The Neo4j Python Driver Manual v5.0

Table of Contents

Quickstart . . . 1
  Installation . . . 1
  Connect to the database . . . 1
  Query the database . . . 1
  Run your own transactions . . . 2
  Close connections and sessions . . . 4
  API documentation . . . 4
Glossary . . . 5
=Regular workflow= . . . 7
  Installation . . . 8
    Install the driver . . . 8
    Get a Neo4j instance . . . 8
  Connection . . . 9
    Connect to the database . . . 9
    Connect to an Aura instance . . . 9
    Close connections . . . 10
    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 . . . 13
    A full example . . . 15
  Manipulate query results . . . 16
    Result as a list . . . 16
    Transform to Pandas DataFrame . . . 16
    Transform to graph . . . 17
    Custom transformers . . . 19
=Advanced usage= . . . 21
  Run your own transactions . . . 22
    Create a session . . . 22
    Run a managed transaction . . . 22
    Run an explicit transaction . . . 25
    Process query results . . . 27
    Session configuration . . . 28

Close sessions . . . 29
Explore the query execution summary. . . . 30
Retrieve the execution summary . . . 30
Query counters. . . . 30
Query execution plan . . . 31
Notifications . . . 32
Coordinate parallel transactions . . . 34
Bookmarks with `.execute_query()` . . . 34
Bookmarks within a single session . . . 34
Bookmarks across multiple sessions . . . 34
Mix `.execute_query()` and sessions . . . 37
Implement a custom `BookmarkManager` . . . 37
Run concurrent transactions . . . 38
Further query mechanisms. . . . 40
Implicit (or auto-commit) transactions . . . 40
Dynamic values in property keys, relationship types, and labels. . . . 41
Logging . . . 42
Performance recommendations . . . 43
Always specify the target database . . . 43
Be aware of the cost of transactions . . . 43
Don’t fetch large result sets all at once . . . 44
Route read queries to cluster readers . . . 47
Create indexes . . . 48
Profile queries. . . . 48
Specify node labels . . . 49
Batch data creation . . . 50
Use query parameters. . . . 50
Concurrency . . . 51
Use `MERGE` for creation only when needed . . . 51
Filter notifications . . . 51
=Reference=. . . . 52
Advanced connection information . . . 53
Connection URI . . . 53
Connection protocols and security. . . . 53
Authentication methods. . . . 53
Custom address resolver . . . 54
Further connection parameters. . . . 55
Data types and mapping to Cypher types. . . . 56
Core types . . . 56
Temporal types. . . . 56
Spatial types . . . 58

Graph types . . . 59
Extended types . . . 62
Exceptions . . . 63
API documentation . . . 64
=GraphAcademy courses= . . . 65
Graph Data Modeling Fundamentals . . . 66
Intermediate Cypher Queries . . . 67
Building Neo4j Applications with Python . . . 68

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. While this guide does not *require* you to be a seasoned Cypher querier, it is going to be easier to focus on the Python-specific bits if you already know some Cypher. For this reason, although this guide does *also* provide a gentle introduction to Cypher along the way, consider checking out Getting started → Cypher for a more detailed walkthrough of graph databases modelling and querying if this is your first approach. You may then apply that knowledge while following this guide to develop your Python application.

Installation

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 = "<URI for Neo4j database>"
AUTH = ("<Username>", "<Password>")

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

More info on connecting to a database →

Query the database

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

```
# Get the name of all 42 year-olds
records, summary, keys = driver.execute_query(
    "MATCH (p:Person {age: $age}) RETURN p.name AS name",
    age=42,
    database_="neo4j",
)

# Loop through results and do something with them
for person in records:
    print(person)

# 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 →

Run your own transactions

For more advanced use-cases, you can run transactions. Use the methods `Session.execute_read()` and `Session.execute_write()` to run managed transactions.

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

```python
from neo4j import GraphDatabase


URI = "<URI for Neo4j database>"
AUTH = ("<Username>", "<Password>")
employee_threshold=10


def main():
    with GraphDatabase.driver(URI, auth=AUTH) as driver:
        with driver.session(database="neo4j") 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()
```

More info on running transactions →

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.

