Building Neo4j Applications with Python

Learn how to interact with Neo4j from Python using the Neo4j Python Driver

Installing the driver

Introduction

In the **Cypher Fundamentals** course, we covered how to query Neo4j using Cypher.

To run Cypher statements in a Python application, you'll need the **Neo4j Python Driver**. The driver acts as a bridge between your Python code and Neo4j, handling connections to the database and the execution of Cypher queries.

Installing the Driver

To install the driver, use pip:

```shell
pip install neo4j
```

Creating a Driver Instance

First, import the driver and create an instance:

```python
from neo4j import GraphDatabase

driver = GraphDatabase.driver(
    "neo4j://localhost:7687",        # (1)
    auth=("neo4j", "your-password") # (2)
)
```

1. The connection string for your Neo4j database

2. Your Neo4j username and password

Verifying Connectivity

You can verify that the connection details used during driver instantiation are correct by calling the `verifyConnectivity()` function. This function will raise a `Neo4jException` with a `code` property of `Neo.ClientError.Security.Unauthorized` if a connection could not be made.

```python
driver.verify_connectivity()
```

Best Practice

Create just **one** Driver instance for your entire application. This single instance can be safely shared across your code.

Running Your First Query

The `execute_query()` executes a Cypher query and returns the results.

```python
records, summary, keys = driver.execute_query( # (
    "RETURN COUNT {()} AS count"
)

# Get the first record
first = records[0]        # (2)

# Print the count entry
print(first["count"])    # (3)
```

What is happening here?

1. `execute_query()` runs a Cypher query to get the count of all nodes in the database

2. `records` contains a list of the rows returned

3. Keys from the `RETURN` clause are accessed using dictionary-style indexing with square brackets (`[]`)

Full driver lifecycle

Once you have finished with the driver, call `close()` to release any resources held by the driver.

```python
driver.close()
```

For an all-in-one solution, use the `try-with-resources` statement.

```python
with GraphDatabase.driver(NEO4J_URI, auth=(NEO4J_USERNAME, NEO4J_PASSWORD)) as driver:
    result, summary, keys = driver.execute_query("RETURN COUNT {()} AS count")
```

Once the block is exited, the driver is closed automatically.

Executing Cypher statements

Introduction

To run one-off Cypher statements or statements that return a small number of records, use the `execute_query()` method. This method fetches a list of records and loads them into memory.

```python
cypher = """
MATCH (p:Person {name: $name})-[r:ACTED_IN]->(m:Mov
RETURN m.title AS title, r.roles AS roles
"""

name="Tom Hanks"

records, summary, keys = driver.execute_query( # (
    cypher,      # (2)
    name=name    # (3)
)
```

1. The method returns a tuple that can be unpacked to access the query result, summary, and keys.

2. The method expects a Cypher statement as a string as the first argument.

3. Any named parameters not suffixed with an underscore can be accessed in the query by prefixing the name with a dollar sign.

> **Using Parameters**
>
> It is good practice to use parameters in your queries to avoid malicious code being injected into your Cypher statement.

Handling the Result

The `execute_query()` method returns an a tuple containing three objects:

1. A list of `Record` objects

2. A summary of the query execution

3. A list of keys specified in the `RETURN` clause

```python
print(keys) # ['title', 'roles']
print(summary) # A summary of the query execution
```

Accessing results

Each row returned by the query is a `Record` object. The `Record` object is a dictionary-like object that provides access to the data returned by the query.

You can access any item in the `RETURN` clause using square brackets.

```python
# RETURN m.title AS title, r.roles AS roles


for record in result:
    print(record["title"]) # Toy Story
    print(record["roles"]) # ["Woody"]
```

Transforming results

The `execute_query()` method accepts a `result_transformer_` argument that allows you to transform the result into an alternative format.

Rather than returning the tuple, the query will return the output of the `result_transformer_` function.

```python
result = driver.execute_query(
    "UNWIND range(0, 4) AS number RETURN number",
    result_transformer_=lambda result: [
        # double the number for each record in the result
        record['number'] *2 for record in result
    ]
)


print(result) # [0, 2, 4, 6, 8]
```

Working with Dataframes

The `Result` class provides a `to_df()` method that can be used to transform the result into a pandas `DataFrame`.

```python
from neo4j import Result

driver.execute_query(
    cypher,
    result_transformer_=Result.to_df
)
```

Reading and writing

By default, `execute_query()` runs in **WRITE** mode. When in a clustered environment, this sends all queries to the cluster leader. This puts unnecessary load on the leader.

When you're only reading data, you can optimize performance by setting the `routing_` parameter to READ mode. This distributes your read queries across all cluster members.

```python
from neo4j import Result, RoutingControl

driver.execute_query(
    "UNWIND range(0, 4) AS number RETURN number",
    result_transformer_=Result.to_df,
    routing_=RoutingControl.READ # or simply "r"
)
```

You can also pass `r` for read mode and `w` for write mode.

