Build applications with Neo4j and Python

Table of Contents

The Python driver allows you to connect and interact with a Neo4j database through a Python application.

	You need a running Neo4j database to use the driver with it. If you do not have one yet, get an instance for free.

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.

```python
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 by creating a session and using the methods `Session.execute_read()` and `Session.execute_write()`. Do not hardcode or concatenate parameters: use placeholders and specify the parameters as keyword arguments.

Get the name of all 42 year-olds

```
def match_person_nodes(tx, age):
    result = tx.run(
        "MATCH (p:Person {age: $age}) RETURN p.name AS name",
        age=age)
    records = list(result)
    summary = result.consume()
    return records, summary

with driver.session(database="neo4j") as session:
    records, summary = session.execute_read(match_person_nodes, age=42)

# 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,
))

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

More info on querying the database →

Close connections and sessions

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

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

Glossary

LTS

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

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j’s graph query language that lets you retrieve data from the graph. 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 Manual — 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.

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.

> 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. Take a note of the connection URI and of the login credentials.

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

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.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j's graph query language that lets you retrieve data from the graph. 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 Manual — 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.

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

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

Creating a `Driver` instance **(1)** only provides information on *how* to access the database, but does not actually *establish* a connection. Connection is instead deferred to when the first query is executed. To verify immediately that the driver can connect to the database (valid credentials, compatible versions, etc), use the `.verify_connectivity()` method **(2)** 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 subsequnt task, the most common choice is to let the program crash should an exception occurr during connection.

`Driver` objects are *immutable*, *thread-safe*, and fairly *expensive* to create. Share them across threads (but not across processes) and use impersonation to query the database with a different user. If you want to alter a `Driver` configuration, you will need to create a new object.

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.

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.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j’s graph query language that lets you retrieve data from the graph. 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 Manual — 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.

Query the database

You can run a query against the database using Cypher and the methods `Session.execute_read()` and `Session.execute_write()`.

Write to the database

To create a node representing a person named `Alice`, use the `Session.execute_write()` method in combination with the `MERGE` clause:

Create a node representing a person named `Alice`

```
def create_person(tx, name):  ③
    result = tx.run(
        "MERGE (:Person {name: $name})",  ①
        name=name  ②
    )
    summary = result.consume()
    return summary

with driver.session(database="neo4j") as session:
    summary = session.execute_write(create_person, name="Alice")  ④

    print("Created {nodes_created} nodes in {time} ms.".format(
        nodes_created=summary.counters.nodes_created,
        time=summary.result_available_after
    ))
```

where **(1)** specifies the Cypher query and **(2)** is a *query parameter*. The query and its parameters are wrapped in a transaction function **(3)**, which is passed as callback to `Session.execute_write()` together with an arbitrary numbers of keyword arguments **(4)**.

`MERGE` creates a new node matching the requirements unless one already exists, in which case nothing is done. For strict node creation, use the `CREATE` clause.

Read from the database

To retrieve information from the database, use the `Session.execute_read()` method in combination with the `MATCH` clause:

Retrieve all `Person` *nodes*

```
def get_people(tx):
    result = tx.run("MATCH (p:Person) RETURN p.name AS name")
    records = list(result)  # a list of Record objects
    summary = result.consume()
    return records, summary

with driver.session(database="neo4j") as session:
    records, summary = session.execute_read(get_people)

    # 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
    ))

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

where `records` contains the actual result as a list of `ResultSummary` object, containing a summary of execution from the server.

Update the database

To update a node's information in the database, use `MATCH` together with the `SET` clause in a `Session.execute_write()` call:

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

```
def update_person(tx, name, age):
    result = tx.run("""
        MATCH (p:Person {name: $name})
        SET p.age = $age
        """, name=name, age=age
    )
    summary = result.consume()
    return summary

with driver.session(database="neo4j") as session:
    summary = session.execute_write(update_person, name="Alice", age=42)
    print(f"Query counters: {summary.counters}.")
```

To create new nodes and relationships linking it to an already existing node, use a combination of `MATCH` and `MERGE` in a `Session.execute_write()` call:

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

