The Neo4j Drivers Manual v1.7 for Python

Table of Contents

This is the driver manual for Neo4j official drivers version 1.7, authored by the Neo4j Team.

This manual covers the following areas:

- Get started — An overview of the official Neo4j drivers and how to connect to a Neo4j database.
- Client applications — How to manage database connections within an application.
- Sessions and transactions — How to create units of work and provide a logical context for that work.
- Working with Cypher values — 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 the developer of a Neo4j client application.

Get started

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

About the official drivers

Neo4j provides official drivers for many popular programming languages. These drivers are supported by Neo4j and come with full cluster routing support.

Community drivers also exist for many languages, but vary greatly in terms of feature sets, maturity, and support. To find out 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 1.x driver series

Language/framework	Versions supported
Java	Java 8+ (latest patch releases).
Python	CPython 2.7, 3.4, 3.5, 3.6 and 3.7 (latest patch releases). Note that Python 2 support is deprecated and will be discontinued in the 2.x series drivers.
JavaScript	All LTS versions of Node.JS, specifically the 4.x and 6.x series runtimes (https://github.com/nodejs/LTS).
.NET	.NET standard 1.3 (https://github.com/dotnet/standard/blob/master/docs/versions.md) and .NET 4.5.2 and above.
Go	Go 1.10, 1.11 (latest patch releases).

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

The driver *major.minor* version number describes the feature set available within that driver. Across languages, drivers are generally released at similar times, although patches are driver-dependent. The 1.x series drivers have been built for Neo4j 3.x.

It is recommended to always use the latest driver release available. This will ensure that all server functionality is made available to client applications. To install a driver or to find out more about which driver versions are available, use the relevant language distribution system, described below.

Example 1. Acquire the driver

To find out the latest version of the driver, visit https://pypi.python.org/pypi/neo4j-driver.

To install the latest version of the driver:

```
pip install neo4j-driver
```

You can also choose to install a certain version of the driver.

The following is the syntax for installing a certain version of the driver.

```
pip install neo4j-driver==$PYTHON_DRIVER_VERSION
```

In the following example we are installing driver version 1.7.1.

```
pip install neo4j-driver==1.7.1
```

You can review the release notes for this driver here.

A "Hello World" example

The example below shows the minimal configuration necessary to interact with Neo4j through a driver.

Example 4. Hello World

```
class HelloWorldExample(object):

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

    def close(self):
        self._driver.close()

    def print_greeting(self, message):
        with self._driver.session() as session:
            greeting = session.write_transaction(self._create_and_return_greeting, message)
            print(greeting)

    @staticmethod
    def _create_and_return_greeting(tx, message):
        result = tx.run("CREATE (a:Greeting) "
                        "SET a.message = $message "
                        "RETURN a.message + ', from node ' + id(a)", message=message)
        return result.single()[0]
```

Driver API docs

For a comprehensive listing of all driver functionality, refer to the API documentation for the specific

language driver.

Example 5. API docs

```
https://neo4j.com/docs/api/python-driver/1.7/
```

Client applications

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

The driver object

A Neo4j client application will require a driver instance in order to provide access to the database. This driver should be made available to all parts of the application that need to interact with Neo4j. In languages where thread safety is an issue, the driver can be considered thread-safe.

A note on lifecycle

Applications will typically construct a driver instance on startup and destroy it on exit. Destroying a driver instance will immediately shut down any connections previously opened via that driver; for drivers that contain a connection pool, the entire pool will be shut down.

To construct a driver instance, a connection URI and authentication information must be supplied. Additional configuration details can be supplied if required. All of these details are immutable for the lifetime of the driver. Therefore, if multiple configurations are required (such as when working with multiple database users) then multiple driver instances must be used.

An example of driver construction and destruction can be seen below:

Example 6. The driver lifecycle

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

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

Connection URIs

A connection URI identifies a graph database and how to connect to it. The official Neo4j drivers currently support the following URI schemes and driver object types:

Table 2. Available URI schemes

URI Scheme	Driver object type
bolt	Direct driver
bolt+routing	Routing driver

Direct drivers (bolt)

A direct driver is created via a `bolt` URI, for example: `bolt://localhost:7687`. This kind of driver is used to maintain connections to a single database server and is typically used when working with a single Neo4j instance or when targeting a specific member of a cluster. Note that a routing driver is generally preferable when working with a Causal Cluster.

