The Neo4j Java Driver Manual v4.3

Table of Contents

> *This is the manual for Neo4j Java Driver version 4.3, authored by the Neo4j Team.*

This manual covers the following areas:

- Get started — An overview of the official Neo4j Java 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 Java developers building a Neo4j client application.

Get started

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

About the official Java 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 use the Java driver, it is recommended employing a dependency manager, such as Maven or Gradle. To find the latest version of the driver, visit the Maven Central Repository.

Dependencies

- org.reactivestreams:reactive-streams
- org.apache.logging.log4j:log4j (optional)

The driver is dependent on the Reactive Streams API, thus maintaining JDK 8 compatibility. To make optimal use of the reactive APIs, we suggest an additional framework like Project Reactor or RxJava2. Both implement the Reactive Streams API and provide an exhaustive set of operators.

Example 1. Installation via Maven

When using Maven, add the following block to the *pom.xml* file. The driver version can either be declared as a property (as in the first example) or as an explicit version number (as in the second).

```
<dependencies>
    <dependency>
        <groupId>org.neo4j.driver</groupId>
        <artifactId>neo4j-java-driver</artifactId>
        <version>$JAVA_DRIVER_VERSION</version>
    </dependency>
</dependencies>
```

In the following example, driver version 4.3.3 is added.

```
<dependencies>
    <dependency>
        <groupId>org.neo4j.driver</groupId>
        <artifactId>neo4j-java-driver</artifactId>
        <version>4.3.3</version>
    </dependency>
</dependencies>
```

Example 2. Installation via Gradle

For Gradle, a compile line will be required. Again, this can use a property or an explicit version number.

```
compile 'org.neo4j.driver:neo4j-java-driver:$JAVA_DRIVER_VERSION'
```

In the following example, a driver version 4.3.3 is added.

```
compile 'org.neo4j.driver:neo4j-java-driver:4.3.3'
```

The release notes for this driver are available here.

A "Hello World" example

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

Example 4. Hello World

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Result;
import org.neo4j.driver.Session;
import org.neo4j.driver.Transaction;
import org.neo4j.driver.TransactionWork;

import static org.neo4j.driver.Values.parameters;
```

```
public class HelloWorldExample implements AutoCloseable
{
    private final Driver driver;

    public HelloWorldExample( String uri, String user, String password )
    {
        driver = GraphDatabase.driver( uri, AuthTokens.basic( user, password ) );
    }

    @Override
    public void close() throws Exception
    {
        driver.close();
    }

    public void printGreeting( final String message )
    {
        try ( Session session = driver.session() )
        {
            String greeting = session.writeTransaction( new TransactionWork<String>()
            {
                @Override
                public String execute( Transaction tx )
                {
                    Result result = tx.run( "CREATE (a:Greeting) " +
                                            "SET a.message = $message " +
                                            "RETURN a.message + ', from node ' + id(a)",
                            parameters( "message", message ) );
                    return result.single().get( 0 ).asString();
                }
            } );
            System.out.println( greeting );
        }
    }

    public static void main( String... args ) throws Exception
    {
        try ( HelloWorldExample greeter = new HelloWorldExample( "bolt://localhost:7687", "neo4j",
"password" ) )
        {
            greeter.printGreeting( "hello, world" );
        }
    }
}
```

Driver API docs

For a comprehensive listing of all driver functionality, refer to the **API documentation**: https://neo4j.com/docs/api/java-driver/4.3/

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

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
```

```
public class DriverLifecycleExample implements AutoCloseable
{
    private final Driver driver;

    public DriverLifecycleExample( String uri, String user, String password )
    {
        driver = GraphDatabase.driver( uri, AuthTokens.basic( user, password ) );
    }

    @Override
    public void close() throws Exception
    {
        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.

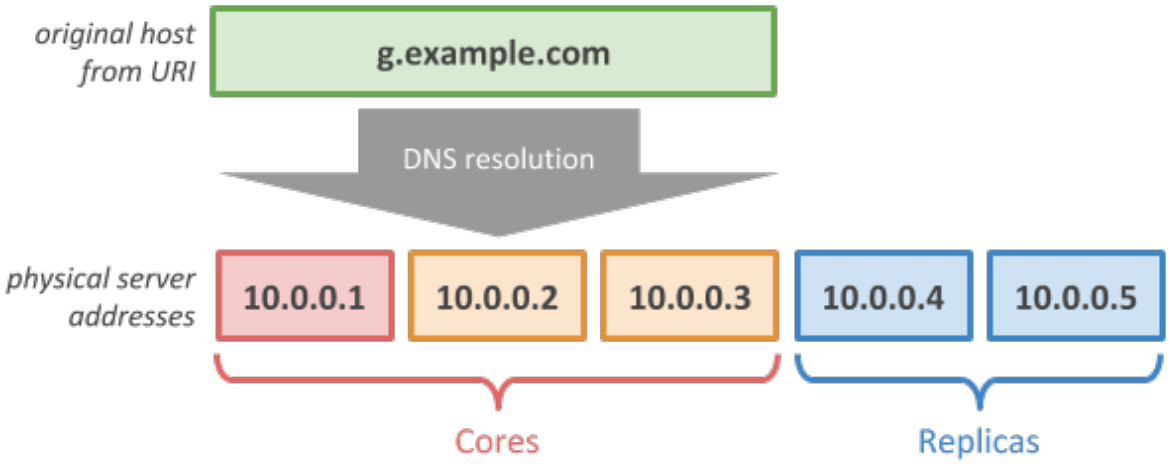

Figure 1. Initial address resolution over DNS

Custom middleware

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

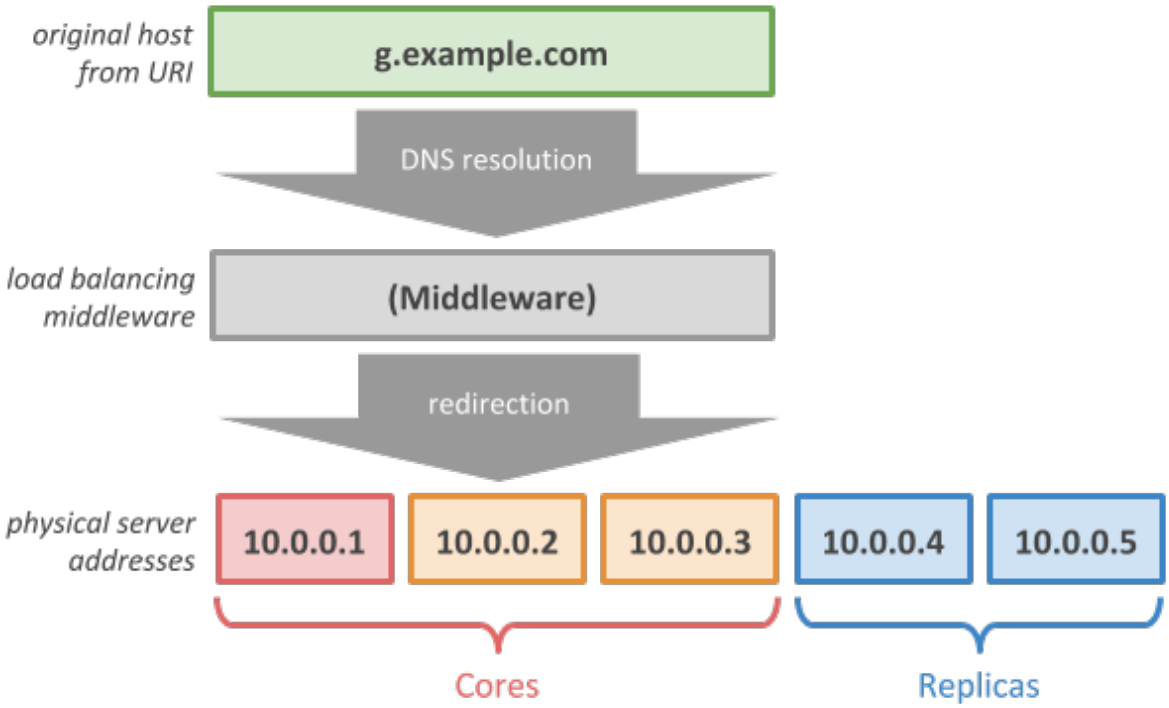

Figure 2. Initial address resolution using custom middleware

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.

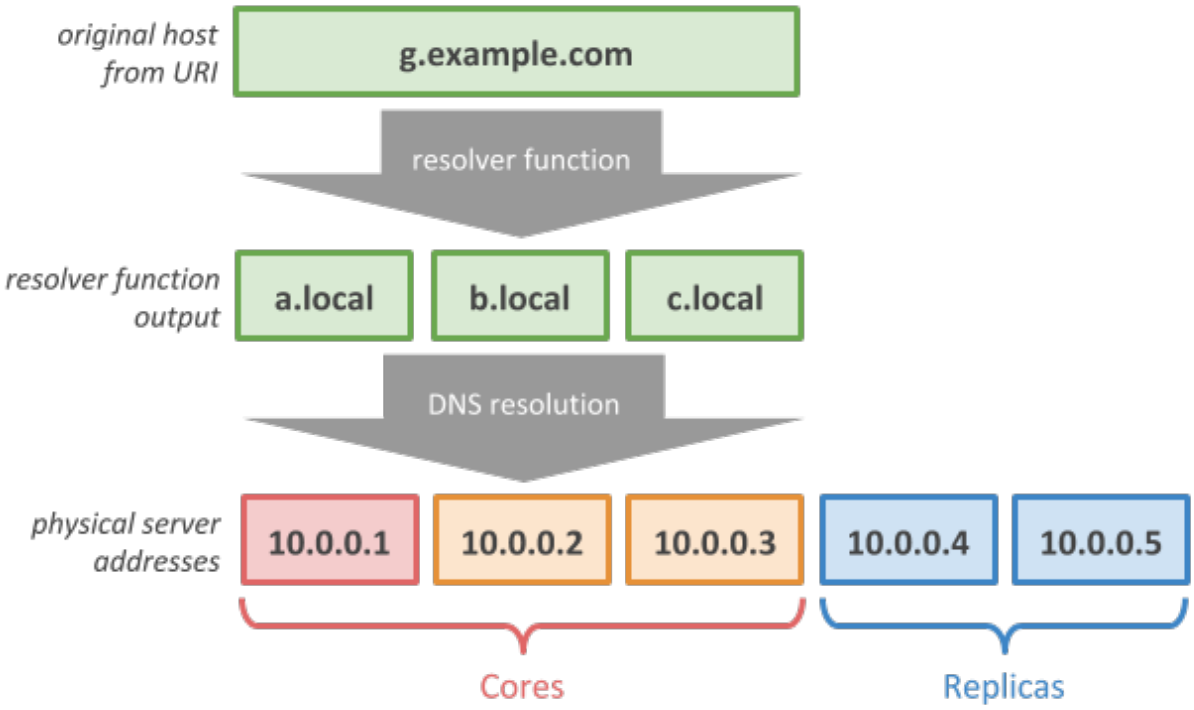

Figure 3. Initial address resolution using resolver function

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

Example 6. Custom Address Resolver

