Java Reference

Table of Contents

Neo4j v3.5

License: Creative Commons 4.0

The Java Reference for Neo4j v3.5.

The Java Reference contains information on advanced Java-centric usage of Neo4j. Among the topics covered are embedding Neo4j in your own software and writing extensions.

The following topics are:

- Extending Neo4j — How to build unmanaged extensions and procedures.
- Using Neo4j embedded in Java applications — Instructions on embedding Neo4j in an application.
- The traversal framework — A walkthrough of the traversal framework.
- Transaction management — Details on transaction semantics in Neo4j.
- JMX metrics — How to monitor Neo4j with JMX and a reference of available metrics.
- Explicit indexing Deprecated — Explicit indexing operations. Deprecated

> You might want to keep the Neo4j Javadocs (Neo4j Java API Documentation) handy while reading.

Who should read this?

The Java Reference is written for the advanced Java developer who is extending Neo4j's functionality, or embedding Neo4j in their own software.

Extending Neo4j

This section describes how to extend Neo4j and Cypher using procedures, functions, and plugins. This section introduces different methods to extend the standard Neo4j functionality. How to set up remote debugging is also explained.

Neo4j provides the following methods to extend the standard functionality:

- **Procedures and functions** extend the capabilities of the Cypher query language.

- **Authentication and authorization plugins** extend the Neo4j security framework.

- **Server extensions** enable new surfaces to be created in the HTTP API.

Writing extensions requires the user to be familiar with Java or other JVM programming languages, and to have an environment set up for compiling such code.

Best practices

It is also important to consider any security implications of deploying customized code. Refer to the Operations Manual → Securing Extensions for details on best practices for securing user-defined procedures and functions.

Since you will be running customized-built code and Neo4j in the same JVM, there are a few things you should keep in mind:

- Do not create or retain more objects than you strictly need to. Large caches in particular tend to promote more objects to the old generation, thus increasing the need for expensive full garbage collections.

- Do not use internal Neo4j APIs. They are internal to Neo4j and subject to change without notice, which may break or change the behavior of your code.

- If possible, avoid using Java object serialization or reflection in your code or in any runtime dependency that you use. Otherwise, if you cannot avoid using Java object serialization and reflection, then ensure that the `-XX:+TrustFinalNonStaticFields` JVM flag is disabled in `neo4j.conf`.

Neo4j customized code

User-defined procedures and *user-defined functions* are mechanisms that enable you to extend Neo4j by writing customized code, which can be invoked directly from Cypher. This is the preferred means for extending Neo4j.

Examples of use cases for procedures and functions are:

- To provide access to functionality that is not available in Cypher.

- To provide access to third party systems.

- To perform graph-global operations, such as counting connected components or finding dense nodes.

- To express a procedural operation that is difficult to express declaratively with Cypher.

Procedures and functions are written in Java and compiled into *jar* files. They are deployed to the database by dropping that *jar* file into the *plugins* directory on each standalone or clustered server. For the location of the *plugins* directory, refer to Operations Manual → File locations. The database must be re-started on each server to pick up new procedures and functions.

Procedures and functions can take arguments and return results. In addition, procedures can perform write operations on the database.

Type	Description	Syntax	Read/Write	Cardinality
`Procedure`	For each row the procedure takes parameters and returns multiple results.	`CALL abc(...)`	Update allowed.	Changes cardinality similarly to a `MATCH` clause (0, 1, or many).
`Scalar function`	For each row the function takes parameters and returns a single result.	`abc(...)`	Read-only.	Maintains cardinality, one for one.
`Aggregating function`	Consumes many rows and produces an aggregated result.	`WITH abc(...)`	Read-only.	Reduces cardinality, many down to one.

Neo4j also comes bundled with a number of *built-in* procedures and functions.

The available built-in procedures varies depending edition and mode, as described in Operations Manual → Procedures. Running `CALL dbms.procedures()` will display the full list of procedures available in a particular database instance, including user-defined procedures.

The built-in functions are described in Cypher Manual → Functions. Running `CALL dbms.functions()` will display the full list of all the functions available in a particular database instance, including user-defined functions.

Setting up a plugin project

This section describes how to prepare a project for writing user-defined procedures and functions in Neo4j. The following shows the steps to create and deploy a new procedure.

> �----Q   The example described in this section is available on GitHub (neo4j-examples/neo4j-procedure-template).

Set up a project with Maven

A project can be set up in any way that allows for compiling a procedure and producing a *jar* file.

Below are the main parts of the example configuration, using the Maven build system.

Excerpts from the Maven pom.xml file

```xml
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.neo4j.example</groupId>
    <artifactId>procedure-template</artifactId>
    <version>1.0.0-SNAPSHOT</version>

    <packaging>jar</packaging>
    <name>Neo4j Procedure Template</name>
    <description>A template project for building a Neo4j Procedure</description>

    <properties>
      <neo4j.version>3.5.30</neo4j.version>
    </properties>
```

Build dependencies

Add a dependency section that includes the procedure and function APIs, which procedures and functions use at runtime.

The scope is set to `provided`, because once the procedure is deployed to a Neo4j instance, this dependency is provided by Neo4j. If non-Neo4j dependencies are added to the project, their scope should normally be `compile`.

```xml
<dependency>
  <groupId>org.neo4j</groupId>
  <artifactId>neo4j</artifactId>
  <version>${neo4j.version}</version>
  <scope>provided</scope>
</dependency>
```

Add dependencies that are necessary for testing the procedure.

Neo4j Harness, a utility that allows for starting a lightweight Neo4j instance. It is used to start Neo4j with a specific procedure or function deployed, which greatly simplifies testing.

Neo4j Java Driver, used to send Cypher statements that call the procedure or function.

JUnit, a common Java test framework.

```xml
<dependency>
  <groupId>org.neo4j.test</groupId>
  <artifactId>neo4j-harness</artifactId>
  <version>${neo4j.version}</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>org.neo4j.driver</groupId>
  <artifactId>neo4j-java-driver</artifactId>
  <version>1.7.5</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
```

Build steps

The steps that Maven will go through to build the project.

The goal is first to *compile* the source, then to *package* it in a *jar* that can be deployed to a Neo4j instance.

> Procedures and functions require at least Java 8, so the version `1.8` should be defined as the source and target version in the configuration for the Maven compiler plugin.

The Maven Shade plugin is used to package the compiled procedure. It also includes all dependencies in the package, unless the dependency scope is set to *test* or *provided*.

Once the procedure has been deployed to the *plugins* directory of each Neo4j instance and the instances have restarted, the procedure is available for use.

```xml
<build>
<plugins>
 <plugin>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.1</version>
   <configuration>
     <source>1.8</source>
     <target>1.8</target>
   </configuration>
 </plugin>
 <plugin>
   <artifactId>maven-shade-plugin</artifactId>
   <executions>
     <execution>
       <phase>package</phase>
       <goals>
         <goal>shade</goal>
       </goals>
     </execution>
   </executions>
 </plugin>
</plugins>
</build>
```

Values and types

This describes how to work with values and types when writing user-defined procedures and functions.

The *input* and *output* to and from a procedure or a function must be one of the supported types, as described in Cypher Manual → Values and types.

Composite types are supported via:

- `List<T>`, where `T` is one the supported types, and

- `Map<String, Object>`, where the values in the map must have one of the supported types.

The use of `Object` is supported for the case where the type is not known beforehand. Note, however, that the actual value must still have one of the aforementioned types.

The Cypher types and their equivalents in Java are outlined in the table below:

Table 1. Supported types

Cypher type	Java type
String	String
Integer	Long
Float	Double
Boolean	Boolean
Point	org.neo4j.graphdb.spatial.Point
Date	java.time.LocalDate
Time	java.time.OffsetTime
LocalTime	java.time.LocalTime
DateTime	java.time.ZonedDateTime
LocalDateTime	java.time.LocalDateTime
Duration	java.time.temporal.TemporalAmount
Node	org.neo4j.graphdb.Node
Relationship	org.neo4j.graphdb.Relationship
Path	org.neo4j.graphdb.Path

For more details, see the Neo4j Javadocs `org.neo4j.procedure.Procedure`.

Take note that there are two cases where more than one Java type is mapped to a single Cypher type. When this happens, type information is lost. If the following objects are returned from procedures, the original types cannot be recreated:

- A Cypher `Duration` is created when either a `java.time.Duration` or a `java.time.Period` is provided. If a `Duration` is returned, only the common interface `java.time.temporal.TemporalAmount` will remain.

- A Cypher `DateTime` is created when a `java.time.OffsetDateTime` is provided. If a `DateTime` is returned, it will be converted into a `java.time.ZonedDateTime`.

User-defined procedures

This describes how to write, test, and deploy a user-defined procedure for Neo4j.

A *user-defined procedure* is a mechanism that enables you to extend Neo4j by writing customized code, which can be invoked directly from Cypher. Procedures can take arguments, perform operations on the database, and return results. For a comparison between user-defined procedures, functions, and aggregation functions see Neo4j customized code.

Call a procedure

To call a user-defined procedure, use a Cypher `CALL` clause. The procedure name must be fully qualified, so a procedure named `findDenseNodes` defined in the package `org.neo4j.examples` could be called using:

```
CALL org.neo4j.examples.findDenseNodes(1000)
```

A `CALL` may be the only clause within a Cypher statement or may be combined with other clauses. Arguments can be supplied directly within the query or taken from the associated parameter set. For full details, see the documentation in Cypher Manual → `CALL` procedure.

Create a procedure

Make sure you have read and followed the preparatory setup instructions in Setting up a plugin project.

💡	The example discussed below is available as a repository on GitHub. To get started quickly you can fork the repository and work with the code as you follow along in the guide below.

1. Create integration tests.
2. Define the procedure.

Integration tests

The test dependencies include *Neo4j Harness* and *JUnit*. These can be used to write integration tests for procedures.

First, decide what the procedure should do, then write a test that proves that it does it right. Finally, write a procedure that passes the test.

Below is a template for testing a procedure that accesses Neo4j's full-text indexes from Cypher.

```java
package example;

import org.junit.Rule;
import org.junit.Test;
import org.neo4j.driver.v1.*;
import org.neo4j.graphdb.factory.GraphDatabaseSettings;
import org.neo4j.harness.junit.Neo4jRule;

import static org.hamcrest.core.IsEqual.equalTo;
import static org.junit.Assert.assertThat;
import static org.neo4j.driver.v1.Values.parameters;

public class ManualFullTextIndexTest
{
    // This rule starts a Neo4j instance
    @Rule
    public Neo4jRule neo4j = new Neo4jRule()

            // This is the Procedure to test
            .withProcedure( FullTextIndex.class );

    @Test
    public void shouldAllowIndexingAndFindingANode() throws Throwable
    {
        // In a try-block, to make sure you close the driver after the test
        try( Driver driver = GraphDatabase.driver( neo4j.boltURI() , Config.build().withoutEncryption
().toConfig() ) )
        {

            // Given I've started Neo4j with the FullTextIndex procedure class
            //       which my 'neo4j' rule above does.
            Session session = driver.session();

            // And given I have a node in the database
            long nodeId = session.run( "CREATE (p:User {name:'Brookreson'}) RETURN id(p)" )
                    .single()
                    .get( 0 ).asLong();

            // When I use the index procedure to index a node
            session.run( "CALL example.index($id, ['name'])", parameters( "id", nodeId ) );

            // Then I can search for that node with lucene query syntax
            StatementResult result = session.run( "CALL example.search('User', 'name:Brook*')" );
            assertThat( result.single().get( "nodeId" ).asLong(), equalTo( nodeId ) );
        }
    }
}
```

Define a procedure

With the test in place, write a procedure that fulfills the expectations of the test. The full example is available in the Neo4j Procedure Template repository.

Particular things to note:

- All procedures are annotated @Procedure.

- The procedure annotation can take three optional arguments: name, mode, and eager.

  ° name is used to specify a different name for the procedure than the default generated, which is class.path.nameOfMethod. If mode is specified then name must be specified as well.

  ° mode is used to declare the types of interactions that the procedure will perform. A procedure will fail if it attempts to execute database operations that violates its mode. The default mode is READ. The following modes are available:

    ▪ READ This procedure will only perform read operations against the graph.

- **WRITE** This procedure will perform read and write operations against the graph.

- **SCHEMA** This procedure will perform operations against the schema, i.e. create and drop indexes and constraints. A procedure with this mode is able to read graph data, but not write.

- **DBMS** This procedure will perform system operations such as user management and query management. A procedure with this mode is not able to read or write graph data.

  ◦ **eager** is a boolean setting defaulting to `false`. If it is set to `true` then the Cypher planner will plan an extra `eager` operation before calling the procedure. This is useful in cases where the procedure makes changes to the database in a way that could interact with the operations preceding the procedure. For example:

```
MATCH (n)
WHERE n.key = 'value'
WITH n
CALL deleteNeighbours(n, 'FOLLOWS')
```

  This query could delete some of the nodes that would be matched by the Cypher query, and then the `n.key` lookup will fail. Marking this procedure as `eager` will prevent this from causing an error in Cypher code. However, it is still possible for the procedure to interfere with itself by trying to read entities it has previously deleted. It is the responsibility of the procedure author to handle that case.

- The context of the procedure, which is the same as each resource that the procedure wants to use, is annotated `@Context`.