Routing drivers (bolt+routing)

A routing driver is created via a `bolt+routing` URI, for example: `bolt+routing://graph.example.com:7687`. The address in the URI must be that of a Core Server. This kind of driver uses the Bolt Routing Protocol and works in tandem with the cluster to route transactions to available cluster members.

Routing drivers with routing context

Routing drivers with routing context are an available option when using drivers of version 1.3 or above together with a Neo4j Causal Cluster of version 3.2 or above. In such a setup, a routing driver can include a preferred routing context via the query part of the `bolt+routing` URI.

In the standard Neo4j configuration, routing contexts are defined on the server side by means of *server policies*. Thus the driver communicates the routing context to the cluster in the form of a server policy. It then obtains refined routing information back from the cluster, based on the server policy.

The address in the URI of a routing driver with routing context must be that of a Core Server.

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`:

```
bolt+routing://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.

Cluster bootstrap address resolution

Drivers have the ability to intercept cluster bootstrap address resolution. This is generally performed by a routing driver on construction or after complete contact has been lost with a cluster. This hook can be useful to augment regular DNS resolution and can be used to initialize a routing driver with multiple server addresses.

The intercept hook is provided as a driver configuration option and takes the form of a callback function that accepts a single input address and yields or returns multiple output addresses. This function may hard code the output addresses or may draw them from another configuration source, as required. Output addresses can contain host names or IP addresses and are forwarded to the connection pool as initial routers.

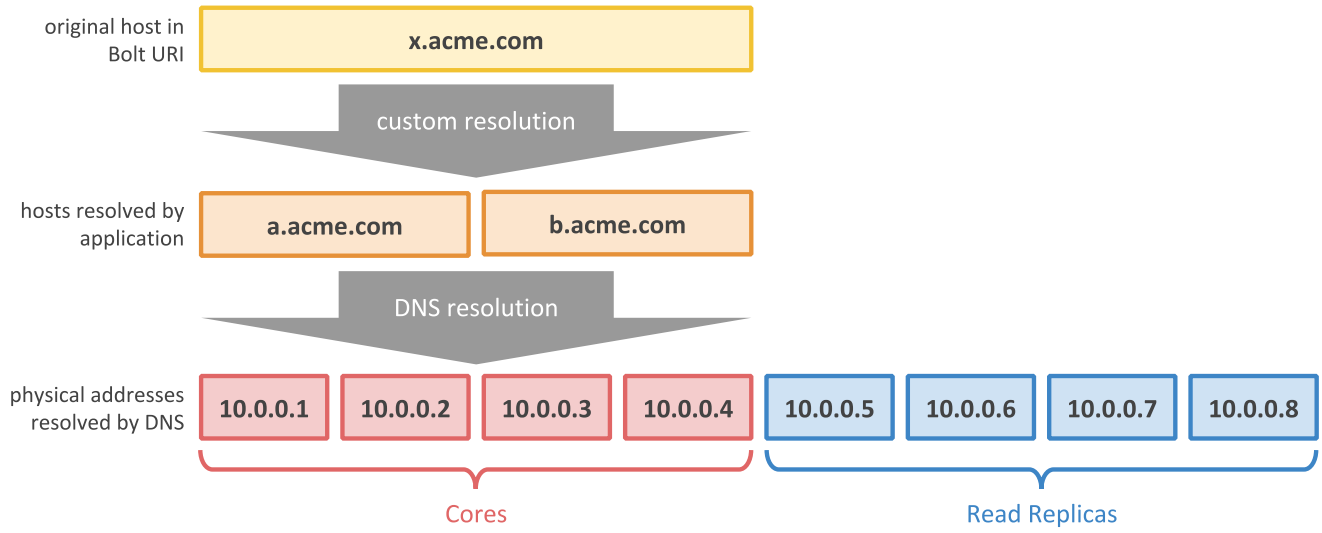

Figure 1. Cluster bootstrap address resolution

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

Example 8. Custom Address Resolver

```
class CustomResolverExample:

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

    def resolve(self, 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

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

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 9. Basic authentication

```
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 10. Kerberos authentication

```
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 11. Custom authentication

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

Configuration

Encryption

TLS encryption is enabled for all connections by default. This can be disabled through configuration as follows:

Example 12. Unencrypted

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

The server can be modified to require encryption for all connections. Please see the Operations Manual Configure Neo4j connectors for more information.

