

The Neo4j Python Driver Manual v6.0

Table of Contents

Quickstart 1
- Install 1
- Connect to the database 1
- Create an example graph 1
- Query a graph 2
- Close connections and sessions 2

Glossary 3

Regular workflow 5
- Installation 6
  - Install the driver 6
  - Get a Neo4j instance 6
- Connection 8
  - Connect to the database 8
  - Connect to an Aura instance 8
  - Connect to a cluster 9
  - Close connections 9
  - Further connection parameters 10
- Query the database 11
  - Write to the database 11
  - Read from the database 11
  - Update the database 12
  - Delete from the database 12
  - Query parameters 12
  - Error handling 13
  - Query configuration 15
    - Database selection 15
    - Request routing 15
    - Run queries as a different user 16
    - Transform query result 16
  - A full example 16
- Manipulate query results 18
  - Result as a list 18
  - Transform to Pandas DataFrame 18
  - Transform to graph 19
  - Custom transformers 20

Advanced usage 22
- Run your own transactions 23
  - Create a session 23

- Run a managed transaction 23
  - Transaction function configuration 25
- Run an explicit transaction 26
- Process query results 27
- Session configuration 28
  - Database selection 28
  - Request routing 29
  - Run queries as a different user 29
- Close sessions 30

Explore the query execution summary 31
- Retrieve the execution summary 31
- Query counters 31
- Query execution plan 32
- Notifications 33
  - Filter notifications 34

Coordinate parallel transactions 36
- Bookmarks with `.execute_query()` 36
- Bookmarks within a single session 36
- Bookmarks across multiple sessions 36
- Mix `.execute_query()` and sessions 38
- Implement a custom `BookmarkManager` 38

Run concurrent transactions 39

Further query mechanisms 41
- Implicit (or auto-commit) transactions 41
  - Import CSV files 41
  - Transaction configuration 42
- Dynamic values in property keys, relationship types, and labels 42
- Logging 43

Performance recommendations 45
- Use the Rust extension 45
- Always specify the target database 45
- Be aware of the cost of transactions 45
- Don’t fetch large result sets all at once 46
- Route read queries to cluster readers 49
- Create indexes 49
- Profile queries 50
- Specify node labels 50
- Batch data creation 50
- Use query parameters 51
- Concurrency 51
- Use `MERGE` for creation only when needed 51

Filter notifications 52
Reference 53
Advanced connection information 54
Connection URI 54
Connection protocols and security 54
Authentication methods 54
Basic authentication (default) 54
Kerberos authentication 55
Bearer authentication 55
Custom authentication 55
No authentication 55
Rotating authentication tokens 55
Mutual TLS (client-side certificates as 2FA) 56
Custom address resolver 58
Disable telemetry 58
Further connection parameters 58
Data types and mapping to Cypher types 59
Core types 59
Temporal types 59
Date 60
Time 60
LocalTime 61
DateTime 61
LocalDateTime 61
Duration 61
Spatial types 62
CartesianPoint 62
WGS84Point 62
Graph types 63
Node 63
Relationship 63
Path 64
Extended types 65
Vector type 65
UUID 67
UnsupportedType 67
Upgrade from older versions 68
New features 68
Breaking changes and deprecations 68
API documentation 71
GraphAcademy courses 72

Graph Data Modeling Fundamentals 73
Intermediate Cypher Queries 74
Building Neo4j Applications with Python 75

Quickstart

The Neo4j Python driver is the official library to interact with a Neo4j instance through a Python application.

At the hearth of Neo4j lies Cypher, the query language to interact with a Neo4j database. Although this guide does not *require* you to be a seasoned Cypher querier, it's easier to focus on the Python-specific bits if you know some Cypher already. You will also get a *gentle* introduction to Cypher in these pages, but check out Getting started → Cypher for a more detailed walkthrough of graph databases modelling and querying if this is your first approach.

Install

Install the Neo4j Python driver with `pip`:

```
pip install neo4j
```

More info on installing the driver

Connect to the database

Connect to a database by creating a `Driver` object and providing a URL and an authentication token. Once you have a `Driver` instance, use the `.verify_connectivity()` method to ensure that a working connection can be established.

```
from neo4j import GraphDatabase

# URI examples: "neo4j://localhost", "neo4j+s://xxx.databases.neo4j.io"
URI = "<database-uri>"
AUTH = ("<username>", "<password>")

with GraphDatabase.driver(URI, auth=AUTH) as driver:
    driver.verify_connectivity()
```

More info on connecting to a database

Create an example graph

Run a Cypher query with the method `Driver.execute_query()`. Do not hardcode or concatenate parameters: use placeholders and specify the parameters as keyword arguments.

Create two `Person` *nodes and a* `KNOWS` *relationship between them*

```
summary = driver.execute_query("""
    CREATE (a:Person {name: $name})
    CREATE (b:Person {name: $friendName})
    CREATE (a)-[:KNOWS]->(b)
    """,
    name="Alice", friendName="David",
    database_="<database-name>",
).summary
print("Created {nodes_created} nodes in {time} ms.".format(
```

```
    nodes_created=summary.counters.nodes_created,
    time=summary.result_available_after
))
```

More info on querying the database

Query a graph

To retrieve information from the database, use the Cypher clause `MATCH`:

Retrieve all `Person` *nodes who know other persons*

```
records, summary, keys = driver.execute_query("""
    MATCH (p:Person)-[:KNOWS]->(:Person)
    RETURN p.name AS name
    """,
    database_="<database-name>",
)

# Loop through results and do something with them
for record in records:
    print(record.data())  # obtain record as dict

# Summary information
print("The query `{query}` returned {records_count} records in {time} ms.".format(
    query=summary.query, records_count=len(records),
    time=summary.result_available_after
))
```

More info on querying the database

Close connections and sessions

Unless you created them using the `with` statement, call the `.close()` method on all `Driver` and `Session` instances to release any resources still held by them.

```
from neo4j import GraphDatabase


driver = GraphDatabase.driver(URI, auth=AUTH)
session = driver.session(database="<database-name>")

# session/driver usage

session.close()
driver.close()
```

Glossary

LTS

A *Long Term Support* release is one guaranteed to be supported for a number of years. Neo4j 4.4 and 5.26 are LTS versions.

Aura

Aura is Neo4j's fully managed cloud service. It comes with both free and paid plans.

Cypher

Cypher is Neo4j's graph query language that lets you retrieve data from the database. It is like SQL, but for graphs.

APOC

Awesome Procedures On Cypher (APOC) is a library of (many) functions that can not be easily expressed in Cypher itself.

Bolt

Bolt is the protocol used for interaction between Neo4j instances and drivers. It listens on port 7687 by default.

ACID

Atomicity, Consistency, Isolation, Durability (ACID) are properties guaranteeing that database transactions are processed reliably. An ACID-compliant DBMS ensures that the data in the database remains accurate and consistent despite failures.

eventual consistency

A database is eventually consistent if it provides the guarantee that all cluster members will, *at some point in time*, store the latest version of the data.

causal consistency

A database is causally consistent if read and write queries are seen by every member of the cluster in the same order. This is stronger than *eventual consistency*.

NULL

The null marker is not a type but a placeholder for absence of value. For more information, see Cypher → Working with `null`.

transaction

A transaction is a unit of work that is either *committed* in its entirety or *rolled back* on failure. An example is a bank transfer: it involves multiple steps, but they must *all* succeed or be reverted, to avoid money being subtracted from one account but not added to the other.

backpressure

Backpressure is a force opposing the flow of data. It ensures that the client is not being overwhelmed by data faster than it can handle.

bookmark

A *bookmark* is a token representing some state of the database. By passing one or multiple bookmarks along with a query, the server will make sure that the query does not get executed before the represented state(s) have been established.

transaction function

A transaction function is a callback executed by an `execute_read` or `execute_write` call. The driver automatically re-executes the callback in case of server failure.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database.

Regular workflow

Installation

To start creating a Neo4j Python application, you first need to install the Python Driver and get a Neo4j database instance to connect to.

> *Server compatibility*
>
> The latest driver in the `6.x` series supports connection to Neo4j instances version 4.4.x, 5.x, 2025.x, and 2026.x. It is also guaranteed to be forward-compatible with the next major release.

Install the driver

Use `pip` to install the Neo4j Python Driver (requires Python >= 3.10):

```
pip install neo4j
```

▼ *Activate Python's development mode*

For development, running Python in development mode could help you uncover API misusages and performance leaks early on. You can do so via the `-X dev` command line option or by setting the `PYTHONDEVMODE` environment variable to `1`.
For the driver, this results in:

- `ResourceWarning` being emitted if resources aren't properly closed.
- `DeprecationWarning` being emitted if deprecated APIs are used.
- The driver's debug mode to be enabled (achieved also by setting the environment variable `PYTHONNEO4JDEBUG`).

> The Rust extension to the Python driver is an alternative package that yields a 3x to 10x speedup compared to the regular driver. You can install it with `pip install neo4j-rust-ext`, either alongside the `neo4j` package or as a replacement to it. Usage-wise, the libraries are identical: everything in this guide applies to both.

> To get the driver on an air-gapped machine, download the latest driver tarball and install it with `pip install neo4j-<version>.tar.gz`.

Get a Neo4j instance

You need a running Neo4j database in order to use the driver with it. The easiest way to spin up a **local instance** is through a Docker container (requires `docker.io`). The command below runs the latest Neo4j version in Docker, setting the admin username to `neo4j` and password to `secretgraph`:

```
docker run \
   -p7474:7474 \                    # forward port 7474 (HTTP)
```

```
    -p7687:7687 \                       # forward port 7687 (Bolt)
    -d \                                # run in background
    -e NEO4J_AUTH=neo4j/secretgraph \   # set login credentials
    neo4j:latest
```

Alternatively, you can obtain a free **cloud instance** through Aura.

You can also install Neo4j on your system, or use Neo4j Desktop to create a local development environment (not for production).

Connection

Once you have installed the driver and have a running Neo4j instance, you are ready to connect your application to the database.

Connect to the database

You connect to a database by creating a Driver object and providing a URL and your login credentials.

```
from neo4j import GraphDatabase

# URI examples: "neo4j://localhost", "neo4j+s://xxx.databases.neo4j.io"
URI = "<database-uri>"
AUTH = ("<username>", "<password>")

with GraphDatabase.driver(URI, auth=AUTH) as driver:
    driver.verify_connectivity()
    print("Connection established.")
```

> Creating a `Driver` instance only provides information on *how* to access the database, but does not actually *establish* a connection. Connection is instead deferred to when the first query is executed.
>
> To verify immediately that the driver can connect to the database (valid credentials, compatible versions, etc), use the `.verify_connectivity()` method after initializing the driver. In case of failure, enabling the driver's logs can help diagnosing the issue.

Both the creation of a `Driver` object and the connection verification can raise a number of different exceptions. Since error handling can get quite verbose, and a connection error is a blocker for any subsequent task, a common choice is to let the program crash should an exception occur during connection.

`Driver` objects are immutable, thread-safe, and expensive to create, so your application should create only one instance and pass it around (you can share `Driver` instances across threads). If you need to query the database through several different users, use impersonation without creating a new `Driver` instance. If you want to alter a `Driver` configuration, you need to create a new object.



The driver also supports other authentication methods (kerberos, bearer, custom).

Connect to an Aura instance

When you create an Aura instance, you get to download a text file (a so-called *Dotenv file*) containing the connection information to the database as environment variables. The file has a name of the form `Neo4j-a0a2fa1d-Created-2023-11-06.txt`.

You can either manually extract the URI and the credentials from that file, or use a third party-package (ex. `python-dotenv`) to load them.

```
import dotenv
import os
from neo4j import GraphDatabase
```

```
load_status = dotenv.load_dotenv("Neo4j-a0a2fa1d-Created-2023-11-06.txt")
if load_status is False:
    raise RuntimeError('Environment variables not loaded.')

URI = os.getenv("NEO4J_URI")
AUTH = (os.getenv("NEO4J_USERNAME"), os.getenv("NEO4J_PASSWORD"))

with GraphDatabase.driver(URI, auth=AUTH) as driver:
    driver.verify_connectivity()
    print("Connection established.")
```



An Aura instance is not conceptually different from any other Neo4j instance, as Aura is simply a *deployment mode* for Neo4j. When interacting with a Neo4j database through the driver, it doesn't make a difference whether it is an Aura instance it is working with or a different deployment.

Connect to a cluster

When working with a Neo4j cluster, you have a few options to connect to it, depending on how it's deployed.

Upon connection, the driver gets a *routing table* containing information about all the machines in the cluster (including their roles), and will point requests to primaries or secondaries as needed. The routing table is periodically refreshed.

- If the cluster members are all discoverable via a hostname that provides multiple `A`/`AAAA` records, you can use the hostname and port.
- If the addresses of the cluster members are known, but not advertised via `A`/`AAAA` records, you can use a Custom address resolver to extend the DNS resolution on the client side.

  ```
  from neo4j import GraphDatabase


  def custom_resolver(socket_address):
      yield "server01.example.com", 7687
      yield "server02.example.com", 7687
      yield "server03.example.com", 7687
      yield "backup.example.org", 7687


  driver = GraphDatabase.driver(
      "neo4j://example.com:7687", auth=(USERNAME, PASSWORD),
      resolver=custom_resolver)
  ```

- You can also use the address of any machine belonging to the cluster: the driver will discover the others. This is the least fail-safe option, because if the machine you use as entrypoint becomes unavailable, the driver won't be able to fetch a routing table.

You can also bypass the routing table and connect to a specific machine in the cluster using its address together with the `bolt[+s[sc]]://` scheme (see connection URI for more information).

Close connections

Always close `Driver` objects to free up all allocated resources, even upon unsuccessful connection or

runtime errors. Either instantiate the `Driver` object using the `with` statement, or call the `Driver.close()` method explicitly.

Further connection parameters

For more `Driver` configuration parameters and further connection settings, see Advanced connection information.

Query the database

Once you have connected to the database, you can run queries using Cypher and the method `Driver.execute_query()`.

