouveau contexte, s'il s'avère que c'est autorisé par la politique de sécurité. Le domaine client a besoin du droit `setcurrent` ainsi que du droit `dyntransition` de l'ancien domaine vers le nouveau domaine.

Les transitions de domaine dynamique doivent être considérées avec attention car elles permettent aux utilisateurs de basculer leur label, et du coup leurs droits, quand ils le souhaitent, plutôt que (dans le cas d'une procédure de confiance) lorsque c'est demandé par le système. Du coup, le droit `dyntransition` est seulement considéré sûr quand il est utilisé pour basculer vers un domaine avec un plus petit ensemble de droits que le domaine original. Par exemple :

```
regression=# select sepgsql_getcon();
                    sepgsql_getcon
-------------------------------------------------------
 unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
(1 row)

regression=# SELECT
 sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-
s0:c1.c4');
 sepgsql_setcon
----------------
 t
(1 row)

regression=# SELECT
 sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-
s0:c1.c1023');
```

```
ERROR:  SELinux: security policy violation
```

Dans l'exemple ci-dessus, nous sommes autorisés à basculer du gros intervalle MCS `c1.c1023` vers l'intervalle `c1.c4` beaucoup plus petit. Par contre, la bascule inverse est interdite.

Une combinaison de transition de domaine dynamique et de procédure de confiance permet un cas d'utilisation intéressant qui correspond au cycle de vie typique d'un processus pour un logiciel de pooling de connexions. Même si votre pooler de connexions n'est pas autorisé à exécuter la plupart des commandes SQL, vous pouvez l'autoriser à basculer le label de sécurité du client en utilisant la fonction `sepgsql_setcon()` à l'intérieur d'une procédure de confiance. Après cela, cette session aura les droits de l'utilisateur cible plutôt que ceux du pooler de connexions. Le pooler de connexions peut ensuite annuler le changement du label de sécurité en utilisant de nouveau `sepgsql_setcon()` avec l'argument `NULL`, encore une fois en l'appelant à partir d'une procédure de confiance avec les droits appropriés. Le point ici est que seule la procédure de confiance a réellement le droit de modifier le label de sécurité en cours et ne le fait que si autorisé. Bien sûr, pour un traitement sécurisé, le stockage des autorisations (table, définition de procédure, ou autres) doit être protégé des accès non autorisés.

F.40.5.6. Divers

Nous rejetons la commande `LOAD` car tout module chargé pourrait facilement court-circuiter la politique de sécurité.

F.40.6. Fonctions Sepgsql

Tableau F.29 affiche la liste des fonctions disponibles.

Tableau F.29. Fonctions Sepgsql

Fonction Description
`sepgsql_getcon` () → `text` Renvoie le domaine client, le label de sécurité actuel du client.
`sepgsql_setcon` (`text`) → `boolean` Bascule le domaine client de la session actuelle sur un autre domaine, si cela est autorisé par la politique de sécurité. Cette fonction accepte aussi `NULL` en entrée comme demande de transaction vers le domaine original du client.
`sepgsql_mcstrans_in` (`text`) → `text` Traduit l'intervalle MLS/MCS donné en un format brut si le démon mcstrans est en cours d'exécution.
`sepgsql_mcstrans_out` (`text`) → `text` Traduit l'intervalle MCS/MCS brut donné en son format qualifié si le démon mcstrans est en cours d'exécution.
`sepgsql_restorecon` (`text`) → `boolean` Configure les labels de sécurité initiaux pour tous les objets à l'intérieur de la base de données actuelle. L'argument peut être NULL ou le nom d'un specfile à utiliser comme alternative du fichier système par défaut.

F.40.7. Limitations

Droits DDL

À cause de restrictions d'implémentations, certaines opérations DDL ne vérifient pas les droits.

Droits DCL

À cause de restrictions d'implémentations, les droits DCL ne vérifient pas les droits.

Contrôle d'accès au niveau ligne

PostgreSQL propose le contrôle d'accès au niveau ligne. Cependant, `sepgsql` ne le supporte pas.

Canaux cachés

`sepgsql` n'essaie pas de cacher l'existence d'un objet particulier, même si l'utilisateur n'est pas autorisé à y accéder. Par exemple, nous pouvons inférer l'existence d'un objet invisible suite à un conflit de clé primaire, à des violations de clés étrangères et ainsi de suite, même si nous ne pouvons pas accéder au contenu de ces objets. L'existence d'une table secrète ne peut pas être caché. Nous ne faisons que verrouiller l'accès à son contenu.

F.40.8. Ressources externes

SE-PostgreSQL Introduction[10]

Cette page wiki fournit un bref aperçu, le concept de la sécurité, l'architecture, l'administration et les fonctionnalités futures.

SELinux User's and Administrator's Guide[11]

Ce document fournit une connaissance large pour administrer SELinux sur vos systèmes. Il cible principalement les systèmes d'exploitation Red Hat mais n'y est pas limité.

Fedora SELinux FAQ[12]

Ce document répond aux questions fréquemment posées sur SELinux. Il cible principalement Fedora mais n'y est pas limité.

F.40.9. Auteur

KaiGai Kohei `<kaigai@ak.jp.nec.com>`

F.41. spi

Le module spi fournit plusieurs exemples fonctionnels d'utilisation de l'interface de programmation du serveur (*Server Programming Interface*) (SPI) et des triggers. Bien que ces fonctions aient un intérêt certain, elles sont encore plus utiles en tant qu'exemples à modifier pour atteindre ses propres buts. Les fonctions sont suffisamment généralistes pour être utilisées avec une table quelconque, mais la création d'un trigger impose que les noms des tables et des champs soient précisés (comme cela est décrit ci-dessous).

Chaque groupe de fonctions décrit ci-dessous est fourni comme une extension installable séparément.

F.41.1. refint -- fonctions de codage de l'intégrité référentielle

`check_primary_key()` et `check_foreign_key()` sont utilisées pour vérifier les contraintes de clé étrangère. (Cette fonctionnalité est dépassée depuis longtemps par le mécanisme interne, mais le module conserve un rôle d'exemple. Ce module sera supprimée dans PostgreSQL 20.)

[10] https://wiki.postgresql.org/wiki/SEPostgreSQL
[11] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/index
[12] https://fedoraproject.org/wiki/SELinux_FAQ

> **Note**
>
> `refint` requiert un modèle d'utilisation sécurisée du schéma et les types de données où l'opérateur d'égalité est nommé `=`.

`check_primary_key()` vérifie la table de référence. Pour l'utiliser, on crée un trigger `BEFORE INSERT OR UPDATE` qui utilise cette fonction sur une table référençant une autre table. En arguments du trigger, on trouve : le nom de la colonne de la table référençant qui forme la clé étrangère, le nom de la table référencée et le nom de la colonne de la table référencée qui forme la clé primaire/unique. Il peut y avoir plusieurs colonnes. Pour gérer plusieurs clés étrangères, on crée un trigger pour chaque référence.

> **Note**
>
> Les arguments de nom de la table et des colonnes *référencés* à `check_primary_key()` sont copiés tels quels dans les requêtes SQL générées en interne et, de ce fait, doivent être entre guillemets doubles par l'utilisateur si nécessaire dans la commande `CREATE TRIGGER`. Voir Section 4.1.1 pour plus d'informations sur la mise entre guillemets des identifiants SQL. Inversement, les arguments des noms de colonne de la table *référençante* ne doivent pas être entre guillemets doubles. Voir l'exemple suivant d'utilisation correcte de `check_primary_key()` :
>
> ```
> CREATE TRIGGER mytrigger
> BEFORE INSERT OR UPDATE ON referencing_table
> FOR EACH ROW EXECUTE PROCEDURE
> check_primary_key (
>     'column A', 'column B',         -- colonnes de la table
>  référençante
>     'myschema."referenced table"',  -- table référencée
>     '"column A"', '"column B"'      -- colonnes de la table
>  référencée
> );
> ```

`check_foreign_key()` vérifie la table référencée. Pour l'utiliser, on crée un trigger `BEFORE DELETE OR UPDATE` qui utilise cette fonction sur une table référencée par d'autres tables. En arguments du trigger, on trouve : le nombre de tables référençant pour lesquelles la fonction réalise la vérification, l'action à exécuter si une clé de référence est trouvée (`cascade` -- pour supprimer une ligne qui référence, `restrict` -- pour annuler la transaction si des clés de référence existent, `setnull` -- pour initialiser les champs des clés référençant à NULL), les noms des colonnes de la table surveillées par le trigger, colonnes qui forment la clé primaire/unique, puis le nom de la table référençant et les noms des colonnes (répétés pour autant de tables référençant que cela est précisé par le premier argument). Les colonnes de clé primaire/unique doivent être marquées NOT NULL et posséder un index d'unicité.

> **Note**
>
> Le nom de la table et colonne *référençante* en arguments de `check_foreign_key()` sont copiés tels quels dans les requêtes SQL générées en interne et doivent de ce fait être mises entre guillemets doubles par l'utilisateur si nécessaire dans la commande `CREATE TRIGGER`. Voir Section 4.1.1 pour plus d'informations sur la mise entre guillemets des identifiants SQL. À l'inverse, le nom de colonne *référencée* en argument ne doit pas être mis entre guillemets doubles. Voir l'exemple suivant d'utilisation correcte de `check_foreign_key()` :

```
CREATE TRIGGER mytrigger
BEFORE DELETE OR UPDATE ON referenced_table
FOR EACH ROW EXECUTE PROCEDURE
check_foreign_key (
    1,                              -- nombre de tables
 référençantes
    'cascade',                      -- action
    'column A', 'column B',         -- colonnes référencées
    'myschema."referencing table"', -- table référençante
    '"column A"', '"column B"'      -- colonnes de la table
 référençante
);
```

Il y a des exemples dans `refint.example`.

F.41.2. autoinc -- fonctions pour l'incrémentation automatique d'un champ

`autoinc()` est un trigger qui stocke la prochaine valeur d'une séquence dans un champ de type integer. Cela recouvre quelque peu la fonctionnalité interne de la colonne « serial », mais ce n'est pas strictement identique : `autoinc()` surcharge les tentatives de substitution d'une valeur différente pour ce champ lors des insertions et, optionnellement, peut aussi être utilisé pour incrémenter le champ lors des mises à jour.

Pour l'utiliser, on crée un trigger `BEFORE INSERT` (ou en option `BEFORE INSERT OR UPDATE`) qui utilise cette fonction. Le trigger accepte deux arguments : le nom de la colonne de type integer à modifier et le nom de la séquence qui fournit les valeurs. (En fait, plusieurs paires de noms peuvent être indiquées pour actualiser plusieurs colonnes.)

Un exemple est fourni dans `autoinc.example`.

F.41.3. insert_username -- fonctions pour tracer les utilisateurs qui ont modifié une table

`insert_username()` est un trigger qui stocke le nom de l'utilisateur courant dans un champ texte. C'est utile pour savoir quel est le dernier utilisateur à avoir modifié une ligne particulière d'une table.

Pour l'utiliser, on crée un trigger `BEFORE INSERT` et/ou `UPDATE` qui utilise cette fonction. Le trigger prend pour seul argument le nom de la colonne texte à modifier.

Un exemple est fourni dans `insert_username.example`.

F.41.4. moddatetime -- fonctions pour tracer la date et l'heure de la dernière modification

`moddatetime()` est un trigger qui stocke la date et l'heure de la dernière modification dans un champ de type `timestamp`. C'est utile pour savoir quand a eu lieu la dernière modification sur une ligne particulière d'une table.

Pour l'utiliser, on crée un trigger `BEFORE UPDATE` qui utilise cette fonction. Le trigger prend pour seul argument le nom de la colonne de type à modifier. La colonne doit être de type `timestamp` ou `timestamp with time zone`.

Un exemple est fourni dans `moddatetime.example`.

F.42. sslinfo

Le module `sslinfo` fournit des informations sur le certificat SSL que le client actuel a fourni lors de sa connexion à PostgreSQL. Le module est inutile (la plupart des fonctions renvoient NULL) si la connexion actuelle n'utilise pas SSL.

Certaines des informations disponibles dans ce module peuvent aussi être obtenues en utilisant la vue système interne `pg_stat_ssl`.

Cette extension ne se construira pas du tout sauf si l'installation était configurée avec `--with-ssl=openssl`.

F.42.1. Fonctions

`ssl_is_used() returns boolean`

Renvoie TRUE si la connexion actuelle au serveur utilise SSL.

`ssl_version() returns text`

Renvoie le nom du protocole utilisé pour la connexion SSL (c'est-à-dire TLSv1.0, TLSv1.1, TLSv1.2 ou TLSv1.3).

`ssl_cipher() returns text`

Renvoie le nom du chiffrement utilisé pour la connexion SSL (par exemple DHE-RSA-AES256-SHA).

`ssl_client_cert_present() returns boolean`

Renvoie TRUE si le client actuel a présenté un certificat client SSL au serveur. (Le serveur pourrait être configuré pour réclamer un certificat client.)

`ssl_client_serial() returns numeric`

Renvoie un numéro de série du certificat actuel du client. La combinaison du numéro de série de certificat et du créateur du certificat garantit une identification unique du certificat (mais pas son propriétaire -- le propriétaire doit régulièrement changer ses clés et obtenir de nouveaux certificats à partir du créateur).

Donc, si vous utilisez votre propre CA et autorisez seulement les certificats de ce CA par le serveur, le numéro de série est le moyen le plus fiable (bien que difficile à retenir) pour identifier un utilisateur.

`ssl_client_dn() returns text`

Renvoie le sujet complet du certificat actuel du client, convertissant des données dans l'encodage actuel de la base de données. Nous supposons que si vous utilisez des caractères non ASCII dans le nom des certificats, votre base de données est capable de représenter ces caractères aussi. Si votre base de données utilise l'encodage SQL_ASCII, les caractères non ASCII seront représentés par des séquences UTF-8.

Le résultat ressemble à ceci : `/CN=Somebody /C=Some country/O=Some organization`.

`ssl_issuer_dn() returns text`

Renvoie le nom complet du créateur du certificat actuel du client, convertissant les données caractères dans l'encodage actuel de la base de données. Les conversions d'encodage sont gérées de la même façon que pour `ssl_client_dn`.

La combinaison de la valeur en retour de cette fonction avec le numéro de série du certificat identifie de façon unique le certificat.

Cette fonction est réellement utile si vous avez plus d'un certificat d'un CA de confiance dans le fichier d'autorité de certificat de votre serveur, ou si ce CA a envoyé quelques certificats intermédiaires d'autorité.

`ssl_client_dn_field(fieldname text) returns text`

Cette fonction renvoie la valeur du champ spécifié dans le sujet du certificat, ou NULL si le champ n'est pas présent. Les noms du champ sont des constantes de chaîne qui sont converties dans des identifiants d'objet ASN1 en utilisant la base de données des objets OpenSSL. Les valeurs suivantes sont acceptables :

```
commonName (alias CN)
surname (alias SN)
name
givenName (alias GN)
countryName (alias C)
localityName (alias L)
stateOrProvinceName (alias ST)
organizationName (alias O)
organizationalUnitName (alias OU)
title
description
initials
postalCode
streetAddress
generationQualifier
description
dnQualifier
x500UniqueIdentifier
pseudonym
role
emailAddress
```

Tous ces champs sont optionnels, sauf `commonName`. L'inclusion des champs dépend entièrement de la politique de votre CA. Par contre, la signification des champs est strictement définie par les standards X.500 et X.509, donc vous ne pouvez pas leur donner des significations arbitraires.

`ssl_issuer_field(fieldname text) returns text`

Identique à `ssl_client_dn_field`, mais pour le créateur du certificat, plutôt que pour le sujet du certificat.

`ssl_extension_info() returns setof record`

Fournit des informations sur les extensions des certificats clients : nom de l'extension, valeur de l'extension, et s'il s'agit d'une extension critique.

F.42.2. Auteur

Victor Wagner <vitus@cryptocom.ru>, Cryptocom LTD

Dmitry Voronin <carriingfate92@yandex.ru>

E-Mail du groupe de développement Cryptocom OpenSSL : <openssl@cryptocom.ru>

F.43. tablefunc

Le module `tablefunc` inclut plusieurs fonctions permettant de renvoyer des tables (c'est-à-dire plusieurs lignes). Ces fonctions sont utiles directement et comme exemples sur la façon d'écrire des fonctions C qui renvoient plusieurs lignes.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

F.43.1. Fonctions

Tableau F.30 résume les fonctions fournies par le module `tablefunc`.

Tableau F.30. Fonctions `tablefunc`

Fonction Description
`normal_rand` (*`numvals`* `integer`, *`mean`* `float8`, *`stddev`* `float8`) → `setof float8` Produit un ensemble de valeurs aléatoire avec une distribution normale.
`crosstab` (*`sql`* `text`) → `setof record` Produit une « table pivot » contenant les noms des lignes ainsi que *`N`* colonnes de valeur, où *`N`* est déterminé par le type de ligne spécifié dans la requête appelante.
`crosstab`*`N`* (*`sql`* `text`) → `setof table_crosstab_`*`N`* Produit une « table pivot » contenant les noms des lignes ainsi que *`N`* colonnes de valeur. `crosstab2`, `crosstab3` et `crosstab4` sont prédéfinies mais vous pouvez créer les fonctions `crosstab`*`N`* supplémentaires comme décrit ci-dessous.
`crosstab` (*`source_sql`* `text`, *`category_sql`* `text`) → `setof record` Produit une « table pivot » avec les colonnes de valeur indiquées par une deuxième requête.
`crosstab` (*`sql`* `text`, *`N`* `integer`) → `setof record` Version obsolète de `crosstab(text)`. Le paramètre *`N`* est maintenant ignoré car le nombre de colonnes de valeur est toujours déterminé par la requête appelante.
`connectby` (*`relname`* `text`, *`keyid_fld`* `text`, *`parent_keyid_fld`* `text` [, *`orderby_fld`* `text`], *`start_with`* `text`, *`max_depth`* `integer` [, *`branch_delim`* `text`]) → `setof record` Produit une représentation d'une structure d'arbre hiérarchique.

F.43.1.1. `normal_rand`

```
normal_rand(int numvals, float8 mean, float8 stddev) returns setof
 float8
```

`normal_rand` produit un ensemble de valeurs distribuées au hasard (distribution gaussienne).

`numvals` est le nombre de valeurs que la fonction doit renvoyer. *`mean`* est la moyenne de la distribution normale des valeurs et *`stddev`* est la déviation standard de la distribution normale des valeurs.

Par exemple, cet appel demande 1000 valeurs avec une moyenne de 5 et une déviation standard de 3 :

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
     normal_rand
----------------------
     1.56556322244898
     9.10040991424657
     5.36957140345079
    -0.369151492880995
     0.283600703686639
        .
        .
        .
     4.82992125404908
     9.71308014517282
     2.49639286969028
(1000 rows)
```

F.43.1.2. `crosstab(text)`

```
crosstab(text sql)
crosstab(text sql, int N)
```

La fonction `crosstab` est utilisé pour créer un affichage « pivot » où les données sont listées de gauche à droite plutôt que de haut en bas. Par exemple, avec ces données

```
row1    val11
row1    val12
row1    val13
...
row2    val21
row2    val22
row2    val23
...
```

l'affiche ressemble à ceci

```
row1    val11   val12   val13   ...
row2    val21   val22   val23   ...
...
```

La fonction `crosstab` prend un paramètre texte qui est une requête SQL produisant des données brutes formatées de la façon habituelle et produit une table avec un autre formatage.

Le paramètre *`sql`* est une instruction SQL qui produit l'ensemble source des données. Cette instruction doit renvoyer une colonne `row_name`, une colonne `category` et une colonne `value`. *`N`* est un paramètre obsolète, ignoré quand il est fourni (auparavant, il devait correspondre au nombre de colonnes de valeurs en sortie, mais maintenant ceci est déterminé par la requête appelant).

Par exemple, la requête fournie peut produire un ensemble ressemblant à ceci :

```
 row_name    cat    value
----------+-------+-------
  row1      cat1    val1
  row1      cat2    val2
  row1      cat3    val3
  row1      cat4    val4
  row2      cat1    val5
  row2      cat2    val6
  row2      cat3    val7
  row2      cat4    val8
```

La fonction `crosstab` déclare renvoyer un `setof record`, donc les noms et types réels des colonnes doivent être définis dans la clause `FROM` de l'instruction `SELECT` appelante. Par exemple : statement, for example:

```
    SELECT * FROM crosstab('...') AS ct(row_name text, category_1
 text, category_2 text);
```

Cet exemple produit un ensemble ressemblant à ceci :

```
          <== value  columns  ==>
row_name   category_1   category_2
 ---------+------------+------------
   row1        val1         val2
   row2        val5         val6
```

La clause `FROM` doit définir la sortie comme une colonne `row_name` (du même type que la première colonne du résultat de la requête SQL) suivie par N colonnes `value` (tous du même type de données que la troisième colonne du résultat de la requête SQL). Vous pouvez configurer autant de colonnes de valeurs en sortie que vous voulez. Les noms des colonnes en sortie n'ont pas d'importance en soi.

La fonction `crosstab` produit une ligne en sortie pour chaque groupe consécutif de lignes en entrée avec la même valeur `row_name`. Elle remplit les colonnes de `value`, de gauche à droite, avec les champs `value` provenant de ces lignes. S'il y a moins de lignes dans un groupe que de colonnes `value` en sortie, les colonnes supplémentaires sont remplies avec des valeurs NULL ; s'il y a trop de ligne, les colonnes en entrée supplémentaires sont ignorées.

En pratique, la requête SQL devrait toujours spécifier `ORDER BY 1,2` pour s'assurer que les lignes en entrée sont bien ordonnées, autrement dit que les valeurs de même `row_name` sont placées ensemble et sont correctement ordonnées dans la ligne. Notez que `crosstab` ne fait pas attention à la deuxième colonne du résultat de la requête ; elle est là pour permettre le tri, pour contrôler l'ordre dans lequel les valeurs de la troisième colonne apparaissent dans la page.

Voici un exemple complet :

```
CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value)
 VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value)
 VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value)
 VALUES('test1','att3','val3');
```

```
INSERT INTO ct(rowid, attribute, value)
 VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value)
 VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value)
 VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value)
 VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value)
 VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3
 text);

 row_name | category_1 | category_2 | category_3
----------+------------+------------+------------
 test1    | val2       | val3       |
 test2    | val6       | val7       |
(2 rows)
```

Vous pouvez toujours éviter d'avoir à écrire une clause `FROM` pour définir les colonnes en sortie, en définissant une fonction crosstab personnalisée qui a le type de ligne désiré en sortie en dur dans sa définition. Ceci est décrit dans la prochaine section. Une autre possibilité est d'embarquer la clause `FROM` requise dans la définition d'une vue.

> **Note**
>
> Voir aussi la commande `\crosstabview` dans psql. Elle fournit des fonctionnalités similaires à `crosstab()`.

F.43.1.3. `crosstabN(text)`

```
crosstabN(text sql)
```

Les fonctions `crosstabN` sont des exemples de configuration de fonctions d'emballage pour la fonction généraliste `crosstab`. Cela vous permet de ne pas avoir à écrire les noms et types des colonnes dans la requête `SELECT` appelante. Le module `tablefunc` inclut `crosstab2`, `crosstab3` et `crosstab4`, dont les types de ligne en sortie sont définis ainsi :

```
CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
        .
        .
        .
    category_N TEXT
```

```
);
```

Du coup, ces fonctions peuvent être utilisées directement quand la requête en entrée produit des colonnes `row_name` et `value` de type `text`, et que vous voulez 2, 3 ou 4 colonnes de valeur en sortie. Autrement, elles se comportent exactement la fonction `crosstab` décrite précédemment.

L'exemple de la section précédente pourrait aussi fonctionner ainsi :

```
SELECT *
FROM crosstab3(
  'select rowid, attribute, value
   from ct
   where attribute = ''att2'' or attribute = ''att3''
   order by 1,2');
```

Ces fonctions sont fournies principalement comme exemples. Vous pouvez créer vos propres types de retour et fonctions basées sur la fonction `crosstab()`. Il existe deux façons de le faire :

- Créer un type composite décrivant les colonnes désirées en sortie, similaire aux exemples disponibles dans le fichier `contrib/tablefunc/tablefunc--1.0.sql`. Ensuite, définir un nom de fonction unique acceptant un paramètre de type `text` et renvoyant `setof nom_de_votre_type`, mais renvoyant à la fonction C `crosstab`. Par exemple, si votre source de données produit des noms de ligne qui sont de type `text`, et des valeurs qui sont de type `float8`, et que vous voulez cinq colonnes de valeurs :

```
      CREATE TYPE my_crosstab_float8_5_cols AS (
          my_row_name text,
          my_category_1 float8,
          my_category_2 float8,
          my_category_3 float8,
          my_category_4 float8,
          my_category_5 float8
      );

      CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(text)
        RETURNS setof my_crosstab_float8_5_cols
        AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE
 STRICT;
```

- Utiliser des paramètres `OUT` pour définir implicitement le type en retour. Le même exemple pourrait s'écrire ainsi :

```
      CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(
          IN text,
          OUT my_row_name text,
          OUT my_category_1 float8,
          OUT my_category_2 float8,
          OUT my_category_3 float8,
          OUT my_category_4 float8,
          OUT my_category_5 float8)
        RETURNS setof record
        AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE
 STRICT;
```

F.43.1.4. crosstab(text, text)

```
crosstab(text source_sql, text category_sql)
```

La limite principale de la forme à un paramètre de `crosstab` est qu'elle traite toutes les valeurs d'un groupe de la même façon, en insérant chaque valeur dans la première colonne disponible. Si vous voulez les colonnes de valeur correspondant à des catégories spécifiques de données, et que certains groupes n'ont pas de données pour certaines des catégories, alors cela ne fonctionne pas. La forme à deux paramètres de la fonction `crosstab` gère ce cas en fournissant une liste explicite des catégories correspondant aux colonnes en sortie.

`source_sql` est une instruction SQL qui produit l'ensemble source des données. Cette instruction doit renvoyer une colonne `row_name`, une colonne `category` et une colonne `value`. Elle pourrait aussi avoir une ou plusieurs colonnes « extra ». La colonne `row_name` doit être la première. Les colonnes `category` et `value` doivent être les deux dernières colonnes, dans cet ordre. Toutes les colonnes entre `row_name` et `category` sont traitées en « extra ». Les colonnes « extra » doivent être les mêmes pour toutes les lignes avec la même valeur `row_name`.

Par exemple, *`source_sql`* produit un ensemble ressemblant à ceci :

```
SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;

 row_name    extra_col   cat    value
----------+------------+-----+---------
  row1         extra1    cat1    val1
  row1         extra1    cat2    val2
  row1         extra1    cat4    val4
  row2         extra2    cat1    val5
  row2         extra2    cat2    val6
  row2         extra2    cat3    val7
  row2         extra2    cat4    val8
```

`category_sql` est une instruction SQL qui produit l'ensemble des catégories. Cette instruction doit renvoyer seulement une colonne. Cela doit produire au moins une ligne, sinon une erreur sera générée. De plus, cela ne doit pas produire de valeurs dupliquées, sinon une erreur sera aussi générée. *`category_sql`* doit ressembler à ceci :

```
SELECT DISTINCT cat FROM foo ORDER BY 1;
    cat
  -------
    cat1
    cat2
    cat3
    cat4
```

La fonction `crosstab` déclare renvoyer `setof record`, donc les noms et types réels des colonnes en sortie doivent être définis dans la clause `FROM` de la requête `SELECT` appelante, par exemple :

```
    SELECT * FROM crosstab('...', '...')
      AS ct(row_name text, extra text, cat1 text, cat2 text, cat3
 text, cat4 text);
```

Ceci produira un résultat ressemblant à ceci :

```
                    <==  value  columns   ==>
row_name   extra   cat1   cat2   cat3   cat4
---------+-------+------+------+------+------
  row1     extra1  val1   val2          val4
  row2     extra2  val5   val6   val7   val8
```

La clause `FROM` doit définir le bon nombre de colonnes en sortie avec les bons types de données. S'il y a *`N`* colonnes dans le résultat de la requête *`source_sql`*, les *`N`*-2 premiers d'entre eux doivent correspondre aux *`N`*-2 premières colonnes en sortie. Les colonnes restantes en sortie doivent avoir le type de la dernière colonne du résultat de la requête The remaining output columns *`source_sql`*, et il doit y en avoir autant que de lignes dans le résultat de la requête *`category_sql`*.

La fonction `crosstab` produit une ligne en sortie pour chaque groupe consécutif de lignes en entrée avec la même valeur `row_name`. La colonne en sortie `row_name` ainsi que toutes colonnes « extra » sont copiées à partir de la première ligne du groupe. Les colonnes `value` en sortie sont remplies avec les champs `value` à partir des lignes ayant une correspondance avec des valeurs `category`. Si la `category` d'une ligne ne correspond pas à une sortie de la requête *`category_sql`*, sa `value` est ignorée. Les colonnes en sortie dont la catégorie correspondante est absente de toute ligne en entrée du groupe sont remplies avec des valeurs NULL.

En pratique, la requête *`source_sql`* doit toujours spécifier `ORDER BY 1` pour s'assurer que les valeurs du même `row_name` sont assemblées. Néanmoins, l'ordre des catégories dans un groupe n'est pas important. De plus, il est essentiel que l'ordre du résultat de la requête *`category_sql`* corresponde à l'ordre des colonnes spécifiées en sortie.

Voici deux exemples complets :

```
create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);

select * from crosstab(
  'select year, month, qty from sales order by 1',
  'select m from generate_series(1,12) m'
) as (
  year int,
  "Jan" int,
  "Feb" int,
  "Mar" int,
  "Apr" int,
  "May" int,
  "Jun" int,
  "Jul" int,
  "Aug" int,
  "Sep" int,
  "Oct" int,
  "Nov" int,
  "Dec" int
);
 year | Jan  | Feb  | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct
 | Nov  | Dec
```

```
------+------+------+-----+-----+-----+-----+-----+-----+-----
+-----+------+------
 2007 | 1000 | 1500 |     |     |     |     | 500 |     |     |
 | 1500 | 2000
 2008 | 1000 |      |     |     |     |     |     |     |     |
 |      |
(2 rows)



CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val
 text);
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March
 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March
 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01
 March 2003');
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');

SELECT * FROM crosstab
(
  'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
  'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)
AS
(
       rowid text,
       rowdt timestamp,
       temperature int4,
       test_result text,
       test_startdate timestamp,
       volts float8
);
 rowid |          rowdt           | temperature | test_result |
  test_startdate      | volts
-------+--------------------------+-------------+-------------
+--------------------------+--------
 test1 | Sat Mar 01 00:00:00 2003 |          42 | PASS        |
                      | 2.6987
 test2 | Sun Mar 02 00:00:00 2003 |          53 | FAIL        | Sat
 Mar 01 00:00:00 2003 | 3.1234
(2 rows)
```

Vous pouvez créer des fonctions prédéfinies pour éviter d'avoir à écrire les noms et types des colonnes en résultat dans chaque requête. Voir les exemples dans la section précédente. La fonction C sous-jacente pour cette forme de `crosstab` est appelée `crosstab_hash`.

F.43.1.5. connectby

```
connectby(text relname, text keyid_fld, text parent_keyid_fld
          [, text orderby_fld ], text start_with, int max_depth
          [, text branch_delim ])
```

La fonction `connectby` réalise un affichage de données hiérarchiques stockées dans une table. La table doit avoir un champ clé qui identifie de façon unique les lignes et un champ clé qui référence le parent de chaque ligne. `connectby` peut afficher le sous-arbre à partir de n'importe quelle ligne.

Tableau F.31 explique les paramètres.

Tableau F.31. Paramètres `connectby`

Paramètre	Description
`relname`	Nom de la relation source
`keyid_fld`	Nom du champ clé
`parent_keyid_fld`	Nom du champ clé du parent
`orderby_fld`	Nom du champ des autres relations (optionnel)
`start_with`	Valeur de la clé de la ligne de début
`max_depth`	Profondeur maximum pour la descente, ou zéro pour une profondeur illimitée
`branch_delim`	Chaîne pour séparer les clés des branches (optionnel)

Les champs clé et clé du parent peuvent être de tout type mais ils doivent être du même type. Notez que la valeur *`start_with`* doit être saisi comme une chaîne de caractères, quelque soit le type du champ clé.

La fonction `connectby` déclare renvoyer un `setof record`, donc les noms et types réels des colonnes en sortie doivent être définis dans la clause `FROM` de l'instruction `SELECT` appelante, par exemple :

```
   SELECT * FROM connectby('connectby_tree', 'keyid',
'parent_keyid', 'pos', 'row2', 0, '~')
     AS t(keyid text, parent_keyid text, level int, branch text,
pos int);
```

Des deux premières colonnes en sortie sont utilisées pour la clé de la ligne en cours et la clé de son parent ; elles doivent correspondre au type du champ clé de la table. La troisième colonne est la profondeur de l'arbre et doit être du type `integer`. Si un paramètre *`branch_delim`* est renseigné, la prochaine colonne en sortie est l'affichage de la branche et doit être de type `text`. Enfin, si le paramètre *`orderby_fld`* est renseigné, la dernière colonne en sortie est un numéro de série et doit être de type `integer`.

La colonne « branch » en sortie affiche le chemin des clés utilisé pour atteindre la ligne actuelle. Les clés sont séparées par la chaîne *`branch_delim`* spécifiée. Si l'affichage des branches n'est pas voulu, omettez le paramètre *`branch_delim`* et la colonne branche dans la liste des colonnes en sortie.

Si l'ordre des relations du même parent est important, incluez le paramètre *`orderby_fld`* pour indiquer par quel champ ordonner les relations. Ce champ doit être de tout type de données triable. La liste des colonnes en sortie doit inclure une colonne numéro de série de type integer si, et seulement si, *`orderby_fld`* est spécifiée.

Les paramètres représentant table et noms de champs sont copiés tels quel dans les requêtes SQL que `connectby` génère en interne. Du coup, ajoutez des guillemets doubles si les noms utilisent majuscules et minuscules ou s'ils contiennent des caractères spéciaux. Vous pouvez aussi avoir besoin de qualifier le nom de la table avec le nom du schéma.

Dans les grosses tables, les performances seront faibles sauf si un index est créé sur le champ clé parent.

Il est important que la chaîne *branch_delim* n'apparaisse pas dans les valeurs des clés, sinon connectby pourrait rapporter des erreurs de récursion infinie totalement erronées. Notez que si *branch_delim* n'est pas fourni, une valeur par défaut ~ est utilisé pour des raisons de détection de récursion.

Voici un exemple :

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos
 int);

INSERT INTO connectby_tree VALUES('row1',NULL, 0);
INSERT INTO connectby_tree VALUES('row2','row1', 0);
INSERT INTO connectby_tree VALUES('row3','row1', 0);
INSERT INTO connectby_tree VALUES('row4','row2', 1);
INSERT INTO connectby_tree VALUES('row5','row2', 0);
INSERT INTO connectby_tree VALUES('row6','row4', 0);
INSERT INTO connectby_tree VALUES('row7','row3', 0);
INSERT INTO connectby_tree VALUES('row8','row6', 0);
INSERT INTO connectby_tree VALUES('row9','row5', 0);

-- with branch, without orderby_fld (order of results is not
 guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid',
 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text);
 keyid | parent_keyid | level |       branch
-------+--------------+-------+--------------------
 row2  |              |     0 | row2
 row4  | row2         |     1 | row2~row4
 row6  | row4         |     2 | row2~row4~row6
 row8  | row6         |     3 | row2~row4~row6~row8
 row5  | row2         |     1 | row2~row5
 row9  | row5         |     2 | row2~row5~row9
(6 rows)

-- without branch, without orderby_fld (order of results is not
 guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid',
 'row2', 0)
 AS t(keyid text, parent_keyid text, level int);
 keyid | parent_keyid | level
-------+--------------+-------
 row2  |              |     0
 row4  | row2         |     1
 row6  | row4         |     2
 row8  | row6         |     3
 row5  | row2         |     1
 row9  | row5         |     2
(6 rows)

-- with branch, with orderby_fld (notice that row5 comes before
 row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid',
 'pos', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text, pos
 int);
```

```
 keyid | parent_keyid | level |       branch        | pos
-------+--------------+-------+---------------------+-----
 row2  |              |     0 | row2                |   1
 row5  | row2         |     1 | row2~row5           |   2
 row9  | row5         |     2 | row2~row5~row9      |   3
 row4  | row2         |     1 | row2~row4           |   4
 row6  | row4         |     2 | row2~row4~row6      |   5
 row8  | row6         |     3 | row2~row4~row6~row8 |   6
(6 rows)

-- without branch, with orderby_fld (notice that row5 comes before
 row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid',
 'pos', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int, pos int);
 keyid | parent_keyid | level | pos
-------+--------------+-------+-----
 row2  |              |     0 |   1
 row5  | row2         |     1 |   2
 row9  | row5         |     2 |   3
 row4  | row2         |     1 |   4
 row6  | row4         |     2 |   5
 row8  | row6         |     3 |   6
(6 rows)
```

F.43.2. Auteur

Joe Conway

F.44. tcn

Le module `tcn` fournit une fonction trigger qui notifie les processus à l'écoute de changement sur toutes les tables sur lesquelles elle est attachée. Elle doit être utilisée dans un trigger `AFTER` et `FOR EACH ROW`.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut `SUPERUSER`) et qui ont l'attribut `CREATE` sur la base de données courante.