> 🛈 The correct way to signal an error from within a procedure is to throw a `RuntimeException`.

```java
package example;

import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Stream;

import org.neo4j.graphdb.GraphDatabaseService;
import org.neo4j.graphdb.Label;
import org.neo4j.graphdb.Node;
import org.neo4j.graphdb.index.Index;
import org.neo4j.graphdb.index.IndexManager;
import org.neo4j.logging.Log;
import org.neo4j.procedure.Context;
import org.neo4j.procedure.Name;
import org.neo4j.procedure.PerformsWrites;
import org.neo4j.procedure.Procedure;

import static org.neo4j.helpers.collection.MapUtil.stringMap;
import static org.neo4j.procedure.Procedure.Mode.SCHEMA;
import static org.neo4j.procedure.Procedure.Mode.WRITE;

/**
 * This is an example showing how you could expose Neo4j's full-text indexes as
 * two procedures - one for updating indexes, and one for querying by label and
 * the lucene query language.
 */
public class FullTextIndex
{
    // Only static fields and @Context-annotated fields are allowed in
    // Procedure classes. This static field is the configuration we use
    // to create full-text indexes.
    private static final Map<String,String> FULL_TEXT =
            stringMap( IndexManager.PROVIDER, "lucene", "type", "fulltext" );
```

```java
    // This field declares that we need a GraphDatabaseService
    // as context when any procedure in this class is invoked
    @Context
    public GraphDatabaseService db;

    // This gives us a log instance that outputs messages to the
    // standard log, `neo4j.log`
    @Context
    public Log log;

    /**
     * This declares the first of two procedures in this class - a
     * procedure that performs queries in a manual index.
     *
     * It returns a Stream of Records, where records are
     * specified per procedure. This particular procedure returns
     * a stream of {@link SearchHit} records.
     *
     * The arguments to this procedure are annotated with the
     * {@link Name} annotation and define the position, name
     * and type of arguments required to invoke this procedure.
     * There is a limited set of types you can use for arguments,
     * these are as follows:
     *
     * <ul>
     *     <li>{@link String}</li>
     *     <li>{@link Long} or {@code long}</li>
     *     <li>{@link Double} or {@code double}</li>
     *     <li>{@link Number}</li>
     *     <li>{@link Boolean} or {@code boolean}</li>
     *     <li>{@link org.neo4j.graphdb.Node}</li>
     *     <li>{@link org.neo4j.graphdb.Relationship}</li>
     *     <li>{@link org.neo4j.graphdb.Path}</li>
     *     <li>{@link java.util.Map} with key {@link String} and value of any type in this list, including
{@link java.util.Map}</li>
     *     <li>{@link java.util.List} of elements of any valid field type, including {@link
java.util.List}</li>
     *     <li>{@link Object}, meaning any of the types above</li>
     *
     * @param label the label name to query by
     * @param query the lucene query, for instance `name:Brook*` to
     *              search by property `name` and find any value starting
     *              with `Brook`. Please refer to the Lucene Query Parser
     *              documentation for full available syntax.
     * @return the nodes found by the query
     */
    @Procedure( name = "example.search", mode = WRITE )
    public Stream<SearchHit> search( @Name("label") String label,
                                     @Name("query") String query )
    {
        String index = indexName( label );

        // Avoid creating the index, if it is not there we will not be
        // finding anything anyway!
        if( !db.index().existsForNodes( index ))
        {
            // Just to show how you would do logging
            log.debug( "Skipping index query since index does not exist: `%s`", index );
            return Stream.empty();
        }

        // If there is an index, do a lookup and convert the result
        // to our output record.
        return db.index()
                .forNodes( index )
                .query( query )
                .stream()
                .map( SearchHit::new );
    }

    /**
     * This is the second procedure defined in this class, it is used to update the
     * index with nodes that should be queryable. You can send the same node multiple
     * times, if it already exists in the index the index will be updated to match
     * the current state of the node.
     *
```

```java
     * This procedure works largely the same as {@link #search(String, String)},
     * with three notable differences. One, it is annotated with `mode = SCHEMA`,
     * which is <i>required</i> if you want to perform updates to the graph in your
     * procedure.
     *
     * Two, it returns {@code void} rather than a stream. This is a short-hand
     * for saying our procedure always returns an empty stream of empty records.
     *
     * Three, it uses a default value for the property list, in this way you can call
     * the procedure by invoking {@code CALL index(nodeId)}. Default values are
     * are provided as the Cypher string representation of the given type, e.g.
     * {@code {default: true}}, {@code null}, or {@code -1}.
     *
     * @param nodeId the id of the node to index
     * @param propKeys a list of property keys to index, only the ones the node
     *                 actually contains will be added
     */
    @Procedure( name = "example.index", mode = SCHEMA )
    public void index( @Name("nodeId") long nodeId,
                       @Name(value = "properties", defaultValue = "[]") List<String> propKeys )
    {
        Node node = db.getNodeById( nodeId );

        // Load all properties for the node once and in bulk,
        // the resulting set will only contain those properties in `propKeys`
        // that the node actually contains.
        Set<Map.Entry<String,Object>> properties =
                node.getProperties( propKeys.toArray( new String[0] ) ).entrySet();

        // Index every label (this is just as an example, we could filter which labels to index)
        for ( Label label : node.getLabels() )
        {
            Index<Node> index = db.index().forNodes( indexName( label.name() ), FULL_TEXT );

            // In case the node is indexed before, remove all occurrences of it so
            // we do not get old or duplicated data
            index.remove( node );

            // And then index all the properties
            for ( Map.Entry<String,Object> property : properties )
            {
                index.add( node, property.getKey(), property.getValue() );
            }
        }
    }


    /**
     * This is the output record for our search procedure. All procedures
     * that return results return them as a Stream of Records, where the
     * records are defined like this one - customized to fit what the procedure
     * is returning.
     *
     * The fields must be one of the following types:
     *
     * <ul>
     *     <li>{@link String}</li>
     *     <li>{@link Long} or {@code long}</li>
     *     <li>{@link Double} or {@code double}</li>
     *     <li>{@link Number}</li>
     *     <li>{@link Boolean} or {@code boolean}</li>
     *     <li>{@link org.neo4j.graphdb.Node}</li>
     *     <li>{@link org.neo4j.graphdb.Relationship}</li>
     *     <li>{@link org.neo4j.graphdb.Path}</li>
     *     <li>{@link java.util.Map} with key {@link String} and value {@link Object}</li>
     *     <li>{@link java.util.List} of elements of any valid field type, including {@link
java.util.List}</li>
     *     <li>{@link Object}, meaning any of the valid field types</li>
     * </ul>
     */
    public static class SearchHit
    {
        // This records contain a single field named 'nodeId'
        public long nodeId;

        public SearchHit( Node node )
        {
```

```
            this.nodeId = node.getId();
        }
    }

    private String indexName( String label )
    {
        return "label-" + label;
    }
}
```

Injectable resources

When writing procedures, some resources can be injected into the procedure from the database. To inject these, use the `@Context` annotation. The classes that can be injected are:

- `Log`

- `TerminationGuard`

- `GraphDatabaseService`

All of the above classes are considered safe and future-proof, and will not compromise the security of the database. There are also several classes that can be injected that are unsupported (restricted) and can be changed with little or no notice. Procedures written to use these restricted API's will not be loaded by default, and it will be necessary to use the `dbms.security.procedures.unrestricted` to load unsafe procedures. Read more about this config setting in Operations Manual → Securing extensions.

User-defined functions

This describes how to write, test and deploy a user-defined function for Neo4j.

User-defined functions are a simpler form of procedures that are read-only and always return a single value. Although they are not as powerful in capability, they are often easier to use and more efficient than procedures for many common tasks. For a comparison between user-defined procedures, functions, and aggregation functions see Neo4j customized code.

Call a user-defined function

User-defined functions are called in the same way as any other Cypher function. The function name must be fully qualified, so a function named `join` defined in the package `org.neo4j.examples` could be called using:

```
MATCH (p: Person) WHERE p.age = 36
RETURN org.neo4j.examples.join(collect(p.names))
```

Create a function

User-defined functions are created similarly to how procedures are created, but are instead annotated with `@UserFunction` and instead of returning a stream of values it returns a single value.

See Values and types for details on values and types.

For more details, see the Neo4j Javadocs for `org.neo4j.procedure.UserFunction`.

> **ℹ** The correct way to signal an error from within a function is to throw a `RuntimeException`.

```java
package example;

import org.neo4j.procedure.Name;
import org.neo4j.procedure.Procedure;
import org.neo4j.procedure.UserFunction;

public class Join
{
    @UserFunction
    @Description("example.join(['s1','s2',...], delimiter) - join the given strings with the given delimiter.")
    public String join(
            @Name("strings") List<String> strings,
            @Name(value = "delimiter", defaultValue = ",") String delimiter) {
        if (strings == null || delimiter == null) {
            return null;
        }
        return String.join(delimiter, strings);
    }
}
```

Integration tests

Tests for user-defined functions are created in the same way as those for procedures.

Below is a template for testing a user-defined function that joins a list of strings.

```
package example;

import org.junit.Rule;
import org.junit.Test;
import org.neo4j.driver.v1.*;
import org.neo4j.harness.junit.Neo4jRule;

import static org.hamcrest.core.IsEqual.equalTo;
import static org.junit.Assert.assertThat;

public class JoinTest
{
    // This rule starts a Neo4j instance
    @Rule
    public Neo4jRule neo4j = new Neo4jRule()

            // This is the function to test
            .withFunction( Join.class );

    @Test
    public void shouldAllowIndexingAndFindingANode() throws Throwable
    {
        // This is in a try-block, to make sure you close the driver after the test
        try( Driver driver = GraphDatabase.driver( neo4j.boltURI() , Config.build().withEncryptionLevel(
Config.EncryptionLevel.NONE ).toConfig() ) )
        {
            // Given
            Session session = driver.session();

            // When
            String result = session.run( "RETURN example.join(['Hello', 'World']) AS result").single().
get("result").asString();

            // Then
            assertThat( result, equalTo( "Hello,World" ) );
        }
    }
}
```

User-defined aggregation functions

This describes how to write, test and deploy a user-defined aggregation function for Neo4j.

User-defined aggregation functions are functions that aggregate data and return a single result. For a comparison between user-defined procedures, functions, and aggregation functions see Neo4j customized code.

Call a aggregation function

User-defined aggregation functions are called in the same way as any other Cypher aggregation function. The function name must be fully qualified, so a function named longestString defined in the package org.neo4j.examples could be called using:

```
MATCH (p: Person) WHERE p.age = 36
RETURN org.neo4j.examples.longestString(p.name)
```

Writing a user-defined aggregation function

User-defined aggregation functions are annotated with @UserAggregationFunction. The annotated function must return an instance of an aggregator class. An aggregator class contains one method

annotated with @UserAggregationUpdate and one method annotated with @UserAggregationResult. The method annotated with @UserAggregationUpdate will be called multiple times and enables the class to aggregate data. When the aggregation is done the method annotated with @UserAggregationResult is called once and the result of the aggregation will be returned.

See Values and types for details on values and types.

For more details, see the Neo4j Javadocs for org.neo4j.procedure.UserAggregationFunction.

> **ℹ** The correct way to signal an error from within an aggregation function is to throw a RuntimeException.

```java
package example;

import org.neo4j.procedure.Description;
import org.neo4j.procedure.Name;
import org.neo4j.procedure.UserAggregationFunction;
import org.neo4j.procedure.UserAggregationResult;
import org.neo4j.procedure.UserAggregationUpdate;

public class LongestString
{
    @UserAggregationFunction
    @Description( "org.neo4j.function.example.longestString(string) - aggregates the longest string found" )
    public LongStringAggregator longestString()
    {
        return new LongStringAggregator();
    }

    public static class LongStringAggregator
    {
        private int longest;
        private String longestString;

        @UserAggregationUpdate
        public void findLongest(
                @Name( "string" ) String string )
        {
            if ( string != null && string.length() > longest)
            {
                longest = string.length();
                longestString = string;
            }
        }

        @UserAggregationResult
        public String result()
        {
            return longestString;
        }
    }
}
```

Integration tests

Tests for user-defined aggregation functions are created in the same way as those for normal user-defined functions.

Below is a template for testing a user-defined aggregation function that finds the longest string.

```java
package example;

import org.junit.Rule;
import org.junit.Test;
import org.neo4j.driver.v1.*;
import org.neo4j.harness.junit.Neo4jRule;

import static org.hamcrest.core.IsEqual.equalTo;
import static org.junit.Assert.assertThat;

public class LongestStringTest
{
    // This rule starts a Neo4j instance
    @Rule
    public Neo4jRule neo4j = new Neo4jRule()

            // This is the function to test
            .withAggregationFunction( LongestString.class );

    @Test
    public void shouldAllowIndexingAndFindingANode() throws Throwable
    {
        // This is in a try-block, to make sure you close the driver after the test
        try( Driver driver = GraphDatabase.driver( neo4j.boltURI() , Config.build().withEncryptionLevel(
Config.EncryptionLevel.NONE ).toConfig() ) )
        {
            // Given
            Session session = driver.session();

            // When
            String result = session.run( "UNWIND ["abc", "abcd", "ab"] AS string RETURN
example.longestString(string) AS result").single().get("result").asString();

            // Then
            assertThat( result, equalTo( "abcd" ) );
        }
    }
}
```

Authentication and authorization plugins

This describes Neo4j support for customized-built authentication and authorization plugins.

Neo4j provides authentication and authorization plugin interfaces to support real-world deployment scenarios not covered by native users or the built-in configuration-based LDAP connector.

The SPI (Service Provider Interface) lives in the `com.neo4j.server.security.enterprise.auth.plugin.spi` package.

Customized-built plugins have access to the *<neo4j-home>* directory in case you want to load any customized settings from a file located there. Plugins can also write to the security event log.

Authentication plugin

The authentication plugin implements the `AuthenticationPlugin` interface with the `authenticate` method.

The example below shows a minimal authentication plugin that checks for Neo4j user with Neo4j password:

```
@Override
public AuthenticationInfo authenticate( AuthToken authToken )
{
    String principal = authToken.principal();
    char[] credentials = authToken.credentials();

    if ( principal.equals( "neo4j" ) && Arrays.equals( credentials, "neo4j".toCharArray() ) ) )
    {
        return (AuthenticationInfo) () -> "neo4j";
    }
    return null;
}
```

Authorization plugin

The authorization plugin implements the `AuthorizationPlugin` interface with the `authorize` method.

The example below shows a minimal authorization plugin that assigns the reader role to a user named neo4j:

```
@Override
public AuthorizationInfo authorize( Collection<PrincipalAndProvider> principals )
{
    if ( principals.stream().anyMatch( p -> "neo4j".equals( p.principal() ) ) )
    {
        return (AuthorizationInfo) () -> Collections.singleton( PredefinedRoles.READER );
    }
    return null;
}
```

Note the usage of the helper class `PredefinedRole`.

Simplified combined plugin

There is also a simplified combined plugin interface `AuthPlugin` that provides both authentication and authorization in a single method called `authenticateAndAuthorize`.

The example below shows a combined plugin verifying neo4j/neo4j credentials and returning reader role authorization:

```
@Override
public AuthInfo authenticateAndAuthorize( AuthToken authToken )
{
    String principal = authToken.principal();
    char[] credentials = authToken.credentials();

    if ( principal.equals( "neo4j" ) && Arrays.equals( credentials, "neo4j".toCharArray() ) ) )
    {
        return AuthInfo.of( "neo4j", Collections.singleton( PredefinedRoles.READER ) );
    }
    return null;
}
```

Extendable platform

Neo4j provides an extendable platform as some user deployment scenarios may not be easily configured through standard LDAP connector. One known complexity is integrating with LDAP user directory where groups have users as a member and the not other way around.

The example below first searches for a group that the user is member of, and then maps that group to the Neo4j role by calling the customized-built `getNeo4jRoleForGroupId` method:

```java
@Override
public AuthInfo authenticateAndAuthorize( AuthToken authToken ) throws AuthenticationException
{
    try
    {
        String username = authToken.principal();
        char[] password = authToken.credentials();

        LdapContext ctx = authenticate( username, password );
        Set<String> roles = authorize( ctx, username );

        return AuthInfo.of( username, roles );
    }
    catch ( NamingException e )
    {
        throw new AuthenticationException( e.getMessage() );
    }
}

private LdapContext authenticate( String username, char[] password ) throws NamingException
{
    Hashtable<String,Object> env = new Hashtable<>();
    env.put( Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory" );
    env.put( Context.PROVIDER_URL, "ldap://0.0.0.0:10389" );

    env.put( Context.SECURITY_PRINCIPAL, String.format( "cn=%s,ou=users,dc=example,dc=com", username ) );
    env.put( Context.SECURITY_CREDENTIALS, password );

    return new InitialLdapContext( env, null );
}

private Set<String> authorize( LdapContext ctx, String username ) throws NamingException
{
    Set<String> roleNames = new LinkedHashSet<>();

    // Set up our search controls
    SearchControls searchCtls = new SearchControls();
    searchCtls.setSearchScope( SearchControls.SUBTREE_SCOPE );
    searchCtls.setReturningAttributes( new String[]{GROUP_ID} );

    // Use a search argument to prevent potential code injection
    Object[] searchArguments = new Object[]{username};

    // Search for groups that has the user as a member
    NamingEnumeration result = ctx.search( GROUP_SEARCH_BASE, GROUP_SEARCH_FILTER, searchArguments,
searchCtls );

    if ( result.hasMoreElements() )
    {
        SearchResult searchResult = (SearchResult) result.next();

        Attributes attributes = searchResult.getAttributes();
        if ( attributes != null )
        {
            NamingEnumeration attributeEnumeration = attributes.getAll();
            while ( attributeEnumeration.hasMore() )
            {
                Attribute attribute = (Attribute) attributeEnumeration.next();
                String attributeId = attribute.getID();
                if ( attributeId.equalsIgnoreCase( GROUP_ID ) )
                {
                    // Found a group that the user is a member of. See if it has a role mapped to it
                    String groupId = (String) attribute.get();
                    String neo4jGroup = getNeo4jRoleForGroupId( groupId );
                    if ( neo4jGroup != null )
                    {
                        // Yay! Add it to your set of roles
                        roleNames.add( neo4jGroup );
                    }
                }
            }
        }
    }
    return roleNames;
}
```

Unmanaged server extensions

Introduction

Sometimes you will want a finer-grained level of control over your application's interactions with Neo4j than Cypher provides. For these situations you can use the unmanaged extension API.

🔥 This is a sharp tool, that enables users to deploy arbitrary JAX-RS classes to the server so be careful when using this. In particular, it is possible to consume lots of heap space on the server and degrade performance. If in doubt, please ask for help via one of the community channels.

The first step when writing an unmanaged extension is to create a project which includes dependencies to the Neo4j core jars. In Maven this would be achieved by adding the following lines to the pom file:

```
1  <dependency>
2      <groupId>org.neo4j.3rdparty.javax.ws.rs</groupId>
3      <artifactId>jsr311-api</artifactId>
4      <version>1.1.2.r612</version>
5      <scope>provided</scope>
6  </dependency>
```

```
1  <dependency>
2      <groupId>org.neo4j</groupId>
3      <artifactId>neo4j</artifactId>
4      <version>3.5.30</version>
5      <scope>provided</scope>
6  </dependency>
```

Now you are ready to write your extension.

