



The Neo4j Python Driver Manual v4.4

Table of Contents

Get started	2
About the official Python driver	2
Driver versions and installation	3
A "Hello World" example	4
Driver API docs	6
Client applications	7
The Driver Object	7
Connection URIs	7
Authentication	15
Configuration	17
Logging	20
Cypher workflow	21
Overview	21
Sessions	21
Transactions	22
Queries and results	22
Causal chaining and bookmarks	23
Routing transactions using access modes	24
Databases and execution context	25
Type mapping	27
Exceptions and error handling	33
Exceptions and error handling	34
The session API	36
Simple sessions	36
Session configuration	39
Appendix A: Driver terminology	41

Copyright © 2021 Neo4j, Inc.

License: [Creative Commons 4.0](#)

This is the manual for Neo4j Python Driver version 4.4, authored by the Neo4j Team.

This manual covers the following areas:

- [Get started](#) — An overview of the official Neo4j Python Driver and how to connect to a Neo4j database.
- [Client applications](#) — How to manage database connections within an application.
- [Cypher workflow](#) — How to create units of work and provide a logical context for that work.
- [The session API](#) — How the types and values used by Cypher map to native language types.
- [Driver terminology](#) — Terminology for drivers.

Who should read this?

This manual is written for Python developers building a Neo4j client application.

Get started

This section gives an overview of the official Neo4j Python Driver and how to connect to a Neo4j database with a "Hello World" example.

About the official Python driver

Neo4j provides official drivers for a number of popular programming languages. These drivers are supported by Neo4j.

Community drivers also exist for many languages, but vary greatly in terms of feature sets, maturity, and support. To find more about community drivers, visit <https://neo4j.com/developer/language-guides/>.

The following languages and frameworks are officially supported by Neo4j:

Table 1. Supported languages and frameworks for the 4.x driver series

Language/framework	Versions supported
.NET	.NET Standard 2.0
Go	Go 1.10
Java	Java 8+ (latest patch releases).
JavaScript	All LTS versions of Node.JS, specifically the 4.x and 6.x series runtimes (https://github.com/nodejs/LTS).
Python	Python 3.5 and above.

The driver API is intended to be topologically agnostic. This means that the underlying database topology — single instance, Causal Cluster, etc. — can be altered without requiring a corresponding alteration to application code.

In the general case, only the [connection URI](#) needs to be modified when changes are made to the topology.



The official drivers do not support HTTP communication. If you need an HTTP driver, choose one of the community drivers.

See also the [HTTP API documentation](#).

Driver versions and installation

Starting with Neo4j 4.0, the *versioning scheme for the database, driver and protocol have all been aligned*. This simplifies general compatibility concerns.

Cross-version compatibility is still available, and minimum support for current and previous versions between both server and driver is guaranteed. More specifically, this means that Neo4j 4.0 is guaranteed to be compatible with both 4.0 Drivers and 1.7 Drivers, and the 4.0 Drivers are guaranteed to be compatible with both Neo4j 4.0 and Neo4j 3.5. In cases where at least one peer is below version 4.0, communication will occur in *fallback mode, limiting functionality to that available in the lowest-versioned component*.



Drivers 1.7 do not support multiple databases and Neo4j Fabric, features introduced in Neo4j 4.0. To be able to run multiple databases online concurrently and to do distributed queries over them, you must upgrade Drivers from 1.7 to 4.0. For information, see [4.0 Migration Guide → Chapter 6. Upgrade Neo4j drivers](#).

Wherever possible, it is recommended to use the latest stable driver release available. This will provide the greatest degree of stability and will ensure that the full set of server functionality is available. The drivers, when used with Neo4j Enterprise Edition, come with full cluster routing support. The drivers make no explicit distinction between Enterprise Edition and Community Edition however, and simply operate with the functionality made available by Neo4j itself.

Example 3. Acquire the driver

To find the latest stable version of the Python Driver, visit <https://pypi.org/project/neo4j/>

To find a list of all available releases, visit <https://pypi.org/simple/neo4j/>

To install the latest stable version of the Python Driver:

```
pip install neo4j
```

It is also an option to install a certain version of the driver.

Example 1. Installation with Python

The following is the syntax for installing a certain version of the Python Driver:

```
pip install neo4j==$PYTHON_DRIVER_VERSION
```

In the following example we are installing Python Driver version 4.4.

```
pip install neo4j==4.4
```

Example 2. Installation with Python, get Python Driver prerelease

In the following example we get the latest prerelease version:

```
pip install neo4 --pre
```

The release notes for the Python Driver are available [here](#)

A "Hello World" example

The example below shows the minimal configuration necessary to interact with **Neo4j** through the Python driver:

Example 4. Hello World

```
import logging
import sys

from neo4j import GraphDatabase
from neo4j.exceptions import ServiceUnavailable


class App:

    def __init__(self, uri, user, password):
        self.driver = GraphDatabase.driver(uri, auth=(user, password))

    def close(self):
        # Don't forget to close the driver connection when you are finished with it
        self.driver.close()

    @staticmethod
    def enable_log(level, output_stream):
        handler = logging.StreamHandler(output_stream)
        handler.setLevel(level)
        logging.getLogger("neo4j").addHandler(handler)
        logging.getLogger("neo4j").setLevel(level)

    def create_friendship(self, person1_name, person2_name, knows_from):
        with self.driver.session() as session:
            # Write transactions allow the driver to handle retries and transient errors
            result = session.write_transaction(
                self._create_and_return_friendship, person1_name, person2_name, knows_from)
            for row in result:
                print("Created friendship between: {p1}, {p2} from {knows_from}"
                      .format(
                        p1=row['p1'],
                        p2=row['p2'],
                        knows_from=row["knows_from"]))

    @staticmethod
    def _create_and_return_friendship(tx, person1_name, person2_name, knows_from):
        # To learn more about the Cypher syntax, see https://neo4j.com/docs/cypher-manual/current/
        # The Reference Card is also a good resource for keywords https://neo4j.com/docs/cypher-
        refcard/current/
        query = (
            "CREATE (p1:Person { name: $person1_name }) "
            "CREATE (p2:Person { name: $person2_name }) "
            "CREATE (p1)-[k:KNOWS { from: $knows_from }]->(p2) "
            "RETURN p1, p2, k"
        )
        result = tx.run(query, person1_name=person1_name,
                        person2_name=person2_name, knows_from=knows_from)
        try:
            return [{
                "p1": row["p1"]["name"],
                "p2": row["p2"]["name"],
                "knows_from": row["k"]["from"]
            }
                for row in result]
        except ServiceUnavailable as exception:
            logging.error("{query} raised an error: \n {exception}".format(
                query=query, exception=exception))
            raise

    def find_person(self, person_name):
        with self.driver.session() as session:
            result = session.read_transaction(self._find_and_return_person, person_name)
            for row in result:
                print("Found person: {row}".format(row=row))

    @staticmethod
    def _find_and_return_person(tx, person_name):
        query = (
            "MATCH (p:Person) "
            "WHERE p.name = $person_name "
```

```

        "RETURN p.name AS name"
    )
    result = tx.run(query, person_name=person_name)
    return [row["name"] for row in result]

if __name__ == "__main__":
    bolt_url = "%%BOLT_URL_PLACEHOLDER%"
    user = "<Username for database>"
    password = "<Password for database>"
    App.enable_log(logging.INFO, sys.stdout)
    app = App(bolt_url, user, password)
    app.create_friendship("Alice", "David", "School")
    app.find_person("Alice")
    app.close()

```

Driver API docs

For a comprehensive listing of all driver functionality, please see the [Python API reference documentation](#).

Client applications

This section describes how to manage database connections within an application.

The Driver Object

Neo4j client applications require a **Driver Object** which, from a data access perspective, forms the backbone of the application. It is through this object that all Neo4j interaction is carried out, and it should therefore be made available to all parts of the application that require data access.

In languages where [thread safety](#) is an issue, the Driver Object can be considered **thread-safe**.



A note on lifecycle

Applications will typically construct a Driver Object on startup and destroy it on exit.

Destroying a Driver Object will immediately shut down any connections previously opened via that Driver Object, by closing the associated connection pool.

This will have the consequence of rolling back any open transactions, and closing any unconsumed results.

To construct a driver instance, a [connection URI](#) and [authentication information](#) must be supplied.

Additional configuration details can be supplied if required. The configuration details are immutable for the lifetime of the Driver Object. Therefore, if multiple configurations are required (such as when working with multiple database users) then multiple Driver Objects must be used.

Example 5. The driver lifecycle

```
from neo4j import GraphDatabase
```

```
class DriverLifecycleExample:
    def __init__(self, uri, auth):
        self.driver = GraphDatabase.driver(uri, auth=auth)

    def close(self):
        self.driver.close()
```

Connection URIs

A connection URI identifies a graph database and how to connect to it.

The **encryption** and **trust** settings provide detail to how that connection should be secured.



There are significant changes to security settings between Neo4j 3.x and Neo4j 4.x

Please consider the information in this section before upgrading from a previous version.