Un seul paramètre peut être fourni à la fonction dans une instruction `CREATE TRIGGER` et il est optionnel. S'il est fourni, il sera utilisé comme nom de canal pour les notifications. S'il est omis, `tcn` sera utilisé comme nom de canal.

Le contenu des notifications consiste en le nom de la table, une lettre pour indiquer le type d'opération réalisée, et des paires nom de colonne/valeur pour les colonnes de la clé primaire. Chaque partie est séparée de la suivante par une virgule. Pour faciliter l'analyse en utilisant des expressions régulières, les noms de tables et de colonnes sont toujours entourés de guillemets doubles et les valeurs sont toujours entourées de guillemets simples. Les guillemets à l'intérieur sont doublés.

Voici un bref exemple d'utilisation de cette extension :

```
test=# create table tcndata
test-#   (
test(#     a int not null,
test(#     b date not null,
test(#     c text,
```

```
test(#     primary key (a, b)
test(#   );
CREATE TABLE
test=# create trigger tcndata_tcn_trigger
test-#   after insert or update or delete on tcndata
test-#   for each row execute function
 triggered_change_notification();
CREATE TRIGGER
test=# listen tcn;
LISTEN
test=# insert into tcndata values (1, date '2012-12-22', 'one'),
test-#                            (1, date '2012-12-23',
 'another'),
test-#                            (2, date '2012-12-23', 'two');
INSERT 0 3
Asynchronous notification "tcn" with payload
 ""tcndata",I,"a"='1',"b"='2012-12-22'" received from server
 process with PID 22770.
Asynchronous notification "tcn" with payload
 ""tcndata",I,"a"='1',"b"='2012-12-23'" received from server
 process with PID 22770.
Asynchronous notification "tcn" with payload
 ""tcndata",I,"a"='2',"b"='2012-12-23'" received from server
 process with PID 22770.
test=# update tcndata set c = 'uno' where a = 1;
UPDATE 2
Asynchronous notification "tcn" with payload
 ""tcndata",U,"a"='1',"b"='2012-12-22'" received from server
 process with PID 22770.
Asynchronous notification "tcn" with payload
 ""tcndata",U,"a"='1',"b"='2012-12-23'" received from server
 process with PID 22770.
test=# delete from tcndata where a = 1 and b = date '2012-12-22';
DELETE 1
Asynchronous notification "tcn" with payload
 ""tcndata",D,"a"='1',"b"='2012-12-22'" received from server
 process with PID 22770.
```

F.45. test_decoding

`test_decoding` est un exemple de plugin de sortie pour le décodage logique. Il ne fait rien de particulièrement utile, mais peut servir de point de départ pour créer votre propre plugin de sortie.

`test_decoding` reçoit les journaux de transactions à travers les mécanismes de décodage logique, et les décode sous forme de représentation au format texte des opérations effectuées.

La sortie typique de ce plugin, utilisé sur l'interface de décodage logique SQL, serait :

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('test_slot',
 NULL, NULL, 'include-xids', '0');
   lsn     | xid |                       data
-----------+-----
+-------------------------------------------------
 0/16D30F8 | 691 | BEGIN
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:2
 data[text]:'arg'
```

```
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:3
 data[text]:'demo'
 0/16D32A0 | 691 | COMMIT
 0/16D32D8 | 692 | BEGIN
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:2
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:3
 0/16D3398 | 692 | COMMIT
(8 rows)
```

Nous pouvons aussi obtenir les changements de transactions en cours. La sortie ressemblerait typiquement à ceci :

```
postgres[33712]=#* SELECT * FROM
 pg_logical_slot_get_changes('test_slot', NULL, NULL, 'stream-
changes', '1');
    lsn    | xid |                       data
-----------+-----
+-----------------------------------------------
 0/16B21F8 | 503 | opening a streamed block for transaction TXN 503
 0/16B21F8 | 503 | streaming change for TXN 503
 0/16B2300 | 503 | streaming change for TXN 503
 0/16B2408 | 503 | streaming change for TXN 503
 0/16BEBA0 | 503 | closing a streamed block for transaction TXN 503
 0/16B21F8 | 503 | opening a streamed block for transaction TXN 503
 0/16BECA8 | 503 | streaming change for TXN 503
 0/16BEDB0 | 503 | streaming change for TXN 503
 0/16BEEB8 | 503 | streaming change for TXN 503
 0/16BEBA0 | 503 | closing a streamed block for transaction TXN 503
(10 rows)
```

F.46. tsm_system_rows

Le module `tsm_system_rows` fournit la méthode d'échantillonage de table `SYSTEM_ROWS`, qui peut être utilisé dans la clause `TABLESAMPLE` d'une commande `SELECT`.

Cette méthode d'échantillonnage accepte un argument de type entier correspondant au nombre maximum de lignes à lire. L'échantillon résultant contiendra toujours ce nombre exact de lignes, sauf si la table ne contient pas suffisamment de lignes, auquel cas la table entière est sélectionnée.

Comme la méthode interne `SYSTEM`, `SYSTEM_ROWS` réalise un échantillonnage au niveau des blocs de table, si bien que l'échantillonnage n'est pas complètement aléatoire mais peut être sensible à un effet de regroupement (*clustering*), surtout si un petit nombre de lignes est demandé.

`SYSTEM_ROWS` ne supporte pas la clause `REPEATABLE`.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut `SUPERUSER`) et qui ont l'attribut `CREATE` sur la base de données courante.

F.46.1. Exemples

Voici un exemple de sélection d'un échantillon d'une table avec `SYSTEM_ROWS`. Il faut tout d'abord installer l'extension :

```
CREATE EXTENSION tsm_system_rows;
```

Puis vous pouvez l'utiliser dans une commande `SELECT`, par exemple :

```
SELECT * FROM ma_table TABLESAMPLE SYSTEM_ROWS(100);
```

Cette commande renverra un échantillon de 100 lignes depuis la table `ma_table` (sauf si la table contient moins de 100 lignes visibles, auquel cas toutes les lignes sont renvoyées).

F.47. tsm_system_time

Le module `tsm_system_time` fournit la méthode d'échantillonage de table `SYSTEM_TIME`, qui peut être utilisé par la clause `TABLESAMPLE` d'une commande `SELECT`.

Cette méthode d'échantillonnage d'une table accepte un unique argument, de type nombre à virgule flottante, correspondant au nombre maximum de millisecondes passé à lire la table. Ceci vous donne un contrôle direct sur la durée de la requête, au prix d'une taille d'échantillon difficile à prédire. Ce résultat contiendra autant de lignes qu'il a été possible d'en lire pendant la durée spécifiée, sauf si la table a pu être lue entièrement avant.

De la même façon que la méthode interne `SYSTEM`, `SYSTEM_TIME` réalise un échantillonnage au niveau des blocs de table, si bien que l'échantillonnage n'est pas complètement aléatoire mais peut être sensible à un effet de regroupement (*clustering*), surtout si un petit nombre de lignes est sélectionné.

`SYSTEM_TIME` ne supporte pas la clause `REPEATABLE`.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut `SUPERUSER`) et qui ont l'attribut `CREATE` sur la base de données courante.

F.47.1. Exemples

Voici un exemple de sélection d'un échantillon d'une table avec `SYSTEM_TIME`. Il faut tout d'abord installer l'extension :

```
CREATE EXTENSION tsm_system_time;
```

Puis vous pouvez l'utiliser dans une commande `SELECT`, par exemple :

```
SELECT * FROM ma_table TABLESAMPLE SYSTEM_TIME(1000);
```

Cette commande renverra autant de lignes de `ma_table` qu'il a pu en lire en une seconde (1000 millisecondes). Bien sûr, si toute la table peut être lue en moins d'une seconde, toutes les lignes seront renvoyées.

F.48. unaccent

`unaccent` est un dictionnaire de recherche plein texte qui supprime les accents d'un lexème. C'est un dictionnaire de filtre, ce qui signifie que sa sortie est passée au prochain dictionnaire (s'il y en a un), contrairement au comportement normal des dictionnaires. Cela permet le traitement des accents pour la recherche plein texte.

L'implémentation actuelle d'`unaccent` ne peut pas être utilisée comme un dictionnaire de normalisation pour un dictionnaire `thesaurus`.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut `SUPERUSER`) et qui ont l'attribut `CREATE` sur la base de données courante.

F.48.1. Configuration

Le dictionnaire `unaccent` accepte les options suivantes :

- `RULES` est le nom de base du fichier contenant la liste des règles de traduction. Ce fichier doit être stocké dans le répertoire `$SHAREDIR/tsearch_data/` (`$SHAREDIR` étant le répertoire des

données partagées de PostgreSQL). Son nom doit se terminer avec l'extension `.rules` (qui ne doit pas être inclus dans le paramètre `RULES`).

Le fichier des règles a le format suivant :

- Chaque ligne représente une règle de traduction, consistant en un caractère avec accent, suivi d'un caractère sans accent. Le premier est traduit avec le second. Par exemple :

  ```
  À       A
  Á       A
  Â        A
  Ã       A
  Ä          A
  Å        A
  Æ        AE
  ```

  Les deux caractères doivent être séparés par des espaces blancs, et tout espace blanc au début et à la fin d'une ligne est ignoré.

- Sinon, si seulement un caractère est donné sur une ligne, les occurrences de ce caractère sont supprimées. Ceci est utile dans les langues où les accents sont représentés par des caractères séparés.

- Actuellement, chaque caractère peut être une chaîne ne contenant pas d'espace blanc, pour que les dictionnaires `unaccent` puissent être utilisés pour d'autres types de substitutions de sous- chaînes au delà des suppressions de signes diacritiques.

- Comme avec d'autres fichiers de configuration de la recherche plein texte avec PostgreSQL, le fichier de règles doit être stocké dans l'encodage UTF-8. Les données sont automatiquement traduites dans l'encodage courant de la base de données lors du chargement. Toute ligne contenant des caractères non traduisibles est ignorée silencieusement, de façon à ce que les fichiers de règles puissent contenir des règles qui ne sont pas applicables dans l'encodage courant.

Un exemple plus complet, qui est directement utile pour les langages européens, se trouve dans `unaccent.rules`, qui est installé dans le répertoire `$SHAREDIR/tsearch_data/` une fois le module `unaccent` installé. Ce fichier de règles traduit les caractères avec accent vers le même caractère sans accent. Il étend aussi les ligatures en une série équivalente de caractères simples (par exemple, Æ devient AE).

F.48.2. Utilisation

Installer l'extension `unaccent` crée un modèle de recherche de texte appelé `unaccent` et un dictionnaire basé sur ce modèle, appelé lui-aussi `unaccent`. Le dictionnaire `unaccent` a le paramètre par défaut `RULES='unaccent'`, qui le rend directement utilisable avec le fichier standard `unaccent.rules`. Si vous le souhaitez, vous pouvez modifier le paramètre. Par exemple :

```
ma_base=# ALTER TEXT SEARCH DICTIONARY unaccent
 (RULES='mes_regles');
```

Vous pouvez aussi créer de nouveaux dictionnaires basés sur le modèle.

Pour tester le dictionnaire, vous pouvez essayer la requête suivante :

```
ma_base=# select ts_lexize('unaccent','Hôtel');
 ts_lexize
-----------
 {Hotel}
```

```
(1 row)
```

Voici un exemple montrant comment installer le dictionnaire `unaccent` dans une configuration de recherche plein texte :

```
ma_base=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
ma_base=# ALTER TEXT SEARCH CONFIGURATION fr
        ALTER MAPPING FOR hword, hword_part, word
        WITH unaccent, french_stem;
ma_base=# select to_tsvector('fr','Hôtels de la Mer');
    to_tsvector
-------------------
 'hotel':1 'mer':4
(1 row)

ma_base=# select to_tsvector('fr','Hôtel de la Mer') @@
 to_tsquery('fr','Hotels');
 ?column?
----------
 t
(1 row)
ma_base=# select ts_headline('fr','Hôtel de la
 Mer',to_tsquery('fr','Hotels'));
      ts_headline
------------------------
  <b>Hôtel</b>de la Mer
(1 row)
```

F.48.3. Fonctions

La fonction `unaccent()` supprime les accents d'une chaîne de caractères donnée. Elle utilise un dictionnaire de type `unaccent` mais elle peut être utilisé en dehors du contexte normal de la recherche plein texte.

```
unaccent([dictionary regdictionary, ] string text) returns text
```

Si l'argument *`dictionary`* est omis, le dictionnaire de recherche plein texte nommé `unaccent` et apparaissant dans le même schéma que la fonction `unaccent()` elle-même est utilisé

```
SELECT unaccent('unaccent','Hôtel');
SELECT unaccent('Hôtel');
```

F.49. uuid-ossp

Le module `uuid-ossp` fournit des fonctions qui permettent de créer des identifiants uniques universels (UUIDs) à l'aide d'algorithmes standard. Ce module fournit aussi des fonctions pour produire certaines constantes UUID spéciales. Ce module est uniquement nécessaire pour des fonctionnalités supplémentaires qui vont au delà de ce que fournit déjà le moteur PostgreSQL. Voir Section 9.14 pour les méthodes internes de génération d'UUID.

Ce module est considéré comme « trusted », ce qui signifie qu'il peut être installé par des utilisateurs simples (sans attribut SUPERUSER) et qui ont l'attribut CREATE sur la base de données courante.

F.49.1. Fonctions de uuid-ossp

Tableau F.32 montre les fonctions disponibles pour générer des UUIDs. Les standards en question, ITU-T Rec. X.667, ISO/IEC 9834-8:2005, et RFC 4122[13], spécifient quatre algorithmes pour produire des UUID identifiés par les numéros de version 1, 3, 4 et 5. (Il n'existe pas d'algorithme version 2.) Chacun de ces algorithmes peut convenir pour un ensemble différent d'applications.

Tableau F.32. Fonctions pour la génération d'UUID

Fonction Description
`uuid_generate_v1` () → `uuid` Cette fonction crée un UUID version 1. Ceci implique l'adresse MAC de l'ordinateur et un horodatage. Les UUID de ce type révèlent l'identité de l'ordinateur qui a créé l'identifiant et l'heure de création de cet identifiant, ce qui peut ne pas convenir pour certaines applications sensibles à la sécurité.
`uuid_generate_v1mc` () → `uuid` Cette fonction crée un UUID version 1, mais utilise une adresse MAC multicast à la place de la vraie adresse de l'ordinateur.
`uuid_generate_v3` (*`namespace`* `uuid`, *`name`* `text`) → `uuid` Cette fonction crée un UUID version 3 dans l'espace de nom donné en utilisant le nom indiqué en entrée. L'espace de nom doit être une des constantes spéciales produites par les fonctions `uuid_ns_*()` indiquées dans Tableau F.33. (En théorie, cela peut être tout UUID.) Le nom est un identifiant dans l'espace de nom sélectionné. Par exemple : `SELECT uuid_generate_v3(uuid_ns_url(), 'http://www.postgresql.org');` Le paramètre name sera haché avec MD5, donc la version claire ne peut pas être récupérée à partir de l'UUID généré. La génération des UUID par cette méthode ne comprend aucun élément au hasard ou dépendant de l'environnement et est du coup reproductible.
`uuid_generate_v4` () → `uuid` Cette fonction crée un UUID version 4 qui est entièrement réalisé à partir de nombres aléatoires.
`uuid_generate_v5` (*`namespace`* `uuid`, *`name`* `text`) → `uuid` Cette fonction crée un UUID version 5 qui fonctionne comme un UUID version 3 sauf que SHA-1 est utilisé comme méthode de hachage. La version 5 devrait être préférée à la version 3 car SHA-1 est considéré plus sécurisé que MD5.

Tableau F.33. Fonctions renvoyant des constantes UUID

Fonction Description
`uuid_nil` () → `uuid` Renvoie une constante UUID « nil », qui ne correspond pas à un UUID réel.

[13] https://datatracker.ietf.org/doc/html/rfc4122

Fonction Description
`uuid_ns_dns` () → `uuid` Renvoie une constante désignant l'espace de nom pour les UUID.
`uuid_ns_url` () → `uuid` Renvoie une constante désignant l'espace de nom URL pour les UUID.
`uuid_ns_oid` () → `uuid` Renvoie une constante désignant l'espace de nom des identifiants d'objets ISO pour les UUIDs. (Ceci aboutit aux OID ASN.1, mais n'a pas de relation avec les OID de PostgreSQL.)
`uuid_ns_x500` () → `uuid` Renvoie une constante désignant l'espace de nom « X.500 distinguished name » (DN) pour les UUID.

F.49.2. Construire `uuid-ossp`

Historiquement, ce module dépendant de la bibliothèque OSSP UUID, d'où provient le nom de ce module. Bien que la bibliothèque OSSP UUID soit toujours disponible sur http://www.ossp.org/pkg/lib/uuid/, elle n'est pas correctement maintenue, et devient de plus en plus difficile à porter vers de nouvelles plateformes. `uuid-ossp` peut maintenant être construit sans la bibliothèque OSSP sur certaines plateformes. Sur FreeBSD et certains dérivés BSD, les fonctions de création UUID sont incluses dans la bibliothèque `libc`. Sur Linux, macOS et quelques autres plateformes, les fonctions convenables sont fournies avec la bibliothèque `libuuid`, qui vient à l'origine du projet `e2fsprogs` (bien que sur les Linux modernes, il est considéré comme faisant partie de `util-linux-ng`). Lors de l'appel à `configure`, spécifiez `--with-uuid=bsd` pour utiliser les fonctions BSD ou `--with-uuid=e2fs` pour utiliser la `libuuid` de `e2fsprogs` ou encore `--with-uuid=ossp` pour utiliser la bibliothèque OSSP UUID. Il se peut que plusieurs versions de cette bibliothèque soient disponibles sur une même machine, de ce fait, `configure` n'en choisit pas une de façon automatique.

F.49.3. Auteur

Peter Eisentraut `<peter_e@gmx.net>`

F.50. xml2

Le module `xml2` fournit des fonctionnalités pour les requêtes XPath et pour XSLT.

F.50.1. Notice d'obsolescence

À partir de PostgreSQL 8.3, les fonctionnalités XML basées sur le standard SQL/XML sont dans le cœur du serveur. Cela couvre la vérification de la syntaxe XML et les requêtes XPath, ce que fait aussi ce module (en dehors d'autres choses) mais l'API n'est pas du tout compatible. Il est prévu que ce module soit supprimé dans une future version de PostgreSQL pour faire place à une nouvelle API standard, donc vous êtes encouragés à convertir vos applications. Si vous trouvez que des fonctionnalités de ce module ne sont pas disponibles dans un format adéquat avec la nouvelle API, merci d'expliquer votre problème sur la liste `<pgsql-hackers@lists.postgresql.org>` pour que ce problème soit corrigé.

F.50.2. Description des fonctions

Tableau F.34 montre les fonctions fournies par ce module. Ces fonctions fournissent une analyse XML et les requêtes XPath.

Tableau F.34. Fonctions `xml2`

Fonction Description
`xml_valid` (*`document`* `text`) → `boolean` Analyse le document donné et renvoie la valeur true si le document est du XML bien formé. (Note : ceci est un alias pour la fonction PostgreSQL native `xml_is_well_formed()`. Le nom `xml_valid()` est techniquement incorrect car la validité et le fait d'être bien formé sont deux notions différentes en XML.)
`xpath_string` (*`document`* `text`, *`query`* `text`) → `text` Évalue le chemin XPath sur le document fourni, et convertit le résultat en type `text`.
`xpath_number` (*`document`* `text`, *`query`* `text`) → `real` Évalue le chemin XPath sur le document fourni, et convertit le résultat en type `real`.
`xpath_bool` (*`document`* `text`, *`query`* `text`) → `boolean` Évalue le chemin XPath sur le document fourni, et convertit le résultat en type `boolean`.
`xpath_nodeset` (*`document`* `text`, *`query`* `text`, *`toptag`* `text`, *`itemtag`* `text`) → `text` Évalue la requête sur le document et emballe le résultat dans des balises XML. Si le résultat est à plusieurs valeurs, la sortie ressemblera à ceci : `<toptag>` `<itemtag>Value 1 which could be an XML fragment</itemtag>` `<itemtag>Value 2....</itemtag>` `</toptag>` Si *`toptag`* ou *`itemtag`* est une chaîne vide, la balise en question est omise.
`xpath_nodeset` (*`document`* `text`, *`query`* `text`, *`itemtag`* `text`) → `text` Identique à `xpath_nodeset(document, query, toptag, itemtag)` mais le résultat omet *`toptag`*.
`xpath_nodeset` (*`document`* `text`, *`query`* `text`) → `text` Identique à `xpath_nodeset(document, query, toptag, itemtag)` mais le résultat omet les balises.
`xpath_list` (*`document`* `text`, *`query`* `text`, *`separator`* `text`) → `text` Évalue la requête dans le document et renvoie plusieurs valeurs séparées par le séparateur indiqué, par exemple `Value 1,Value 2,Value 3` si *`separator`* est une virgule (`,`).
`xpath_list` (*`document`* `text`, *`query`* `text`) → `text` Ceci est un emballage pour la fonction ci-dessus qui utilise la virgule (`,`) comme séparateur.

F.50.3. `xpath_table`

```
  xpath_table(text key, text document, text relation, text xpaths,
text criteria) returns setof record
```

`xpath_table` est une fonction SRF qui évalue un ensemble de requêtes XPath sur chaque ensemble de documents et renvoie les résultats comme une table. Le champ de clé primaire de la table des

documents est renvoyé comme première colonne des résultats pour que les résultats puissent être utilisés dans des jointures. Les paramètres sont décrits dans Tableau F.35.

Tableau F.35. Paramètres de `xpath_table`

Paramètre	Description
key	Le nom du champ de la clé primaire (« key »). C'est simplement le champ à utiliser comme première colonne de la table en sortie, autrement dit celle qui identifie l'enregistrement (voir la note ci-dessous sur les valeurs multiples).
document	Le nom du champ contenant le document XML.
relation	Le nom de la table ou de la vue contenant les documents.
xpaths	Une ou plusieurs expressions XPath séparées par des `|`
criteria	Le contenu de la clause WHERE. Elle doit être spécifiée, donc utilisez `true` ou `1=1` si vous voulez traiter toutes les lignes de la relation.

Ces paramètres (en dehors des chaînes XPath) sont simplement substitués dans une instruction SELECT, donc vous avez de la flexibilité. L'instruction est celle qui suit :

```
SELECT <key>, <document> FROM <relation> WHERE <criteria>
```

Donc les paramètres peuvent être *tout* ce qui est valide dans ces emplacements particuliers. Le résultat de ce SELECT a besoin de renvoyer exactement deux colonnes (ce qu'il fera sauf si vous essayez d'indiquer plusieurs champs pour la clé ou le document). Cette approche simpliste implique que vous validiez avant tout valeur fournie par un utilisateur pour éviter les attaques par injection de code SQL.

La fonction doit être utilisée dans une expression `FROM` avec une clause `AS` pour indiquer les colonnes en sortie. Par exemple :

```
SELECT * FROM
xpath_table('article_id',
            'article_xml',
            'articles',
            '/article/author|/article/pages|/article/title',
            'date_entered > ''2003-01-01'' ')
AS t(article_id integer, author text, page_count integer, title
 text);
```

La clause `AS` définit les noms et types des colonnes de la table en sortie. La première est le champ « key » et le reste correspond à la requête XPath. S'il y a plus de requêtes XPath que de colonnes résultats, les requêtes supplémentaires seront ignorées, S'il y a plus de colonnes résultats que de requêtes XPath, les colonnes supplémentaires seront NULL.

Notez que cet exemple définit la colonne résultat `page_count` en tant qu'entier (integer). La fonction gère en interne les représentations textes, donc quand vous dites que vous voulez un entier en sortie, il prendra la représentation texte du résultat XPath et utilisera les fonctions en entrée de PostgreSQL pour la transformer en entier (ou tout type que la clause `AS` réclame). Vous obtiendrez une erreur s'il ne peut pas le faire -- par exemple si le résultat est vide -- donc rester sur du texte est préférable si vous pensez que vos données peuvent poser problème.

L'instruction SELECT n'a pas besoin d'être un `SELECT *`. Elle peut référencer les colonnes par nom ou les joindre à d'autres tables. La fonction produit une table virtuelle avec laquelle vous pouvez réaliser toutes les opérations que vous souhaitez (c'est-à-dire agrégation, jointure, tri, etc.) Donc nous pouvons aussi avoir :

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
                 '/article/title|/article/author/@id',
                 'xpath_string(article_xml,''/article/@date'') >
 ''2003-03-20'' ')
       AS t(article_id integer, title text, author_id integer),
     tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

comme exemple plus compliqué. Bien sûr, vous pouvez placer tout ceci dans une vue pour une utilisation plus simple.

F.50.3.1. Résultats à plusieurs valeurs

La fonction `xpath_table` suppose que les résultats de chaque requête XPath ramènent plusieurs valeurs, donc le nombre de lignes renvoyées par la fonction pourrait ne pas être le même que le nombre de documents en entrée. La première ligne renvoyée contient le premier résultat de chaque requête, la deuxième le second résultat de chaque requête. Si une des requêtes a moins de valeurs que les autres, des valeurs NULL seront renvoyées.

Dans certains cas, un utilisateur saura qu'une requête XPath renverra seulement un seul résultat, peut-être un identifiant unique de document) -- si elle est utilisée avec une requête XPath renvoyant plusieurs résultats, le résultat sur une ligne apparaîtra seulement sur la première ligne du résultat. La solution à cela est d'utiliser le champ clé pour une jointure avec une requête XPath. Comme exemple :

```
    CREATE TABLE test (
        id int PRIMARY KEY,
        xml text
    );

    INSERT INTO test VALUES (1, '<doc num="C1">
    <line num="L1"><a>1</a><b>2</b><c>3</c></line>
    <line num="L2"><a>11</a><b>22</b><c>33</c></line>
    </doc>');

    INSERT INTO test VALUES (2, '<doc num="C2">
    <line num="L1"><a>111</a><b>222</b><c>333</c></line>
    <line num="L2"><a>111</a><b>222</b><c>333</c></line>
    </doc>');

    SELECT * FROM
      xpath_table('id','xml','test',
                  '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/
b|/doc/line/c',
                  'true')
      AS t(id int, doc_num varchar(10), line_num varchar(10), val1
 int4, val2 int4, val3 int4)
    WHERE id = 1 ORDER BY doc_num, line_num

     id | doc_num | line_num | val1 | val2 | val3
    ----+---------+----------+------+------+------
      1 | C1      | L1       |    1 |    2 |    3
      1 |         | L2       |   11 |   22 |   33
```

Pour obtenir `doc_num` sur chaque ligne, la solution est d'utiliser deux appels à `xpath_table` et joindre les résultats :

```
   SELECT t.*,i.doc_num FROM
     xpath_table('id', 'xml', 'test',
                 '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/
c',
                 'true')
       AS t(id int, line_num varchar(10), val1 int4, val2 int4,
 val3 int4),
     xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
       AS i(id int, doc_num varchar(10))
   WHERE i.id=t.id AND i.id=1
   ORDER BY doc_num, line_num;

    id | line_num | val1 | val2 | val3 | doc_num
   ----+----------+------+------+------+---------
     1 | L1       |    1 |    2 |    3 | C1
     1 | L2       |   11 |   22 |   33 | C1
   (2 rows)
```

F.50.4. Fonctions XSLT

Les fonctions suivantes sont disponibles si libxslt est installé.

F.50.4.1. `xslt_process`