Write to the database

To create two nodes representing persons named `Alice` and `David`, and a relationship `KNOWS` between them, use the Cypher clause `CREATE`:

Create two nodes and a relationship

```
summary = driver.execute_query("""
    CREATE (a:Person {name: $name})
    CREATE (b:Person {name: $friendName})
    CREATE (a)-[:KNOWS]->(b)
    """,
    name="Alice", friendName="David",
    database_="<database-name>",
).summary
print("Created {nodes_created} nodes in {time} ms.".format(
    nodes_created=summary.counters.nodes_created,
    time=summary.result_available_after
))
```

The Cypher query

The *query parameters*, as keyword arguments

The database to run the query on

Read from the database

To retrieve information from the database, use the Cypher clause `MATCH`:

Retrieve all `Person` *nodes who like other* `Person` *s*

```
records, summary, keys = driver.execute_query("""
    MATCH (p:Person)-[:KNOWS]->(:Person)
    RETURN p.name AS name
    """,
    database_="<database-name>",
)

# Loop through results and do something with them
for record in records:
    print(record.data())  # get record as dict

# Summary information
print("The query `{query}` returned {records_count} records in {time} ms.".format(
    query=summary.query, records_count=len(records),
    time=summary.result_available_after
))
```

`records` contains the result as an array of `Record` objects

`summary` contains the summary of execution returned by the server

Update the database

To update an entity's information in the database, use the Cypher clauses `SET`:

Update node `Alice` to add an `age` property

```
records, summary, keys = driver.execute_query("""
    MATCH (p:Person {name: $name})
    SET p.age = $age
    """, name="Alice", age=42,
    database_="<database-name>",
)
print(f"Query counters: {summary.counters}.")
```

To create a new relationship, linking it to two already existing node, use a combination of the Cypher clauses `MATCH` and `CREATE`:

Create a relationship `:KNOWS` between `Alice` and `Bob`

```
records, summary, keys = driver.execute_query("""
    MATCH (alice:Person {name: $name})
    MATCH (bob:Person {name: $friend})
    CREATE (alice)-[:KNOWS]->(bob)
    """, name="Alice", friend="Bob",
    database_="<database-name>",
)
print(f"Query counters: {summary.counters}.")
```

Retrieve the person node named `Alice` and bind it to a variable `alice`

Retrieve the person node named `Bob` and bind it to a variable `bob`

Create a new `:KNOWS` relationship outgoing from the node bound to `alice` and attach to it the `Person` node named `Bob`

Delete from the database

To remove a node and any relationship attached to it, use the Cypher clause `DETACH DELETE`:

Remove the `Alice` node

```
# This does not delete _only_ p, but also all its relationships!
records, summary, keys = driver.execute_query("""
    MATCH (p:Person {name: $name})
    DETACH DELETE p
    """, name="Alice",
    database_="<database-name>",
)
print(f"Query counters: {summary.counters}.")
```

Query parameters

Do not hardcode or concatenate parameters directly into queries. Instead, always use placeholders and provide dynamic data as Cypher parameters. This is for:

1. **performance benefits:** Neo4j compiles and caches queries, but can only do so if the query structure is unchanged;

2. **security reasons**: see protecting against Cypher injection.

Query parameters can be passed either as several keyword arguments, or grouped together in a dictionary passed as value to the `parameters_` keyword argument. In case of mix, keyword-argument parameters take precedence over dictionary ones.

Pass query parameters as keyword arguments

```
driver.execute_query(
    "MERGE (:Person {name: $name})",
    name="Alice", age=42,
    database_="<database-name>",
)
```

Pass query parameters in a dictionary

```
parameters = {
    "name": "Alice",
    "age": 42
}
driver.execute_query(
    "MERGE (:Person {name: $name})",
    parameters_=parameters,
    database_="<database-name>",
)
```

None of your keyword query parameters may end with a single underscore. This is to avoid collisions with the keyword configuration parameters. If you need to use such parameter names, pass them in the `parameters_` dictionary.



There can be circumstances where your query structure prevents the usage of parameters in all its parts. For those rare use cases, see Dynamic values in property keys, relationship types, and labels.

Error handling

A query run may fail for a number of reasons, with different exceptions being raised. When using `driver.execute_query()`, the driver automatically retries to run a failed query if the failure is deemed to be transient (for example due to temporary server unavailability).

An exception will be raised if the operation keeps failing after the configured maximum retry time.

All exceptions coming from the server are subclasses of `Neo4jError`. You can use an exception's code to stably identify a specific error; error messages are instead not stable markers, and should not be relied upon.

Basic error handling

```
# from neo4j.exceptions import Neo4jError

try:
    driver.execute_query('MATCH (p:Person) RETURN', database_='<database-name>')
except Neo4jError as e:
    print('Neo4j error code:', e.code)
    print('Exception message:', e.message)
'''
```

```
Neo4j error code: Neo.ClientError.Statement.SyntaxError
Exception message: Invalid input '': expected an expression, '*', 'ALL' or 'DISTINCT' (line 1, column 24
(offset: 23))
"MATCH (p:Person) RETURN"
                        ^
'''
```

Exception objects also expose errors as GQL-status objects. The main difference between Neo4j error codes and GQL error codes is that the GQL ones are more granular: a single Neo4j error code might be broken in several, more specific GQL error codes.

The actual *cause* that triggered an exception is sometimes found in the optional GQL-status object `__cause__`, which is itself an exception (or `None`). You might need to recursively traverse the cause chain before reaching the root cause of the exception you caught. In the example below, the exception's GQL status code is `42001`, but the actual source of the error has status code `42I06`.

Usage of `Neo4jError` with GQL-related methods

```
# from neo4j.exceptions import Neo4jError

try:
    driver.execute_query('MATCH (p:Person) RETURN', database_='<database-name>')
except Neo4jError as e:
    print('Exception GQL status:', e.gql_status)
    print('Exception GQL status description:', e.gql_status_description)
    print('Exception GQL classification:', e.gql_classification)
    print('Exception GQL cause:', e.__cause__)
    print('Exception GQL diagnostic record:', e.diagnostic_record)
'''
Exception GQL status: 42001
Exception GQL status description: error: syntax error or access rule violation - invalid syntax
Exception GQL classification: GqlErrorClassification.CLIENT_ERROR
Exception GQL cause: {gql_status: 42I06} {gql_status_description: error: syntax error or access rule
violation - invalid input. Invalid input '', expected: an expression, '*', 'ALL' or 'DISTINCT'.} {message:
42I06: Invalid input '', expected: an expression, '*', 'ALL' or 'DISTINCT'.} {diagnostic_record:
{'_classification': 'CLIENT_ERROR', '_position': {'line': 1, 'column': 24, 'offset': 23}, 'OPERATION': '',
'OPERATION_CODE': '0', 'CURRENT_SCHEMA': '/'}} {raw_classification: CLIENT_ERROR}
Exception GQL diagnostic record: {'_classification': 'CLIENT_ERROR', '_position': {'line': 1, 'column':
24, 'offset': 23}, 'OPERATION': '', 'OPERATION_CODE': '0', 'CURRENT_SCHEMA': '/'}
'''
```

GQL status codes are particularly helpful when you want your application to behave differently depending on the exact error that was raised by the server.

Distinguishing between different error codes

```
# from neo4j.exceptions import Neo4jError

try:
    driver.execute_query('MATCH (p:Person) RETURN', database_='<database-name>')
except Neo4jError as e:
    if e.find_by_gql_status('42001'):
        # Neo.ClientError.Statement.SyntaxError
        # special handling of syntax error in query
        print(e.message)
    elif e.find_by_gql_status('42NFF'):
        # Neo.ClientError.Security.Forbidden
        # special handling of user not having CREATE permissions
        print(e.message)
    else:
        # handling of all other exceptions
        print(e.message)
```



Note:

The GQL status code `50N42` is returned when an exception does not have a GQL-status object. This can happen if the driver is connected to an older Neo4j server. Don't rely on this status code, as future Neo4j server versions might change it with a more appropriate one.



Tip:

Transient server errors can be retried without need to alter the original request. You can discover whether an error is transient via the method `Neo4jError.is_retryable()`, which gives insights into whether a further attempt might be successful. This is particular useful when running queries in explicit transactions, to know if a failed query is worth re-running.

Query configuration

You can supply further keyword arguments to alter the default behavior of `.execute_query()`. Configuration parameters are suffixed with `_`.

Database selection

Always specify the database explicitly with the `database_` parameter, even on single-database instances. This allows the driver to work more efficiently, as it saves a network round-trip to the server to resolve the home database. If no database is given, the user's home database is used.

```
driver.execute_query(
    "MATCH (p:Person) RETURN p.name",
    database_="<database-name>",
)
```



Specifying the database through the configuration method is preferred over the `USE` Cypher clause. If the server runs on a cluster, queries with `USE` require server-side routing to be enabled. Queries can also take longer to execute as they may not reach the right cluster member at the first attempt, and need to be routed to one containing the requested database.

Request routing

In a cluster environment, all queries are directed to the leader node by default. To improve performance on read queries, you can use the argument `routing_="r"` to route a query to the read nodes.

```
driver.execute_query(
    "MATCH (p:Person) RETURN p.name",
    routing_="r",  # short for neo4j.RoutingControl.READ
    database_="<database-name>",
)
```



Note:

Although executing a *write* query in read mode results in a runtime error, **you should not**

rely on this for access control. The difference between the two modes is that *read* transactions will be routed to any node of a cluster, whereas *write* ones are directed to primaries. There is no security guarantee that a write query submitted in read mode will be rejected.

Run queries as a different user

You can execute a query through a different user with the parameter `auth_`. Switching user at the query level is cheaper than creating a new `Driver` object. The query is then run within the security context of the given user (i.e., home database, permissions, etc.).

```
driver.execute_query(
    "MATCH (p:Person) RETURN p.name",
    auth_=("<username>", "<password>"),
    database_="<database-name>",
)
```

The parameter `impersonated_user_` provides a similar functionality. The difference is that you don't need to know a user's password to impersonate them, but the user under which the `Driver` was created needs to have the appropriate permissions.

```
driver.execute_query(
    "MATCH (p:Person) RETURN p.name",
    impersonated_user_="<username>",
    database_="<database-name>",
)
```

Transform query result

You can transform a query's result into a different data structure using the `result_transformer_` argument. The driver provides built-in methods to transform the result into a pandas dataframe or into a graph, but you can also craft your own transformer.

For more information, see Manipulate query results.

A full example

```
from neo4j import GraphDatabase
from neo4j.exceptions import Neo4jError


URI = "<database-uri>"
AUTH = ("<username>", "<password>")

people = [{"name": "Alice", "age": 42, "friends": ["Bob", "Peter", "Anna"]},
          {"name": "Bob", "age": 19},
          {"name": "Peter", "age": 50},
          {"name": "Anna", "age": 30}]

with GraphDatabase.driver(URI, auth=AUTH) as driver:
    try:
        # Create some nodes
        for person in people:
            records, summary, keys = driver.execute_query(
                "MERGE (p:Person {name: $person.name, age: $person.age})",
                person=person,
```

```
                database_="<database-name>",
            )

        # Create some relationships
        for person in people:
            if person.get("friends"):
                records, summary, keys = driver.execute_query("""
                    MATCH (p:Person {name: $person.name})
                    UNWIND $person.friends AS friend_name
                    MATCH (friend:Person {name: friend_name})
                    MERGE (p)-[:KNOWS]->(friend)
                    """, person=person,
                    database_="<database-name>",
                )

        # Retrieve Alice's friends who are under 40
        records, summary, keys = driver.execute_query("""
            MATCH (p:Person {name: $name})-[:KNOWS]-(friend:Person)
            WHERE friend.age < $age
            RETURN friend
            """, name="Alice", age=40,
            routing_="r",
            database_="<database-name>",
        )
        # Loop through results and do something with them
        for record in records:
            print(record)
        # Summary information
        print("The query `{query}` returned {records_count} records in {time} ms.".format(
            query=summary.query, records_count=len(records),
            time=summary.result_available_after
        ))

    except Neo4jError as e:
        print(e)
        # further logging/processing
```

For more information see API documentation → Driver.execute_query().

Manipulate query results

This section shows how to work with a query's result so as to extract data in the form that is most convenient for your application.

Result as a list

By default, `Driver.execute_query()` returns an `EagerResult` object.

```
records, summary, keys = driver.execute_query(
    "MATCH (a:Person) RETURN a.name AS name, a.age AS age",
    database_="<database-name>",
)
for person in records:
    print(person)
    # person["name"] or person["age"] are also valid

# Some summary information
print("Query `{query}` returned {records_count} records in {time} ms.".format(
    query=summary.query, records_count=len(records),
    time=summary.result_available_after
))

print(f"Available keys are {keys}")  # ['name', 'age']
```

The result records come in a list.

A summary of execution, with metadata and information about the result.

The `keys` available in the returned rows.

Transform to Pandas DataFrame

The driver can transform the result into a Pandas DataFrame by setting `result_transformer_=neo4j.Result.to_df`. This method requires the `pandas` library is installed.

Return a DataFrame with two columns (`n`, `m`) and 10 rows

```
import neo4j

pandas_df = driver.execute_query(
    "UNWIND range(1, 10) AS n RETURN n, n+1 AS m",
    database_="<database-name>",
    result_transformer_=neo4j.Result.to_df
)
print(type(pandas_df))  # <class 'pandas.core.frame.DataFrame'>
```

This transformer accepts two optional arguments:

- `expand` — If `True`, some data structures in the result will be recursively expanded and flattened. More info in the API documentation.
- `parse_dates` — If `True`, columns exclusively containing `time.DateTime` objects, `time.Date` objects, or `None`, will be converted to `pandas.Timestamp`.



To pass parameters to `to_df`, use lambda functions:

```
        result_transformer_=lambda res: res.to_df(True)
```

Transform to graph

The driver can transform the result into a collection of graph objects by setting `result_transformer_=neo4j.Result.graph`. To make the most out of this method, your query should return a graph-like result instead of a single column. The graph transformer returns a `Graph` object exposing the properties `nodes` and `relationships`, which are set views into `Node` and `Relationship` objects.

You can use the graph format for further processing or to visualize the query result. An example implementation that uses the `pyvis` library to draw the graph is below.