An attempt to connect to a server using an encryption setting not allowed by that server will result in a *Service unavailable* status.

Trust

During a TLS handshake, the server provides a certificate to the client application. The application can choose to accept or reject this certificate based on one of the following trust strategies:

Table 3. Trust strategies

Trust strategy	Description
`TRUST_ALL_CERTIFICATES` (default)	Accept any certificate provided by the server, regardless of CA chain.
`TRUST_CUSTOM_CA_SIGNED_CERTIFICATES`	Accept any certificate that can be verified against a custom CA.
`TRUST_SYSTEM_CA_SIGNED_CERTIFICATES`	Accept any certificate that can be verified against the system store.

Server Name Indication (SNI) is enabled by default, allowing proxy middleware to route encrypted Bolt connections to the correct backend server.

Example 13. Trust

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

Connection pool management

The driver maintains a pool of connections. The pooled connections are reused by sessions and transactions to avoid the overhead added by establishing new connections for every query. The connection pool always starts up empty. New connections are created on demand by sessions and transactions. When a session or a transaction is done with its execution, the connection will be returned to the pool to be reused.

Application users can tune connection pool settings to configure the driver for different use cases based on client performance requirements and database resource consumption limits.

Detailed descriptions of connection pool settings available via driver configuration are listed below:

`MaxConnectionLifetime`

Pooled connections older than this threshold will be closed and removed from the pool. The actual removal happens during connection acquisition so that the new session returned is never backed by an old connection. Setting this option to a low value will cause a high connection churn and might 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 value: 1h.

`MaxConnectionPoolSize`

This setting defines the maximum total number of connections allowed, per host, to be managed by the connection pool. In other words, for a direct driver, this sets the maximum number of connections towards a single database server. For a routing driver 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`. Default value: This is different for different drivers, but is a number in the order of 100.

`ConnectionAcquisitionTimeout`

This setting limits the amount of time a session or transaction can spend waiting for a free connection to appear in the pool before throwing an exception. The exception thrown in this case is `ClientException`. Timeout only applies when connection pool is at its max capacity. Default value: 1m.

Example 14. Connection pool management

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

Connection timeout

To configure the maximum time allowed to establish a connection, pass a duration value to the driver configuration. For example:

Example 15. Connection timeout

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

Max retry time

To configure retry behavior, supply a value for the maximum time in which to keep attempting retries of transaction functions. For example:

Example 16. Max retry time

```
def __init__(self, uri, user, password):
    self._driver = GraphDatabase.driver(uri, auth=(user, password), max_retry_time=15)
```

Note that the time specified here does not take into account the running time of the unit of work itself, merely a limit after which retries will no longer be attempted.

Service unavailable

A Service unavailable status 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. Applications should be designed to cater for this eventuality.

Example 17. Service unavailable

```
def add_item(self):
    try:
        with self._driver.session() as session:
            session.write_transaction(lambda tx: tx.run("CREATE (a:Item)"))
        return True
    except ServiceUnavailable:
        return False
```

Sessions and transactions

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

Sessions

A session is a container for a sequence of transactions. Sessions borrow connections from a pool as required and so should be considered lightweight and disposable. In languages where thread safety is an issue, a session should *not* be considered thread-safe.

In languages that support them, 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.

Example 18. Session

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

Transactions

Transactions are atomic units of work consisting of one or more Cypher statement executions. A transaction is executed within a session.

To execute a Cypher statement, two pieces of information are required: the statement template and a keyed set of parameters. The template is a string containing placeholders that are substituted with parameter values at runtime. While it is possible to run non-parameterized Cypher, good programming practice is to use parameters in Cypher statements. This allows for caching of statements within the Cypher engine, which is beneficial for performance. Parameter values should adhere to Working with Cypher values.