```python
from neo4j import GraphDatabase


driver = GraphDatabase.driver(URI, auth=AUTH)
session = driver.session(database="neo4j")

# session/driver usage

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

API documentation

For in-depth information about driver features, check out the API documentation.

Glossary

LTS

A *Long Term Support* release is one guaranteed to be supported for a number of years. Neo4j 4.4 is LTS, and Neo4j 5 will also have an LTS version.

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.

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.

Install the driver

Use `pip` to install the Neo4j Python Driver (requires Python >= 3.7):

```
pip install neo4j
```

Always use the latest version of the driver, as it will always work both with the previous Neo4j LTS release and with the current and next major releases. The latest `5.x` driver supports connection to any Neo4j 5 and 4.4 instance, and will also be compatible with Neo4j 6. For a detailed list of changes across versions, see the driver's changelog.

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.

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 an authentication token.

```
from neo4j import GraphDatabase

# URI examples: "neo4j://localhost", "neo4j+s://xxx.databases.neo4j.io"
URI = "<URI for Neo4j database>"
AUTH = ("<Username>", "<Password>")

with GraphDatabase.driver(URI, auth=AUTH) as driver: ①
    driver.verify_connectivity() ②
    print("Connection established.")
```

① 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.

② 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.

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, the most 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 may 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.

Connect to an Aura instance

When you create an Aura instance, you may download a text file (a so-called *Dotenv file*) containing the connection information to the database in the form of 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-module to load them. We recommend the package `python-dotenv` for that purpose.

```python
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.

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()`.

`Driver.execute_query()` was introduced with the version 5.8 of the driver.
For queries with earlier versions, use sessions and transactions.

Write to the database

To create a node representing a person named `Alice`, use the Cypher clause `CREATE`:

Create a node representing a person named `Alice`

```
summary = driver.execute_query(
    "CREATE (:Person {name: $name})",  ①
    name="Alice",  ②
    database_="neo4j",  ③
).summary
print("Created {nodes_created} nodes in {time} ms.".format(
    nodes_created=summary.counters.nodes_created,
    time=summary.result_available_after
))
```

① The Cypher query

② A map of *query parameters*

③ Which database the query should be run against

Read from the database

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

Retrieve all `Person` *nodes*

```
records, summary, keys = driver.execute_query(
    "MATCH (p:Person) RETURN p.name AS name",
    database_="neo4j",
)

# Loop through results and do something with them
for record in records:  ①
    print(record.data())  # obtain record as dict

# Summary information  ②
print("The query `{query}` returned {records_count} records in {time} ms.".format(
    query=summary.query, records_count=len(records),
    time=summary.result_available_after
))
```

① `records` contains the result as an array of `Record` objects

② `summary` contains the summary of execution returned by the server

Update the database

To update a node'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_="neo4j",
)
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})  ①
    MATCH (bob:Person {name: $friend})  ②
    CREATE (alice)-[:KNOWS]->(bob)  ③
    """, name="Alice", friend="Bob",
    database_="neo4j",
)
print(f"Query counters: {summary.counters}.")
```

① Retrieve the person node named Alice and bind it to a variable alice

② Retrieve the person node named Bob and bind it to a variable bob

③ 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

```
records, summary, keys = driver.execute_query("""
    MATCH (p:Person {name: $name})
    DETACH DELETE p
    """, name="Alice",
    database_="neo4j",
)
print(f"Query counters: {summary.counters}.")
```

Query parameters

Do not hardcode or concatenate parameters directly into queries. Instead, always use placeholders and specify the Cypher parameters, as shown in the previous examples. 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 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_="neo4j",
)
```

Pass query parameters in a dictionary

```
parameters = {
    "name": "Alice",
    "age": 42
}
driver.execute_query(
    "MERGE (:Person {name: $name})",
    parameters_=parameters,
    database_="neo4j",
)
```

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

Because `.execute_query()` can potentially raise a number of different exceptions, the best way to handle errors is to catch all exceptions in a single `try/except` block:

```
try:
    driver.execute_query(...)
except Exception as e:
    ...  # handle exception
```

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 a number of attempts.

Query configuration

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

Database selection

It is recommended to **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_="neo4j",
)
```

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 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_="neo4j",
)
```

Although executing a *write* query in read mode likely 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 will be directed to the leader. In other words, there is no guarantee that a write query submitted in read mode will be rejected.

Run queries as a different user

You can execute a query under the security context of a different user with the parameter `impersonated_user_`, specifying the name of the user to impersonate. For this to work, the user under which the `Driver` was created needs to have the appropriate permissions. Impersonating a user is cheaper than creating a new `Driver` object.

```
driver.execute_query(
    "MATCH (p:Person) RETURN p.name",
    impersonated_user_="somebody_else",
    database_="neo4j",
)
```

When impersonating a user, the query is run within the complete security context of the impersonated user and not the authenticated user (i.e. home database, permissions, etc.).

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


URI = "<URI for Neo4j database>"
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_="neo4j",
            )

        # 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_="neo4j",
                )

        # 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_="neo4j",
        )
        # 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 Exception 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_="neo4j",
)
for person in records:  ①
    print(person)
    # person["name"] or person["age"] are also valid

# Some summary information  ②
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'] ③
```

① The result records as a list, so it is easy to loop through them.

② A summary of execution, with metadata and information about the result.

③ The `keys` available in the returned rows.

Transform to Pandas DataFrame

The driver can transform the result into a Pandas DataFrame. To achieve this, use the `.execute_query()` keyword argument `result_transformer_` and set it to `neo4j.Result.to_df`. This method is only available if 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_="neo4j",
    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`.

If you need 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. To achieve this, use the `.execute_query()` keyword argument `result_transformer_` and set it to `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`

```python
import pyvis
from neo4j import GraphDatabase
import neo4j


URI = "<URI for Neo4j database>"
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_="neo4j",
        )

        # 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_="neo4j",
        )

        # 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()
```

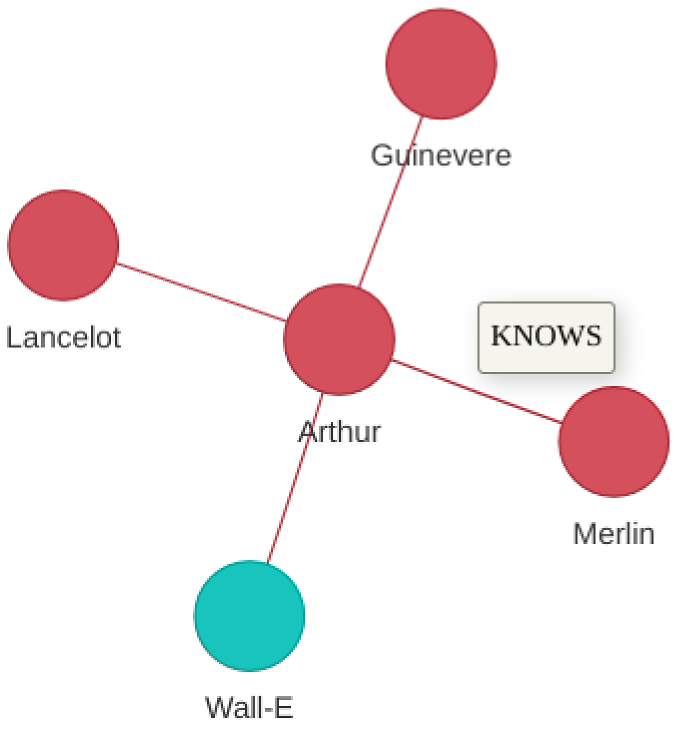


Figure 1. Graph visualization of example above

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`

```python
# 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_="neo4j",
    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`

```python
# 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_="neo4j",
    result_transformer_=exactly_5,
)
```

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.