```
private Driver createDriver( String virtualUri, String user, String password, ServerAddress...
addresses )
{
    Config config = Config.builder()
            .withResolver( address -> new HashSet<>( Arrays.asList( addresses ) ) )
            .build();

    return GraphDatabase.driver( virtualUri, AuthTokens.basic( user, password ), config );
}

private void addPerson( String name )
{
    String username = "neo4j";
    String password = "some password";

    try ( Driver driver = createDriver( "neo4j://x.example.com", username, password, ServerAddress.
of( "a.example.com", 7676 ),
            ServerAddress.of( "b.example.com", 8787 ), ServerAddress.of( "c.example.com", 9898 ) ) )
    {
        try ( Session session = driver.session( builder().withDefaultAccessMode( AccessMode.WRITE
).build() ) )
        {
            session.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
        }
    }
}
```

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
`neo4j`	Yes	Unsecured
`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	`GraphDatabase.driver("neo4j+s://graph.example.com:7687", auth)`
	
	If you do not have at least the Java Driver 4.0.1 patch installed, you will need this snippet instead:
	
	`String uri = "neo4j://graph.example.com:7687";`
	`Config config = Config.builder()`
	`.withEncryption()`
	`.build();`
	`Driver driver = GraphDatabase.driver(uri, auth, config);`

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	`GraphDatabase.driver("neo4j://graph.example.com:7687", auth)`
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	`GraphDatabase.driver("neo4j+s://graph.example.com:7687", auth)`
	
	If you do not have at least the Java Driver 4.0.1 patch installed, you will need this snippet instead:
	
	`String uri = "neo4j://graph.example.com:7687";`
	`Config config = Config.builder()`
	`.withEncryption()`
	`.build();`
	`Driver driver = GraphDatabase.driver(uri, auth, config);`

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

Product	Neo4j 4.x

Security	Secured with self-signed certificate

Code snippet	`GraphDatabase.driver("neo4j+ssc://graph.example.com:7687", auth)`
	
	If you do not have at least the Java Driver 4.0.1 patch installed, you will need this snippet instead:
	
	`String uri = "neo4j://graph.example.com:7687";`
	`Config config = Config.builder()`
	`.withEncryption()`
	`.withTrustStrategy( trustAllCertificates() )`
	`.build();`
	`Driver driver = GraphDatabase.driver(uri, auth, config);`

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

Product	Neo4j 3.x
Security	Secured with full certificate

Code snippet	`GraphDatabase.driver("neo4j+s://graph.example.com:7687", auth)`
	
	If you do not have at least the Java Driver 4.0.1 patch installed, you will need this snippet instead:
	
	`String uri = "neo4j://graph.example.com:7687";`
	`Config config = Config.builder()`
	`.withEncryption()`
	`.build();`
	`Driver driver = GraphDatabase.driver(uri, auth, config);`

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

Product	Neo4j 3.x
Security	Secured with self-signed certificate

Code snippet	`GraphDatabase.driver("neo4j+ssc://graph.example.com:7687", auth)`
	
	If you do not have at least the Java Driver 4.0.1 patch installed, you will need this snippet instead:
	
	`String uri = "neo4j://graph.example.com:7687";`
	`Config config = Config.builder()`
	`.withEncryption()`
	`.withTrustStrategy( trustAllCertificates() )`
	`.build();`
	`Driver driver = GraphDatabase.driver(uri, auth, config);`

Comments	This is the default for Neo4j 3.x series

Table 10. Neo4j 3.x, unsecured

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

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

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Result;
```

```
public BasicAuthExample( String uri, String user, String password )
{
    driver = GraphDatabase.driver( uri, AuthTokens.basic( 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

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
```

```
public KerberosAuthExample( String uri, String ticket )
{
    driver = GraphDatabase.driver( uri, AuthTokens.kerberos( 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

```
import java.util.Map;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
```

```
public CustomAuthExample( String uri, String principal, String credentials, String realm, String
scheme,
        Map<String,Object> parameters )
{
    driver = GraphDatabase.driver( uri, AuthTokens.custom( principal, credentials, realm, scheme,
parameters ) );
}
```

Configuration

`ConnectionAcquisitionTimeout`

The maximum amount of time 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 seconds

Example 11. Configure connection pool

```
import java.util.concurrent.TimeUnit;

import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Config;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Result;
```

```
public ConfigConnectionPoolExample( String uri, String user, String password )
{
    Config config = Config.builder()
            .withMaxConnectionLifetime( 30, TimeUnit.MINUTES )
            .withMaxConnectionPoolSize( 50 )
            .withConnectionAcquisitionTimeout( 2, TimeUnit.MINUTES )
            .build();

    driver = GraphDatabase.driver( uri, AuthTokens.basic( user, password ), config );
}
```

`ConnectionTimeout`

The maximum amount of time 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 seconds

Example 12. Configure connection timeout

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Config;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;

import static java.util.concurrent.TimeUnit.SECONDS;
```

```
public ConfigConnectionTimeoutExample( String uri, String user, String password )
{
    Config config = Config.builder()
            .withConnectionTimeout( 15, SECONDS )
            .build();

    driver = GraphDatabase.driver( uri, AuthTokens.basic( user, password ), config );
}
```

`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

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Config;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
```

```
public ConfigUnencryptedExample( String uri, String user, String password )
{
    Config config = Config.builder()
            .withoutEncryption()
            .build();

    driver = GraphDatabase.driver( uri, AuthTokens.basic( user, password ), config );
}
```

`MaxConnectionLifetime`

The maximum duration 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: 1 hour (3600 seconds)

`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 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 seconds

Example 14. Configure maximum retry time

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Config;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;

import static java.util.concurrent.TimeUnit.SECONDS;
```

```
public ConfigMaxRetryTimeExample( String uri, String user, String password )
{
    Config config = Config.builder()
            .withMaxTransactionRetryTime( 15, SECONDS )
            .build();

    driver = GraphDatabase.driver( uri, AuthTokens.basic( user, password ), config );
}
```

`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

```
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Config;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
```

```
public ConfigTrustExample( String uri, String user, String password )
{
    Config config = Config.builder()
            .withTrustStrategy( Config.TrustStrategy.trustSystemCertificates() )
            .build();

    driver = GraphDatabase.driver( uri, AuthTokens.basic( user, password ), config );
}
```

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:

```
ConfigBuilder.withLogging(Logging.console(Level.DEBUG))
```

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:

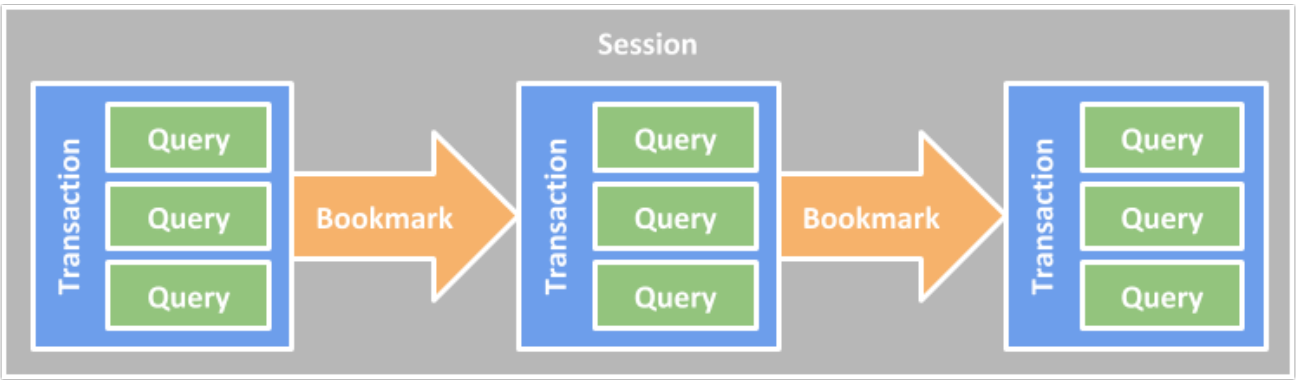

Figure 4. Sessions, queries and transactions

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.