```
def update_person(tx, name, friend):
    result = tx.run("""
        MATCH (p:Person {name: $name})
        MERGE (friend:Person {name: $friend_name})
        MERGE (p)-[:KNOWS]->(friend)
        """, name="Alice", friend_name="Bob"
    )
    summary = result.consume()
    return summary

with driver.session(database="neo4j") as session:
    summary = session.execute_write(update_person,
                                    name="Alice", friend="Bob")
    print(f"Query counters: {summary.counters}.")
```

It might feel tempting to create new relationships with a single `MERGE` clause, such as: `MERGE (:Person {name: "Alice"})-[KNOWS]->(:Person {name: "Bob"})`.
However, this would result in the creation of an extra `Alice` node, so that you would end up with unintended duplicate records. To avoid this, always `MATCH` the elements that you want to update, and use the resulting reference in the `MERGE` clause (as shown in the previous example). See Understanding how MERGE works.

Delete from the database

To remove a node and any relationship attached to it, use `DETACH DELETE` in a `Session.execute_write()` call:

Remove the `Alice` node

```python
def delete_person(tx, name):
    result = tx.run("""
        MATCH (p:Person {name: $name})
        DETACH DELETE p
        """, name="Alice"
    )
    summary = result.consume()
    return summary

with driver.session(database="neo4j") as session:
    summary = session.execute_write(delete_person, name="Alice")
    print(f"Query counters: {summary.counters}.")
```

How to pass parameters to queries

Do not hardcode or concatenate parameters directly into queries. Instead, always use placeholders and specify the Cypher parameters as keyword arguments or in a dictionary, 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**: protecting against Cypher injection.

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

Anatomy of a query

Before you can run a query, you need to obtain a session from the driver with the `Driver.session()` method **(1)**. The `database` parameter is optional but recommended for performance, and further configuration parameters can be included.

Within a session, use the methods `Session.execute_write()` **(2)**, depending on whether you want to retrieve data from the database or alter it. Both methods take a *transaction function* callback **(3)** and an arbitrary number of positional and keyword arguments **(4)** which are handed down to the transaction function.

The transaction function **(5)** is responsible for actually carrying out the queries and processing the result. Queries are specified with the `Result` object. You can then process the result **(7)** using any of the `Result` methods, or simply casting it to `list`.

Retrieve people whose name starts with `Al`.

```
def match_person_nodes(tx, name_filter): ⑤
    result = tx.run(""" ⑥
        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 ⑦

with driver.session(database="neo4j") as session:  ①
    people = session.execute_read(  ②
        match_person_nodes, ③
        "Al", ④
    )
    for person in people:
        print(person.data())  # obtain dict representation
```

You can find more information about sessions and transactions in the section Run your own transactions.

The driver may automatically retry to run a failed transaction. 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 how many times they might 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 full example

```python
from neo4j import GraphDatabase


URI = "<URI to 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}]

def create_person(tx, person):
    tx.run("MERGE (p:Person {name: $person.name, age: $person.age})",
           person=person)

def add_friends(tx, person):
    tx.run("""
        MATCH (p:Person {name: $person.name})
        UNWIND $person.friends AS friend_name
        MATCH (friend:Person {name: friend_name})
        MERGE (p)-[:KNOWS]->(friend)
        """, person=person
    )

def get_friends(tx, name, age):
    result = tx.run("""
        MATCH (p:Person {name: $name})-[:KNOWS]-(friend:Person)
        WHERE friend.age < $age
        RETURN friend
        """, name="Alice", age=40
    )
    records = list(result)  # transform to a list of Node objects
    summary = result.consume()
    return records, summary


with GraphDatabase.driver(URI, auth=AUTH) as driver:
    with driver.session(database="neo4j") as session:
        # Create some nodes
        for person in people:
            session.execute_write(create_person, person)

        # Create some relationships
        for person in people:
            if person.get("friends"):
                session.execute_write(add_friends, person)