The [Migration Guide](#) is also a good source of information about changes in functionality.

Starting with Neo4j 4.0, client-server communication uses only *unencrypted local connections by default*.

This is a change from previous versions, which switched on encryption by default, but generated a self-signed certificate out of the box.

When a full certificate is installed, and encryption is enabled on the driver, full certificate checks are carried out (refer to [Operations Manual → SSL framework](#)). Full certificates provide better overall security than self-signed certificates as they include a complete chain of trust back to a root certificate authority.



[Neo4j Aura](#) is a *secure hosted service* backed by full certificates signed by a root certificate authority.

To connect to [Neo4j Aura](#), driver users must *enable encryption* and the complete set of certificate checks (the latter of which are enabled by default).

For more information, see [Examples](#) below.

Table 2. Changes in default security settings between 3.x and 4.x

Setting	Neo4j 4.x	Neo4j 3.x (Drivers 1.x)
Bundled certificate	none	auto-generated, self-signed
Driver encryption	off	on
Bolt interface	localhost	localhost
Certificate expiry check	on	on
Certificate CA check	on	off
Certificate hostname check	on	off

Initial address resolution

The address provided in a `neo4j://` URI is used for initial and fallback communication only.

This communication occurs *to bootstrap the routing table*, through which all subsequent communication is carried out. Fallback occurs when the driver is unable to contact any of the addresses held in the routing table. The initial address is once again reused to bootstrap the system.

Several options are available for providing this initial logical-to-physical host resolution. These include *regular DNS*, *custom middleware* such as a load balancer, and the Driver Object *resolver function*, all of which are described in the following sections.

DNS resolution

DNS resolution is the default, and always-available option. As it is possible to configure DNS to resolve a single host name down to multiple IP addresses, this can be used to expose all core server IP addresses under a single host name.

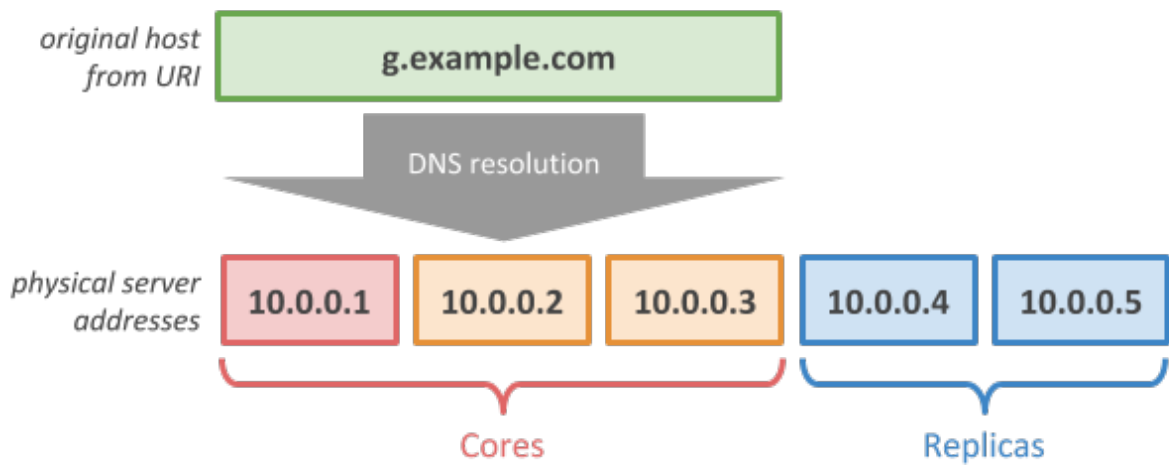


Figure 1. Initial address resolution over DNS

Custom middleware

Middleware, such as a load balancer, can be used to group the core servers under a single public address.

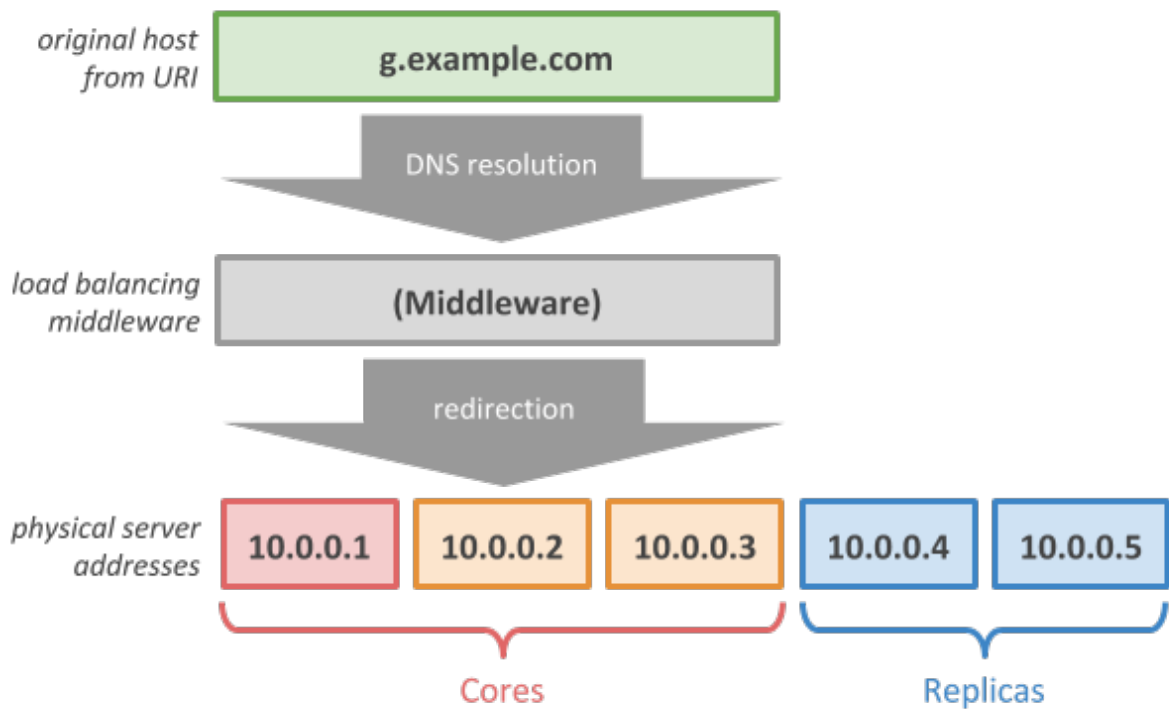


Figure 2. Initial address resolution using custom middleware

Resolver function

Neo4j Drivers also present an address resolution intercept hook called the *resolver function*.

This takes the form of a callback function that accepts a single input address and returns multiple output addresses. The function may hard code the output addresses or may draw them from another configuration source, as required.

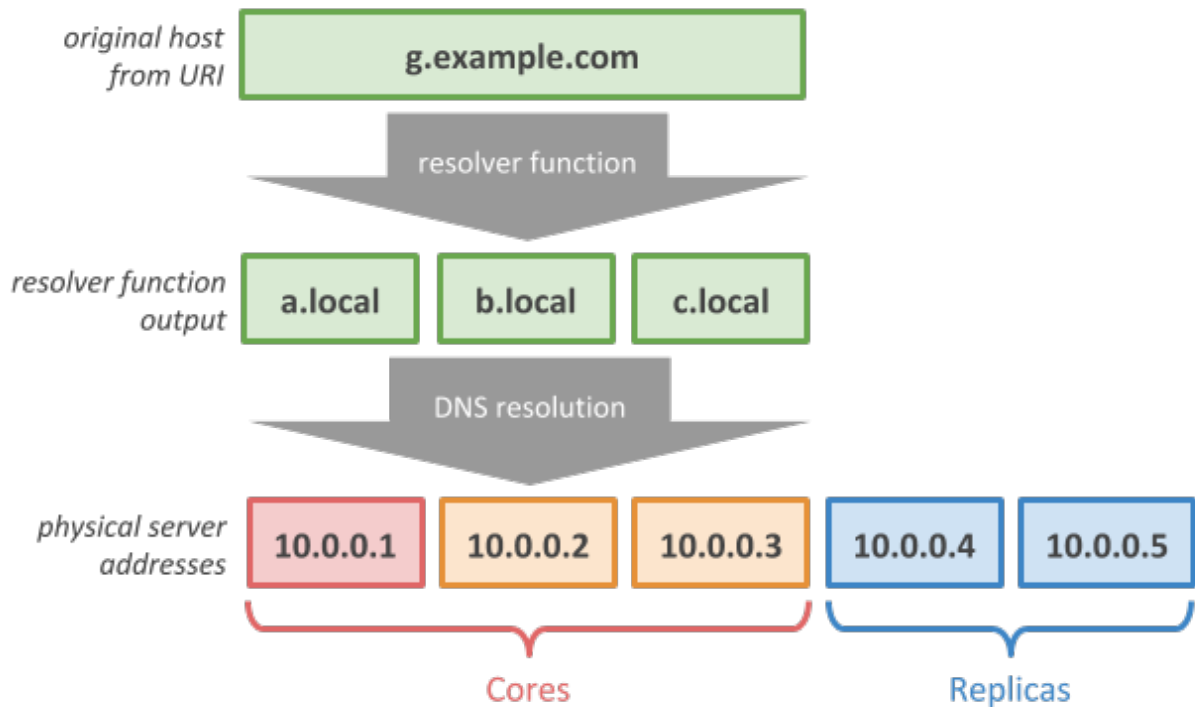


Figure 3. Initial address resolution using resolver function

The example below shows how to expand a single address into multiple (hard-coded) output addresses:

Example 6. Custom Address Resolver

```
from neo4j import (
    GraphDatabase,
    WRITE_ACCESS,
)

def create_driver(uri, user, password):

    def resolver(address):
        host, port = address
        if host == "x.example.com":
            yield "a.example.com", port
            yield "b.example.com", port
            yield "c.example.com", port
        else:
            yield host, port

    return GraphDatabase.driver(uri, auth=(user, password), resolver=resolver)

def add_person(name):
    driver = create_driver("neo4j://x.example.com", user="neo4j", password="password")
    session = driver.session(default_access_mode=WRITE_ACCESS)
    session.run("CREATE (a:Person {name: $name})", {"name": name})
    session.close()
    driver.close()
```

Routing table

The routing table acts like the glue between the driver connectivity layer and the database surface. This table contains a list of server addresses, grouped as **readers** and **writers**, and is refreshed automatically by the driver as required.

The driver does not expose any API to work directly with the routing table, but it can sometimes be useful to explore when troubleshooting a system.

Routing context

A routing context can be included as the query part of a `neo4j://` URI.

Routing contexts are defined by means of **server policies** and allow customization of the contents of the routing table.

Example 7. Configure a routing driver with routing context

This example will assume that Neo4j has been configured for server policies as described in [Neo4j Operations Manual → Load balancing for multi-data center systems](#). In particular, a server policy called **europe** has been defined. Additionally, we have a server `neo01.graph.example.com` to which we wish to direct the driver.

This URI will use the server policy **europe**:

```
neo4j://neo01.graph.example.com?policy=europe
```



Server-side configuration to enable routing drivers with routing context

A prerequisite for using a routing driver with routing context is that the Neo4j database is operated on a [Causal Cluster](#) with the [Multi-data center licensing option](#) enabled. Additionally, the routing contexts must be defined within the cluster as *routing policies*.

For details on how to configure *multi-data center routing policies* for a Causal Cluster, please refer to [Operations Manual → Causal Clustering](#).



Exposing a single instance deployment on a remote host using the neo4j URI scheme

If you are using a *single instance* of Neo4j, deployed on a remote machine whilst using the neo4j URI scheme, **you will need to complete additional configuration of the server.**

To make the server aware of its deployment environment, you need to configure `default_advertised_address` with your deployment machine's host name, as visible from the client machine.

Examples

Connection URIs are typically formed according to the following pattern:

```
neo4j://<HOST>:<PORT>[?<ROUTING_CONTEXT>]
```

This targets a routed Neo4j service that may be fulfilled by either a cluster or a single instance. The `HOST` and `PORT` values contain a logical hostname and port number targeting the entry point to the Neo4j service (e.g. `neo4j://graph.example.com:7687`).

In a clustered environment, the URI address will resolve to one of more of the core members; for standalone installations, this will simply point to that server address. The `ROUTING_CONTEXT` option allows for customization of the routing table and is discussed in more detail in [Routing context](#).

An alternative URI form, using the `bolt` URI scheme (e.g. `bolt://graph.example.com:7687`), can be used when a *single point-to-point connection is required*. This variant is useful for the subset client applications (such as admin tooling) that need to be aware of individual servers, as opposed to those which require a highly available database service.

```
bolt://<HOST>:<PORT>
```

Each of the `neo4j` and `bolt` URI schemes permit variants that contain extra encryption and trust information. The `+s` variants enable encryption with a full certificate check, and the `+ssc` variants enable encryption, but with no certificate check. This latter variant is designed specifically for use with self-signed certificates.

Table 3. Available URI schemes

URI scheme	Routing	Description
<code>neo4j</code>	Yes	Unsecured

URI scheme	Routing	Description
neo4j+s	Yes	Secured with full certificate
neo4j+ssc	Yes	Secured with self-signed certificate
bolt	No	Unsecured
bolt+s	No	Secured with full certificate
bolt+ssc	No	Secured with self-signed certificate



Neo4j 3.x did not provide a routing table in single instance mode and therefore you should use a **bolt://** URI if targeting an older, non-clustered server.

The table below provides example code snippets for different deployment configurations. Each snippet expects an **auth** variable to have been previously defined, containing the authentication details for that connection.

Connecting to a service

The tables below illustrate examples of how to connect to a service with routing:

Table 4. Neo4j Aura, secured with full certificate

Product	Neo4j Aura
Security	Secured with full certificate
Code snippet	<pre>GraphDatabase.driver("neo4j+s://graph.example.com:7687", auth)</pre> <p>For Python Driver versions before 4.0.0 you need this snippet instead:</p> <pre>GraphDatabase.driver("neo4j://graph.example.com:7687", auth, "encrypted"=True, "trust"=TRUST_SYSTEM_CA_SIGNED_CERTIFICATES)</pre>
Comments	This is the default (and only option) for Neo4j Aura.

Table 5. Neo4j 4.x, unsecured

Product	Neo4j 4.x
---------	-----------

Security	Unsecured
Code snippet	<pre>GraphDatabase.driver("neo4j://graph.example.com:7687", auth)</pre>
Comments	This is the default for Neo4j 4.x series

Table 6. Neo4j 4.x, secured with full certificate

Product	Neo4j 4.x
Security	Secured with full certificate
Code snippet	<pre>GraphDatabase.driver("neo4j+s://graph.example.com:7687", auth)</pre> <p>For Python Driver versions before 4.0.0 you need this snippet instead:</p> <pre>GraphDatabase.driver("neo4j://graph.example.com:7687", auth, "encrypted"=True, "trust"=TRUST_SYSTEM_CA_SIGNED_CERTIFICATES)</pre>

Table 7. Neo4j 4.x, secured with self-signed certificate

Product	Neo4j 4.x
Security	Secured with self-signed certificate
Code snippet	<pre>neo4j.driver("neo4j+ssc://graph.example.com:7687", auth)</pre> <p>For Python Driver versions before 4.0.0 you need this snippet instead:</p> <pre>GraphDatabase.driver("neo4j://graph.example.com:7687", auth, "encrypted"=True, "trust"=TRUST_ALL_CERTIFICATES)</pre>

Table 8. Neo4j 3.x, secured with full certificate

Product	Neo4j 3.x
Security	Secured with full certificate

Code snippet	<pre>neo4j.driver("neo4j+s://graph.example.com:7687", auth)</pre> <p>For Python Driver versions before 4.0.0 you need this snippet instead:</p> <pre>GraphDatabase.driver("neo4j://graph.example.com:7687", auth, "encrypted"=True, "trust"=TRUST_SYSTEM_CA_SIGNED_CERTIFICATES)</pre>
--------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Table 9. Neo4j 3.x, secured with self-signed certificate

Product	Neo4j 3.x
Security	Secured with self-signed certificate
Code snippet	<pre>neo4j.driver("neo4j+ssc://graph.example.com:7687", auth)</pre> <p>For Python Driver versions before 4.0.0 you need this snippet instead:</p> <pre>GraphDatabase.driver("neo4j://graph.example.com:7687", auth, "encrypted"=True, "trust"=TRUST_ALL_CERTIFICATES)</pre>
Comments	This is the default for Neo4j 3.x series

Table 10. Neo4j 3.x, unsecured

Product	Neo4j 3.x
Security	Unsecured
Code snippet	<pre>neo4j.driver("neo4j://graph.example.com:7687", auth)</pre>



To connect to a service without routing, you can replace **neo4j** with **bolt**.

Authentication

Authentication details are provided as an auth token which contains the user names, passwords or other credentials required to access the database. Neo4j supports multiple authentication standards but uses *basic authentication* by default.

Basic authentication

The basic authentication scheme is backed by a password file stored within the server and requires

applications to provide a user name and password. For this, use the basic auth helper:

Example 8. Basic authentication

```
from neo4j import GraphDatabase
```

```
def __init__(self, uri, user, password):  
    self.driver = GraphDatabase.driver(uri, auth=(user, password))
```



The basic authentication scheme can also be used to authenticate against an LDAP server.

Kerberos authentication

The Kerberos authentication scheme provides a simple way to create a Kerberos authentication token with a base64 encoded server authentication ticket. The best way to create a Kerberos authentication token is shown below:

Example 9. Kerberos authentication

```
from neo4j import (  
    GraphDatabase,  
    kerberos_auth,  
)
```

```
def __init__(self, uri, ticket):  
    self._driver = GraphDatabase.driver(uri, auth=kerberos_auth(ticket))
```



The Kerberos authentication token can only be understood by the server if the server has the [Kerberos Add-on](#) installed.