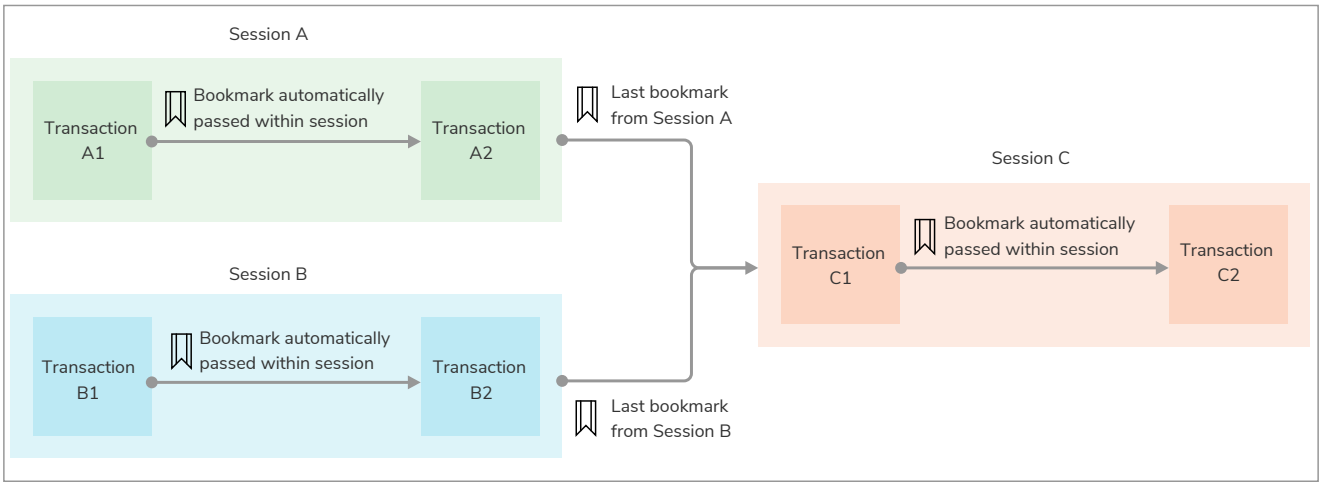

Figure 5. Passing bookmarks

Example 16. Pass bookmarks

```
import java.util.ArrayList;
import java.util.List;

import org.neo4j.driver.AccessMode;
import org.neo4j.driver.Record;
import org.neo4j.driver.Session;
import org.neo4j.driver.Result;
import org.neo4j.driver.Transaction;
import org.neo4j.driver.Bookmark;

import static org.neo4j.driver.Values.parameters;
import static org.neo4j.driver.SessionConfig.builder;
```

```
// Create a company node
private Result addCompany(final Transaction tx, final String name )
{
    return tx.run( "CREATE (:Company {name: $name})", parameters( "name", name ) );
}

// Create a person node
private Result addPerson(final Transaction tx, final String name )
{
    return tx.run( "CREATE (:Person {name: $name})", parameters( "name", name ) );
}

// Create an employment relationship to a pre-existing company node.
// This relies on the person first having been created.
private Result employ(final Transaction tx, final String person, final String company )
{
    return tx.run( "MATCH (person:Person {name: $person_name}) " +
                   "MATCH (company:Company {name: $company_name}) " +
                   "CREATE (person)-[:WORKS_FOR]->(company)",
            parameters( "person_name", person, "company_name", company ) );
}

// Create a friendship between two people.
private Result makeFriends(final Transaction tx, final String person1, final String person2 )
{
    return tx.run( "MATCH (a:Person {name: $person_1}) " +
                   "MATCH (b:Person {name: $person_2}) " +
                   "MERGE (a)-[:KNOWS]->(b)",
            parameters( "person_1", person1, "person_2", person2 ) );
}

// Match and display all friendships.
private Result printFriends(final Transaction tx )
{
    Result result = tx.run( "MATCH (a)-[:KNOWS]->(b) RETURN a.name, b.name" );
    while ( result.hasNext() )
    {
        Record record = result.next();
        System.out.println( String.format( "%s knows %s", record.get( "a.name" ).asString(), record
.get( "b.name" ).toString() ) );
    }
    return result;
}

public void addEmployAndMakeFriends()
{
    // To collect the session bookmarks
    List<Bookmark> savedBookmarks = new ArrayList<>();

    // Create the first person and employment relationship.
    try ( Session session1 = driver.session( builder().withDefaultAccessMode( AccessMode.WRITE
).build() ) )
    {
        session1.writeTransaction( tx -> addCompany( tx, "Wayne Enterprises" ) );
        session1.writeTransaction( tx -> addPerson( tx, "Alice" ) );
        session1.writeTransaction( tx -> employ( tx, "Alice", "Wayne Enterprises" ) );

        savedBookmarks.add( session1.lastBookmark() );
```

```
    }

    // Create the second person and employment relationship.
    try ( Session session2 = driver.session( builder().withDefaultAccessMode( AccessMode.WRITE
).build() ) )
    {
        session2.writeTransaction( tx -> addCompany( tx, "LexCorp" ) );
        session2.writeTransaction( tx -> addPerson( tx, "Bob" ) );
        session2.writeTransaction( tx -> employ( tx, "Bob", "LexCorp" ) );

        savedBookmarks.add( session2.lastBookmark() );
    }

    // Create a friendship between the two people created above.
    try ( Session session3 = driver.session( builder().withDefaultAccessMode( AccessMode.WRITE
).withBookmarks( savedBookmarks ).build() ) )
    {
        session3.writeTransaction( tx -> makeFriends( tx, "Alice", "Bob" ) );

        session3.readTransaction( this::printFriends );
    }
}
```

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. Import read-write transaction

```
import org.neo4j.driver.Session;
import org.neo4j.driver.Result;
import org.neo4j.driver.Transaction;
import org.neo4j.driver.TransactionWork;

import static org.neo4j.driver.Values.parameters;
```

Example 18. Read-write transaction

```
public long addPerson( final String name )
{
    try ( Session session = driver.session() )
    {
        session.writeTransaction( new TransactionWork<Void>()
        {
            @Override
            public Void execute( Transaction tx )
            {
                return createPersonNode( tx, name );
            }
        } );
        return session.readTransaction( new TransactionWork<Long>()
        {
            @Override
            public Long execute( Transaction tx )
            {
                return matchPersonNode( tx, name );
            }
        } );
    }
}

private static Void createPersonNode( Transaction tx, String name )
{
    tx.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
    return null;
}

private static long matchPersonNode( Transaction tx, String name )
{
    Result result = tx.run( "MATCH (a:Person {name: $name}) RETURN id(a)", parameters( "name", name )
);
    return result.single().get( 0 ).asLong();
}
```

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 19. Database selection on session creation

Import database selection on session creation:

```
import org.neo4j.driver.AccessMode;
import org.neo4j.driver.Session;
import org.neo4j.driver.SessionConfig;
```

Database selection on session creation:

```
try ( Session session = driver.session( SessionConfig.forDatabase( "examples" ) ) )
{
    session.run( "CREATE (a:Greeting {message: 'Hello, Example-Database'}) RETURN a" ).consume();
}

SessionConfig sessionConfig = SessionConfig.builder()
        .withDatabase( "examples" )
        .withDefaultAccessMode( AccessMode.READ )
        .build();
try ( Session session = driver.session( sessionConfig ) )
{
    String msg = session.run( "MATCH (a:Greeting) RETURN a.message as msg" ).single().get( "msg"
).asString();
    System.out.println(msg);
}
```

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
`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 → 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 20. Map Neo4j types to Java types

Neo4j Cypher Type	Java Type
`null`	`null`
`List`	`List<Object>`
`Map`	`Map<String, Object>`
`Boolean`	`boolean`
`Integer`	`long`
`Float`	`double`
`String`	`String`
`ByteArray`	`byte[]`
`Date`	`LocalDate`
`Time`	`OffsetTime`
`LocalTime`	`LocalTime`
`DateTime`	`ZonedDateTime`
`LocalDateTime`	`LocalDateTime`
`Duration`	**`IsoDuration`***
`Point`	**`Point`**
`Node`	**`Node`**
`Relationship`	**`Relationship`**
`Path`	**`Path`**

* A `Duration` or `Period` passed as a parameter will always be implicitly converted to `IsoDuration`.

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.

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.

```
try (Session session = driver.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.

```
public void addPerson( final String name )
{
    try ( Session session = driver.session() )
    {
        session.writeTransaction( tx -> {
            tx.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
            return 1;
        } );
    }
}
```

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 replayed on failure. Therefore any error scenarios must be handled by the client application itself.

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.

Unlike other kinds of Cypher Query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Therefore, the only way to execute `PERIODIC COMMIT` from a driver is to use auto-commit transactions.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public void addPerson( String name )
{
    try ( Session session = driver.session() )
    {
        session.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
    }
}
```

Consuming results

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

```
public List<String> getPeople()
{
    try ( Session session = driver.session() )
    {
        return session.readTransaction( tx -> {
            List<String> names = new ArrayList<>();
            Result result = tx.run( "MATCH (a:Person) RETURN a.name ORDER BY a.name" );
            while ( result.hasNext() )
            {
                names.add( result.next().get( 0 ).asString() );
            }
            return names;
        } );
    }
}
```

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.