        # Retrieve Alice's friends who are under 40
        records, summary = session.execute_read(get_friends, "Alice", 40)

        # 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
        ))

        # Loop through results and do something with them
        for person in records:
            print(person.data())  # get a dict view
```

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.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j’s graph query language that lets you retrieve data from the graph. 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 Manual — 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.

Manipulate query results

In the basic query workflows examples, transaction functions never returned a query result directly. Instead, they always *manipulated* it in some way: either casting it to list, or calling a specific method. This is because the driver's output of a query is a `Result` object, which does not directly contain the result records. Rather, it encapsulates the Cypher result in a rich data structure that requires some parsing within the transaction function.

This section shows how to work with a `Result` object so as to extract data in the form that is most convenient for your application.

Manipulate the result

When running a query, **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*. There is thus no way to retrieve a previous record from the stream, unless manually saved in an auxiliary data structure. When no more records are available, the result is *exhausted*.

It is up to the transaction function to do all the processing. To process a result, you have two main options:

1. cast the result object to `list`, which yields a list of `Record` objects. Use the `.data()` method on a `Record` to obtain a dict-like view, or the `.get(key, default=None)` method to retrieve the value for a given key.

   Processing result by casting it to list or with list comprehension

   ```
   def match_person_nodes(tx, name_filter):
       result = tx.run("""
           MATCH (p:Person) WHERE p.name STARTS WITH $filter
           RETURN p.name as name ORDER BY name
           """, filter=name_filter)
       return list(result)
       # or: return [record["name"] for record in result]

   with driver.session(database="neo4j") as session:
       people = session.execute_read(match_person_nodes, "Al")

       for person in people:
           print(person.data())  # or person.get("name")
   ```

2. use any of the of the `Result` methods. The ones most commonly needed are listed in the table below.

   Table 1. An essential list of `Result` methods.

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, and `default` allows to specify a value for nodes lacking that property.
`fetch(n)`	Return up to `n` records from the result.

Name	Description
`single(strict=False)`	Return the next and only remaining record, or `None`. Calling this method always exhausts the result. If more than one record is available, - `strict==False` — a warning is generated and the first of these is returned; - `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 purposes.
`consume()`	Return the query `ResultSummary`. It exhausts the result, so should only be called when data processing is over.

See the API documentation for a complete list of `Result` methods.

A transaction function must not return the `neo4j.Result` 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.

Examples

Processing result with `single` and `consume`.

```
def get_single_person(tx, name):
    """Get a single record (or an exception) and the summary from a result."""

    result = tx.run("MATCH (a:Person {name: $name}) RETURN a.name AS name",
                    name=name)
    record = result.single(strict=True)
    summary = result.consume()
    return record, summary


with driver.session(database="neo4j") as session:
    record, summary = session.execute_read(
        get_single_person, name="Alice"
    )
    print(record)
```

Processing result with `fetch` and `peek`.

```python
def get_exactly_5_people(tx):
    result = tx.run("MATCH (a:Person) RETURN a.name AS name")
    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


with driver.session(database="neo4j") as session:
    records = session.execute_read(get_exactly_5_people)
    print(records)
```

Transform to pandas DataFrame

The method `.to_df()` converts the result into a pandas DataFrame. This method is only available if the `pandas` library is installed.

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

```python
def pandas_df(tx):
    result = tx.run("UNWIND range(1, 10) AS n RETURN n, n+1 AS m")
    return result.to_df()