```
    xslt_process(text document, text stylesheet, text paramlist)
 returns text
```

Cette fonction applique la feuille de style XSLT au document et renvoie le résultat transformé. Le paramètre `paramlist` est une liste de paramètres à utiliser dans la transformation, spécifiée sous la forme 'a=1,b=2'. Notez que l'analyse des paramètres est simpliste : les valeurs des paramètres ne peuvent pas contenir de virgules !

Il existe aussi une version de `xslt_process` à deux paramètres qui ne passe pas de paramètres pour la transformation.

F.50.5. Auteur

John Gray `<jgray@azuli.co.uk>`

Le développement de ce module a été sponsorisé par Torchbox Ltd. (www.torchbox.com) Il utilise la même licence BSD que PostgreSQL.

Annexe G. Programmes supplémentaires fournis

Cette annexe et la précédente contiennent des informations sur les modules disponibles dans le répertoire `contrib` de la distribution PostgreSQL. Voir Annexe F pour plus d'informations sur la section `contrib` en général et sur les extensions et plug-ins serveurs disponibles spécifiquement dans `contrib`.

Cette annexe couvre les programmes outils disponibles dans `contrib`. Une fois installés, soit à partir des sources soit à partir du système de gestion des paquets, ils sont disponibles dans le répertoire `bin` de l'installation PostgreSQL et peuvent être utilisés comme n'importe quel autre programme.

G.1. Applications clients

Cette section couvre les applications clients PostgreSQL du répertoire `contrib`. Elles peuvent être exécutées n'importe où, indépendamment du serveur hôte de la base de données. Voir aussi Applications client de PostgreSQL pour des informations sur les applications clients qui font partie du cœur de PostgreSQL.

oid2name

oid2name — résoudre les OID et les noms de fichiers dans le répertoire des données de PostgreSQL

Synopsis

`oid2name` [*option*...]

Description

oid2name est un outil qui aide les administrateurs à examiner la structure des fichiers utilisée par PostgreSQL. Pour l'utiliser, vous devez connaître la structure de fichiers utilisée de la base de données. Elle est décrite dans Chapitre 73.

> **Note**
>
> Le nom « oid2name » est historique, et est maintenant plutôt contradictoire car la plupart du temps, quand vous l'utiliserez, vous aurez besoin de connaître les numéros des filenode des tables (qui sont le nom des fichiers visibles dans les répertoires des bases de données). Assurez-vous de bien comprendre la différence entre les OID des tables et leur filenode !

oid2name se connecte à une base de données cible et extrait OID, filenode, et/ou nom de table. Vous pouvez aussi afficher les OID des bases et des tablespaces.

Options

oid2name accepte les arguments suivants en ligne de commande :

`-f` *`filenode`*
`--filenode=`*`filenode`*

affiche des informations sur la table identifiée par *`filenode`*

`-i`
`--indexes`

inclut les index et séquences dans la liste

`-o` *`oid`*
`--oid=`*`oid`*

affiche des informations sur la table d'OID *`oid`*

`-q`
`--quiet`

omet les en-têtes (utile pour scripter)

`-s`
`--tablespaces`

affiche les OID des tablespaces

`-S`
`--system-objects`

inclut les objets systèmes (ceux compris dans les schémas `information_schema`, `pg_toast` et `pg_catalog`)

`-t` *`tablename_pattern`*
`--table=`*`tablename_pattern`*

affiche des informations sur les tables dont le nom correspond au motif *`motif_nom_table`*

`-V`
`--version`

Affiche la version d'oid2name, puis quitte.

`-x`
`--extended`

affiche plus d'informations sur chaque objet affiché : nom du tablespace, nom du schéma et OID

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de oid2name, puis quitte.

oid2name accepte aussi les arguments suivants sur la ligne de commande, en tant que paramètres de connexion :

`-d` *`nom_base`*
`--dbname=`*`nom_base`*

base de données où se connecter

`-h` *`hôte`*
`--host=`*`hôte`*

hôte du serveur de base de données

`-H` *`host`*

hôte du serveur de base de données. L'utilisation de ce paramètre est *dépréciée* à compter de PostgreSQL 12.

`-p` *`port`*
`--port=`*`port`*

port du serveur de base de données

`-U` *`nom_utilisateur`*
`--username=`*`nom_utilisateur`*

nom d'utilisateur pour la connexion

Pour afficher des tables spécifiques, sélectionnez les tables à afficher en utilisant `-o`, `-f` et/ou `-t`. `-o` prend un OID, `-f` prend un filenode, et `-t` prend un nom de table (en fait, c'est un modèle de type `LIKE`, donc vous pouvez utiliser `foo%` par exemple). Vous pouvez utiliser autant d'options que vous le souhaitez, et la liste incluera tous les objets en se basant sur chaque option. Mais notez que ces options peuvent seulement afficher des objets appartenant à la base de données indiquée par l'option `-d`.

Si vous n'utilisez pas `-o`, `-f` et `-t`, mais que vous passez l'option `-d`, cela listera toutes les tables dans la base nommée par l'option `-d`. Dans ce mode, les options `-S` et `-i` contrôlent ce qui est listé.

Si vous ne passez pas non plus `-d`, cela affichera une liste des OID de bases de données. Autrement, vous pouvez passer l'option `-s` pour obtenir une liste des tablespaces.

Environnement

`PGHOST`
`PGPORT`
`PGUSER`

Paramètres de connexion par défaut.

Cet utilitaire, comme la plupart des autres utilitaires PostgreSQL, utilise également les variables d'environnement supportées par libpq (voir Section 34.15).

La variable d'environnement `PG_COLOR` indique s'il faut utiliser la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto` et `never`.

Notes

oid2name requiert une base de données en cours d'exécution avec des catalogues systèmes non corrompus. Du coup, son utilisation est assez limitée si le but est de restaurer certains objets après une corruption importante de la base de données.

Exemples

```
$ # quelles sont les bases disponibles ?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # OK, jetons un œil à la base alvherre
$ cd $PGDATA/base/17228

$ # récupérons les 10 premiers objets de la base dans le tablespace
 par défaut
$ # et triés par taille
$ ls -lS * | head -10
-rw-------  1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw-------  1 alvherre alvherre  17965056 sep 14 09:51 1155291
-rw-------  1 alvherre alvherre   1204224 sep 14 09:51 16717
-rw-------  1 alvherre alvherre    581632 sep  6 17:51 1255
-rw-------  1 alvherre alvherre    237568 sep 14 09:50 16674
-rw-------  1 alvherre alvherre    212992 sep 14 09:51 1249
-rw-------  1 alvherre alvherre    204800 sep 14 09:51 16684
-rw-------  1 alvherre alvherre    196608 sep 14 09:50 16700
-rw-------  1 alvherre alvherre    163840 sep 14 09:50 16699
-rw-------  1 alvherre alvherre    122880 sep  6 17:51 16751
```

```
$ # à quoi correspond le fichier 155173 ?
$ oid2name -d alvherre -f 155173
From database "alvherre":
  Filenode  Table Name
----------------------
    155173    accounts

$ # vous pouvez demander plus d'un objet à la fois
$ oid2name -d alvherre -f 155173 -f 1155291
From database "alvherre":
  Filenode     Table Name
------------------------
    155173       accounts
   1155291  accounts_pkey

$ # vous pouvez mélanger les options et obtenir plus de détails
 avec -x
$ oid2name -d alvherre -t accounts -f 1155291 -x
From database "alvherre":
  Filenode     Table Name      Oid  Schema  Tablespace
-----------------------------------------------------
    155173       accounts   155173  public  pg_default
   1155291  accounts_pkey  1155291  public  pg_default

$ # affiche l'espace disque pour chaque objet d'une base de données
$ du [0-9]* |
> while read SIZE FILENODE
> do
>   echo "$SIZE       `oid2name -q -d alvherre -i -f $FILENODE`"
> done
16            1155287  branches_pkey
16            1155289  tellers_pkey
17561            1155291  accounts_pkey
...

$ # pareil, mais trié par taille
$ du [0-9]* | sort -rn | while read SIZE FN
> do
>   echo "$SIZE   `oid2name -q -d alvherre -f $FN`"
> done
133466             155173    accounts
17561            1155291  accounts_pkey
1177              16717  pg_proc_proname_args_nsp_index
...

$ # Si vous voulez voir ce qu'il y a dans un tablespace, utilisez
 le répertoire
$ # pg_tblspc
$ cd $PGDATA/pg_tblspc
$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk
```

```
$ # quelle base de données a des objets dans le tablespace
 "fastdisk" ?
$ ls -d 155151/*
155151/17228/  155151/PG_VERSION

$ # Oh, quelle était la base de données 17228 ?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ # Voyons si quels objets de cette base sont dans ce tablespace.
$ cd 155151/17228
$ ls -l
total 0
-rw-------  1 postgres postgres 0 sep 13 23:20 155156

$ # OK, c'est une table très petite, mais laquelle est-ce ?
$ oid2name -d alvherre -f 155156
From database "alvherre":
  Filenode  Table Name
----------------------
    155156         foo
```

Auteur

B. Palmer <bpalmer@crimelabs.net>

vacuumlo

vacuumlo — supprimer les Large Objects orphelins à partir d'une base de données PostgreSQL

Synopsis

`vacuumlo` [*option*...] *nom_base*...

Description

vacuumlo est un outil simple qui supprimera tous les « Large Objects » « orphelins » d'une base de données PostgreSQL. Un « Large Object » orphelin est tout « Large Object » dont l'OID n'apparaît dans aucune colonne `oid` ou `lo` de la base de données.

Si vous l'utilisez, vous pourriez être intéressé par le trigger `lo_manage` du module lo. `lo_manage` est utile pour tenter d'éviter la création de « Large Object » orphelins.

Toutes les bases de données indiquées sur la ligne de commande sont traitées.

Options

vacuumlo accepte les arguments suivants en ligne de commande :

`-l` *limite*
`--limit=`*limite*

Ne supprime pas plus de *limite* Large Objects par transactions (par défaut 1000). Comme le serveur acquiert un verrou par Large Object à supprimer, supprimer beaucoup de Large Objects en une seule transaction risque de dépasser la limite imposée par le paramètre max_locks_per_transaction. Configurez la limite à zéro si vous voulez tout supprimer en une seule transaction.

`-n`
`--dry-run`

Ne supprime rien, affiche simplement ce qu'il aurait fait.

`-v`
`--verbose`

Écrit de nombreux messages de progression.

`-V`
`--version`

Affiche la version de vacuumlo, puis quitte.

`-?`
`--help`

Affiche l'aide sur les arguments en ligne de commande de vacuumlo, puis quitte.

vacuumlo accepte aussi les arguments en ligne de commande pour les paramètres de connexion :

`-h` *nom_hôte*
`--host=`*nom_hôte*

Hôte du serveur de la base.

`-p` *port*
`--port=`*port*

Port du serveur.

`-U` *nom_utilisateur*
`--username=`*nom_utilisateur*

Nom d'utilisateur pour la connexion.

`-w`
`--no-password`

Ne demande jamais un mot de passe. Si le serveur en réclame un pour l'authentification et qu'un mot de passe n'est pas disponible d'une autre façon (par exemple avec le fichier `.pgpass`), la tentative de connexion échouera. Cette option peut être utile pour les scripts où aucun utilisateur n'est présent pour saisir un mot de passe.

`-W`
`--password`

Force vacuumlo à demander un mot de passe avant la connexion à une base de données.

Cette option n'est jamais obligatoire car vacuumlo demandera automatiquement un mot de passe si le serveur exige une authentification par mot de passe. Néanmoins, vacuumlo perdra une tentative de connexion pour trouver que le serveur veut un mot de passe. Dans certains cas, il est préférable d'ajouter l'option `-W` pour éviter la tentative de connexion.

Environnement

`PGHOST`
`PGPORT`
`PGUSER`

Paramètres de connexion par défaut.

Cet utilitaire, comme la plupart des autres utilitaires PostgreSQL, utilise également les variables d'environnement supportées par libpq (voir Section 34.15).

La variable d'environnement `PG_COLOR` indique s'il faut, ou non, utiliser de la couleur dans les messages de diagnostic. Les valeurs possibles sont `always`, `auto` et `never`.

Notes

vacuumlo fonctionne avec la méthode suivante. Tout d'abord, vacuumlo construit une table temporaire contenant tous les OID des Large Objects se trouvant dans la base sélectionnée. Puis, il parcourt toutes les colonnes de la base qui sont du type `oid` ou `lo`, et supprime toutes les entrées correspondantes de la table temporaire. (Note : seuls sont pris en compte les types de ces noms ; en particulier, les domaines utilisant ces types ne sont pas pris en compte.) Les enregistrements restants dans la table temporaire sont identifiés comme les Large Objects orphelins. Ils sont supprimés.

Auteur

Peter Mount `<peter@retep.org.uk>`

G.2. Applications serveurs

Certaines applications s'exécutent sur le serveur PostgreSQL lui-même. Actuellement, aucune application de ce type n'est incluse dans le répertoire `contrib`. Voir aussi Applications relatives au

serveur PostgreSQL pour des informations sur les applications serveurs qui font partie du cœur de PostgreSQL.

Annexe H. Projets externes

PostgreSQL est un projet complexe et difficile à gérer. Il est souvent plus efficace de développer des améliorations à l'extérieur du projet principal.

H.1. Interfaces client

Il n'existe que deux interfaces clients dans la distribution de base de PostgreSQL :

- libpq, car il s'agit de l'interface principale pour le langage C et parce que de nombreuses interfaces clients sont construites par dessus ;
- ECPG, car il dépend de la grammaire SQL côté serveur et est donc sensible aux modifications internes de PostgreSQL.

Toutes les autres interfaces sont des projets externes et sont distribuées séparément. Une liste des interfaces[1] est maintenue dans le wiki PostgreSQL. Certains de ces projets peuvent ne pas être distribués sous la même licence que PostgreSQL. Pour obtenir plus d'informations sur chaque interface, avec les termes de la licence, référez-vous au site web et à la documentation.

https://wiki.postgresql.org/wiki/List_of_drivers

H.2. Outils d'administration

Différents outils d'administration sont disponibles pour PostgreSQL. Le plus populaire est pgAdmin[2] mais il existe aussi plusieurs outils commerciaux.

H.3. Langages procéduraux

PostgreSQL inclut plusieurs langages procéduraux avec la distribution de base : PL/PgSQL, PL/Tcl, PL/Perl et PL/Python.

Il existe également d'autres langages procéduraux développés et maintenus en dehors de la distribution principale de PostgreSQL. distribution. Une liste de langages de procédures[3] est maintenu sur le wiki PostgreSQL. Certains de ces projets peuvent ne pas être distribués sous la même licence que PostgreSQL. Pour obtenir plus d'informations sur chaque langage, avec les termes de la licence, on se référera au site web et à la documentation.

https://wiki.postgresql.org/wiki/PL_Matrix

H.4. Extensions

PostgreSQL est conçu pour être facilement extensible. C'est pour cette raison que les extensions chargées dans la base de données peuvent fonctionner comme les fonctionnalités intégrées au SGBD. Le répertoire `contrib/` livré avec le code source contient un grand nombre d'extensions, qui sont décrites dans Annexe F. D'autres extensions sont développées indépendamment, comme PostGIS[4]. Même les solutions de réplication de PostgreSQL peuvent être développées en externe. Ainsi, Slony-I[5], solution populaire de réplication primaire/secondaire, est développée indépendamment du projet principal.

[1] https://wiki.postgresql.org/wiki/List_of_drivers
[2] https://www.pgadmin.org/
[3] https://wiki.postgresql.org/wiki/PL_Matrix
[4] https://postgis.net/
[5] https://www.slony.info

Annexe I. Dépôt du code source

Le code source de PostgreSQL est stocké et géré en utilisant le système de contrôle de version appelé Git. Un miroir public du dépôt maître est disponible ; il est mis à jour une minute après chaque changement du dépôt maître.

Notre wiki, https://wiki.postgresql.org/wiki/Working_with_Git, contient des informations sur l'utilisation de Git.

Notez que la construction de PostgreSQL à partir du dépôt des sources nécessite des versions raisonnablement récentes de bison, flex et Perl. Ces outils ne sont pas nécessaires pour construire à partir d'une archive tar distribuée car les fichiers que ces outils sont utilisés pour les construire sont inclus dans l'archive tar. Les autres besoins en outillage sont les mêmes que ceux exposés dans Section 17.2.

I.1. Récupérer les sources via Git

Avec Git, vous devez avoir une copie du dépôt de code sur votre machine locale, pour que vous ayez accès à tout l'historique et les branches sans avoir besoin d'être en ligne. C'est le moyen le plus rapide et le plus flexible pour développer ou tester des correctifs.

Git

1. Vous aurez besoin d'une version installée de Git, que vous pouvez obtenir sur https://git-scm.com. La plupart des systèmes ont actuellement une version récente de Git installée par défaut ou disponible dans le système de paquets.

2. Pour commencer à utiliser le dépôt Git, commencez par faire un clone du miroir officiel :

   ```
   git clone https://git.postgresql.org/git/postgresql.git
   ```

   Ceci va copier intégralement le dépôt sur votre machine locale, ce qui prendra un peu de temps, notamment si vous avez une connexion lente. Les fichiers seront placés dans un nouveau sous-répertoire de votre répertoire courant, qui sera nommé `postgresql`.

3. À chaque fois que vous voulez obtenir les dernières mises à jour, allez dans le dépôt avec la commande `cd` et exécutez la commande qui suit :

   ```
   git fetch
   ```

Git peut faire bien plus de choses que de simplement récupérer les sources. Pour plus d'informations, consultez les pages de manuel de Git ou visitez le site https://git-scm.com.

Annexe J. Documentation

PostgreSQL fournit quatre formats principaux de documentation :

- le texte brut, pour les informations de pré-installation ;
- HTML, pour la lecture en ligne et les références ;
- PDF, pour l'impression ;
- les pages man (de manuel), pour la référence rapide.

De plus, un certain nombre de fichiers `README` peuvent être trouvés à divers endroits de l'arbre des sources de PostgreSQL. Ils renseignent l'utilisateur sur différents points d'implantation.

La documentation HTML et les pages de manuel font parties de la distribution standard et sont installées par défaut. Les documents au format PDF sont disponibles indépendemment par téléchargement.

J.1. DocBook

Les sources de la documentation sont écrites en *DocBook*, langage assez semblable au XML. Dans ce qui suit, les termes DocBook et XML sont tous deux utilisés, mais ils ne sont pas techniquement interchangeables.

DocBook permet à l'auteur de spécifier la structure et le contenu d'un document technique sans qu'il ait à se soucier du détail de la présentation. Un style de document définit le rendu du contenu dans un des formats de sortie finaux. DocBook est maintenu par le groupe OASIS[1]. Le site officiel de DocBook[2] présente une bonne documentation d'introduction et de référence ainsi qu'un livre complet de chez O'Reilly disponible à la lecture en ligne. Le projet de documentation FreeBSD[3] utilise également DocBook et fournit également de bonnes informations, incluant un certain nombre de lignes directrices qu'il peut être bon de prendre en considération.

J.2. Ensemble d'outils

Les outils qui suivent sont utilisés pour produire la documentation. Certains sont optionnels (comme mentionné).

DTD DocBook[4]

Il s'agit de la définition de DocBook elle-même. C'est actuellement la version 4.5 qui est utilisée. Vous avez besoin de la variante XML de la DTD DocBook, et non pas SGML, de même version. Ils seront généralement dans des paquets séparés.

DocBook XSL Stylesheets[5]

Ils contiennent les instructions de traitement pour convertir les sources DocBook vers d'autres formats, comme par exemple le HTML.

La version minimale requise est actuellement la 1.77.0, mais il est recommandé d'utiliser la dernière version disponible pour de meilleurs résultats.

[1] https://www.oasis-open.org
[2] https://www.oasis-open.org/docbook
[3] https://www.freebsd.org/docproj/
[4] https://www.oasis-open.org/docbook/
[5] https://github.com/docbook/wiki/wiki/DocBookXslStylesheets

Libxml2[6] for `xmllint`

Cette bibliothèque et l'outil `xmllint` qu'il contient sont utilisés pour traiter du XML. Beaucoup de développeurs ont déjà Libxml2 installé car il est aussi utilisé lors de la compilation de PostgreSQL. Néanmoins, notez que `xmllint` doit être installé à partir d'un sous-paquet séparé.

Libxslt[7] pour `xsltproc`

`xsltproc` est un processeur XSLT, c'est-à-dire, un programme pour convertir le XML vers d'autres formats en utilisant des fichiers de style XSLT.

FOP[8]

Il s'agit d'un programme pour convertir, entre autres choses, du XML vers du PDF. Il est uniquement nécessaire si vous voulez construire la documentation au format PDF.

Différentes méthodes d'installation sont détaillées ci-après pour les divers outils nécessaires au traitement de la documentation. Il peut exister d'autres types de distributions empaquetées de ces outils. Tout changement du statut d'un paquetage peut être rapporté auprès de la liste de discussion de la documentation, afin d'inclure ces informations ici-même.

J.2.1. Installation sur Fedora, RHEL et dérivés

Pour installer les packages requis, lancez :

```
yum install docbook-dtds docbook-style-xsl libxslt fop
```

J.2.2. Installation sur FreeBSD

Pour installer les paquets requis avec `pkg`, utiliser :

```
pkg install docbook-xml docbook-xsl libxslt fop
```

Quand vous compilez la documentation depuis le répertoire `doc` vous aurez besoin d'utiliser `gmake`, car le makefile fourni n'est pas correct pour le `make` de FreeBSD.

J.2.3. Paquetages Debian

Un ensemble complet de paquetages d'outils de documentation est disponible pour Debian GNU/Linux. Pour l'installer, il suffit de taper :

```
apt-get install docbook-xml docbook-xsl libxml2-utils xsltproc fop
```

J.2.4. macOS

Si vous utilisez MacPorts, les commandes suivantes vous aideront à la configuration du système :

```
sudo port install docbook-xml docbook-xsl-nons libxslt fop
```

Si vous utilisez Homebrew, utilisez plutôt ceci :

[6] http://xmlsoft.org/
[7] http://xmlsoft.org/XSLT/
[8] https://xmlgraphics.apache.org/fop/

```
brew install docbook docbook-xsl libxslt fop
```

Le programme fourni par Homebrew requiert la configuration de la variable d'environnement suivante :

```
export XML_CATALOG_FILES=/usr/local/etc/xml/catalog
```

Sur les machines Apple Silicon, utilisez ceci :

```
export XML_CATALOG_FILES=/opt/homebrew/etc/xml/catalog
```

Sans cela, `xsltproc` renverra ce genre d'erreurs :

```
I/O error : Attempt to load network entity http://www.oasis-
open.org/docbook/xml/4.5/docbookx.dtd
postgres.sgml:21: warning: failed to load external entity "http://
www.oasis-open.org/docbook/xml/4.5/docbookx.dtd"
...
```

Alors qu'il est possible d'utiliser les versions fournies par Apple de `xmllint` et `xsltproc` à la place de celles fournies par MacPorts ou Homebrew, vous aurez toujours besoin d'installer les DTD et feuilles de style DocBook, et de configurer un fichier catalogue qui pointe vers eux.

J.2.5. Installation manuelle à partir des sources

L'installation manuelle des outils DocBook est quelque peu complexe. Il est donc préférable d'utiliser des paquetages pré-compilés. Seule une procédure de mise en œuvre standard, qui utilise des répertoires d'installation standard et sans fonctionnalités particulières, est ici décrite. Pour les détails, on peut étudier la documentation respective de chaque paquetage et lire les documents d'introduction à SGML.

J.2.5.1. Installer OpenSP

L'installation d'OpenSP fournit un processus de compilation `./configure; make; make install` de style GNU. Les détails peuvent être trouvés dans la distribution source d'OpenSP. En un mot :

```
./configure --enable-default-catalog=/usr/local/etc/sgml/catalog
make
make install
```

Veillez à vous rappeler de l'endroit où vous stockez le « catalogue par défaut »; vous en aurez besoin ci-dessous. Vous pouvez également le laisser de côté, mais vous aurez alors besoin de positionner la variable d'environnement `SGML_CATALOG_FILES` pour qu'elle point vers le fichier à chaque fois que vous utilisez l'un des programme d'OpenSP. (Cette méthode est également envisageable si OpenSP est déjà installé et que vous voulez installer le reste de la suite d'outil localement.)

J.2.5.2. Installation du kit DTD de DocBook

1. Récupérer la distribution DocBook V4.2[9].

[9] http://www.docbook.org/sgml/4.5/docbook-4.2.zip

2. Créer le répertoire `/usr/local/share/sgml/docbook-4.2` et s'y placer. (L'emplacement exact importe peu mais celui-ci a le bénéfice d'être cohérent avec le schéma d'installation proposé ici.)

   ```
   $ mkdir /usr/local/share/sgml/docbook-4.2
   $ cd /usr/local/share/sgml/docbook-4.2
   ```

3. Décompresser l'archive.

   ```
   $ unzip -a ...../docbook-4.2.zip
   ```

   (L'archive décompresse ses fichier dans le répertoire courant.)

4. Éditer le fichier `/usr/local/share/sgml/catalog` (ou celui précisé à jade lors de l'installation) et y placer une ligne similaire à celle-ci :

   ```
   CATALOG "docbook-4.2/docbook.cat"
   ```

5. Télécharger l'archive contenant les entités de caractères ISO 8879[10], la décompresser et placer les fichiers dans le même répertoire que celui des fichiers de DocBook.

   ```
   $ cd /usr/local/share/sgml/docbook-4.2
   $ unzip ...../ISOEnts.zip
   ```

6. Lancer la commande suivante dans le répertoire contenant les fichiers DocBook et ISO :

   ```
   perl -pi -e 's/iso-(.*).gml/ISO\1/g' docbook.cat
   ```

   (Cette opération permet de corriger les mélanges entre le fichier de catalogue de DocBook et les noms réels des fichiers contenant les entités de caractères ISO.)

J.2.6. Détection par `configure`

Avant de pouvoir construire la documentation, le script `configure` doit être lancé, comme cela se fait pour la construction des programmes PostgreSQL eux-mêmes. La fin de l'affichage de l'exécution de ce script doit ressembler à :

```
checking for xmllint... xmllint
checking for xsltproc... xsltproc
checking for fop... fop
checking for dbtoepub... dbtoepub
```

Si `xmllint` ou `xsltproc` est introuvable, vous ne pourrez pas construire la documentation. `fop` est seulement nécessaire pour construire la documentation en PDF. `dbtoepub` est seulement nécessaire pour construire la documentation en EPUB.

Si nécessaire, vous pouvez indiquer à `configure` où trouver ces programmes, par exemple

```
./configure ... XMLLINT=/opt/local/bin/xmllint ...
```

J.3. Construire la documentation

Lorsque tout est en place, se placer dans le répertoire `doc/src/sgml` et lancer une des commandes décrites dans les sections suivantes afin de produire la documentation. (Il est impératif d'utiliser la version GNU de make.)

[10] http://www.oasis-open.org/cover/ISOEnts.zip

J.3.1. HTML

Pour engendrer la version HTML de la documentation, effectuer :

```
doc/src/sgml$ make html
```

Il s'agit également de la cible par défaut. La sortie apparaît dans le sous-répertoire `html`.

Pour générer la documentation HTML avec la feuille de style utilisée sur postgresql.org[11] à la place de la feuille de style par défaut, utilisez :

```
doc/src/sgml$ make STYLE=website html
```

Si l'option `STYLE=website` est utilisé, les fichiers HTML générés incluent des références aux feuilles de style intégrées sur postgresql.org[12] et nécessite l'accès réseau pour les visualiser.

J.3.2. Pages man (de manuel)

Nous utilisons les feuilles de style XSL DocBook pour convertir les pages de références DocBook dans un format *roff compatible avec les pages man. Pour créer les pages man, utiliser les commandes :

```
doc/src/sgml$ make man
```

J.3.3. PDF

Pour produire un rendu PDF de la documentation en utilisant FOP, vous pouvez utiliser l'une des commandes suivantes, en fonction du format de papier préféré :

- Pour un format A4 :

  ```
  doc/src/sgml$ make postgres-A4.pdf
  ```

- Pour un format U.S. letter :

  ```
  doc/src/sgml$ make postgres-US.pdf
  ```

Puisque la documentation de PostgreSQL est assez grosse, FOP nécessitera une quantité de mémoire significative. À cause de ça, sur certains systèmes, la compilation échouera avec un message d'erreur lié à la mémoire. Cela peut généralement être corrigé en configurant les réglages de mémoire Java dans le fichier de configuration `~/.foprc`, par exemple :

```
# FOP binary distribution
FOP_OPTS='-Xmx1500m'
# Debian
JAVA_ARGS='-Xmx1500m'
# Red Hat
ADDITIONAL_FLAGS='-Xmx1500m'
```

Il y a une quantité minimale de mémoire qui est nécessaire, et utiliser plus de mémoire à l'air de rendre les choses plus rapides jusqu'à un certain point. Sur les systèmes disposant de très peu de mémoire

[11] https://www.postgresql.org/docs/current/
[12] https://www.postgresql.org/docs/current/

(moins d'1 Go), la compilation sera soit très lente du faire de l'utilisation du SWAP ou ne fonctionnera pas du tout.

D'autres processeurs XSL-FO peuvent également être utilisés manuellement, mais le processus de compilation automatique ne supporte que FOP.

J.3.4. Fichiers texte

Les instructions d'installation sont aussi distribuées sous la forme d'un fichier texte, au cas où cela se révélerait nécessaire (par exemple si des outils plus avancés n'étaient pas disponibles). Le fichier `INSTALL` correspond au Chapitre 17, avec quelques changements mineurs pour tenir compte de contextes différents. Pour recréer le fichier, se placer dans le répertoire `doc/src/sgml` et entrer la commande **`make INSTALL`**. Pour construire la sortie texte, il faut disposer de Pandoc version 1.13, ou une version ultérieure, en tant qu'outil de construction supplémentaire.

Dans le passé, les notes de versions et les instructions pour les tests de régression étaient aussi distribuées sous la forme de fichiers textes mais ce n'est plus le cas.

J.3.5. Vérification syntaxique

Fabriquer la documentation peut prendre beaucoup de temps. Il existe cependant une méthode, qui ne prend que quelques secondes, permettant juste de vérifier que la syntaxe est correcte dans les fichiers de documentation :