Visualize graph result with `pyvis`

```
import pyvis
from neo4j import GraphDatabase
import neo4j


URI = "<database-uri>"
AUTH = ("<username>", "<password>")


def main():
    with GraphDatabase.driver(URI, auth=AUTH) as driver:
        # Create some friends
        input_list = [("Arthur", "Guinevre"),
                      ("Arthur", "Lancelot"),
                      ("Arthur", "Merlin")]
        driver.execute_query("""
            UNWIND $pairs AS pair
            MERGE (a:Person {name: pair[0]})
            MERGE (a)-[:KNOWS]->(friend:Person {name: pair[1]})
            """, pairs=input_list,
            database_="<database-name>",
        )

        # Create a film
        driver.execute_query("""
            MERGE (film:Film {title: $title})
            MERGE (liker:Person {name: $person_name})
            MERGE (liker)-[:LIKES]->(film)
            """, title="Wall-E", person_name="Arthur",
            database_="<database-name>",
        )

        # Query to get a graphy result
        graph_result = driver.execute_query("""
            MATCH (a:Person {name: $name})-[r]-(b)
            RETURN a, r, b
            """, name="Arthur",
            result_transformer_=neo4j.Result.graph,
        )

        # Draw graph
        nodes_text_properties = {  # what property to use as text for each node
            "Person": "name",
            "Film": "title",
        }
        visualize_result(graph_result, nodes_text_properties)


def visualize_result(query_graph, nodes_text_properties):
    visual_graph = pyvis.network.Network()

    for node in query_graph.nodes:
        node_label = list(node.labels)[0]
        node_text = node[nodes_text_properties[node_label]]
        visual_graph.add_node(node.element_id, node_text, group=node_label)
```

```
    for relationship in query_graph.relationships:
        visual_graph.add_edge(
            relationship.start_node.element_id,
            relationship.end_node.element_id,
            title=relationship.type
        )

    visual_graph.show('network.html', notebook=False)


if __name__ == "__main__":
    main()
```

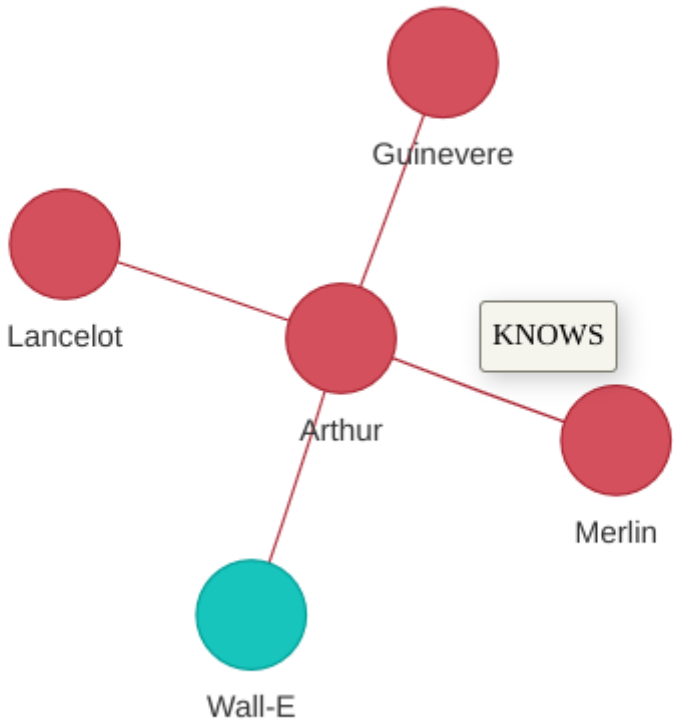


Figure 1. Graph visualization of example above

Custom transformers

For more advanded scenarios, you can use the parameter `result_transformer_` to provide a custom function that further manipulates the `Result` object resulting from your query. A transformer takes a `Result` object and can output any data structure. The transformer's return value is in turn returned by `.execute_query()`.

Inside a transformer function you can use any of the `Result` methods.

A custom transformer using `single` and `consume`

```
# Get a single record (or an exception) and the summary from a result.
def get_single_person(result):
    record = result.single(strict=True)
    summary = result.consume()
    return record, summary
```

```
record, summary = driver.execute_query(
    "MERGE (a:Person {name: $name}) RETURN a.name AS name",
    name="Alice",
    database_="<database-name>",
    result_transformer_=get_single_person,
)
print("The query `{query}` returned {record} in {time} ms.".format(
      query=summary.query, record=record, time=summary.result_available_after))
```

A *custom transformer using* `fetch` *and* `peek`

```
# Get exactly 5 records, or an exception.
def exactly_5(result):
    records = result.fetch(5)

    if len(records) != 5:
        raise Exception(f"Expected exactly 5 records, found only {len(records)}.")
    if result.peek():
        raise Exception("Expected exactly 5 records, found more.")

    return records


records = driver.execute_query("""
    UNWIND ['Alice', 'Bob', 'Laura', 'John', 'Patricia'] AS name
    MERGE (a:Person {name: name}) RETURN a.name AS name
    """, database_="<database-name>",
    result_transformer_=exactly_5,
)
```



Warning:

A transformer **must not** return the `Result` object itself. Doing so is roughly equivalent to returning a *pointer* to the result buffer, which gets invalidated as soon as the query's transaction is over.

```
def transformer(result):
    return result

result = driver.execute_query(
    "MATCH (a:Person) RETURN a.name",
    result_transformer_=transformer)
print(result)
print(result.single())
```

```
neo4j.exceptions.ResultConsumedError: The result is out of scope.
The associated transaction has been closed.
Results can only be used while the transaction is open.
```

Advanced usage

Run your own transactions

When querying the database with `execute_query()`, the driver automatically creates a *transaction*. A transaction is a unit of work that is either *committed* in its entirety or *rolled back* on failure. You can include multiple Cypher statements in a single query, as for example when using `MATCH` and `CREATE` in sequence to update the database, but you cannot have multiple queries and interleave some client-logic in between them.

For these more advanced use-cases, the driver provides functions to manually control transactions. The most common form is *managed transactions*, and you can think of them as a way of unwrapping the flow of `.execute_query()` and being able to specify its desired behavior in more places.

Create a session

Before running a transaction, you need to obtain a *session*. Sessions act as query channels between the driver and the server, and ensure causal consistency is enforced.

Sessions are created with the method `Driver.session()`, with the keyword argument `database` allowing to specify the target database. For further parameters, see Session configuration.

```
with driver.session(database="<database-name>") as session:
    ...
```

Creating a session is a lightweight operation, so sessions can be created and destroyed without significant cost. Always close sessions when you are done with them.

Sessions are *not* thread safe: you can share the main `Driver` object across threads, but each thread should create its own sessions.

Run a managed transaction

A transaction can contain multiple queries. As Neo4j is ACID compliant, **queries within a transaction will either be executed as a whole or not at all**: you cannot get a part of the transaction succeeding and another failing. Use transactions to group together related queries which work together to achieve a single *logical* database operation.

You create a managed transaction with the methods `Session.execute_read()` and `Session.execute_write()`, depending on whether you want to retrieve data from the database or alter it. Both methods take a *transaction function* callback, which is responsible for carrying out the queries and processing the result.

Retrieve people whose name starts with `Al`.

```
def match_person_nodes(tx, name_filter):
    result = tx.run("""
        MATCH (p:Person) WHERE p.name STARTS WITH $filter
        RETURN p.name AS name ORDER BY name
        """, filter=name_filter)
    return list(result)  # a list of Record objects

with driver.session(database="<database-name>") as session:
```

```
    people = session.execute_read(
        match_person_nodes,
        "Al",
    )
    for person in people:
        print(person.data())  # obtain dict representation
```

Create a session. A single session can be the container for multiple queries. Unless created using the `with` construct, remember to close it when done.

The `.execute_read()` (or `.execute_write()`) method is the entry point into a transaction. It takes a callback to a *transaction function* and an arbitrary number of positional and keyword arguments which are handed down to the transaction function.

The transaction function callback is responsible of running queries.

Use the method `Transaction.run()` to run queries. Each query run returns a `Result` object.

Process the result using any of the methods on `Result`.

Do not hardcode or concatenate parameters directly into the query. Use query parameters instead, both for performance and security reasons.

Transaction functions should never return the `Result` object directly. Instead, always process the result in some way; at minimum, cast it to list. Within a transaction function, a `return` statement results in the transaction being committed, while the transaction is automatically rolled back if an exception is raised.

A transaction with multiple queries, client logic, and potential roll backs

```
from neo4j import GraphDatabase


URI = "<database-uri>"
AUTH = ("<username>", "<password>")
employee_threshold=10


def main():
    with GraphDatabase.driver(URI, auth=AUTH) as driver:
        with driver.session(database="<database-name>") as session:
            for i in range(100):
                name = f"Thor{i}"
                org_id = session.execute_write(employ_person_tx, name)
                print(f"User {name} added to organization {org_id}")


def employ_person_tx(tx, name):
    # Create new Person node with given name, if not exists already
    result = tx.run("""
        MERGE (p:Person {name: $name})
        RETURN p.name AS name
        """, name=name
    )

    # Obtain most recent organization ID and the number of people linked to it
    result = tx.run("""
        MATCH (o:Organization)
        RETURN o.id AS id, COUNT{(p:Person)-[r:WORKS_FOR]->(o)} AS employees_n
        ORDER BY o.created_date DESC
        LIMIT 1
    """)
    org = result.single()

    if org is not None and org["employees_n"] == 0:
        raise Exception("Most recent organization is empty.")
        # Transaction will roll back -> not even Person is created!

    # If org does not have too many employees, add this Person to that
```

```
    if org is not None and org.get("employees_n") < employee_threshold:
        result = tx.run("""
            MATCH (o:Organization {id: $org_id})
            MATCH (p:Person {name: $name})
            MERGE (p)-[r:WORKS_FOR]->(o)
            RETURN $org_id AS id
            """, org_id=org["id"], name=name
        )

    # Otherwise, create a new Organization and link Person to it
    else:
        result = tx.run("""
            MATCH (p:Person {name: $name})
            CREATE (o:Organization {id: randomuuid(), created_date: datetime()})
            MERGE (p)-[r:WORKS_FOR]->(o)
            RETURN o.id AS id
            """, name=name
        )

    # Return the Organization ID to which the new Person ends up in
    return result.single()["id"]


if __name__ == "__main__":
    main()
```

The driver automatically retries to run a failed transaction if the failure is deemed to be transient (for example due to temporary server unavailability). An error will be raised if the operation keeps failing after the configured maximum retry time.

Because the transaction might be re-run, **transaction functions must be *idempotent*** (i.e., they should produce the same effect when run several times), because you do not know upfront how many times they are going to be executed. In practice, this means that you should not edit nor rely on globals, for example. Note that although transactions functions might be executed multiple times, the database queries inside it will always run only once.

A session can chain multiple transactions, but **only one single transaction can be active within a session at any given time**. To maintain multiple concurrent transactions, use multiple concurrent sessions.

Transaction function configuration

The decorator `unit_of_work()` gives further control on transaction functions. It allows to specify:

- a transaction timeout (in seconds). Transactions that run longer will be terminated by the server. The default value is set on the server side. The minimum value is one millisecond (`0.001`).
- a dictionary of metadata that gets attached to the transaction. These metadata get logged in the server `query.log`, and are visible in the output of the `SHOW TRANSACTIONS` Cypher command. Use this to tag transactions.

```
from neo4j import unit_of_work

@unit_of_work(timeout=5, metadata={"app_name": "people_tracker"})
def count_people(tx):
    result = tx.run("MATCH (a:Person) RETURN count(a) AS people")
    record = result.single()
    return record["people"]


with driver.session(database="<database-name>") as session:
    people_n = session.execute_read(count_people)
```

Run an explicit transaction

You can achieve full control over transactions by manually beginning one with the method `Session.begin_transaction()`. You may then run queries inside an explicit transaction with the method `Transaction.run()`.

```
with driver.session(database="<database-name>") as session:
    with session.begin_transaction() as tx:
        # use tx.run() to run queries and tx.commit() when done
        tx.run("<QUERY 1>")
        tx.run("<QUERY 2>")

        tx.commit()
```

An explicit transaction can be committed with `Transaction.commit()` or rolled back with `Transaction.rollback()`. If no explicit action is taken, the driver will automatically roll back the transaction at the end of its lifetime.



Tip:

Queries run with `tx.run()` failing due to a transient server error can be retried without need to alter the original request. You can discover whether an error is transient via the method `Neo4jError.is_retryable()`, which gives insights into whether a further attempt might be successful.

Explicit transactions are most useful for applications that need to distribute Cypher execution across multiple functions for the same transaction, or for applications that need to run multiple queries within a single transaction but without the automatic retries provided by managed transactions.

An explicit transaction example involving an external API

```
import neo4j


URI = "<database-uri>"
AUTH = ("<username>", "<password>")


def main():
    with neo4j.GraphDatabase.driver(URI, auth=AUTH) as driver:
        customer_id = create_customer(driver)
        other_bank_id = 42
        transfer_to_other_bank(driver, customer_id, other_bank_id, 999)


def create_customer(driver):
    result, _, _ = driver.execute_query("""
        MERGE (c:Customer {id: rand()})
        RETURN c.id AS id
    """, database_ = "<database-name>")
    return result[0]["id"]


def transfer_to_other_bank(driver, customer_id, other_bank_id, amount):
    with driver.session(database="<database-name>") as session:
        with session.begin_transaction() as tx:
            if not customer_balance_check(tx, customer_id, amount):
                # give up
                return

            other_bank_transfer_api(customer_id, other_bank_id, amount)
```

```
            # Now the money has been transferred => can't rollback anymore
            # (cannot rollback external services interactions)

            try:
                decrease_customer_balance(tx, customer_id, amount)
                tx.commit()
            except Exception as e:
                request_inspection(customer_id, other_bank_id, amount, e)
                raise  # roll back


def customer_balance_check(tx, customer_id, amount):
    query = ("""
        MATCH (c:Customer {id: $id})
        RETURN c.balance >= $amount AS sufficient
    """)
    result = tx.run(query, id=customer_id, amount=amount)
    record = result.single(strict=True)
    return record["sufficient"]


def other_bank_transfer_api(customer_id, other_bank_id, amount):
    # make some API call to other bank
    pass


def decrease_customer_balance(tx, customer_id, amount):
    query = ("""
        MATCH (c:Customer {id: $id})
        SET c.balance = c.balance - $amount
    """)
    result = tx.run(query, id=customer_id, amount=amount)
    result.consume()


def request_inspection(customer_id, other_bank_id, amount, e):
    # manual cleanup required; log this or similar
    print("WARNING: transaction rolled back due to exception:", repr(e))
    print("customer_id:", customer_id, "other_bank_id:", other_bank_id,
          "amount:", amount)


if __name__ == "__main__":
    main()
```

Process query results

The driver's output of a query is a `Result` object, which encapsulates the Cypher result in a rich data structure that requires some parsing on the client side. There are two main points to be aware of:

- **The result records are not immediately and entirely fetched and returned by the server**. Instead, results come as a *lazy stream*. In particular, when the driver receives some records from the server, they are initially *buffered* in a background queue. Records stay in the buffer until they are *consumed* by the application, at which point they are *removed from the buffer*. When no more records are available, the result is *exhausted*.
- **The result acts as a cursor**. This means that there is no way to retrieve a previous record from the stream, unless you saved it in an auxiliary data structure.