Custom authentication

For advanced deployments, where a custom security provider has been built, the custom authentication helper can be used.

Example 10. Custom authentication

```
from neo4j import (
    GraphDatabase,
    custom_auth,
)
```

```
def __init__(self, uri, principal, credentials, realm, scheme, **parameters):
    auth = custom_auth(principal, credentials, realm, scheme, **parameters)
    self.driver = GraphDatabase.driver(uri, auth=auth)
```

Configuration

ConnectionAcquisitionTimeout

The maximum amount of time in seconds^[1] a session will wait when requesting a connection from the connection pool. For connection pools where all connections are currently being used and the `MaxConnectionPoolSize` limit has been reached, a session will wait this duration for a connection to be made available. Since the process of acquiring a connection may involve creating a new connection, **ensure that the value of this configuration is higher than the configured `ConnectionTimeout`.**

Setting a low value will allow for transactions to **fail fast** when all connections in the pool have been acquired by other transactions. Setting a higher value will result in these transactions being queued, increasing the chances of eventually acquiring a connection at the cost of longer time to receive feedback on failure. Finding an optimal value may require an element of experimentation, taking into consideration the expected levels of parallelism within your application as well as the `MaxConnectionPoolSize`.

Default: 60.0

Type: float

Example 11. Configure connection pool

```
from neo4j import GraphDatabase
```

```
def __init__(self, uri, auth):
    self.driver = GraphDatabase.driver(uri, auth=auth,
                                       max_connection_lifetime=30 * 60,
                                       max_connection_pool_size=50,
                                       connection_acquisition_timeout=2 * 60)
```

ConnectionTimeout

The maximum amount of time in seconds^[2] to wait for a TCP connection to be established. Connections are only created when a session requires one unless there is an available connection in the connection pool. The driver maintains a pool of open connections which can be loaned to a session when one is available. If a connection is not available, then an attempt to create a new connection (provided the `MaxConnectionPoolSize` limit has not been reached) is made with this configuration option, providing

the maximum amount of time to wait for the connection to be established.

In environments with high latency and high occurrences of connection timeouts it is recommended to configure a higher value. For lower latency environments and quicker feedback on potential network issues configure with a lower value.

Default: 30.0

Type: float

Example 12. Configure connection timeout

```
from neo4j import GraphDatabase

def __init__(self, uri, auth):
    self.driver = GraphDatabase.driver(uri, auth=auth, connection_timeout=15)
```

CustomResolver

Specify a custom server address resolver used by the routing driver to resolve the initial address used to create the driver. See [Resolver function](#) for more details.

Encryption

Specify whether to use an encrypted connection between the driver and server.

Default: None

Example 13. Unencrypted configuration

```
from neo4j import GraphDatabase

def __init__(self, uri, auth):
    self.driver = GraphDatabase.driver(uri, auth=auth, encrypted=False)
```

MaxConnectionLifetime

The maximum duration in seconds^[3] the driver will keep a connection for before being removed from the pool. Note that while the driver will respect this value, it is possible that the network environment will close connections inside this lifetime. This is beyond the control of the driver. The check on the connection's lifetime happens when a session requires a connection. If the available connection's lifetime is over this limit it is closed and a new connection is created, added to the pool and returned to the requesting session. Changing this configuration value would be useful in environments where users don't have full control over the network environment and wish to proactively ensure all connections are ready.

Setting this option to a low value will cause a high connection churn rate, and can result in a performance drop. It is recommended to pick a value smaller than the maximum lifetime exposed by the surrounding system infrastructure (such as operating system, router, load balancer, proxy and firewall).

Negative values result in lifetime not being checked.

Default: 3600

Type: float

MaxConnectionPoolSize

The maximum total number of connections allowed, per host (i.e. cluster nodes), to be managed by the connection pool. In other words, for a direct driver using the `bolt://` scheme, this sets the maximum number of connections towards a single database server. For a driver connected to a cluster using the `neo4j://` scheme, this sets the maximum amount of connections per cluster member. If a session or transaction tries to acquire a connection at a time when the pool size is at its full capacity, it must wait until a free connection is available in the pool or the request to acquire a new connection times out. The connection acquiring timeout is configured via `ConnectionAcquisitionTimeout`.

This configuration option allows you to manage the memory and I/O resources being used by the driver and tuning this option is dependent on these factors, in addition to number of cluster members.

Default: 100 connections

MaxTransactionRetryTime

The maximum amount of time in seconds^[4] that a managed transaction will retry for before failing. Queries that are executed within a managed transaction gain the benefit of being retried when a transient error occurs. When this happens the transaction is retired multiple times up to the `MaxTransactionRetryTime`.

Configure this option higher in high latency environments or if you are executing many large transactions which could limit the number of times that they are retired and therefore their chance to succeed. Configure lower in low latency environments and where your workload mainly consists of many smaller transactions. Failing transactions faster may highlight the reasons behind the transient errors making it easier to fix underlying issues.

Default: 30.0

Type: float

Example 14. Configure maximum retry time

```
from neo4j import GraphDatabase
```

```
def __init__(self, uri, auth):  
    self.driver = GraphDatabase.driver(uri, auth=auth, max_transaction_retry_time=15)
```

TrustStrategy

Specify how to determine the authenticity of encryption certificates provided by the Neo4j instance that you are connecting to. There are three choices as to which strategy to use:

- `TRUST_SYSTEM_CA_SIGNED_CERTIFICATES` - Accept any certificate that can be verified against the

system store.

- **TRUST_CUSTOM_CA_SIGNED_CERTIFICATES** - Accept any certificate that can be verified against a custom CA.
- **TRUST_ALL_CERTIFICATES** - Accept any certificate provided by the server, regardless of CA chain. We do not recommend using this setting for production environments.

Default: **TRUST_SYSTEM_CA_SIGNED_CERTIFICATES** (Note - only when encryption is enabled)

Example 15. Configure trusted certificates

```
from neo4j import (
    GraphDatabase,
    TRUST_SYSTEM_CA_SIGNED_CERTIFICATES,
    TRUST_ALL_CERTIFICATES,
)

def __init__(self, uri, auth):
    self.driver = GraphDatabase.driver(uri, auth=auth, encrypted=True, trust
    =TRUST_ALL_CERTIFICATES)
```

Logging

All official Neo4j Drivers log information to standard logging channels. This can typically be accessed in an ecosystem-specific way.

The code snippet below demonstrates how to redirect log messages to standard output:

```
import logging
import sys

handler = logging.StreamHandler(sys.stdout)
handler.setLevel(logging.DEBUG)
logging.getLogger("neo4j").addHandler(handler)
logging.getLogger("neo4j").setLevel(logging.DEBUG)
```

[1] Other drivers may use milliseconds instead of seconds for this configuration.

[2] Other drivers may use milliseconds instead of seconds for this configuration.

[3] Other drivers may use milliseconds instead of seconds for this configuration.

[4] Other drivers may use milliseconds instead of seconds for this configuration.

Cypher workflow

This section describes how to create units of work and provide a logical context for that work.

Overview

The Neo4j Drivers expose a Cypher Channel over which database work can be carried out (see the [Cypher Manual](#) for more information on the Cypher Query Language).

Work itself is organized into **sessions**, **transactions** and **queries**, as follows:

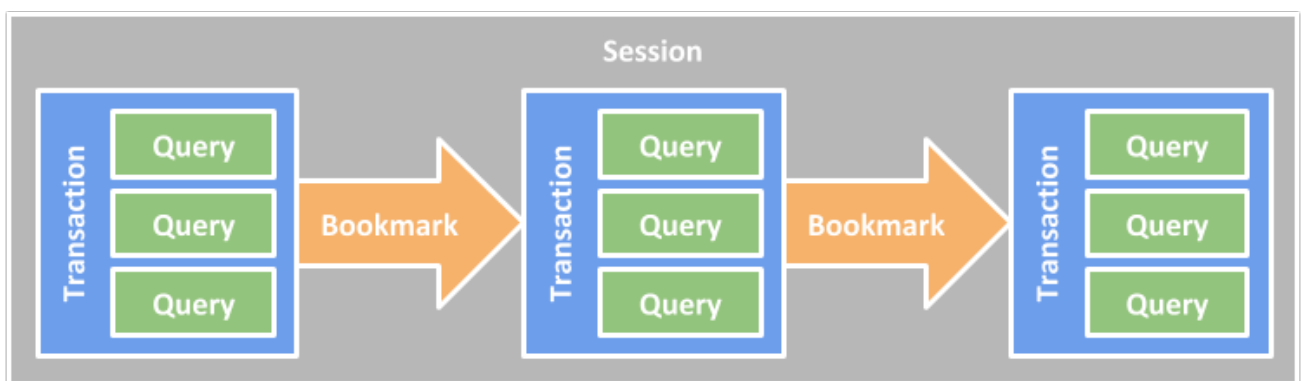


Figure 4. Sessions, queries and transactions

Sessions are always bound to a **single transaction context**, which is typically an individual database.