The Neo4j driver API provides for three forms of transaction:

- Auto-commit transactions
- Transaction functions
- Explicit transactions

Of these, only transaction function can be automatically replayed on failure.

Auto-commit Transactions

An auto-commit transaction is a simple but limited form of transaction. Such a transaction consists of only one Cypher statement, cannot be automatically replayed on failure, and cannot take part in a causal chain.

An auto-commit transaction is invoked using the `session.run` method:

Example 19. Auto-commit transaction

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

# Alternative implementation, with timeout
def add_person_within_half_a_second(self, name):
    with self._driver.session() as session:
        session.run(Statement("CREATE (a:Person {name: $name})", timeout=0.5), name=name)
```

Auto-commit transactions are sent to the network and acknowledged immediately. This means that multiple transactions cannot share network packets, thereby exhibiting a lesser network efficiency than other forms of transaction.

Auto-commit transactions are intended to be used for simple use cases such as when learning Cypher or writing one-off scripts. It is not recommended to use auto-commit transactions in production environments or when performance or resilience are a primary concern.

However, Auto-commit transactions are the only way to execute `USING PERIODIC COMMIT` Cypher statements.

Transaction functions

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

Example 20. Transaction function

```
def add_person(driver, name):
    with driver.session() as session:
        # Caller for transactional unit of work
        return session.write_transaction(create_person_node, name)


# Simple implementation of the unit of work
def create_person_node(tx, name):
    return tx.run("CREATE (a:Person {name: $name}) RETURN id(a)", name=name).single().value()


# Alternative implementation, with timeout
@unit_of_work(timeout=0.5)
def create_person_node_within_half_a_second(tx, name):
    return tx.run("CREATE (a:Person {name: $name}) RETURN id(a)", name=name).single().value()
```

Transaction functions are also able to handle connection problems and transient errors using an automatic retry mechanism. This retry capability can be configured on Driver construction.

Any query results obtained within a transaction function should be consumed within that function. Transaction functions can return values but these should be derived values rather than raw results.

Explicit transactions

Explicit transactions are the longhand form of transaction functions, providing access to explicit `BEGIN`,

`COMMIT` and `ROLLBACK` operations. While this form is useful for a handful of use cases, it is recommended to use transaction functions wherever possible.

Transaction configuration

Both auto-commit and explicit transactions can be assigned *metadata*, and a *timeout*.

Metadata is a map of arbitrary values that will be attached to the executing transaction. It will be visible in the output of the procedures `dbms.listQueries` and `dbms.listTransactions`. It will also get logged to the `query.log`.

A timeout will cause the related transaction to be terminated by the database if its execution time is longer than the provided value. This functionality allows to limit query/transaction execution time. A specified timeout overrides the default timeout defined by the configuration setting `config_dbms.transaction.timeout`.

For usage patterns, please refer to the API documentation for your specific language.

Cypher errors

When executing Cypher, it is possible for an exception to be thrown by the Cypher engine. Each such exception is associated with a status code that describes the nature of the error and a message that provides more detail.

The error classifications are listed in the table below.

Table 4. Error 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.

Causal chaining

When working with a Causal Cluster, transactions can be chained 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.

Due to protocol limitations, auto-commit transactions cannot currently take part in the causal chain. While this is expected to change in a future protocol release, `session.run` calls should be avoided in places where causal consistency is important.

Causal chaining is carried out by passing bookmarks between transactions. Each bookmark records a point in transactional history and can be used to inform cluster members to carry out units of work in a particular sequence. Internally, a bookmark is passed from server to client on a successful COMMIT and back from client to server on BEGIN. On receipt of one or more bookmarks, the transaction server will block until it has fast forwarded to catch up with the latest of these.

Within a session, bookmark propagation is carried out automatically 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. Propagation between sessions can be achieved by extracting the last bookmarks from one or more sessions and passing these into the construction of another. This is generally the only case in which an application will need to work with bookmarks directly.

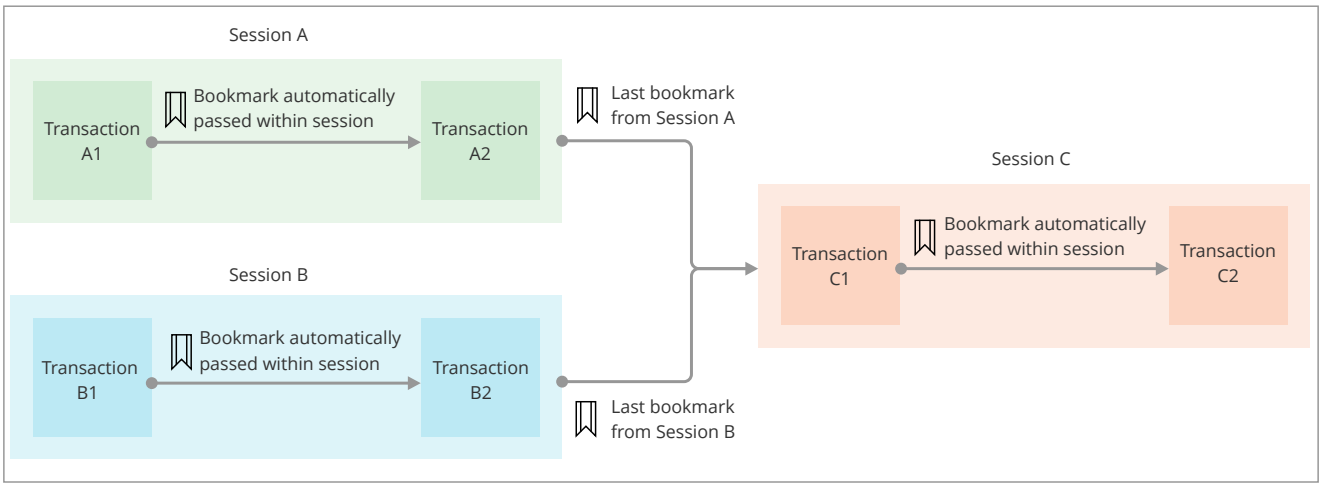

Figure 2. Passing bookmarks

The following example illustrates the passing of bookmarks between sessions. First consider the code:

Example 21. Passing bookmarks between sessions