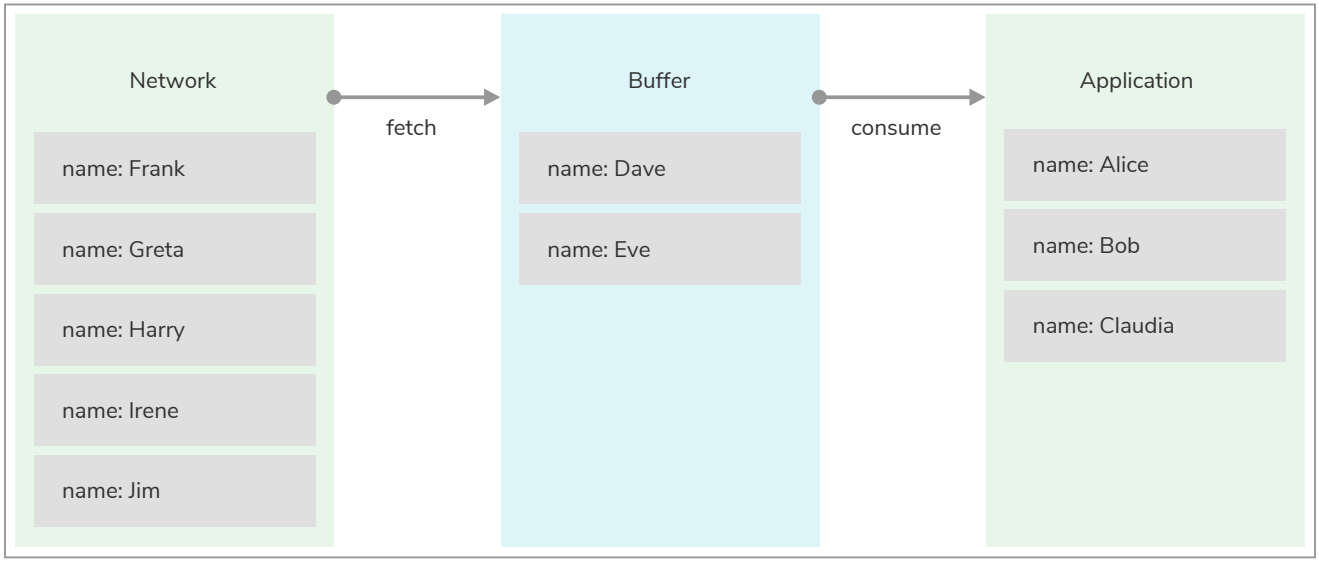

Figure 6. Result buffer

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:

```
public int addEmployees( final String companyName )
{
    try ( Session session = driver.session() )
    {
        int employees = 0;
        List<Record> persons = session.readTransaction( new TransactionWork<List<Record>>()
        {
            @Override
            public List<Record> execute( Transaction tx )
            {
                return matchPersonNodes( tx );
            }
        } );
        for ( final Record person : persons )
        {
            employees += session.writeTransaction( new TransactionWork<Integer>()
            {
                @Override
                public Integer execute( Transaction tx )
                {
                    tx.run( "MATCH (emp:Person {name: $person_name}) " +
                            "MERGE (com:Company {name: $company_name}) " +
                            "MERGE (emp)-[:WORKS_FOR]->(com)",
                            parameters( "person_name", person.get( "name" ).asString(), "company_name",
                                    companyName ) );
                    return 1;
                }
            } );
        }
        return employees;
    }
}

private static List<Record> matchPersonNodes( Transaction tx )
{
    return tx.run( "MATCH (a:Person) RETURN a.name AS name" ).list();
}
```

Asynchronous sessions

Asynchronous sessions provide an API wherein function calls typically return available objects such as futures. This allows client applications to work within asynchronous frameworks and take advantage of cooperative multitasking.

Lifecycle

Session lifetime begins with session construction. A session then exists until it is closed, which is typically set to occur after its contained query results have been consumed.

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 ***the recommended form 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.

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 in a single instance environment, this routing has no impact but it does give you the flexibility should you choose to later adopt a clustered environment.

Before writing a transaction function it is important to ensure that any side-effects carried out by a transaction function should be 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.

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. Retry logic can be configured when creating a session.

```
public CompletionStage<ResultSummary> printAllProducts()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    AsyncSession session = driver.asyncSession();

    return session.readTransactionAsync( tx ->
            tx.runAsync( query, parameters )
                    .thenCompose( cursor -> cursor.forEachAsync( record ->
                            // asynchronously print every record
                            System.out.println( record.get( 0 ).asString() ) ) )
    );
}
```

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 replayed on failure. Therefore any error scenarios must be handled by the client application itself.

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.

Unlike other kinds of Cypher Query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Therefore, the only way to execute them from a driver is to use auto-commit transactions.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public CompletionStage<List<String>> readProductTitles()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    AsyncSession session = driver.asyncSession();

    return session.runAsync( query, parameters )
            .thenCompose( cursor -> cursor.listAsync( record -> record.get( 0 ).asString() ) )
            .exceptionally( error ->
            {
                // query execution failed, print error and fallback to empty list of titles
                error.printStackTrace();
                return Collections.emptyList();
            } )
            .thenCompose( titles -> session.closeAsync().thenApply( ignore -> titles ) );
}
```

Consuming results

The asynchronous session API provides language-idiomatic methods to aid integration with asynchronous applications and frameworks.

```
public CompletionStage<List<String>> getPeople()
{

    String query = "MATCH (a:Person) RETURN a.name ORDER BY a.name";
    AsyncSession session = driver.asyncSession();
    return session.readTransactionAsync( tx ->
            tx.runAsync( query )
                    .thenCompose( cursor -> cursor.listAsync( record ->
                            record.get( 0 ).asString() ) )
    );
}
```

Reactive Sessions

Starting with **Neo4j 4.0**, the reactive processing of queries is supported. This can be achieved through reactive sessions. Reactive sessions allow for dynamic management of the data that is being exchanged between the driver and the server.

Typical of reactive programming, consumers control the rate at which they consume records from queries and the driver in turn manages the rate at which records are requested from the server. Flow control is supported throughout the entire Neo4j stack, meaning that the query engine responds correctly to the flow control signals. This results in far more efficient resource handling and ensures that the receiving side is not forced to buffer arbitrary amounts of data.

For more information about reactive stream, please see the following:

- The Reactive Manifesto
- Reactive Streams for JVM and JavaScript
- Project Reactor reference documentation
- Introduction to Reactive Extensions to .NET

Reactive sessions will typically be used in a client application that is already oriented towards the reactive style; it is expected that a reactive dependency or framework is in place.

Refer to Get started for more information on recommended dependencies.

Lifecycle

Session lifetime begins with session construction. A session then exists until it is closed, which is typically set to occur after its contained query results have been consumed.

Transaction functions

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.

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 in a single instance environment, this routing has no impact but it does give you the flexibility should you choose to later adopt a clustered environment.

Before writing a transaction function it is important to ensure that any side-effects carried out by a transaction function should be 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.

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. Retry logic can be configured when creating a session.

```
public Flux<ResultSummary> printAllProducts()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> session.readTransaction( tx -> {
                RxResult result = tx.run( query, parameters );
                return Flux.from( result.records() )
                        .doOnNext( record -> System.out.println( record.get( 0 ).asString() ) ).then(
Mono.from( result.consume() ) );
            }
        ), RxSession::close );
}
```

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.

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 replayed on failure. Therefore any error scenarios will need to be handled by the client application itself.

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.

The only way to execute `PERIODIC COMMIT` Cypher Queries is to auto-commit the transaction.

Unlike other kinds of Cypher query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public Flux<String> readProductTitles()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> Flux.from( session.run( query, parameters ).records() ).map( record -> record.get(
0 ).asString() ),
            RxSession::close );
}
```

Consuming results

To consume data from a query in a reactive session, a subscriber is required to handle the results that are being returned by the publisher.

Each transaction corresponds to a data flow which supplies the data from the server. Result processing begins when records are pulled from this flow. Only one subscriber may pull data from a given flow.

```
public Flux<String> getPeople()
{
    String query = "MATCH (a:Person) RETURN a.name ORDER BY a.name";

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> session.readTransaction( tx -> {
                        RxResult result = tx.run( query );
                        return Flux.from( result.records() )
                                .map( record -> record.get( 0 ).asString() );
                    }
            ), RxSession::close );
}
```

Cancellation

As per the reactive stream specification, a reactive data flow can be cancelled part way through. This

prematurely commits or rolls back the transaction and stops the query engine from producing any more records.

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. 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. This `FetchSize` applies to simple sessions and async-sessions whereas reactive sessions can be controlled directly using the request method of the subscription.

Default: 1000 records

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.

```
try (Session session = driver.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.

```
public void addPerson( final String name )
{
    try ( Session session = driver.session() )
    {
        session.writeTransaction( tx -> {
            tx.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
            return 1;
        } );
    }
}
```

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 replayed on failure. Therefore any error scenarios must be handled by the client application itself.

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.

Unlike other kinds of Cypher Query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Therefore, the only way to execute `PERIODIC COMMIT` from a driver is to use auto-commit transactions.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public void addPerson( String name )
{
    try ( Session session = driver.session() )
    {
        session.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
    }
}
```

Consuming results

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

```
public List<String> getPeople()
{
    try ( Session session = driver.session() )
    {
        return session.readTransaction( tx -> {
            List<String> names = new ArrayList<>();
            Result result = tx.run( "MATCH (a:Person) RETURN a.name ORDER BY a.name" );
            while ( result.hasNext() )
            {
                names.add( result.next().get( 0 ).asString() );
            }
            return names;
        } );
    }
}
```

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.