with driver.session(database="neo4j") as session:
    df = session.execute_read(pandas_df)
```

This method accepts two optional arguments:

- `expand` — If `True`, some data structures in the result are 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`, are converted to `pandas.Timestamp`.

Transform to graph

The method `.graph()` converts the result into a graph. To make the most out of this method, your query should return a graph-like result instead of a single column. The resulting graph object exposes 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


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')


def main():
    URI = "neo4j://localhost:7687"
    AUTH = ("neo4j", "secretgraph")

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

        friends_list = [("Arthur", "Guinevre"),
                        ("Arthur", "Lancelot"),
                        ("Arthur", "Merlin")]

        with driver.session(database="neo4j") as session:
            for pair in friends_list:
                session.execute_write(create_friends, pair[0], pair[1])
            session.execute_write(create_film, "Wall-E")
            session.execute_write(like_film, "Wall-E", "Arthur")
            graph = session.execute_read(get_person_graph, "Arthur")

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


def create_friends(tx, name, friend_name):
    tx.run("""
        MERGE (a:Person {name: $name})
        MERGE (a)-[:KNOWS]->(friend:Person {name: $friend_name})
        """, name=name, friend_name=friend_name
    )


def create_film(tx, title):
    tx.run("MERGE (film:Film {title: $title})", title=title)


def like_film(tx, title, person_name):
    tx.run("""
        MATCH (film:Film {title: $title})
        MATCH (liker:Person {name: $person_name})
        MERGE (liker)-[:LIKES]->(film)
        """, title=title, person_name=person_name,
    )


def get_person_graph(tx, name):
    result = tx.run("""
        MATCH (a:Person {name: $name})-[r]-(b)
        RETURN a,r,b
        """, name=name
    )
    return result.graph()


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

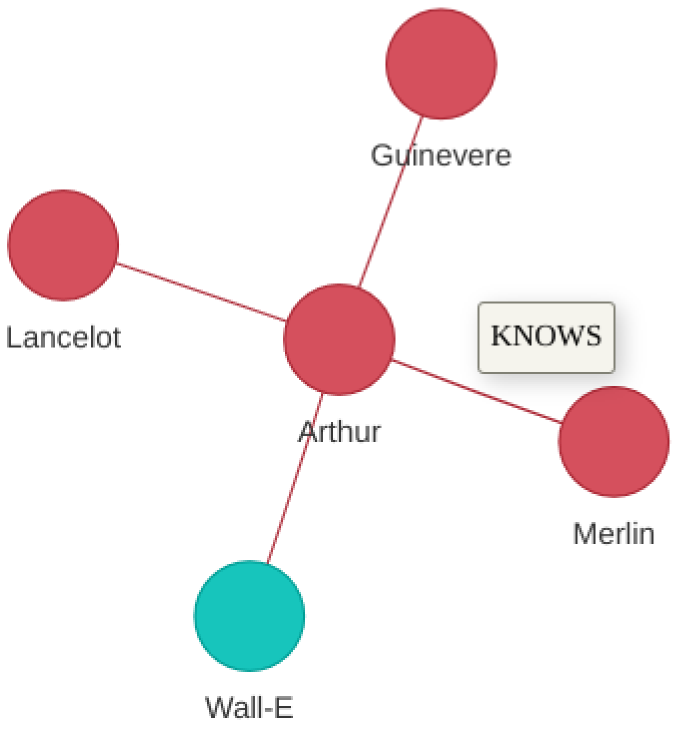


Figure 1. Graph visualization of example above

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.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j's graph query language that lets you retrieve data from the graph. 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 Manual — 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.

Run your own transactions

When querying the database with transaction functions, the driver automatically creates a **transaction**. You can easily include multiple queries in a single transaction function, but to design more involved use-cases it is important to understand how transactions work within the driver.

Obtain a session

Before running a transaction, you need to obtain a *session*. Sessions act as concrete query channels between the driver and the server. In particular, sessions borrow connections from the *connection pool* as needed, 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: 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 **(1)** and an arbitrary number of positional and keyword arguments **(2)** which are handed down to the transaction function. The transaction function **(3)** is responsible for actually carrying out the queries and processing the result. Queries are specified with the `Result` object. You can then process the result **(5)** using any of the `Result` methods, or simply casting it to list.

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

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

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

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

```python
from neo4j import GraphDatabase


URI = "neo4j://localhost"
AUTH = ("neo4j", "secret")
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 = "Thor"+str(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 existing 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:
        from neo4j.time import Date
        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, date=Date.today()
        )

    # 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.

Further 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.
- 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"})
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 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
```

Closing an explicit transaction can either happen automatically at the end of a `with` block, or can be explicitly controlled through the methods `Transaction.close()`.

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.