Using the bookmark mechanism, sessions also provide a guarantee of correct transaction sequencing, even when transactions occur over multiple cluster members. This effect is known as **causal chaining**.

Sessions

Sessions are **lightweight containers for causally chained sequences of transactions** (see [Operations Manual → Causal consistency](#)). They essentially provide context for storing transaction sequencing information in the form of bookmarks.

When a transaction begins, the session in which it is contained acquires a connection from the driver connection pool. On commit (or rollback) of the transaction, the session releases that connection again. This means that it is only when a session is carrying out work that it occupies a connection resource. When idle, no such resource is in use.

Due to the sequencing guaranteed by a session, sessions may only host **one transaction at a time**. For parallel execution, multiple sessions should be used. In languages where **thread safety** is an issue, **sessions should not be considered thread-safe**.

Closing a session forces any open transaction to be rolled back, and its associated connection to consequently be released back into the pool.

Sessions are bound to a single transactional context, specified on construction. Neo4j exposes each database inside its own context, thereby prohibiting cross-database transactions (or sessions) by design. Similarly, *sessions bound to different databases may not be causally chained by propagating bookmarks between them.*

Individual language drivers provide several session classes, each oriented around a particular programming style. Each session class provides a similar set of functionality but offers client applications a choice based on how the application is structured and what frameworks are in use, if any.

The session classes are described in [The session API](#). For more details, please see the [API documentation](#).

Transactions

Transactions are **atomic units of work** containing one or more **Cypher Queries**. Transactions may contain read or write work, and will generally be routed to an appropriate server for execution, where they will be carried out in their entirety. In case of a transaction failure, the transaction needs to be retried from the beginning. This is the responsibility of the [transaction manager](#).

The Neo4j Drivers provide **transaction management** via the [transaction function](#) mechanism. This mechanism is exposed through methods on the Session object which accept a function object that can be played multiple times against different servers until it either succeeds or a timeout is reached. This approach is recommended for most client applications.

A convenient **short-form alternative** is the [auto-commit](#) transaction mechanism. This provides a limited form of transaction management for single-query transactions, as a trade-off for a slightly smaller code overhead. This form of transaction is useful for **quick scripts and environments where high availability guarantees are not required**. It is also the required form of transaction for running **PERIODIC COMMIT** queries, which are the only type of Cypher Query to manage their own transactions.

A lower-level **unmanaged transaction API** is also available for advanced use cases. This is useful when an alternative transaction management layer is applied by the client, in which error handling and retries need to be managed in a custom way.

To learn more about how to use unmanaged transactions, see [API documentation](#) for the relevant language.

Queries and results

Queries consist of a request to the server to execute a Cypher statement, followed by a response back to the client with the result. Results are transmitted as a **stream of records**, along with header and footer metadata, and can be incrementally consumed by a client application. With reactive capabilities, the semantics of the record stream can be enhanced by allowing a Cypher result to be paused or cancelled part-way through.

To execute a Cypher Query, the **query text** is required along with an optional set of **named parameters**. The text can contain **parameter placeholders** that are substituted with the corresponding values at

runtime. While it is possible to run non-parameterized Cypher Queries, **good programming practice is to use parameters in Cypher Queries whenever possible**. This allows for the caching of queries within the Cypher Engine, which is beneficial for performance. Parameter values should adhere to [Cypher values](#).

A result summary is also generally available. This contains additional information relating to the query execution and the result content. For an **EXPLAIN** or **PROFILE** query, this is where the query plan is returned. See [Cypher Manual → Profiling a query](#) for more information on these queries.

Causal chaining and bookmarks

When working with a **Causal Cluster**, transactions can be chained, via a session, to ensure **causal consistency**. This means that for any two transactions, it is guaranteed that the second transaction will begin only after the first has been successfully committed. This is true even if the transactions are carried out on different physical cluster members. For more information on Causal Clusters, please refer to [Operations Manual → Clustering](#).

Internally, causal chaining is carried out by passing bookmarks between transactions. Each bookmark records one or more points in transactional history for a particular database, and can be used to inform cluster members to carry out units of work in a particular sequence. On receipt of a bookmark, the server will block until it has caught up with the relevant transactional point in time.

An initial bookmark is sent from client to server on beginning a new transaction, and a final bookmark is returned on successful completion. Note that this applies to both read and write transactions.

Bookmark propagation is carried out automatically within sessions and does not require any explicit signal or setting from the application. To opt out of this mechanism, for unrelated units of work, applications can use multiple sessions. This avoids the small latency overhead of the causal chain.

Bookmarks can be passed between sessions by extracting the last bookmark from a session and passing this into the construction of another. Multiple bookmarks can also be combined if a transaction has more than one logical predecessor. Note that it is only when chaining across sessions that an application will need to work with bookmarks directly.

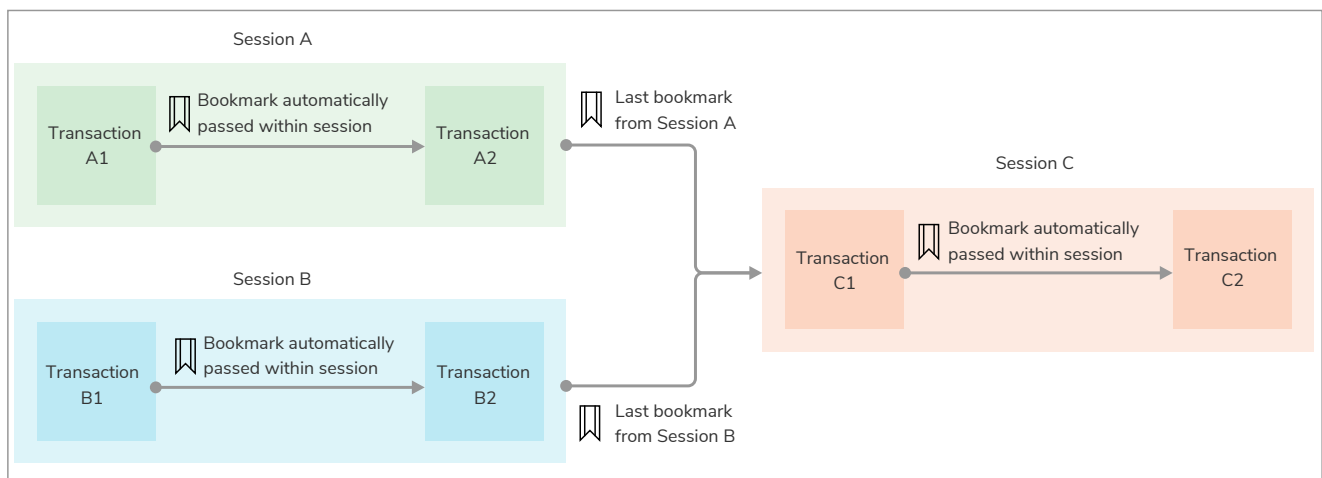


Figure 5. Passing bookmarks

Example 16. Pass bookmarks

```
from neo4j import GraphDatabase

class BookmarksExample:

    def __init__(self, uri, auth):
        self.driver = GraphDatabase.driver(uri, auth=auth)

    def close(self):
        self.driver.close()

    # Create a person node.
    @classmethod
    def create_person(cls, 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.
    @classmethod
    def employ(cls, 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.
    @classmethod
    def create_friendship(cls, 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)

    # Match and display all friendships.
    @classmethod
    def print_friendships(cls, 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"]))

    def main(self):
        saved_bookmarks = [] # To collect the session bookmarks

        # Create the first person and employment relationship.
        with self.driver.session() as session_a:
            session_a.write_transaction(self.create_person, "Alice")
            session_a.write_transaction(self.employ, "Alice", "Wayne Enterprises")
            saved_bookmarks.append(session_a.last_bookmark())

        # Create the second person and employment relationship.
        with self.driver.session() as session_b:
            session_b.write_transaction(self.create_person, "Bob")
            session_b.write_transaction(self.employ, "Bob", "LexCorp")
            saved_bookmarks.append(session_b.last_bookmark())

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

Routing transactions using access modes

Transactions can be executed in either **read** or **write** mode; this is known as the **access mode**. In a Causal Cluster, each transaction will be routed to an appropriate server based on the mode. When using a single instance, all transactions will be passed to that one server.

Routing Cypher by identifying reads and writes can improve the utilization of available cluster resources. Since read servers are typically more plentiful than write servers, it is beneficial to direct read traffic to read servers instead of the write server. Doing so helps in keeping write servers available for write transactions.

Access mode is generally specified by the method used to call the transaction function. Session classes provide a method for calling reads and another for writes.

As a fallback for *auto-commit* and *unmanaged transactions*, a *default access mode* can also be provided at *session level*. This is only used in cases when the access mode cannot otherwise be specified. *In case a transaction function is used within that session, the default access mode will be overridden.*



The driver does not parse Cypher and therefore cannot automatically determine whether a transaction is intended to carry out read or write operations. As a result, a write transaction tagged as a read will still be sent to a read server, but will fail on execution.

Example 17. Read-write transaction