The animation below follows the path of a single query: it shows how the driver works with result records and how the application should handle results.

```
<video
  class="rounded-corners"
  controls
  width="100%"
  src="../../../common-content/5/_images/result.mp4"
```

```
poster="../../../common-content/5/_images/result-poster.jpg"
type="video/mp4"></video>
```

The easiest way of processing a result is by casting it to list, which yields a list of `Record` objects. Otherwise, a `Result` object implements a number of methods for processing records. The most commonly needed ones are listed below.

Name	Description
`value(key=0, default=None)`	Return the remainder of the result as a list. If `key` is specified, only the given property is included; `default` allows to specify a value for nodes lacking that property.
`fetch(n)`	Return up to `n` records from the result.
`single(strict=False)`	Return the next and only remaining record, or `None`. Calling this method always exhausts the result. If more (or less) than one record is available, • `strict==False` — a warning is generated and the first of these is returned (if any); • `strict==True` — a `ResultNotSingleError` is raised.
`peek()`	Return the next record without consuming it. This leaves the record in the buffer for further processing.
`data(*keys)`	Return a JSON-like dump . Only use it for debugging/prototyping purposes.
`consume()`	Return the query result summary. It exhausts the result, so should only be called when data processing is over.
`graph()`	Transform result into a collection of graph objects. See Transform to graph.
`to_df(expand, parse_dates)`	Transform result into a Pandas Dataframe. See Transform to Pandas Dataframe.

For a complete list of `Result` methods, see API documentation — Result.

Session configuration

Database selection

Always specify the database explicitly with the `database` parameter, even on single-database instances. This allows the driver to work more efficiently, as it saves a network round-trip to the server to resolve the home database. If no database is given, the user's home database is used.

```
with driver.session(
    database="<database-name>"
) as session:
```

```
    ...
```



Specifying the database through the configuration method is preferred over the `USE` Cypher clause. If the server runs on a cluster, queries with `USE` require server-side routing to be enabled. Queries may also take longer to execute as they may not reach the right cluster member at the first attempt, and need to be routed to one containing the requested database.

Request routing

In a cluster environment, all sessions are opened in write mode, routing them to the leader. You can change this by explicitly setting the `default_access_mode` parameter to `neo4j.READ_ACCESS`. Note that `.execute_read()` and `.execute_write()` automatically override the session's default access mode.

```
import neo4j

with driver.session(
    database="<database-name>",
    default_access_mode=neo4j.READ_ACCESS
) as session:
    ...
```



Note:

Although executing a *write* query in read mode results in a runtime error, **you should not rely on this for access control.** The difference between the two modes is that *read* transactions will be routed to any node of a cluster, whereas *write* ones are directed to primaries. There is no security guarantee that a write query submitted in read mode will be rejected.

Run queries as a different user

You can execute a query through a different user with the parameter `auth`. Switching user at the session level is cheaper than creating a new `Driver` object. Queries are then run within the security context of the given user (i.e., home database, permissions, etc.).

```
with driver.session(
    database="<database-name>",
    auth=("<username>", "<password>")
) as session:
    ...
```

The parameter `impersonated_user` provides a similar functionality. The difference is that you don't need to know a user's password to impersonate them, but the user under which the `Driver` was created needs to have the appropriate permissions.

```
with driver.session(
    database="<database-name>",
    impersonated_user="<username>"
) as session:
```

```
    ...
```

Close sessions

Each connection pool has **a finite number of sessions**, so if you open sessions without ever closing them, your application could run out of them. It is thus recommended to create sessions using the `with` statement, which automatically closes them when the application is done with them. When a session is closed, it is returned to the connection pool to be later reused.

If you do not use `with`, remember to call the `.close()` method when you have finished using a session.

```
session = driver.session(database="<database-name>")

# session usage

session.close()
```

Explore the query execution summary

After all records of a query result have been processed, the server ends the transaction by returning a summary of execution. It comes as a `ResultSummary` object, and it contains information among which:

- Query counters — What changes the query triggered on the server
- Query execution plan — How the database would execute (or executed) the query
- Notifications — Extra information raised by the server while running the query
- Timing information and query request summary

Retrieve the execution summary

When running queries with `Driver.execute_query()`, the execution summary is part of the default return value, as second object.

```
records, result_summary, keys = driver.execute_query("""
    UNWIND ["Alice", "Bob"] AS name
    MERGE (p:Person {name: name})
    """, database_="<database-name>",
)
# or result_summary = driver.execute_query('<QUERY>').summary
```

If you are using transaction functions, or a custom transformer with `Driver.execute_query()`, you can retrieve the query execution summary with the method `Result.consume()`. **Notice that once you ask for the execution summary, the result stream is exhausted**: any record yet to be processed is not available anymore.

```
def create_people(tx):
    result = tx.run("""
        UNWIND ["Alice", "Bob"] AS name
        MERGE (p:Person {name: name})
    """)
    return result.consume()

with driver.session(database="<database-name>") as session:
    result_summary = session.execute_write(create_people)
```

Query counters

The property `ResultSummary.counters` contains counters for the operations that a query triggered (as a `SummaryCounters` object).

Insert some data and display the query counters

```
summary = driver.execute_query("""
    MERGE (p:Person {name: $name})
    MERGE (f:Person {name: $friend})
    MERGE (p)-[:KNOWS]->(f)
    """, name="Mark", friend="Bob",
    database_="<database-name>",
).summary
print(summary.counters)
"""
{'_contains_updates': True, 'labels_added': 2, 'relationships_created': 1,
```

```
 'nodes_created': 2, 'properties_set': 2}
"""
```

Two additional boolean properties act as meta-counters:

- `contains_updates` — whether the query triggered any write operation on the database on which it ran
- `contains_system_updates` — whether the query triggered any write operation on the `system` database

Query execution plan

If you prefix a query with `EXPLAIN`, the server returns the plan it *would* use to run the query, but doesn't actually run it. The plan is available under the property `ResultSummary.plan`, and contains the list of Cypher operators that would be used to retrieve the result set. Use this information to locate potential bottlenecks or potential performance improvements (for example the creation of indexes).

```
_, summary, _ = driver.execute_query("EXPLAIN MATCH (p {name: $name}) RETURN p", name="Alice")
print(summary.plan['args']['string-representation'])

"""
Planner COST
Runtime PIPELINED
Runtime version 5.0
Batch size 128

+-----------------+----------------+----------------+---------------------+
| Operator        | Details        | Estimated Rows | Pipeline            |
+-----------------+----------------+----------------+---------------------+
| +ProduceResults | p              |              1 |                     |
| |               +----------------+----------------+                     |
| +Filter         | p.name = $name |              1 |                     |
| |               +----------------+----------------+                     |
| +AllNodesScan   | p              |             10 | Fused in Pipeline 0 |
+-----------------+----------------+----------------+---------------------+

Total database accesses: ?
"""
```

If you instead prefix a query with the keyword `PROFILE`, the server returns the execution plan it has used to run the query, together with profiler statistics. This includes the list of operators that were used and additional profiling information about each intermediate step. The plan is available under the property `ResultSummary.profile`. Notice that the query is also *run*, so the result object also contains any result records.

```
records, summary, _ = driver.execute_query("PROFILE MATCH (p {name: $name}) RETURN p", name="Alice")
print(summary.profile['args']['string-representation'])

"""
Planner COST
Runtime PIPELINED
Runtime version 5.0
Batch size 128

+-----------------+----------------+----------------+------+---------+----------------
+-----------------------+-----------+---------------------+
| Operator        | Details        | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache
Hits/Misses | Time (ms) | Pipeline            |
+-----------------+----------------+----------------+------+---------+----------------
+-----------------------+-----------+---------------------+
| +ProduceResults | p              |              1 |    1 |       3 |                |
|           |                     |
| |               +----------------+----------------+------+---------+----------------+
|           |                     |
```

```
| +Filter         | p.name = $name |             1 |    1 |       4 |               |
|           |                    |
| |               +----------------+---------------+------+---------+---------------+
|           |                    |
| +AllNodesScan   | p              |            10 |    4 |       5 |           120 |
9160/0 |   108.923 | Fused in Pipeline 0 |
+-----------------+----------------+---------------+------+---------+---------------
+-----------------------+-----------+--------------------+

Total database accesses: 12, total allocated memory: 184
"""
```

For more information and examples, see Basic query tuning.

Notifications

After executing a query, the server can return notifications alongside the query result. Notifications contain recommendations for performance improvements, warnings about the usage of deprecated features, and other hints about sub-optimal usage of Neo4j.

Example 1. An unbounded shortest path raises a performance notification

The property `ResultSummary.gql_status_objects` contains a sequence of `GqlStatusObject`s. These are GQL-compliant status objects.

Some (but not all) `GqlStatusObjects` are notifications, whereas some report an *outcome* status: `00000` for "success", `02000` for "no data", and `00001` for "omitted result". `ResultSummary.gql_status_objects` always contains at least one entry, containing the outcome status.

```
records, summary, keys = driver.execute_query("""
    MATCH p=shortestPath((:Person {name: 'Alice'})-[*]->(:Person {name: 'Bob'}))
    RETURN p
    """, database_="<database-name>"
)
for status in summary.gql_status_objects:
    print("GQLSTATUS:", status.gql_status)
    print("Description:", status.status_description)
    # Not all statuses are notifications.
    print("Is notification:", status.is_notification)

    # Notification and thus vendor-specific fields.
    # These fields are only meaningful for notifications.
    if status.is_notification:
        # The position in the query that caused the notification.
        print("Position:", status.position)

        # The notification's classification is counterpart to `neo4j.NotificationCategory`.
        # However, the term `category` has a different meaning in the context of GQL.
        print("Classification:", status.classification)
        print("Unparsed classification:", status.raw_classification)

        print("Severity:", status.severity)
        print("Unparsed severity:", status.raw_severity)

    # Any raw extra information provided by the DBMS:
    print("Diagnostic record:", status.diagnostic_record)
    print("=" * 80)
"""
GQLSTATUS: 02000
Description: note: no data
Is notification: False
Diagnostic record: {'OPERATION': '', 'OPERATION_CODE': '0', 'CURRENT_SCHEMA': '/'}
```

```
================================================================================
GQLSTATUS: 03N91
Description: info: unbounded variable length pattern. The provided pattern `(:Person {name:
'Alice'})-[*]->(:Person {name: 'Bob'})` is unbounded. Shortest path with an unbounded pattern
may result in long execution times. Use an upper limit (e.g. `[*..5]`) on the number of node
hops in your pattern.
Is notification: True
Position: line: 1, column: 22, offset: 21
Classification: NotificationClassification.PERFORMANCE
Unparsed classification: PERFORMANCE
Severity: NotificationSeverity.INFORMATION
Unparsed severity: INFORMATION
Diagnostic record: {'_classification': 'PERFORMANCE', '_status_parameters': {'pat': "(:Person
{name: 'Alice'})-[*]->(:Person {name: 'Bob'})"}, '_severity': 'INFORMATION', '_position':
{'offset': 21, 'line': 1, 'column': 22}, 'OPERATION': '', 'OPERATION_CODE': '0',
'CURRENT_SCHEMA': '/'}
================================================================================
"""
```

Deprecated in 6.0

The property `ResultSummary.summary_notifications` contains a list of `SummaryNotification` objects.

```
records, summary, keys = driver.execute_query("""
    MATCH p=shortestPath((:Person {name: 'Alice'})-[*]->(:Person {name: 'Bob'}))
    RETURN p
    """, database_="<database-name>"
)
"""
[SummaryNotification(
    title='The provided pattern is unbounded, consider adding an upper limit to the number of
node hops.',
    code='Neo.ClientNotification.Statement.UnboundedVariableLengthPattern',
    description='Using shortest path with an unbounded pattern will likely result in long
execution times. It is recommended to use an upper limit to the number of node hops in your
pattern.',
    severity_level=<NotificationSeverity.INFORMATION: 'INFORMATION'>,
    category=<NotificationCategory.PERFORMANCE: 'PERFORMANCE'>,
    raw_severity_level='INFORMATION',
    raw_category='PERFORMANCE',
    position=SummaryNotificationPosition(line=1, column=22, offset=21)
)]
"""
```

Filter notifications

By default, the server analyses each query for all categories and severity of notifications. Use the parameters `notifications_min_severity` and/or `notifications_disabled_categories` / `notifications_disabled_classifications` to restrict the severity and/or category/classification of notifications that you are interested into. There is a slight performance gain in restricting the amount of notifications the server is allowed to raise. You can use any of those parameters either when creating a `Driver` instance, or when creating a session.

The severity filter applies to both Neo4j and GQL notifications. Category and classification filters exist separately only due to the discrepancy in lexicon between GQL and Neo4j; both filters affect either form of notification though, so you should use only one of them. If you provide both a category and a classification filter, their contents will be merged. From version 6.0, `notifications_disabled_categories` is deprecated.

You can disable notifications altogether by setting the minimum severity to `'OFF'`.

Allow only `WARNING` *notifications, but not of* `HINT` *or* `GENERIC` *category*

```
# at driver level
driver = neo4j.GraphDatabase.driver(
    URI, auth=AUTH,
    notifications_min_severity='WARNING',  # or 'OFF' to disable entirely
    notifications_disabled_classifications=['HINT', 'GENERIC'],  # filters categories as well
)

# at session level
session = driver.session(
    database="<database-name>",
    notifications_min_severity='INFORMATION',  # or 'OFF' to disable entirely
    notifications_disabled_classifications=['HINT']  # filters categories as well
)
```

Coordinate parallel transactions

When working with a Neo4j cluster, causal consistency is enforced by default in most cases, guaranteeing that a query is able to read changes made by previous queries. The same does not happen by default for multiple transactions running in parallel though. In that case, you can use *bookmarks* to have one transaction wait for the result of another to be propagated across the cluster before running its own work. This is not a requirement, and **you should only use bookmarks if you need casual consistency across different transactions**, as waiting for bookmarks can have a negative performance impact.

Bookmarks with `.execute_query()`

When querying the database with `.execute_query()`, the driver manages bookmarks for you. In this case, you have the guarantee that subsequent queries can read previous changes without taking further action.

```
driver.execute_query("<QUERY 1>")

# subsequent .execute_query() calls will be causally chained

driver.execute_query("<QUERY 2>") # can read result of <QUERY 1>
driver.execute_query("<QUERY 3>") # can read result of <QUERY 2>
```

To disable bookmark management and causal consistency, set `bookmark_manager_=None` in `.execute_query()` calls.

```
driver.execute_query(
    "<QUERY>",
    bookmark_manager_=None,
)
```

Bookmarks within a single session

Bookmark management happens automatically for queries run within a single session: queries inside the same session are causally chained.

```
with driver.session() as session:
    session.execute_write(lambda tx: tx.run("<QUERY 1>"))
    session.execute_write(lambda tx: tx.run("<QUERY 2>"))  # can read QUERY 1
    session.execute_write(lambda tx: tx.run("<QUERY 3>"))  # can read QUERY 1,2
```

Bookmarks across multiple sessions

If your application uses multiple sessions, you may need to ensure that one session has completed all its transactions before another session is allowed to run its queries.