In your code, you will interact with the database using `GraphDatabaseService`, which you can access by using the `@Context` annotation. The following example serves as a template which you can base your extension on:

```
@Path( "/helloworld" )
public class HelloWorldResource
{
    private final GraphDatabaseService database;

    public HelloWorldResource( @Context GraphDatabaseService database )
    {
        this.database = database;
    }

    @GET
    @Produces( MediaType.TEXT_PLAIN )
    @Path( "/{nodeId}" )
    public Response hello( @PathParam( "nodeId" ) long nodeId )
    {
        // Do stuff with the database
        return Response.status( Status.OK ).entity( UTF8.encode( "Hello World, nodeId=" + nodeId ) ).
build();
    }
}
```

The full source code is found here: HelloWorldResource.java

Having built your code, the resulting jar file (and any customized dependencies) should be placed in the
$NEO4J_SERVER_HOME/plugins directory. You also need to tell Neo4j where to look for the extension by
adding some configuration in *neo4j.conf*:

```
#Comma separated list of JAXRS packages containing JAXRS Resource, one package name for each mountpoint.
dbms.unmanaged_extension_classes=org.neo4j.examples.server.unmanaged=/examples/unmanaged
```

Your hello method will now respond to GET requests at the URI:

```
http://{neo4j_server}:{neo4j_port}/examples/unmanaged/helloworld/{node_id}
```

For example:

```
curl http://localhost:7474/examples/unmanaged/helloworld/123
```

which results in:

```
Hello World, nodeId=123
```

Streaming JSON responses

When writing unmanaged extensions, you have greater control over the amount of memory that your
Neo4j queries use. If you keep too much state around, it can lead to more frequent full Garbage Collection
and subsequent unresponsiveness by the Neo4j server.

A common way that state can increase, is the creation of JSON objects to represent the result of a query
which is then sent back to your application. Neo4j's Transactional Cypher HTTP endpoint (see HTTP API
Docs → transactional Cypher endpoint) streams responses back to the client. For example, the following
unmanaged extension streams an array of a person's colleagues:

```java
@Path("/colleagues")
public class ColleaguesResource
{
    private GraphDatabaseService graphDb;
    private final ObjectMapper objectMapper;

    private static final RelationshipType ACTED_IN = RelationshipType.withName( "ACTED_IN" );
    private static final Label PERSON = Label.label( "Person" );

    public ColleaguesResource( @Context GraphDatabaseService graphDb )
    {
        this.graphDb = graphDb;
        this.objectMapper = new ObjectMapper();
    }

    @GET
    @Path("/{personName}")
    public Response findColleagues( @PathParam("personName") final String personName )
    {
        StreamingOutput stream = new StreamingOutput()
        {
            @Override
            public void write( OutputStream os ) throws IOException, WebApplicationException
            {
                JsonGenerator jg = objectMapper.getJsonFactory().createJsonGenerator( os, JsonEncoding
.UTF8 );
                jg.writeStartObject();
                jg.writeFieldName( "colleagues" );
                jg.writeStartArray();

                try ( Transaction tx = graphDb.beginTx();
                      ResourceIterator<Node> persons = graphDb.findNodes( PERSON, "name", personName ) )
                {
                    while ( persons.hasNext() )
                    {
                        Node person = persons.next();
                        for ( Relationship actedIn : person.getRelationships( ACTED_IN, OUTGOING ) )
                        {
                            Node endNode = actedIn.getEndNode();
                            for ( Relationship colleagueActedIn : endNode.getRelationships( ACTED_IN,
INCOMING ) )
                            {
                                Node colleague = colleagueActedIn.getStartNode();
                                if ( !colleague.equals( person ) )
                                {
                                    jg.writeString( colleague.getProperty( "name" ).toString() );
                                }
                            }
                        }
                    }
                    tx.success();
                }

                jg.writeEndArray();
                jg.writeEndObject();
                jg.flush();
                jg.close();
            }
        };

        return Response.ok().entity( stream ).type( MediaType.APPLICATION_JSON ).build();
    }
}
```

The full source code is found here: ColleaguesResource.java

As well as depending on JAX-RS API, this example also uses Jackson — a Java JSON library. You will need to add the following dependency to your Maven POM file (or equivalent):

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.10.2</version>
</dependency>
```

From Neo4j 3.5.15, a breaking change was introduced following an update to the Jackson dependency.

Jackson v1 is out of support, and has accumulated security issues such as:

- CVE-2017-7525

- CVE-2017-17485

- CVE-2017-15095

- CVE-2018-11307

- CVE-2018-7489

- CVE-2018-5968

For further information about Jackson v2, please see the Jackson Project on GitHub.

Our `findColleagues` method will now respond to `GET` requests at the URI:

```
http://{neo4j_server}:{neo4j_port}/examples/unmanaged/colleagues/{personName}
```

For example:

```
curl http://localhost:7474/examples/unmanaged/colleagues/Keanu%20Reeves
```

which results in:

```
{"colleagues":["Hugo Weaving","Carrie-Anne Moss","Laurence Fishburne"]}
```

Executing Cypher

You can execute Cypher queries by using the `GraphDatabaseService` that is injected into the extension. For example, the following unmanaged extension retrieves a person's colleagues using Cypher:

```
@Path("/colleagues-cypher-execution")
public class ColleaguesCypherExecutionResource
{
    private final ObjectMapper objectMapper;
    private GraphDatabaseService graphDb;

    public ColleaguesCypherExecutionResource( @Context GraphDatabaseService graphDb )
    {
        this.graphDb = graphDb;
        this.objectMapper = new ObjectMapper();
    }

    @GET
    @Path("/{personName}")
    public Response findColleagues( @PathParam("personName") final String personName )
    {
        final Map<String, Object> params = MapUtil.map( "personName", personName );

        StreamingOutput stream = new StreamingOutput()
        {
            @Override
            public void write( OutputStream os ) throws IOException, WebApplicationException
            {
                JsonGenerator jg = objectMapper.getJsonFactory().createJsonGenerator( os, JsonEncoding
.UTF8 );
                jg.writeStartObject();
                jg.writeFieldName( "colleagues" );
                jg.writeStartArray();

                try ( Transaction tx = graphDb.beginTx();
                      Result result = graphDb.execute( colleaguesQuery(), params ) )
                {
                    while ( result.hasNext() )
                    {
                        Map<String,Object> row = result.next();
                        jg.writeString( ((Node) row.get( "colleague" )).getProperty( "name" ).toString()
);
                    }
                    tx.success();
                }

                jg.writeEndArray();
                jg.writeEndObject();
                jg.flush();
                jg.close();
            }
        };

        return Response.ok().entity( stream ).type( MediaType.APPLICATION_JSON ).build();
    }

    private String colleaguesQuery()
    {
        return "MATCH (p:Person {name: $personName })-[:ACTED_IN]->()<-[:ACTED_IN]-(colleague) RETURN
colleague";
    }
}
```

The full source code is found here: ColleaguesCypherExecutionResource.java

Your findColleagues method will now respond to GET requests at the URI:

```
http://{neo4j_server}:{neo4j_port}/examples/unmanaged/colleagues-cypher-execution/{personName}
```

For example:

```
curl http://localhost:7474/examples/unmanaged/colleagues-cypher-execution/Keanu%20Reeves
```

which results in:

```
{"colleagues": ["Hugo Weaving", "Carrie-Anne Moss", "Laurence Fishburne"]}
```

Testing your extension

Neo4j provides tools to help you write integration tests for your extensions. You can access this toolkit by adding the following test dependency to your project:

```
1  <dependency>
2      <groupId>org.neo4j.test</groupId>
3      <artifactId>neo4j-harness</artifactId>
4      <version>3.5.30</version>
5      <scope>test</scope>
6  </dependency>
```

The test toolkit provides a mechanism to start a Neo4j instance with a customized configuration and with extensions of your choice. It also provides mechanisms to specify data fixtures to include when starting Neo4j, as you can see in the example below:

```java
@Path("")
public static class MyUnmanagedExtension
{
    @GET
    public Response myEndpoint()
    {
        return Response.ok().build();
    }
}

@Test
public void testMyExtension() throws Exception
{
    // Given
    try ( ServerControls server = getServerBuilder()
            .withExtension( "/myExtension", MyUnmanagedExtension.class )
            .newServer() )
    {
        // When
        HTTP.Response response = HTTP.GET(
                HTTP.GET( server.httpURI().resolve( "myExtension" ).toString() ).location() );

        // Then
        assertEquals( 200, response.status() );
    }
}

@Test
public void testMyExtensionWithFunctionFixture() throws Exception
{
    // Given
    try ( ServerControls server = getServerBuilder()
            .withExtension( "/myExtension", MyUnmanagedExtension.class )
            .withFixture( new Function<GraphDatabaseService, Void>()
            {
                @Override
                public Void apply( GraphDatabaseService graphDatabaseService ) throws RuntimeException
                {
                    try ( Transaction tx = graphDatabaseService.beginTx() )
                    {
                        graphDatabaseService.createNode( Label.label( "User" ) );
                        tx.success();
                    }
                    return null;
                }
            } )
            .newServer() )
    {
        // When
        Result result = server.graph().execute( "MATCH (n:User) return n" );

        // Then
        assertEquals( 1, count( result ) );
    }
}
```

The full source code of the example is found here: ExtensionTestingDocIT.java

Note the use of `server.httpURI().resolve( "myExtension" )` to ensure that the correct base URI is used.

If you are using the JUnit test framework, there is a JUnit rule available as well:

```
@Rule
public Neo4jRule neo4j = new Neo4jRule()
        .withFixture( "CREATE (admin:Admin)" )
        .withConfig( LegacySslPolicyConfig.certificates_directory.name(),
                getRelativePath( getSharedTestTemporaryFolder(), LegacySslPolicyConfig
.certificates_directory ) )
        .withFixture( new Function<GraphDatabaseService, Void>()
        {
            @Override
            public Void apply( GraphDatabaseService graphDatabaseService ) throws RuntimeException
            {
                try (Transaction tx = graphDatabaseService.beginTx())
                {
                    graphDatabaseService.createNode( Label.label( "Admin" ) );
                    tx.success();
                }
                return null;
            }
        } );

@Test
public void shouldWorkWithServer() throws Exception
{
    // Given
    URI serverURI = neo4j.httpURI();

    // When I access the server
    HTTP.Response response = HTTP.GET( serverURI.toString() );

    // Then it should reply
    assertEquals(200, response.status());

    // and we have access to underlying GraphDatabaseService
    try (Transaction tx = neo4j.getGraphDatabaseService().beginTx()) {
        assertEquals( 2, count(neo4j.getGraphDatabaseService().findNodes( Label.label( "Admin" ) ) ));
        tx.success();
    }
}
```

The full source code of the example is found here: JUnitDocIT.java

Setup for remote debugging

This describes how to configure Neo4j for remote debugging sessions.

In order to configure Neo4j for remote debugging sessions, the Java debugging parameters need to be passed to the Java process through the configuration. They live in the *conf/neo4j.conf* file.

In order to specify the parameters, you must add a line for the additional Java arguments like this:

```
dbms.jvm.additional=-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=*:5005
```

This configuration will start Neo4j, ready for remote debugging attachment at localhost and port 5005. Use these parameters to attach to the process from Eclipse, IntelliJ or your remote debugger of choice after starting the server.

Using Neo4j embedded in Java applications

This section describes how to use Neo4j embedded in Java applications.

The following topics are:

- Including Neo4j in your project
- Hello world
- Property values
- Using indexes
- Explicit indexes `Deprecated` — `Deprecated`
- Resource iterator — Managing resources in long-running transactions.
- Controlling logging
- Unit test — `Deprecated`
- Traversal
- Domain entities
- Graph algorithm examples
- Reading a management attribute `Deprecated` — `Deprecated`
- Unique nodes — Getting or creating a unique node using Cypher and uniqueness constraints.
- Bolt connector — Accessing Neo4j embedded via the Bolt protocol.
- Terminate a transaction — How to terminate (abort) a long-running transaction from another thread.
- Cypher queries — How to use the Cypher query language with Java.
- Query parameters
- Online Backup from Java `Deprecated` — `Deprecated`

⚠️	When running your own code and Neo4j in the same JVM, there are a few things you should keep in mind: • Do not create or retain more objects than you strictly need to. Large caches in particular tend to promote more objects to the old generation, thus increasing the need for expensive full garbage collections. • Do not use internal Neo4j APIs. They are internal to Neo4j and subject to change without notice, which may break or change the behavior of your code. • Do not enable the `-XX:+TrustFinalNonStaticFields` JVM flag when running in embedded mode.

Including Neo4j in your project

This topic describes how to embed Neo4j in your Java application.

After selecting the appropriate edition for your platform, you can embed Neo4j in your Java application by including the Neo4j library jars in your build. The following sections show how to do this by either altering the build path directly or by using dependency management.

Adding Neo4j to the build path

Get the Neo4j libraries from one of these sources:

- Extract a Neo4j zip/tarball, and use the *jar* files found in the *lib/* directory.

- Use the *jar* files available from Maven Central Repository.

Add the jar files to your project:

JDK tools

Append to `-classpath`

Eclipse

- Right-click on the project and then go *Build Path → Configure Build Path*. In the dialog, choose *Add External JARs*, browse to the Neo4j *lib/* directory and select all of the jar files.

- Another option is to use User Libraries.

IntelliJ IDEA

See Libraries, Global Libraries, and the Configure Library dialog.

NetBeans

- Right-click on the *Libraries* node of the project, choose *Add JAR/Folder*, browse to the Neo4j *lib/* directory and select all of the jar files.

- You can also handle libraries from the project node, see Managing a Project's Classpath.

Editions

The following table outlines the available editions and their names for use with dependency management tools.

> 💡 Follow the links in the table for details on dependency configuration with Apache Maven, Apache Buildr, Apache Ivy, Groovy Grape, Grails, Scala SBT!

Table 2. Neo4j editions

Neo4j Edition	Dependency	Description
Community	org.neo4j:neo4j	A high performance, fully ACID transactional graph database.
Enterprise	org.neo4j:neo4j-enterprise	Adding advanced monitoring, online backup, and clustering.

Note that the listed dependencies do not contain the implementation, but pulls it in transitively.

For information regarding licensing, see the Licensing Guide.

Javadocs can be downloaded packaged in jar files from Maven Central or read at Neo4j Javadocs.

Adding Neo4j as a dependency

You can either go with the top-level artifact from the table above or include the individual components directly. The examples included here use the top-level artifact approach.

> **i** *Additional information for Enterprise Edition users.*
>
> The examples below are only valid Neo4j Community Edition.
>
> To add Neo4j Enterprise Edition as a dependency, please get in contact with Neo4j Professional Services.
>
> See Operations Manual → Introduction for details about the Community and Enterprise Editions.

Maven

Add the dependency to your project along the lines of the snippet below. This is usually done in the *pom.xml* file found in the root directory of the project.

```
 1 <project>
 2 ...
 3  <dependencies>
 4   <dependency>
 5    <groupId>org.neo4j</groupId>
 6    <artifactId>neo4j</artifactId>
 7    <version>3.5.30</version>
 8   </dependency>
 9    ...
10  </dependencies>
11 ...
12 </project>
```

Where the `artifactId` is found in the editions table.

Eclipse and Maven

For development in Eclipse, it is recommended to install the m2e plugin and let Maven manage the project build classpath instead, see above. This also adds the possibility to build your project both via the command line with Maven and have a working Eclipse setup for development.

Ivy

Make sure to resolve dependencies from Maven Central. You can use this configuration in your *ivysettings.xml* file:

```
<ivysettings>
  <settings defaultResolver="main"/>
  <resolvers>
    <chain name="main">
      <filesystem name="local">
        <artifact pattern="${ivy.settings.dir}/repository/[artifact]-[revision].[ext]" />
      </filesystem>
      <ibiblio name="maven_central" root="http://repo1.maven.org/maven2/" m2compatible="true"/>
    </chain>
  </resolvers>
</ivysettings>
```

With that in place, add Neo4j by adding this to your 'ivy.xml' file:

```
1 ..
2 <dependencies>
3   ..
4   <dependency org="org.neo4j" name="neo4j" rev="3.5.30"/>
5   ..
6 </dependencies>
7 ..
```

The name can be found in the editions table above.

Gradle

The example below shows an example Gradle build script for including the Neo4j libraries:

```
1 def neo4jVersion = "3.5.30"
2 apply plugin: 'java'
3 repositories {
4     mavenCentral()
5 }
6 dependencies {
7     compile "org.neo4j:neo4j:${neo4jVersion}"
8 }
```

The coordinates (org.neo4j:neo4j in the example) are found in the editions table above.

Starting and stopping

To create a new database or open an existing one you instantiate a
org.neo4j.graphdb.GraphDatabaseService.

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( databaseDirectory );
registerShutdownHook( graphDb );
```

If you are using the Enterprise Edition of Neo4j in embedded standalone mode, you have to create your database with the org.neo4j.graphdb.factory.EnterpriseGraphDatabaseFactory to enable the Enterprise Edition features.