```
doc/src/sgml$ make check
```

J.4. Écriture de la documentation

Les sources de la documentation sont généralement modifiés avec un éditeur disposant d'un mode d'édition XML, et encore plus s'il a certaines connaissances des langages du schéma XML pour qu'il puisse connaître la syntaxe DocBook.

Notez que pour des raisons historiques, les fichiers sources de la documentation sont nommés avec une extension `.sgml` même si ce sont maintenant des fichiers XML. Donc vous aurez besoin d'ajuster la configuration de votre éditeur pour configurer le bon mode.

J.4.1. Emacs

nXML Mode, qui est fourni avec Emacs, est le mode le plus commun pour éditer des documents XML avec Emacs. Il vous permettra d'utiliser Emacs pour insérer les balises et vérifier la cohérence des balises. Il supporte DocBook directement. Vérifiez le manuel nXML[13] pour une documentation détaillée.

`src/tools/editors/emacs.samples` contient les configurations recommendées pour ce mode.

J.5. Guide des styles

J.5.1. Pages de références

Les pages de références obéissent à des règles de standardisation. De cette façon, les utilisateurs retrouvent plus rapidement l'information souhaitée, et cela encourage également les rédacteurs à documenter tous les aspects relatifs à une commande. Cette cohérence n'est pas uniquement souhaitée

[13] https://www.gnu.org/software/emacs/manual/html_mono/nxml-mode.html

pour les pages de références PostgreSQL, mais également pour les pages de références fournies par le système d'exploitation et les autres paquetages. C'est pour cela que les règles suivantes ont été développées. Elles sont, pour la plupart, cohérentes avec les règles similaires établies pour différents systèmes d'exploitation.

Les pages de référence qui décrivent des commandes exécutables doivent contenir les sections qui suivent dans l'ordre indiqué. Les sections qui ne sont pas applicables peuvent être omises. Des sections de premier niveau additionnelles ne doivent être utilisées que dans des circonstances particulières ; dans la plupart des cas, les informations qui y figureraient relèvent de la section « Usage ».

Nom

Cette section est produite automatiquement. Elle contient le nom de la commande et une courte phrase résumant sa fonctionnalité.

Synopsis

Cette section contient le schéma syntaxique de la commande. Le synopsis ne doit en général pas lister toutes les options de la commande, cela se fait juste au dessous. À la place, il est important de lister les composantes majeures de la ligne de commande comme, par exemple, l'emplacement des fichiers d'entrée et sortie.

Description

Plusieurs paragraphes décrivant ce que permet de faire la commande.

Options

Une liste décrivant chacune des options de la ligne de commande. S'il y a beaucoup d'options, il est possible d'utiliser des sous-sections.

Code de sortie

Si le programme utilise 0 en cas de succès et une valeur non-nulle dans le cas contraire, il n'est pas nécessaire de le documenter. S'il y a une signification particulière au code de retour différent de zéro, c'est ici qu'ils faut décrire les codes de retour.

Utilisation

Décrire ici tout sous-programme ou interface de lancement du programme. Si le programme n'est pas interactif, cette section peut être omise. Dans les autres cas, cette section est un fourre-tout pour les fonctionnalités disponibles lors de l'utilisation du programme. Utiliser des sous-sections si cela est approprié.

Environnement

Lister ici toute variable d'environnement utilisable. Il est préférable de ne rien omettre. Même des variables qui semblent triviales, comme `SHELL`, peuvent être d'un quelconque intérêt pour l'utilisateur.

Fichiers

Lister tout fichier que le programme peut accéder, même implicitement. Les fichiers d'entrée ou de sortie indiqués sur la ligne de commande ne sont pas listés, mais plutôt les fichiers de configuration, etc.

Diagnostiques

C'est ici que l'ont trouve l'explication de tout message inhabituel produit par le programme. Il est inutile de lister tous les messages d'erreur possibles. C'est un travail considérable et cela n'a que peu d'intérêt dans la pratique. En revanche, si les messages d'erreurs ont un format particulier, que l'utilisateur peut traiter, c'est dans cette section que ce format doit être décrit.

Notes

Tout ce qui ne peut être contenu dans les autres sections peut être placé ici. En particulier les bogues, les carences d'une implantation, les considérations de sécurité et les problèmes de compatibilité.

Exemples

Les exemples.

Historique

S'il y a eu des échéances majeures dans l'histoire du programme, elles peuvent être listées ici. Habituellement, cette section peut être omise.

Author

Auteur (seulement utilisé dans la section des modules supplémentaires)

Voir aussi

Des références croisées, listées dans l'ordre suivant : pages de référence vers d'autres commandes PostgreSQL, pages de référence de commandes SQL de PostgreSQL, citation des manuels PostgreSQL, autres pages de référence (système d'exploitation, autres paquetages, par exemple), autre documentation. Les éléments d'un même groupe sont listés dans l'ordre alphabétique.

Les pages de référence qui décrivent les commandes SQL doivent contenir les sections suivantes : « Nom », « Synopsis », « Description », « Paramètres », « Sorties », « Notes », « Exemples », « Compatibilité », « Historique », « Voir aussi ». La section « Paramètres » est identique à la section « Options » mais elle offre plus de liberté sur les clauses qui peuvent être listées. La section « Sorties » n'est nécessaire que si la commande renvoie autre chose qu'un complément de commande par défaut. La section « Compatibilité » doit expliquer dans quelle mesure une commande se conforme au standard SQL, ou avec quel autre système de gestion de base de données elle est compatible. La section « Voir aussi » des commandes SQL doit lister les commandes SQL avant de faire référence aux programmes.

Annexe K. Limites PostgreSQL

Tableau K.1 décrit les diverses limites dures de PostgreSQL. Cependant, les limites physiques telles que les limitations de performance ou l'espace disque disponible peuvent s'appliquer avant que des limites dures absolues soient atteintes.

Tableau K.1. Limitations de PostgreSQL

Élément	Limite supérieure	Commentaire
taille de la base de données	illimitée	
nombre de bases de données	4 294 950 911	
relations par base de données	1 431 650 303	
taille d'une relation	32 To	Avec un BLCKSZ par défaut de 8192 octets
lignes par table	limité par le nombre de lignes qui peuvent être contenus dans 4 294 967 295 blocs	
colonnes par table	1600	aussi limité par la taille des lignes contenues dans un simple bloc ; voir la note ci-dessous
colonnes par ensemble de résultats	1664	
taille d'un champ	1 Go	
index par table	illimitée	limité par le nombre maximum de relations par base de données
colonnes par index	32	peut être augmenté en recompilant PostgreSQL
clés de partition	32	peut être augmenté en recompilant PostgreSQL
longueur de l'identifiant	63 octets	peut être augmenté en recompilant PostgreSQL
arguments de fonction	100	peut être augmenté en recompilant PostgreSQL
paramètres de requête	65 535	

Le nombre maximum de colonnes pour une table est d'autant plus réduit que la ligne stockée doit être contenue dans un simple bloc de 8192 octets. Par exemple, sans compter l'entête d'une ligne, une ligne constituée de 1600 colonnes de type `int` consommera 6400 octets et peut être stocké dans un bloc, mais une ligne constituée de 1600 colonnes de type `bigint` consommera 12800 octets et ne pourra pas tenir dans le bloc. Les champs typés `text`, `varchar` et `char` de longueur variable peuvent avoir leurs valeurs stockées à l'extérieur dans une table TOAST lorsque cette valeur est suffisamment importante pour que cela se produise. Un pointeur de 18 octets doit rester à l'intérieur de la ligne dans la table. Pour des longueurs moins importantes de champs à longueur variable, un champ entête de 4-octets ou 1-octet est utilisé et la valeur est stockée à l'intérieur de la ligne.

Les colonnes qui ont été supprimées de la table participent aussi à la limite maximum de colonnes. De plus, bien que les valeurs des colonnes supprimées pour les nouvelles lignes créées soient marquées en interne comme null, ainsi que dans le champ de bits des valeurs null de la ligne, ce champ consomme de l'espace.

Chaque table peut enregistrer un maximum théorique de 2^32 valeurs hors ligne ; voir Section 73.2 pour une discussion détaillée du stockage hors ligne. Cette limite vient de l'utilisation d'un OID sur

32 bits pour identifier chaque valeur. La limite pratique est significativement moindre que la limite théorique parce qu'au fur et à mesure que l'espace OID se remplit, trouver un OID toujours libre devient coûteux, ralentissant de ce fait les instructions INSERT/UPDATE. Typiquement, c'est seulement un problème pour les tables contenant plusieurs téraoctets de données ; partitionner est un contournement possible.

Annexe L. Acronymes

La suite présente la liste des acronymes habituellement utilisés dans la documentation de PostgreSQL et les discussions qui tournent autour de PostgreSQL.

ANSI

American National Standards Institute[1], l'Institut National Américain des Standards

API

Application Programming Interface[2], interface de programmation applicative

ASCII

American Standard Code for Information Interchange[3], Code Standard Américain pour l'échange d'informations

BKI

Backend Interface, interface serveur

CA

Certificate Authority[4], autorité de certification

CIDR

Classless Inter-Domain Routing[5], routage inter-domaine dépourvu de classe

CPAN

Comprehensive Perl Archive Network[6], réseau d'archives Perl

CRL

Certificate Revocation List[7], liste de révocation de certificats

CSV

Comma-Separated Values[8], valeurs séparées par des virgules (format de fichier présentant les données en ligne en séparant les colonnes par des virgules ou points-virgules)

CTE

Common Table Expression

CVE

Common Vulnerabilities and Exposures[9], vulnérabilités et expositions courantes

DBA

Database Administrator[10], administrateur de bases de données

[1] https://fr.wikipedia.org/wiki/American_National_Standards_Institute
[2] https://fr.wikipedia.org/wiki/API
[3] https://fr.wikipedia.org/wiki/Ascii
[4] https://fr.wikipedia.org/wiki/Certificate_authority
[5] https://fr.wikipedia.org/wiki/Classless_Inter-Domain_Routing
[6] https://www.cpan.org/
[7] https://fr.wikipedia.org/wiki/Certificate_revocation_list
[8] https://fr.wikipedia.org/wiki/Comma-separated_values
[9] https://cve.mitre.org/
[10] https://fr.wikipedia.org/wiki/Database_administrator

DBI

Database Interface (Perl)[11], interface avec la base de données

DBMS

Database Management System[12], système de gestion de bases de données (SGBD)

DDL

Data Definition Language[13], langage de définition des données, qui regroupe les commandes SQL telles que `CREATE TABLE`, `ALTER USER`

DML

Data Manipulation Language[14], langage de manipulation des données, qui regroupe les commandes SQL telles que `INSERT`, `UPDATE`, `DELETE`

DST

Daylight Saving Time[15], gestion des changements d'heure instaurés pour des raisons d'économie d'énergie

ECPG

Embedded C for PostgreSQL, C embarqué pour PostgreSQL

ESQL

Embedded SQL[16], SQL embarqué

FAQ

Frequently Asked Questions[17], Foire aux Questions

FSM

Free Space Map, cartographie de l'espace libre

GEQO

Genetic Query Optimizer, optimiseur de requête génétique

GIN

Generalized Inverted Index, index inversé généralisé

GiST

Generalized Search Tree, arbre de recherche généralisé

Git

Git[18],

[11] https://dbi.perl.org/
[12] https://fr.wikipedia.org/wiki/Dbms
[13] https://fr.wikipedia.org/wiki/Data_Definition_Language
[14] https://fr.wikipedia.org/wiki/Data_Manipulation_Language
[15] https://fr.wikipedia.org/wiki/Daylight_saving_time
[16] https://fr.wikipedia.org/wiki/Embedded_SQL
[17] https://fr.wikipedia.org/wiki/FAQ
[18] https://en.wikipedia.org/wiki/Git_(software)

GMT

Greenwich Mean Time[19], heure au méridien de Greenwich

GSSAPI

Generic Security Services Application Programming Interface[20], Interface de programmation applicative pour les services de sécurité génériques

GUC

Grand Unified Configuration, Configuration générale unifiée, le sous-système PostgreSQL qui gère la configuration du serveur

HBA

Host-Based Authentication, authentification fondée sur l'hôte

HOT

Heap-Only Tuples

IEC

International Electrotechnical Commission[21], Commission internationale d'électrotechnique

IEEE

Institute of Electrical and Electronics Engineers[22], institut des ingénieurs en électrique et électronique

IPC

Inter-Process Communication[23], communication inter-processus

ISO

International Organization for Standardization[24], Organisation de standardisation internationale

ISSN

International Standard Serial Number[25], numéro de série international standardisé

JDBC

Java Database Connectivity[26], connecteur de bases de données en Java

JIT

Just-in-Time compilation[27]

JSON

JavaScript Object Notation[28]

[19] https://fr.wikipedia.org/wiki/GMT
[20] https://fr.wikipedia.org/wiki/Generic_Security_Services_Application_Program_Interface
[21] https://fr.wikipedia.org/wiki/International_Electrotechnical_Commission
[22] https://standards.ieee.org/
[23] https://fr.wikipedia.org/wiki/Inter-process_communication
[24] https://www.iso.org/iso/home.htm
[25] https://fr.wikipedia.org/wiki/Issn
[26] https://fr.wikipedia.org/wiki/Java_Database_Connectivity
[27] https://en.wikipedia.org/wiki/Just-in-time_compilation
[28] http://json.org

LDAP

Lightweight Directory Access Protocol[29], protocole léger d'accès aux annuaires

LSN

Log Sequence Number, Numéro de séquence dans les journaux de transaction. Voir `pg_lsn` et Vue interne des journaux de transaction.

MITM

Attaque Man-in-the-middle[30]

MSVC

Microsoft Visual C[31]

MVCC

Multi-Version Concurrency Control, Contrôle de concurrence par multi-versionnage

NLS

National Language Support[32], support des langages nationaux

ODBC

Open Database Connectivity[33], connecteur ouvert de bases de données

OID

Object Identifier, identifiant objet

OLAP

Online Analytical Processing[34], traitement analytique en ligne

OLTP

Online Transaction Processing[35], traitement transactionnel en ligne

ORDBMS

Object-Relational Database Management System[36], système de gestion de bases de données relationnelles objet (SGBR/O)

PAM

Pluggable Authentication Modules[37], modules d'authentification connectables

PGSQL

PostgreSQL

[29] https://fr.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol
[30] https://en.wikipedia.org/wiki/Man-in-the-middle_attack
[31] https://fr.wikipedia.org/wiki/Visual_C++
[32] https://fr.wikipedia.org/wiki/Internationalization_and_localization
[33] https://fr.wikipedia.org/wiki/Open_Database_Connectivity
[34] https://fr.wikipedia.org/wiki/Olap
[35] https://fr.wikipedia.org/wiki/OLTP
[36] https://fr.wikipedia.org/wiki/ORDBMS
[37] https://fr.wikipedia.org/wiki/Pluggable_Authentication_Modules

PGXS

PostgreSQL Extension System, système d'extension de PostgreSQL

PID

Process Identifier[38], identifiant processus

PITR

Point-In-Time Recovery, Restauration d'un instantané

PL

Procedural Languages, langages procéduraux côté serveur

POSIX

Portable Operating System Interface[39], interface portable au système d'exploitation

RDBMS

Relational Database Management System[40], système de gestion de bases de données relationnelles (SGBDR)

RFC

Request For Comments[41]

SGML

Standard Generalized Markup Language[42]

SNI

Server Name Indication[43], RFC 6066[44]

SPI

Server Programming Interface, interface de programmation serveur

SP-GiST

Space-Partitioned Generalized Search Tree

SQL

Structured Query Language[45]

SRF

Set-Returning Function, fonction retournant un ensemble

SSH

Secure Shell[46]

[38] https://fr.wikipedia.org/wiki/Process_identifier
[39] https://fr.wikipedia.org/wiki/POSIX
[40] https://fr.wikipedia.org/wiki/Relational_database_management_system
[41] https://fr.wikipedia.org/wiki/Request_for_Comments
[42] https://fr.wikipedia.org/wiki/SGML
[43] https://en.wikipedia.org/wiki/Server_Name_Indication
[44] https://datatracker.ietf.org/doc/html/rfc6066#section-3
[45] https://fr.wikipedia.org/wiki/SQL
[46] https://fr.wikipedia.org/wiki/Secure_Shell

SSL

Secure Sockets Layer[47]

SSPI

Security Support Provider Interface[48]

SYSV

Unix System V[49]

TCP/IP

Transmission Control Protocol (TCP) / Internet Protocol (IP)[50]

TID

Tuple Identifier, identifiant de tuple

TLS

Transport Layer Security[51]

TOAST

The Oversized-Attribute Storage Technique, technique de stockage des données surdimensionnées

TPC

Transaction Processing Performance Council[52]

URL

Uniform Resource Locator[53]

UTC

Coordinated Universal Time[54]

UTF

Unicode Transformation Format[55]

UTF8

Eight-Bit Unicode Transformation Format[56]

UUID

Universally Unique Identifier, identifiant universel unique

[47] https://fr.wikipedia.org/wiki/Secure_Sockets_Layer
[48] https://msdn.microsoft.com/en-us/library/aa380493%28VS.85%29.aspx
[49] https://fr.wikipedia.org/wiki/System_V
[50] https://fr.wikipedia.org/wiki/Transmission_Control_Protocol
[51] https://en.wikipedia.org/wiki/Transport_Layer_Security
[52] http://www.tpc.org/
[53] https://fr.wikipedia.org/wiki/URL
[54] https://fr.wikipedia.org/wiki/Coordinated_Universal_Time
[55] https://www.unicode.org/
[56] https://fr.wikipedia.org/wiki/Utf8

WAL

Write-Ahead Log

XID

Transaction Identifier, identifiant de transaction

XML

Extensible Markup Language[57]

[57] https://fr.wikipedia.org/wiki/XML

Annexe M. Glossaire

Ce chapitre est une liste de termes et leur signification dans le contexte de PostgreSQL et des systèmes de base de données relationnelles en général.

ACID
: *Atomicité*, *Cohérence*, *Isolation* et *Durabilité*. Cet ensemble de propriétés des transactions de base de données permet de garantir la validité lors des opérations concurrentes et même en cas d'événements d'erreurs, coupures de courant, ...

Analyse (traitement) (*Analysis (process)*)
: Le processus de collecter des statistiques des données en *tables* et autres *relations* pour aider le *planificateur de requêtes* à prendre les bonnes décisions sur l'exécution des *requêtes*.

  (Ne confondez pas ce terme avec l'option ANALYZE de la commande EXPLAIN.)

  Pour plus d'informations, voir ANALYZE.

Atomicité (*Atomicity*)
: La propriété d'une *transaction* qui assure que toutes ses opérations se terminent comme une seule unité au complet ou pas du tout. De plus, si une panne système se produit lors de l'exécution d'une transaction, aucun résultat partiel n'est visible après la restauration. C'est une des propriétés ACID.

Atomique (*Atomic*)
: En référence à une *donnée* : Le fait que sa valeur ne peut pas être divisée en plus petits composants.

  En référence à une *transaction de base de données* : voir *atomicité*.

Attribut
: Un élément avec un certain nom et un certain type de données trouvé dans une *ligne*.

Autovacuum
: Un ensemble de processus en tâche de fond qui effectuent régulièrement les opérations *vacuum* et *analyze*. Le *processus auxiliaire* qui coordonne le travail est toujours présent (sauf si autovacuum est désactivé) est connu sous le nom de *autovacuum launcher*, et les processus qui font le boulot sont appelés *autovacuum workers*.

  Pour plus d'informations, voir Section 25.1.6.

Processus auxiliaire
: Un processus à l'intérieur d'une *instance* en charge d'une tâche de fond spécifique pour l'instance. Voici quelques processus auxiliaire le *autovacuum launcher* (mais pas les workersworkers), le *background writer*, le *checkpointer*, le *logger*, le *startup process*, le *WAL archiver*, le *WAL receiver* (mais par les *WAL senders*), et le *WAL writer*.

Backend (processus)
: Processus d'une *instance* agissant au nom du client et gérant ses requêtes.

  Ne confondez pas ce terme avec les termes similaires *Background Worker* ou *Background Writer*.

Background worker (processus)
: Processus dans l'*instance*, qui exécute le système - ou le code fourni par les utilisateurs. Sert comme infrastructure pour

de nombreuses fonctionnalités dans PostgreSQL, telles que la *réplication logique* et les *requêtes parallélisées*. De plus, les *extensions* peuvent ajouter des processus background workers personnalisés.

Pour plus d'informations, voir Chapitre 48.

Background writer (processus)

Un *processus auxiliaire* qui écrit les *blocs de données* modifiés sur le système de fichiers. Il se réveille périodiquement mais ne fonctionne que sur une courte période de temps dans le but de distribuer sa coûteuse activité I/O dans le temps pour éviter de larges pics d'I/O qui peuvent bloquer les autres processus.

Pour plus d'informations, voir Section 20.4.5.

Base de données (*Database*)

Une collection nommée d' *objets SQL locaux*.

Pour plus d'informations, voir Section 23.1.

Blocs de données (*Data page*)

La structure basique utilisée pour stocker des données de relation. Tous les blocs ont la même taille. Les blocs de données sont typiquement stockés sur disque, chacun dans un fichier spécifique, et peuvent être lus dans le *tampon de mémoire partagée* où ils peuvent être modifiés, devenant *sales* (*dirty* en anglais). Elles deviennent propres quand elles sont écrites sur le disque. Les nouvelles pages, qui initialement existent en mémoire seulement, sont aussi sales jusqu'à leur écriture.

Carte des espaces libres (*Free space map (fork)*)

Une structure de stockage qui garde les méta-données sur chaque bloc de données du fork principal d'une table. Les entrées de la carte des espaces libres pour chaque bloc stocke la quantité d'espace libre disponibles pour les futurs enregistrements, et est structurée pour être rendre performante la recherche d'espace libre pour les nouveaux enregistrements d'une taille donnée.

Pour plus d'informations, voir Section 73.3.

Carte de visibilité (*Visibility Map*)

Une structure sur disque qui conserve des métadonnées sur chaque bloc de données d'une table. Un enregistrement de la carte de visibilité pour chaque bloc enregistre deux bits : le premier (`tous-visibles`) indique que tous les enregistrements de ce blocs sont visibles par toutes les transactions. Le second (`tous-gelés`) indique que toutes les lignes du bloc sont marquées comme étant gelées.

Catalogue (*Catalog*)

Le standard SQL utilise ce terme pour indiquer ce qui est appelé une *base de données* dans la terminologie PostgreSQL.

(Ne confondez pas ce terme avec les *catalogues systèmes*).

Pour plus d'informations, voir Section 23.1.

Catalogue système

Une collection de *tables* décrivant la structure de tous les *objets SQL* de l'instance. Le catalogue système réside dans le schéma `pg_catalog`. Ces tables contiennes les données dans une représentation interne et ne sont pas typiquement considérées utiles pour l'utilisateur ; un nombre de *vues* plus compréhensibles, présentes aussi dans le schéma `pg_catalog`, offrent un accès plus agréable à certaines de ces informations, alors que des tables et vues supplémentaires existent dans

	le schéma `information_schema` (voir Chapitre 37) pour exposer certaines des mêmes informations comme demandé par le *standard SQL*. Pour plus d'informations, voir Section 5.9.
Checkpoint	Un point dans la séquence *WAL* à partir de laquelle il est garanti que les fichiers de données des relations ont été mis à jour avec toutes les informations depuis la *mémoire partagée* modifiées avant ce checkpoint ; un *enregistrement checkpoint* est écrit et déversé dans le WAL pour marquer ce point. Un checkpoint est aussi le fait d'amener à bien toutes les actions qui sont nécessaires pour arriver au checkpoint comme décrit ci-dessus. Ce processus est initié quand des conditions prédéfinies sont rencontrées, comme quand un interval de temps défini est passé, ou parce qu'un certain volume d'enregistrements ont été écrits ; il peut aussi être invoqué manuellement par l'utilisateur avec la commande `CHECKPOINT`. Pour plus d'informations, voir Section 30.5.
Checkpointer (processus)	Un *processus auxiliaire* spécialisé responsable de l'exécution des *checkpoints*.
Classe (archaique)	Voir Relation.
Clé (*Key*)	Un moyen d'identifier un *enregistrement* dans une *table* ou une autre *relation* par valeur contenues dans un ou plusieurs *attributs* de cette relation.
Clé étrangère (*Foreign key*)	Un type de *contrainte* définie sur une ou plusieurs *colonnes* dans une *table* qui requiert la ou les valeur(s) dans ces *colonnes* pour identifier zéro ou un *enregistrement* dans une autre (ou, moins fréquemment, la même) *table*.
Clé primaire	Un cas particulier de *contrainte d'unicité* défini sur une *table* ou une autre *relation* qui garantit en plus que tous les *attributs* formant la *clé primaire* n'ont pas de valeurs *NULL*. Comme le nom l'indique, il ne peut y avoir qu'une seule clé primaire par table, bien qu'il soit possible d'avoir plusieurs contraintes d'unicité qui ont aussi des attributs non NULL.
Client (processus)	N'importe quel processus, potentiellement distant, qui établit une *session* en se *connectant* à une *instance* pour interagir avec une *base de données*.
Cohérence (*Consistency*)	La propriété que les données dans la *base de données* sont toujours conformes avec les *contraintes d'intégrité*. Des transactions peuvent être autorisées à enfreindre certaines contraintes temporairement avant qu'elle soit validée par un `COMMIT`, mais si ces infractions ne sont toujours pas résolues au moment de la validation, alors une telle transaction est automatiquement *annulée (ROLLBACK)*. Ceci est une des propriétés ACID.
Colonne	Un *attribut* trouvé dans une *table* ou une *vue*.
Commit	L'acte de finaliser une *transaction* dans la *base de données*, qui rend ses modifications visible aux autres transactions et assure leur *durabilité*.

Pour plus d'informations, voir COMMIT.

Concurrence (*Concurrency*)

Le concept de multiples opérations indépendantes se réalisant dans la *base de données* en même temps. Dans PostgreSQL, la concurrence est contrôlée par le mécanisme de *contrôle de concurrence multiversion*.

Connexion

Une ligne de communication établie entre un processus client et un processus *backend*, habituellement à travers le réseau, supportant une *session*. Ce terme est parfois utilisé comme synonyme de session.

Pour plus d'informations, voir Section 20.3.

Contrainte (*Constraint*)

Une restriction sur les valeurs des données autorisées dans une *table*, ou sur les attributs d'un *domain*.

Pour plus d'informations, voir Section 5.4.

Contrainte de validation (*Check constraint*)

Un type de *contrainte* défini sur une *relation* qui restreint les valeurs autorisées dans un ou plusieurs *attributs*. La contrainte de validation peut faire référence à n'importe quel attribut du même enregistrement dans la relation, mais ne peut pas référencer d'autres enregistrements de la même relation ou d'autres relations.

Pour plus d'informations, voir Section 5.4.

Contrainte d'unicité

Un type de *contrainte* définie sur une *relation* qui restreint les valeurs autorisées dans une colonne ou une combinaison de colonnes, pour que chaque valeur ou combinaison de valeurs puisse apparaître seulement une fois dans la relation -- c'est-à-dire qu'il n'existe aucune autre ligne dans la relation contenant des valeurs égales à cette ligne.

Comme les *valeurs NULL* ne sont pas considérées égales entre elles, plusieurs lignes peuvent avoir des valeurs NULL sans violer la contrainte d'unicité.

Merge

Une commande SQL utilisée pour ajouter, modifier ou supprimer sous condition des *lignes* dans une *table* donnée, en utilisant les données d'une *relation* source.

Pour plus d'informations, voir MERGE.

Contrôle des versions concurrentes (MVCC) (*Multi-version concurrency control (MVCC)*)

Un mécanisme désigné pour permettre à plusieurs *transactions* de lire et écrire les mêmes enregistrements sans qu'un processus fasse attendre les autres processus. Dans PostgreSQL, le MVCC est implémenté par création de copies (*versions*) des *lignes* dès qu'elles sont modifiées ; une fois que toutes les transactions qui voient les vieilles versions sont terminées, ces vieilles versions peuvent être supprimées.

Conversion (*Cast*)

Une conversion d'une *donnée* depuis son type actuel vers un autre type.

Pour plus d'informations, voir CREATE CAST.

Correspondance d'utilisateur (*User mapping*)

La traduction de l'utilisateur local à la *base* en un utilisateur sur le système distant défini par un *wrapper de données distantes*.

For more information, see CREATE USER MAPPING.

Delete

Une commande SQL qui supprime des *enregistrements* d'une *table* donnée ou d'une *relation*.

Pour plus d'informations, voir DELETE.

Domain

Un type de données défini par l'utilisateur qui est basé sur un autre type de données sous-jacent. Il agit de la même façon que le type sous-jacent sauf pour le restriction potentielle de l'ensemble de valeurs autorisées.

Pour plus d'informations, voir² Section 8.18.

Donnée (*Datum*)

La représentation interne d'une valeur dans un type de données SQL.

Durabilité (*Durability*)

L'assurance qu'une fois qu'une *transaction* a été *validée* par un ordre `COMMIT`, la modification est conservée même après une erreur système ou un arrêt brutal. C'est l'une des propriétés ACID.

Enregistrement (*Record*)

Voir Enregistrement (*Tuple*.

Enregistrement (*Tuple*