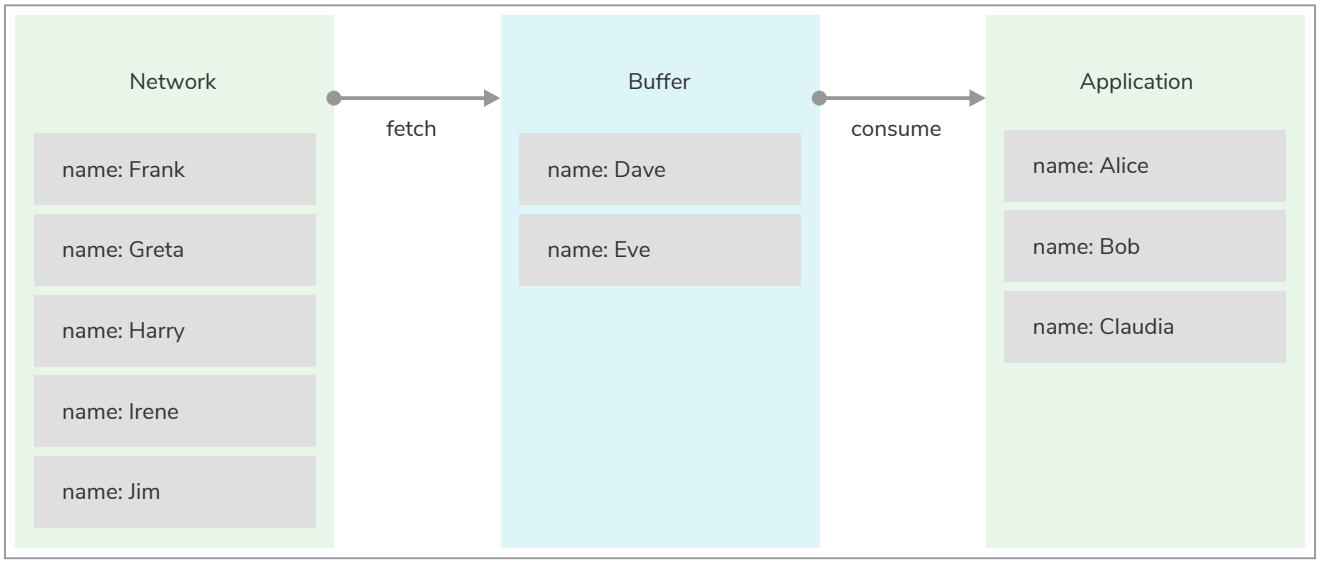

Figure 7. Result buffer

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:

```
public int addEmployees( final String companyName )
{
    try ( Session session = driver.session() )
    {
        int employees = 0;
        List<Record> persons = session.readTransaction( new TransactionWork<List<Record>>()
        {
            @Override
            public List<Record> execute( Transaction tx )
            {
                return matchPersonNodes( tx );
            }
        } );
        for ( final Record person : persons )
        {
            employees += session.writeTransaction( new TransactionWork<Integer>()
            {
                @Override
                public Integer execute( Transaction tx )
                {
                    tx.run( "MATCH (emp:Person {name: $person_name}) " +
                            "MERGE (com:Company {name: $company_name}) " +
                            "MERGE (emp)-[:WORKS_FOR]->(com)",
                            parameters( "person_name", person.get( "name" ).asString(), "company_name",
                                    companyName ) );
                    return 1;
                }
            } );
        }
        return employees;
    }
}

private static List<Record> matchPersonNodes( Transaction tx )
{
    return tx.run( "MATCH (a:Person) RETURN a.name AS name" ).list();
}
```

Asynchronous sessions

Asynchronous sessions provide an API wherein function calls typically return available objects such as futures. This allows client applications to work within asynchronous frameworks and take advantage of cooperative multitasking.

Lifecycle

Session lifetime begins with session construction. A session then exists until it is closed, which is typically set to occur after its contained query results have been consumed.

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 ***the recommended form 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.

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 in a single instance environment, this routing has no impact but it does give you the flexibility should you choose to later adopt a clustered environment.

Before writing a transaction function it is important to ensure that any side-effects carried out by a transaction function should be 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.

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. Retry logic can be configured when creating a session.

```
public CompletionStage<ResultSummary> printAllProducts()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    AsyncSession session = driver.asyncSession();

    return session.readTransactionAsync( tx ->
            tx.runAsync( query, parameters )
                    .thenCompose( cursor -> cursor.forEachAsync( record ->
                            // asynchronously print every record
                            System.out.println( record.get( 0 ).asString() ) ) )
    );
}
```

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 replayed on failure. Therefore any error scenarios must be handled by the client application itself.

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.

Unlike other kinds of Cypher Query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Therefore, the only way to execute them from a driver is to use auto-commit transactions.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public CompletionStage<List<String>> readProductTitles()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    AsyncSession session = driver.asyncSession();

    return session.runAsync( query, parameters )
            .thenCompose( cursor -> cursor.listAsync( record -> record.get( 0 ).asString() ) )
            .exceptionally( error ->
            {
                // query execution failed, print error and fallback to empty list of titles
                error.printStackTrace();
                return Collections.emptyList();
            } )
            .thenCompose( titles -> session.closeAsync().thenApply( ignore -> titles ) );
}
```

Consuming results

The asynchronous session API provides language-idiomatic methods to aid integration with asynchronous applications and frameworks.

```
public CompletionStage<List<String>> getPeople()
{

    String query = "MATCH (a:Person) RETURN a.name ORDER BY a.name";
    AsyncSession session = driver.asyncSession();
    return session.readTransactionAsync( tx ->
            tx.runAsync( query )
                    .thenCompose( cursor -> cursor.listAsync( record ->
                            record.get( 0 ).asString() ) )
    );
}
```

Reactive Sessions

Starting with **Neo4j 4.0**, the reactive processing of queries is supported. This can be achieved through reactive sessions. Reactive sessions allow for dynamic management of the data that is being exchanged between the driver and the server.

Typical of reactive programming, consumers control the rate at which they consume records from queries and the driver in turn manages the rate at which records are requested from the server. Flow control is supported throughout the entire Neo4j stack, meaning that the query engine responds correctly to the flow control signals. This results in far more efficient resource handling and ensures that the receiving side is not forced to buffer arbitrary amounts of data.

For more information about reactive stream, please see the following:

- The Reactive Manifesto
- Reactive Streams for JVM and JavaScript
- Project Reactor reference documentation
- Introduction to Reactive Extensions to .NET

Reactive sessions will typically be used in a client application that is already oriented towards the reactive style; it is expected that a reactive dependency or framework is in place.

Refer to Get started for more information on recommended dependencies.

Lifecycle

Session lifetime begins with session construction. A session then exists until it is closed, which is typically set to occur after its contained query results have been consumed.

Transaction functions

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.

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 in a single instance environment, this routing has no impact but it does give you the flexibility should you choose to later adopt a clustered environment.

Before writing a transaction function it is important to ensure that any side-effects carried out by a transaction function should be 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.

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. Retry logic can be configured when creating a session.

```
public Flux<ResultSummary> printAllProducts()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> session.readTransaction( tx -> {
                RxResult result = tx.run( query, parameters );
                return Flux.from( result.records() )
                        .doOnNext( record -> System.out.println( record.get( 0 ).asString() ) ).then(
Mono.from( result.consume() ) );
            }
        ), RxSession::close );
}
```

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.

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 replayed on failure. Therefore any error scenarios will need to be handled by the client application itself.

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.

The only way to execute `PERIODIC COMMIT` Cypher Queries is to auto-commit the transaction.

Unlike other kinds of Cypher query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public Flux<String> readProductTitles()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> Flux.from( session.run( query, parameters ).records() ).map( record -> record.get(
0 ).asString() ),
            RxSession::close );
}
```

Consuming results

To consume data from a query in a reactive session, a subscriber is required to handle the results that are being returned by the publisher.

Each transaction corresponds to a data flow which supplies the data from the server. Result processing begins when records are pulled from this flow. Only one subscriber may pull data from a given flow.

```
public Flux<String> getPeople()
{
    String query = "MATCH (a:Person) RETURN a.name ORDER BY a.name";

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> session.readTransaction( tx -> {
                        RxResult result = tx.run( query );
                        return Flux.from( result.records() )
                                .map( record -> record.get( 0 ).asString() );
                }
            ), RxSession::close );
}
```

Cancellation

As per the reactive stream specification, a reactive data flow can be cancelled part way through. This prematurely commits or rolls back the transaction and stops the query engine from producing any more

records.

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. 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. This `FetchSize` applies to simple sessions and async-sessions whereas reactive sessions can be controlled directly using the request method of the subscription.

Default: 1000 records

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.

```
try (Session session = driver.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.

```
public void addPerson( final String name )
{
    try ( Session session = driver.session() )
    {
        session.writeTransaction( tx -> {
            tx.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
            return 1;
        } );
    }
}
```

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 replayed on failure. Therefore any error scenarios must be handled by the client application itself.

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.

Unlike other kinds of Cypher Query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Therefore, the only way to execute `PERIODIC COMMIT` from a driver is to use auto-commit transactions.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public void addPerson( String name )
{
    try ( Session session = driver.session() )
    {
        session.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
    }
}
```

Consuming results

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

```
public List<String> getPeople()
{
    try ( Session session = driver.session() )
    {
        return session.readTransaction( tx -> {
            List<String> names = new ArrayList<>();
            Result result = tx.run( "MATCH (a:Person) RETURN a.name ORDER BY a.name" );
            while ( result.hasNext() )
            {
                names.add( result.next().get( 0 ).asString() );
            }
            return names;
        } );
    }
}
```

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.