```python
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 take full control over the transaction lifecycle. These are called *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 concrete 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="neo4j") as session:
    ...
```

Session creation 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 make sure each thread creates its own sessions.

Run a managed transaction

A transaction can contain any number of 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.

A managed transaction is created with the methods `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 actually carrying out the queries and processing the result.

Retrieve people whose name starts with `Al`.

```
def match_person_nodes(tx, name_filter): ③
    result = tx.run(""" ④
        MATCH (p:Person) WHERE p.name STARTS WITH $filter
        RETURN p.name AS name ORDER BY name
        """, filter=name_filter)
    return list(result)  # return a list of Record objects ⑤

with driver.session(database="neo4j") as session:  ①
    people = session.execute_read(  ②
        match_person_nodes,
        "Al",
    )
    for person in people:
        print(person.data())  # obtain dict representation
```

① 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.

② 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.

③ The transaction function callback is responsible of running queries.

④ Use the method `Result` object.

⑤ 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.

The methods `.execute_read()` and `.execute_write()` have replaced `.read_transaction()` and `.write_transaction()`, which are deprecated in version 5.x and will be removed in version 6.0.

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

```python
from neo4j import GraphDatabase


URI = "<URI for Neo4j database>"
AUTH = ("<Username>", "<Password>")
employee_threshold=10


def main():
    with GraphDatabase.driver(URI, auth=AUTH) as driver:
        with driver.session(database="neo4j") 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()
```

Should a transaction fail for a reason that the driver deems transient, it automatically retries to run the transaction function (with an exponentially increasing delay). For this reason, **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 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()` allows to exert 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="neo4j") 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="neo4j") 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.

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

```python
import neo4j


URI = "<URI for Neo4j database>"
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_ = "neo4j")
    return result[0]["id"]


def transfer_to_other_bank(driver, customer_id, other_bank_id, amount):
    with driver.session(database="neo4j") 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, while `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 from the result without consuming it. This leaves the record in the buffer for further processing.
`data(*keys)`	Return a JSON-like dump of the raw result. 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.

Name	Description
`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

You should **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 default database set in the Neo4j instance settings is used.

```
with driver.session(
    database="neo4j"
) 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 either `neo4j.READ_ACCESS` or `neo4j.WRITE_ACCESS`. Note that `.execute_read()` and `.execute_write()` automatically override the session's default access mode.

```
import neo4j

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

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

Similar remarks hold for the `.executeRead()` and `.executeWrite()` methods.

Run queries as a different user (impersonation)

You can execute a query under the security context of a different user with the parameter `impersonated_user`, specifying the name of the user to impersonate. For this to work, the user under which the `Driver` was created needs to have the appropriate permissions. Impersonating a user is cheaper than creating a new `Driver` object.

```
with driver.session(
    database="neo4j",
    impersonated_user="somebody_else"
) as session:
    ...
```

When impersonating a user, the query is run within the complete security context of the impersonated user and not the authenticated user (i.e., home database, permissions, etc.).

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="neo4j")

# session usage

session.close()
```

Explore the query execution summary

After all results coming from a query 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 object, under the `summary` key.

```
records, result_summary, keys = driver.execute_query("""
    UNWIND ["Alice", "Bob"] AS name
    MERGE (p:Person {name: name})
    """, database_="neo4j",
)
# 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**. This means that any record which has not yet been processed is not available any longer.