Une collection d'*attributs* dans un ordre fixe. Cet ordre peut être défini par la *table* (ou une autre relation *relation*) où l'enregistrement est contenu, auquel cas l'enregistrement est souvent appelé une *tuple* en anglais. Il pourrait aussi être défini par la structure d'un ensemble de résultats, auquel case il est souvent appelé *record* en anglais. En français, et particulièrement pour cette traduction, le terme habituellement utilisé est *ligne*.

Enregistrement Log (*Log record*)

Terme archaique pour un *enregistrement WAL*.

Enregistrement WAL (*WAL Record*)

Une description bas niveau d'un changement de données individuel. Il contient suffisamment d'informations sur le changement de données pour être ré-exécuté (*re-joué*) en cas d'un problème système qui aurait causer la perte de ce changement. Les enregistrements WAL utilisent un format binaire non affichable.

Pour plus d'informations, voir Section 30.6.

Ensemble de résultats (*Result set*)

Une *relation* transmise à partir d'un *processus serveur* vers un *client* suite à l'exécution réussie d'une commande SQL, habituellement un `SELECT` mais cela pourrait aussi être une commande `INSERT`, `UPDATE` ou `DELETE` si la clause `RETURNING` est spécifiée.

Le fait qu'un ensemble de résultats est une relation signifie qu'une requête peut être utilisée dans la définition d'une autre requête, devenant ainsi une *sous-requête*.

Epoque (*Epoch*)

Voir Identifiant de transaction (*Transaction ID*.

Extension

Un module complémentaire qui peut être installé sur une *instance* pour fournir des fonctionnalités supplémentaires.

Pour plus d'informations, voir Section 38.17.

Fichier journal, Fichier de trace (*Log file*)

Les fichiers journaux contiennent des lignes de texte sous forme lisible par un être humain sur les événements produits. Les exemples incluent les échecs de connexion, les requêtes longues, etc.

Pour plus d'informations, voir Section 25.3.

Fichier WAL

Aussi connu sous le nom de *segment WAL* ou de *fichier segment WAL*. Chacun des fichiers numérotés séquentiellement qui fournissent de l'espace de stockage pour les *WAL*. Les fichiers sont tous de la même taille prédéfinie et sont écrits dans l'ordre séquentiel, entrecroisant les changements quand ils surviennent dans plusieurs sessions simultanées. Si le système s'arrête brutalement, les fichiers sont lus dans l'ordre et chacun des changements est rejoué pour restaurer le système dans l'état qu'il avait avant le crash.

Chaque fichier WAL peut être relâcher une fois qu'un *checkpoint* a écrit toutes les modifications que le fichier contenait dans les fichiers de données correspondant. Relâcher un fichier se fait soit en le supprimant soit en modifiant son nom pour qu'il soit réutilisé dans le futur, ce qui est appelé *recycler*.

Pour plus d'informations, voir Section 30.6.

Fonction (routine) (*Function (routine)*)

Un type de routine qui recoit zéro ou plusieurs arguments, retourne zéro ou plusieurs valeurs, et sont contraintes de s'exécuter dans une transaction. Les fonctions sont invoquées comme une partie de la requête, par exemple via `SELECT`. Certaines fonctions peuvent retourner des *ensembles*, elles sont appelées *fonctions à retour d'ensemble*.

Les fonctions peuvent être utilisées par les *triggers*.

Pour plus d'informations, voir CREATE FUNCTION.

Fonction analytique (*Analytic function*)

Voir Fonctions (routine) de fenêtrage (*Window Function*).

Fonction d'agrégation (routine) (*aggregate function (routine)*)

Une *fonction* qui combine (*agrège*) de multiples valeurs en entrée, pour, par exemple, les compter, faire une moyenne ou une somme, produisant ainsi une seule valeur en sortie.

Pour plus d'informations, voir Section 9.21.

Voir aussi Fonctions (routine) de fenêtrage (*Window Function*).

Fonctions (routine) de fenêtrage (*Window Function*)

Un genre de *function* utilisé dans une *requête* qui s'applique à une *partition* de l'*ensemble de résultat* de la requête ; le résultat de la fonction est basé sur les valeurs trouvées dans les *lignes* de la même partition ou cadre.

Toutes les *fonctions d'agrégat* peuvent être utilisées comme fonctions de fenêtrage, mais les fonctions de fenêtrage peuvent aussi être utilisées pour donner des rangs à chaque ligne de la partition par exemple. Elles sont aussi connues sous le nom de *fonctions analytiques*.

Pour plus d'informations, voir Section 3.5.

Fork

Chacun des ensembles des fichiers séparés par segment dans lesquels une relation est stockée. Le *fork principal (main fork)* est le fichier où résident les données actuelles. Il existe aussi deux forks secondaires pour les métadonnées : la *carte des espaces*

	libres et la *carte de visibilité*. Les *relations non journalisées* ont aussi un *fork initial*.
Base Backup	Une copie binaire de tous les fichiers de l'*instance*. Elle est générée par l'outil pg_basebackup. Avec les fichiers WAL, elle peut être utilisée comme point de départ pour une restauration et pour une réplication en *log shipping*(envoi des journaux entiers) ou en *streaming*(flux).
Fragmentation (*Bloat*	L'espace dans les pages de données qui ne contient pas de versions vivantes des enregistrements, donc un espace inutilisé (libre).
GMT	Voir UTC.
Grant	Une commande SQL qui est utilisée pour permettre à un *utilisateur* ou à un *rôle* d'accéder à des objets spécifiques dans la *base de données*. Pour plus d'informations, voir GRANT.
Heap	Contient les valeurs des attributs des *enregistrements* (i.e. les données) pour une *relation*. La partie heap est structurée par un ou plusieurs *fichiers segments* dans le *fork principal* de la relation.
Hôte (*Host*)	Un ordinateur qui communique avec d'autres ordinateurs à travers un réseau. Ce terme est souvent synonyme de *serveur*. Il est aussi utilisé pour désigner l'ordinateur quand les *processus client* s'exécutent.
Identifiant de transaction (*Transaction ID*	L'identifiant numérique, unique, affecté en séquence que chaque transaction reçoit quand elle réalise sa première modification en base. Son abréviation fréquente est *xid*. Lorsqu'elle est stockée sur disque, les XID ont 32 bits, donc seuls approximativement 4 milliards d'identifiants de transaction en écriture peuvent être générés ; pour permettre au système de fonctionner plus longtemps que cela, les *epochs* sont utilisés, là-aussi sur 32 bits. Quand le compteur arrive à la valeur maximale d'un XID, il recommence à la valeur 3 (les valeurs inférieures sont réservées) et la valeur epoch est incrémentée de 1. Dans certains contextes, les valeurs epoch et XID sont considérées ensemble comme une seule valeur de 64 bits. Pour plus d'informations, voir Section 8.19.
Index (relation)	Une *relation* qui contient des données dérivées d'une *table* ou d'une *vue matérialisée*. Sa structure interne permet une récupération rapide et un accès aux données originelles. Pour plus d'informations, voir CREATE INDEX.
Insert	Une commande SQL utilisée pour ajouter de nouvelles données dans une *table*. Pour plus d'informations, voir INSERT.
Instance	Un groupe de processus *backend* et *processus auxiliaire* qui communiquent en utilisant une zone de mémoire partagée commune. Un *processus postmaster* gère l'instance ; une *instance* gère exactement toutes ses bases de données. Plusieurs instances peuvent s'exécuter sur le même *serveur* aussi longtemps que leur ports TCP n'entrent pas en conflit.

	L'instance supporte toutes les fonctionnalités clés d'un SGBD : accès en lecture et écriture aux fichiers et mémoire partagée, respect des propriétés ACID, *connexions* aux *processus client*, vérification des droits, restauration après panne, réplication, etc.
Instance (*Database cluster*)	Une collection de bases de données et d'objets SQL globaux, et leurs méta-données statiques et dynamiques communes. Parfois désigné sous le terme de *cluster*. Dans PostgreSQL, le terme *cluster* est aussi parfois utilisé pour désigner une instance. (Ne confondez pas ce terme avec la commande SQL `CLUSTER`.)
Intégrité référentielle	Une façon de restreindre les données dans une *relation* par une *clé étrangère* pour qu'il y ait uniquement des données correspondantes dans une autre *relation*.
Isolation	La propriété que les effets d'une transaction ne sont pas visibles par les *transactions concurrentes* avant qu'elle ne soit validée par un `COMMIT`. C'est une des propriétés ACID. Pour plus d'informations, voir Section 13.2.
Jointure (*Join*)	Une opération et un mot clé SQL (ce dernier pour le mot anglais évidemment) utilisés dans les *requêtes* pour combiner les données de multiples *relations*.
Journal de transaction (*Write-ahead log*)	Le journal qui conserve la trace des changements dans l'*instance du serveur* lorsque des opérations utilisateur et système surviennent. Il comprend de nombreux *enregistrements WAL* écrits séquentiellement dans les *fichiers WAL*.
Journalisé (*Logged*)	Une *table* est considérée comme *journalisée* si ces changements sont envoyés dans les *journaux de transactions (WAL)*. Par défaut, toutes les tables régulières sont journalisées. Une table peut être spécifiée comme *non journalisée* soit à sa création ou via la commande `ALTER TABLE`.
Ligne	Voir Enregistrement (*Tuple*.
Logger (processus) (*Logger (process)*)	Un *processus auxiliaire* Si activé, le processus *Logger* écrit des informations sur les événements de base de données dans le *fichier journal* courant. Au bout d'un certain temps ou après un certain volume, un nouveau fichier journal est créé. Aussi appelé *syslogger*. Pour plus d'informations, voir Section 20.8.
Maître (Serveur) (*Master (server)*)	Voir Primaire (serveur).
Matérialisée (*Materialized*)	La propriété que certaines informations ont été pré-calculées et stockées pour un usage ultérieur, plutôt que calculées à la volée. Ce terme est utilisé dans les *vues matérialisées*, pour signifier que les données dérivées depuis la requête de la vue sont stockées sur disque séparément de la source de ces données. Le terme peut aussi référer à des requêtes multi-étapes pour signifier que les données résultantes de l'exécution d'une étape donnée sont stockées en mémoire (avec possibilité d'être

déversées sur le disque), et ainsi peuvent être lues plusieurs fois par une autre étape.

Mémoire partagée

RAM utilisé par les processus d'une même *instance*. Il contient des parties des fichiers de la *base*, fournit un espace temporaire pour les *enregistrements WAL* et stocke des informations communes supplémentaires. Notez que la mémoire partagée appartient à l'instance complète, pas à une seule base de données.

La partie la plus importante de la mémoire partagée est connue sous le nom anglais de *shared buffers* et est utilisé pour être le miroir des fichiers de données, organisés en blocs. Quand un bloc est modifié, il est appelé *dirty* jusqu'à ce qu'il soit écrit sur le système de fichiers.

Pour plus d'informations, voir Section 20.4.1.

Non journalisé (*Unlogged*)

La propriété de certaines *relations* précisant que leur modification ne doit pas être reflété dans les *WAL*. Ceci désactive la réplication et la restauration en cas de crash pour ces relations.

L'intérêt principal des tables non journalisées est d'enregistrer des données temporaires de travail pouvant être partagées entre plusieurs processus.

Les *tables temporaires* sont toujours non journalisées.

Null

Un concept de non-existence qui est un principe central dans la théorie des bases de données relationnelles. Il représente l'absence d'une valeur définie.

Objet SQL

Tout objet créé avec une commande `CREATE`. La plupart des objets sont spécifiques à une base de données et sont communément appelés des *objets locaux*.

La plupart des objets locaux appartiennent à un *schéma* spécifique dans leur base, comme par exemple les *relations* (tous types), *routines* (tous types), les types de données, etc. Le nom des objets de même type dans le même schéma doit être unique.

Il existe aussi des objets locaux n'appartenant pas aux schémas ; pour exemple, les *extensions*, *conversions de type de données* et les *wrappers de données distantes*. Les noms des objets de même type doivent être uniques au sein de la même base de données.

Les autres types d'objets, tels que les *rôles*, *tablespaces*, origines de réplication, souscriptions pour la réplication logique et les bases elles-mêmes ne sont pas des objets locaux vu qu'ils existent en dehors d'une base spécifique ; ils sont appelés des *objets globaux*. Les noms de ces objets sont uniques au sein de l'instance.

Pour plus d'informations, voir Section 23.1.

Optimiseur (*Optimizer*)

Voir Planificateur de requête (*Query planner*).

Partition

Un ou plusieurs sous-ensembles disjoints (et non chevauchants) d'un plus large ensemble.

En référence aux *tables partitionnées* : une des tables qui contiennent chacune une partie des données de la table partitionnée, et qui est dite être le *parent*. La partition est elle-

même une table, elle peut ainsi être requêtée directement ; en même temps, une partition peut parfois être une table partitionnée, permettant ainsi de créer des hiérarchies.

En référence aux *fonctions de fenêtrage (window functions)* dans une *requête*, une partition est un critère défini par l'utilisateur qui identifie les *enregistrements* voisins de l'*ensemble de résultats d'une requête* qui sont pris en compte par la fonction.

Planificateur de requête (*Query planner*)

La partie de PostgreSQL qui est dédiée à la détermination (*planification*) de la façon la plus rapide pour exécuter des *requêtes*. Connu aussi sour le nom d'*optimiseur de requêtes*, *optimiseur* ou tout simplement *planificateur*.

Postmaster (processus)

Le tout premier processus d'une *instance*. Il démarre et gère les *processus auxiliaires* et crée les *processus backend* à la demande.

Pour plus d'informations, voir Section 19.3.

Primaire (serveur)

Quand deux ou plusieurs *bases* sont liées via la *réplication*, le *serveur* qui est considéré comme la source d'autorité des informations est appelé le *primaire*, aussi connu sous le nom de *maître*.

Procédure (routine)

Un type de routine. Leurs qualités distinctives sont qu'elles ne renvoient pas de valeurs, et qu'elles sont autorisées à utiliser des instructions transactionnelles comme `COMMIT` et `ROLLBACK`. Elles sont appelées via la commande `CALL`.

Pour plus d'informations, voir CREATE PROCEDURE.

Recyclage

Voir Fichier WAL.

Relation

Le terme générique pour tous les objets dans une *base* qui ont un nom et une liste d'*attributs* définis dans un ordre spécifique. Les *tables*, *séquences*, *vues*, *tables distantes*, *vues matérialisées*, types composites et *index* sont tous des relations.

Plus globalement, une relation est un ensemble de lignes. Par exemple, le résultat d'une requête est aussi une relation.

Dans PostgreSQL, *Class* est un synonyme historique pour *relation*.

Répertoire de données (*Data directory*)

Le répertoire de base sur le système de fichiers d'un *serveur* qui contient tous les fichiers de données et sous-répertoires associés avec une *instance* (à l'exception des *tablespaces* et optionnellement de *WAL*). La variable d'environnement `PGDATA` est communément utilisée pour se référer au répertoire de données.

L'espace de stockage d'une *instance* comprend le répertoire de données ainsi que tous les tablespaces supplémentaires.

Pour plus d'informations, voir Section 73.1.

Réplicat (serveur)

Une *base* qui est liée à une base *primaire* et qui maintient une copie de certaines ou toutes les données de la base primaire. Les raisons principales pour faire cela est de permettre un accès plus important aux données et pour maintenir la disponibilité des données dans le cas où le *primaire* devient indisponible.

Réplication

Le fait de reproduire des données d'un *serveur* sur un autre serveur appelé *réplica*. Cela peut prendre la forme d'une *réplication physique*, où toutes les modifications des fichiers d'un serveur sont copiées verbatim, ou la forme d'une *réplication logique* où un sous-ensemble défini de modifications de données est envoyé en utilisant une représentation de plus haut niveau.

Requête (*Query*)

Une requête envoyée par un client à un *processus serveur*, habituellement pour renvoyer des résultats ou pour modifier des données sur la base.

Requête parallèle (*Parallel query*)

La possibilité de gérer des parties d'exécution d'une *requête* pour tirer profit des processus parallèles sur des serveurs avec multiples CPUs.

Revoke

Une commande pour empêcher l'accès à un ensemble nommé d'objets de *base* pour une liste nommée de *rôles*.

Pour plus d'informations, voir REVOKE.

Rôle

Une collection de droits d'accès à l'*instance*. Les rôles sont eux-même un droit qui peut être donné à d'autres rôles. Ceci est fait fréquemment pour simplifier la gestion des droits quand plusieurs *users* ont besoin des mêmes droits.

Pour plus d'informations, voir CREATE ROLE.

Rollback

Une commande pour annuler toutes les opérations réalisées depuis le début d'une *transaction*.

Pour plus d'informations, voir ROLLBACK.

Routine

Un ensemble défini d'instructions enregistrées dans le système de bases de données, qui peut être appelé pour exécution. Une routine doit être écrite dans un des langages de programmation. Les routines peuvent être des *fonctions* (incluant les fonctions renvoyant des ensembles de lignes et les *fonctions trigger*), des *fonctions d'agrégat*, et des *procédures*.

Plusieurs routines sont déjà définies dans PostgreSQL mais des routines définies par les utilisateurs sont ajoutables.

Savepoint

Un marquage spécial dans une séquence d'étapes d'une *transaction*. Les modifications de données après ce point dans le temps peuvent être annulées jusqu'au moment du savepoint.

Pour plus d'informations, voir SAVEPOINT.

Schéma

Un schéma est un espace de nom pour les *objets SQL*, qui appartiennent tous à la même *base*. Chaque objet SQL doit résider dans exactement un schéma.

Tous les objets SQL définis par le système résident dans le schéma `pg_catalog`.

Plus généralement, le terme *schéma* est utilisé pour signifier toutes les descriptions de données (définitions des *tables*, *contraintes*, commentaires, etc) pour une *base* donnée ou pour un sous-ensemble.

Pour plus d'informations, voir Section 5.9.

Segment	Voir Segment de fichier (*File segment*).
Segment de fichier (*File segment*)	Un fichier physique qui stocke des données pour une *relation* donnée. Les segments de fichiers sont limités en taille par une valeur de configuration (typiquement 1 gigaoctet). Ainsi, si une relation excède cette taille, le fichier est séparée dans de multiples segments. Pour plus d'informations, voir Section 73.1. (Ne confondez pas ce terme avec le terme similaire de *segment WAL*).
Segment WAL	Voir Fichier WAL.
Select	La commande SQL utilisée pour réclamer des données d'une *base*. Normalement, les commandes `SELECT` ne modifient pas la *base de données* de quelque façon que ce soit. Cependant, il est possible que des *fonctions* appelées dans la requête aient comme effet de bord de modifier des données. Pour plus d'informations, voir SELECT.
WAL receiver (process)	Un *processus auxiliaire* qui s'exécute sur un *replica* pour recevoir les en provenance d'un *serveur primaire* pour le rejeu par le *startup process*. Pour plus d'informations, voir Section 27.2.
WAL sender (process)	Un *processus backend* spécial qui envoie en flux les WAL sur un réseau. Le bout de réception peut être un *WAL receiver* sur un *réplica*, pg_receivewal. Il peut être aussi tout type de processs qui par le protocole de réplication. that speaks the replication protocol.
Séquence (relation)	Un type de relation utilisé pour générer des valeurs. Typiquement, les valeurs générées sont des nombres séquentiels qui ne se répètent pas. Elles sont habituellement utilisées pour générer des valeurs pour des *clés primaires* de substitut.
Serveur	Un ordinateur sur lequel des *instances* PostgreSQL sont exécutées. Le terme *serveur* dénote le matériel, un containeur ou une *machine virtuelle*. Ce terme est parfois utilisé pour faire référence à une instance ou à un hôte.
Serveur de base de données (*Database server*)	Voir Instance.
Serveur distant (*Foreign server*)	Une collection nommée de *tables distantes* qui utilisent toute le même *wrapper de données distantes* et ont d'autres valeurs de configuration en commun. Pour plus d'informations, voir CREATE SERVER.
Session	Un état qui permet à un client et un processus serveur d'interagir, en communiquant au travers d'une *connexion*.
Standard SQL	Une série de documents qui définit le langage SQL.
Standby (serveur)	Voir Réplicat (serveur).

Startup process	Un *processus auxiliaire* qui rejoue les WAL lors d'un redémarrage après crash et dans un *réplica physique*. (Le nom est historique : le processus de démarrage a été nommé avant la réplication. Le nom fait référence à ce tâche car il est en relation avec l'amorçage du serveur après un crash.
Système de statistiques cumulatives	Un système qui, une fois activé, accumule des informations statistiques sur les activités de l'*instance*. Pour plus d'informations, voir Section 28.2.
Table	Une collection d'*enregistrements* ayant une structure de données communes (même nombre d'*attributs*, dans le même ordre, ayant le même nom et le même type par position). Une table est la forme la plus commune de *relation* dans PostgreSQL. Pour plus d'informations, voir CREATE TABLE.
Table distante (relation) (*Foreign table (relation)*)	Une *relation* qui apparaît avoir des *enregistrements* et *colonnes* similaires à une *table* régulière, mais qui transfèrent les requêtes de données à travers son *wrapper de données distantes*, qui retournera les *ensembles de résultats* structurés selon la définition de la *table distante*. Pour plus d'informations, voir CREATE FOREIGN TABLE.
Table partitionnée (relation)	Une *relation* qui est sémantiquement identique à une *table*, mais dont le stockage est distribué sur plusieurs *partitions*.
Table temporaire	*Tables* qui existent soit pour la durée d'une *session* soit pour la durée d'une *transaction*, comme indiqué au moment de leur création. Les données de ces tables ne sont pas visibles des autres sessions et ne sont pas *journalisées*. Les tables temporaires sont souvent utilisées pour enregistrer des données intermédiaires pour une opération en plusieurs étapes. Pour plus d'informations, voir CREATE TABLE.
Tablespace	Un emplacement nommé sur le système de fichiers du serveur. Tous les *objets SQL* nécessitant du stockage en dehors de leur définition dans le *catalogue système* doivent appartenir à un seul tablespace. Au départ, l'instance contient un seul tablespace utilisable et qui est donc le tablespace par défaut pour tous les objets SQL. Ce tablespace est appelé `pg_default`. Pour plus d'informations, voir Section 23.6.
TOAST	Un mécanisme permettant de diviser les valeurs volumineuses des colonnes d'une ligne d'une table et de les enregistrer dans une table secondaire, appelée *table TOAST*. Chaque relation comprenant des colonnes à taille variable (et donc potentiellement volumineuse) a sa propre table TOAST. Pour plus d'informations, voir Section 73.2.
Transaction	Une combinaison de commandes qui agissent comme une seule commande *atomique* : elles réussissent toutes ou elles échouent toutes, comme une seule unité, et leurs effets ne sont pas visibles pour les autres *sessions* tant que la transaction n'est pas terminée, et même potentiellement après, suivant le niveau d'isolation.

	Pour plus d'informations, voir Section 13.2.
Transactions par seconde (TPS)	Nombre moyen de transactions exécutées par seconde, totalisées sur toutes les sessions actives pendant un certain temps. C'est utilisé comme une mesure des performances d'une instance.
Trigger	Une *fonction* qui peut être définie pour s'exécuter quand une certaine opération (`INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`) est appliquée à une *relation*. Un trigger s'exécute dans la même *transaction* que la requête qui l'a appelé et, si la fonction échoue, alors la requête appelante échoue aussi. Pour plus d'informations, voir CREATE TRIGGER.
Update	Une commande SQL utilisée pour modifier les *lignes* qui existeraient déjà dane une *table* spécifiée. Elle ne peut ni créer ni supprimer des lignes. Pour plus d'informations, voir UPDATE.
Utilisateur (*User*)	Un *role* doté de l'attribut `LOGIN`.
UTC	Universal Coordinated Time, la référence de temps principal, approximativement l'heure prévalant au méridien de longitude zéro. Souvent mais à tort appelé GMT.
Vacuum	Le processus permettant de supprimer les *versions de lignes* obsolètes des tables ou vues matérialisées, ainsi que d'autres processus en relation proche par l'implémentation PostgreSQL de *MVCC*. Ce processus est déclenchable en utilisant la commande `VACUUM`, mais peut aussi être géré automatiquement via les processus *autovacuum*. Pour plus d'informations, Section 25.1 .
Verrou (*Lock*)	Un mécanisme qui assure à un processus de limiter ou prévenir des accès simultanés à une ressource.
Vue	Une *relation* qui est définie par la requête `SELECT`, mais n'a pas de stockage propre. À chaque fois qu'une requête référence une vue, la définition de la vue est substituée dans la requête comme si l'utilisateur l'avait saisi comme une sous-requête à la place du nom de la vue. Pour plus d'informations, voir CREATE VIEW.
Vue matérialisée (relation) (*Materialized view (relation)*)	Une *relation* qui est définie par une expression `SELECT` (juste comme une *vue*), mais stocke les données de la même façon qu'une *table* le fait. Elle ne peut être modifiée via les opérations `INSERT`, `UPDATE`, `DELETE` ou `MERGE`. Pour plus d'informations, voir CREATE MATERIALIZED VIEW.
WAL	Voir Journal de transaction (*Write-ahead log*).
WAL archiver (processus)	Un *processus auxiliaire* qui, une fois activé, sauvegarde des copies des *fichiers WAL* dans le but de créer des sauvegardes ou conserver le *réplicas* actuel. Pour plus d'informations, voir Section 26.3.

WAL writer (processus)	Un *processus auxiliaire* qui écrit les *enregistrements WAL* de la *mémoire partagée* vers les *fichiers WAL*. Pour plus d'informations, voir Section 20.5.
Wrapper de données distantes (*Foreign data wrapper*)	Un moyen de représenter des données qui ne sont pas contenues sur la *base de données* locale de sorte qu'elles apparaissent comme si elles étaient locales. Avec un wrapper de données distantes, il est possible de définir un *serveur distant* et des *tables distantes*. Pour plus d'informations, voir CREATE FOREIGN DATA WRAPPER.
Zone de données (*Data area*)	Voir Répertoire de données (*Data directory*).

Annexe N. Support des couleurs

Les programmes dans la distribution PostgreSQL peuvent produire une sortie console colorée. Cette annexe décrit comment le configurer.

N.1. Quand les couleurs sont utilisées

Pour utiliser la sortie colorée, affecter la variable d'environnement `PG_COLOR` comme suit :

1. Si la valeur est `always`, alors les couleurs sont utilisées.
2. Si la valeur est `auto` et que le flux de sortie standard d'erreur est associé à un périphérique terminal, alors la couleur est utilisée.
3. Autrement, la couleur n'est pas utilisée.

N.2. Configurer les couleurs

Les couleurs actuellement utilisées sont configurées en utilisant la variable d'environnement `PG_COLORS` (notez le pluriel). Sa valeur est une liste de *`clé=valeur`* séparés par des virgules. Les clés spécifient la couleur utilisée à tel usage. Les valeurs suivent les spécifications SGR (Select Graphic Rendition), qui sont interprétées par le terminal.

Les clés suivantes sont actuellement en usage :

`error`

Utilisé pour surligner le texte « error » dans les messages d'erreur.

`warning`

Utilisé pour surligner le texte « warning » dans les messages d'avertissement.

`note`

Utilisé pour surligner les textes « detail » et « hint » dans ces messages

`locus`

Utilisé pour surligner l'information d'emplacement (i.e. le nom du programme et le nom de fichier) dans les messages.

La valeur par défaut est `error=01;31:warning=01;35:note=01;36:locus=01` (`01;31` = rouge en gras, `01;35` = magenta en gras, `01;36` = cyan en gras, `01` = couleur par défaut en gras).

> **Astuce**
>
> Le format des spécifications couleurs est aussi utilisé par d'autres paquets de programmes comme GCC, GNU coreutils et GNU grep.

Annexe O. Fonctionnalités obsolètes ou renommées

Certaines fonctionnalités sont parfois supprimées de PostgreSQL. D'autres fonctionnalités, noms de paramètres, noms de fonctions peuvent changer. La documentation peut changer d'emplacement. Cette section dirige les utilisateurs provenant de vieilles versions de la documentation ou de liens externes vers le nouvel emplacement approprié pour l'information qu'ils recherchent.

O.1. fichier `recovery.conf` intégré à `postgresql.conf`

PostgreSQL 11 et les versions antérieures utilisaient un fichier de configuration nommé `recovery.conf` pour gérer les réplicas et standbys. Le support de ce fichier a été supprimé dans PostgreSQL 12. Voir les notes de version de PostgreSQL 12 pour les détails sur ce changement.

À partir de PostgreSQL 12, la restauration d'archive, la réplication en flux et le PITR sont configurés en utilisant les paramètres de configuration normaux de serveur. Ces paramètres sont configurés dans le fichier `postgresql.conf` ou via ALTER SYSTEM comme tout autre paramètre.

Le serveur ne démarrera pas si un fichier `recovery.conf` existe.

Le paramètre `trigger_file` a été renommé en promote_trigger_file.

Le paramètre `standby_mode` a été supprimé. Un fichier `standby.signal` est utilisé à la place dans le répertoire des données. Voir Fonctionnement du Serveur de Standby pour les détails.