In the example below, `session_a` and `session_b` are allowed to run concurrently, while `session_c` waits until their results have been propagated. This guarantees that the `Person` nodes `session_c` wants to act on actually exist.

Coordinate multiple sessions using bookmarks

```
from neo4j import GraphDatabase, Bookmarks


URI = "<database-uri>"
AUTH = ("<username>", "<password>")

def main():
    with GraphDatabase.driver(URI, auth=AUTH) as driver:
        driver.verify_connectivity()
        create_some_friends(driver)


def create_some_friends(driver):
    saved_bookmarks = Bookmarks()  # To collect the sessions' bookmarks

    # Create the first person and employment relationship
    with driver.session(database="<database-name>") as session_a:
        session_a.execute_write(create_person, "Alice")
        session_a.execute_write(employ, "Alice", "Wayne Enterprises")
        saved_bookmarks += session_a.last_bookmarks()

    # Create the second person and employment relationship
    with driver.session(database="<database-name>") as session_b:
        session_b.execute_write(create_person, "Bob")
        session_b.execute_write(employ, "Bob", "LexCorp")
        saved_bookmarks += session_b.last_bookmarks()

    # Create a friendship between the two people created above
    with driver.session(
        database="<database-name>", bookmarks=saved_bookmarks
    ) as session_c:
        session_c.execute_write(create_friendship, "Alice", "Bob")
        session_c.execute_read(print_friendships)


# Create a person node
def create_person(tx, name):
    tx.run("MERGE (:Person {name: $name})", name=name)


# Create an employment relationship to a pre-existing company node
# This relies on the person first having been created.
def employ(tx, person_name, company_name):
    tx.run("""
        MATCH (person:Person {name: $person_name})
        MATCH (company:Company {name: $company_name})
        CREATE (person)-[:WORKS_FOR]->(company)
        """, person_name=person_name, company_name=company_name
    )


# Create a friendship between two people
def create_friendship(tx, name_a, name_b):
    tx.run("""
        MATCH (a:Person {name: $name_a})
        MATCH (b:Person {name: $name_b})
        MERGE (a)-[:KNOWS]->(b)
        """, name_a=name_a, name_b=name_b
    )

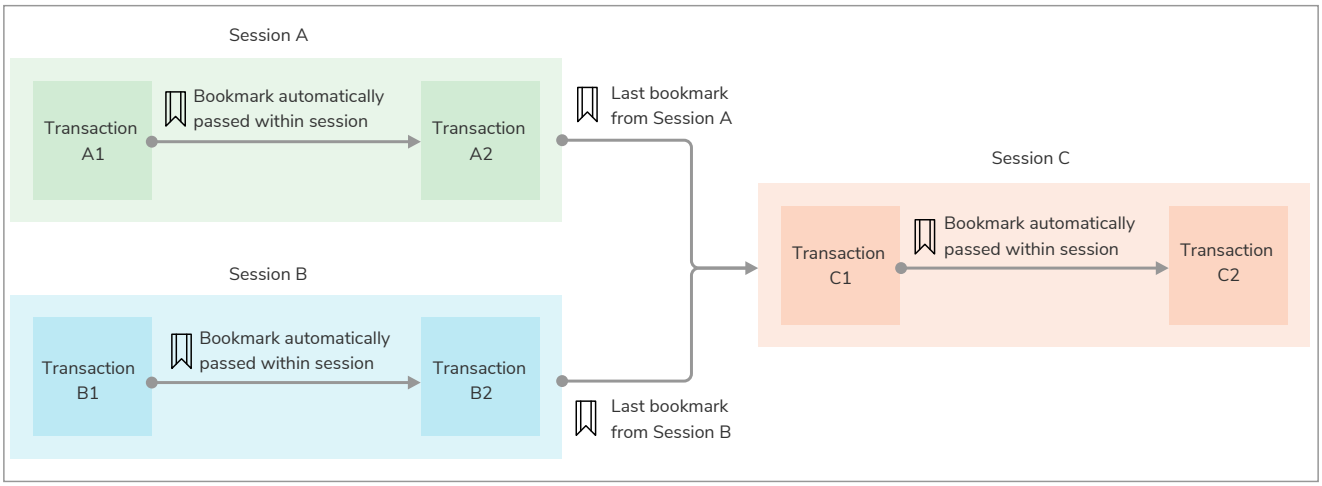

# Retrieve and display all friendships
def print_friendships(tx):
    result = tx.run("MATCH (a)-[:KNOWS]->(b) RETURN a.name, b.name")
    for record in result:
        print("{} knows {}".format(record["a.name"], record["b.name"]))


if __name__ == "__main__":
    main()
```

Collect and combine bookmarks from different sessions using `Session.last_bookmarks()`, storing them

in a `Bookmarks` object.

Use them to initialize another session with the `bookmarks` parameter.



The use of bookmarks can negatively impact performance, since the query is forced to wait for the latest changes to be propagated across the cluster. If possible, try to group queries within a single transaction, or within a single session.

Mix `.execute_query()` and sessions

To ensure causal consistency among transactions executed partly with `.execute_query()` and partly with sessions, create the session with `bookmark_manager=driver.execute_query_bookmark_manager`. Since that is the default bookmark manager for `.execute_query()` calls, this will ensure that all work is executed under the same bookmark manager and thus causally consistent.

```
driver.execute_query("<QUERY 1>")

with driver.session(
    bookmark_manager=driver.execute_query_bookmark_manager
) as session:
    # every query inside this session will be causally chained
    # (i.e., can read what was written by <QUERY 1>)
    session.execute_write(lambda tx: tx.run("<QUERY 2>"))

# subsequent execute_query calls will be causally chained
# (i.e., can read what was written by <QUERY 2>)
driver.execute_query("<QUERY 3>")
```

Implement a custom `BookmarkManager`

The *bookmark manager* is an interface used by the driver for keeping track of the bookmarks and keeping sessions automatically consistent.

You can subclass the `BookmarkManager` interface to implement a custom bookmark manager, or use the default implementation provided by the driver through `GraphDatabase.bookmark_manager()`. When implementing a bookmark manager, keep in mind that all methods must be concurrency safe.

The details of the interface can be found in the API documentation.

Run concurrent transactions

The driver is compatible with Python's `asyncio`, which allows implementing concurrent workflows. To interact with the database in an asynchronous way, create an `AsyncDriver` with `AsyncGraphDatabase.driver()`. The workflow is very similar to the synchronous version, except that you must use `await` on all async calls, and define as `async` all functions that should be awaited. If you need causal consistency across different transactions, use bookmarks.

An async driver example with `execute_query`

```
import asyncio
from neo4j import AsyncGraphDatabase


URI = "<database-uri>"
AUTH = ("<username>", "<password>")


async def main():
    async with AsyncGraphDatabase.driver(URI, auth=AUTH) as driver:
        records, summary, keys = await driver.execute_query(
            "MATCH (a:Person) RETURN a.name AS name",
            database_="<database-name>"
        )
        names = [record["name"] for record in records]
        print(names)


if __name__ == "__main__":
    asyncio.run(main())
```

An async driver example with transaction functions

```
import asyncio
from neo4j import AsyncGraphDatabase


URI = "<database-uri>"
AUTH = ("<username>", "<password>")


async def main():
    async with AsyncGraphDatabase.driver(URI, auth=AUTH) as driver:
        async with driver.session(database="<database-name>") as session:
            records = await session.execute_read(get_people)
            print(records)


async def get_people(tx):
    result = await tx.run("MATCH (a:Person) RETURN a.name AS name")
    records = await result.values()
    return records


if __name__ == "__main__":
    asyncio.run(main())
```



Async implements a concurrency model, but it is not the only possible one. Multithreading is another option, although `asyncio` is often easier to implement in an application.



There is a known issue with Python 3.8 and the async driver where it gradually slows down. It is generally recommended to use the latest supported version of Python for the

best performance, stability, and security.

Further query mechanisms

Implicit (or auto-commit) transactions



Implicit transactions are the only ones that can be used for `CALL { … } IN TRANSACTIONS` queries.

Implicit transactions are the most basic and limited form of transactions. Contrary to `.execute_query()` and managed transactions, implicit transactions offer limited guarantees of automatic retrial. As of version 6.2, implicit transactions are retried once on errors marked as idempotent by the server. You can disable auto-retry on implicit transactions via the configuration option `disable_auto_commit_retries` both at the driver and session level.

You run an implicit transaction with the method `Session.run()`. It returns a `Result` object that needs to be processed accordingly.

```
with driver.session(database="<database-name>") as session:
    res = session.run("CREATE (a:Person {name: $name})", name="Licia")
    res.consume()
```

To ensure an implicit transaction is committed, consume all records (either by calling `.consume()` on its result or by iterating over all records). Not consuming all records yields potentially surprising behavior: there is no guarantee on when exactly an implicit transaction will be committed during the lifetime of a session.



Caution:

Don't rely on session closure for committing implicit transactions. Pending queries will be committed to the database before closing the session, but there is no guarantee that they will succeed, and no exception will be raised unless the result is explicitly processed in some way. The following example contains a broken query which might not result in an exception:

```
with driver.session(database="<database-name>") as session:
    session.run("""
    UNWIND [4, 2, 1, 0] AS i
    CALL (i) {
      CREATE (:Person {num: 100/i}) // Fails for i = 0
    } IN TRANSACTIONS OF 2 ROWS
    RETURN i
    """)
```

Since the driver cannot figure out whether the query in a `session.run()` call requires a read or write session with the database, it defaults to write. If your implicit transaction contains read queries only, there is a performance gain in making the driver aware by setting the keyword argument `default_access_mode=neo4j.READ_ACCESS` when creating the session.

Import CSV files

The most common use case for using `Session.run()` is for importing large CSV files into the database with the `LOAD CSV` Cypher clause, and preventing timeout errors due to the size of the transaction.

Import CSV data into a Neo4j database

```
with driver.session(database="<database-name>") as session:
    result = session.run("""
        LOAD CSV FROM 'https://data.neo4j.com/bands/artists.csv' AS line
        CALL {
            WITH line
            MERGE (:Artist {name: line[1], age: toInteger(line[2])})
        } IN TRANSACTIONS OF 2 ROWS
    """)
    print(result.consume().counters)
```



While `LOAD CSV` can be a convenience, there is nothing *wrong* in deferring the parsing of the CSV file to your Python application and avoiding `LOAD CSV`. In fact, moving the parsing logic to the application can give you more control over the importing process. For efficient bulk data insertion, see Performance → Batch data creation.

For more information, see Cypher → Clauses → Load CSV.

Transaction configuration

The `Query` object allows to specify a query timeout and to attach metadata to the transaction. The metadata is visible in the server logs (as described for the `unit_of_work` decorator).

```
from neo4j import Query

with driver.session(database="<database-name>") as session:
    query = Query("CREATE (a:Person {name: $name})",
                  timeout=1.0,
                  metadata={"app_name": "people"})
    result = session.run(query, name="John")
```

Dynamic values in property keys, relationship types, and labels

In general, you should not concatenate parameters directly into a query, but rather use query parameters. There can however be circumstances where your query structure prevents the usage of parameters in all its parts. In fact, although parameters work for literals and expressions, as well as node labels and relationship types, they can't be used for property keys, so `MATCH (n) WHERE n.$param = 'something'` is invalid.

When using string concatenation, enclose the dynamic values in backticks and escape them yourself to protect against Cypher injections. Notice that Cypher processes Unicode, so take care of the Unicode literal `\u0060` as well.

Manually escaping dynamic property keys before concatenation

```
key = "nam\\u0060e"
# convert \u0060 to literal backtick and then escape backticks
escaped_key = key.replace("\\u0060", "`").replace("`", "``")

driver.execute_query(
    f"MATCH (p:Person {{`{escaped_key}`: $name}}) RETURN p.name",
```

```
    name="Alice",
    database_="<database-name>"
)
# rewritten to
# MATCH (p:Person {`nam``e`: $name}) RETURN p.name
```

Another workaround to avoid string concatenation is to use APOC procedures, such as `apoc.merge.node`, which supports dynamic labels and property keys.

Using `apoc.merge.node` to create a node with dynamic labels/property keys.

```
property_key = "name"
label = "Person"

driver.execute_query(
    "CALL apoc.merge.node($labels, $properties)",
    labels=[label], properties={property_key: "Alice"},
    database_="<database-name>"
)
```



If you are running Neo4j in Docker, APOC needs to be enabled when starting the container. See APOC → Installation → Docker.

Logging

The driver logs messages through the native `logging` library to a logger named `neo4j`. To redirect log messages to standard output, use the watch function:

```
import sys
from neo4j.debug import watch

watch("neo4j", out=sys.stdout)
```

Example of log output upon driver connection

```
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,616  [#0000]  _: <POOL>
created, routing address IPv4Address(('localhost', 7687))
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,616  [#0000]  _: <POOL>
acquire routing connection, access_mode='WRITE', database='neo4j'
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,616  [#0000]  _: <ROUTING>
checking table freshness (readonly=False): table expired=True, has_server_for_mode=False, table
routers={IPv4Address(('localhost', 7687))} => False
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,616  [#0000]  _: <POOL>
attempting to update routing table from IPv4Address(('localhost', 7687))
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,616  [#0000]  _: <RESOLVE>
in: localhost:7687
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,617  [#0000]  _: <RESOLVE>
dns resolver out: 127.0.0.1:7687
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,617  [#0000]  _: <POOL>
_acquire router connection, database='neo4j', address=ResolvedIPv4Address(('127.0.0.1', 7687))
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,617  [#0000]  _: <POOL>
trying to hand out new connection
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,617  [#0000]  C: <OPEN>
127.0.0.1:7687
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,619  [#AF18]  C: <MAGIC>
0x6060B017
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,619  [#AF18]  C:
<HANDSHAKE> 0x00000005 0x00020404 0x00000104 0x00000003
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,620  [#AF18]  S:
<HANDSHAKE> 0x00000005
[DEBUG   ] [Thread 139807941394432] [Task None           ] 2023-03-31 09:31:39,620  [#AF18]  C: HELLO
{'user_agent': 'neo4j-python/5.6.0 Python/3.10.6-final-0 (linux)', 'routing': {'address':
```

```
'localhost:7687'}, 'scheme': 'basic', 'principal': 'neo4j', 'credentials': '*******'}
```

Performance recommendations

Use the Rust extension

The Rust extension to the Python driver is an alternative driver package that yields a 3x to 10x speedup compared to the regular driver. You can install it with `pip install neo4j-rust-ext`, either alongside the `neo4j` package or as a replacement to it. Usage-wise, the drivers are identical: everything in this guide applies to both packages.