Graph types

Introduction

Let's take a look at the types of data returned by a Cypher query.

The majority of the types returned by a Cypher query are mapped directly to Python types, but some more complex types need special handling.

- Graph types - Nodes, Relationships and Paths
- Temporal types - Dates and times
- Spatial types - Points and distances

Types in Neo4j Browser

When graph types are returned by a query, they are visualized in a graph layout.

Table 1. Direct mapping

Python Type	Neo4j Cypher Type
None	null
bool	Boolean
int	Integer
float	Float
str	String
bytearray	Bytes [1]
list	List
dict	Map

Graph types

The following code snippet finds all movies with the specified title and returns `person`, `acted_in` and `movie`.

```python
result = tx.run("""
MATCH path = (person:Person)-[actedIn:ACTED_IN]->(movie:Movie {title: $title})
RETURN path, person, actedIn, movie
""", title=movie)
```

Nodes

Nodes are returned as a `Node` object.

```python
for record in result:
    node = record["movie"]
```

python: **Working with Node Objects**

```python
print(node.id)              # (1)
print(node.labels)          # (2)
print(node.items())         # (3)

# (4)
print(node["name"])
print(node.get("name", "N/A"))
```

1. The `element_id` property provides access to the node's element ID
   eg. `4:97b72e9c-ae4d-427c-96ff-8858ecf16f88:0`

2. The `labels` property is a frozenset containing an array of labels attributed to the Node
   eg. `['Person', 'Actor']`

3. The `items()` method provides access to the node's properties as an iterable of all name-value pairs.
   eg. `{name: 'Tom Hanks', tmdbId: '31' }`

4. A single property can be retrieved by either using `[]` brackets or using the `get()` method. The `get()` method also allows you to define a default property if none exists.

Relationships

Relationships are returned as a `Relationship` object.

```python
acted_in = record["actedIn"]

print(acted_in.id)          # (1)
print(acted_in.type)        # (2)
print(acted_in.items())     # (3)


# 4
print(acted_in["roles"])
print(acted_in.get("roles", "(Unknown)"))

print(acted_in.start_node) # (5)
print(acted_in.end_node)   # (6)
```

1. `id` - Internal ID of the relationship (eg. `9876`)

2. `type` - Type of relationship (eg. `ACTED_IN`)

3. `items()` - Returns relationship properties as name-value pairs (eg. `{role: 'Woody'}`)

4. Access properties using brackets `[]` or `get()` method

5. `start_node` - ID of the starting node

6. `end_node` - ID of the ending node

Paths

A path is a sequence of nodes and relationships and is returned as a `Path` object.

```python
path = record["path"]

print(path.start_node)    # (1)
print(path.end_node)      # (2)
print(len(path))   # (1)
print(path.relationships)   # (1)
```

1. `start_node` - a Neo4j `Integer` representing the internal ID for the node at the start of the path

2. `end_node` - a Neo4j `Integer` representing the internal ID for the node at the end of the path

3. `len(path)` - A count of the number of relationships within the path

4. `relationships` - An array of `Relationship` objects within the path.

> **Paths are iterable**
>
> Use `iter(path)` to iterate over the relationships in a path.

Dates and times

Temporal types

The `neo4j.time` module provides classes for working with dates and times in Python.

Temporal types in Neo4j are a combination of date, time and timezone elements.

Table 1. Temporal Types

Type	Description	Date?	Time?	Timezone?
`Date`	A tuple of Year, Month and Day	Y		
`Time`	A point in time	Y	Y	
`LocalTime`	A time without a timezone		Y	
`DateTime`	A combination of Date and Time	Y	Y	Y
`LocalDateTime`	A combination of Date and Time without a timezone	Y	Y	

Writing temporal types

```python
from neo4j.time import DateTime
from datetime import timezone, timedelta


driver.execute_query("""
CREATE (e:Event {
    startsAt: $datetime,              // (1)
    createdAt: datetime($dtstring),   // (2)
    updatedAt: datetime()             // (3)
})
""",
    datetime=DateTime(
        2024, 5, 15, 14, 30, 0,
        tzinfo=timezone(timedelta(hours=2))
    ),  # (4)
    dtstring="2024-05-15T14:30:00+02:00"
)
```

When you write temporal types to the database, you can pass the object as a parameter to the query or cast the value within a Cypher statement.