```
class BookmarksExample(object):

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

    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)
```

We are using three separate sessions: a, b and c. In *session a* we run two separate transactions. In the first one we create the person `Alice`, and in the second one we record that she works at `Wayne Enterprises`. The bookmark being passed between the two transactions is handled by the session. The bookmark from the last transaction is saved into an array for future use.

In *session b* we also run two separate transactions. In the first one we create the person `Bob`, and in the second one we record that he works at `LexCorp`. Again, the bookmark being passed between the two transactions is handled by the session. The bookmark from the last transaction is saved into an array for future use.

In the last session, *session c*, we wish to create a friendship between Alice and Bob. This can only be done if both `Alice` and `Bob` have been created first. In order to ensure this, we pass the bookmarks from the last transactions in *session a* and *session b*, respectively.

If you try to extract a bookmark from a database which is not running in Causal Cluster mode, you will receive a `null` result.

Access modes

Transactions can be executed in either `read` or `write` 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: as read servers are typically more plentiful than write servers, it is beneficial to direct as much as possible of read transactions to read servers. Doing so helps keeping write servers available for write transactions.

Access modes can be supplied in two ways: per transaction or per session. An access mode specified at session creation can be overridden by the access mode of a transaction within that session. In the general case, access mode should always be specified at transaction level, using transaction functions. The session-level setting is only necessary for explicit and auto-commit transactions.

Note that the driver does not parse Cypher and cannot determine whether a transaction is intended to carry out read or write operations. As a result of this, a `write` transaction tagged for `read` will be sent to a read server, but will fail on execution.

Example 22. Read-write transaction

```
def add_person(self, name):
    with self._driver.session() as session:
        session.write_transaction(self.create_person_node, name)
        return session.read_transaction(self.match_person_node, name)

@staticmethod
def create_person_node(tx, name):
    tx.run("CREATE (a:Person {name: $name})", name=name)
    return None

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

Asynchronous programming

Java, .NET and JavaScript all support asynchronous programming. The examples here highlight specifically how Java and .NET provide for this programming model alongside their blocking API.