O.2. Les rôles par défaut sont renommés en rôles prédéfinis

PostgreSQL 13 et les versions précédentes utilisent le terme « rôle par défaut ». Cependant, comme ces rôles ne peuvent pas être changés et sont installés comme faisant partie du système à l'initialisation, le terme le plus approprié à utiliser est « rôles prédéfinis ». Voir Section 22.5 pour la documentation actuelle à propos des rôles prédéfinis, et les notes de version pour PostgreSQL 14 pour plus de détails sur ce changement.

O.3. `pg_xlogdump` renommé en `pg_waldump`

PostgreSQL 9.6 et les versions antérieures fournissaient une commande nommée `pg_xlogdump` pour lire les fichiers des journaux de transactions (WAL). Cette commande a été renommée en `pg_waldump`, voir pg_waldump pour la documentation de `pg_resetwal` et voir les notes de version de PostgreSQL 10 pour les détails sur ce changement.

O.4. `pg_resetxlog` renommé en `pg_resetwal`

PostgreSQL 9.6 et les versions antérieures fournissaient une commande nommée `pg_resetxlog` pour réinitialiser les fichiers des journaux de transactions (WAL). Cette commande a été renommée en `pg_resetwal`, voir pg_resetwal pour la documentation de `pg_resetwal` et voir les notes de version de PostgreSQL 10 pour les détails sur ce changement.

O.5. `pg_receivexlog` renommé en `pg_receivewal`

PostgreSQL 9.6 et les versions antérieures fournissaient une commande nommée `pg_receivexlog` pour récupérer les fichiers des journaux de transactions (WAL). Cette commande a été renommée en `pg_receivewal`, voir pg_receivewal pour la documentation de `pg_receivewal` et voir les notes de version de PostgreSQL 10 pour les détails sur ce changement.

Annexe P. Traduction française

Ce manuel est une traduction de la version originale. Elle est réalisée uniquement par des volontaires bénévolement. La dernière version à jour est disponible sur le site docs.postgresql.fr[1]. Ce site est hébergé par l'association PostgreSQLFr[2], dont le but est la promotion du logiciel PostgreSQL. Si cette documentation vous a plu, adhérez à l'association[3] et, si vous en avez la possibilité, contribuez.

Cette annexe supplémentaire présente la liste des contributeurs, et la procédure pour remonter des informations aux traducteurs.

Voici les contributeurs par ordre alphabétique (prénom puis nom) :

- Antoine
- Bruno Levêque
- Cédric Duprez
- Christophe 'nah-ko' Truffier
- Christophe Bredel
- Christophe Courtois
- Christophe 'KrysKool' Chauvet
- Claude Castello
- Claude Thomassin
- Damien Clochard
- Daniel Vérité
- David Bidoc
- Denis Laxalde
- Emmanuel Magin
- Emmanuel Seyman
- Erwan Duroselle
- Fabien Foglia
- Fabien Grumelard
- Flavie Perette
- Florence Cousin
- Franck Boudehen
- François Suter
- Frédéric Andres

[1] https://docs.postgresql.fr/
[2] https://www.postgresql.fr/
[3] http://www.postgresqlfr.org/adherents:adhesion

- Guillaume 'gleu' Lelarge
- Hervé Dumont
- Jacques Massé
- Jean-Christophe 'jca' Arnu
- Jean-Christophe Weis
- Jean-Max Reymond
- Jean-Michel Poure
- Jean-Paul Argudo
- Jehan-Guillaume 'ioguix' de Rorthais
- Jérôme Seyler
- Julien Moquet
- Julien 'rjuju' Rouhaud
- Lætitia Avrot
- Marc Blanc
- Marc Cousin
- Mathieu Lafage
- Matthieu Clavier
- Nicolas Coudert
- Nicolas Gollet
- Philippe Rimbault
- Pierre Jarillon
- Robin Portigliatti
- Rodolphe 'rodo' Quiedville
- Stephan Fercot
- Stéphane Lorek
- Stéphane 'SAS' Schildknecht
- Thom Brown
- Thomas Reiss
- Thomas Silvi
- Vincent Picavet
- Yves Darmaillac

Tous ont participé en traduisant, en relisant ou en rédigeant un rapport de bogue. N'hésitez pas à nous signaler toute personne qui aurait été oubliée.

De même que tout logiciel peut contenir des erreurs de programmation, toute traduction n'est pas exempte d'erreurs : faute d'orthographe, faute de grammaire, erreur de saisie, voire, bien pire, contresens. Bien que l'équipe de traduction passe beaucoup de temps à relire la documentation traduite, il lui arrive de laisser passer des erreurs. Elle a donc besoin de vous.

Si vous découvrez une erreur ou si un passage n'est pas compréhensible, l'équipe de traduction souhaite le savoir. Pour cela, vous pouvez envoyer un mail au coordinateur (Guillaume Lelarge[4]). C'est le moyen le plus simple et le plus sûr. Tout problème remonté sera pris en considération.

[4] mailto:guillaume@lelarge.info

Bibliographie

Références sélectionnées et lectures autour du SQL et de PostgreSQL.

Quelques livres blancs et rapports techniques réalisés par l'équipe d'origine de développement de POSTGRES sont disponibles sur le site web[1] du département des sciences informatiques de l'université de Californie.

Livres de référence sur SQL

[bowman01] *The Practical SQL Handbook*. Using SQL Variants. Quatrième édition. Judith Bowman, Sandra Emerson, et Marcy Darnovsky. ISBN 0-201-70309-2. Addison-Wesley Professional. 2001.

[date97] *A Guide to the SQL Standard*. A user's guide to the standard database language SQL. Quatrième édition. C. J. Date et Hugh Darwen. ISBN 0-201-96426-0. Addison-Wesley. 1997.

[date04] *An Introduction to Database Systems*. Huitième édition. C. J. Date. ISBN 0-321-19784-4. Addison-Wesley. 2003.

[elma04] *Fundamentals of Database Systems*. Quatrième édition. Ramez Elmasri et Shamkant Navathe. ISBN 0-321-12226-7. Addison-Wesley. 2003.

[melt93] *Understanding the New SQL*. A complete guide. Jim Melton et Alan R. Simon. ISBN 1-55860-245-3. Morgan Kaufmann. 1993.

[ull88] *Principles of Database and Knowledge-Base Systems*. Classical Database Systems. Jeffrey D. Ullman. Volume 1. Computer Science Press. 1988.

[sqltr-19075-6] *SQL Technical Report*. Part 6: SQL support for JavaScript Object Notation (JSON). Première édition. 2017.

Documentation spécifique sur PostgreSQL

[sim98] *Enhancement of the ANSI SQL Implementation of PostgreSQL*. Stefan Simkovics. Département des systèmes d'informations, université de technologie de Vienne. Vienne, Autriche. 29 novembre 1998.

[yu95] *The Postgres95. User Manual*. A. Yu et J. Chen. Université de Californie. Berkeley, Californie. 5 septembre 1995.

[fong] *The design and implementation of the POSTGRES query optimizer*[2]. Zelaine Fong. Université de Californie, Berkeley, département des sciences informatiques.

Procédures et articles

[ports12] « Serializable Snapshot Isolation in PostgreSQL[3] ». D. Ports et K. Grittner. VLDB Conference, Août 2012.

[berenson95] « A Critique of ANSI SQL Isolation Levels[4] ». H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, et P. O'Neil. ACM-SIGMOD Conference on Management of Data, Juin 1995.

[olson93] *Partial indexing in POSTGRES: research project*. Nels Olson. UCB Engin T7.49.1993 O676. Université de Californie. Berkeley, Californie. 1993.

[1] http://dsf.berkeley.edu/papers/
[2] https://dsf.berkeley.edu/papers/UCB-MS-zfong.pdf
[3] https://arxiv.org/pdf/1208.4179
[4] https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf

[ong90] « A Unified Framework for Version Modeling Using Production Rules in a Database System ». L. Ong et J. Goh. *ERL Technical Memorandum M90/33*. Université de Californie. Berkeley, Californie. Avril 1990.

[rowe87] « The POSTGRES data model[5] ». Rowe and Stonebraker, 1987. L. Rowe et M. Stonebraker. VLDB Conference, Septembre 1987.

[seshadri95] « Generalized Partial Indexes[6] ». P. Seshadri et A. Swami. Eleventh International Conference on Data Engineering, 6–10 mars 1995. Cat. No.95CH35724. IEEE Computer Society Press. Los Alamitos, Californie. 1995. 420–7.

[ston86] « The design of POSTGRES[7] ». M. Stonebraker et L. Rowe. ACM-SIGMOD Conference on Management of Data, Mai 1986.

[ston87a] « The design of the POSTGRES. rules system ». M. Stonebraker, E. Hanson, et C. H. Hong. IEEE Conference on Data Engineering, Février 1987.

[ston87b] « The design of the POSTGRES storage system[8] ». M. Stonebraker. VLDB Conference, Septembre 1987.

[ston89] « A commentary on the POSTGRES rules system[9] ». M. Stonebraker, M. Hearst, et S. Potamianos. *SIGMOD Record 18(3)*. Septembre 1989.

[ston89b] « The case for partial indexes[10] ». M. Stonebraker. *SIGMOD Record 18(4)*. Décembre 1989. 4–11.

[ston90a] « The implementation of POSTGRES[11] ». M. Stonebraker, L. A. Rowe, et M. Hirohama. *Transactions on Knowledge and Data Engineering 2(1)*. IEEE. Mars 1990.

[ston90b] « On Rules, Procedures, Caching and Views in Database Systems[12] ». M. Stonebraker, A. Jhingran, J. Goh, et S. Potamianos. ACM-SIGMOD Conference on Management of Data, Juin 1990.

[ston92] « Un aperçu du projet Sequoia 2000 [13] ». M. Stonebraker. *Résumé des articles COMPCON printemps 1992*. 1992. 383–388.

[5] https://dsf.berkeley.edu/papers/ERL-M87-13.pdf
[6] https://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.5740
[7] http://dsf.berkeley.edu/papers/ERL-M85-95.pdf
[8] http://dsf.berkeley.edu/papers/ERL-M87-06.pdf
[9] http://dsf.berkeley.edu/papers/ERL-M89-82.pdf
[10] http://dsf.berkeley.edu/papers/ERL-M89-17.pdf
[11] http://dsf.berkeley.edu/papers/ERL-M90-34.pdf
[12] http://dsf.berkeley.edu/papers/ERL-M90-36.pdf
[13] https://dsf.berkeley.edu/papers/S2K-91-05.pdf

Index

Symboles

$, 43
$libdir, 1251
$libdir/plugins, 697, 2021
*, 134
.pgpass, 1016
.pg_service.conf, 1017
::, 50
_PG_archive_module_init, 1580
_PG_init, 1250
_PG_output_plugin_init, 1567

A

abbrev, 314
ABORT, 1591
abs, 235
ACL, 77
aclcontains, 394
acldefault, 394
aclexplode, 394
aclitem, 81
aclitemeq, 394
acos, 238
acosd, 239
acosh, 240
activité de la base de données
  superviser, 827
actualisation, 114
adminpack, 2867
adresse MAC (voir macaddr)
adresse MAC (format EUI-64) (voir macaddr)
age, 289, 289
agrégat à ensemble trié
  natif, 376
agrégat avec ensemble trié, 45
AIX
  configuration IPC, 601
  installation sur, 581
akeys, 2935
alias
  dans la clause FROM, 122
  dans la liste de sélection, 134
  pour le nom d'une table dans une requête, 12
ALL, 380, 383
  GROUP BY ALL, 130
  SELECT ALL, 135
ALTER AGGREGATE, 1592
ALTER COLLATION, 1594
ALTER CONVERSION, 1597
ALTER DATABASE, 1599
ALTER DEFAULT PRIVILEGES, 1602
ALTER DOMAIN, 1606
ALTER EVENT TRIGGER, 1610
ALTER EXTENSION, 1611
ALTER FOREIGN DATA WRAPPER, 1615
ALTER FOREIGN TABLE, 1617
ALTER FUNCTION, 1622
ALTER GROUP, 1626
ALTER INDEX, 1628
ALTER LANGUAGE, 1631
ALTER LARGE OBJECT, 1632
ALTER MATERIALIZED VIEW, 1633
ALTER OPERATOR, 1635
ALTER OPERATOR CLASS, 1637
ALTER OPERATOR FAMILY, 1638
ALTER POLICY, 1642
ALTER PROCEDURE, 1644
ALTER PUBLICATION, 1647
ALTER ROLE, 742, 1650
ALTER ROUTINE, 1654
ALTER RULE, 1656
ALTER SCHEMA, 1657
ALTER SEQUENCE, 1658
ALTER SERVER, 1662
ALTER STATISTICS, 1664
ALTER SUBSCRIPTION, 1666
ALTER SYSTEM, 1669
ALTER TABLE, 1671
ALTER TABLESPACE, 1690
ALTER TEXT SEARCH CONFIGURATION, 1692
ALTER TEXT SEARCH DICTIONARY, 1694
ALTER TEXT SEARCH PARSER, 1696
ALTER TEXT SEARCH TEMPLATE, 1697
ALTER TRIGGER, 1698
ALTER TYPE, 1700
ALTER USER, 1705
ALTER USER MAPPING, 1706
ALTER VIEW, 1708
amcheck, 2868
ANALYZE, 776, 1711
AND (opérateur), 228
annulation
  commande SQL, 992
anomalie de sérialisation, 508, 511
any, 226
ANY, 374, 380, 383
anyarray, 226
anycompatible, 226
anycompatiblearray, 226
anycompatiblemultirange, 226
anycompatiblenonarray, 226
anycompatiblerange, 226
anyelement, 226
anyenum, 226
anymultirange, 226
anynonarray, 226
anyrange, 226
applicable role, 1167
arbre de requêtes, 1338
Archivage continu
  côté secondaire, 817
archivage en continu, 787

area, 309
armor, 2977
ARRAY, 52
  détermination du type de résultat, 441
array_agg, 371, 2941
array_append, 362
array_cat, 363
array_dims, 363
array_fill, 363
array_length, 363
array_lower, 363
array_ndims, 363
array_position, 363
array_positions, 363
array_prepend, 364
array_remove, 364
array_replace, 364
array_to_json, 341
array_to_string, 364
array_to_tsvector, 317
array_upper, 364
arrêt, 608
ascii, 243
asin, 239
asind, 239
asinh, 240
ASSERT
  en PL/pgSQL, 1416
assertions
  en PL/pgSQL, 1416
AT TIME ZONE, 301
atan, 239
atan2, 239
atan2d, 239
atand, 239
atanh, 240
Authentification BSD, 737
authentification client, 715
  délai lors de, 630
auth_delay, 2873
auto-increment (voir serial)
autocommit
  gros chargement de données, 543
  psql, 2270
autovacuum
  information générale, 783
  paramètres de configuration, 688
autovacuum_analyze_scale_factor
  paramètre de configuration, 690
  paramètre de stockage, 1869
autovacuum_analyze_threshold
  paramètre de configuration, 689
  paramètre de stockage, 1869
autovacuum_freeze_max_age
  paramètre de configuration, 690
  paramètre de stockage, 1869
autovacuum_max_workers
  paramètre de configuration, 689
autovacuum_multixact_freeze_max_age
  paramètre de configuration, 690
  paramètre de stockage, 1870
autovacuum_naptime
  paramètre de configuration, 689
autovacuum_vacuum_cost_delay
  paramètre de configuration, 690
  paramètre de stockage, 1869
autovacuum_vacuum_cost_limit
  paramètre de configuration, 690
  paramètre de stockage, 1869
autovacuum_vacuum_insert_scale_factor
  paramètre de configuration, 689
  paramètre de stockage, 1869
autovacuum_vacuum_insert_threshold
  paramètre de configuration, 689
  paramètre de stockage, 1869
autovacuum_vacuum_scale_factor
  paramètre de configuration, 689
  paramètre de stockage, 1869
autovacuum_vacuum_threshold
  paramètre de configuration, 689
  paramètre de stockage, 1869
auto_explain, 2874
auto_explain.log_level configuration parameter, 2876
auto_explain.log_settings configuration parameter, 2875
avals, 2935
average, 371
avg, 371

B

B-Tree (voir index)
Background workers, 1557
backup, 787
base de données, 747
  création, 4
  droit de création, 740
base de données hiérarchique, 7
base de données orientée objets, 7
base de données relationnelle, 7
basebackup_to_shell, 2877
BASE_BACKUP, 2468
basic_archive, 2877
BEGIN, 1714
BETWEEN, 231
BETWEEN SYMMETRIC, 231
BGWORKER_BACKEND_DATABASE_CONNECTION, 1558
BGWORKER_SHMEM_ACCESS, 1558
bibliothèque partagée, 1258
bibliothèques partagées, 579
bigint, 39, 151
bigserial, 156
binary data, 160
binary string
  concaténation, 251
  length, 253

bison, 564
bit_and, 372
bit_count, 252, 256
bit_length, 241, 252, 256
bit_or, 372
bit_xor, 372
BLOB (voir Large Object)
bloc magique, 1250
blocage de verrous, 518
blocs de code anonymes, 1931
bloom, 2878
boîte (type de données), 177
boolean
  type de données, 172
Booléen
  opérateurs (voir opérateurs, logique)
bool_and, 372
bool_or, 372
boucle
  en PL/pgSQL, 1397
bound_box, 310
box, 310
BRIN (voir index)
brin_desummarize_range, 421
brin_metapage_info, 2964
brin_page_items, 2965
brin_page_type, 2964
brin_revmap_data, 2964
brin_summarize_new_values, 421
brin_summarize_range, 421
broadcast, 314
btree_gin, 2881
btree_gist, 2882
btrim, 243, 253
bt_index_check, 2868
bt_index_parent_check, 2869
bt_metap, 2961
bt_page_items, 2962, 2963
bt_page_stats, 2962
bytea, 160

C

C, 941, 1049
C++, 1271
calendrier Grégorien, 2705
CALL, 1716
cardinality, 364
Carte de visibilité, 2655
CASCADE
  action clé étrangère, 70
  with DROP, 111
CASE, 358
  détermination du type de résultat, 441
cast
  conversion I/O, 1750
catalogue système
  schéma, 91
cbrt, 235
ceil, 235
ceiling, 235
center, 309
Certificat, 736
Chaîne (voir Chaîne de caractères)
chaîne binaire
  convertir en une chaîne de caractères, 254
chaîne de bit
  constante, 38
  longueur, 257
chaîne de bits
  type de données, 181
chaîne de caractères
  concaténation, 241
  constante, 35
  convertir en une chaîne binaire, 254
  longueur, 241
  test de préfixe, 243
Chaîne de caractères
  types de données, 158
chaînes d'échappement
  dans libpq, 979
chaînes de bit
  fonctions, 256
champ
  calculé, 214
champ calculé, 214
char, 158
character, 158
character set, 700
character varying, 158
character_length, 241
chargement dynamique, 701
char_length, 241
CHECK OPTION, 1915
CHECKPOINT, 1718
chemin de recherche, 89
  visibilité de l'objet, 395
chemin rapide, 993
chiffrement, 613
  pour des colonnes spécifiques, 2972
chiffres significatifs, 700
chr, 243
cid, 223
cidr, 179
circle, 178, 310
citext, 2883
classe d'opérateurs, 457
clause OVER, 48
clé étrangère, 68
  auto-référencé, 69
  foreign key, 16
clé primaire, 68
clock_timestamp, 290
CLOSE, 1719
CLUSTER, 1720
clusterdb, 2125
clustering, 803

cmax, 73
cmin, 73
COALESCE, 360
codes d'erreur
  liste de, 2690
COLLATE, 51
collation, 756
  dans les fonctions SQL, 1246
COLLATION FOR, 399
collationnement
  dans PL/pgSQL, 1380
colonne, 7, 60
  ajouter, 74
  colonne système, 73
  renommer, 76
  supprimer, 74
colonne générée, 62, 1777, 1863
  dans les triggers, 1320
col_description, 403
commande TABLE, 2066
COMMENT, 1723
commentaire
  sur les objets de la base, 403
  en SQL, 41
COMMIT, 1728
COMMIT PREPARED, 1729
common table expression (voir WITH)
comparaison
  constructeur de ligne, 383
  ligne résultat de la sous-requête, 380
  opérateurs, 229
  type composite, 383
comparaison de ligne entière, 383
compilation
  applications libpq, 1024
Compilation à la volée (voir JIT)
compilation croisée, 573, 573
concat, 243
concaténation de tsvector, 476
concat_ws, 244
concurrence, 507
configuration
  de la restauration
    configuration générale, 652
    d'un serveur standby, 652
  du serveur, 621
  du serveur
    fonctions, 408
configure, 564
configure (options), 567
configure (variables d'environnement), 576
conjunction, 228
connectby, 3037, 3044
connexion non bloquante, 943, 982
conninfo, 949
constante, 35
constraint exclusion, 670
contexte mémoire
  dans SPI, 1540
CONTINUE
  in PL/pgSQL, 1398
contrainte, 63
  ajouter, 75
  clé étrangère, 69
  clé primaire, 68
  exclusion, 72
  nom, 64
  NOT NULL, 66
  supprimer, 76
  unicité, 66
  vérification, 63
contrainte d'exclusion, 72
Contrainte d'exclusion, 108
contrainte d'unicité, 66
contrainte de vérification, 63
contrainte NOT NULL, 66
conversion de type, 39, 50
convert, 254
convert_from, 254
convert_to, 255
COPY, 9, 1730
  avec libpq, 995
corr, 374
correlation, 374
corrélation
  dans l'optimiseur de requêtes, 538
Correspondance d'identités, 724
correspondance d'utilisateur, 110
Correspondance de motif, 257
cos, 239
cosd, 239
cosh, 240
cot, 239
cotd, 239
couleur, 3102
count, 372
covariance
  population, 374
  sample, 375
covar_pop, 374
covar_samp, 375
CREATE ACCESS METHOD, 1741
CREATE AGGREGATE, 1742
CREATE CAST, 1750
CREATE COLLATION, 1755
CREATE CONVERSION, 1758
CREATE DATABASE, 748, 1760
CREATE DOMAIN, 1765
CREATE EVENT TRIGGER, 1768
CREATE EXTENSION, 1770
CREATE FOREIGN DATA WRAPPER, 1773
CREATE FOREIGN TABLE, 1775
CREATE FUNCTION, 1780
CREATE GROUP, 1790
CREATE INDEX, 1791
CREATE LANGUAGE, 1801

CREATE MATERIALIZED VIEW, 1804
CREATE OPERATOR, 1806
CREATE OPERATOR CLASS, 1809
CREATE OPERATOR FAMILY, 1812
CREATE POLICY, 1813
CREATE PROCEDURE, 1820
CREATE PUBLICATION, 1824
CREATE ROLE, 739, 1829
CREATE RULE, 1834
CREATE SCHEMA, 1837
CREATE SEQUENCE, 1840
CREATE SERVER, 1844
CREATE STATISTICS, 1846
CREATE SUBSCRIPTION, 1850
CREATE TABLE, 7, 1854
CREATE TABLE AS, 1878
CREATE TABLESPACE, 751, 1881
CREATE TEXT SEARCH CONFIGURATION, 1883
CREATE TEXT SEARCH DICTIONARY, 1885
CREATE TEXT SEARCH PARSER, 1887
CREATE TEXT SEARCH TEMPLATE, 1889
CREATE TRANSFORM, 1891
CREATE TRIGGER, 1894
CREATE TYPE, 1902
CREATE USER, 1912
CREATE USER MAPPING, 1913
CREATE VIEW, 1915
createdb, 4, 748, 2128
createuser, 739, 2132
CREATE_REPLICATION_SLOT, 2463
crossed joind, 118
crosstab, 3038, 3040, 3042
crypt, 2973
cstring, 226
ctid, 73, 1346
CUBE, 130
cube (extension), 2886
cume_dist, 379
  hypothetical, 377
current_catalog, 389
current_database, 389
current_date, 290
current_logfiles
  ainsi que le paramètre de configuration log_destination, 671
  et la fonction pg_current_logfile, 390
current_query, 389
current_role, 389
current_schema, 389
current_schemas, 389
current_setting, 408
current_time, 290
current_timestamp, 290
current_user, 390
currval, 357
curseur
  afficher le plan de requête, 1993
  CLOSE, 1719
  DECLARE, 1922
  en PL/pgSQL, 1406
  FETCH, 1999
  MOVE, 2031
cycle
  d'identifiants multixact, 782
Cygwin
  installation sur, 583

D

data type
  énumération (enum), 173
date, 162, 164
  actuelle, 302
  constantes, 166
  format d'affichage, 167
    (voir aussi formatage)
Date Julien, 2706
date_bin, 300
date_part, 290, 294
date_trunc, 290, 299
dblink, 2891, 2897
dblink_build_sql_delete, 2920
dblink_build_sql_insert, 2918
dblink_build_sql_update, 2922
dblink_cancel_query, 2916
dblink_close, 2906
dblink_connect, 2892
dblink_connect_u, 2895
dblink_disconnect, 2896
dblink_error_message, 2909
dblink_exec, 2900
dblink_fetch, 2904
dblink_get_connections, 2908
dblink_get_notify, 2912
dblink_get_pkey, 2917
dblink_get_result, 2913
dblink_is_busy, 2911
dblink_open, 2902
dblink_send_query, 2910
deadlock
  timeout, 702
DEALLOCATE, 1921
dearmor, 2977
decimal (voir numeric)
DECLARE, 1922
Décodage logique, 1565
decode, 255
decode_bytea
  en PL/Perl, 1465
decrypt, 2981
decrypt_iv, 2981
deduplicate_items storage parameter, 1795
default-roles, 3103
deferrable transaction, 693
defined, 2937
degrees, 235
délai

authentification client, 630
delay, 303
DELETE, 15, 115, 1926
RETURNING, 115
delete, 2937
démarrage
au lancement du serveur, 596
dense_rank, 378
hypothetical, 377
dépendance fonctionnel, 129
diagonal, 309
diameter, 309
dict_int, 2923
dict_xsyn, 2924
difference, 2930
digest, 2972
DISCARD, 1929
disjunction, 228
disques durs, 899
DISTINCT, 10
GROUP BY DISTINCT, 130
SELECT DISTINCT, 135
distribution inverse, 376
div, 235
dmetaphone, 2932
dmetaphone_alt, 2932
DO, 1931
document
recherche de texte, 462
domaine, 222
donnée globale
in PL/Tcl, 1446
données binaires
fonctions, 251
données de fuseau horaire, 573
données distantes, 110
données globales
en PL/Python, 1479
double, 10
double precision, 154, 154
droit, 77
avec les règles, 1362
requêtage, 392
avec les vues, 1362
droit de connexion, 740
droits
sur les schémas, 91
DROP ACCESS METHOD, 1933
DROP AGGREGATE, 1934
DROP CAST, 1936
DROP COLLATION, 1937
DROP CONVERSION, 1938
DROP DATABASE, 750, 1939
DROP DOMAIN, 1941
DROP EVENT TRIGGER, 1942
DROP EXTENSION, 1943
DROP FOREIGN DATA WRAPPER, 1945
DROP FOREIGN TABLE, 1946
DROP FUNCTION, 1948
DROP GROUP, 1950
DROP INDEX, 1951
DROP LANGUAGE, 1953
DROP MATERIALIZED VIEW, 1954
DROP OPERATOR, 1955
DROP OPERATOR CLASS, 1957
DROP OPERATOR FAMILY, 1959
DROP OWNED, 1961
DROP POLICY, 1962
DROP PROCEDURE, 1963
DROP PUBLICATION, 1966
DROP ROLE, 739, 1967
DROP ROUTINE, 1968
DROP RULE, 1970
DROP SCHEMA, 1971
DROP SEQUENCE, 1973
DROP SERVER, 1974
DROP STATISTICS, 1975
DROP SUBSCRIPTION, 1976
DROP TABLE, 8, 1978
DROP TABLESPACE, 1979
DROP TEXT SEARCH CONFIGURATION, 1980
DROP TEXT SEARCH DICTIONARY, 1981
DROP TEXT SEARCH PARSER, 1982
DROP TEXT SEARCH TEMPLATE, 1983
DROP TRANSFORM, 1984
DROP TRIGGER, 1986
DROP TYPE, 1987
DROP USER, 1988
DROP USER MAPPING, 1989
DROP VIEW, 1990
dropdb, 750, 2137
dropuser, 739, 2140
DROP_REPLICATION_SLOT, 2467
DTD, 186
DTrace, 575, 876
duplication, 135
dynamic loading, 1250
dynamic_library_path, 1251

E

each, 2937
earth, 2926
earthdistance, 2926
earth_box, 2927
earth_distance, 2927
échappement Unicode
dans les identificateurs, 34
Échappement Unicode
dans des constantes de chaîne, 36
échappements d'antislash, 35
échec de sérialisation, 522
ECPG, 1049
ecpg, 2143
élagage de partition, 107
elog, 2504
dans PL/Python, 1486

en PL/Perl, 1464
in PL/Tcl, 1449
emplacement des données (voir groupe de bases de données)
enabled role, 1186
encodage, 707
encode, 255
encode_array_constructor
en PL/Perl, 1465
encode_array_literal
en PL/Perl, 1465
encode_bytea
en PL/Perl, 1465
encode_typed_literal
en PL/Perl, 1465
encrypt, 2981
encrypt_iv, 2981
END, 1991
enum_first, 304
enum_last, 304
enum_range, 304
ereport, 2504
erreur réessayable, 522
error codes
libpq, 972
error message, 962
espace disque, 775
estampille temporelle, 165
estimation de ligne
multivariée, 2676
estimation de lignes
planificateur, 2670
event_trigger, 226, 226
every, 372
EXCEPT, 135
exceptions
en PL/PgSQL, 1402
in PL/Tcl, 1451
EXECUTE, 1992
exist, 2937
EXISTS, 380
EXIT
en PL/pgSQL, 1397
exp, 235
EXPLAIN, 524, 1993
expression
ordre d'évaluation, 55
syntaxe, 42
expression conditionnelle, 358
expression de table, 117
expression de valeur, 42
expression rationnelle, 259, 261
(voir aussi correspondance de motif)
expressions rationnelles
et locales, 754
expressions rationnelles XQuery, 277
extension, 1304
Extension de SQL, 1224
extensions
maintenus en externe, 3068
extract, 291, 294

F

factorial, 235
failover, 803
false, 172
famille d'opérateur, 1298
famille d'opérateurs, 457
family, 314
fast path, 993
fdw_handler, 226
FETCH, 1999
Fichier d'initialisation, 2655
Fichier de contrôle, 1305
fichier de mots de passe, 1016
fichier des services de connexion, 1017
file_fdw, 2927
FILTER, 45
first_value, 379
flex, 564
float4 (voir real)
float8 (voir double precision)
floor, 235
fonction, 228
appel, 45
argument nommé, 1230
avec SETOF, 1239
dans la clause FROM, 123
définie par l'utilisateur
en C, 1250
en SQL, 1229
interne, 1250
notation mixée, 58
notation par nom, 58
notation par position, 57
paramètre en sortie, 1235
polymorphe, 1225
résolution de types dans un appel, 436
RETURNS TABLE, 1243
statistiques, 430
utilisateur, 1228
valeurs par défaut pour les arguments, 1238
variadic, 1237
fonction d'agrégat, 13
agrégat en déplacement, 1275
agrégation partielle, 1279
appel, 45
ensemble trié, 1278
fonctions de support, 1280
native, 371
polymorphique, 1276
variadique, 1276
fonction d'entrée, 1281
fonction d'initialisation de la bibliothèque, 1250
fonction de fenêtrage, 19
appel, 48

native, 378
fonction de sortie, 1281
fonction de table, 123
XMLTABLE, 330
fonction polymorphe, 1225
fonction variadic, 1237
fonction window
ordre d'exécution, 133
fonctions agrégat
extension, 1273
fonctions de support
in_range, 2586
fonctions de support in_range, 2586
fonctions renvoyant des ensembles
fonctions, 386
Fonctions XML, 322
format, 244, 249
utilisation avec PL/pgSQL, 1387
format base64, 255
Format CSV (Comma-Separated Values)
dans psql, 2262
format escape, 255
format hex, 255
formatage, 279
format_type, 396
Free Space Map, 2654
freebsd
script de lancement, 597
FreeBSD
bibliothèque partagée, 1259
configuration IPC, 601
FSM (voir Free Space Map)
fsm_page_contents, 2960
fuseau horaire, 168, 699
specification POSIX, 2703
fuseaux horaires
saisie d'abréviations, 2702
fuzzystrmatch, 2930

G

gcd, 236
gc_to_sec, 2926
generate_series, 386
generate_subscripts, 387
gen_random_bytes, 2982
gen_random_uuid, 322, 2982
gen_salt, 2973
GEQO (voir optimisation génétique des requêtes)
get_bit, 253, 257
get_byte, 253
get_current_ts_config, 317
get_raw_page, 2959
GIN (voir index)
gin_clean_pending_list, 421
gin_leafpage_items, 2966
gin_metapage_info, 2965
gin_page_opaque_info, 2966
gin_pending_list_limit
paramètre de configuration, 699
paramètre de stockage, 1795
GiST (voir index)
gist_page_items, 2967
gist_page_items_bytea, 2967
gist_page_opaque_info, 2966
GRANT, 77, 2003
GREATEST, 361
détermination du type de résultat, 441
GROUP BY, 13, 128
groupe de bases de données, 593
groupement, 128
GROUPING, 377
GROUPING SETS, 130
gssapi, 618
GSSAPI, 728
avec libpq, 955
GUID, 185
guillemet dollar, 37

H

hash (voir index)
hash_bitmap_info, 2969
hash_metapage_info, 2969
hash_page_items, 2968
hash_page_stats, 2968
hash_page_type, 2968
has_any_column_privilege, 392
has_column_privilege, 392
has_database_privilege, 392
has_foreign_data_wrapper_privilege, 393
has_function_privilege, 393
has_language_privilege, 393
has_parameter_privilege, 393
has_schema_privilege, 393
has_sequence_privilege, 393
has_server_privilege, 393
has_tablespace_privilege, 393
has_table_privilege, 393
has_type_privilege, 393
haute disponibilité, 803
HAVING, 13, 129
heap_page_items, 2960
heap_page_item_attrs, 2961
heap_tuple_infomask_flags, 2961
height, 309
héritage, 93
inheritance, 22
heure, 164
actuelle, 302
constantes, 166
format de sortie, 167
(voir aussi formatage)
heure avec fuseau horaire, 164
heure sans fuseau horaire, 164
historique
de PostgreSQL, xxxiii
hmac, 2972

horodatage, 165
host, 314
host name, 951
hostmask, 314
Hot Standby, 803
HP-UX
  bibliothèque partagée, 1259
  configuration IPC, 602
hstore, 2933, 2935
hstore_to_array, 2936
hstore_to_json, 2936
hstore_to_jsonb, 2936
hstore_to_jsonb_loose, 2936
hstore_to_json_loose, 2936
hstore_to_matrix, 2936
hypothetical-set aggregate
  built-in, 377

I

icount, 2943
ICU, 569, 755, 759, 1756, 1761
ident, 731
identifiant d'objet
  type de données, 223
identifiant de transaction
  cycle, 778
identificateur
  longueur, 34
  syntaxe de, 33
IDENTIFY_SYSTEM, 2462
idx, 2943
IFNULL, 360
IMMUTABLE, 1248
IMPORT FOREIGN SCHEMA, 2009
IN, 380, 383
INCLUDE
  dans la définition des index, 455
include
  dans le fichier de configuration, 624
include_dir
  dans le fichier de configuration, 624
include_if_exists
  dans le fichier de configuration, 622, 624
index, 444, 2955
  B-Tree, 445, 2585
  BRIN, 447, 2633
  combiner des index multiples, 449
  construction en parallèle, 1796
  couvrant, 454
  et ORDER BY, 448
  examiner l'utilisation, 459
  sur expressions, 450
  GIN, 446, 2626
    recherche plein texte, 501
  GiST, 446, 2593
    recherche plein texte, 501
  hash, 446
  Hash, 2647
  multicolonne, 447
  parcours d'index seul, 454
  partiel, 451
  reconstruction en parallèle, 2046
  SP-GiST, 446, 2611
  unique, 450
  verrous, 523
index couvrant, 454
index scan, 664
indexam
  Méthode d'accès aux index, 2564
index_am_handler, 226
indice, 44
inet (type de données), 179
inet_client_addr, 390
inet_client_port, 390
inet_merge, 314
inet_same_family, 314
inet_server_addr, 390
inet_server_port, 390
infinity
  double précision, 155
  numeric (type de données), 153
information d'optimisation
  pour les opérateurs, 1286
informations d'optimisation
  pour les fonctions, 1272
initcap, 244
initdb, 593, 2296