If you wish to use the High Availability clustering features, instead use the org.neo4j.graphdb.factory.HighlyAvailableGraphDatabaseFactory. This includes Enterprise Edition.

| | org.neo4j.graphdb.factory.HighlyAvailableGraphDatabaseFactory is deprecated and will be replaced in a future release. Deprecated |

There is no factory for the Causal Clustering features, as it is currently not possible to run a Causal Cluster in embedded mode.

| | A GraphDatabaseService instance can be shared among multiple threads. Note however, that you cannot create multiple services pointing to the same database. |

To stop the database, call the shutdown() method:

```
graphDb.shutdown();
```

To make sure Neo4j is shut down properly, add a shutdown hook:

```java
private static void registerShutdownHook( final GraphDatabaseService graphDb )
{
    // Registers a shutdown hook for the Neo4j instance so that it
    // shuts down nicely when the VM exits (even if you "Ctrl-C" the
    // running application).
    Runtime.getRuntime().addShutdownHook( new Thread()
    {
        @Override
        public void run()
        {
            graphDb.shutdown();
        }
    } );
}
```

Starting an embedded database with configuration settings

To start Neo4j with configuration settings, a Neo4j properties file can be loaded like this:

```java
GraphDatabaseService graphDb = new GraphDatabaseFactory()
    .newEmbeddedDatabaseBuilder( testDirectory.databaseDir() )
    .loadPropertiesFromFile( pathToConfig + "neo4j.conf" )
    .newGraphDatabase();
```

Configuration settings can also be applied programmatically, like so:

```java
GraphDatabaseService graphDb = new GraphDatabaseFactory()
    .newEmbeddedDatabaseBuilder( testDirectory.databaseDir() )
    .setConfig( GraphDatabaseSettings.pagecache_memory, "512M" )
    .setConfig( GraphDatabaseSettings.string_block_size, "60" )
    .setConfig( GraphDatabaseSettings.array_block_size, "300" )
    .newGraphDatabase();
```

Starting an embedded read-only instance

If you want a *read-only view* of the database, create an instance this way:

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabaseBuilder( dir )
        .setConfig( GraphDatabaseSettings.read_only, "true" )
        .newGraphDatabase();
```

Obviously, the database has to already exist in this case.

> **i** Concurrent access to the same database files by multiple (read-only or write) instances is not supported.

Hello world

This describes how to create and access nodes and relationships.

For information on project setup, see Including Neo4j in your project.

A Neo4j graph consists of:

- nodes
- relationships that connects the nodes
- properties on both nodes and relationships

All relationships have a type. For example, if the graph represents a social network, a relationship type could be KNOWS. If a relationship of the type KNOWS connects two nodes, that is likely to represent two people that know each other. A lot of the semantics of a graph is encoded in the relationship types of the application. Although relationships are directed, they are equally traversed regardless of direction.

> **💡** The source code of this example is found here: EmbeddedNeo4j.java

Preparing the database

Relationship types can be created by using an enum. In this example, you only need a single relationship type. This is how to define it:

```
private enum RelTypes implements RelationshipType
{
    KNOWS
}
```

You can also prepare some variables to use:

```
GraphDatabaseService graphDb;
Node firstNode;
Node secondNode;
Relationship relationship;
```

The next step is to start the database server. Note that if the directory given for the database does not already exist, it will be created.

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( databaseDirectory );
registerShutdownHook( graphDb );
```

> ℹ️ Starting a database server is an expensive operation, so do not start up a new instance every time you need to interact with the database. The instance can be shared by multiple threads, and transactions are thread confined.

As seen, you can register a shutdown hook that will make sure the database shuts down when the JVM exits.

```java
private static void registerShutdownHook( final GraphDatabaseService graphDb )
{
    // Registers a shutdown hook for the Neo4j instance so that it
    // shuts down nicely when the VM exits (even if you "Ctrl-C" the
    // running application).
    Runtime.getRuntime().addShutdownHook( new Thread()
    {
        @Override
        public void run()
        {
            graphDb.shutdown();
        }
    } );
}
```

Next step is to interact with the database.

Wrapping operations in a transaction

All operations have to be performed in a transaction. This is a deliberate design decision, since we believe transaction demarcation to be an important part of working with a real enterprise database. The example below illustrates transaction handling in Neo4j:

```java
try ( Transaction tx = graphDb.beginTx() )
{
    // Database operations go here
    tx.success();
}
```

For more information on transactions, see Transaction management and the Neo4j Javadocs for `org.neo4j.graphdb.Transaction`.

> ℹ️ For brevity, wrapping of operations in a transaction is not spelled out throughout the manual.

Creating a small graph

You can now create a few nodes. The API is very intuitive as you can see in the Neo4j Javadocs (they are included in the distribution as well).
This is how to create a small graph consisting of two nodes, connected with one relationship and some properties:
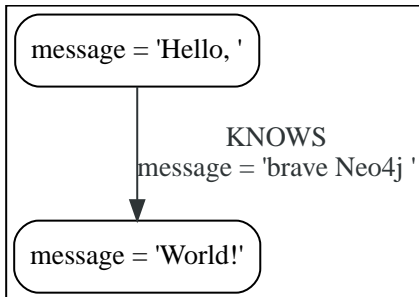
```
firstNode = graphDb.createNode();
firstNode.setProperty( "message", "Hello, " );
secondNode = graphDb.createNode();
secondNode.setProperty( "message", "World!" );

relationship = firstNode.createRelationshipTo( secondNode, RelTypes.KNOWS );
relationship.setProperty( "message", "brave Neo4j " );
```

You now have a graph that looks like this:

Printing the result

After you have created your graph, you can read from it and print the result.

```
System.out.print( firstNode.getProperty( "message" ) );
System.out.print( relationship.getProperty( "message" ) );
System.out.print( secondNode.getProperty( "message" ) );
```

Which will output:

```
Hello, brave Neo4j World!
```

Removing the data

In this case, the data will be removed before committing:

```
firstNode.getSingleRelationship( RelTypes.KNOWS, Direction.OUTGOING ).delete();
firstNode.delete();
secondNode.delete();
```

> ℹ️ Deleting a node which still has relationships when the transaction commits will fail. This is to make sure relationships always have a start node and an end node.

Shutting down the database server

Finally, shut down the database server *when the application finishes:*

```
graphDb.shutdown();
```

Property values

This describes how both nodes and relationships can have properties.

Properties are named values where the name is a string. Property values can be either a primitive or an array of one primitive type. For example `String`, `int` and `int[]` values are valid for properties.

> **i** `NULL` is **not a valid** property value. Setting a property to `NULL` is equivalent to deleting the property.

Table 3. Property value types

Type	Description
`boolean`	
`byte`	8-bit integer.
`short`	16-bit integer.
`int`	32-bit integer.
`long`	64-bit integer.
`float`	32-bit IEEE 754 floating-point number.
`double`	64-bit IEEE 754 floating-point number.
`char`	16-bit unsigned integers representing Unicode characters.
`String`	Sequence of Unicode characters.
`org.neo4j.graphdb.spatial.Point`	A 2D or 3D point object in a given coordinate system.
`java.time.LocalDate`	An instant capturing the date, but not the time, nor the timezone.
`java.time.OffsetTime`	An instant capturing the time of day, and the timezone offset, but not the date.
`java.time.LocalTime`	An instant capturing the time of day, but not the date, nor the timezone.
`java.time.ZonedDateTime`	An instant capturing the date, the time, and the timezone.
`java.time.LocalDateTime`	An instant capturing the date and the time, but not the timezone.
`java.time.temporal.TemporalAmount`	A temporal amount. This captures the difference in time between two instants.

For further details on float/double values, see Java Language Specification.

Note that there are two cases where more than one Java type is mapped to a single Cypher type. When this happens, type information is lost. If these objects are returned from procedures, the original types cannot be recreated:

- A Cypher `Duration` is created when either a `java.time.Duration` or a `java.time.Period` is provided. If a `Duration` is returned, only the common interface `java.time.temporal.TemporalAmount` will remain.

- A Cypher `DateTime` is created when a `java.time.OffsetDateTime` is provided. If a `DateTime` is returned, it will be converted into a `java.time.ZonedDateTime`.

> ℹ️ Strings that contain special characters can have inconsistent or non-deterministic ordering in Neo4j. For details, see Cypher Manual → Sorting of special characters.

Using indexes

This demonstrates how to work with indexes with an example of a user database.

> 💡 The source code used in this example is found here:
> EmbeddedNeo4jWithNewIndexing.java

Begin with starting the database server:

```java
GraphDatabaseService graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( databaseDirectory );
```

Then you can configure the database to index users by name. This only needs to be done once. However, note that schema changes and data changes are not allowed in the same transaction. Each transaction must either change the schema or the data, but not both.

```java
IndexDefinition indexDefinition;
try ( Transaction tx = graphDb.beginTx() )
{
    Schema schema = graphDb.schema();
    indexDefinition = schema.indexFor( Label.label( "User" ) )
            .on( "username" )
            .create();
    tx.success();
}
```

Indexes are populated asynchronously when they are first created. It is possible to use the core API to wait for index population to complete:

```java
try ( Transaction tx = graphDb.beginTx() )
{
    Schema schema = graphDb.schema();
    schema.awaitIndexOnline( indexDefinition, 10, TimeUnit.SECONDS );
}
```

It is also possible to query the progress of the index population:

```java
try ( Transaction tx = graphDb.beginTx() )
{
    Schema schema = graphDb.schema();
    System.out.println( String.format( "Percent complete: %1.0f%%",
            schema.getIndexPopulationProgress( indexDefinition ).getCompletedPercentage() ) );
}
```

Now you can add the users:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = Label.label( "User" );

    // Create some users
    for ( int id = 0; id < 100; id++ )
    {
        Node userNode = graphDb.createNode( label );
        userNode.setProperty( "username", "user" + id + "@neo4j.org" );
    }
    System.out.println( "Users created" );
    tx.success();
}
```

> ℹ️ Please read Resource iterator on how to properly close `ResourceIterators` returned from index lookups.

And this is how to find a user by ID:

```
Label label = Label.label( "User" );
int idToFind = 45;
String nameToFind = "user" + idToFind + "@neo4j.org";
try ( Transaction tx = graphDb.beginTx() )
{
    try ( ResourceIterator<Node> users =
                graphDb.findNodes( label, "username", nameToFind ) )
    {
        ArrayList<Node> userNodes = new ArrayList<>();
        while ( users.hasNext() )
        {
            userNodes.add( users.next() );
        }

        for ( Node node : userNodes )
        {
            System.out.println(
                    "The username of user " + idToFind + " is " + node.getProperty( "username" ) );
        }
    }
}
```

When updating the name of a user, the index is updated as well:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = Label.label( "User" );
    int idToFind = 45;
    String nameToFind = "user" + idToFind + "@neo4j.org";

    for ( Node node : loop( graphDb.findNodes( label, "username", nameToFind ) ) )
    {
        node.setProperty( "username", "user" + (idToFind + 1) + "@neo4j.org" );
    }
    tx.success();
}
```

When deleting a user, it is automatically removed from the index:

```

```
try (Transaction tx = graphDb.beginTx())
{
    Label label = Label.label("User");
    int idToFind = 46;
    String nameToFind = "user" + idToFind + "@neo4j.org";

    for (Node node : loop(graphDb.findNodes(label, "username", nameToFind)))
    {
        node.delete();
    }
    tx.success();
}
```

In case you change your data model, you can drop the index as well:

```
try ( Transaction tx = graphDb.beginTx() )
{
    Label label = Label.label( "User" );
    for ( IndexDefinition indexDefinition : graphDb.schema()
            .getIndexes( label ) )
    {
        // There is only one index
        indexDefinition.drop();
    }

    tx.success();
}
```

Explicit indexes Deprecated

⚠️ Explicit indexes are deprecated and due to be removed in Neo4j 4.0. Please see Cypher Manual → Indexes to support full-text search to learn about using full-text schema indexes instead.

This demonstrates how to work with indexes with an example of a user database.

💡 The source code used in this example is found here: EmbeddedNeo4jWithIndexing.java

First create two helper methods:

```
private static String idToUserName( final int id )
{
    return "user" + id + "@neo4j.org";
}

private static Node createAndIndexUser( final String username )
{
    Node node = graphDb.createNode();
    node.setProperty( USERNAME_KEY, username );
    nodeIndex.add( node, USERNAME_KEY, username );
    return node;
}
```

The next step is to start the database server:

```
graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( databaseDirectory );
registerShutdownHook();
```

Now you can add some users:

```
try ( Transaction tx = graphDb.beginTx() )
{
    nodeIndex = graphDb.index().forNodes( "nodes" );
    // Create some users and index their names with the IndexService
    for ( int id = 0; id < 100; id++ )
    {
        createAndIndexUser( idToUserName( id ) );
    }
}
```

And here is how to find a user by ID:

```
int idToFind = 45;
String userName = idToUserName( idToFind );
Node foundUser = nodeIndex.get( USERNAME_KEY, userName ).getSingle();

System.out.println( "The username of user " + idToFind + " is "
    + foundUser.getProperty( USERNAME_KEY ) );
```

> ℹ️ Please read Resource iterator on how to properly close ResourceIterators returned from index lookups.

Resource iterator

This describes best practices for managing resources in long-running transactions.

Inside a long-running transaction it is good practice to ensure that any org.neo4j.graphdb.ResourceIterator obtained inside the transaction are closed as early as possible. This is either achieved by exhausting the iterator or by explicitly calling its close method.

Below is an example of how to work with a ResourceIterator. As you do not exhaust the iterator, you can close it explicitly using the close() method.

```
Label label = Label.label( "User" );
int idToFind = 45;
String nameToFind = "user" + idToFind + "@neo4j.org";
try ( Transaction tx = graphDb.beginTx();
        ResourceIterator<Node> users = graphDb.findNodes( label, "username", nameToFind ) )
{
    Node firstUserNode;
    if ( users.hasNext() )
    {
        firstUserNode = users.next();
    }
    users.close();
}
```

Controlling logging

This describes how to control logging in Neo4j embedded using the Neo4j embedded logging framework.

Neo4j embedded provides logging via its own org.neo4j.logging.Log layer, and does not natively use any existing Java logging framework. All logging events produced by Neo4j have a name, a level and a message. The name is a FQCN (fully qualified class name).

Neo4j uses the following log levels:

Level Name	Description
ERROR	For serious errors that are almost always fatal.
WARN	For events that are serious, but not fatal.
INFO	Informational events.
DEBUG	Debugging events.

To enable logging, an implementation of `org.neo4j.logging.LogProvider` must be provided to the `org.neo4j.graphdb.factory.GraphDatabaseFactory`, as follows:

```
LogProvider logProvider = new MyCustomLogProvider( output );
graphDb = new GraphDatabaseFactory().setUserLogProvider( logProvider ).newEmbeddedDatabase(
databaseDirectory );
```

Neo4j also includes a binding for SLF4J, which is available in the `neo4j-slf4j` library jar. This can be obtained via Maven:

```
<project>
...
  <dependencies>
    <dependency>
      <groupId>org.neo4j</groupId>
      <artifactId>neo4j-slf4j</artifactId>
      <version>3.5.30</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>${slf4j-version}</version>
    </dependency>
    ...
  </dependencies>
...
</project>
```

To use this binding, pass an instance of `org.neo4j.logging.slf4j.Slf4jLogProvider` to the `org.neo4j.graphdb.factory.GraphDatabaseFactory`, as follows:

```
graphDb = new GraphDatabaseFactory().setUserLogProvider( new Slf4jLogProvider() )
          .newEmbeddedDatabase( databaseDirectory );
```

All log output can then be controlled via SLF4J configuration.

Unit test

⚠️ The testing methods described in this section rely on internal artifacts and APIs, and are deprecated. New methods for testing will be provided in Neo4j 4.0.

Please use `it-test-support` to access Neo4j testing facilities. They are available for download at Maven Central `it-test-support`.

The basic pattern of unit testing with Neo4j is illustrated by the following example.

Example 1. Basic unit testing

> **ℹ**   For the full source code of this example see `Neo4jBasicDocTest.java`.

Using Maven as a dependency manager you would typically add this dependency together with JUnit and Hamcrest like so:

```xml
<project>
...
  <dependencies>
    <dependency>
      <groupId>org.neo4j.community</groupId>
      <artifactId>it-test-support</artifactId>
      <version>3.5.30</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.hamcrest</groupId>
      <artifactId>hamcrest-all</artifactId>
      <version>1.3</version>
      <scope>test</scope>
    </dependency>
    ...
  </dependencies>
...
</project>
```

With that in place, you are ready to code some unit tests.

Before each test, create a fresh database:

```java
@Before
public void prepareTestDatabase()
{
    graphDb = new TestGraphDatabaseFactory().newImpermanentDatabase( testDirectory.directory() );
}
```

After the test has executed, the database should be shut down:

```java
@After
public void destroyTestDatabase()
{
    graphDb.shutdown();
}
```

During a test, create nodes and check to see that they are there, while enclosing write operations in a transaction.