This example demonstrates how to:

1. Use a `DateTime` object as a parameter to the query (`<4>`)

2. Cast an **ISO 8601 format string** within a Cypher statement

3. Get the current date and time using the `datetime()` function.

Reading temporal types

When reading temporal types from the database, you will receive an instance of the corresponding Python type unless you cast the value within your query.

```python
# Query returning temporal types
result = driver.execute_query("""
RETURN date() as date, time() as time, datetime() as datetime, toString(datetime()) as asString
""")

# Access the first record
for record in result:
    # Automatic conversion to Python driver types
    date = record["date"]          # neo4j.time.Date
    time = record["time"]          # neo4j.time.Time
    datetime = record["datetime"]  # neo4j.time.DateTime
    asString = record["asString"]  # str
```

Working with Durations

```python
from neo4j.time import Duration, DateTime

startsAt = DateTime.now()
eventLength = Duration(hours=1, minutes=30)
endsAt = startsAt + eventLength

driver.execute_query("""
CREATE (e:Event {
  startsAt: $startsAt, endsAt: $endsAt,
  duration: $eventLength, // (1)
  interval: duration('P30M') // (2)
})
""",
  startsAt=startsAt, endsAt=endsAt, eventLength=ev
)
```

Durations represent a period of time and can be used for date arithmetic in both Python and Cypher. These types can also be created in Python or cast within a Cypher statement.

1. Pass an instance of `Duration` to the query

2. Use the `duration()` function to create a `Duration` object from an ISO 8601 format string

Calculating durations

You can use the `duration.between` method to calculate the duration between two date or time objects.

Spatial types

Points and locations

Neo4j has built-in support for two-dimensional and three-dimensional spatial data types. These are referred to as **points**. A point may represent geographic coordinates (latitude, longitude) or Cartesian coordinates (x, y).

Depending on the values used to create the point, it can either be a `CartesianPoint` or a `WGS84Point` . If you specify three values, these are considered three dimensional points. Otherwise, they are considered two dimensional points.

In Python, these values are represented by the `neo4j.spatial.CartesianPoint` and `neo4j.spatial.WGS84Point` classes, which are subclasses of the `neo4j.spatial.Point` class.

Cypher Type	Python Type	SRID	3D SRID
Point (Cartesian)	`neo4j.spatial.CartesianPoint`	`7203`	`9157`
Point (WGS-84)	`neo4j.spatial.WGS84Point`	`4326`	`4979`

CartesianPoint

A Cartesian Point defines a point with x and y coordinates. An additional z value can be provided to define a three-dimensional point.

You can create a cartesian point by passing a tuple of values to the `CartesianPoint` constructor or by passing `x`, `y` and `z` values to the point function in Cypher.

```python
# CartesianPoint
twoD = CartesianPoint((x, y))
threeD = CartesianPoint((x, y, z))
```

The driver will convert `point` data types created with an x, y and z value to an instance of the `CartesianPoint` class.

```python
result = driver.execute_query("""
RETURN point({x: 1.23, y: 4.56, z: 7.89}) AS three
""")


point = result.records[0]["threeD"]

# <1> Accessing attributes
print(point.x, point.y, point.z, point.srid) # 1.2

# <2> Destructuring
longitude, latitude, height = point
```

The values can be accessed using the `x`, `y` and `z` attributes `<1>` or by destructuring the point `<2>`.

WGS84Point

A WSG *(World Geodetic System)* point consists of a `latitude` and `longitude` value. An additional `height` value can be provided to define a three-dimensional point and can be created by passing a tuple of values to the `WGS84Point` constructor or by passing `longitude`, `latitude` and `height` values to the point function in Cypher.

```python
# python:    WGS84Point

ldn = WGS84Point((-0.118092, 51.509865))
print(ldn.longitude, ldn.latitude, ldn.srid) # -0.

shard = WGS84Point((-0.086500, 51.504501, 310))
print(shard.longitude, shard.latitude, shard.heigh

# Using destructuring
longitude, latitude, height = shard
```

The driver will return `WGS84Point` objects when `point` data types are created with `latitude` and `longitude` values in Cypher. The values can be accessed using the `longitude`, `latitude` and `height` attributes or by destructuring the point.

```python
# python:    Using point()

result = driver.execute_query("""
RETURN point({
    latitude: 51.5,
    longitude: -0.118,
    height: 100
}) AS point
""")


point = result.records[0]["point"]
longitude, latitude, height = point
```

Distance

The `point.distance` function can be used to calculate the distance between two points with the same SRID. The result is a `float` representing the distance in a straight line between the two points.

> **SRIDs must be compatible**
>
> If the SRID values are different, the function will return `None`.

```python
from neo4j.spatial import CartesianPoint

# Create two points
point1 = CartesianPoint((1, 1))
point2 = CartesianPoint((10, 10))

# Query the distance using Cypher
result = driver.execute_query("""
RETURN point.distance($p1, $p2) AS distance
""", p1=point1, p2=point2)

# Print the distance from the result
distance = result.records[0]["distance"]
print(distance)  # 12.727922061357855
```

Transaction management

Introduction

In the previous module, you learned how to execute one-off Cypher statements using the `execute_query()` method.