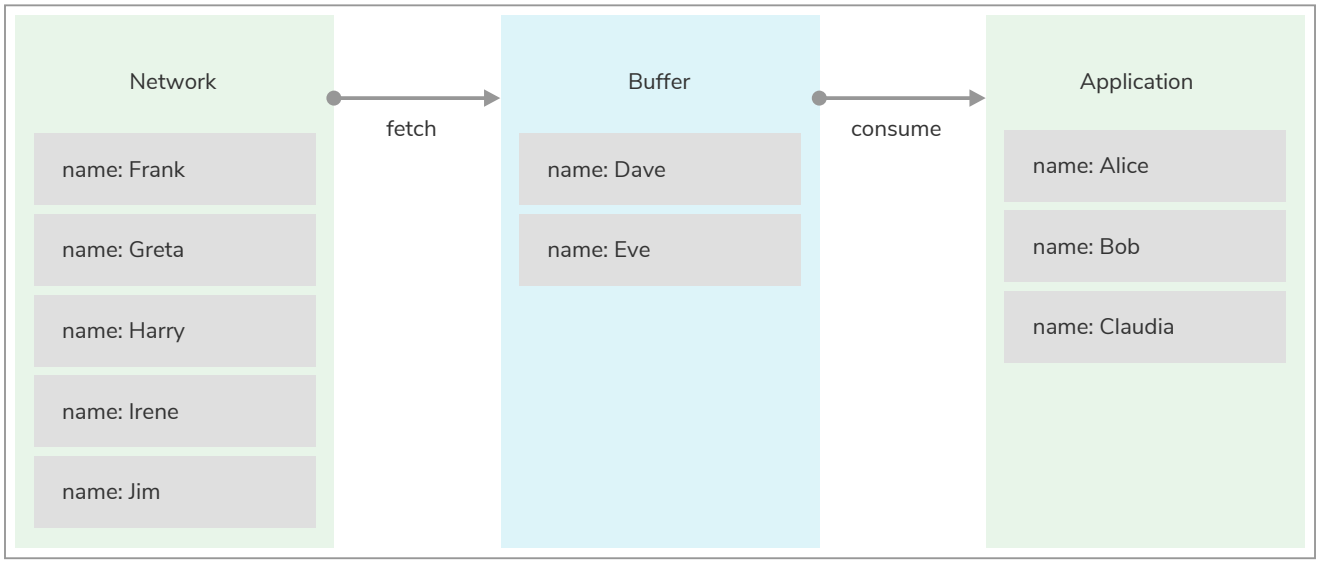

Figure 8. Result buffer

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:

```
public int addEmployees( final String companyName )
{
    try ( Session session = driver.session() )
    {
        int employees = 0;
        List<Record> persons = session.readTransaction( new TransactionWork<List<Record>>()
        {
            @Override
            public List<Record> execute( Transaction tx )
            {
                return matchPersonNodes( tx );
            }
        } );
        for ( final Record person : persons )
        {
            employees += session.writeTransaction( new TransactionWork<Integer>()
            {
                @Override
                public Integer execute( Transaction tx )
                {
                    tx.run( "MATCH (emp:Person {name: $person_name}) " +
                            "MERGE (com:Company {name: $company_name}) " +
                            "MERGE (emp)-[:WORKS_FOR]->(com)",
                            parameters( "person_name", person.get( "name" ).asString(), "company_name",
                                    companyName ) );
                    return 1;
                }
            } );
        }
        return employees;
    }
}

private static List<Record> matchPersonNodes( Transaction tx )
{
    return tx.run( "MATCH (a:Person) RETURN a.name AS name" ).list();
}
```

Asynchronous sessions

Asynchronous sessions provide an API wherein function calls typically return available objects such as futures. This allows client applications to work within asynchronous frameworks and take advantage of cooperative multitasking.

Lifecycle

Session lifetime begins with session construction. A session then exists until it is closed, which is typically set to occur after its contained query results have been consumed.

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 ***the recommended form 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.

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 in a single instance environment, this routing has no impact but it does give you the flexibility should you choose to later adopt a clustered environment.

Before writing a transaction function it is important to ensure that any side-effects carried out by a transaction function should be 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.

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. Retry logic can be configured when creating a session.

```
public CompletionStage<ResultSummary> printAllProducts()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    AsyncSession session = driver.asyncSession();

    return session.readTransactionAsync( tx ->
            tx.runAsync( query, parameters )
                    .thenCompose( cursor -> cursor.forEachAsync( record ->
                            // asynchronously print every record
                            System.out.println( record.get( 0 ).asString() ) ) )
    );
}
```

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 replayed on failure. Therefore any error scenarios must be handled by the client application itself.

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.

Unlike other kinds of Cypher Query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Therefore, the only way to execute them from a driver is to use auto-commit transactions.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public CompletionStage<List<String>> readProductTitles()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    AsyncSession session = driver.asyncSession();

    return session.runAsync( query, parameters )
            .thenCompose( cursor -> cursor.listAsync( record -> record.get( 0 ).asString() ) )
            .exceptionally( error ->
            {
                // query execution failed, print error and fallback to empty list of titles
                error.printStackTrace();
                return Collections.emptyList();
            } )
            .thenCompose( titles -> session.closeAsync().thenApply( ignore -> titles ) );
}
```

Consuming results

The asynchronous session API provides language-idiomatic methods to aid integration with asynchronous applications and frameworks.

```
public CompletionStage<List<String>> getPeople()
{

    String query = "MATCH (a:Person) RETURN a.name ORDER BY a.name";
    AsyncSession session = driver.asyncSession();
    return session.readTransactionAsync( tx ->
            tx.runAsync( query )
                    .thenCompose( cursor -> cursor.listAsync( record ->
                            record.get( 0 ).asString() ) )
    );
}
```

Reactive Sessions

Starting with **Neo4j 4.0**, the reactive processing of queries is supported. This can be achieved through reactive sessions. Reactive sessions allow for dynamic management of the data that is being exchanged between the driver and the server.

Typical of reactive programming, consumers control the rate at which they consume records from queries and the driver in turn manages the rate at which records are requested from the server. Flow control is supported throughout the entire Neo4j stack, meaning that the query engine responds correctly to the flow control signals. This results in far more efficient resource handling and ensures that the receiving side is not forced to buffer arbitrary amounts of data.

For more information about reactive stream, please see the following:

- The Reactive Manifesto
- Reactive Streams for JVM and JavaScript
- Project Reactor reference documentation
- Introduction to Reactive Extensions to .NET

Reactive sessions will typically be used in a client application that is already oriented towards the reactive style; it is expected that a reactive dependency or framework is in place.

Refer to Get started for more information on recommended dependencies.

Lifecycle

Session lifetime begins with session construction. A session then exists until it is closed, which is typically set to occur after its contained query results have been consumed.

Transaction functions

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.

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 in a single instance environment, this routing has no impact but it does give you the flexibility should you choose to later adopt a clustered environment.

Before writing a transaction function it is important to ensure that any side-effects carried out by a transaction function should be 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.

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. Retry logic can be configured when creating a session.

```
public Flux<ResultSummary> printAllProducts()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> session.readTransaction( tx -> {
                RxResult result = tx.run( query, parameters );
                return Flux.from( result.records() )
                        .doOnNext( record -> System.out.println( record.get( 0 ).asString() ) ).then(
Mono.from( result.consume() ) );
            }
        ), RxSession::close );
}
```

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.

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 replayed on failure. Therefore any error scenarios will need to be handled by the client application itself.

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.

The only way to execute `PERIODIC COMMIT` Cypher Queries is to auto-commit the transaction.

Unlike other kinds of Cypher query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public Flux<String> readProductTitles()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> Flux.from( session.run( query, parameters ).records() ).map( record -> record.get(
0 ).asString() ),
            RxSession::close );
}
```

Consuming results

To consume data from a query in a reactive session, a subscriber is required to handle the results that are being returned by the publisher.

Each transaction corresponds to a data flow which supplies the data from the server. Result processing begins when records are pulled from this flow. Only one subscriber may pull data from a given flow.

```
public Flux<String> getPeople()
{
    String query = "MATCH (a:Person) RETURN a.name ORDER BY a.name";

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> session.readTransaction( tx -> {
                        RxResult result = tx.run( query );
                        return Flux.from( result.records() )
                                .map( record -> record.get( 0 ).asString() );
                }
            ), RxSession::close );
}
```

Cancellation

As per the reactive stream specification, a reactive data flow can be cancelled part way through. This prematurely commits or rolls back the transaction and stops the query engine from producing any more

records.

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. 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. This `FetchSize` applies to simple sessions and async-sessions whereas reactive sessions can be controlled directly using the request method of the subscription.

Default: 1000 records

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.

```
try (Session session = driver.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.

```
public void addPerson( final String name )
{
    try ( Session session = driver.session() )
    {
        session.writeTransaction( tx -> {
            tx.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
            return 1;
        } );
    }
}
```

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 replayed on failure. Therefore any error scenarios must be handled by the client application itself.

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.

Unlike other kinds of Cypher Query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Therefore, the only way to execute `PERIODIC COMMIT` from a driver is to use auto-commit transactions.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public void addPerson( String name )
{
    try ( Session session = driver.session() )
    {
        session.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
    }
}
```

Consuming results

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

```
public List<String> getPeople()
{
    try ( Session session = driver.session() )
    {
        return session.readTransaction( tx -> {
            List<String> names = new ArrayList<>();
            Result result = tx.run( "MATCH (a:Person) RETURN a.name ORDER BY a.name" );
            while ( result.hasNext() )
            {
                names.add( result.next().get( 0 ).asString() );
            }
            return names;
        } );
    }
}
```

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.