```
def create_person_node(tx, name):
    tx.run("CREATE (a:Person {name: $name})", name=name)

def match_person_node(tx, name):
    result = tx.run("MATCH (a:Person {name: $name}) RETURN count(a)", name=name)
    return result.single()[0]

def add_person(name):
    with driver.session() as session:
        session.write_transaction(create_person_node, name)
        persons = session.read_transaction(match_person_node, name)
    return persons
```

Databases and execution context

Neo4j offers the ability to work with *multiple databases* within the same DBMS.

For *Community Edition*, this is limited to *one user database*, plus the *system* database.

From a driver API perspective, sessions have a DBMS scope, and the default database for a session can be selected on session construction. The default database is used as a target for queries that don't explicitly specify a database with a *USE* clause. See [Cypher Manual → USE](#) for details about the *USE* clause.

In a multi-database environment, *the server tags one database as default*. This is selected whenever a session is created without naming a particular database as default. In an environment with a single database, that database is always the default.

For more information about managing multiple databases within the same DBMS, refer to [Cypher Manual → Neo4j databases and graphs](#), which has a full breakdown of the Neo4j data storage hierarchy.

The following example illustrates how to work with databases:

```

var session = driver.session(SessionConfig.forDatabase( "foo" ))
// lists nodes from database foo
session.run("MATCH (n) RETURN n").list()

// lists nodes from database bar
session.run("USE bar MATCH (n) RETURN n").list()

// creates an index in foo
session.run("CREATE INDEX foo_idx FOR (n:Person) ON n.id").consume()

// creates an index in bar
session.run("USE bar CREATE INDEX bar_idx FOR (n:Person) ON n.id").consume()

// targets System database
session.run("SHOW USERS")

```

Database selection

You pass the name of the database to the driver during session creation. If you don't specify a name, the default database is used. The database name must not be `null`, nor an empty string.



The selection of database is only possible when the driver is connected against Neo4j Enterprise Edition. Changing to any other database than the default database in Neo4j Community Edition will lead to a runtime error.

The following example illustrates the concept of DBMS [Cypher Manual → Transactions](#) and shows how queries to multiple databases can be issued in one driver transaction. It is annotated with comments describing how each operation impacts database transactions.

```

var session = driver.session(SessionConfig.forDatabase( "foo" ))
// a DBMS-level transaction is started
var transaction = session.begin()

// a transaction on database "foo" is started automatically with the first query targeting foo
transaction.run("MATCH (n) RETURN n").list()

// a transaction on database "bar" is started
transaction.run("USE bar MATCH (n) RETURN n").list()

// executes in the transaction on database "foo"
transaction.run("RETURN 1").consume()

// executes in the transaction on database "bar"
transaction.run("USE bar RETURN 1").consume()

// commits the DBMS-level transaction which commits the transactions on databases "foo" and
// "bar"
transaction.commit()

```

Please note that the database that is requested must exist.

Example 18. Database selection on session creation

```
from neo4j import READ_ACCESS
```

```
with driver.session(database="example") as session:  
    session.run("CREATE (a:Greeting {message: 'Hello, Example-Database'}) RETURN a").consume()  
  
with driver.session(database="example", default_access_mode=READ_ACCESS) as session:  
    message = session.run("MATCH (a:Greeting) RETURN a.message as msg").single().get("msg")  
    print(message)
```

Type mapping

Drivers translate between *application language types* and the *Cypher Types*.

To pass parameters and process results, it is important to know the basics of how Cypher works with types and to understand how the Cypher Types are mapped in the driver.

The table below shows the available data types. All types can be potentially found in the result, although not all types can be used as parameters.

Cypher Type	Parameter	Result
<code>null*</code>	<code>⚠</code>	<code>⚠</code>
<code>List</code>	<code>⚠</code>	<code>⚠</code>
<code>Map</code>	<code>⚠</code>	<code>⚠</code>
<code>Boolean</code>	<code>⚠</code>	<code>⚠</code>
<code>Integer</code>	<code>⚠</code>	<code>⚠</code>
<code>Float</code>	<code>⚠</code>	<code>⚠</code>
<code>String</code>	<code>⚠</code>	<code>⚠</code>
<code>ByteArray</code>	<code>⚠</code>	<code>⚠</code>
<code>Date</code>	<code>⚠</code>	<code>⚠</code>
<code>Time</code>	<code>⚠</code>	<code>⚠</code>
<code>LocalTime</code>	<code>⚠</code>	<code>⚠</code>
<code>DateTime</code>	<code>⚠</code>	<code>⚠</code>
<code>LocalDateTime</code>	<code>⚠</code>	<code>⚠</code>
<code>Duration</code>	<code>⚠</code>	<code>⚠</code>
<code>Point</code>	<code>⚠</code>	<code>⚠</code>
<code>Node**</code>		<code>⚠</code>
<code>Relationship**</code>		<code>⚠</code>
<code>Path**</code>		<code>⚠</code>

* The null marker is not a type but a placeholder for absence of value. For information on how to work with null in Cypher, please refer to [Cypher Manual → Working with null](#).

** Nodes, relationships and paths are passed in results as snapshots of the original graph entities. While the original entity IDs are included in these snapshots, no permanent link is retained back to the underlying server-side entities, which may be deleted or otherwise altered independently of the client copies. Graph structures may not be used as parameters because it depends on application context whether such a parameter would be passed by reference or by value, and Cypher provides no mechanism to denote this. Equivalent functionality is available by simply passing either the ID for pass-by-reference, or an extracted map of properties for pass-by-value.

The Neo4j Drivers map Cypher Types to and from native language types as depicted in the table below. Custom types (those not available in the language or standard library) are highlighted in **bold**.

Example 19. Map Neo4j types to Python types

Neo4j Type	Python 3 Type
<code>null</code>	<code>None</code>
<code>List</code>	<code>list</code>
<code>Map</code>	<code>dict</code>
<code>Boolean</code>	<code>bool</code>
<code>Integer</code>	<code>int</code>
<code>Float</code>	<code>float</code>
<code>String</code>	<code>str</code>
<code>ByteArray</code>	<code>bytearray</code>
<code>Date</code>	<code>neo4j.time.Date</code>
<code>Time</code>	<code>neo4j.time.Time</code> [†]
<code>LocalTime</code>	<code>neo4j.time.Time</code> ^{††}
<code>DateTime</code>	<code>neo4j.time.DateTime</code> [†]
<code>LocalDateTime</code>	<code>neo4j.time.DateTime</code> ^{††}
<code>Duration</code>	<code>neo4j.time.Duration</code> [*]
<code>Point</code>	<code>neo4j.spatial.Point</code>
<code>Node</code>	<code>neo4j.graph.Node</code>
<code>Relationship</code>	<code>neo4j.graph.Relationship</code>
<code>Path</code>	<code>neo4j.graph.Path</code>

^{*} A `datetime.timedelta` object passed as a parameter will always be implicitly converted to `neo4j.time.Duration`

[†] Where `tzinfo` is not `None`

† Where `tzinfo` is `None`

Temporal Types

This section lists some basic usage of the temporal types provided by the Python Driver.

DateTime

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

```
from datetime import datetime

from neo4j.time import DateTime
import pytz
```

```
# Create datetimes to be used as query parameters
# Python's builtin datetimes works as well. However, they don't support
# the full feature-set of Neo4j's durations: it has no nanosecond precision.
py_dt = datetime(2021, month=11, day=2, hour=7, minute=47, microsecond=4)
py_dt = pytz.timezone("US/Eastern").localize(py_dt)

# A DateTime can be created from a native datetime
dt = DateTime.from_native(py_dt)
# or directly
dt = DateTime(year=2021, month=11, day=2, hour=7, minute=47,
              nanosecond=4123)
dt = pytz.timezone("US/Eastern").localize(dt)

# Reading a DateTime from a record
dt = record.get("fieldName") # type: DateTime
str(dt) # '2021-11-02T07:47:09.232260000-04:00'

# converting DateTime to native datetime (lossy)
native = dt.to_native() # type: datetime
```

Date

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

```
from datetime import date

from neo4j.time import Date
```

```
# Create dates to be used as query parameters
# Python's builtin dates works as well.
py_d = date(year=2021, month=11, day=2)

# A Date can be created from a native date
d = Date.from_native(py_d)
# or directly
d = Date(year=2021, month=11, day=2)

# Reading a Date from a record
d = record.get("fieldName") # type: Date
str(d) # '2021-11-02'

# converting Date to native date
native = d.to_native() # type: date
```


Time

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

```
from datetime import time

from neo4j.time import Time
import pytz
```

```
# Create datetimes to be used as query parameters
# Python's builtin datetimes works as well. However, they don't support
# the full feature-set of Neo4j's durations: it has no nanosecond precision.
py_t = time(hour=7, minute=47, microsecond=4, tzinfo=pytz.FixedOffset(-240))