```python
from os import sleep
import neo4j


def transfer_to_other_bank(driver, customer_id, other_bank_id, amount):
    with driver.session(database="neo4j") as session:
        tx = session.begin_transaction()
        # or just use a `with` context on `tx` instead of try/finally
        try:
            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
        finally:
            tx.close()  # rolls back if not yet committed


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


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)
```

Session configuration

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 does not have to resolve the home database first. If no database is given, the default database set in the Neo4j instance settings is used.

```
import neo4j

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

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 will be routed to any node of a cluster, whereas *write* ones will be directed to the leader. Still, depending on the server version and settings, the server might allow none, some, or all *write* statements to be executed even in *read* transactions.

Similar remarks hold for the `.execute_read()` and `.execute_write()` 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()
```

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.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j's graph query language that lets you retrieve data from the graph. 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 Manual — 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.

Coordinate parallel transactions

When working with a Neo4j cluster, the driver automatically enforces causal consistency for transactions within the same session, 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.**

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 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
    ...
```

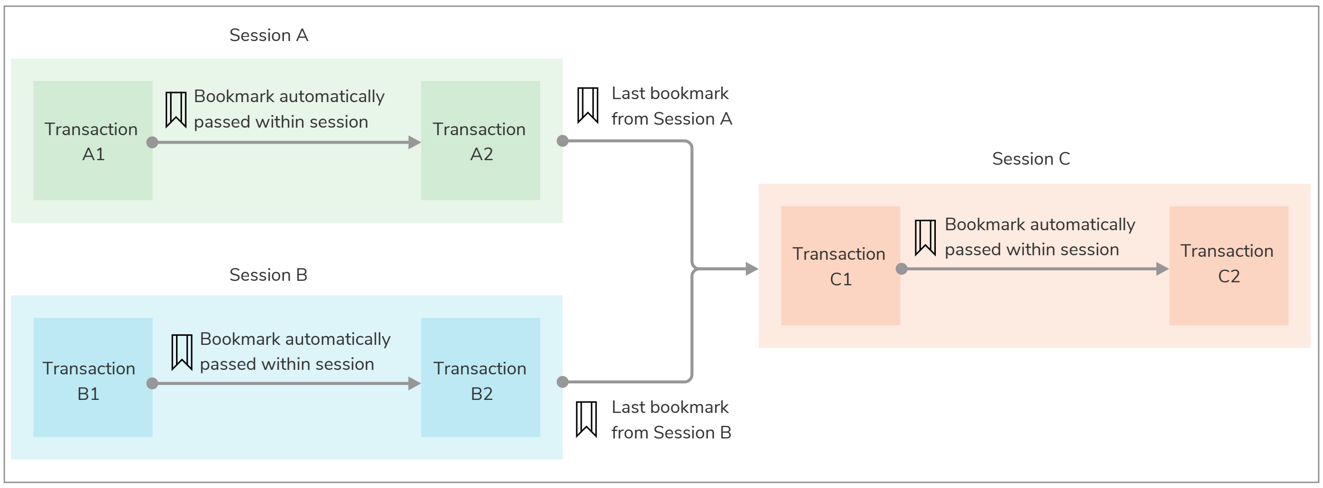
Bookmarks across multiple sessions

If your application uses multiple sessions instead, you may need to ensure that one session has completed all its transactions before another session is allowed to run its queries. In those cases, you can collect the bookmarks from some sessions using the method `Session.last_bookmarks()` **(1)**, **(2)**, store them into a `Bookmarks` object, and use them to initialize another session with the `bookmarks` parameter **(3)**.

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

```python
from neo4j import GraphDatabase, Bookmarks


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

def main():
    with GraphDatabase.driver(URI, auth=AUTH) as driver:
        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() as session_a:
        session_a.execute_write(create_person, "Alice")
        session_a.execute_write(employ, "Alice", "Wayne Enterprises")
        saved_bookmarks += session_a.last_bookmarks()

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

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


# Create a person node.
def create_person(tx, name):
    tx.run("CREATE (: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()
```


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.

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.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j's graph query language that lets you retrieve data from the graph. 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 Manual — 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.