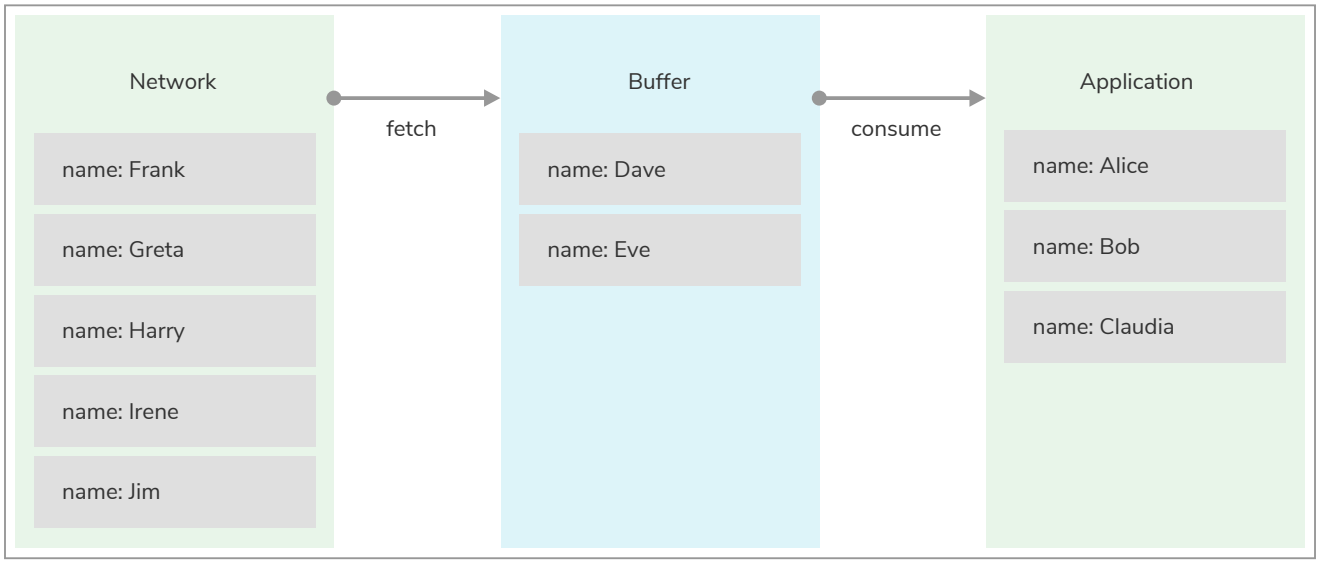

Figure 9. Result buffer

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:

```
public int addEmployees( final String companyName )
{
    try ( Session session = driver.session() )
    {
        int employees = 0;
        List<Record> persons = session.readTransaction( new TransactionWork<List<Record>>()
        {
            @Override
            public List<Record> execute( Transaction tx )
            {
                return matchPersonNodes( tx );
            }
        } );
        for ( final Record person : persons )
        {
            employees += session.writeTransaction( new TransactionWork<Integer>()
            {
                @Override
                public Integer execute( Transaction tx )
                {
                    tx.run( "MATCH (emp:Person {name: $person_name}) " +
                            "MERGE (com:Company {name: $company_name}) " +
                            "MERGE (emp)-[:WORKS_FOR]->(com)",
                            parameters( "person_name", person.get( "name" ).asString(), "company_name",
                                    companyName ) );
                    return 1;
                }
            } );
        }
        return employees;
    }
}

private static List<Record> matchPersonNodes( Transaction tx )
{
    return tx.run( "MATCH (a:Person) RETURN a.name AS name" ).list();
}
```

Asynchronous sessions

Asynchronous sessions provide an API wherein function calls typically return available objects such as futures. This allows client applications to work within asynchronous frameworks and take advantage of cooperative multitasking.

Lifecycle

Session lifetime begins with session construction. A session then exists until it is closed, which is typically set to occur after its contained query results have been consumed.

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 ***the recommended form 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.

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 in a single instance environment, this routing has no impact but it does give you the flexibility should you choose to later adopt a clustered environment.

Before writing a transaction function it is important to ensure that any side-effects carried out by a transaction function should be 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.

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. Retry logic can be configured when creating a session.

```
public CompletionStage<ResultSummary> printAllProducts()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    AsyncSession session = driver.asyncSession();

    return session.readTransactionAsync( tx ->
            tx.runAsync( query, parameters )
                    .thenCompose( cursor -> cursor.forEachAsync( record ->
                            // asynchronously print every record
                            System.out.println( record.get( 0 ).asString() ) ) )
    );
}
```

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 replayed on failure. Therefore any error scenarios must be handled by the client application itself.

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.

Unlike other kinds of Cypher Query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Therefore, the only way to execute them from a driver is to use auto-commit transactions.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public CompletionStage<List<String>> readProductTitles()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    AsyncSession session = driver.asyncSession();

    return session.runAsync( query, parameters )
            .thenCompose( cursor -> cursor.listAsync( record -> record.get( 0 ).asString() ) )
            .exceptionally( error ->
            {
                // query execution failed, print error and fallback to empty list of titles
                error.printStackTrace();
                return Collections.emptyList();
            } )
            .thenCompose( titles -> session.closeAsync().thenApply( ignore -> titles ) );
}
```

Consuming results

The asynchronous session API provides language-idiomatic methods to aid integration with asynchronous applications and frameworks.

```
public CompletionStage<List<String>> getPeople()
{

    String query = "MATCH (a:Person) RETURN a.name ORDER BY a.name";
    AsyncSession session = driver.asyncSession();
    return session.readTransactionAsync( tx ->
            tx.runAsync( query )
                    .thenCompose( cursor -> cursor.listAsync( record ->
                            record.get( 0 ).asString() ) )
    );
}
```

Reactive Sessions

Starting with **Neo4j 4.0**, the reactive processing of queries is supported. This can be achieved through reactive sessions. Reactive sessions allow for dynamic management of the data that is being exchanged between the driver and the server.

Typical of reactive programming, consumers control the rate at which they consume records from queries and the driver in turn manages the rate at which records are requested from the server. Flow control is supported throughout the entire Neo4j stack, meaning that the query engine responds correctly to the flow control signals. This results in far more efficient resource handling and ensures that the receiving side is not forced to buffer arbitrary amounts of data.

For more information about reactive stream, please see the following:

- The Reactive Manifesto
- Reactive Streams for JVM and JavaScript
- Project Reactor reference documentation
- Introduction to Reactive Extensions to .NET

Reactive sessions will typically be used in a client application that is already oriented towards the reactive style; it is expected that a reactive dependency or framework is in place.

Refer to Get started for more information on recommended dependencies.

Lifecycle

Session lifetime begins with session construction. A session then exists until it is closed, which is typically set to occur after its contained query results have been consumed.

Transaction functions

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.

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 in a single instance environment, this routing has no impact but it does give you the flexibility should you choose to later adopt a clustered environment.

Before writing a transaction function it is important to ensure that any side-effects carried out by a transaction function should be 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.

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. Retry logic can be configured when creating a session.

```
public Flux<ResultSummary> printAllProducts()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> session.readTransaction( tx -> {
                RxResult result = tx.run( query, parameters );
                return Flux.from( result.records() )
                        .doOnNext( record -> System.out.println( record.get( 0 ).asString() ) ).then(
Mono.from( result.consume() ) );
            }
        ), RxSession::close );
}
```

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.

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 replayed on failure. Therefore any error scenarios will need to be handled by the client application itself.

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.

The only way to execute `PERIODIC COMMIT` Cypher Queries is to auto-commit the transaction.

Unlike other kinds of Cypher query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public Flux<String> readProductTitles()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> Flux.from( session.run( query, parameters ).records() ).map( record -> record.get(
0 ).asString() ),
            RxSession::close );
}
```

Consuming results

To consume data from a query in a reactive session, a subscriber is required to handle the results that are being returned by the publisher.

Each transaction corresponds to a data flow which supplies the data from the server. Result processing begins when records are pulled from this flow. Only one subscriber may pull data from a given flow.

```
public Flux<String> getPeople()
{
    String query = "MATCH (a:Person) RETURN a.name ORDER BY a.name";

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> session.readTransaction( tx -> {
                        RxResult result = tx.run( query );
                        return Flux.from( result.records() )
                                .map( record -> record.get( 0 ).asString() );
                }
            ), RxSession::close );
}
```

Cancellation

As per the reactive stream specification, a reactive data flow can be cancelled part way through. This prematurely commits or rolls back the transaction and stops the query engine from producing any more

records.

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. 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. This `FetchSize` applies to simple sessions and async-sessions whereas reactive sessions can be controlled directly using the request method of the subscription.

Default: 1000 records

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.

```
try (Session session = driver.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.

```
public void addPerson( final String name )
{
    try ( Session session = driver.session() )
    {
        session.writeTransaction( tx -> {
            tx.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
            return 1;
        } );
    }
}
```

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 replayed on failure. Therefore any error scenarios must be handled by the client application itself.

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.

Unlike other kinds of Cypher Query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Therefore, the only way to execute `PERIODIC COMMIT` from a driver is to use auto-commit transactions.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public void addPerson( String name )
{
    try ( Session session = driver.session() )
    {
        session.run( "CREATE (a:Person {name: $name})", parameters( "name", name ) );
    }
}
```

Consuming results

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

```
public List<String> getPeople()
{
    try ( Session session = driver.session() )
    {
        return session.readTransaction( tx -> {
            List<String> names = new ArrayList<>();
            Result result = tx.run( "MATCH (a:Person) RETURN a.name ORDER BY a.name" );
            while ( result.hasNext() )
            {
                names.add( result.next().get( 0 ).asString() );
            }
            return names;
        } );
    }
}
```

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.