Always specify the target database

Specify the target database on all queries, either with the `database_` parameter in `Driver.execute_query()` or with the `database` parameter when creating new sessions. If no database is provided, the driver has to send an extra request to the server to figure out what the default database is. The overhead is minimal for a single query, but becomes significant over hundreds of queries.

Good practices

```
driver.execute_query("<QUERY>", database_="<database-name>")
```

```
driver.session(database="<database-name>")
```

Bad practices

```
driver.execute_query("<QUERY>")
```

```
driver.session()
```

Be aware of the cost of transactions

Queries are always run within transactions on the server, so that the database always ends up in a consistent state regardless of what happens during a transaction execution (power outages, software crashes, etc). The driver automatically retries transactions started with `.execute_query()` or `.execute_read/write()` in case of failure. A transactions yields an overhead that is small, but that adds up as the number of transactions increases.

As your workload grows, evaluate what level of isolation each query needs and be intentional about your usage of transactions. There are essentially three approaches to running queries depending on your needs for *safety* and *performance*. Remember that there's no free lunch and that it's always a trade-off between the two.

1. **One transaction One query (lowest throughput, highest safety)**
   The safest mode of execution with respect to failures, because of the isolation and auto-retry

guarantees, but also the slowest due to the overhead of transactions scaling with the number of queries.

Run 1000 queries, each as a separate transaction

```
for i in range(1000):
    driver.execute_query("<QUERY>", database_="<database-name>")
    # or session.execute_read/write() calls, each with one query
```

2. **One transaction ▯ Multiple queries (higher throughput, high safety)**
   When multiple queries are grouped into a single transaction, the transaction as a whole is isolated from others, whereas individual queries in the transaction are not isolated and failure of one query results in a rollback of all queries.

   Run 10 transactions with 100 queries each

```
def query(tx):
    for i in range(100):
        tx.run("<QUERY>")

with driver.session(database="<database-name>") as session:
    for i in range(10):
        people = session.execute_read(query)
```

3. **Auto-commit transactions (highest throughput, lowest safety)**
   Auto-commit transactions shoot queries to the server as fast as it can handle. Failed queries are not retried, and there's no exact guarantee on when changes get written to the database. One upper limit on the client side is given by the size of the connection pool: each call to `session.run()` borrows a connection, so the amount of parallel work is limited by the number of available connections.

   Run 1000 queries as auto-commit transactions

```
with driver.session(database="<database-name>") as session:
    for i in range(1000):
        session.run("<QUERY>")
```

Don't fetch large result sets all at once

When submitting queries that may result in a lot of records, don't retrieve them all at once. The Neo4j server can retrieve records in batches and stream them to the driver as they become available. Lazy-loading a result spreads out network traffic and memory usage (both client- and server-side).

For convenience, `.execute_query()` always retrieves all result records at once (it is what the `Eager` in `EagerResult` stands for). To lazy-load a result, you have to use `.execute_read/write()` (or other forms of manually-handled transactions) and **not** cast the `Result` object to `list` when processing the result; iterate on it instead.

For more information on result processing, see Transactions → Process query results.

Example 2. Comparison between eager and lazy loading

Eager loading	Lazy loading
• The server has to read all records from the storage before it can send even the first one to the driver (i.e. it takes more time for the client to receive the first record). • Before any record is available to the application, the driver has to receive all records. • The client has to hold in memory all records.	• The server reads the first record and sends it to the driver. • The application can process records as soon as the first record is transferred. • Waiting time and resource consumption for the remaining records is deferred to when the application requests more records. • The server's fetch time can be used for client-side processing. • Resource consumption is bounded by the driver's fetch size.

Time and memory comparison between eager and lazy loading

```
import neo4j
from time import sleep, time
import tracemalloc



URI = "<database-uri>"
AUTH = ("<username>", "<password>")

# Returns 250 records, each with properties
# - `output` (an expensive computation, to slow down retrieval)
# - `dummyData` (a list of 10000 ints, about 8 KB).
slow_query = '''
UNWIND range(1, 250) AS s
RETURN reduce(s=s, x in range(1,1000000) | s + sin(toFloat(x))+cos(toFloat(x))) AS output,
       range(1, 10000) AS dummyData
'''
# Delay for each processed record
sleep_time = 0.5


def main():
    with neo4j.GraphDatabase.driver(URI, auth=AUTH) as driver:
        driver.verify_connectivity()

        start_time = time()
        log('LAZY LOADING (execute_read)')
        tracemalloc.start()
        lazy_loading(driver)
        log(f'Peak memory usage: {tracemalloc.get_traced_memory()[1]} bytes')
        tracemalloc.stop()
        log('--- %s seconds ---' % (time() - start_time))

        start_time = time()
        log('EAGER LOADING (execute_query)')
        tracemalloc.start()
        eager_loading(driver)
        log(f'Peak memory usage: {tracemalloc.get_traced_memory()[1]} bytes')
        tracemalloc.stop()
        log('--- %s seconds ---' % (time() - start_time))


def lazy_loading(driver):

    def process_records(tx):
        log('Submit query')
        result = tx.run(slow_query)
```

```
        for record in result:
            log(f'Processing record {int(record.get("output"))}')
            sleep(sleep_time)  # proxy for some expensive operation

    with driver.session(database='<database-name>') as session:
        processed_result = session.execute_read(process_records)


def eager_loading(driver):
    log('Submit query')
    records, _, _ = driver.execute_query(slow_query, database_='<database-name>')

    for record in records:
        log(f'Processing record {int(record.get("output"))}')
        sleep(sleep_time)  # proxy for some expensive operation


def log(msg):
    print(f'[{round(time(), 2)}] {msg}')


if __name__ == '__main__':
    main()
```

Output

```
[1718014256.98] LAZY LOADING (execute_read)
[1718014256.98] Submit query
[1718014256.21] Processing record 0
[1718014256.71] Processing record 1
[1718014257.21] Processing record 2
...
[1718014395.42] Processing record 249
[1718014395.92] Peak memory usage: 786254 bytes
[1718014395.92] --- 135.9284942150116 seconds ---

[1718014395.92] EAGER LOADING (execute_query)
[1718014395.92] Submit query
[1718014419.82] Processing record 0
[1718014420.33] Processing record 1
[1718014420.83] Processing record 2
...
[1718014544.52] Processing record 249
[1718014545.02] Peak memory usage: 89587150 bytes
[1718014545.02] --- 149.70468592643738 seconds ---
```

With lazy loading, the first record is available sooner.

With eager loading, the first record is available once the result is consumed (i.e. after the server has retrieved all 250 records).

Memory usage is larger with eager loading because the application materializes a list of 250 records.

The total running time is longer with eager loading because the client waits until it receives the last record, whereas with lazy loading the client can process records while the server fetches the next ones. With lazy loading, the client can also stop requesting records after some condition is met (by calling `.consume()` on the `Result` object) to save time and resources.



Tip:

The driver's fetch size affects the behavior of lazy loading. It instructs the server to stream an amount of records equal to the fetch size, and then wait until the client has caught up before retrieving and sending more.

The fetch size usually allows to bound memory consumption on the client side, especially for results in which the variance of the records size is small. If one single record is very large, the driver still needs to allocate space for the whole object, so memory usage may get large even with a small fetch size.

On the other hand, the fetch size doesn't always bound memory consumption on the server side: that depends on the query. For example, a query with `ORDER BY` requires the whole result set to be loaded into memory for sorting, before records can be streamed to the client.

The lower the fetch size, the more messages client and server have to exchange. Especially if the server's latency is high, a low fetch size can deteriorate performance.

Route read queries to cluster readers

In a cluster, **route read queries to any reader node**. You do this by:

- specifying `routing_="r"` in a `Driver.execute_query()` call
- using `Session.execute_read()` instead of `Session.execute_write()` (for managed transactions)
- setting `default_access_mode=neo4j.READ_ACCESS` when creating a new session (for explicit transactions).

Good practices

```
driver.execute_query("MATCH (p:Person) RETURN p", routing_="r")
```

```
session.execute_read(lambda tx: tx.run("MATCH (p:Person) RETURN p"))
```

Bad practices

```
driver.execute_query("MATCH (p:Person) RETURN p")
# defaults to routing = writers
```

```
session.execute_write(lambda tx: tx.run("MATCH (p:Person) RETURN p"))
# don't ask to write on a read-only operation
```

Create indexes

Create indexes for properties that you often filter against. For example, if you often look up `Person` nodes by the `name` property, it is beneficial to create an index on `Person.name`. You can create indexes with the `CREATE INDEX` Cypher clause, for both nodes and relationships.

```
# Create an index on Person.name
driver.execute_query("CREATE INDEX person_name FOR (n:Person) ON (n.name)")
```

For more information, see Indexes for search performance.

Profile queries

Profile your queries to locate queries whose performance can be improved. You can profile queries by prepending them with `PROFILE`. In case some queries are so slow that you are unable to even run them in reasonable times, you can prepend them with `EXPLAIN` instead of `PROFILE`. This will return the plan that the server would use to run the query, but without executing it.

For more information, see Explore the query execution summary → Query execution plan.

Specify node labels

Specify node labels in all queries. This allows the query planner to work more efficiently, and to leverage indexes where available. To learn how to combine labels, see Cypher → Label expressions.

Good practices

```
driver.execute_query("MATCH (p:Person|Animal {name: $name}) RETURN p", name="Alice")
```

```
with driver.session(database="<database-name>") as session:
    session.run("MATCH (p:Person|Animal {name: $name}) RETURN p", name="Alice")
```

Bad practices

```
driver.execute_query("MATCH (p {name: $name}) RETURN p", name="Alice")
```

```
with driver.session(database="<database-name>") as session:
    session.run("MATCH (p {name: $name}) RETURN p", name="Alice")
```

Batch data creation

Batch queries when creating a lot of records using the `UNWIND` Cypher clauses.

Good practice

Submit one single queries with all values inside

```
numbers = [{"value": random()} for _ in range(10000)]
driver.execute_query("""
    UNWIND $numbers AS node
    MERGE (:Number {value: node.value})
    """, numbers=numbers,
)
```

Bad practice

Submit a lot of single queries, one for each value

```
for _ in range(10000):
    driver.execute_query("MERGE (:Number {value: $value})", value=random())
```



The most efficient way of performing a *first import* of large amounts of data into a new database is the `neo4j-admin database import` command.

Use query parameters

Always use query parameters instead of hardcoding or concatenating values into queries. Besides protecting from Cypher injections, this allows to better leverage the database query cache.

Good practices

```
driver.execute_query("MATCH (p:Person {name: $name}) RETURN p", name="Alice")
```

```
with driver.session(database="<database-name>") as session:
    session.run("MATCH (p:Person {name: $name}) RETURN p", name="Alice")
```

Bad practices

```
driver.execute_query("MATCH (p:Person {name: 'Alice'}) RETURN p")
# or
name = "Alice"
driver.execute_query("MATCH (p:Person {name: '" + name + "'}) RETURN p")
```

```
with driver.session(database="<database-name>") as session:
    session.run("MATCH (p:Person {name: 'Alice'}) RETURN p")
    # or
    name = "Alice"
    session.run("MATCH (p:Person {name: '" + name + "'}) RETURN p")
```

Concurrency

Use concurrency, either in the form of multithreading or with the async version of the driver. This is likely to be more impactful on performance if you parallelize complex and time-consuming queries in your application, but not so much if you run many simple ones.

Use `MERGE` for creation only when needed

The Cypher clause `MERGE` is convenient for data creation, as it allows to avoid duplicate data when an exact clone of the given pattern exists. However, it requires the database to run two queries: it first needs to `CREATE` it (if needed).

If you know already that the data you are inserting is new, avoid using `MERGE` and use `CREATE` directly instead — this practically halves the number of database queries.

Filter notifications

Filter the category and/or severity of notifications the server should raise.

Reference

Advanced connection information

Connection URI

The driver supports connection to URIs of the form

```
<SCHEME>://<HOST>[:<PORT>[?policy=<POLICY-NAME>]]
```

- `<SCHEME>` is one among `neo4j`, `neo4j+s`, `neo4j+ssc`, `bolt`, `bolt+s`, `bolt+ssc`.
- `<HOST>` is the host name where the Neo4j server is located.
- `<PORT>` is optional, and denotes the port the Bolt protocol is available at.
- `<POLICY-NAME>` is an optional *server policy* name. Server policies need to be set up prior to usage.



The driver does not support connection to a nested path, such as `example.com/neo4j/`. The server must be reachable from the domain root.

Connection protocols and security

Communication between the driver and the server is mediated by Bolt. The scheme of the server URI determines whether the connection is encrypted and, if so, what type of certificates are accepted.

URL scheme	Encryption	Comment
neo4j	✖	Default for local setups
neo4j+s	✔ (only CA-signed certificates)	Default for Aura
neo4j+ssc	✔ (CA- and self-signed certificates)	



The driver receives a *routing table* from the server upon successful connection, regardless of whether the instance is a proper cluster environment or a single-machine environment. The driver's routing behavior works in tandem with Neo4j's clustering by directing read/write transactions to appropriate cluster members. If you want to target a *specific* machine, use the `bolt`, `bolt+s`, or `bolt+ssc` URI schemes instead.

The connection scheme to use is not your choice, but is rather determined by the server requirements. You must know the right server scheme upfront, as no metadata is exposed prior to connection. The connection scheme is part of the instance URI. If you are unsure, ask the database administrator.

Authentication methods

Basic authentication (default)

The basic authentication scheme relies on traditional username and password.

```
from neo4j import GraphDatabase

driver = GraphDatabase.driver(URI, auth=(USERNAME, PASSWORD))
```

The basic authentication scheme can also be used to authenticate against an LDAP server (Enterprise Edition only).

Kerberos authentication

The Kerberos authentication scheme requires a base64-encoded ticket. It can only be used if the server has the Kerberos Add-on installed.

```
from neo4j import GraphDatabase, kerberos_auth

driver = GraphDatabase.driver(URI, auth=kerberos_auth(ticket))
```

Bearer authentication

The bearer authentication scheme requires a base64-encoded token provided by an Identity Provider through Neo4j's Single Sign-On feature.

```
from neo4j import GraphDatabase, bearer_auth

driver = GraphDatabase.driver(URI, auth=bearer_auth(token))
```



The bearer authentication scheme requires configuring Single Sign-On on the server. Once configured, clients can discover Neo4j's configuration through the Discovery API.