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

with driver.session(database="neo4j") 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 (p)-[:KNOWS]->(:Person {name: $friend})
    """, name="Mark", friend="Bob",
    database_="neo4j",
).summary
print(summary.counters)
"""
{'_contains_updates': True, 'labels_added': 2, 'relationships_created': 1,
 'nodes_created': 2, 'properties_set': 2}
"""
```

There are two additional boolean properties which 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 updated the `system` database

Query execution plan

If you prefix a query with `EXPLAIN`, the server will return the plan it *would* use to run the query, but will not actually run it. The plan is then available under the property `ResultSummary.plan`, and contains the list of Cypher operators that would be used to retrieve the result set. You may use this information to locate potential bottlenecks or room for performance improvements (for example through 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 will return 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.

For driver version >= 5.25 and server version >= 5.23, two forms of notifications are available (*Neo4j status codes* and *GQL status codes*). For earlier versions, only *Neo4j status codes* are available.
GQL status codes are planned to supersede Neo4j status codes.

Example 1. An unbounded shortest path raises a performance notification

Filter notifications

By default, the server analyses each query for all categories and severity of notifications. Starting from version 5.7, you can 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.

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. You can use any of those parameters either when creating a `Driver` instance, or when creating a session.

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="neo4j",
    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, which guarantees 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.

A *bookmark* is a token that represents 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.

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, so that you can trust that queries inside one 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.

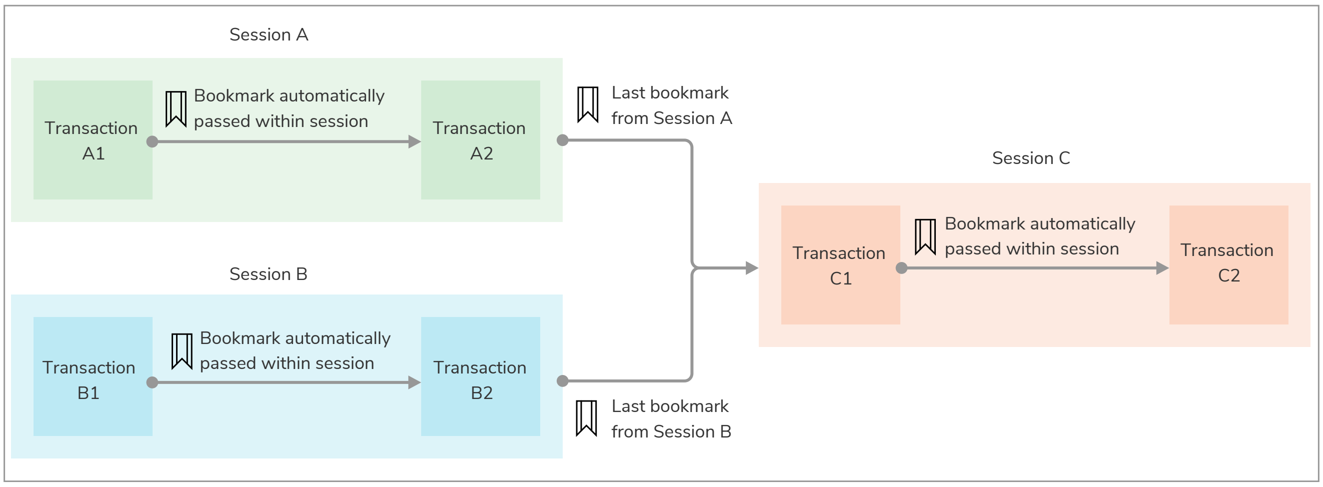
In the example below, `session_a` and `session_b` are allowed to run concurrently, while `session_c` waits until their results have been propagated. This guarantees the `Person` nodes `session_c` wants to act on actually exist.

Coordinate multiple sessions using bookmarks

```
from neo4j import GraphDatabase, Bookmarks


URI = "<URI for Neo4j database>"
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="neo4j") as session_a:
        session_a.execute_write(create_person, "Alice")
        session_a.execute_write(employ, "Alice", "Wayne Enterprises")
        saved_bookmarks += session_a.last_bookmarks()  ①

    # Create the second person and employment relationship
    with driver.session(database="neo4j") as session_b:
        session_b.execute_write(create_person, "Bob")
        session_b.execute_write(employ, "Bob", "LexCorp")
        saved_bookmarks += session_b.last_bookmarks()  ①