In addition to the methods listed in the previous sections, there also exist several asynchronous methods which allow for better integration with applications written in an asynchronous style. Asynchronous methods are named as their synchronous counterparts but with an additional *async* prefix.

Example 23. Asynchronous programming examples

Working with Cypher values

This section describes the types and values used by Cypher and how they map to native language types.

The Cypher type system

Drivers translate between application language types and the Cypher type system. 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 can be potentially found in results although not all can be used as parameters.

Cypher Type	Parameter	Result
`null`*	✓	✓
`List`	✓	✓
`Map`	✓	✓
`Boolean`	✓	✓
`Integer`	✓	✓
`Float`	✓	✓
`String`	✓	✓
`ByteArray`	✓	✓
`Date`	✓	✓
`Time`	✓	✓
`LocalTime`	✓	✓
`DateTime`	✓	✓
`LocalDateTime`	✓	✓
`Duration`	✓	✓
`Point`	✓	✓
`Node`**		✓
`Relationship`**		✓
`Path`**		✓

* 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.pdf.

** 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 driver maps 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 24. Map Neo4j types to native language types

Neo4j type	Python 2 type	Python 3 type
`null`	`None`	`None`
`List`	`list`	`list`
`Map`	`dict`	`dict`
`Boolean`	`bool`	`bool`
`Integer`	`int / long`*	`int`
`Float`	`float`	`float`
`String`	`unicode`**	`str`
`ByteArray`	`bytearray`	`bytearray`
`Date`	**`neotime.Date`**	**`neotime.Date`**
`Time`	**`neotime.Time`**†	**`neotime.Time`**†
`LocalTime`	**`neotime.Time`**††	**`neotime.Time`**††
`DateTime`	**`neotime.DateTime`**†	**`neotime.DateTime`**†
`LocalDateTime`	**`neotime.DateTime`**††	**`neotime.DateTime`**††
`Duration`	**`neotime.Duration`*****	**`neotime.Duration`*****
`Point`	**`Point`**	**`Point`**
`Node`	**`Node`**	**`Node`**
`Relationship`	**`Relationship`**	**`Relationship`**
`Path`	**`Path`**	**`Path`**

* While Cypher uses 64-bit signed integers, `int` can only hold integers up to `sys.maxint` in Python 2; `long` is used for values above this.

** In Python 2, a `str` passed as a parameter will always be implicitly converted to `unicode` via UTF-8.

*** A `timedelta` passed as a parameter will always be implicitly converted to `neotime.Duration`.

† Where tzinfo is not None.

†† Where tzinfo is None.

Statement results

A statement result is comprised of a stream of records. The result is typically handled by the receiving application as it arrives, although it is also possible to retain all or part of a result for future consumption.

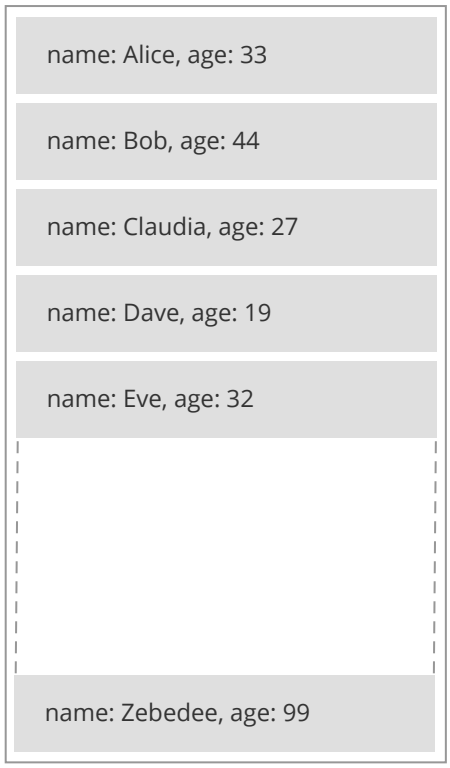

Figure 3. Result stream

Records

A record provides an immutable view of part of a result. It is an ordered map of keys and values. As values within a record are both keyed and ordered, that value can be accessed either by position (0-based integer) or by key (string).