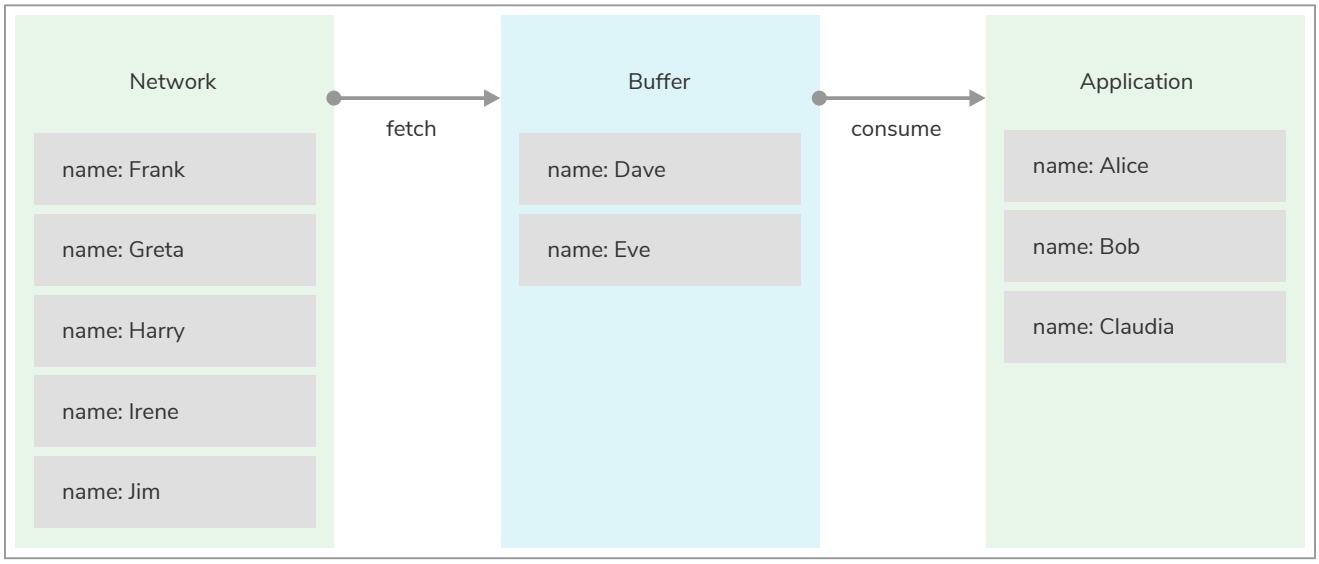

Figure 10. Result buffer

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:

```
public int addEmployees( final String companyName )
{
    try ( Session session = driver.session() )
    {
        int employees = 0;
        List<Record> persons = session.readTransaction( new TransactionWork<List<Record>>()
        {
            @Override
            public List<Record> execute( Transaction tx )
            {
                return matchPersonNodes( tx );
            }
        } );
        for ( final Record person : persons )
        {
            employees += session.writeTransaction( new TransactionWork<Integer>()
            {
                @Override
                public Integer execute( Transaction tx )
                {
                    tx.run( "MATCH (emp:Person {name: $person_name}) " +
                            "MERGE (com:Company {name: $company_name}) " +
                            "MERGE (emp)-[:WORKS_FOR]->(com)",
                            parameters( "person_name", person.get( "name" ).asString(), "company_name",
                                    companyName ) );
                    return 1;
                }
            } );
        }
        return employees;
    }
}

private static List<Record> matchPersonNodes( Transaction tx )
{
    return tx.run( "MATCH (a:Person) RETURN a.name AS name" ).list();
}
```

Asynchronous sessions

Asynchronous sessions provide an API wherein function calls typically return available objects such as futures. This allows client applications to work within asynchronous frameworks and take advantage of cooperative multitasking.

Lifecycle

Session lifetime begins with session construction. A session then exists until it is closed, which is typically set to occur after its contained query results have been consumed.

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 ***the recommended form 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.

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 in a single instance environment, this routing has no impact but it does give you the flexibility should you choose to later adopt a clustered environment.

Before writing a transaction function it is important to ensure that any side-effects carried out by a transaction function should be 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.

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. Retry logic can be configured when creating a session.

```
public CompletionStage<ResultSummary> printAllProducts()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    AsyncSession session = driver.asyncSession();

    return session.readTransactionAsync( tx ->
            tx.runAsync( query, parameters )
                    .thenCompose( cursor -> cursor.forEachAsync( record ->
                            // asynchronously print every record
                            System.out.println( record.get( 0 ).asString() ) ) )
    );
}
```

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 replayed on failure. Therefore any error scenarios must be handled by the client application itself.

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.

Unlike other kinds of Cypher Query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Therefore, the only way to execute them from a driver is to use auto-commit transactions.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public CompletionStage<List<String>> readProductTitles()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    AsyncSession session = driver.asyncSession();

    return session.runAsync( query, parameters )
            .thenCompose( cursor -> cursor.listAsync( record -> record.get( 0 ).asString() ) )
            .exceptionally( error ->
            {
                // query execution failed, print error and fallback to empty list of titles
                error.printStackTrace();
                return Collections.emptyList();
            } )
            .thenCompose( titles -> session.closeAsync().thenApply( ignore -> titles ) );
}
```

Consuming results

The asynchronous session API provides language-idiomatic methods to aid integration with asynchronous applications and frameworks.

```
public CompletionStage<List<String>> getPeople()
{

    String query = "MATCH (a:Person) RETURN a.name ORDER BY a.name";
    AsyncSession session = driver.asyncSession();
    return session.readTransactionAsync( tx ->
            tx.runAsync( query )
                    .thenCompose( cursor -> cursor.listAsync( record ->
                            record.get( 0 ).asString() ) )
    );
}
```

Reactive Sessions

Starting with **Neo4j 4.0**, the reactive processing of queries is supported. This can be achieved through reactive sessions. Reactive sessions allow for dynamic management of the data that is being exchanged between the driver and the server.

Typical of reactive programming, consumers control the rate at which they consume records from queries and the driver in turn manages the rate at which records are requested from the server. Flow control is supported throughout the entire Neo4j stack, meaning that the query engine responds correctly to the flow control signals. This results in far more efficient resource handling and ensures that the receiving side is not forced to buffer arbitrary amounts of data.

For more information about reactive stream, please see the following:

- The Reactive Manifesto
- Reactive Streams for JVM and JavaScript
- Project Reactor reference documentation
- Introduction to Reactive Extensions to .NET

Reactive sessions will typically be used in a client application that is already oriented towards the reactive style; it is expected that a reactive dependency or framework is in place.

Refer to Get started for more information on recommended dependencies.

Lifecycle

Session lifetime begins with session construction. A session then exists until it is closed, which is typically set to occur after its contained query results have been consumed.

Transaction functions

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.

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 in a single instance environment, this routing has no impact but it does give you the flexibility should you choose to later adopt a clustered environment.

Before writing a transaction function it is important to ensure that any side-effects carried out by a transaction function should be 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.

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. Retry logic can be configured when creating a session.

```
public Flux<ResultSummary> printAllProducts()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> session.readTransaction( tx -> {
                RxResult result = tx.run( query, parameters );
                return Flux.from( result.records() )
                        .doOnNext( record -> System.out.println( record.get( 0 ).asString() ) ).then(
Mono.from( result.consume() ) );
            }
        ), RxSession::close );
}
```

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.

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 replayed on failure. Therefore any error scenarios will need to be handled by the client application itself.

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.

The only way to execute `PERIODIC COMMIT` Cypher Queries is to auto-commit the transaction.

Unlike other kinds of Cypher query, `PERIODIC COMMIT` queries do not participate in the causal chain.

Please refer to the Cypher Manual → `PERIODIC COMMIT` query hint.

```
public Flux<String> readProductTitles()
{
    String query = "MATCH (p:Product) WHERE p.id = $id RETURN p.title";
    Map<String,Object> parameters = Collections.singletonMap( "id", 0 );

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> Flux.from( session.run( query, parameters ).records() ).map( record -> record.get(
0 ).asString() ),
            RxSession::close );
}
```

Consuming results

To consume data from a query in a reactive session, a subscriber is required to handle the results that are being returned by the publisher.

Each transaction corresponds to a data flow which supplies the data from the server. Result processing begins when records are pulled from this flow. Only one subscriber may pull data from a given flow.

```
public Flux<String> getPeople()
{
    String query = "MATCH (a:Person) RETURN a.name ORDER BY a.name";

    return Flux.usingWhen( Mono.fromSupplier( driver::rxSession ),
            session -> session.readTransaction( tx -> {
                        RxResult result = tx.run( query );
                        return Flux.from( result.records() )
                                .map( record -> record.get( 0 ).asString() );
                }
            ), RxSession::close );
}
```

Cancellation

As per the reactive stream specification, a reactive data flow can be cancelled part way through. This prematurely commits or rolls back the transaction and stops the query engine from producing any more

records.

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. 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. This `FetchSize` applies to simple sessions and async-sessions whereas reactive sessions can be controlled directly using the request method of the subscription.

Default: 1000 records

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.