    # Create a friendship between the two people created above
    with driver.session(
        database="neo4j", bookmarks=saved_bookmarks
    ) as session_c:  ②
        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()
```

① Collect and combine bookmarks from different sessions using `Session.last_bookmarks()`, storing them in a Bookmarks object.

② Use them to initialize another session with the bookmarks parameter.


Session A
Transaction A1
Bookmark automatically passed within session
Transaction A2
Last bookmark from Session A
Session B
Transaction B1
Bookmark automatically passed within session
Transaction B2
Last bookmark from Session B
Session C
Transaction C1
Bookmark automatically passed within session
Transaction C2

The use of bookmarks can negatively impact performance, since all queries are forced to wait for the latest changes to be propagated across the cluster. For simple use-cases, 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, you can use the parameter bookmark_manager upon session creation, setting it to 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 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`

```python
import asyncio
from neo4j import AsyncGraphDatabase


URI = "<URI for Neo4j database>"
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_="neo4j"
        )
        names = [record["name"] for record in records]
        print(names)


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

An async driver example with transaction functions

```python
import asyncio
from neo4j import AsyncGraphDatabase


URI = "<URI for Neo4j database>"
AUTH = ("<Username>", "<Password>")


async def main():
    async with AsyncGraphDatabase.driver(URI, auth=AUTH) as driver:
        async with driver.session(database="neo4j") 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 also possible, 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

This is the most basic and limited form with which to run a Cypher query. The driver will not automatically retry implicit transactions, as it does instead for queries run with `execute_query()` and with managed transactions. Implicit transactions should only be used when the other driver query interfaces do not fit the purpose, or for quick prototyping.

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="neo4j") as session:
    session.run("CREATE (a:Person {name: $name})", name="Licia")
```

An implicit transaction gets committed *at the latest* when the session is destroyed, or before another transaction is executed within the same session. Other than that, there is no clear guarantee on when exactly an implicit transaction will be committed during the lifetime of a session. To ensure an implicit transaction is committed, you can call the `.consume()` method on its result.

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.

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

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="neo4j") 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="neo4j") 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 can be used for literals and expressions as well as node and relationship ids, they cannot be used for the following constructs:

- property keys, so `MATCH (n) WHERE n.$param = 'something'` is invalid;
- relationship types, so `MATCH (n)-[:$param]→(m)` is invalid;
- labels, so `MATCH (n:$param)` is invalid.

For those queries, you are forced to use string concatenation. To protect against link:Cypher injections, you should enclose the dynamic values in backticks and escape them yourself. Notice that Cypher processes Unicode, so take care of the Unicode literal `\u0060` as well.

Manually escaping dynamic labels before concatenation.

```
label = "Person\\u0060n"
# convert \u0060 to literal backtick and then escape backticks
escaped_label = label.replace("\\u0060", "`").replace("`", "``")

driver.execute_query(
    f"MATCH (p:`{escaped_label}` {{name: $name}}) RETURN p.name",
    name="Alice",
    database_="neo4j"
)
```

Another workaround, which avoids string concatenation, is using 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_="neo4j"
)
```

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

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_="<DB NAME>")
```

```
driver.session(database="<DB NAME>")
```

Bad practices

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

```
driver.session()
```

Be aware of the cost of transactions

When submitting queries through `.execute_query()` or through `.execute_read/write()`, the server automatically wraps them into a transaction. This behavior ensures that the database always ends up in a consistent state, regardless of what happens during the execution of a transaction (power outages, software crashes, etc).

Creating a safe execution context around a number of queries yields an overhead that is not present if the driver just shoots queries at the server and hopes they will get through. The overhead is small, but can add up as the number of queries increases. For this reason, if your use case values throughput more than data integrity, you may extract further performance by running all queries within a single (auto-commit) transaction. You do this by creating a session and using `session.run()` to run as many queries as needed.

Privilege throughput over data integrity

```
with driver.session(database="neo4j") as session:
    for i in range(1000):
        session.run("<QUERY>")
```

Privilege data integrity over throughput

```
for i in range(1000):
    driver.execute_query("<QUERY>")
    # or session.execute_read/write() calls
```

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.