INSERT, 8, 113, 2011
  RETURNING, 115
insertion, 113
installation, 562
  binaires, 561
  sur Windows, 587
instance
  de bases de données (voir instance de bases de données)
instance de bases de données, 7
instr function, 1440, 1440
instructions préparées
  afficher le plan de requête, 1993
  création, 2036
  exécution, 1992
  suppression, 1921
int2 (voir smallint)
int4 (voir integer)
int8 (voir bigint)
intagg, 2941
intarray, 2942
integer, 39, 151
intégrité référentielle, 69
  referential integrity, 16
interfaces
  maintenues en externe, 3068
internal, 226
INTERSECT, 135
interval, 162

intervalle, 170
  format d'affichage, 172
    (voir aussi formatage)
intset, 2943
int_array_aggregate, 2941
int_array_enum, 2941
IS DISTINCT FROM, 231, 383
IS DOCUMENT, 327
IS FALSE, 232
IS NOT DISTINCT FROM, 231, 383
IS NOT DOCUMENT, 327
IS NOT FALSE, 232
IS NOT NULL, 232
IS NOT TRUE, 232
IS NOT UNKNOWN, 232
IS NULL, 232, 704
IS TRUE, 232
IS UNKNOWN, 232
isclosed, 309
isempty, 370, 370
isfinite, 291
isn, 2946
ISNULL, 232
isn_weak, 2948
isolation des transactions, 507
isopen, 309
is_array_ref
  en PL/Perl, 1465
is_valid, 2948

J

jeu de caractère, 763
jeu de caractères, 707
JIT, 922
join, 118
  cross, 118
  left, 119
  natural, 120
  outer, 119
  right, 119
jointure, 11
  contrôlant l'ordre, 541
  croisée, 118
  droite, 119
  externe, 12, 119
  gauche, 119
  réflexive, 12
jointure croisée, 118
jointure droite, 119
jointure externe, 119
jointure gauche, 119
journal des événements
  journal des événements, 619
JSON, 188
  fonctions et opérateurs, 338
JSONB, 188
jsonb
  existence, 191
  inclusion, 191
  index, 193
jsonb_agg, 372
jsonb_array_elements, 342
jsonb_array_elements_text, 342
jsonb_array_length, 343
jsonb_build_array, 341
jsonb_build_object, 341
jsonb_each, 343
jsonb_each_text, 343
jsonb_extract_path, 343
jsonb_extract_path_text, 343
jsonb_insert, 346
jsonb_object, 342
jsonb_object_agg, 372
jsonb_object_keys, 344
jsonb_path_exists, 347
jsonb_path_exists_tz, 348
jsonb_path_match, 347
jsonb_path_match_tz, 348
jsonb_path_query, 347
jsonb_path_query_array, 347
jsonb_path_query_array_tz, 348
jsonb_path_query_first, 347
jsonb_path_query_first_tz, 348
jsonb_path_query_tz, 348
jsonb_populate_record, 344
jsonb_populate_recordset, 345
jsonb_pretty, 348
jsonb_set, 346
jsonb_set_lax, 346
jsonb_strip_nulls, 347
jsonb_to_record, 345
jsonb_to_recordset, 345
jsonb_to_tsvector, 319
jsonb_typeof, 348
jsonpath, 198
json_agg, 372
json_array_elements, 342
json_array_elements_text, 342
json_array_length, 343
json_build_array, 341
json_build_object, 341
json_each, 343
json_each_text, 343
json_extract_path, 343
json_extract_path_text, 343
json_object, 342
json_object_agg, 372
json_object_keys, 344
json_populate_record, 344
json_populate_recordset, 345
json_strip_nulls, 346
json_to_record, 345
json_to_recordset, 345
json_to_tsvector, 319
json_typeof, 348
justify_days, 291

justify_hours, 291
justify_interval, 291

L

label (voir alias)
lag, 379
langage de chemin SQL/JSON, 349
langage de procédures, 1368
langage procédural
  gestionnaire, 2520
Langage procédural
  maintenu en externe, 3068
language_handler, 226
Large Object, 1037
lastval, 357
last_value, 379
LATERAL
  dans la clause FROM, 125
latitude, 2927
lca, 2955
lcm, 236
LDAP, 570, 732
ldconfig, 580
lead, 379
LEAST, 361
  détermination du type de résultat, 441
lecture fantôme, 508
lecture non reproductible, 507
lecture sale, 507
left, 244
left join, 119
length, 244, 253, 257, 309, 317
  of a binary string (voir binary strings, length)
length(tsvector), 477
levenshtein, 2931
levenshtein_less_equal, 2931
lex, 564
libedit, 562
  avec psql, 2277
libperl, 563
libpq, 941
  mode ligne-à-ligne, 991
  mode pipeline, 987
libpq-fe.h, 941, 959
libpq-int.h, 959
libpython, 563
ligne, 7, 60, 177
ligne temporelle, 787
LIKE, 258
  et locales, 754
LIKE_REGEX, 277
  en SQL/JSON, 356
LIMIT, 138
line, 176, 311
linux
  script de lancement, 597
Linux
  bibliothèque partagée, 1259
  configuration IPC, 602
liste cible, 1339
LISTEN, 2019
llvm-config, 570
ll_to_earth, 2927
ln, 236
lo, 2949
LOAD, 2021
locale, 595, 753
localtime, 291
localtimestamp, 291
LOCK, 514, 2022
lock
  advisory, 519
log, 236
log shipping, 803
log transaction (voir WAL)
log10, 236
Logical Decoding, 1561
log_autovacuum_min_duration
  paramètre de configuration, 678
  paramètre de stockage, 1870
longitude, 2927
longueur
  d'une chaîne de caractères (voir chaîne de caractères, longueur)
looks_like_number
  en PL/Perl, 1465
lower, 241, 369, 370
  et locales, 754
lower_inc, 370, 370
lower_inf, 370, 370
lo_close, 1042
lo_creat, 1038, 1042
lo_create, 1038, 1042
lo_export, 1039, 1042
lo_from_bytea, 1042
lo_get, 1042
lo_import, 1038, 1042
lo_import_with_oid, 1039
lo_lseek, 1040
lo_lseek64, 1040
lo_open, 1039
lo_put, 1042
lo_read, 1040
lo_tell, 1041
lo_tell64, 1041
lo_truncate, 1041
lo_truncate64, 1041
lo_unlink, 1042, 1042
lo_write, 1040
lpad, 244
lseg, 177, 311
LSN, 898
ltree, 2951
ltree2text, 2955
ltrim, 244, 253

M

macaddr (type de données), 180
macaddr8 (data type), 180
macaddr8_set7bit, 315
macOS
  bibliothèque partagée, 1259
  configuration IPC, 603
  installation sur, 583
maintenance, 774
make, 562
makeaclitem, 395
make_date, 292
make_interval, 292
make_time, 292
make_timestamp, 292
make_timestamptz, 292
make_valid, 2948
Manifeste de sauvegarde, 2681
MANPATH, 580
marques de citation
  échappement, 35
  et identificateurs, 34
masklen, 314
max, 373
md5, 244, 253
MD5, 727
médian, 47
  (voir aussi pourcentage)
mémoire partagée, 599
memory overcommit, 606
MERGE, 2025
metaphone, 2932
Méthode d'accès aux index, 2564
Méthode d'accès aux tables, 2562
méthode d'échantillonnage de table, 2548
Méthode TABLESAMPLE, 2548
min, 373
MinGW
  installation sur, 584
min_scale, 236
mise à jour, 114, 609
mod, 236
mode
  statistical, 376
mode batch
  dans libpq, 987
mode d'agrégat en déplacement, 1275
mode mono-utilisateur, 2341
modification, 114
module de parcours personnalisé
  gestionnaire, 2552
Modules d'archivage, 1580
mot de passe, 741
  authentification, 727
  du superutilisateur, 594
mot-clé
  liste de, 2708
  syntaxe de, 33
motifs
  dans psql et pg_dump, 2268
MOVE, 2031
multirange (function), 371
Multiversion Concurrency Control, 507
MultiXactId, 782
MVCC, 507
mxid_age, 289

N

NaN (voir not a number (autrement dit, pas un nombre))
natural join, 120
negation, 228
netbsd
  script de lancement, 598
NetBSD
  bibliothèque partagée, 1260
  configuration IPC, 602
netmask, 314
network, 314
nextval, 357
NFS, 595
niveau d'isolation de la transaction
  configuration, 2100
  valeur par défaut, 693
niveau d'isolation de transaction, 508
  lecture validée, 508
  repeatable read, 510
  serializable, 511
nlevel, 2955
nom
  non qualifié, 89
  qualifié, 89
  syntaxe de, 33
nom non qualifié, 89
nom qualifié, 89
nombre
  constante, 38
nombre à virgule flottante
  affichage, 700
nombres à virgule flottante, 152
normalisation Unicode, 241, 242
normalize, 242
normalized, 241
normal_rand, 3037
NOT (opérateur), 228
not a number
  double précision, 155
  numeric (type de données), 153
NOT IN, 380, 383
notation
  fonctions, 57
NOTIFY, 2033
  dans libpq, 994
NOTNULL, 232
now, 292
npoints, 309

nth_value, 379
ntile, 379
NULLIF, 361
numeric, 39
numeric (data type), 151
numnode, 317, 478
num_nonnulls, 233
num_nulls, 233
NVL, 360

O

obj_description, 403
OCCURRENCES_REGEX, 277
octet_length, 242, 242, 252, 257
OFFSET, 138
oid, 223
OID
  dans libpq, 979
oid2name, 3060
old_snapshot, 2958
ON CONFLICT, 2011
ONLY, 118
OOM, 606
openbsd
  script de lancement, 597
OpenBSD
  bibliothèque partagée, 1260
  configuration IPC, 602
OpenSSL, 570
  (voir aussi SSL)
opérateur, 228
  appel, 45
  logical, 228
  précédence, 41
  résolution de types dans un appel, 432
  syntaxe, 40
Opérateur d'ordonnancement, 1301
optimisation génétique des requêtes, 668
option XML, 697
OR (opérateur), 228
Oracle
  porter de PL/SQL vers PL/pgSQL, 1433
ORDER BY, 10, 137
  et locales, 754
ordinality, 388
Origines de la réplication, 1579
où tracer, 671
outer join, 119
outils d'administration
  maintenus en externe, 3068
overcommit, 606
OVERLAPS, 293
overlay, 242, 252, 257

P

pageinspect, 2959
page_checksum, 2959
page_header, 2959
palloc, 1258
PAM, 571, 736
paramètre
  syntaxe, 43
paramètre de configuration allow_in_place_tablespaces, 708
paramètre de configuration allow_system_table_mods, 708
paramètre de configuration application_name, 678
paramètre de configuration archive_cleanup_command, 653
paramètre de configuration archive_command, 651
paramètre de configuration archive_library, 651
paramètre de configuration archive_mode, 650
paramètre de configuration archive_timeout, 651
paramètre de configuration array_nulls, 703
paramètre de configuration authentication_timeout, 630
paramètre de configuration auth_delay.milliseconds, 2874
paramètre de configuration autovacuum, 688
paramètre de configuration autovacuum_work_mem, 637
paramètre de configuration auto_explain.log_analyze, 2874
paramètre de configuration auto_explain.log_buffers, 2875
paramètre de configuration auto_explain.log_format, 2875
paramètre de configuration auto_explain.log_min_duration, 2874
paramètre de configuration auto_explain.log_nested_statements, 2876
paramètre de configuration auto_explain.log_timing, 2875
paramètre de configuration auto_explain.log_triggers, 2875
paramètre de configuration auto_explain.log_verbose, 2875
paramètre de configuration auto_explain.log_wal, 2875
paramètre de configuration auto_explain.sample_rate, 2876
paramètre de configuration backend_flush_after, 641
paramètre de configuration backslash_quote, 703
paramètre de configuration backtrace_functions, 708
Paramètre de configuration basebackup_to_shell.command, 2877
Paramètre de configuration basebackup_to_shell.required_role, 2877
Paramètre de configuration basic_archive.archive_directory, 2877
paramètre de configuration bgwriter_delay, 640
paramètre de configuration bgwriter_flush_after, 641
paramètre de configuration bgwriter_lru_maxpages, 641
paramètre de configuration bgwriter_lru_multiplier, 641

paramètre de configuration block_size, 706
paramètre de configuration bonjour, 629
paramètre de configuration bonjour_name, 629
paramètre de configuration bytea_output, 696
paramètre de configuration checkpoint_completion_target, 649
paramètre de configuration checkpoint_flush_after, 650
paramètre de configuration checkpoint_timeout, 649
paramètre de configuration checkpoint_warning, 650
paramètre de configuration check_function_bodies, 693
paramètre de configuration client_connection_check_interval, 630
paramètre de configuration client_encoding, 700
paramètre de configuration client_min_messages, 690
paramètre de configuration cluster_name, 686
paramètre de configuration commit_delay , 649
paramètre de configuration commit_siblings , 649
paramètre de configuration compute_query_id, 688
paramètre de configuration config_file , 626
paramètre de configuration constraint_exclusion, 670
paramètre de configuration cpu_index_tuple_cost, 667
paramètre de configuration cpu_operator_cost, 667
paramètre de configuration cpu_tuple_cost, 667
paramètre de configuration cursor_tuple_fraction, 670
paramètre de configuration data_checksums, 706
paramètre de configuration data_directory, 626
paramètre de configuration data_directory_mode, 706
paramètre de configuration data_sync_retry, 705
paramètre de configuration datestyle, 699
paramètre de configuration db_user_namespace, 631
paramètre de configuration deadlock_timeout, 702
paramètre de configuration debug_assertions, 706
paramètre de configuration debug_deadlocks, 711
paramètre de configuration debug_discard_caches, 709
paramètre de configuration debug_pretty_print, 678
paramètre de configuration debug_print_parse, 678
paramètre de configuration debug_print_plan, 678
paramètre de configuration debug_print_rewritten, 678
paramètre de configuration default_statistics_target, 669
paramètre de configuration default_tablespace, 692
paramètre de configuration default_table_access_method, 692
paramètre de configuration default_text_search_config, 701
paramètre de configuration default_toast_compression, 692
paramètre de configuration default_transaction_deferrable, 693
paramètre de configuration default_transaction_isolation, 693
paramètre de configuration default_transaction_read_only, 693
paramètre de configuration dynamic_library_path, 701
paramètre de configuration dynamic_shared_memory_type, 638
paramètre de configuration effective_cache_size, 668
paramètre de configuration effective_io_concurrency, 642
paramètre de configuration enable_async_append, 664
paramètre de configuration enable_bitmapscan, 664
Paramètre de configuration enable_gathermerge, 664
paramètre de configuration enable_hashagg, 664
paramètre de configuration enable_hashjoin, 664
paramètre de configuration enable_incremental_sort, 664
paramètre de configuration enable_indexonlyscan, 664
paramètre de configuration enable_indexscan, 664
paramètre de configuration enable_material, 665
paramètre de configuration enable_memoize, 665
paramètre de configuration enable_mergejoin, 665
paramètre de configuration enable_nestloop, 665
paramètre de configuration enable_parallel_append, 665
paramètre de configuration enable_parallel_hash, 665
paramètre de configuration enable_partitionwise_aggregate, 665
paramètre de configuration enable_partitionwise_join, 665
paramètre de configuration enable_partition_pruning, 665
paramètre de configuration enable_seqscan, 666
paramètre de configuration enable_sort, 666
paramètre de configuration enable_tidscan, 666
paramètre de configuration escape_string_warning, 703
paramètre de configuration event_source, 675
paramètre de configuration exit_on_error, 705
paramètre de configuration external_pid_file, 626
paramètre de configuration extra_float_digits, 700
paramètre de configuration force_parallel_mode, 709
paramètre de configuration from_collapse_limit, 670
paramètre de configuration fsync, 645
paramètre de configuration full_page_writes, 647
paramètre de configuration geqo, 669
paramètre de configuration geqo_effort, 669
paramètre de configuration geqo_generations , 669
paramètre de configuration geqo_pool_size , 669
paramètre de configuration geqo_seed, 669
paramètre de configuration geqo_selection_bias , 669
paramètre de configuration geqo_threshold, 669
paramètre de configuration gin_fuzzy_search_limit, 701
paramètre de configuration hash_mem_multiplier, 636
paramètre de configuration hba_file, 626
paramètre de configuration hot_standby, 660
paramètre de configuration hot_standby_feedback, 661
paramètre de configuration huge_pages, 635
paramètre de configuration huge_page_size, 635
paramètre de configuration ident_file, 626
paramètre de configuration idle_in_transaction_session_timeout, 694
paramètre de configuration idle_session_timeout, 695

paramètre de configuration ignore_checksum_failure, 712
paramètre de configuration ignore_invalid_pages, 712
paramètre de configuration ignore_system_indexes, 709
paramètre de configuration integer_datetimes, 706
paramètre de configuration IntervalStyle, 699
paramètre de configuration in_hot_standby, 706
paramètre de configuration jit, 671
paramètre de configuration jit_above_cost, 668
paramètre de configuration jit_debugging_support, 713
paramètre de configuration jit_dump_bitcode, 713
paramètre de configuration jit_expressions, 713
paramètre de configuration jit_inline_above_cost, 668
paramètre de configuration jit_optimize_above_cost, 668
paramètre de configuration jit_profiling_support, 713
paramètre de configuration jit_provider, 699
paramètre de configuration jit_tuple_deforming, 713
paramètre de configuration join_collapse_limit, 671
paramètre de configuration krb_caseins_users, 631
paramètre de configuration krb_server_keyfile, 630
paramètre de configuration lc_collate, 706
paramètre de configuration lc_ctype, 707
paramètre de configuration lc_messages, 700
paramètre de configuration lc_monetary, 700
paramètre de configuration lc_numeric, 701
paramètre de configuration lc_time, 701
paramètre de configuration listen_addresses , 627
paramètre de configuration local_preload_libraries, 697
paramètre de configuration lock_timeout, 694
paramètre de configuration logging_collector, 672
paramètre de configuration logical_decoding_work_mem, 637
paramètre de configuration log_btree_build_stats, 711
paramètre de configuration log_checkpoints, 679
paramètre de configuration log_connections, 679
paramètre de configuration log_destination, 671
paramètre de configuration log_directory , 673
paramètre de configuration log_disconnections, 679
paramètre de configuration log_duration, 679
paramètre de configuration log_error_verbosity, 679
paramètre de configuration log_executor_stats, 688
paramètre de configuration log_filename , 673
paramètre de configuration log_file_mode, 673
paramètre de configuration log_hostname, 680
paramètre de configuration log_line_prefix, 680
paramètre de configuration log_lock_waits, 682
paramètre de configuration log_min_duration_sample, 676
paramètre de configuration log_min_duration_statement, 676
paramètre de configuration log_min_error_statement, 675
paramètre de configuration log_min_messages, 675
paramètre de configuration log_parameter_max_length, 682
paramètre de configuration log_parameter_max_length_on_error, 682
paramètre de configuration log_parser_stats, 688
paramètre de configuration log_planner_stats, 688
paramètre de configuration log_recovery_conflict_waits, 682
paramètre de configuration log_replication_commands, 683
paramètre de configuration log_rotation_age , 674
paramètre de configuration log_rotation_size, 674
paramètre de configuration log_startup_progress_interval, 677
paramètre de configuration log_statement, 682
paramètre de configuration log_statement_sample_rate, 676
paramètre de configuration log_statement_stats, 688
paramètre de configuration log_temp_files, 683
paramètre de configuration log_timezone, 683
paramètre de configuration log_transaction_sample_rate, 677
paramètre de configuration log_truncate_on_rotation, 674
paramètre de configuration lo_compat_privileges , 704
paramètre de configuration maintenance_io_concurrency, 642
paramètre de configuration maintenance_work_mem, 637
paramètre de configuration max_connections , 627
paramètre de configuration max_files_per_process, 639
paramètre de configuration max_function_args, 707
paramètre de configuration max_identifier_length, 707
paramètre de configuration max_index_keys, 707
paramètre de configuration max_locks_per_transaction, 702
Paramètre de configuration max_logical_replication_workers, 663
paramètre de configuration max_parallel_maintenance_workers, 643
paramètre de configuration max_parallel_workers, 643
paramètre de configuration max_parallel_workers_per_gather, 642
paramètre de configuration max_pred_locks_per_page, 703
Paramètre de configuration max_pred_locks_per_relation, 702
paramètre de configuration max_pred_locks_per_transaction, 702
paramètre de configuration max_prepared_transactions, 636
paramètre de configuration max_replication_slots, 656
paramètre de configuration max_slot_wal_keep_size, 657
paramètre de configuration max_stack_depth , 637
paramètre de configuration max_standby_archive_delay, 660
paramètre de configuration max_standby_streaming_delay, 661

Paramètre de configuration max_sync_workers_per_subscription, 663
paramètre de configuration max_wal_senders, 656
paramètre de configuration max_wal_size, 650
paramètre de configuration max_worker_processes, 642
paramètre de configuration min_dynamic_shared_memory, 638
paramètre de configuration min_parallel_index_scan_size, 668
paramètre de configuration min_parallel_table_scan_size, 667
paramètre de configuration min_wal_size, 650
paramètre de configuration old_snapshot_threshold, 643
paramètre de configuration output_plugin_libraries, 656
paramètre de configuration parallel_leader_participation, 643
paramètre de configuration parallel_setup_cost, 667
paramètre de configuration parallel_tuple_cost, 667
paramètre de configuration password_encryption, 630
paramètre de configuration pg_stat_statements.max, 2992
paramètre de configuration pg_stat_statements.save, 2992
paramètre de configuration pg_stat_statements.track, 2992
paramètre de configuration pg_stat_statements.track_planning, 2992
paramètre de configuration pg_stat_statements.track_utility, 2992
paramètre de configuration pg_trgm.similarity_threshold, 3004
paramètre de configuration pg_trgm.strict_word_similarity_threshold , 3004
paramètre de configuration pg_trgm.word_similarity_threshold , 3004
paramètre de configuration plan_cache_mode, 671
paramètre de configuration plperl.on_init, 1470
paramètre de configuration plperl.on_plperlu_init, 1471
paramètre de configuration plperl.on_plperl_init, 1471
paramètre de configuration plperl.use_strict, 1471
paramètre de configuration plpgsql.check_asserts, 1416
paramètre de configuration pltcl.start_proc , 1454
paramètre de configuration pltclu.start_proc, 1454
paramètre de configuration port, 627
paramètre de configuration postgres_fdw.application_name, 3019
paramètre de configuration post_auth_delay, 709
paramètre de configuration pre_auth_delay, 710
paramètre de configuration primary_conninfo, 660
paramètre de configuration primary_slot_name, 660
paramètre de configuration promote_trigger_file, 660
paramètre de configuration quote_all_identifiers, 704
paramètre de configuration random_page_cost, 666
paramètre de configuration recovery_end_command, 653
paramètre de configuration recovery_init_sync_method, 705
paramètre de configuration recovery_min_apply_delay, 662
paramètre de configuration recovery_prefetch, 652
paramètre de configuration recovery_target, 654
paramètre de configuration recovery_target_action, 655
paramètre de configuration recovery_target_inclusive, 654
paramètre de configuration recovery_target_lsn, 654
paramètre de configuration recovery_target_name, 654
paramètre de configuration recovery_target_time, 654
paramètre de configuration recovery_target_timeline, 655
paramètre de configuration recovery_target_xid, 654
paramètre de configuration recursive_worktable_factor, 671
paramètre de configuration regex_flavor, 703
paramètre de configuration remove_temp_files_after_crash, 713
paramètre de configuration restart_after_crash, 705
paramètre de configuration restore_command, 652
paramètre de configuration row_security, 691
paramètre de configuration search_path, 90, 691
  utilisé pour sécuriser les fonctions, 1787
paramètre de configuration segment_size, 707
paramètre de configuration sepgsql.debug_audit, 3027
paramètre de configuration sepgsql.permissive, 3027
paramètre de configuration seq_page_cost, 666
paramètre de configuration server_encoding, 707
paramètre de configuration server_version, 707
paramètre de configuration server_version_num, 707
paramètre de configuration session_preload_libraries, 698
paramètre de configuration session_replication_role, 694
paramètre de configuration shared_buffers, 634
paramètre de configuration shared_memory_size, 707
paramètre de configuration shared_memory_size_in_huge_pages, 707
paramètre de configuration shared_memory_type, 638
paramètre de configuration shared_preload_libraries, 698
paramètre de configuration ssl, 631
paramètre de configuration ssl_ca_file, 632
paramètre de configuration ssl_cert_file, 632
paramètre de configuration ssl_ciphers, 631
paramètre de configuration ssl_crl_dir, 633
paramètre de configuration ssl_crl_file, 633
paramètre de configuration ssl_dh_params_file, 634
paramètre de configuration ssl_ecdh_curve, 632
paramètre de configuration ssl_key_file, 633
paramètre de configuration ssl_library, 707
paramètre de configuration ssl_max_protocol_version, 633

paramètre de configuration ssl_min_protocol_version, 633
paramètre de configuration ssl_passphrase_command, 634
paramètre de configuration ssl_passphrase_command_supports_reload, 634
paramètre de configuration ssl_prefer_server_ciphers, 632
paramètre de configuration standard_conforming_strings, 704
paramètre de configuration statement_timeout, 694
paramètre de configuration superuser_reserved_connections, 627
paramètre de configuration synchronize_seqscans, 704
paramètre de configuration synchronous_commit, 645
paramètre de configuration synchronous_standby_names, 658
paramètre de configuration syslog_facility, 674
paramètre de configuration syslog_ident, 675
paramètre de configuration syslog_sequence_numbers, 675
paramètre de configuration syslog_split_messages, 675
paramètre de configuration tcp_keepalives_count, 629
paramètre de configuration tcp_keepalives_idle, 629
paramètre de configuration tcp_keepalives_interval, 629
paramètre de configuration tcp_user_timeout, 630
paramètre de configuration temp_buffers, 636
paramètre de configuration temp_file_limit, 638
paramètre de configuration temp_tablespaces, 692
paramètre de configuration TimeZone, 699
paramètre de configuration timezone_abbreviations, 700
paramètre de configuration trace_locks, 710
paramètre de configuration trace_lock_oidmin, 711
paramètre de configuration trace_lock_table, 711
paramètre de configuration trace_lwlocks, 711
paramètre de configuration trace_notify, 710
paramètre de configuration trace_recovery_messages, 710
paramètre de configuration trace_sort, 710
paramètre de configuration trace_userlocks, 711
paramètre de configuration track_activities, 686
paramètre de configuration track_activity_query_size, 687
paramètre de configuration track_commit_timestamp, 658
paramètre de configuration track_counts, 687
paramètre de configuration track_functions, 687
paramètre de configuration track_io_timing, 687
paramètre de configuration track_wal_io_timing, 687
paramètre de configuration transaction_deferrable, 693
paramètre de configuration transaction_isolation, 693
paramètre de configuration transaction_read_only, 693
paramètre de configuration transform_null_equals, 704
paramètre de configuration unix_socket_directories, 627
paramètre de configuration unix_socket_group , 628
paramètre de configuration unix_socket_permissions, 628
paramètre de configuration update_process_title, 686
paramètre de configuration vacuum_cost_delay, 639
paramètre de configuration vacuum_cost_limit, 640
paramètre de configuration vacuum_cost_page_dirty, 640
paramètre de configuration vacuum_cost_page_hit, 639
paramètre de configuration vacuum_cost_page_miss, 640
paramètre de configuration vacuum_defer_cleanup_age, 659
paramètre de configuration vacuum_failsafe_age, 695
paramètre de configuration vacuum_freeze_min_age, 695
paramètre de configuration vacuum_freeze_table_age, 695
paramètre de configuration vacuum_multixact_failsafe_age, 696
paramètre de configuration vacuum_multixact_freeze_min_age, 696
paramètre de configuration vacuum_multixact_freeze_table_age, 696
paramètre de configuration wal_block_size, 708
paramètre de configuration wal_buffers, 648
paramètre de configuration wal_compression, 647
Paramètre de configuration wal_consistency_checking, 711
paramètre de configuration wal_debug, 712
paramètre de configuration wal_decode_buffer_size, 652
paramètre de configuration wal_init_zero, 648
paramètre de configuration wal_keep_size, 657
paramètre de configuration wal_level, 644
paramètre de configuration wal_log_hints, 647
paramètre de configuration wal_receiver_create_temp_slot, 661
paramètre de configuration wal_receiver_status_interval, 661
paramètre de configuration wal_receiver_timeout, 662
paramètre de configuration wal_recycle, 648
paramètre de configuration wal_retrieve_retry_interval, 662
paramètre de configuration wal_segment_size, 708
paramètre de configuration wal_sender_timeout, 657
paramètre de configuration wal_skip_threshold, 649
paramètre de configuration wal_sync_method, 646
paramètre de configuration wal_writer_delay, 648
paramètre de configuration wal_writer_flush_after, 648
paramètre de configuration work_mem, 636
paramètre de configuration xmlbinary, 696
paramètre de configuration xmloption, 697
paramètre de configuration zero_damaged_pages, 712
paramètre de stockage autosummarize, 1795
paramètre de stockage autovacuum_enabled, 1868
paramètre de stockage autovacuum_freeze_min_age, 1869

paramètre de stockage autovacuum_freeze_table_age, 1869
paramètre de stockage autovacuum_multixact_freeze_min_age, 1870
paramètre de stockage autovacuum_multixact_freeze_table_age, 1870
paramètre de stockage buffering, 1795
paramètre de stockage fastupdate, 1795
paramètre de stockage fillfactor, 1794, 1868
paramètre de stockage pages_per_range, 1795
paramètre de stockage parallel_workers, 1868
paramètre de stockage toast_tuple_target, 1868
paramètre de stockage user_catalog_table, 1870
paramètre de stockage vacuum_index_cleanup, 1868
paramètre de stockagevacuum_truncate, 1868
paramètres de stockage, 1867
parcours bitmap, 664
parcours d'index seul, 454
parcours de bitmap, 449
parcours index-only, 664
parcours séquentiel, 666
parenthèses, 43
parse_ident, 244
partitionnement, 96
partitionnement de données, 803
password
  authentification, 727
passwordcheck, 2969
path, 311
  pour schémas, 691
PATH, 580
path (type de données), 177
pclose, 309
peer, 731
percentile
  continuous, 376
  discrete, 376
percent_rank, 378
  hypothetical, 377
performance, 524
perl, 564
Perl, 1455
permission (voir droit)
perte acceptée, 546
pfree, 1258
PGAPPNAME, 1015
pgbench, 2162
PGcancel, 992
PGCHANNELBINDING, 1015
PGCLIENTENCODING, 1016
PGconn, 941
PGCONNECT_TIMEOUT, 1016
pgcrypto, 2972
pgdata, 594
PGDATABASE, 1015
PGDATESTYLE, 1016
PGEventProc, 1010
PGGEQO, 1016
PGGSSENCMODE, 1016
PGGSSLIB, 1016
PGHOST, 1015
PGHOSTADDR, 1015
PGKRBSRVNAME, 1016
PGLOCALEDIR, 1016
PGOPTIONS, 1015
PGPASSFILE, 1015
PGPASSWORD, 1015
PGPORT, 1015
pgp_armor_headers, 2977
pgp_key_id, 2976