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

```python
import asyncio
from neo4j import AsyncGraphDatabase


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


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


async def main():
    async with AsyncGraphDatabase.driver(URI, auth=AUTH) as driver:
        async with driver.session(database="neo4j") as session:
            records = await session.read_transaction(get_people)
            print(records)


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.

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.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j’s graph query language that lets you retrieve data from the graph. 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 Manual — 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.

Further query mechanisms

Auto-commit transactions

This is the most basic and limited form with which to run a Cypher query. The driver will not automatically retry auto-commit transactions, as it does instead for queries run with managed transactions.

You run an auto-commit transaction with the method `Session.run()`. It returns a `Result` object that needs to be processed with the related methods.

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

An auto-commit 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 auto-commit transaction will be committed during the lifetime of a session. To ensure an auto-commit transaction is committed, you can call the `.consume()` method on its result.

Auto-commit transactions should only be used when transaction functions do not fit the purpose, or for quick prototyping.

This method is the only one that can be used for `PERIODIC COMMIT` queries.

Configure auto-commit transactions with the `Query` object

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() as session:
    query = Query("CREATE (a:Person {name: $name})",
                  timeout=1.0,
                  metadata={"app_name": "people"})
    result = session.run(query, name=name)
```

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

with driver.session(database="neo4j") as session:
    session.run(
        f"MATCH (p:`{escaped_label}` {{name: $name}}) RETURN p.name",
        name="Alice",
    )
```

Another workaround, which avoids string concatenation, is using the APOC procedure `apoc.merge.node`. It supports dynamic labels and property keys, but only for node insertion.

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

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

with driver.session(database="neo4j") as session:
    session.run(
        "CALL apoc.merge.node($labels, $properties)",
        labels=[label], properties={property_key: "Alice"},
    )
```

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)
```

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.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j's graph query language that lets you retrieve data from the graph. 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 Manual — 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.

Performance recommendations

- **Specify the target database on all queries** 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 session, but becomes significant over hundreds of sessions.

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

```
# Bad practice
driver.session()
```

- **Use query parameters** instead of hardcoding or concatenating values into queries. This allows the database to properly cache queries.

```
# Good practice
session.run("MATCH (p:Person {name: $name}) RETURN p", name="Alice")
```

```
# Bad practice
session.run("MATCH (p:Person {name: 'Alice'}) RETURN p")
session.run("MATCH (p:Person {name: " + name + "}) RETURN p")
```

- **Specify node labels** in all queries. To learn how to combine labels, see Cypher — Label expressions.

```
# Good practice
session.run("MATCH (p:Person|Animal {name: $name}) RETURN p", name="Alice")
```

```
# Bad practice
session.run("MATCH (p {name: $name}) RETURN p", name="Alice")
```

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

```
# Good practice
numbers = [{"value": random()} for _ in range(10000)]
session.run("""
    WITH $numbers AS batch
    UNWIND batch AS node
    MERGE (n:Number)
    SET n.value = node.value
    """, numbers=numbers,
)
```

```
# Bad practice
for _ in range(10000):
    session.run("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.

- **Filter for properties inline**, as opposed to filtering in the `WHERE` clause.

```
# Good practice
session.run("MATCH (p:Person {name: $name}) RETURN p", name="Alice")
```

```
# Bad practice
session.run("MATCH (p:Person) WHERE p.name = $name RETURN p", name="Alice")
```

- **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 function, for both nodes and relationships. For more information, see Indexes for search performance.

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

- 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.

```
res = session.run("PROFILE MATCH (p {name: $name}) RETURN p", name="Alice")
summary = res.consume()
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.

```
res = session.run("EXPLAIN MATCH (p {name: $name}) RETURN p", name="Alice")
summary = res.consume()
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: ?
"""
```

- 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.

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.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j's graph query language that lets you retrieve data from the graph. 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 Manual — 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.

Advanced connection information

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 a `BoltDriver` object and `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=(user, 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.

If authentication is disabled on the server, the `auth` 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)

In the following example, the connection to `example.com` on port `9999` is resolved to `localhost` on port `7687`:

```
import neo4j


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

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

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


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

Further connection parameters

You can find all `Driver` configuration parameters in 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.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j's graph query language that lets you retrieve data from the graph. 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 Manual — 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.