The buffer

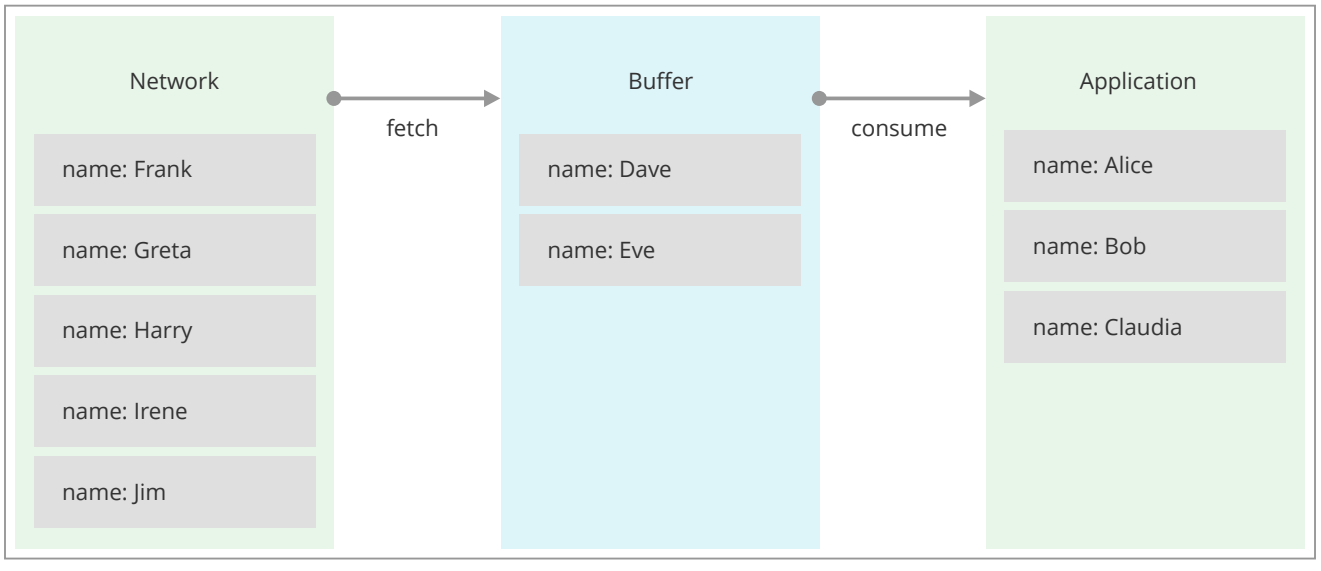

Figure 4. Result buffer

Most drivers contain a result buffer. This 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).

If results are consumed in the same order as they are produced, records merely pass through the buffer; if they are consumed out of order, the buffer will be utilized to retain records until they are

consumed by the application. For large results, this may require a significant amount of memory and impact performance. For this reason, it is recommended to consume results in order wherever possible.

Consuming the stream

Query results will often be consumed as a stream. Drivers provide a language-idiomatic way to iterate through a result stream.

Example 25. Consuming the stream

```
def get_people(self):
    with self._driver.session() as session:
        return session.read_transaction(self.match_person_nodes)

@staticmethod
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]
```

Retaining results

The result record stream is available until another statement is run in the session, or until the current transaction is closed. To hold on to the results beyond this scope, the results need to be explicitly retained. For retaining results, each driver offers methods that collect the result stream and translate it into standard data structures for that language. Retained results can be processed while the session is free to take on the next workload. The saved results can also be used directly to run a new statement.

Example 26. Retain results for further processing

```
def add_employees(self, company_name):
    with self._driver.session() as session:
        employees = 0
        persons = session.read_transaction(self.match_person_nodes)

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

        return employees

@staticmethod
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

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

Statement result summaries

Supplementary information such as query statistics, timings and server information can be obtained from the statement result summary. If this detail is accessed before the entire result has been consumed, the remainder of the result will be buffered.

See also the language-specific driver API documentation.

Appendix A: Driver terminology

This section lists the relevant terminology related to Neo4j drivers.

acquire (connection)
: To borrow a driver connection that is not currently in use from a connection pool.

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.

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.

direct driver
: A driver that can connect to a single server address.

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.

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.

statement result

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

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.