The drawback of this method is that the entire record set is only available once the final result is returned. For longer running queries or larger datasets, this can consume a lot of memory and a long wait for the final result.

In a production application, you may also need finer control of database transactions or to run multiple related queries as part of a single transaction.

Transaction functions allow you to run multiple queries in a single transaction while accessing results immediately.

> ## Understanding Transactions
>
> Neo4j is an ACID-compliant transactional database, which means queries are executed as part of a single atomic transaction. This ensures your data operations are consistent and reliable.

Sessions

To execute transactions, you need to open a session. The session object manages the underlying database connections and provides methods for executing transactions.

```python
with driver.session() as session:
    # Call transaction functions here
```

Consuming a session within a try-with-resources statement will automatically close the session and release any underlying connections when the block is exited.

Specifying a database

In a multi-database instance, you can specify the database to use when creating a session using the `database` parameter.

Transaction functions

The session object provides two methods for managing transactions:

- `Session.execute_read()`
- `Session.execute_write()`

If the entire function runs successfully, the transaction is committed automatically. If any errors occur, the entire transaction is rolled back.

> **Transient errors**
>
> These functions will also retry if the transaction fails due to a transient error, for example, a network issue.

Unit of work patterns

A unit of work is a pattern that groups related operations into a single transaction.

```python
def create_person(tx, name, age): # (1)
  result = tx.run("""
  CREATE (p:Person {name: $name, age: $age})
  RETURN p
  """, name=name, age=age) # (2)
```

1. The first argument to the transaction function is always a `Transaction` object. Any additional keyword arguments are passed to the transaction function.

2. The `run()` method on the `Transaction` object is called to execute a Cypher statement.

Multiple Queries in One Transaction

You can execute multiple queries within the same transaction function to ensure that all operations are completed or fail as a single unit.

```python
def transfer_funds(tx, from_account, to_account, amount):
    # Deduct from first account
    tx.run("MATCH (a:Account {id: $from}) SET a.balance = a.balance - $amount", from=from_account, amount=am

    # Add to second account
    tx.run("MATCH (a:Account {id: $to}) SET a.balance = a.balance + $amount", to=to_account, amount=amount)
```

> Transaction state is maintained in memory, so be mindful of running too many operations in a single transaction. Break up very large operations into smaller transactions when possible.

Handling outputs

The `execute_read()` and `execute_write()` methods
return a `Result` object.

The records contained within the result will be iterated over
as soon as they are available.

The result must be consumed within the transaction
function.

The `consume()` method discards any remaining records
and returns a `Summary` object that can be used to access
metadata about the transaction.

```python
with driver.session() as session:
    def get_answer(tx, answer):
        result = tx.run("RETURN $answer AS answer",
            answer=answer)

        for record in result: # (1)
            print(record["answer"])

        return result.consume()

    # Call the transaction function
    summary = session.execute_read(get_answer, answer

    # Output the summary
    print("Results available after", summary.result_
        "ms and consumed after", summary.result_consume
```

Handling Database Errors

Error Handling

When working with Neo4j, you may encounter various database errors that need to be handled gracefully in your application. The driver exports a `Neo4jError` class that is inherited by all exceptions thrown by the database. Exceptions related to the driver and its connection are subclasses of `DriverError`.

Common exceptions

- `CypherSyntaxError` -Raised when the Cypher syntax is invalid
- `ConstraintError` -Raised when a constraint unique or other is violated
- `AuthError` -Raised when authentication fails
- `TransientError` -Raised when the database is not accessible

Handling errors

Any errors raised by the driver will have `code` and `message` properties that describe the error.

```python
from neo4j.exceptions import Neo4jError


try:
    # Run a Cypher statement
except Neo4jError as e:
    print(e.code)
    print(e.message)
    print(e.gql_status)
```

The `gql_status` property contains an error code that corresponds to an error in the ISO GQL standard.

A full list of error codes can be found in **Status Codes for Errors & Notifications**.

Example: Handling unique constraint violations

One common scenario is dealing with constraint violations when inserting data. A unique constraint ensures that a property value is unique across all nodes with a specific label.

The following Cypher statement creates a unique constraint named `unique_email` to ensure that the `email` property is unique for the `User` label:

```cypher
CREATE CONSTRAINT unique_email IF NOT EXISTS
FOR (u:User) REQUIRE u.email IS UNIQUE
```

If a Cypher statement violates this constraint, Neo4j will raise a `ConstraintError`.

```python
from neo4j.exceptions import ConstraintError

def create_user(tx, name, email):
    try:
        result = tx.run("""
            CREATE (u:User {name: $name, email: $e
            RETURN u
        """, name=name, email=email)

    except ConstraintError as e:
        print(e.code)
        # Neo.ClientError.Schema.ConstraintValidat
        print(e.message)
        # The value [email] for property [email] v
        print(e.gql_status) # 22N41
```