# A Time can be created from a native time
t = Time.from_native(py_t)
# or directly
t = Time(hour=7, minute=47, nanosecond=4123, tzinfo=pytz.FixedOffset(-240))

# Reading a Time from a record
t = record.get("fieldName") # type: Time
str(t) # 'T07:47:09.232260000-04:00'

# converting Time to native time (lossy)
native = t.to_native() # type: time
```

Local DateTime

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

```
from datetime import datetime

from neo4j.time import DateTime
import pytz
```

```
# Create datetimes to be used as query parameters
# Python's builtin datetimes works as well. However, they don't support
# the full feature-set of Neo4j's durations: it has no nanosecond precision.
py_dt = datetime(2021, month=11, day=2, hour=7, minute=47, microsecond=4)

# A DateTime can be created from a native datetime
dt = DateTime.from_native(py_dt)
# or directly
dt = DateTime(year=2021, month=11, day=2, hour=7, minute=47,
              nanosecond=4123)

# Reading a DateTime from a record
dt = record.get("fieldName") # type: DateTime
str(dt) # '2021-11-02T07:47:09.232260000'

# converting DateTime to native datetime (lossy)
native = dt.to_native() # type: datetime
```

Local Time

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

```
from datetime import time

from neo4j.time import Time
import pytz
```

```
# Create datetimes to be used as query parameters
# Python's builtin datetimes works as well. However, they don't support
# the full feature-set of Neo4j's durations: it has no nanosecond precision.
py_t = time(hour=7, minute=47, microsecond=4)

# A Time can be created from a native time
t = Time.from_native(py_t)
# or directly
t = Time(hour=7, minute=47, nanosecond=4123)

# Reading a Time from a record
t = record.get("fieldName") # type: Time
str(t) # 'T07:47:09.232260000'

# converting Time to native time (lossy)
native = t.to_native() # type: time
```

Duration

Represents an ISO 8601 duration. Contains both date-based values (years, months, days) and time-based values (seconds, nanoseconds).

```
from datetime import timedelta
from neo4j.time import Duration
```

```
# Creating durations to be used as query parameters
duration = Duration(years=1, days=2, seconds=3, nanoseconds=4)
# Python's builtin timedeltas works as well. However, they don't support
# the full feature-set of Neo4j's durations,
# e.g., no nanoseconds and no months.
py_duration = timedelta(days=2, seconds=3, microseconds=4)

# Reading a Duration from a record
duration = record.get("fieldName") # type: Duration
str(duration) # 'P1Y2DT3.000000004S'
```

Geospatial Types

This section lists some basic usage of the geospatial types provided by the Python Driver.

CartesianPoint

Represents 2 or 3 dimensional point in Cartesian space.

```
from neo4j.spatial import CartesianPoint
```

```
# Creating a 2D point in Cartesian space
point2d = CartesianPoint((1, 5.1))
# Or in 3D
point3d = CartesianPoint((1, -2., 3.1))

# Reading a 2D point from a record
point2d = record_with_2d_point.get("fieldName") # type: CartesianPoint
str(point2d) # POINT(1.0 5.1)
point2d.x # 1.0
point2d.y # 5.1
# point2d.z raises AttributeError
point2d.srid # 7203
len(point2d) # 2

# Reading a 3D point from a record
point3d = record_with_3d_point.get("fieldName") # type: CartesianPoint
str(point3d) # POINT(1.0 -2.0 3.1)
point3d.x # 1.0
point3d.y # -2.0
point3d.z # 3.1
point3d.srid # 9157
len(point2d) # 3
```

WGS84Point

Represents an instant capturing the wgs84, but not the time, nor the timezone.

```
from neo4j.spatial import WGS84Point
```

```
# Creating a 2D point in WSG84 space
point2d = WGS84Point((1, 5.1))
# Or in 3D
point3d = WGS84Point((1, -2., 3.1))

# Reading a 2D point from a record
point2d = record_with_2d_point.get("fieldName") # type: WGS84Point
str(point2d) # POINT(1.0 5.1)
point2d.longitude # 1.0 (point2d.x is an alias for longitude)
point2d.latitude # 5.1 (point2d.y is an alias for latitude)
# point2d.height raises AttributeError (same with point2d.z)
point2d.srid # 4326
len(point2d) # 2

# Reading a 3D point from a record
point3d = record_with_3d_point.get("fieldName") # type: WGS84Point
str(point3d) # POINT(1.0 -2.0 3.1)
point3d.longitude # 1.0 (point3d.x is an alias for longitude)
point3d.latitude # -2.0 (point3d.y is an alias for latitude)
point3d.height # 3.1 (point3d.z is an alias for height)
point3d.srid # 4979
len(point2d) # 3
```

Exceptions and error handling

When executing Cypher or carrying out other operations with the driver, certain exceptions and error cases may arise. **Server-generated exceptions** are each associated with a **status code** that describes the nature of the problem and a message that provides more detail.

The classifications are listed in the table below.

Table 11. Server status code classifications

Classification	Description
ClientError	The client application has caused an error. The application should amend and retry the operation.
DatabaseError	The server has caused an error. Retrying the operation will generally be unsuccessful.
TransientError	A temporary error has occurred. The application should retry the operation.

Service unavailable

A **Service Unavailable** exception will be signalled when the driver is no longer able to establish communication with the server, even after retries.

Encountering this condition usually indicates a fundamental networking or database problem.

While certain mitigations can be made by the driver to avoid this issue, there will always be cases when this is impossible. As such, it is highly recommended to ensure client applications contain a code path that can be followed when the client is no longer able to communicate with the server.

Transient errors

Transient errors are those which are generated by the server and marked as safe to retry without alteration to the original request. Examples of such errors are deadlocks and memory issues.

When using transaction functions, the driver will usually be able to automatically retry when a transient failure occurs.

Exceptions and error handling

When executing Cypher or carrying out other operations with the driver, certain exceptions and error cases may arise. **Server-generated exceptions** are each associated with a **status code** that describes the nature of the problem and a message that provides more detail.

The classifications are listed in the table below.

Table 12. Server status code classifications

Classification	Description
ClientError	The client application has caused an error. The application should amend and retry the operation.
DatabaseError	The server has caused an error. Retrying the operation will generally be unsuccessful.

Classification	Description
<code>TransientError</code>	A temporary error has occurred. The application should retry the operation.

Service unavailable

A **Service Unavailable** exception will be signalled when the driver is no longer able to establish communication with the server, even after retries.

Encountering this condition usually indicates a fundamental networking or database problem.

While certain mitigations can be made by the driver to avoid this issue, there will always be cases when this is impossible. As such, it is highly recommended to ensure client applications contain a code path that can be followed when the client is no longer able to communicate with the server.

Transient errors

Transient errors are those which are generated by the server and marked as safe to retry without alteration to the original request. Examples of such errors are deadlocks and memory issues.

When using transaction functions, the driver will usually be able to automatically retry when a transient failure occurs.

The session API

This section details the Session API that is made available by the driver.

Simple sessions

Simple sessions provide a "classic" **blocking style** API for Cypher execution. In general, simple sessions provide the easiest programming style to work with since API calls are executed in a strictly sequential fashion.

Lifecycle

The session lifetime extends from session construction to session closure. In languages that support them, simple sessions are usually scoped within a context block; this ensures that they are properly closed and that any underlying connections are released and not leaked.

```
with driver.session(...) as session:  
    // transactions go here
```

Sessions can be configured in a number of different ways. This is carried out by supplying configuration inside the session constructor. See [Session configuration](#) for more details.

Transaction functions

Transaction functions are used for containing transactional units of work. This form of transaction requires minimal boilerplate code and allows for a clear separation of database queries and application logic.

Transaction functions are also desirable since they encapsulate retry logic and allow for the greatest degree of flexibility when swapping out a single instance of server for a cluster.

Transaction functions can be called as either read or write operations. This choice will route the transaction to an appropriate server within a clustered environment. If you are operating in a single instance environment, this routing has no impact. It does give you flexibility if you choose to adopt a clustered environment later on.

Before writing a transaction function *it is important to ensure that it is designed to be idempotent*. This is because a function may be executed multiple times if initial runs fail.

Any query results obtained within a transaction function should be consumed within that function, as connection-bound resources cannot be managed correctly when out of scope. To that end, transaction functions can return values but these should be derived values rather than raw results.



Transaction functions are the recommended form for containing transactional units of work.

When a transaction fails, the driver retry logic is invoked. For several failure cases, the transaction can be immediately retried against a different server. These cases include connection issues, server role changes (e.g. leadership elections) and transient errors.

Example 20. Transaction function

```
from neo4j import unit_of_work
```

```
@unit_of_work(timeout=5)
def create_person(tx, name):
    return tx.run("CREATE (a:Person {name: $name}) RETURN id(a)", name=name).single().value()

def add_person(driver, name):
    with driver.session() as session:
        return session.write_transaction(create_person, name)
```

Auto-commit transactions

An auto-commit transaction is a basic but limited form of transaction. Such a transaction consists of only one Cypher query and is not automatically retried on failure. Therefore, any error scenarios will need to be handled by the client application itself.

Auto-commit transactions serve the following purposes:

- simple use cases such as when learning Cypher or writing one-off scripts.
- operations such as batched data load operations, where the driver cannot be aware of the committed state and therefore cannot safely request a retry. The operator will have to perform a **retry** or **undo** under these circumstances.



The driver does not retry auto-commit queries on failure since it is unaware of what state has been committed at the point of failure.



Unlike other kinds of Cypher query, **PERIODIC COMMIT** and **CALL {} IN TRANSACTIONS** queries do not participate in the causal chain. Therefore, the only way to execute these queries from a driver is to use auto-commit transactions.

Please refer to the following for more details:

- [Cypher Manual](#) → **PERIODIC COMMIT** query hint
- [Cypher Manual](#) → **CALL {} IN TRANSACTIONS**

Example 21. Simple auto-commit transactions

```
from neo4j import Query
```

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

# Alternative implementation, with a one second timeout
def add_person_within_a_second(self, name):
    with self.driver.session() as session:
        session.run(Query("CREATE (a:Person {name: $name})", timeout=1.0), name=name)
```

Consuming results

Query results are typically consumed as a stream of records. The drivers provide a way to iterate through that stream.