Example 2. Comparison between eager and lazy loading

Eager loading	Lazy loading
• The server has to read all 250 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 250 records. • The client has to hold in memory all 250 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

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



URI = "<URI for Neo4j database>"
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='neo4j') as session:
        processed_result = session.execute_read(process_records)


def eager_loading(driver):
    log('Submit query')
    records, _, _ = driver.execute_query(slow_query, database_='neo4j')

    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  ①
[1718014256.71] Processing record 1
[1718014257.21] Processing record 2
...
[1718014395.42] Processing record 249
[1718014395.92] Peak memory usage: 786254 bytes
[1719984711.39] --- 135.9284942150116 seconds ---

[1718014395.92] EAGER LOADING (execute_query)
[1718014395.92] Submit query
[1718014419.82] Processing record 0  ②
[1718014420.33] Processing record 1
[1718014420.83] Processing record 2
...
[1718014544.52] Processing record 249
[1718014545.02] Peak memory usage: 89587150 bytes  ③
[1719984861.09] --- 149.70468592643738 seconds ---  ④
```

① With lazy loading, the first record is quickly available.

② With eager loading, the first record is available ~25 seconds after the query has been submitted (i.e. after the server has retrieved all 250 records).

③ Memory usage is larger with eager loading than with lazy loading, because the application materializes a list of 250 records.

④ The total running time is lower with lazy loading, because while the client processes records the server can fetch the next ones. With lazy loading, the client could also stop requesting records after some condition is met (by calling `.consume()` on the `Result`), saving time and resources.

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 allows to bound memory consumption on the client side. It doesn't always bound memory consumption on the server side though: 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 may deteriorate performance.

Route read queries to cluster readers

In a cluster, **route read queries to** secondary nodes. 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`. The server output is available in the `profile` property of the `ResultSummary` object.

```
_, 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
"""
```

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. The server output is available in the `plan` property of the `ResultSummary` object.

```
_, 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: ?
"""
```

Specify node labels

Specify node labels in all queries. This allows the query planner to work much 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="<DB 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="<DB 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("""
    WITH $numbers AS batch
    UNWIND batch AS node
    MERGE (n:Number)
    SET n.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="<DB 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="<DB 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 `MATCH` the pattern, and only then can it `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. If you are unsure, ask the database administrator.

Authentication methods

Basic authentication (default)

The basic authentication scheme relies on traditional username and password. These can either be the credentials for your local installation, or the ones provided with an Aura instance.

```
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.

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` objects (or values that can be used to construct `Address` objects)

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

```python
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)
```

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

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`).

Sub-second values are measured to nanosecond precision and the types are mostly compatible with pytz. Some timezones (e.g., `pytz.utc`) work exclusively with the built-in `datetime.datetime`.

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†
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

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


URI = "<URI for Neo4j database>"
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.

```python
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.

```python
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.

```python
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.

```python
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.

```python
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).

```python
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 may not 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.

```python
from neo4j import GraphDatabase


URI = "<URI for Neo4j database>"
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_="neo4j",
    )
    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.

```python
from neo4j import GraphDatabase


URI = "<URI for Neo4j database>"
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 = "<URI for Neo4j database>"
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_="neo4j",
    )

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_="neo4j",
    )
    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']
```

Exceptions

The driver can raise a number of different exceptions. A full list is available in the API documentation. For a list of errors the server can return, see the Status code page.

Table 1. Root exception types

Classification	Description
`Neo4jError`	Errors reported by the Neo4j server (e.g., wrong Cypher syntax, bad connection, wrong credentials, …)
`DriverError`	Errors reported by the driver (e.g., bad usage of parameters or transactions, improper handling of results, …)

Some server errors are marked as safe to retry without need to alter the original request. Examples of such errors are deadlocks, memory issues, or connectivity issues. All driver's exception types implement the method `.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.

API documentation

=GraphAcademy courses=

Graph Data Modeling Fundamentals

Intermediate Cypher Queries

Building Neo4j Applications with Python

License