```java
Node n;
try ( Transaction tx = graphDb.beginTx() )
{
    n = graphDb.createNode();
    n.setProperty( "name", "Nancy" );
    tx.success();
}

// The node should have a valid id
assertThat( n.getId(), is( greaterThan( -1L ) ) );

// Retrieve a node by using the id of the created node. The id's and
// property should match.
try ( Transaction tx = graphDb.beginTx() )
{
    Node foundNode = graphDb.getNodeById( n.getId() );
    assertThat( foundNode.getId(), is( n.getId() ) );
    assertThat( (String) foundNode.getProperty( "name" ), is( "Nancy" ) );
}
```

If you want to set configuration parameters at database creation, it is done like this:

```java
GraphDatabaseService db = new TestGraphDatabaseFactory()
    .newImpermanentDatabaseBuilder()
    .setConfig( GraphDatabaseSettings.pagecache_memory, "512M" )
    .setConfig( GraphDatabaseSettings.string_block_size, "60" )
    .setConfig( GraphDatabaseSettings.array_block_size, "300" )
    .newGraphDatabase();
```

Traversal

This describes traversing into another graph.

For more information about traversals, see The traversal framework.

The Matrix

This is the first graph to traverse into:

Figure 1. Matrix node space view

💡 | The source code of this example is found here: `NewMatrix.java`

Friends and friends of friends:

```java
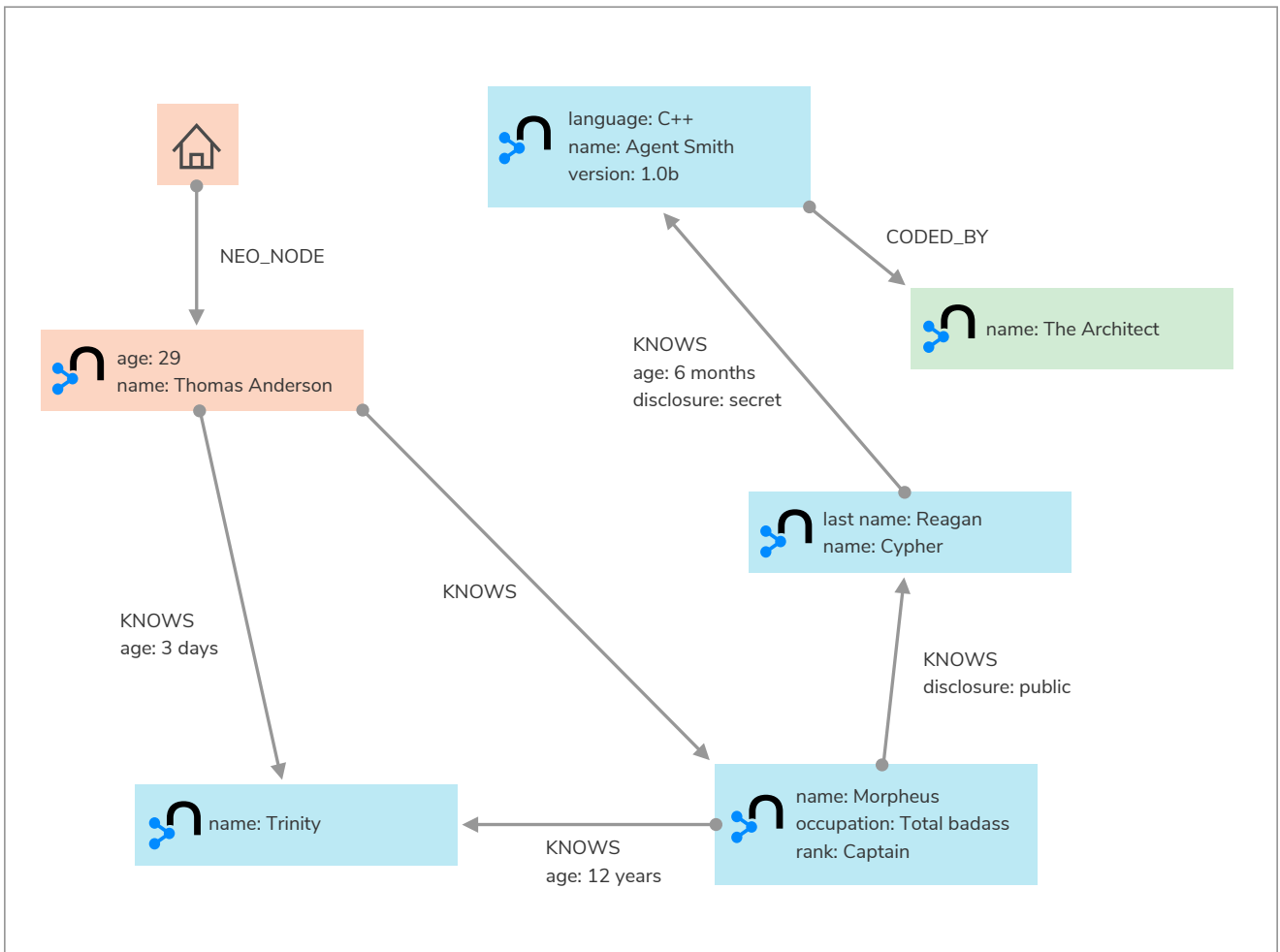private Traverser getFriends(
        final Node person )
{
    TraversalDescription td = graphDb.traversalDescription()
            .breadthFirst()
            .relationships( RelTypes.KNOWS, Direction.OUTGOING )
            .evaluator( Evaluators.excludeStartPosition() );
    return td.traverse( person );
}
```

You can perform the actual traversal and print the results:

```java
int numberOfFriends = 0;
String output = neoNode.getProperty( "name" ) + "'s friends:\n";
Traverser friendsTraverser = getFriends( neoNode );
for ( Path friendPath : friendsTraverser )
{
    output += "At depth " + friendPath.length() + " => "
            + friendPath.endNode()
                    .getProperty( "name" ) + "\n";
    numberOfFriends++;
}
output += "Number of friends found: " + numberOfFriends + "\n";
```

Which will give you the following output:

```
Thomas Anderson's friends:
At depth 1 => Morpheus
At depth 1 => Trinity
At depth 2 => Cypher
At depth 3 => Agent Smith
Number of friends found: 4
```

Who coded the Matrix?

```java
private Traverser findHackers( final Node startNode )
{
    TraversalDescription td = graphDb.traversalDescription()
            .breadthFirst()
            .relationships( RelTypes.CODED_BY, Direction.OUTGOING )
            .relationships( RelTypes.KNOWS, Direction.OUTGOING )
            .evaluator(
                    Evaluators.includeWhereLastRelationshipTypeIs( RelTypes.CODED_BY ) );
    return td.traverse( startNode );
}
```

Print out the result:

```java
String output = "Hackers:\n";
int numberOfHackers = 0;
Traverser traverser = findHackers( getNeoNode() );
for ( Path hackerPath : traverser )
{
    output += "At depth " + hackerPath.length() + " => "
                + hackerPath.endNode()
                        .getProperty( "name" ) + "\n";
    numberOfHackers++;
}
output += "Number of hackers found: " + numberOfHackers + "\n";
```

Now you know who coded the Matrix:

```
Hackers:
At depth 4 => The Architect
Number of hackers found: 1
```

Walking an ordered path

This example shows how to use a path context holding a representation of a path.

The source code of this example is found here: `OrderedPath.java`

Create a toy graph:

```java
Node A = tx.createNode();
Node B = tx.createNode();
Node C = tx.createNode();
Node D = tx.createNode();

A.createRelationshipTo( C, REL2 );
C.createRelationshipTo( D, REL3 );
A.createRelationshipTo( B, REL1 );
B.createRelationshipTo( C, REL2 );
```
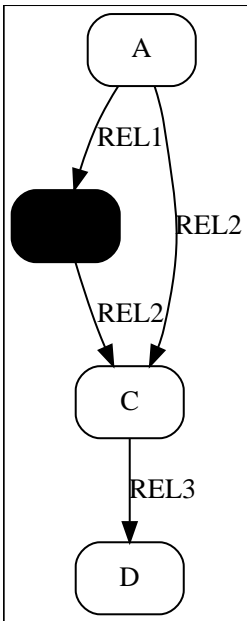
Now, the order of relationships (REL1 → REL2 → REL3) is stored in an `ArrayList`. Upon traversal, the `Evaluator` can check against it to ensure that only paths are included and returned that have the predefined order of relationships:

```java
final ArrayList<RelationshipType> orderedPathContext = new ArrayList<>();
orderedPathContext.add( REL1 );
orderedPathContext.add( withName( "REL2" ) );
orderedPathContext.add( withName( "REL3" ) );
TraversalDescription td = db.traversalDescription()
        .evaluator( new Evaluator()
        {
            @Override
            public Evaluation evaluate( final Path path )
            {
                if ( path.length() == 0 )
                {
                    return Evaluation.EXCLUDE_AND_CONTINUE;
                }
                RelationshipType expectedType = orderedPathContext.get( path.length() - 1 );
                boolean isExpectedType = path.lastRelationship()
                        .isType( expectedType );
                boolean included = path.length() == orderedPathContext.size() && isExpectedType;
                boolean continued = path.length() < orderedPathContext.size() && isExpectedType;
                return Evaluation.of( included, continued );
            }
        } )
        .uniqueness( Uniqueness.NODE_PATH ); ①
```

① Note that we set the uniqueness to `Uniqueness.NODE_PATH`, as you want to be able to revisit the same node during the traversal, but not the same path.

Perform the traversal and print the result:

```java
Traverser traverser = td.traverse( tx.getNodeById( A.getId() ) );
PathPrinter pathPrinter = new PathPrinter( "name" );
for ( Path path : traverser )
{
    output += Paths.pathToString( path, pathPrinter );
}
```

Which will output:

```
(A)--[REL1]-->(B)--[REL2]-->(C)--[REL3]-->(D)
```

In this case, a customized class is used to format the path output. This is how it is done:

```java
static class PathPrinter implements Paths.PathDescriptor<Path>
{
    private final String nodePropertyKey;

    public PathPrinter( String nodePropertyKey )
    {
        this.nodePropertyKey = nodePropertyKey;
    }

    @Override
    public String nodeRepresentation( Path path, Node node )
    {
        return "(" + node.getProperty( nodePropertyKey, "" ) + ")";
    }

    @Override
    public String relationshipRepresentation( Path path, Node from, Relationship relationship )
    {
        String prefix = "--", suffix = "--";
        if ( from.equals( relationship.getEndNode() ) )
        {
            prefix = "<--";
        }
        else
        {
            suffix = "-->";
        }
        return prefix + "[" + relationship.getType().name() + "]" + suffix;
    }
}
```

Uniqueness of Paths in traversals

This example is demonstrating the use of node uniqueness. Below an imaginary domain graph with Principals that own pets that are descendant to other pets.

Descendants example graph
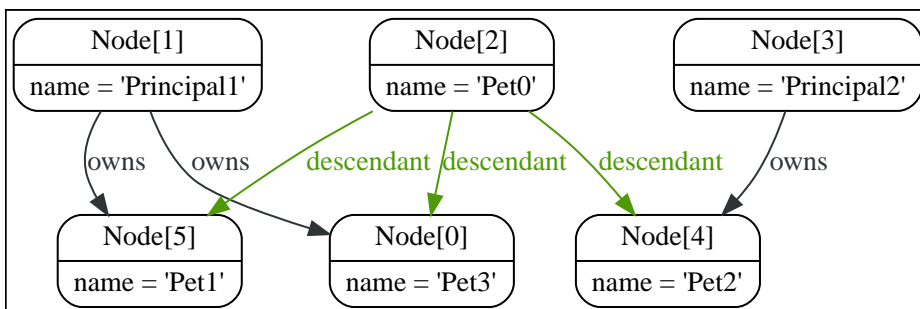
In order to return all descendants of `Pet0` which have the relation `owns` to `Principal1` (`Pet1` and `Pet3`), the Uniqueness of the traversal needs to be set to `NODE_PATH` rather than the default `NODE_GLOBAL`. This way nodes can be traversed more that once, and paths that have different nodes but can have some nodes in common (like the start and end node) can be returned.

> 💡 The source code of this example is found here: `UniquenessOfPathsDocTest.java`

```
final Node target = data.get().get( "Principal1" );
TraversalDescription td = db.traversalDescription()
        .uniqueness( Uniqueness.NODE_PATH )
        .evaluator( new Evaluator()
{
    @Override
    public Evaluation evaluate( Path path )
    {
        boolean endNodeIsTarget = path.endNode().equals( target );
        return Evaluation.of( endNodeIsTarget, !endNodeIsTarget );
    }
} );

Traverser results = td.traverse( start );
```

This will return the following paths:

```
(2)-[descendant,2]->(0)<-[owns,5]-(1)
(2)-[descendant,0]->(5)<-[owns,3]-(1)
```

In the default `path.toString()` implementation, `(1)--[knows,2]-→(4)` denotes a node with `ID=1` having a relationship with `ID=2` or type `knows` to a node with `ID=4`.

Let's create a new `TraversalDescription` from the old one, having `NODE_GLOBAL` uniqueness to see the difference.

> The `TraversalDescription` object is immutable, so we have to use the new instance returned with the new uniqueness setting.

```
TraversalDescription nodeGlobalTd = td.uniqueness( Uniqueness.NODE_GLOBAL );
results = nodeGlobalTd.traverse( start );
```

Now only one path is returned:

```
(2)-[descendant,2]->(0)<-[owns,5]-(1)
```

Domain entities

This describes one way to handle domain entities when using Neo4j.

Entities can be wrapped around a node. The same approach can be used with relationships.

> The source code of the examples is found here: `Person.java`

First off, store the node and make it accessible inside the package:

```java
private final Node underlyingNode;

Person( Node personNode )
{
    this.underlyingNode = personNode;
}

protected Node getUnderlyingNode()
{
    return underlyingNode;
}
```

Delegate attributes to the node:

```java
public String getName()
{
    return (String)underlyingNode.getProperty( NAME );
}
```

Make sure to override these methods:

```java
@Override
public int hashCode()
{
    return underlyingNode.hashCode();
}

@Override
public boolean equals( Object o )
{
    return o instanceof Person &&
            underlyingNode.equals( ( (Person)o ).getUnderlyingNode() );
}

@Override
public String toString()
{
    return "Person[" + getName() + "]";
}
```

Graph algorithm examples

For details on the graph algorithm usage, see the Neo4j Javadocs for
`org.neo4j.graphalgo.GraphAlgoFactory`.

> The source code used in the example is found here: `PathFindingDocTest.java`

Calculating the shortest path (least number of relationships) between two nodes:

```
Node startNode = tx.createNode();
Node middleNode1 = tx.createNode();
Node middleNode2 = tx.createNode();
Node middleNode3 = tx.createNode();
Node endNode = tx.createNode();
createRelationshipsBetween( startNode, middleNode1, endNode );
createRelationshipsBetween( startNode, middleNode2, middleNode3, endNode );

// Will find the shortest path between startNode and endNode via
// "MY_TYPE" relationships (in OUTGOING direction), like f.ex:
//
// (startNode)-->(middleNode1)-->(endNode)
//
PathFinder<Path> finder = GraphAlgoFactory.shortestPath( new BasicEvaluationContext( tx, graphDb ),
    PathExpanders.forTypeAndDirection( ExampleTypes.MY_TYPE, Direction.OUTGOING ), 15 );
Iterable<Path> paths = finder.findAllPaths( startNode, endNode );
```

Using Dijkstra's algorithm to calculate cheapest path between node A and B where each relationship can have a weight (i.e. cost) and the path(s) with least cost are found.