Data types and mapping to Cypher types

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

Core data 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`.

Cypher Type	Python Type
Date	`neo4j.time.Date`
Time	`neo4j.time.Time`†
LocalTime	`neo4j.time.Time`††
DateTime	`neo4j.time.DateTime`†
LocalDateTime	`neo4j.time.DateTime`††
Duration	`neo4j.time.Duration`

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

How to use temporal types in queries

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


URI = "neo4j://localhost:7687"
AUTH = ("neo4j", "secretgraph")


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:
    with driver.session() as session:
        result = session.run("""
            MERGE (a:Person {name: $name})
            MERGE (b:Person {name: $friend})
            MERGE (a)-[friendship:KNOWS]->(b)
            SET friendship.since = $friends_since
            RETURN friendship.since
            """, name="Alice", friend="Bob",
            friends_since=friends_since  # or friends_since=py_friends_since
        )
        records = list(result)
        out_datetime = records[0]["friendship.since"]
        print(out_datetime)  # 1999-11-23T07:47:00.000004123-05:00

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

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

Date

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

```
from neo4j.time import Date

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

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

Time

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

```
from neo4j.time import Time

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

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

`LocalTime`

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

```
from neo4j.time import Time

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

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

`DateTime`

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

```
from neo4j.time import DateTime
import pytz

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

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

`LocalDateTime`

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

```
from neo4j.time import DateTime

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

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

`Duration`

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

```
from neo4j.time import Duration

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

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

Spatial types

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

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

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

For full documentation, see API documentation — Spatial types.

CartesianPoint

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

```python
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.

```python
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 Work with the result — 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.

```python
from neo4j import GraphDatabase


URI = "neo4j://localhost:7687"
AUTH = ("neo4j", "secretgraph")

with GraphDatabase.driver(URI, auth=AUTH) as driver:
    with driver.session() as session:
        result = session.run(
            "MERGE (p:Person {name: $name}) RETURN p AS person",
            name="Alice"
        )
        record = result.single()
        node = record["person"]
        print(f"Node ID: {node.element_id}\n"
              f"Labels: {node.labels}\n"
              f"Properties: {node.items()}\n"
        )

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

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

Relationship

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

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


URI = "neo4j://localhost:7687"
AUTH = ("neo4j", "secretgraph")

with GraphDatabase.driver(URI, auth=AUTH) as driver:
    with driver.session() as session:
        result = session.run("""
            MERGE (p:Person {name: $name})
            MERGE (friend:Person {name: $friend_name})
            MERGE (p)-[r:KNOWS]->(friend)
            SET r.status = $status, r.since = $date
            RETURN r AS friendship
            """, name="Alice", status="BFF", date=Date.today(), friend_name="Bob",
        )
        record = result.single()
        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.

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


URI = "neo4j://localhost:7687"
AUTH = ("neo4j", "secretgraph")

def add_friend(tx, name, status, date, friend_name):
    tx.run("""
        MERGE (p:Person {name: $name})
        MERGE (friend:Person {name: $friend_name})
        MERGE (p)-[r:KNOWS]->(friend)
        SET r.status = $status, r.since = $date
        """, name=name, status=status, date=date, friend_name=friend_name
    )

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

        # Follow :KNOWS relationships outgoing from Alice three times, return as path
        result = session.run("""
            MATCH path=(:Person {name: $name})-[:KNOWS*3]->(:Person)
            RETURN path
            """, name="Alice"
        )
        record = result.single()
        path = record["path"]

        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.

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 2. 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 and memory 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 should be run again.

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.

Driver

A `Driver` object holds the details required to establish connections with a Neo4j database. Every Neo4j-backed application requires a `Driver` object.

Cypher

Cypher is Neo4j's graph query language that lets you retrieve data from the graph. 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 Manual — 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.

License