pgp_pub_decrypt, 2976
pgp_pub_decrypt_bytea, 2976
pgp_pub_encrypt, 2976
pgp_pub_encrypt_bytea, 2976
pgp_sym_decrypt, 2976
pgp_sym_decrypt_bytea, 2976
pgp_sym_encrypt, 2975
pgp_sym_encrypt_bytea, 2975
PGREQUIREPEER, 1015
PGREQUIRESSL, 1015
PGresult, 969
pgrowlocks, 2986, 2986
PGSERVICE, 1015
PGSERVICEFILE, 1015
PGSSLCERT, 1015
PGSSLCOMPRESSION, 1015
PGSSLCRL, 1015
PGSSLCRLDIR, 1015
PGSSLKEY, 1015
PGSSLMAXPROTOCOLVERSION, 1016
PGSSLMINPROTOCOLVERSION, 1015
PGSSLMODE, 1015
PGSSLROOTCERT, 1015
PGSSLSNI, 1015
pgstatginindex, 2998
pgstathashindex, 2998
pgstatindex, 2997
pgstattuple, 2995, 2997
pgstattuple_approx, 2999
PGSYSCONFDIR, 1016
PGTARGETSESSIONATTRS, 1016
PGTZ, 1016
PGUSER, 1015
pgxs, 1314
pg_advisory_lock, 424
pg_advisory_lock_shared, 424
pg_advisory_unlock, 425
pg_advisory_unlock_all, 425
pg_advisory_unlock_shared, 425
pg_advisory_xact_lock, 425
pg_advisory_xact_lock_shared, 425
pg_aggregate, 2361
pg_am, 2362
pg_amcheck, 2146
pg_amop, 2363
pg_amproc, 2364

pg_archivecleanup, 2301
pg_attrdef, 2364
pg_attribute, 2365
pg_authid, 2367
pg_auth_members, 2368
pg_available_extensions, 2416
pg_available_extension_versions, 2416
pg_backend_memory_contexts, 2417
pg_backend_pid, 390
pg_backup_start, 411
pg_backup_stop, 411
pg_basebackup, 2152
pg_blocking_pids, 390
pg_buffercache, 2970
pg_buffercache_pages, 2970
pg_cancel_backend, 409
pg_cast, 2369
pg_char_to_encoding, 396
pg_checksums, 2303
pg_class, 2369
pg_client_encoding, 245
pg_collation, 2372
pg_collation_actual_version, 420
pg_collation_is_visible, 395
PG_COLOR, 3102
PG_COLORS, 3102
pg_column_compression, 418
pg_column_size, 418
pg_config, 2188, 2418
  avec des fonctions C définies par l'utilisateur, 1258
  avec ecpg, 1113
  avec libpq, 1024
pg_conf_load_time, 390
pg_constraint, 2373
pg_controldata, 2305
pg_control_checkpoint, 406
pg_control_init, 406
pg_control_recovery, 407
pg_control_system, 406
pg_conversion, 2375
pg_conversion_is_visible, 395
pg_copy_logical_replication_slot, 416
pg_copy_physical_replication_slot, 416
pg_create_logical_replication_slot, 416
pg_create_physical_replication_slot, 415
pg_create_restore_point, 410
pg_ctl, 594, 596, 2306
pg_current_logfile, 390
pg_current_snapshot, 404
pg_current_wal_flush_lsn, 410
pg_current_wal_insert_lsn, 411
pg_current_wal_lsn, 411
pg_current_xact_id, 404
pg_current_xact_id_if_assigned, 404
pg_cursors, 2418
pg_database, 749, 2375
pg_database_collation_actual_version, 420
pg_database_size, 418
pg_db_role_setting, 2377
pg_ddl_command, 226
pg_default_acl, 2377
pg_depend, 2378
pg_describe_object, 402
pg_description, 2380
pg_drop_replication_slot, 415
pg_dump, 2191
pg_dumpall, 2206
  utilisation lors d'une mise à jour, 610
pg_encoding_to_char, 396
pg_enum, 2380
pg_event_trigger, 2381
pg_event_trigger_ddl_commands, 427
pg_event_trigger_dropped_objects, 428
pg_event_trigger_table_rewrite_oid, 429
pg_event_trigger_table_rewrite_reason, 429
pg_export_snapshot, 415
pg_extension, 2381
pg_extension_config_dump, 1309
pg_filenode_relation, 420
pg_file_rename, 2867
pg_file_settings, 2419
pg_file_sync, 2867
pg_file_unlink, 2868
pg_file_write, 2867
pg_foreign_data_wrapper, 2382
pg_foreign_server, 2383
pg_foreign_table, 2383
pg_freespace, 2983
pg_freespacemap, 2983
pg_function_is_visible, 395
pg_get_catalog_foreign_keys, 396
pg_get_constraintdef, 396
pg_get_expr, 397
pg_get_functiondef, 397
pg_get_function_arguments, 397
pg_get_function_identity_arguments, 397
pg_get_function_result, 397
pg_get_indexdef, 397
pg_get_keywords, 397
pg_get_multixact_members, 404
pg_get_object_address, 403
pg_get_partition_constraintdef, 395
pg_get_ruledef, 397
pg_get_serial_sequence, 397
pg_get_statisticsobjdef, 398
pg_get_triggerdef, 398
pg_get_userbyid, 398
pg_get_viewdef, 398
pg_get_wal_replay_pause_state, 413
pg_get_wal_resource_managers, 413
pg_group, 2420
pg_has_role, 394
pg_hba.conf, 715
pg_hba_file_rules, 2420
pg_ident.conf, 724
pg_identify_object, 402

pg_identify_object_as_address, 403
pg_ident_file_mappings, 2421
pg_import_system_collations, 420
pg_index, 2384
pg_indexam_has_property, 398
pg_indexes, 2422
pg_indexes_size, 418
pg_index_column_has_property, 398
pg_index_has_property, 398
pg_inherits, 2385
pg_init_privs, 2386
pg_isready, 2214
pg_is_in_recovery, 413
pg_is_other_temp_schema, 391
pg_is_wal_replay_paused, 413
pg_jit_available, 391
pg_language, 2386
pg_largeobject, 2387
pg_largeobject_metadata, 2388
pg_last_committed_xact, 406
pg_last_wal_receive_lsn, 413
pg_last_wal_replay_lsn, 413
pg_last_xact_replay_timestamp, 413
pg_listening_channels, 391
pg_locks, 2422
pg_logdir_ls, 2868
pg_logical_emit_message, 418
pg_logical_slot_get_binary_changes, 416
pg_logical_slot_get_changes, 416
pg_logical_slot_peek_binary_changes, 417
pg_logical_slot_peek_changes, 416
pg_log_backend_memory_contexts, 409
pg_lsn, 225
pg_ls_archive_statusdir, 423
pg_ls_dir, 422
pg_ls_logdir, 422
pg_ls_logicalmapdir, 423
pg_ls_logicalsnapdir, 423
pg_ls_replslotdir, 423
pg_ls_tmpdir, 423
pg_ls_waldir, 422
pg_matviews, 2425
pg_mcv_list_items, 430
pg_my_temp_schema, 391
pg_namespace, 2388
pg_notification_queue_usage, 391
pg_notify, 2034
pg_opclass, 2388
pg_opclass_is_visible, 395
pg_operator, 2389
pg_operator_is_visible, 395
pg_opfamily, 2390
pg_opfamily_is_visible, 395
pg_options_to_table, 399
pg_parameter_acl, 2390
pg_partitioned_table, 2391
pg_partition_ancestors, 421
pg_partition_root, 421
pg_partition_tree, 420
pg_policies, 2425
pg_policy, 2391
pg_postmaster_start_time, 391
pg_prepared_statements, 2426
pg_prepared_xacts, 2426
pg_prewarm, 2984
pg_prewarm.autoprewarm configuration parameter, 2985
pg_prewarm.autoprewarm_interval configuration parameter, 2985
pg_proc, 2392
pg_promote, 414
pg_publication, 2395
pg_publication_namespace, 2395
pg_publication_rel, 2396
pg_publication_tables, 2427
pg_range, 2396
pg_read_binary_file, 424
pg_read_file, 423
pg_receivewal, 2216
pg_receivexlog, 3104 (voir pg_receivewal)
pg_recvlogical, 2221
pg_relation_filenode, 419
pg_relation_filepath, 420
pg_relation_size, 418
pg_reload_conf, 409
pg_relpages, 2999
pg_replication_origin, 2397
pg_replication_origin_advance, 417
pg_replication_origin_create, 417
pg_replication_origin_drop, 417
pg_replication_origin_oid, 417
pg_replication_origin_progress, 418
pg_replication_origin_session_is_setup, 417
pg_replication_origin_session_progress, 417
pg_replication_origin_session_reset, 417
pg_replication_origin_session_setup, 417
pg_replication_origin_status, 2427
pg_replication_origin_xact_reset, 417
pg_replication_origin_xact_setup, 417
pg_replication_slots, 2428
pg_replication_slot_advance, 417
pg_resetwal, 2312
pg_resetxlog, 3103 (voir pg_resetwal)
pg_restore, 2225
pg_rewind, 2316
pg_rewrite, 2397
pg_roles, 2429
pg_rotate_logfile, 409
pg_rules, 2430
pg_safe_snapshot_blocking_pids, 391
pg_seclabel, 2398
pg_seclabels, 2431
pg_sequence, 2398
pg_sequences, 2431
pg_service.conf, 1017
pg_settings, 2432

pg_settings_get_flags, 399
pg_shadow, 2434
pg_shdepend, 2399
pg_shdescription, 2400
pg_shmem_allocations, 2435
pg_shseclabel, 2400
pg_size_bytes, 419
pg_size_pretty, 419
pg_sleep, 303
pg_sleep_for, 303
pg_sleep_until, 303
pg_snapshot_xip, 404
pg_snapshot_xmax, 404
pg_snapshot_xmin, 404
pg_statio_all_indexes, 832, 862
pg_statio_all_sequences, 832, 863
pg_statio_all_tables, 831, 862
pg_statio_sys_indexes, 832
pg_statio_sys_sequences, 832
pg_statio_sys_tables, 832
pg_statio_user_indexes, 832
pg_statio_user_sequences, 832
pg_statio_user_tables, 832
pg_statistic, 537, 2401
pg_statistics_obj_is_visible, 395
pg_statistic_ext, 538, 2402
pg_statistic_ext_data, 538, 2402
pg_stats, 537, 2436
pg_stats_ext, 2437
pg_stats_ext_exprs, 2438
pg_stat_activity, 830, 833
pg_stat_all_indexes, 831, 861
pg_stat_all_tables, 831, 859
pg_stat_archiver, 830, 855
pg_stat_bgwriter, 831, 855
pg_stat_clear_snapshot, 865
pg_stat_database, 831, 857
pg_stat_database_conflicts, 831, 859
pg_stat_file, 424
pg_stat_get_activity, 865
pg_stat_get_backend_activity, 866
pg_stat_get_backend_activity_start, 866
pg_stat_get_backend_client_addr, 866
pg_stat_get_backend_client_port, 867
pg_stat_get_backend_dbid, 867
pg_stat_get_backend_idset, 866
pg_stat_get_backend_pid, 867
pg_stat_get_backend_start, 867
pg_stat_get_backend_userid, 867
pg_stat_get_backend_wait_event, 867
pg_stat_get_backend_wait_event_type, 867
pg_stat_get_backend_xact_start, 867
pg_stat_get_snapshot_timestamp, 865
pg_stat_get_xact_blocks_fetched, 865
pg_stat_get_xact_blocks_hit, 865
pg_stat_gssapi, 830, 854
pg_stat_progress_analyze, 830, 867
pg_stat_progress_basebackup, 830, 874
pg_stat_progress_cluster, 830, 873
pg_stat_progress_copy, 830, 875
pg_stat_progress_create_index, 830, 869
pg_stat_progress_vacuum, 830, 871
pg_stat_recovery_prefetch, 830, 852
pg_stat_replication, 830, 848
pg_stat_replication_slots, 832, 850
pg_stat_reset, 865
pg_stat_reset_replication_slot, 866
pg_stat_reset_shared, 865
pg_stat_reset_single_function_counters, 865
pg_stat_reset_single_table_counters, 865
pg_stat_reset_slru, 866
pg_stat_reset_subscription_stats, 866
pg_stat_slru, 832, 864
pg_stat_ssl, 830, 853
pg_stat_statements, 2987
  fonction, 2992
pg_stat_statements_info, 2991
pg_stat_statements_reset, 2991
pg_stat_subscription, 830, 853
pg_stat_subscription_stats, 832, 853
pg_stat_sys_indexes, 831
pg_stat_sys_tables, 831
pg_stat_user_functions, 832, 863
pg_stat_user_indexes, 831
pg_stat_user_tables, 831
pg_stat_wal, 831, 856
pg_stat_wal_receiver, 830, 851
pg_stat_xact_all_tables, 831
pg_stat_xact_sys_tables, 831
pg_stat_xact_user_functions, 832
pg_stat_xact_user_tables, 831
pg_subscription, 2404
pg_subscription_rel, 2405
pg_surgery, 3000
pg_switch_wal, 411
pg_tables, 2440
pg_tablespace, 2405
pg_tablespace_databases, 399
pg_tablespace_location, 399
pg_tablespace_size, 419
pg_table_is_visible, 395
pg_table_size, 419
pg_temp, 691
  sécuriser les fonctions, 1787
pg_terminate_backend, 409
pg_test_fsync, 2320
pg_test_timing, 2321
pg_timezone_abbrevs, 2440
pg_timezone_names, 2441
pg_total_relation_size, 419
pg_transform, 2406
pg_trgm, 3001
pg_trigger, 2406
pg_trigger_depth, 391
pg_try_advisory_lock, 425
pg_try_advisory_lock_shared, 425

pg_try_advisory_xact_lock, 425
pg_try_advisory_xact_lock_shared, 425
pg_ts_config, 2408
pg_ts_config_is_visible, 395
pg_ts_config_map, 2408
pg_ts_dict, 2409
pg_ts_dict_is_visible, 396
pg_ts_parser, 2409
pg_ts_parser_is_visible, 396
pg_ts_template, 2410
pg_ts_template_is_visible, 396
pg_type, 2410
pg_typeof, 399
pg_type_is_visible, 396
pg_upgrade, 2325
pg_user, 2441
pg_user_mapping, 2414
pg_user_mappings, 2442
pg_verifybackup, 2235
pg_views, 2442
pg_visibility, 3007
pg_visible_in_snapshot, 404
pg_waldump, 2335
pg_walfile_name, 412
pg_walfile_name_offset, 412
pg_walinspect, 3009
pg_wal_lsn_diff, 412
pg_wal_replay_pause, 414
pg_wal_replay_resume, 414
pg_xact_commit_timestamp, 406
pg_xact_commit_timestamp_origin, 406
pg_xact_status, 404
pg_xlogdump, 3103 (voir pg_waldump)
phraseto_tsquery, 317, 470
pi, 236
PIC, 1259
PID
  déterminer le PID du processus du serveur
    dans libpq, 963
pipeline
  dans libpq, 987
pipelines
  spécification du protocole, 2454
PITR, 787
PITR standby, 803
pkg-config, 569
  avec ecpg, 1113
  avec libpq, 1024
PL/Perl, 1455
PL/PerlU, 1467
PL/pgSQL, 1371
PL/Python, 1472
PL/SQL
  porter vers PL/pgSQL, 1433
PL/Tcl, 1443
plainto_tsquery, 317, 470
plan de requête, 524
plpgsql.variable_conflict paramètre de configuration, 1426
point, 176, 311
point de sauvegarde
  annulation, 2059
  définition, 2061
  destruction, 2050
points de montage d'un système de fichiers, 595
politique, 81
polygon, 178, 311
popcount (voir bit_count)
popen, 310
populate_record, 2937
port, 952
position, 242, 252, 257
POSITION_REGEX, 277
postgres, 3, 596, 748, 2338
postgresql.auto.conf, 622
postgresql.conf, 622
postgres_fdw, 3011
postmaster, 2346
power, 236
PQbackendPID, 963
PQbinaryTuples, 976
  with COPY, 996
PQcancel, 993
PQclear, 974
PQclientEncoding, 1000
PQcmdStatus, 978
PQcmdTuples, 978
PQconndefaults, 945
PQconnectdb, 942
PQconnectdbParams, 941
PQconnectionNeedsPassword, 963
PQconnectionUsedPassword, 963
PQconnectPoll, 943
PQconnectStart, 943
PQconnectStartParams, 943
PQconninfo, 946
PQconninfoFree, 1003
PQconninfoParse, 946
PQconsumeInput, 985
PQcopyResult, 1004
PQdb, 959
PQdescribePortal, 969
PQdescribePrepared, 968
PQencryptPassword, 1003
PQencryptPasswordConn, 1003
PQendcopy, 999
PQenterPipelineMode, 990
PQerrorMessage, 962
PQescapeBytea, 981, 982
PQescapeIdentifier, 980
PQescapeLiteral, 979
PQescapeString, 981
PQescapeStringConn, 980
PQexec, 965
PQexecParams, 966

PQexecPrepared, 968
PQexitPipelineMode, 990
PQfformat, 975
  with COPY, 996
PQfinish, 947
PQfireResultCreateEvents, 1004
PQflush, 986
PQfmod, 976
PQfn, 994
PQfname, 974
PQfnumber, 974
PQfreeCancel, 993
PQfreemem, 1002
PQfsize, 976
PQftable, 975
PQftablecol, 975
PQftype, 976
PQgetCancel, 992
PQgetCopyData, 997
PQgetisnull, 977
PQgetlength, 977
PQgetline, 998
PQgetlineAsync, 998
PQgetResult, 984
PQgetssl, 965
PQgetSSLKeyPassHook_OpenSSL, 949
PQgetvalue, 976
PQhost, 959
PQhostaddr, 960
PQinitOpenSSL, 1023
PQinitSSL, 1023
PQinstanceData, 1011
PQisBusy, 985
PQisnonblocking, 986
PQisthreadsafe, 1023
PQlibVersion, 1005
  (voir aussi PQserverVersion)
PQmakeEmptyPGresult, 1004
PQnfields, 974
  with COPY, 996
PQnotifies, 995
PQnparams, 977
PQntuples, 974
PQoidStatus, 979
PQoidValue, 979
PQoptions, 960
PQparameterStatus, 961
PQparamtype, 977
PQpass, 959
PQping, 948
PQpingParams, 948
PQpipelineStatus, 989
PQpipelineSync, 990
PQport, 960
PQprepare, 967
PQprint, 978
PQprotocolVersion, 962
PQputCopyData, 996
PQputCopyEnd, 997
PQputline, 999
PQputnbytes, 999
PQregisterEventProc, 1010
PQrequestCancel, 993
PQreset, 947
PQresetPoll, 947
PQresetStart, 947
PQresStatus, 970
PQresultAlloc, 1005
PQresultErrorField, 971
PQresultErrorMessage, 971
PQresultInstanceData, 1011
PQresultMemorySize, 1005
PQresultSetInstanceData, 1011
PQresultStatus, 969
PQresultVerboseErrorMessage, 971
PQsendDescribePortal, 984
PQsendDescribePrepared, 984
PQsendFlushRequest, 990
PQsendPrepare, 983
PQsendQuery, 983
PQsendQueryParams, 983
PQsendQueryPrepared, 984
PQserverVersion, 962
PQsetClientEncoding, 1000
PQsetdb, 943
PQsetdbLogin, 943
PQsetErrorContextVisibility, 1001
PQsetErrorVerbosity, 1000
PQsetInstanceData, 1011
PQsetnonblocking, 986
PQsetNoticeProcessor, 1006
PQsetNoticeReceiver, 1006
PQsetResultAttrs, 1005
PQsetSingleRowMode, 992
PQsetSSLKeyPassHook_OpenSSL, 948
PQsetTraceFlags, 1002
PQsetvalue, 1005
PQsocket, 962
PQsslAttribute, 963
PQsslAttributeNames, 964
PQsslInUse, 963
PQsslStruct, 964
PQstatus, 961
PQtrace, 1001
PQtransactionStatus, 961
PQtty, 960
PQunescapeBytea, 982
PQuntrace, 1002
PQuser, 959
PREPARE, 2036
PREPARE TRANSACTION, 2039
préparer une requête
  en PL/pgSQL, 1427
  en PL/Tcl, 1447
Préparer une requête
  en PL/Python, 1481

procédure
  utilisateur, 1228
procedures
  output parameter, 1236
propriétaire, 77
protocole
  frontend-backend, 2444
ps
  pour surveiller l'activité, 827
psql, 5, 2238
Python, 1472

Q

querytree, 318, 478
quote_ident, 245
  en PL/Perl, 1465
  utilisation dans PL/PgSQL, 1387
quote_literal, 245
  en PL/Perl, 1464
  utilisation dans PL/PgSQL, 1387
quote_nullable, 245
  en PL/Perl, 1464
  utilisation dans PL/PgSQL, 1387

R

radians, 236
radius, 310
RADIUS, 735
RAISE
  en PL/pgSQL, 1414
random, 238
range_agg, 373
range_intersect_agg, 373
range_merge, 370, 371
rank, 378
  hypothetical, 377
rapporter des erreurs
  en PL/PgSQL, 1414
ré-indexation, 784
ré-utilisation
  des identifiants de transaction, 778
read committed, 509
read-only transaction, 693
readline, 562
Readline
  avec psql, 2277
READ_REPLICATION_SLOT, 2464
real, 154
REASSIGN OWNED, 2041
receveur de message, 1006
recherche de texte, 461
  fonctions et opérateurs, 182
Recherche LDAP des paramètres de connexion, 1017
recherche plein texte, 461
  fonctions et opérateurs, 182
  index, 501
  types de données, 182
recherche textuelle
  types de données, 182
record, 226
recovery.conf, 3103, 3103
recovery.signal, 652
rectangle, 177
récupération d'un instantané, 787
RECURSIVE
  dans les CTE, 140
  dans les vues, 1915
référence de colonne, 43
REFRESH MATERIALIZED VIEW, 2042
regclass, 223
regcollation, 223
regconfig, 223
regdictionary, 223
regexp_count, 245, 261
regexp_instr, 245, 261
regexp_like, 246, 261
regexp_match, 246, 261
regexp_matches, 246, 261
regexp_replace, 246, 261
regexp_split_to_array, 246, 261
regexp_split_to_table, 246, 261
regexp_substr, 246, 261
règle, 1338
  pour delete, 1350
  et vues, 1340
  et vues matérialisées, 1347
  pour insert, 1350
  pour select, 1340
  comparée aux triggers, 1365
  pour update, 1350
regnamespace, 223
regoper, 223
regoperator, 223
regproc, 223
regprocedure, 223
regression intercept, 375
régression linéaire, 374
regression slope, 375
regrole, 223
regr_avgx, 375
regr_avgy, 375
regr_count, 375
regr_intercept, 375
regr_r2, 375
regr_slope, 375
regr_sxx, 375
regr_sxy, 375
regr_syy, 375
regtype, 223
regular expressions, 703
REINDEX, 2044
reindexdb, 2285
relation, 7
relation nommée éphémère
  désinscription de SPI, 1530

s'enregistre auprès de SPI, 1529, 1531
RELEASE SAVEPOINT, 2050
répartition de charge, 803
repeat, 247
repeatable read, 510
replace, 247
réplication, 803
Réplication en cascade, 803
Réplication en flux, 803
Réplication Synchrone, 803
requête, 9, 117
requête parallélisée, 548
réseau
types de données, 178
RESET, 2052
restartpoint, 896
RESTRICT
action clé étrangère, 70
with DROP, 111
restrict_nonsystem_relation_kind
configuration parameter, 699
RETURN NEXT
dans PL/PgSQL, 1391
RETURN QUERY
dans PL/PgSQL, 1391
RETURNING, 115
RETURNING INTO
en PL/pgSQL, 1384
reverse, 247
REVOKE, 77, 2053
right, 247
right join, 119
rôle, 739
appartenance, 742
droit à contourner, 741
droit à hériter, 741, 741
droit de contourner, 741
droit de création, 740
droit pour limiter les connexions, 741
possibilité de limiter les connexions, 741
role, 744
applicable, 1167
droit d'initier une réplication, 741
enabled, 1186
ROLLBACK, 2057
rollback
psql, 2273
ROLLBACK PREPARED, 2058
ROLLBACK TO SAVEPOINT, 2059
ROLLUP, 130
round, 237
routine, 1229
routine maintenance, 774
ROW, 54
row-level security, 81
row_number, 378
row_security_active, 394
row_to_json, 341
rpad, 247
rtrim, 247, 253

S

sauvegarde, 410
SAVEPOINT, 2061
scalaire (voir expression)
scale, 237
schéma, 88, 747
courant, 90
créer, 88
public, 89
suppression, 89
schema
current, 389
schéma d'information, 1166
SCRAM, 727
search path
current, 389
SECURITY LABEL, 2063
sec_to_gc, 2926
seg, 3021
SELECT, 9, 117, 2066, 2066
détermination du type de résultat, 443
liste de sélection, 134
SELECT INTO, 2089
en PL/pgSQL, 1384
sélection de champs, 44
sémaphores, 599
sensibilité à la casse
des commandes SQL, 34
sepgsql, 3024
séquence, 357
type serial, 156
serial, 156
serial2, 156
serial4, 156
serial8, 156
serializable, 511
Serializable Snapshot Isolation, 507
Server Name Indication, 957
serveur de standby, 803
serveur témoin, 803
session_user, 391
SET, 408, 2091
SET CONSTRAINTS, 2094
set difference, 135
set intersection, 135
set operation, 135
SET ROLE, 2096
SET SESSION AUTHORIZATION, 2098
SET TRANSACTION, 2100
set union, 135
SET XML OPTION, 697
setseed, 238
setval, 357
setweight, 318, 476
setweight for specific lexeme(s), 318

set_bit, 253, 257
set_byte, 253
set_config, 408
set_limit, 3002
set_masklen, 314
sha224, 253
sha256, 253
sha384, 254
sha512, 254
shared_preload_libraries, 1271
shobj_description, 403
SHOW, 408, 2103, 2462
show_limit, 3002
show_trgm, 3002
SIGHUP, 622, 721, 725
sigint, 608
sign, 237
signal
  processus serveur, 408
sigquit, 608
SIGTERM, 608
SIMILAR TO, 259
similarity, 3002
sin, 239
sind, 239
sinh, 240
skeys, 2935
sleep, 303
slice, 2937
sliced bread (voir TOAST)
slope, 310
slot de réplication
  réplication en flux, 812
  réplication logique, 1565
SLRU, 864
smallint, 151
smallserial, 156
socket de domaine Unix, 952
Solaris
  bibliothèque partagée, 1260
  installation sur, 585
solaris
  script de lancement, 598
SOME, 374, 380, 383
sommes de contrôle, 892
sort, 2943
sort_asc, 2943
sort_desc, 2943
soundex, 2930
sous-chaîne, 259
sous-requête, 52, 123
sous-requêtes, 13
sous-transactions
  dans PL/Tcl, 1452
SP-GiST (voir index)
SPI, 1489
  examples, 3032
spi_commit
  dans PL/Perl, 1464
SPI_commit, 1551
SPI_commit_and_chain, 1551
SPI_connect_ext, 1490
SPI_copytuple, 1544
spi_cursor_close
  en PL/Perl, 1461
SPI_cursor_close, 1526
SPI_cursor_fetch, 1522
SPI_cursor_find, 1521
SPI_cursor_move, 1523
SPI_cursor_open, 1514
SPI_cursor_open_with_args, 1516
SPI_cursor_open_with_paramlist, 1518
SPI_cursor_parse_open, 1519
SPI_exec, 1496
SPI_execp, 1513
SPI_execute, 1492
SPI_execute_extended, 1497
SPI_execute_plan, 1509
SPI_execute_plan_extended, 1510
SPI_execute_plan_with_paramlist, 1512
SPI_execute_with_args, 1499
spi_exec_prepared
  en PL/Perl, 1462
spi_exec_query
  en PL/Perl, 1460
spi_fetchrow
  en PL/Perl, 1461
SPI_finish, 1491
SPI_fname, 1532
SPI_fnumber, 1533
spi_freeplan
  en PL/Perl, 1462
SPI_freeplan, 1550
SPI_freetuple, 1548
SPI_freetuptable, 1549
SPI_getargcount, 1506
SPI_getargtypeid, 1507
SPI_getbinval, 1535
SPI_getnspname, 1539
SPI_getrelname, 1538
SPI_gettype, 1536
SPI_gettypeid, 1537
SPI_getvalue, 1534
SPI_is_cursor_plan, 1508
SPI_keepplan, 1527
SPI_modifytuple, 1546
SPI_palloc, 1541
SPI_pfree, 1543
spi_prepare
  en PL/Perl, 1462
SPI_prepare, 1501
SPI_prepare_cursor, 1503
SPI_prepare_extended, 1504
SPI_prepare_params, 1505
spi_query
  en PL/Perl, 1461

spi_query_prepared
  en PL/Perl, 1462
SPI_register_relation, 1529
SPI_register_trigger_data, 1531
SPI_repalloc, 1542
SPI_result_code_string, 1540
SPI_returntuple, 1545
spi_rollback
  dans PL/Perl, 1464
SPI_rollback, 1552
SPI_rollback_and_chain, 1552
SPI_saveplan, 1528
SPI_scroll_cursor_fetch, 1524
SPI_scroll_cursor_move, 1525
SPI_start_transaction, 1553
SPI_unregister_relation, 1530
split_part, 247
SQL embarqué
  en C, 1049
SQL/CLI, 2733
SQL/Foundation, 2733
SQL/Framework, 2733
SQL/JRT, 2733
SQL/MDA, 2733
SQL/MED, 2733
SQL/OLB, 2733
SQL/PSM, 2733
SQL/Schemata, 2733
SQL/XML, 2733
  limitations et conformité, 2753
sqrt, 237
ssh, 618
SSI, 507
SSL
  avec libpq, 956
  dans libpq, 965
  TLS, 614, 1018
sslinfo, 3035
ssl_cipher, 3035
ssl_client_cert_present, 3035
ssl_client_dn, 3035
ssl_client_dn_field, 3036
ssl_client_serial, 3035
ssl_extension_info, 3036
ssl_issuer_dn, 3035
ssl_issuer_field, 3036
ssl_is_used, 3035
ssl_version, 3035
SSPI, 730
STABLE, 1248
standard deviation, 375
  population, 375
  sample, 375
standby.signal, 652, 808, 809
  pg_basebackup --write-recovery-conf, 2153
  pour un hot standby, 822
standby_mode (voir standby.signal)
START TRANSACTION, 2105
starts_with, 247
START_REPLICATION, 2464
statement_timestamp, 292
statistics, 374
statistiques, 828
  de l'optimiseur, 538, 776
  du planificateur, 536
stats_fetch_consistency configuration parameter, 687
stddev, 375
stddev_pop, 375
stddev_samp, 375
STONITH, 803
strict_word_similarity, 3002
strings
  backslash quotes, 703
  escape warning, 703
  standard conforming, 704
string_agg, 373
string_to_array, 247
string_to_table, 248
strip, 318, 477
strpos, 248
subarray, 2943
subltree, 2954
subpath, 2954
subquery, 380
substr, 248, 254
substring, 242, 252, 257, 261
SUBSTRING_REGEX, 277
sum, 373
superutilisateur, 5, 740
superviser
  activité de la base de données, 827
suppression, 115
suppress_redundant_updates_trigger, 426
surcharge
  fonctions, 1247
svals, 2936
syntaxe
  SQL, 33
syntaxe d'échappement de chaîne, 35
systemd, 571, 597
  RemoveIPC, 604

T

table, 7, 60
  création, 60
  héritage, 93
  modification, 73
  partitionnement, 96
  renommer, 77
  suppression, 61
table d'échelles, 1338
table distante, 110
table partitionnée, 96
tableam
  Méthode d'accès aux tables, 2562
tableau

accès, 201
array, 199
constante, 200
constructeur, 52
déclaration, 199
entrée/sortie, 208
modification, 204
recherche, 206
types utilisateur, 1283
tablefunc, 3037
tableoid, 73
tables de transition, 1894
(voir aussi relation nommée éphémère)
implémentation en PL, 1531
référencées à partir d'un trigger C, 1322
tablespace, 751
par défaut, 692
temporary, 692
table_am_handler, 226
tan, 239
tand, 240
tanh, 240
Tcl, 1443
tcn, 3047
template0, 748, 749
template1, 748, 749
test, 926
tests de régression, 566, 926
test_decoding, 3048
text, 158, 315
text2ltree, 2955
threads
avec libpq, 1023
tid, 223
time, 162
time span, 162
time with time zone, 162
time without time zone, 162
time zone
conversion, 301
time zone names, 700
timelines, 787
TIMELINE_HISTORY, 2463
timeofday, 292
timeout
deadlock, 702
timestamp, 162, 165
timestamp with time zone, 162, 165
timestamp without time zone, 162, 165
timestamptz, 162
TOAST, 2651
paramètres de stockage par type, 1701
réglages de stockage par colonne, 1675
types utilisateur, 1284
versus large objects, 1037
token, 33
to_ascii, 248
to_char, 279
et locales, 755
to_date, 279
to_hex, 248
to_json, 341
to_jsonb, 341
to_number, 279
to_regclass, 400
to_regcollation, 400
to_regnamespace, 400
to_regoper, 400
to_regoperator, 400
to_regproc, 400
to_regprocedure, 400
to_regrole, 400
to_regtype, 400
to_timestamp, 280, 293
to_tsquery, 318, 469
to_tsvector, 318, 468
Tracer la progression de la réplication, 1579
Traces
fonction pg_current_logfile, 390
traces
fichier current_logfiles et la fonction pg_current_logfile, 390
traces serveur, 671
maintenance du fichier de traces, 785
traitement des messages
dans libpq, 1006
traiteur de messages, 1006
transaction, 17
transaction différable
configuration, 2100
valeur par défaut, 693
transaction en lecture seule
configuration, 2100
valeur par défaut, 693
transaction isolation level, 693
Transactions chaînées
dans PL/pgSQL, 1412
transactions chaînées, 1728, 2057
transaction_timestamp, 292
translate, 248
TRANSLATE_REGEX, 277
transparent huge pages, 635
tri, 137
trigger, 226, 1318
arguments pour la fonction trigger, 1321
constraint trigger, 1895
en C, 1322
en PL/Python, 1479
en PL/Tcl, 1449
pour mettre à jour une colonne tsvector dérivée, 480
comparé aux règles, 1365
trigger (trigger)
en PL/pgSQL, 1416
trigger sur événement, 1329
en C, 1334
en PL/Tcl, 1451

triggered_change_notification, 3047
trigger_file (voir promote_trigger_file)
trim, 242, 252
trim_array, 364
trim_scale, 237
true, 172
trunc, 237, 315
TRUNCATE, 2106
trusted
  PL/Perl, 1466
tsm_handler, 226
tsm_system_rows, 3049
tsm_system_time, 3050
tsquery (type de données), 183
tsquery_phrase, 320, 477
tsvector (data type), 182
tsvector_to_array, 321
tsvector_update_trigger, 426
tsvector_update_trigger_column, 426
ts_debug, 321, 496
ts_delete, 319
ts_filter, 319
ts_headline, 319, 474
ts_lexize, 321, 500
ts_parse, 322, 499
ts_rank, 320, 472
ts_rank_cd, 320, 472
ts_rewrite, 320, 478
ts_stat, 322, 481
ts_token_type, 322, 499
tuple_data_split, 2960
txid_current, 405
txid_current_if_assigned, 405
txid_current_snapshot, 405
txid_snapshot_xip, 406
txid_snapshot_xmax, 406
txid_snapshot_xmin, 406
txid_status, 406
txid_visible_in_snapshot, 406
type (voir type de données)
  polymorphe, 1225
type composé, 1224
type composite, 209
  comparaison, 383
  constante, 210
  constructeur, 54
type conteneur, 1224
type de base, 1224
type de données
  base, 1224
  catégorie, 432
  composé, 1224
  constante, 39
  conteneur, 1224
  conversion, 431
  conversion de type, 50
  domaine, 222
  organisation interne, 1251
  utilisateur, 1281
Type de données
  numeric, 150
type de données d'une colonne
  modification, 76
type de ligne, 209
  constructeur, 54
type intervalle de valeur
  exclusion, 221
type intervalle de valeurs, 215
  index sur, 221
type multirange, 215
type polymorphe, 1225
type range, 215
Types de données, 149
types énumérations, 173

U

UESCAPE, 35, 37
unaccent, 3050, 3052
UNION, 135
  détermination du type de résultat, 441
uniq, 2943
unistr, 248
unknown, 226
UNLISTEN, 2109
unnest, 364
  for multirange, 371
  for tsvector, 321
UPDATE, 15, 114, 2111
  RETURNING, 115
upper, 243, 369, 370
  et locales, 754
upper_inc, 370, 370
upper_inf, 370, 371
UPSERT, 2011
URI, 949
user, 391
  current, 390
usurpation de serveur (server spoofing), 612
utilisateur postgres, 593
utilisation des disques, 888
UUID, 185
  génération, 322
uuid-ossp, 3052
uuid_generate_v1, 3053
uuid_generate_v1mc, 3053
uuid_generate_v3, 3053

V

vacuum, 774
VACUUM, 2116
vacuumdb, 2289
vacuumlo, 3065
valeur NULL
  avec contraintes de vérification, 65
  dans libpq, 977

en PL/Perl, 1456
en PL/Python, 1474
avec contrainte d'unicité, 67
valeur par défaut, 61
valeur null
comparaison, 232
dans DISTINCT, 135
valeur par défaut, 61
modifier, 76
validation asynchrone, 893
validation synchrone, 893
VALUES, 138, 2121
détermination du type de résultat, 441
varchar, 158
variable d'environnement, 1014
variance, 375
population, 376
sample, 376
var_pop, 376
var_samp, 376
verrou, 514
informatif, 519
surveillance, 867
verrou informatif, 519
verrou mort, 518
verrouillage de prédicat, 511
version, 6, 391
compatibilité, 609
VM (voir Carte de visibilité)
void, 226
VOLATILE, 1248
volatilité
fonctions, 1248
VPATH, 565, 1316
vue
implémentation par les règles, 1340
matérialisée, 1347
mise à jour, 1355
view, 16
vue matérialisée
implémentation via les règles, 1347
vues matérialisées, 2425
vues modifiables, 1918

W

WAL, 890
warm standby, 803
websearch_to_tsquery, 318
WHERE, 127
WHILE
en PL/pgSQL, 1399
width, 310
width_bucket, 237
WITH
dans SELECT, 139
dans un SELECT, 2066
WITH CHECK OPTION, 1915
WITHIN GROUP, 45
word_similarity, 3002
wrapper de données distantes
gestionnaire, 2523

X

xid, 223
xid8, 223
xmax, 73
xmin, 73
XML, 185
XML export, 334
XML OPTION, 186
xml2, 3054
xmlagg, 327, 373
xmlcomment, 323
xmlconcat, 323
xmlelement, 324
XMLEXISTS, 328
xmlforest, 325
xmlparse, 186
xmlpi, 326
xmlroot, 326
xmlserialize, 186
xmltable, 330
xml_is_well_formed, 328
xml_is_well_formed_content, 328
xml_is_well_formed_document, 328
XPath, 329
xpath_exists, 330
xpath_table, 3055
xslt_process, 3058

Y

yacc, 564

Z

zlib, 563, 572