Custom authentication

Use the function `custom_auth` to log into a server having a custom authentication scheme.

No authentication

If authentication is disabled on the server, the authentication parameter can be omitted entirely.

Rotating authentication tokens

It is possible to rotate authentication tokens that are expected to expire (e.g. SSO). Transactions running longer than the validity of a token will carry through without re-authenticating, whereas different transactions within the same session might work under different authentication tokens.

To work with rotating tokens, you need to provide a `AuthManager` instance (or `AsyncAuthManager` for the async driver) when instantiating the `Driver`, rather than a static authentication token. The easiest way to get started is to use one of built-in `AuthManager` implementations.

Rotating a bearer token expiring every 60 seconds

```
import neo4j
from neo4j.auth_management import (
    AuthManagers,
    ExpiringAuth,
)


def auth_provider():
    # Some way to get a token
    sso_token = get_sso_token()
    # Assume we know tokens expire every 60 seconds
    expires_in = 60
    # Include a little buffer so that new token is fetched before the old one expires
    expires_in -= 10

    auth = neo4j.bearer_auth(sso_token)
    return ExpiringAuth(auth=auth).expires_in(expires_in)

with neo4j.GraphDatabase.driver(
    URI,
    auth=AuthManagers.bearer(auth_provider)
) as driver:
    ...
```



This API must not be used for switching users. Auth managers must always return tokens for the same identity. You can switch users at both query level and session level.



`AuthManagers` (including provider functions passed to `AuthManagers.expiration_based()`) must not interact with the driver in any way, as this can cause deadlocks and undefined behavior.

Mutual TLS (client-side certificates as 2FA)

Not available on Aura

Mutual TLS (mTLS) allows you to use a client certificate as second factor for authenticating with the server. The certificate can only be used together with an authentication token and is not a replacement of regular authentication, unless authentication is disabled on the server.

The client's certificate and public key must be placed in the server's `<NEO4J_HOME>/certificates/bolt/trusted` directory. For more information on server setup, see Configure SSL over Bolt.



For mTLS to work, the driver's connection with the server must be encrypted, i.e. the connection URI scheme must be either `+s` or `+ssc` (ex. `neo4j+s://example.com:7687`).

Use `ClientCertificateProviders.static()` for static certificates.
The method takes a `ClientCertificate` instance, which takes the same parameters as Python's `ssl.SSLContext.load_cert_chain()`.

```
import neo4j
from neo4j.auth_management import (
    ClientCertificate,
    ClientCertificateProviders,
)
URI = "<database-uri>"
AUTH = ("<username>", "<password>")
cert_provider = ClientCertificateProviders.static(
    ClientCertificate(
        # path to certificate
        "path/to/cert.pem",
        # path to private key (optional; needed if certificate does not contain private key too)
        "path/to/key.pem",
        # password to decrypt private key (can be function or string) (optional)
        lambda: "password",
    )
)
with neo4j.GraphDatabase.driver(
    URI,
    auth=AUTH,
    client_certificate=cert_provider,
) as driver:
    ...
```

Use `ClientCertificateProviders.rotating()` for rotating certificates.
The method takes a `ClientCertificate` instance.

```
import neo4j
from neo4j.auth_management import (
    ClientCertificate,
    ClientCertificateProviders,
)
URI = "<database-uri>"
AUTH = ("<username>", "<password>")
cert_provider = ClientCertificateProviders.rotating(
    ClientCertificate(
        # path to public certificate to load
        "path/to/cert.pem",
        # path to private key to load
        "path/to/key.pem",
        # password to decrypt private key (can be a function or string)
        # see also Python's ssl.SSLContext.load_cert_chain()
        lambda: "password",
    )
)
driver = neo4j.GraphDatabase.driver(
   URI
   auth=(USERNAME, PASSWORD),
   client_certificate=cert_provider
)
# use the driver...
# ... until the certificate needs to be rotated
cert_provider.update_certificate(
    ClientCertificate(
        certfile="path/to/new/cert.pem",
        keyfile="path/to/new/key.pem",
        password=lambda: "new_super_secret_password"
    )
)
# use the driver again, until the certificate needs to be
# rotated again
# ...
```

For more information, see API docs → `ClientCertificateProvider`.

Custom address resolver

When creating a `Driver` object, you can specify a *resolver* function to resolve any addresses the driver receives ahead of DNS resolution. Your resolver function is called with an `Address` object and should return an iterable of `Address` objects (or values that can be used to construct `Address` objects).



A custom address resolver cannot alter the URI scheme (`neo4j://` in the example below).

Connection to `example.com` on port `9999` is resolved to `localhost` on port `7687`

```
import neo4j


def custom_resolver(socket_address):
    # assert isinstance(socket_address, neo4j.Address)
    if socket_address != ("example.com", 9999):
        raise OSError(f"Unexpected socket address {socket_address!r}")

    # You can return any neo4j.Address object
    yield neo4j.Address(("localhost", 7687))  # IPv4
    yield neo4j.Address(("::1", 7687, 0, 0))  # IPv6
    yield neo4j.Address.parse("localhost:7687")
    yield neo4j.Address.parse("[::1]:7687")

    # or any tuple that can be passed to neo4j.Address().
    # This will initially be interpreted as IPv4, but DNS resolution
    # will turn it into IPv6 if appropriate.
    yield "::1", 7687
    # This will be interpreted as IPv6 directly, but DNS resolution will
    # still happen.
    yield "::1", 7687, 0, 0
    yield "127.0.0.1", 7687


driver = neo4j.GraphDatabase.driver("neo4j://example.com:9999",
                                    auth=(USERNAME, PASSWORD),
                                    resolver=custom_resolver)
```

Disable telemetry

The driver can send anonymous usage statistics to the server it connects to, if the server requests those. Telemetry is managed on the server-side via the configuration setting `server.bolt.telemetry.enabled`. By default, telemetry is disabled on self-managed instances and enabled on Aura instances.

By setting `telemetry_disabled=True` upon connection, the driver will never send any telemetry data.

Disable telemetry on the driver side

```
import neo4j


driver = neo4j.GraphDatabase.driver(URI, auth=(USERNAME, PASSWORD), telemetry_disabled=True)
```

For more information on what information is transmitted, see API documentation → `telemetry_disabled`.

Further connection parameters

You can find all `Driver` configuration parameters in the API documentation.

Data types and mapping to Cypher types

The tables in this section show the mapping between Cypher data types and Python types.

Core types

Cypher Type	Python Type
NULL	`None`
`LIST`	`list`
`MAP`	`dict`
`BOOLEAN`	`bool`
`INTEGER`	`int`
`FLOAT`	`float`
`STRING`	`str`
`ByteArray`	`bytearray`
`UUID`	`uuid.UUID`

For full documentation, see API documentation — Core data types.

Temporal types

Temporal data types are implemented by the `neo4j.time` module. It provides a set of types compliant with ISO-8601 and Cypher, which are similar to those found in Python's native `datetime` module. To convert between driver and native types, use the methods `.from_native()` and `.to_native()` (does not apply to `Duration`). This conversion is lossy, as the driver's types support nanosecond precision, while Python's native types do not.



The driver's temporal types are designed to be used with pytz. Other `datetime.tzinfo` implementations (e.g., `datetime.timezone`, `zoneinfo`, `dateutil.tz`) are not supported and are unlikely to work well.

For a list of time zone abbreviations, see List of tz database time zones.

Cypher Type	Python Type
`DATE`	`neo4j.time.Date`
`ZONED TIME`	`neo4j.time.Time` †
`LOCAL TIME`	`neo4j.time.Time` ††
`ZONED DATETIME`	`neo4j.time.DateTime` †

Cypher Type	Python Type
`LOCAL DATETIME`	`neo4j.time.DateTime` ††
`DURATION`	`neo4j.time.Duration`

† Where `tzinfo` is not `None`.
†† Where `tzinfo` is `None`.

How to use temporal types in queries

```
from datetime import datetime
import pytz
from neo4j import GraphDatabase
from neo4j.time import DateTime


URI = "<database-uri>"
AUTH = ("<username>", "<password>")


friends_since = DateTime(year=1999, month=11, day=23,
                         hour=7, minute=47, nanosecond=4123)
friends_since = pytz.timezone("US/Eastern").localize(friends_since)

# Python's native datetimes work as well.
# They don't support the full feature-set of Neo4j's type though.
# py_friends_since = datetime(year=1999, month=11, day=23, hour=7, minute=47)
# py_friends_since = pytz.timezone("US/Eastern").localize(py_friends_since)

# Create a friendship with the given DateTime, and return the DateTime
with GraphDatabase.driver(URI, auth=AUTH) as driver:
    records, summary, keys = driver.execute_query("""
        MERGE (a:Person {name: $name})
        MERGE (b:Person {name: $friend})
        MERGE (a)-[friendship:KNOWS {since: $friends_since}]->(b)
        RETURN friendship.since
        """, name="Alice", friend="Bob",
        friends_since=friends_since  # or friends_since=py_friends_since
    )
    out_datetime = records[0]["friendship.since"]
    print(out_datetime)  # 1999-11-23T07:47:00.000004123-05:00

    # Converting DateTime to native datetime (lossy)
    py_out_datetime = out_datetime.to_native()  # type: datetime
    print(py_out_datetime)  # 1999-11-23 07:47:00.000004-05:00
```

For full documentation, see API documentation — Temporal data types.

Date

Represents an instant capturing the date, but not the time, nor the timezone.

```
from neo4j.time import Date

d = Date(year=2021, month=11, day=2)
print(d)  # '2021-11-02'
```

For full documentation, see API documentation — Temporal data types — Date.

Time

Represents an instant capturing the time, and the timezone offset in seconds, but not the date.

```
from neo4j.time import Time
import pytz

t = Time(hour=7, minute=47, nanosecond=4123, tzinfo=pytz.FixedOffset(-240))
print(t)  # '07:47:00.000004123-04:00'
```

For full documentation, see API documentation — Temporal data types — Time.

LocalTime

Represents an instant capturing the time of day, but not the date, nor the timezone.

```
from neo4j.time import Time

t = Time(hour=7, minute=47, nanosecond=4123)
print(t)  # '07:47:00.000004123'
```

For full documentation, see API documentation — Temporal data types — Time.

DateTime

Represents an instant capturing the date, the time, and the timezone identifier.

```
from neo4j.time import DateTime
import pytz

dt = DateTime(year=2021, month=11, day=2, hour=7, minute=47, nanosecond=4123)
dt = pytz.timezone("US/Eastern").localize(dt)  # time zone localization (optional)
print(dt)  # '2021-11-02T07:47:00.000004123-04:00'
```

For full documentation, see API documentation — Temporal data types — DateTime.

LocalDateTime

Represents an instant capturing the date and the time, but not the timezone.

```
from neo4j.time import DateTime

dt = DateTime(year=2021, month=11, day=2, hour=7, minute=47, nanosecond=4123)
print(dt)  # '2021-11-02T07:47:00.000004123'
```

For full documentation, see API documentation — Temporal data types — DateTime.

Duration

Represents the difference between two points in time. A `datetime.timedelta` object passed as a parameter will always be implicitly converted to `neo4j.time.Duration`. It is not possible to convert from `neo4j.time.Duration` to `datetime.timedelta` (because `datetime.timedelta` lacks month support).

```
from neo4j.time import Duration
```

```
duration = Duration(years=1, days=2, seconds=3, nanoseconds=4)
print(duration)  # 'P1Y2DT3.000000004S'
```

For full documentation, see API documentation — Temporal data types — Duration.

Spatial types

Cypher supports spatial values (points), and Neo4j can store these point values as properties on nodes and relationships.

The object attribute `srid` (short for *Spatial Reference Identifier*) is a number identifying the coordinate system the spatial type is to be interpreted in. You can think of it as a unique identifier for each spatial type.

Cypher Type	Python Type
`POINT`	`neo4j.spatial.Point`
`POINT (Cartesian)`	`neo4j.spatial.CartesianPoint`
`POINT (WGS-84)`	`neo4j.spatial.WGS84Point`

For full documentation, see API documentation — Spatial types.

CartesianPoint

Represents a point in 2D/3D Cartesian space.
Exposes properties `x`, `y`, `z` (the latter for 3D points only).

```
from neo4j.spatial import CartesianPoint

# A 2D CartesianPoint
point = CartesianPoint((1.23, 4.56))
print(point.x, point.y, point.srid)
# 1.23 4.56 7203

# A 3D CartesianPoint
point = CartesianPoint((1.23, 4.56, 7.89))
print(point.x, point.y, point.z, point.srid)
# 1.23 4.56 7.8 9157
```

For full documentation, see API documentation — Spatial types — CartesianPoint.

WGS84Point

Represents a point in the World Geodetic System (WGS84).
Exposes properties `longitude`, `latitude`, `height` (the latter for 3D points only), which are aliases for `x`, `y`, `z`.

```
from neo4j.spatial import WGS84Point

# A 2D WGS84Point
point = WGS84Point((1.23, 4.56))
print(point.longitude, point.latitude, point.srid)
# or print(point.x, point.y, point.srid)
```

```
# 1.23 4.56 4326

# A 3D WGS84Point
point = WGS84Point((1.23, 4.56, 7.89))
print(point.longitude, point.latitude, point.height, point.srid)
# or print(point.x, point.y, point.z, point.srid)
# 1.23 4.56 7.89 4979
```

For full documentation, see API documentation — Spatial types — WSG84Point.

Graph types

Graph types are only passed as results and can't be used as parameters. The section Manipulate query results — Transform to graph contains an example with graph types.

Cypher Type	Python Type
`NODE`	`neo4j.graph.Node`
`RELATIONSHIP`	`neo4j.graph.Relationship`
`PATH`	`neo4j.graph.Path`

For full documentation, see API documentation — Graph types.

Node

Represents a node in a graph.
The property `element_id` provides an identifier for the entity. This should be used with care, as no guarantees are given about the mapping between id values and elements outside the scope of a single transaction. In other words, using an `element_id` to `MATCH` an element across different transactions is risky.

```
from neo4j import GraphDatabase


URI = "<database-uri>"
AUTH = ("<username>", "<password>")

with GraphDatabase.driver(URI, auth=AUTH) as driver:
    records, _, _ = driver.execute_query(
        "MERGE (p:Person {name: $name}) RETURN p AS person",
        name="Alice",
        database_="<database-name>",
    )
    for record in records:
        node = record["person"]
        print(f"Node ID: {node.element_id}\n"
              f"Labels: {node.labels}\n"
              f"Properties: {node.items()}\n"
        )

# Node ID: 4:73e9a61b-b501-476d-ad6f-8d7edf459251:0
# Labels: frozenset({'Person'})
# Properties: dict_items([('name', 'Alice')])
```

For full documentation, see API documentation — Graph types — Node.