```
PathFinder<WeightedPath> finder = GraphAlgoFactory.dijkstra( new BasicEvaluationContext( tx, graphDb ),
    PathExpanders.forTypeAndDirection( ExampleTypes.MY_TYPE, Direction.BOTH ), "cost" );

WeightedPath path = finder.findSinglePath( nodeA, nodeB );

// Get the weight for the found path
path.weight();
```

Using A* to calculate the cheapest path between node A and B, where cheapest means, for example, the path in a network of roads which has the shortest length between node A and B. This is the example graph:
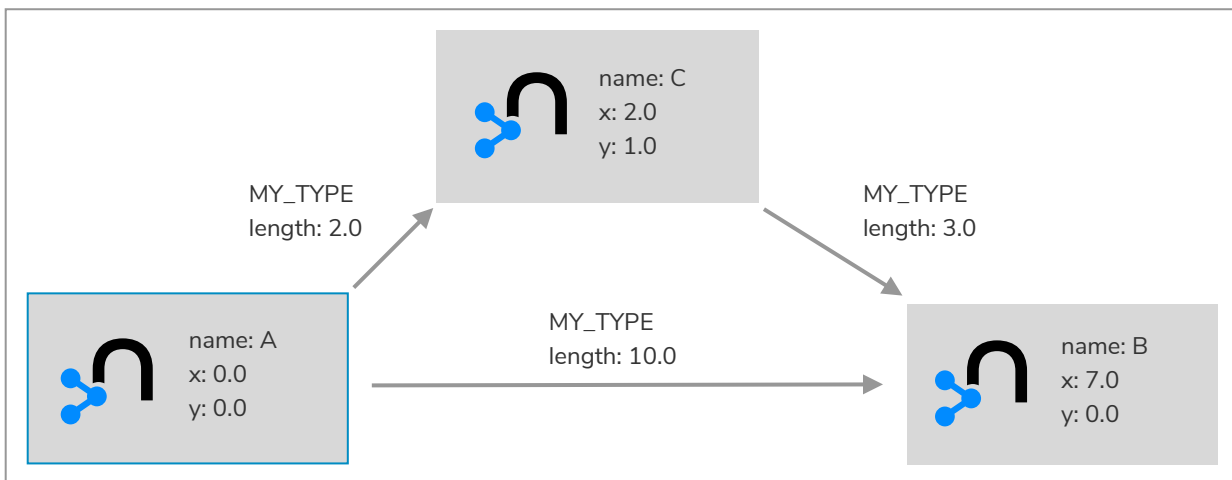
```
Node nodeA = createNode( "name", "A", "x", 0d, "y", 0d );
Node nodeB = createNode( "name", "B", "x", 7d, "y", 0d );
Node nodeC = createNode( "name", "C", "x", 2d, "y", 1d );
Relationship relAB = createRelationship( nodeA, nodeC, "length", 2d );
Relationship relBC = createRelationship( nodeC, nodeB, "length", 3d );
Relationship relAC = createRelationship( nodeA, nodeB, "length", 10d );

EstimateEvaluator<Double> estimateEvaluator = new EstimateEvaluator<Double>()
{
    @Override
    public Double getCost( final Node node, final Node goal )
    {
        double dx = (Double) node.getProperty( "x" ) - (Double) goal.getProperty( "x" );
        double dy = (Double) node.getProperty( "y" ) - (Double) goal.getProperty( "y" );
        double result = Math.sqrt( Math.pow( dx, 2 ) + Math.pow( dy, 2 ) );
        return result;
    }
};
PathFinder<WeightedPath> astar = GraphAlgoFactory.aStar(
        PathExpanders.allTypesAndDirections(),
        CommonEvaluators.doubleCostEvaluator( "length" ), estimateEvaluator );
WeightedPath path = astar.findSinglePath( nodeA, nodeB );
```

Reading a management attribute Deprecated

This topic describes how to access the Neo4j management beans that are included in the `org.neo4j.jmx.JmxUtils` class.

The JmxUtils class includes methods to access Neo4j management beans. The common JMX service can be used as well, but from your code you probably rather want to use the approach outlined below.

> **i** The source code of the example is found here: `JmxDocTest.java`

This example shows how to get the start time of a database:

```
private static Date getStartTimeFromManagementBean(
        GraphDatabaseService graphDbService )
{
    ObjectName objectName = JmxUtils.getObjectName( graphDbService, "Kernel" );
    Date date = JmxUtils.getAttribute( objectName, "KernelStartTime" );
    return date;
}
```

Depending on which Neo4j edition you are using different sets of management beans are available.

- For all editions, see the `org.neo4j.jmx` package.

- For the Enterprise Edition, see the `org.neo4j.management` package as well.

Unique nodes

This describes how to ensure uniqueness of a property when creating nodes.

Getting or creating a unique node using Cypher and uniqueness constraints.

For an overview of unique nodes, see Transaction management → Creating unique nodes.

> 💡 The source code for the examples can be found here: GetOrCreateDocIT.java

Create a unique constraint:

```
try ( Transaction tx = graphdb.beginTx() )
{
    graphdb.schema()
            .constraintFor( Label.label( "User" ) )
            .assertPropertyIsUnique( "name" )
            .create();
    tx.success();
}
```

Use `MERGE` to create a unique node:

```
Node result = null;
ResourceIterator<Node> resultIterator = null;
try ( Transaction tx = graphDb.beginTx() )
{
    String queryString = "MERGE (n:User {name: $name}) RETURN n";
    Map<String, Object> parameters = new HashMap<>();
    parameters.put( "name", username );
    resultIterator = graphDb.execute( queryString, parameters ).columnAs( "n" );
    result = resultIterator.next();
    tx.success();
    return result;
}
```

Getting or creating a unique node using an explicit index Deprecated

> ℹ️ While this is a working solution, please consider using the preferred solution at Getting or creating a unique node using Cypher and uniqueness constraints. instead.

By using `put-if-absent` functionality, entity uniqueness can be guaranteed using an index.

Here the index acts as the lock and will only lock the smallest part needed to guarantee uniqueness across threads and transactions. To get the more high-level get-or-create functionality make use of `org.neo4j.graphdb.index.UniqueFactory` as seen in the example below.

Create a factory for unique nodes at application start:

```java
try ( Transaction tx = graphDb.beginTx() )
{
    UniqueFactory.UniqueNodeFactory result = new UniqueFactory.UniqueNodeFactory( graphDb, "users" )
    {
        @Override
        protected void initialize( Node created, Map<String, Object> properties )
        {
            created.addLabel( Label.label( "User" ) );
            created.setProperty( "name", properties.get( "name" ) );
        }
    };
    tx.success();
    return result;
}
```

Use the unique node factory to get or create a node:

```java
try ( Transaction tx = graphDb.beginTx() )
{
    Node node = factory.getOrCreate( "name", username );
    tx.success();
    return node;
}
```

Pessimistic locking for node creation Deprecated

> ℹ️ While this is a working solution, please consider using the preferred solution at Getting or creating a unique node using Cypher and uniqueness constraints. instead.

One might be tempted to use Java synchronization for pessimistic locking, but this is dangerous. By mixing locks in Neo4j and in the Java runtime, it is possible to produce deadlocks that are not detectable by Neo4j. As long as all locking is done by Neo4j, all deadlocks will be detected and avoided. Also, a solution using manual synchronization does not ensure uniqueness in an HA environment.

This example uses a single "lock node" for locking. We create it only as a place to put locks, nothing else.

Create a lock node at application start:

```java
try ( Transaction tx = graphDb.beginTx() )
{
    final Node lockNode = graphDb.createNode();
    tx.success();
    return lockNode;
}
```

Use the lock node to ensure nodes are not created concurrently:

```java
try ( Transaction tx = graphDb.beginTx() )
{
    Index<Node> usersIndex = graphDb.index().forNodes( "users" );
    Node userNode = usersIndex.get( "name", username ).getSingle();
    if ( userNode != null )
    {
        return userNode;
    }

    tx.acquireWriteLock( lockNode );
    userNode = usersIndex.get( "name", username ).getSingle();
    if ( userNode == null )
    {
        userNode = graphDb.createNode( Label.label( "User" ) );
        usersIndex.add( userNode, "name", username );
        userNode.setProperty( "name", username );
    }
    tx.success();
    return userNode;
}
```

Note that finishing the transaction will release the lock on the lock node.

Bolt connector

This describes how to open a Bolt connector to your embedded instance to get GUI administration and other benefits. Accessing Neo4j embedded via the Bolt protocol.

The Neo4j Browser and the official Neo4j Drivers use the Bolt database protocol to communicate with Neo4j. By default, Neo4j Embedded does not expose a Bolt connector, but you can enable one. Doing so allows you to connect the services Neo4j Browser to your embedded instance.

It also gives you a way to incrementally transfer an existing Embedded application to use Neo4j Drivers instead. Migrating to Neo4j Drivers means you can run Neo4j Embedded or Neo4j Server, without having to change your application code.

To add a Bolt Connector to your embedded database, you must add the Bolt extension to your class path. This is done by adding an additional dependency to your project:

```xml
1 <project>
2 ...
3  <dependencies>
4
5   <dependency>
6     <groupId>org.neo4j</groupId>
7     <artifactId>neo4j-bolt</artifactId>
8     <version>3.5.30</version>
9   </dependency>
10   ...
11  </dependencies>
12 ...
13 </project>
```

With this dependency in place, you can configure Neo4j to enable a Bolt connector:

The source code for the example can be found here: EmbeddedNeo4jWithBolt.java

```
GraphDatabaseSettings.BoltConnector bolt = GraphDatabaseSettings.boltConnector( "0" );

GraphDatabaseService graphDb = new GraphDatabaseFactory()
        .newEmbeddedDatabaseBuilder( DB_PATH )
        .setConfig( bolt.type, "BOLT" )
        .setConfig( bolt.enabled, "true" )
        .setConfig( bolt.address, "localhost:7687" )
        .newGraphDatabase();
```

Terminate a transaction

The describes how to terminate (abort) a long-running transaction from another thread.

The source code for the examples can be found here: TerminateTransactions.java

First, start the database server:

```
GraphDatabaseService graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( databaseDirectory );
```

Then start creating an infinite binary tree of nodes in the database, as an example of a long-running transaction:

```
RelationshipType relType = RelationshipType.withName( "CHILD" );
Queue<Node> nodes = new LinkedList<>();
int depth = 1;

try ( Transaction tx = graphDb.beginTx() )
{
    Node rootNode = tx.createNode();
    nodes.add( rootNode );

    for (; true; depth++) {
        int nodesToExpand = nodes.size();
        for (int i = 0; i < nodesToExpand; ++i) {
            Node parent = nodes.remove();

            Node left = tx.createNode();
            Node right = tx.createNode();

            parent.createRelationshipTo( left, relType );
            parent.createRelationshipTo( right, relType );

            nodes.add( left );
            nodes.add( right );
        }
    }
}
catch ( TransactionTerminatedException ignored )
{
    return String.format( "Created tree up to depth %s in 1 sec", depth );
}
```

After waiting for some time, you decide to terminate the transaction. This is done from a separate thread:

```
tx.terminate();
```

Running this will execute the long-running transaction for about one second and prints the maximum depth of the tree that was created before the transaction was terminated. No changes are actually made to the data — because the transaction has been terminated, the end result is as if no operations were

performed.

This is an example output:

```
Created tree up to depth 16 in 1 sec
```

Finally, the database can be shut down again:

```
graphDb.shutdown();
```

Cypher queries

In Java, you can use the Cypher query language as per the example below.

> 💡 The source code for the examples can be found here: JavaQuery.java

First, you can add some data:

```
GraphDatabaseService db = new GraphDatabaseFactory().newEmbeddedDatabase( databaseDirectory );

try ( Transaction tx = db.beginTx())
{
    Node myNode = db.createNode();
    myNode.setProperty( "name", "my node" );
    tx.success();
}
```

Execute a query:

```
try ( Transaction ignored = db.beginTx();
    Result result = db.execute( "MATCH (n {name: 'my node'}) RETURN n, n.name" ) )
{
    while ( result.hasNext() )
    {
        Map<String,Object> row = result.next();
        for ( Entry<String,Object> column : row.entrySet() )
        {
            rows += column.getKey() + ": " + column.getValue() + "; ";
        }
        rows += "\n";
    }
}
```

In the above example, you can also see how to iterate over the rows of the `org.neo4j.graphdb.Result`.

The code will generate:

```
n: Node[0]; n.name: my node;
```

> When using an `Result`, you should consume the entire result (iterate over all rows using `next()`, iterating over the iterator from `columnAs()` or calling for example `resultAsString()`). Failing to do so will not properly clean up resources used by the `Result` object, leading to unwanted behavior, such as leaking transactions. In case you do not want to iterate over all of the results, make sure to invoke `close()` as soon as you are done, to release the resources tied to the result.

The recommended way to handle results is to use a try-with-resources statement. This will make sure that the result is closed at the end of the statement.

You can also get a list of the columns in the result like this:

```
List<String> columns = result.columns();
```

This gives you:

```
[n, n.name]
```

Use the following to fetch the result items from a single column. In this case, you will have to read the property from the node, and not from the result:

```
Iterator<Node> n_column = result.columnAs( "n" );
for ( Node node : Iterators.asIterable( n_column ) )
{
    nodeResult = node + ": " + node.getProperty( "name" );
}
```

In this case, there is only one node in the result:

```
Node[0]: my node
```

Only use this if the result only contains a single column, or you are only interested in a single column of the result.

> `resultAsString()`, `writeAsStringTo()`, `columnAs()` cannot be called more than once on the same `Result` object, as they consume the result. In the same way, part of the result gets consumed for every call to `next()`. You should instead use only one and if you need the facilities of the other methods on the same query result instead create a new `Result`.

For more information on the Java interface to Cypher, see the Neo4j Javadocs.

For more information and examples for Cypher, see Neo4j Cypher Manual.

Query parameters

Here are some examples for query parameters. The examples below illustrate how to use parameters when executing Cypher queries from Java.

For more information on parameters see the Neo4j Cypher Manual.

Node ID:

```java
Map<String, Object> params = new HashMap<>();
params.put( "id", 0 );
String query = "MATCH (n) WHERE id(n) = $id RETURN n.name";
Result result = db.execute( query, params );
```

Node object:

```java
Map<String, Object> params = new HashMap<>();
params.put( "node", bobNode );
String query = "MATCH (n:Person) WHERE n = $node RETURN n.name";
Result result = db.execute( query, params );
```

Multiple node IDs:

```java
Map<String, Object> params = new HashMap<>();
params.put( "ids", Arrays.asList( 0, 1, 2 ) );
String query = "MATCH (n) WHERE id(n) IN $ids RETURN n.name";
Result result = db.execute( query, params );
```

String literal:

```java
Map<String, Object> params = new HashMap<>();
params.put( "name", "Johan" );
String query = "MATCH (n:Person) WHERE n.name = $name RETURN n";
Result result = db.execute( query, params );
```

Index value:

```java
Map<String, Object> params = new HashMap<>();
params.put( "value", "Michaela" );
String query = "START n=node:people(name = $value) RETURN n";
Result result = db.execute( query, params );
```

Index query:

```java
Map<String, Object> params = new HashMap<>();
params.put( "query", "name:Bob" );
String query = "START n=node:people($query) RETURN n";
Result result = db.execute( query, params );
```

Numeric parameters for SKIP and LIMIT:

```java
Map<String, Object> params = new HashMap<>();
params.put( "s", 1 );
params.put( "l", 1 );
String query = "MATCH (n:Person) RETURN n.name SKIP $s LIMIT $l";
Result result = db.execute( query, params );
```

Regular expression:

```java
Map<String, Object> params = new HashMap<>();
params.put( "regex", ".*h.*" );
String query = "MATCH (n:Person) WHERE n.name =~ $regex RETURN n.name";
Result result = db.execute( query, params );
```

Create node with properties:

```java
Map<String, Object> props = new HashMap<>();
props.put( "name", "Andy" );
props.put( "position", "Developer" );

Map<String, Object> params = new HashMap<>();
params.put( "props", props );
String query = "CREATE ($props)";
db.execute( query, params );
```

Create multiple nodes with properties:

```java
Map<String, Object> n1 = new HashMap<>();
n1.put( "name", "Andy" );
n1.put( "position", "Developer" );
n1.put( "awesome", true );

Map<String, Object> n2 = new HashMap<>();
n2.put( "name", "Michael" );
n2.put( "position", "Developer" );
n2.put( "children", 3 );

Map<String, Object> params = new HashMap<>();
List<Map<String, Object>> maps = Arrays.asList( n1, n2 );
params.put( "props", maps );
String query = "UNWIND $props AS properties CREATE (n:Person) SET n = properties RETURN n";
db.execute( query, params );
```

Setting all properties on node:

```java
Map<String, Object> n1 = new HashMap<>();
n1.put( "name", "Andy" );
n1.put( "position", "Developer" );

Map<String, Object> params = new HashMap<>();
params.put( "props", n1 );

String query = "MATCH (n:Person) WHERE n.name='Michaela' SET n = $props";
db.execute( query, params );
```

Online Backup from Java Deprecated

In order to programmatically backup your data, full or subsequently incremental, from a JVM-based program you need to write Java code like the following:

```java
OnlineBackup backup = OnlineBackup.from( "127.0.0.1" );
backup.backup( backupPath );
assertTrue( "Should be consistent", backup.isConsistent() );
backup.backup( backupPath );
```

For more information, please see the Javadocs for `org.neo4j.backup.OnlineBackup`.

The traversal framework

This section provides an overview of the concepts of the traversal framework, and a detailed description of the Neo4j traversal framework Java API.

The Neo4j Traversal framework Java API is a callback-based, lazily-executed way of specifying desired movements through a graph in Java. Some traversal examples are collected under Traversal.

You can also use the Cypher query language as a powerful declarative way to query the graph.

Main concepts

Below is a short explanation of all different methods that can modify or add to a traversal description.

- *Pathexpanders* — define what to traverse, typically in terms of relationship direction and type.

- *Order* — for example depth-first or breadth-first.

- *Uniqueness* — visit nodes (relationships, paths) only once.

- *Evaluator* — decide what to return and whether to stop or continue traversal beyond the current position.

- *Starting nodes* where the traversal will begin.
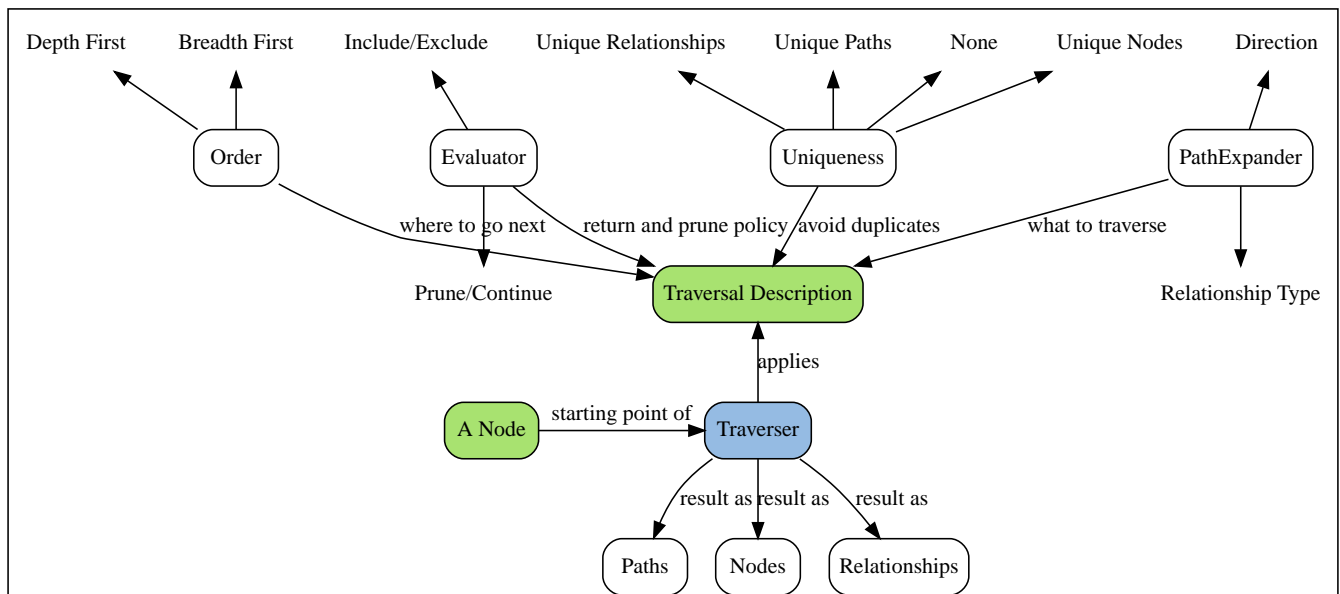
Traversal framework Java API

The traversal framework consists of a few main interfaces in addition to `Node` and `Relationship`: `TraversalDescription`, `Evaluator`, `Traverser` and `Uniqueness` are the main ones. The `Path` interface also has a special purpose in traversals, since it is used to represent a position in the graph when evaluating that position. Furthermore the `PathExpander` (replacing `RelationshipExpander` and `Expander`) interface is central to traversals, but users of the API rarely need to implement it. There are also a set of interfaces for advanced use, when explicit control over the traversal order is required: `BranchSelector`, `BranchOrderingPolicy` and `TraversalBranch`.

TraversalDescription

The `org.neo4j.graphdb.traversal.TraversalDescription` is the main interface used for defining and initializing traversals. It is not meant to be implemented by users of the traversal framework, but rather to be provided by the implementation of the traversal framework as a way for the user to describe traversals. `TraversalDescription` instances are immutable and its methods returns a new `TraversalDescription` that is modified compared to the object the method was invoked on with the arguments of the method.

Relationships

Adds a relationship type to the list of relationship types to traverse. By default this list is empty and it means that it will traverse *all relationships*, regardless of type. If one or more relationships are added to this list *only the added* types will be traversed. There are two methods, one `including direction` and another one `excluding direction`, where the latter traverses relationships in `both directions`.

Evaluator

The `org.neo4j.graphdb.traversal.Evaluator` is used for deciding, at each position (represented as a `Path`) whether the traversal should continue, and/or whether the node should be included in the result.

Given a `Path`, it asks for one of four actions for that branch of the traversal:

- `Evaluation.INCLUDE_AND_CONTINUE`: Include this node in the result and continue the traversal.
- `Evaluation.INCLUDE_AND_PRUNE`: Include this node in the result, but do not continue the traversal.
- `Evaluation.EXCLUDE_AND_CONTINUE`: Exclude this node from the result, but continue the traversal.
- `Evaluation.EXCLUDE_AND_PRUNE`: Exclude this node from the result and do not continue the traversal.

More than one evaluator can be added.

> **i** Evaluators will be called for all positions the traverser encounters, even for the start node.

Traverser

The `org.neo4j.graphdb.traversal.Traverser` object is the result of invoking `traverse()` of a `org.neo4j.graphdb.traversal.TraversalDescription` object. It represents a traversal positioned in the graph, and a specification of the format of the result. The actual traversal is performed lazily each time the `next()`-method of the iterator of the `Traverser` is invoked.

Uniqueness

Sets the rules for how positions can be revisited during a traversal as stated in `org.neo4j.graphdb.traversal.Uniqueness`. Default if not set is `NODE_GLOBAL`.

A Uniqueness can be supplied to the `TraversalDescription` to dictate under what circumstances a traversal may revisit the same position in the graph. The various uniqueness levels that can be used in Neo4j are:

- `NONE`: Any position in the graph may be revisited.

- `NODE_GLOBAL` uniqueness: No node in the entire graph may be visited more than once. This could potentially consume a lot of memory since it requires keeping an in-memory data structure remembering all the visited nodes.

- `RELATIONSHIP_GLOBAL` uniqueness: no relationship in the entire graph may be visited more than once. Just like `NODE_GLOBAL` uniqueness, this could potentially use up a lot of memory. But since graphs typically have a larger number of relationships than nodes, the memory overhead of this uniqueness level could grow even quicker.

- `NODE_PATH` uniqueness: A node may not occur previously in the path reaching up to it.

- `RELATIONSHIP_PATH` uniqueness: A relationship may not occur previously in the path reaching up to it.

- `NODE_RECENT` uniqueness: Similar to `NODE_GLOBAL` uniqueness in that there is a global collection of visited nodes each position is checked against. This uniqueness level does however have a cap on how much memory it may consume in the form of a collection that only contains the most recently visited nodes. The size of this collection can be specified by providing a number as the second argument to the TraversalDescription.uniqueness()-method along with the uniqueness level.

- `RELATIONSHIP_RECENT` uniqueness: Works like `NODE_RECENT` uniqueness, but with relationships instead of nodes.

Depth first / Breadth first

These are convenience methods for setting preorder depth-first / breadth-first `BranchSelector` | `ordering` policies. The same result can be achieved by calling the `order` method with ordering policies from `org.neo4j.graphdb.traversal.BranchOrderingPolicies`, or to write your own `BranchSelector` / `BranchOrderingPolicy` and pass in.

Order - how to move through branches?

This is a more generic version of depthFirst/ breadthFirst methods in that it enables an arbitrary `org.neo4j.graphdb.traversal.BranchOrderingPolicy` to be injected into the description.

BranchSelector

A `BranchSelector` / `BranchOrderingPolicy` is used for selecting which branch of the traversal to attempt next. This is used for implementing traversal orderings. The traversal framework provides a few basic ordering implementations:

- `BranchOrderingPolicies.PREORDER_DEPTH_FIRST`: Traversing depth first, visiting each node before visiting its child nodes.

- `BranchOrderingPolicies.POSTORDER_DEPTH_FIRST`: Traversing depth first, visiting each node after visiting its child nodes.

- `BranchOrderingPolicies.PREORDER_BREADTH_FIRST`: Traversing breadth first, visiting each node before visiting its child nodes.

- `BranchOrderingPolicies.POSTORDER_BREADTH_FIRST`: Traversing breadth first, visiting each node after visiting its child nodes.

> **ℹ** Breadth-first traversals have a higher memory overhead than depth-first traversals.

A `BranchSelector` carries state and hence needs to be uniquely instantiated for each traversal. Therefore it is supplied to the `TraversalDescription` through a `BranchOrderingPolicy` interface, which is a factory of `BranchSelector` instances.

A user of the Traversal framework rarely needs to implement his own `BranchSelector` or `BranchOrderingPolicy`, it is provided to let graph algorithm implementors provide their own traversal orders. The Neo4j Graph Algorithms package contains for example a `BestFirst` order `BranchSelector` / `BranchOrderingPolicy` that is used in BestFirst search algorithms such as A* and Dijkstra.

BranchOrderingPolicy

A factory for creating `BranchSelector`s to decide in what order branches are returned (where a branch's position is represented as a `Path` from the start node to the current node). Common policies are `depth-first` and `breadth-first` and that is why there are convenience methods for those.

For example, calling `TraversalDescription#depthFirst()` is equivalent to:

```
description.order( BranchOrderingPolicies.PREORDER_DEPTH_FIRST );
```

TraversalBranch

An object used by the BranchSelector to get more branches from a certain branch. In essence these are a composite of a Path and a RelationshipExpander that can be used to get a new `org.neo4j.graphdb.traversal.TraversalBranch` from the current one.

Path

A `org.neo4j.graphdb.Path` is a general interface that is part of the Neo4j API. In the traversal API of Neo4j the use of Paths are twofold. Traversers can return their results in the form of the Paths of the visited positions in the graph that are marked for being returned. Path objects are also used in the evaluation of positions in the graph, for determining if the traversal should continue from a certain point or not, and whether a certain position should be included in the result set or not.

PathExpander / RelationshipExpander

The traversal framework use the `PathExpander` (replacing `RelationshipExpander`) to discover the relationships that should be followed from a particular path to further branches in the traversal.

Expander

This is a more generic version of relationships where a `RelationshipExpander` is injected, defining all relationships to be traversed for any given node.

The `Expander` interface is an extension of the `RelationshipExpander` interface that makes it possible to build customized versions of an `Expander`. The implementation of `TraversalDescription` uses this to

provide methods for defining which relationship types to traverse, this is the usual way a user of the API would define a `RelationshipExpander` — by building it internally in the `TraversalDescription`.

All the RelationshipExpanders provided by the Neo4j traversal framework also implement the Expander interface. For a user of the traversal API it is easier to implement the PathExpander/RelationshipExpander interface, since it only contains one method — the method for getting the relationships from a path/node, the methods that the Expander interface adds are just for building new Expanders.

How to use the Traversal framework

A `org.neo4j.graphdb.traversal.TraversalDescription` is built using a fluent interface and such a description can then spawn several `org.neo4j.graphdb.traversal.Traverser` objects.

> ℹ️ The source code for the examples can be found here: `TraversalExample.java`.

With the definition of the RelationshipTypes as:

```
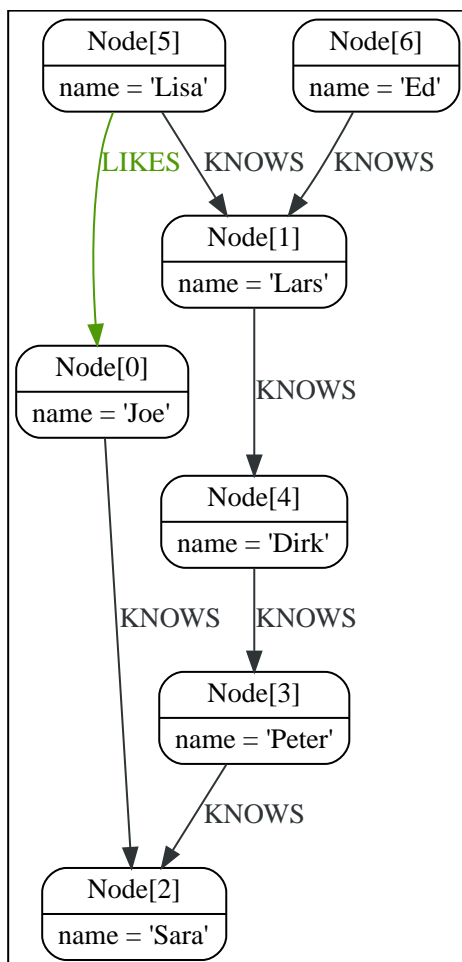private enum Rels implements RelationshipType
{
    LIKES, KNOWS
}
```

The graph can be traversed with for example the following traverser, starting at the node with the `name = 'Joe'`:

```
for ( Path position : db.traversalDescription()
        .depthFirst()
        .relationships( Rels.KNOWS )
        .relationships( Rels.LIKES, Direction.INCOMING )
        .evaluator( Evaluators.toDepth( 5 ) )
        .traverse( node ) )
{
    output += position + "\n";
}
```

The traversal will output:

```
(0)
(0)<-[LIKES,1]-(5)
(0)<-[LIKES,1]-(5)-[KNOWS,6]->(1)
(0)<-[LIKES,1]-(5)-[KNOWS,6]->(1)<-[KNOWS,5]-(6)
(0)<-[LIKES,1]-(5)-[KNOWS,6]->(1)-[KNOWS,4]->(4)
(0)<-[LIKES,1]-(5)-[KNOWS,6]->(1)-[KNOWS,4]->(4)-[KNOWS,3]->(3)
(0)<-[LIKES,1]-(5)-[KNOWS,6]->(1)-[KNOWS,4]->(4)-[KNOWS,3]->(3)-[KNOWS,2]->(2)
```

Since a `TraversalDescription` is immutable it is also useful to create template descriptions which holds common settings shared by different traversals. For example, start with this traverser:

```
friendsTraversal = db.traversalDescription()
        .depthFirst()
        .relationships( Rels.KNOWS )
        .uniqueness( Uniqueness.RELATIONSHIP_GLOBAL );
```

This traverser would yield the following output (starting from the node with the `name = 'Joe'`):

```
(0)
(0)-[KNOWS,0]->(2)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)<-[KNOWS,4]-(1)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)<-[KNOWS,4]-(1)<-[KNOWS,6]-(5)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)<-[KNOWS,4]-(1)<-[KNOWS,5]-(6)
```

Create a new traverser from it, restricting depth to three:

```
for ( Path path : friendsTraversal
        .evaluator( Evaluators.toDepth( 3 ) )
        .traverse( node ) )
{
    output += path + "\n";
}
```

This will give the following output:

```
(0)
(0)-[KNOWS,0]->(2)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)
```

Or how about from depth two to four? That is done like this:

```
for ( Path path : friendsTraversal
        .evaluator( Evaluators.fromDepth( 2 ) )
        .evaluator( Evaluators.toDepth( 4 ) )
        .traverse( node ) )
{
    output += path + "\n";
}
```

This will give the following output:

```
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)
(0)-[KNOWS,0]->(2)<-[KNOWS,2]-(3)<-[KNOWS,3]-(4)<-[KNOWS,4]-(1)
```

For various useful evaluators, see the `org.neo4j.graphdb.traversal.Evaluators` Java API or simply implement the `org.neo4j.graphdb.traversal.Evaluator` interface yourself.

If you are not interested in the Paths, but the Nodes you can transform the traverser into an iterable of nodes like this:

```
for ( Node currentNode : friendsTraversal
        .traverse( node )
        .nodes() )
{
    output += currentNode.getProperty( "name" ) + "\n";
}
```

This will give the following output:

```
Joe
Sara
Peter
Dirk
Lars
Lisa
Ed
```

Relationships are fine as well, here is an example how to get them:

```
for ( Relationship relationship : friendsTraversal
        .traverse( node )
        .relationships() )
{
    output += relationship.getType().name() + "\n";
}
```