Example 22. Consuming results

```
def match_person_nodes(tx):
    result = tx.run("MATCH (a:Person) RETURN a.name ORDER BY a.name")
    return [record["a.name"] for record in result]

with driver.session() as session:
    people = session.read_transaction(match_person_nodes)
```

Retaining results

Within a session, **only one result stream can be active at any one time**. Therefore, if the result of one query is not fully consumed before another query is executed, the remainder of the first result will be automatically buffered within the result object.

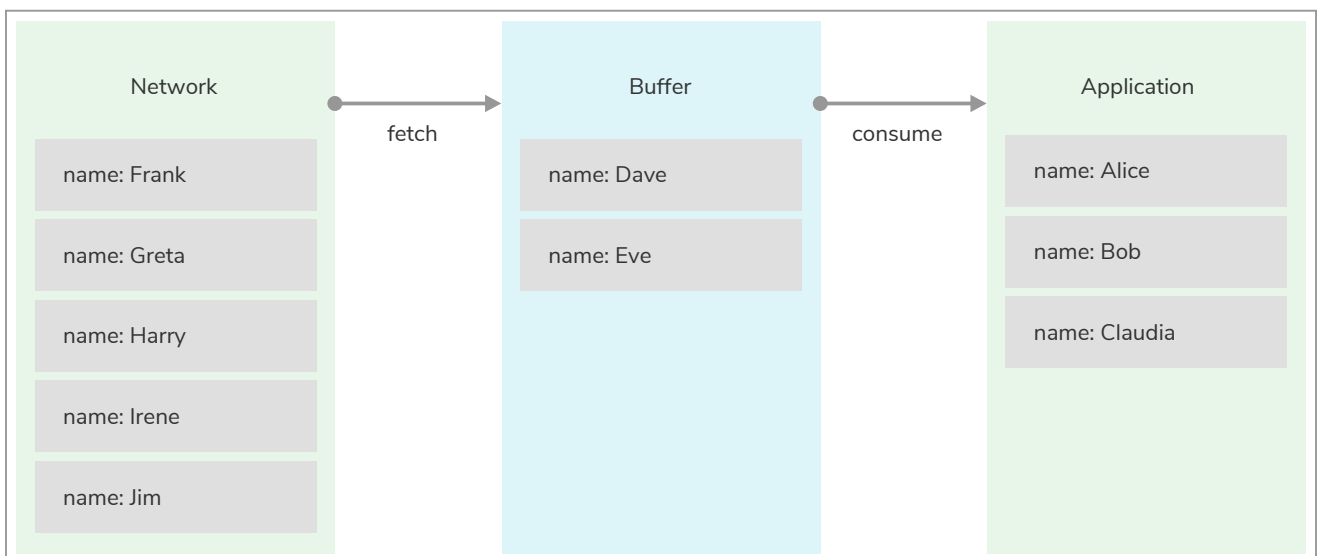


Figure 6. Result buffer

This buffer provides a staging point for results, and divides result handling into **fetching** (moving from the network to the buffer) and **consuming** (moving from the buffer to the application).



For large results, the result buffer may require a significant amount of memory.

For this reason, *it is recommended to consume results in order wherever possible.*

Client applications can choose to take control of more advanced query patterns by explicitly retaining results. Such explicit retention may also be useful when a result needs to be saved for future processing. The drivers offer support for this process, as per the example below:

```
def add_employee_to_company(tx, person, company_name):
    tx.run("MATCH (emp:Person {name: $person_name}) "
           "MERGE (com:Company {name: $company_name}) "
           "MERGE (emp)-[:WORKS_FOR]->(com)",
           person_name=person["name"], company_name=company_name)
    return 1

def match_person_nodes(tx):
    return list(tx.run("MATCH (a:Person) RETURN a.name AS name"))

def add_employees(company_name):
    employees = 0
    with driver.session() as session:
        persons = session.read_transaction(match_person_nodes)

        for person in persons:
            employees += session.write_transaction(add_employee_to_company, person, company_name)

    return employees
```

Session configuration

Bookmarks

The mechanism which ensures causal consistency between transactions within a session. Bookmarks are implicitly passed between transactions within a single session to meet the causal consistency requirements. There may be scenarios where you might want to use the bookmark from one session in a different new session.

Default: None (Sessions will initially be created without a bookmark)

DefaultAccessMode

A fallback for the access mode setting when transaction functions are not used. Typically, access mode is set per transaction by calling the appropriate transaction function method. In other cases, this setting is inherited. Note that transaction functions will ignore/override this setting.

Default: Write

Database

The database with which the session will interact. When you are working with a database which is not the default (i.e. the **system** database or another database in Neo4j 4.0 Enterprise Edition), you can explicitly configure the database which the driver is executing transactions against.



The resolution of database aliases occurs at connection creation time, which is not under the control of the session. It is therefore not recommended to alter database aliases while there are live sessions. The per-user home database is resolved at session creation and when first impersonating a user. Therefore, the creation of a new session is necessary to reflect a changed home database.

See [Operations Manual → The default database](#) for more information on databases.

Default: the default database as configured on the server.

Fetch Size

The number of records to fetch in each batch from the server. Neo4j 4.0 introduces the ability to pull records in batches, allowing the client application to take control of data population and apply back pressure to the server.

Default: 1000 records

ImpersonatedUser

Users can run transactions against the database as different users if they have been granted explicit permission for doing so. 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.).

See [Operations Manual → Per-user home databases](#) for more information on default database per-user.

Default: None (Sessions will be created with the logged user)

Appendix A: Driver terminology

This section lists the relevant terminology related to Neo4j drivers.

access mode

The mode in which a transaction is executed, either read or write.

acquire (connection)

To borrow a driver connection that is not currently in use from a connection pool.

auto-commit

A single query which is wrapped in a transaction and committed automatically.

Bolt

Bolt is a Neo4j proprietary, binary protocol used for communication between client applications and database servers. Bolt is versioned independently from the database and the drivers.

Bolt Routing Protocol

The steps required for a driver to obtain a routing table from a cluster member.

Bolt server

A Neo4j instance that can accept incoming Bolt connections.

bookmark

A marker for a point in the transactional history of Neo4j.

causal chaining

A mechanism to ensure that the transactions carried out in a session are executed in order, even when each transaction may be carried out on different cluster members.

client application

A piece of software that interacts with a database server via a driver.

connection

A persistent communication channel between a client application and a database server.

connection pool

A set of connections maintained for quick access, that can be acquired and released as required.

driver (object)

A globally accessible controller for all database access.

driver (package)

A software library that provides access to Neo4j from a particular programming language. The Neo4j drivers implement the [Bolt](#) protocol.

query result

The stream of records that are returned on execution of a query.

release (connection)

To return a connection back into a connection pool after use.

routing driver

A driver that can route traffic to multiple members of a cluster using the routing protocol.

routing table

A set of server addresses that identify cluster members associated with roles.

server address

A combination of host name and port or IP address and port that targets a server.

session

A causally linked sequence of transactions.

thread safety

See https://en.wikipedia.org/wiki/Thread_safety.

transaction

A transaction comprises a unit of work performed against a database. It is treated in a coherent and reliable way, independent of other transactions. A transaction, by definition, must be atomic, consistent, isolated, and durable.

transaction function

The method of grouping a number of queries together which, when run in a session, are retried on failure.

transaction manager

The component/code responsible for deciding what to do if a transaction fails, i.e to retry, give up or do something else.

License

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

You are free to

Share

copy and redistribute the material in any medium or format

Adapt

remix, transform, and build upon the material

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

Under the following terms

Attribution

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

NonCommercial

You may not use the material for commercial purposes.

ShareAlike

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

No additional restrictions

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

Notices

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

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

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