Relationship

Represents a relationship in a graph.
The property `element_id` provides an identifier for the entity. This should be used with care, as no guarantees are given about the mapping between id values and elements outside the scope of a single transaction.

```
from neo4j import GraphDatabase


URI = "<database-uri>"
AUTH = ("<username>", "<password>")

with GraphDatabase.driver(URI, auth=AUTH) as driver:
    records, _, _ = driver.execute_query("""
        MERGE (p:Person {name: $name})
        MERGE (p)-[r:KNOWS {status: $status, since: date()}]->(friend:Person {name: $friend_name})
        RETURN r AS friendship
        """, name="Alice", status="BFF", friend_name="Bob",
    )
    for record in records:
        relationship = record["friendship"]
        print(f"Relationship ID: {relationship.element_id}\n"
              f"Start node: {relationship.start_node}\n"
              f"End node: {relationship.end_node}\n"
              f"Type: {relationship.type}\n"
              f"Friends since: {relationship.get('since')}\n"
              f"All properties: {relationship.items()}\n"
        )

# Relationship ID: 5:73e9a61b-b501-476d-ad6f-8d7edf459251:1
# Start node: <Node element_id='4:73e9a61b-b501-476d-ad6f-8d7edf459251:0' labels=frozenset({'Person'})
properties={'name': 'Alice'}>
# End node: <Node element_id='4:73e9a61b-b501-476d-ad6f-8d7edf459251:2' labels=frozenset({'Person'})
properties={'name': 'Bob'}>
# Type: KNOWS
# Friends since: 2022-11-07
# All properties: dict_items([('since', neo4j.time.Date(2022, 11, 7)), ('status', 'BFF')])
```

For full documentation, see API documentation — Graph types — Relationship.

Path

Represents a path in a graph.

```
from neo4j import GraphDatabase
from neo4j.time import Date


URI = "<database-uri>"
AUTH = ("<username>", "<password>")

def add_friend(driver, name, status, date, friend_name):
    driver.execute_query("""
        MERGE (p:Person {name: $name})
        MERGE (p)-[r:KNOWS {status: $status, since: $date}]->(friend:Person {name: $friend_name})
        """, name=name, status=status, date=date, friend_name=friend_name,
        database_="<database-name>",
    )

with GraphDatabase.driver(URI, auth=AUTH) as driver:
    # Create some :Person nodes linked by :KNOWS relationships
    add_friend(driver, name="Alice", status="BFF", date=Date.today(), friend_name="Bob")
    add_friend(driver, name="Bob", status="Fiends", date=Date.today(), friend_name="Sofia")
    add_friend(driver, name="Sofia", status="Acquaintances", date=Date.today(), friend_name="Sofia")

    # Follow :KNOWS relationships outgoing from Alice three times, return as path
    records, _, _ = driver.execute_query("""
        MATCH path=(:Person {name: $name})-[:KNOWS*3]->(:Person)
        RETURN path AS friendship_chain
```

```
        """, name="Alice",
        database_="<database-name>",
    )
    path = records[0]["friendship_chain"]

    print("-- Path breakdown --")
    for friendship in path:
        print("{name} is friends with {friend} ({status})".format(
            name=friendship.start_node.get("name"),
            friend=friendship.end_node.get("name"),
            status=friendship.get("status"),
        ))
```

For full documentation, see API documentation — Graph types — Path.

Extended types

The driver supports more types as query parameters, which get automatically mapped to one of the core types. Because of this conversion, and because the server does not know anything about the extended types, the driver will never return these types in results, but always their corresponding mapping.

Parameter Type	Mapped Python Type
tuple	`list`
bytearray	`bytes`
numpy `ndarray`	`list` (nested)
pandas `DataFrame`	`dict`
pandas `Series`	`list`
pandas `Array`	`list`

In general, if you are unsure about the type conversion that would happen on a given parameter, you can test it as in the following example:

```
import neo4j

with neo4j.GraphDatabase.driver(URI, auth=AUTH) as driver:
    type_in = ("foo", "bar")
    records, _, _ = driver.execute_query("RETURN $x AS type_out", x=type_in)
    type_out = records[0].get("type_out")
    print(type(type_out))  # <class 'list'>
    print(type_out)        # ['foo', 'bar']
```

Vector type

Introduced in 6.0

The type `Vector` maps to the Cypher type `VECTOR`. `Vector` objects are stored as contiguous blocks of memory (`bytes`), containing homogeneous values encoded in big-endian order.

You can create a `Vector` from various data types.

Some examples of valid `Vector` invocations

```
Vector([1, 2, 3], "i8")
Vector(b"\x00\x01\x00\x02", "i16")
Vector(b"\x01\x00\x02\x00", "i16", byteorder="little")
Vector(numpy.array([1, 2, 3]))
Vector(pyarrow.array([1, 2, 3]))
```

An iterable of floats/ints, and the type (`VectorDType`) of the individual entries.
Supported types are:

- `f32`: 32-bit floating point number (single)
- `f64`: 64-bit floating point number (double)
- `i8`: 8-bit integer
- `i16`: 16-bit integer
- `i32`: 32-bit integer
- `i64`: 64-bit integer

Raw bytes, and the type (`VectorDType`) to parse the bytes with.

When providing raw bytes, `byteorder` is optional and defaults to `"big"`.

A `numpy.ndarray`. No further parameters are accepted, and the `VectorDType` is inferred from the input.

A `pyarrow.Array`. No further parameters are accepted, and the `VectorDType` is inferred from the input.

Create and store a vector

```
from neo4j.vector import Vector

# Create a vector (parsing bytes as big-endian by default)
v = Vector(b"\x00\x01\x02\x03", "i16")
# or (less performant, but friendlier)
v = Vector([1, 513], "i16")

# Store the vector as a node property
driver.execute_query("MERGE (d:Doc {embedding: $vector})", vector=v)
```



Storing `VECTOR` objects in the database requires a server version >= 2025.10, Enterprise Edition.

Retrieve and work with a vector

```
from neo4j.vector import Vector

# Retrieve a vector
v = driver.execute_query(
    "MATCH (d:Doc) RETURN d.embedding AS v LIMIT 1",
    database_="neo4j",
).records[0]["v"]

# Work with the vector
print(f"Got vector of type {v.dtype} with {len(v)} elements: {v}")
print("raw bytes:", v.raw())
print("python list:", v.to_native())
print("as numpy array:", v.to_numpy())
```



Because Python has no native `float32` type, vectors of `float32` converted into native

Python lists become lists of `float64`, leading to potential representation imprecisions. To retain the `float32` representation, transform the result vector into a numpy array (`Vector.to_numpy`) or into a pyarrow array (`Vector.to_pyarrow`).

UUID

Introduced in 6.3 | Enterprise Edition

The Cypher type `UUID` maps directly to `uuid.UUID`.



The `UUID.is_safe` property is not preserved when a UUID is stored and retrieved from the database.

Store and work with UUID

```
import uuid

# Insert and read a UUID
uuid = driver.execute_query(
    "CREATE (d:Doc {id: $uuid}) RETURN d.id AS id LIMIT 1",
    uuid=uuid.uuid1(),
    database_="neo4j",
).records[0]["id"]

print(f"Got UUID: {uuid}")
```

UnsupportedType

Introduced in 6.0

The type `UnsupportedType` is used for data types returned by Cypher queries, but that the driver doesn't recognize. This happens when the client version is too old with respect to the server version.

Upgrade from older versions

This page contains the list of new features and breaking changes of the driver from version 5.28 to 6.x. For a full list of changes, see the driver changelog.

The latest driver version of the 6.x series is compatible with Neo4j server both 4.4, 5.x, and 202[56].x, so you can upgrade the driver before you upgrade the server. At the same time, the driver version 5.28 is forward compatible with Neo4j server 202[56].x, so you could also upgrade the server before the driver; however, given that it's easier to roll back an application upgrade than a server upgrade, it's recommended to start with the driver.



When upgrading the Neo4j server to a newer version, the Cypher queries in your application might also need updating. The Drivers Migration Assistent doesn't cover Cypher changes.
See Cypher → Deprecations, additions, and compatibility.

New features

▼ New type `Vector`

The type `Vector` allows for storing and retrieving Cypher `VECTOR` objects to/from the database. The `VECTOR` type is suitable for efficiently storing lists of homogeneous numbers, such as embeddings.

▼ *GQL status objects in errors*

Exception objects also expose errors as GQL-status objects. The main difference between Neo4j error codes and GQL error codes is that the latter are more granular. For more information, see Error handling.

▼ *Support for pandas 3*

As of version 6.2, the driver is compatible with `pandas` 3.x.

The driver might indirectly behave differently when used with pandas 3. For example, calling `Result.to_df` on a `Result` containing a string-only column produces a `pandas.DataFrame` with `dtype` `object` for this column. When using pandas 3, the `dtype` will be `StringDtype` instead.

Breaking changes and deprecations

Deprecated features are likely to be removed in the next major release.

Version	Message	Status
6.0	Environment • Dropped support for Python 3.7, 3.8, 3.9. • Package-alias `neo4j-driver` stopped receiving updates starting with 6.0. Install `neo4j` instead. • `setup.py` has been removed. Use a recent enough packaging/build tool supporting `pyproject.toml`.	Removed

Version	Message	Status
6.0	`Session` — Methods `.read_transaction()` and `.write_transaction()` have been removed in favor of `.execute_read()` and `.execute_write()`. Through the new methods, the first argument of transaction functions is a `ManagedTransaction` object. It behaves like a regular `Transaction` object, except it does not offer the `.commit()`, `.rollback()`, `.close()`, and `.closed()` methods.	Removed
6.0	Driver and session objects are not implicitly closed through `__del__()` (destructor) anymore. This behaviour was non-deterministic as there is no guarantee that the destructor would ever be called. A `ResourceWarning` is emitted instead. Call `driver.close()` explicitly or create the driver via a `with` statement.	Removed
6.0	`Driver` — The config option `trust` is not available anymore. The options `trusted_certificates` and `ssl_context` are available.	Removed
6.0	Bookmarks • The method `Session.last_bookmark()` has been removed in favor of `Session.last_bookmarks()`. The logic is similar, but the new method returns `Bookmarks` instead of `str`. • `Bookmark` has been removed in favor of `Bookmarks`. • `Driver.session()` no longer accepts raw string bookmarks as bookmarks argument. Use a `Bookmarks` object instead.	Removed
6.0	Imports • Import of the following modules has been removed without replacement, as they are internal and should not be used by client code: `neo4j.packstream`, `neo4j.routing`, `neo4j.config`, `neo4j.meta`, `neo4j.data`. `neo4j.meta.version` is exposed through `neo4j.__version__`. • Importing submodules from `neo4j.time` is not possible anymore. Import everything from `neo4j.time` directly instead. • Importing `neo4j.work` and its submodules is not possible anymore. Import everything from `neo4j` directly instead. • The following objects have been removed without replacement, as they are internal and should not be used by client code: `neo4j.spatial.hydrate_point`, `neo4j.spatial.dehydrate_point`, `neoj4.Config`, `neoj4.PoolConfig`, `neoj4.SessionConfig`, `neoj4.WorkspaceConfig`, `neo4j.data.DataDehydrator`, `neo4j.data.DataHydrator`. • The following modules have been removed: `neo4j.conf`, `neo4j.data`, `neo4j.meta`, `neo4j.packstream`, `neo4j.routing`, `neo4j.time.arithmetic`, `neo4j.time.clock_implementation`, `neo4j.time.hydration`, `neo4j.time.metaclasses`, `neo4j.work`, `neo4j.work.query`, `neo4j.work.summary` • `api.Version` has been removed as it's unused now. `ServerInfo.protocol_version` is now a `tuple[int, int]` instead of a `api.Version`. This should be drop-in replacement is most cases. • Make undocumented constants, helper functions, and other internal items private. For a full list, see the changelog.	Removed
6.0	Class `SummaryNotificationPosition` has been removed in favor of `SummaryInputPosition`.	Removed
6.0	`Neo4jError` — Method `.is_retriable()` has been removed in favor of `.is_retryable()`	Removed

Version	Message	Status
6.0	The magic method `Record.__getslice__()` has been removed. If you were calling it directly, use `Record.__getitem__(slice(…))` or simply `record[…]` instead.	Removed
6.0	Changed errors raised under certain circumstances. For a full list, see the changelog.	Changed
6.0	Graph type sets (`neo4j.graph.EntitySetView`) can no longer be indexed by legacy `id` (`int`, e.g., `graph.nodes[0]`). Use the `element_id` instead (`str`, e.g., `graph.nodes["…"]`).	Changed
5.0	`Node`, `Relationship` — Property `id` (int) is deprecated in favor of `element_id` (str). This also affects `Graph` objects, as indexing `graph.nodes[…]` and `graph.relationships[…]` with integers is deprecated in favor of indexing them with strings.	Deprecated
6.0	Notifications as Neo4j status codes are deprecated in favor of GQL status objects. As a result, • Object `SummaryNotification` is deprecated in favor of `GqlStatusObject`. • Objects `NotificationCategory` and `NotificationDisabledCategory` are deprecated in favor of `NotificationClassification` and `NotificationDisabledClassification`. • Attributes `ResultSummary.notifications` and `ResultSummary.summary_notifications` are deprecated in favor of `ResultSummary.gql_status_objects`. • Filtering notifications via `notifications_disabled_categories` is deprecated in favor of `notifications_disabled_classifications`.	Deprecated
6.0	Module `neo4j.time.ClockTime` and its accessors are deprecated.	Deprecated
6.0	Passing raw string bookmarks as `initial_bookmarks` to `GraphDatabase.bookmark_manager()` is deprecated. Use a `Bookmarks` object instead.	Deprecated

API documentation

GraphAcademy courses

Graph Data Modeling Fundamentals

Intermediate Cypher Queries

Building Neo4j Applications with Python

License

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)

You are free to

Share

copy and redistribute the material in any medium or format

Adapt

remix, transform, and build upon the material

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms

Attribution

You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial

You may not use the material for commercial purposes.

ShareAlike

If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions

You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

Notices

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permissions necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

See https://creativecommons.org/licenses/by-nc-sa/4.0/ for further details. The full license text is available at https://creativecommons.org/licenses/by-nc-sa/4.0/legalcode.