```
KNOWS
KNOWS
KNOWS
KNOWS
KNOWS
KNOWS
```

Transaction management

This topic describes transactional management and behavior.

In order to fully maintain data integrity and ensure good transactional behavior, Neo4j DBMS supports the ACID properties:

- Atomicity — If any part of a transaction fails, the database state is left unchanged.

- Consistency — Any transaction will leave the database in a consistent state.

- Isolation — During a transaction, modified data cannot be accessed by other operations.

- Durability — The DBMS can always recover the results of a committed transaction.

Specifically:

- All database operations that access the graph, indexes, or the schema must be performed in a transaction.

- The default isolation level is *read-committed isolation level*.

- Data retrieved by traversals is not protected from modification by other transactions.

- Non-repeatable reads may occur (i.e., only write locks are acquired and held until the end of the transaction).

- One can manually acquire write locks on nodes and relationships to achieve higher level of isolation — *serialization isolation level*.

- Locks are acquired at the Node and Relationship level.

- Deadlock detection is built into the core transaction management.

Interaction cycle

There are database operations that must be performed in a transaction to ensure the ACID-properties. Specifically operations that access the graph, indexes, or the schema are such operations. Transactions are single-threaded, confined, and independent. Multiple transactions can be started in a single thread and they will be independent from each other.

The interaction cycle of working with transactions looks like this:

1. Begin a transaction.

2. Perform database operations.

3. Commit or roll back the transaction.

> ℹ️ It is very important to finish each transaction. The transaction will not release the locks or memory it has acquired until it has been finished.

The idiomatic use of transactions in Neo4j is to use a `try-with-resources` statement and declare `transaction` as one of the resources. Then start the transaction and try to perform graph operations. The last operation in the `try` block should commit or roll back the transaction, depending on the business logic.

In this scenario, `try-with-resources` is used as a guard against unexpected exceptions and as an additional safety mechanism to ensure that the transaction gets closed no matter what happens inside the statement block. All non-committed transactions will be rolled back as part of resource cleanup at the end of the statement. In case a transaction has been explicitly committed or rolled back, resource cleanup will not be required and the transaction closure will be an empty operation.

> **ℹ** All modifications performed in a transaction are kept in memory. This means that very large updates must be split into several transactions in order to avoid running out of memory.

Isolation levels

Transactions in Neo4j use a *read-committed isolation level*, which means they will see data as soon as it has been committed but will not see data in other transactions that have not yet been committed. This type of isolation is weaker than serialization but offers significant performance advantages while being sufficient for the overwhelming majority of cases.

In addition, the Neo4j Java API enables explicit locking of nodes and relationships. Using locks gives the opportunity to simulate the effects of higher levels of isolation by obtaining and releasing locks explicitly. For example, if a write lock is taken on a common node or relationship, then all transactions will serialize on that lock — giving the effect of a *serialization isolation level*.

Lost updates in Cypher

In Cypher it is possible to acquire write locks to simulate improved isolation in some cases. Consider the case where multiple concurrent Cypher queries increment the value of a property. Due to the limitations of the *read-committed isolation level*, the increments might not result in a deterministic final value. If there is a direct dependency, Cypher will automatically acquire a write lock before reading. A direct dependency is when the right-hand side of a `SET` has a dependent property read in the expression, or in the value of a key-value pair in a literal map.

For example, the following query, if run by one hundred concurrent clients, will very likely not increment the property `n.prop` to 100, unless a write lock is acquired before reading the property value. This is because all queries would read the value of `n.prop` within their own transaction, and would not see the incremented value from any other transaction that has not yet committed. In the worst case scenario the final value would be as low as 1, if all threads perform the read before any has committed their transaction.

Example 2. Cypher can acquire a write lock

The following example requires a write lock, and Cypher automatically acquires one:

```
MATCH (n:Example {id: 42})
SET n.prop = n.prop + 1
```

Example 3. Cypher can acquire a write lock

This example also requires a write lock, and Cypher automatically acquires one:

```
MATCH (n)
SET n += {prop: n.prop + 1}
```

Due to the complexity of determining such a dependency in the general case, Cypher does not cover any of the example cases below.

Example 4. Complex Cypher

Variable depending on results from reading the property in an earlier statement:

```
MATCH (n)
WITH n.prop AS p
// ... operations depending on p, producing k
SET n.prop = k + 1
```

Example 5. Complex Cypher

Circular dependency between properties read and written in the same query:

```
MATCH (n)
SET n += {propA: n.propB + 1, propB: n.propA + 1}
```

To ensure deterministic behavior also in the more complex cases, it is necessary to explicitly acquire a write lock on the node in question. In Cypher there is no explicit support for this, but it is possible to work around this limitation by writing to a temporary property.

Example 6. Explicitly acquire a write lock

This example acquires a write lock for the node by writing to a dummy property before reading the requested value:

```
MATCH (n:Example {id: 42})
SET n._LOCK_ = true
WITH n.prop AS p
// ... operations depending on p, producing k
SET n.prop = k + 1
REMOVE n._LOCK_
```

The existence of the `SET n._LOCK_` statement before the read of the `n.prop` read ensures the lock is acquired before the read action, and no updates will be lost due to enforced serialization of all concurrent queries on that specific node.

Default locking behavior

- When adding, changing or removing a property on a node or relationship a write lock will be taken on the specific node or relationship.

- When creating or deleting a node a write lock will be taken for the specific node.

- When creating or deleting a relationship a write lock will be taken on the specific relationship and both its nodes.

The locks will be added to the transaction and released when the transaction finishes.

Deadlocks

Since locks are used it is possible for deadlocks to happen. Neo4j will however detect any deadlock (caused by acquiring a lock) before they happen and throw an exception. The transaction is marked for rollback before the exception is thrown. All locks acquired by the transaction will still be held but will be released when the transaction is finished (in the finally block as pointed out earlier). Once the locks are released, other transactions that were waiting for locks held by the transaction causing the deadlock, can proceed. The work performed by the transaction causing the deadlock can then be retried by the user if needed.

Experiencing frequent deadlocks is an indication of concurrent write requests happening in such a way that it is not possible to execute them while at the same time live up to the intended isolation and consistency. The solution is to make sure concurrent updates happen in a reasonable way. For example, given two specific nodes (A and B), adding or deleting relationships to both these nodes in random order for each transaction, will result in deadlocks when there are two or more transactions doing that concurrently. One option is to make sure that updates always happens in the same order (first A then B). Another option is to make sure that each thread/transaction does not have any conflicting writes to a node or relationship as some other concurrent transaction. This can, for example, be achieved by letting a single thread do all updates of a specific type.

> ⓘ Deadlocks caused by the use of other synchronization than the locks managed by Neo4j can still happen. Since all operations in the Neo4j API are thread safe unless specified otherwise, there is no need for external synchronization. Other code that requires synchronization should be synchronized in such a way that it never performs any Neo4j operation in the synchronized block.

Deadlock handling an example

Below, you will find an example of how deadlocks can be handled in procedures, server extensions, or when using Neo4j embedded.

> 💡 The full source code used for the code snippet can be found at DeadlockDocTest.java.

When dealing with deadlocks in code, there are several issues you may want to address:

- Only do a limited amount of retries, and fail if a threshold is reached.

- Pause between each attempt to allow the other transaction to finish before trying again.

- A retry-loop can be useful not only for deadlocks, but for other types of transient errors as well.

Below is an example that shows how this can be implemented.

Example 7. Handling deadlocks using TransactionTemplate

If you do not want to write all the code yourself, there is a class called `org.neo4j.helpers.TransactionTemplate` that will help you achieve what is needed. Below is an example of how to create, customize, and use this template for retries in transactions.

First, define the base template:

```
TransactionTemplate template = new TransactionTemplate(  ).retries( 5 ).backoff( 3, TimeUnit.SECONDS
);
```

Next, specify the database to use and a function to execute:

```
Object result = template.with(graphDatabaseService).execute( transaction -> {
    Object result1 = null;
    return result1;
} );
```

The operations that could lead to a deadlock should go into the `apply` method.

The `TransactionTemplate` uses a fluent API for configuration, and you can choose whether to set everything at once, or (as in the example) provide some details just before using it. The template enables setting a predicate for what exceptions to retry on, and also allows for monitoring of events that take place.

Example 8. Handling deadlocks using a retry loop

This example shows how to use a retry loop for handling deadlocks:

```java
Throwable txEx = null;
int RETRIES = 5;
int BACKOFF = 3000;
for ( int i = 0; i < RETRIES; i++ )
{
    try ( Transaction tx = graphDatabaseService.beginTx() )
    {
        Object result = doStuff(tx);
        tx.success();
        return result;
    }
    catch ( Throwable ex )
    {
        txEx = ex;

        // Add whatever exceptions to retry on here
        if ( !(ex instanceof DeadlockDetectedException) )
        {
            break;
        }
    }

    // Wait so that we don't immediately get into the same deadlock
    if ( i < RETRIES - 1 )
    {
        try
        {
            Thread.sleep( BACKOFF );
        }
        catch ( InterruptedException e )
        {
            throw new TransactionFailureException( "Interrupted", e );
        }
    }
}

if ( txEx instanceof TransactionFailureException )
{
    throw ((TransactionFailureException) txEx);
}
else if ( txEx instanceof Error )
{
    throw ((Error) txEx);
}
else if ( txEx instanceof RuntimeException )
{
    throw ((RuntimeException) txEx);
}
else
{
    throw new TransactionFailureException( "Failed", txEx );
}
```

Delete semantics

When deleting a node or a relationship all properties for that entity will be automatically removed but the relationships of a node will not be removed. Neo4j enforces a constraint (upon commit) that all relationships must have a valid start node and end node. In effect, this means that trying to delete a node that still has relationships attached to it will throw an exception upon commit. It is, however, possible to choose in which order to delete the node and the attached relationships as long as no relationships exist when the transaction is committed.

The delete semantics can be summarized as follows:

- All properties of a node or relationship will be removed when it is deleted.

- A deleted node cannot have any attached relationships when the transaction commits.

- It is possible to acquire a reference to a deleted relationship or node that has not yet been committed.

- Any write operation on a node or relationship after it has been deleted (but not yet committed) will throw an exception.

- Trying to acquire a new or work with an old reference to a deleted node or relationship after commit, will throw an exception.

Creating unique nodes

In many use cases, a certain level of uniqueness is desired among entities. For example, only one user with a certain email address may exist in a system. If multiple concurrent threads naively try to create the user, duplicates will be created.

The following are the main strategies for ensuring uniqueness, and they all work across cluster and single-instance deployments.

Single thread

By using a single thread, no two threads will even try to create a particular entity simultaneously. On cluster an external single-threaded client can perform the operations.

Get or create

Defining a uniqueness constraint and using the Cypher `MERGE` clause is the most efficient way to get or create a unique node. See Unique nodes for more information.

Transaction events

A transaction event handler can be registered to receive Neo4j transaction events. Once it has been registered at a `GraphDatabaseService` instance it receives events for transactions before they are committed. Handlers get notified about transactions that have performed any write operation, and that will be committed. If `Transaction#success()` has not been called or the transaction has been marked as failed `Transaction#failure()` it will be rolled back, and no events are sent to the Handler.

Before a transaction is committed the Handler's `beforeCommit` method is called with the entire diff of modifications made in the transaction. At this point the transaction is still running, so changes can still be made. The method may also throw an exception, which will prevent the transaction from being committed. If the transaction is rolled back, a call to the handler's `afterRollback` method will follow.

> ℹ️ The order in which handlers are executed is undefined -- there is no guarantee that changes made by one handler will be seen by other handlers.

If `beforeCommit` is successfully executed in all registered handlers the transaction is committed and the

`afterCommit` method is called with the same transaction data. This call also includes the object returned from `beforeCommit`.

In `afterCommit` the transaction has been closed and access to anything outside `TransactionData` requires a new transaction to be opened. A `TransactionEventHandler` gets notified about transactions that have any changes accessible via `TransactionData` so some indexing and schema changes will not be triggering these events.

JMX metrics

Neo4j provides different levels of monitoring facilities in order to supply a continuous overview of the system's health. For a description of the monitoring options, see Neo4j Operations Manual → Monitoring. Many of the metrics are exposed through JMX.

> ⚠️ Please note that the available JMX MBeans and their names will be updated in Neo4j version 4.0. Beans that duplicate metrics or monitoring options, described in Neo4j Operations Manual → Monitoring, will be removed.

Adjusting remote JMX access to Neo4j

Per default, the Neo4j Enterprise Server edition does not allow remote JMX connections, since the relevant options in the *conf/neo4j.conf* configuration file are commented out. To enable this feature, you have to remove the `#` characters from the various `com.sun.management.jmxremote` options there.

When commented in, the default values are set up to enable remote JMX connections with certain roles, refer to the *conf/jmx.password*, *conf/jmx.access*, and *conf/neo4j.conf* files for details.

Make sure that *conf/jmx.password* has the correct file permissions. The owner of the file has to be the user that will run the service, and the permissions should be read only for that user. On Unix systems, this is `0600`.

On Windows, follow the tutorial security on windows to set the correct permissions. If you are running the service under the Local System Account, the user that owns the file and has access to it should be SYSTEM.

With this setup, you should be able to connect to JMX monitoring of the Neo4j server using `<IP-OF-SERVER>:3637`, with the username `monitor` and the password `Neo4j`.

Note that it is possible that you have to update the permissions and/or ownership of the *conf/jmx.password* and *conf/jmx.access* files — refer to the relevant section in *conf/neo4j.conf* for details.

> ⚠️ For maximum security, please adjust at least the password settings in *conf/jmx.password* for a production installation.

For more details, see: management agent.

Connecting to a Neo4j instance using JMX and JConsole

First, start your Neo4j instance, for example using:

```
$NEO4j_HOME/bin/neo4j start
```

Now, start JConsole with:

```
$JAVA_HOME/bin/jconsole
```

Connect to the process running your Neo4j database instance:
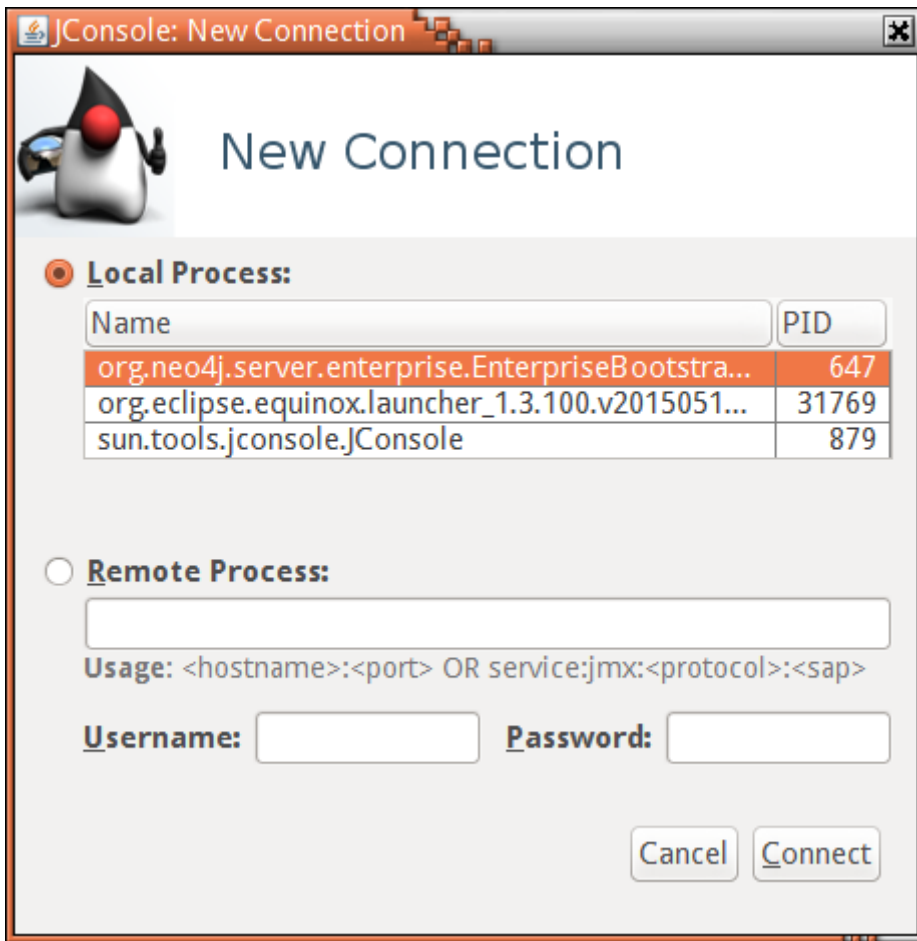
Figure 2. Connecting JConsole to the Neo4j Java process

Now, beside the MBeans exposed by the JVM, you will see an `org.neo4j` section in the MBeans tab. Under that, you will have access to all the monitoring information exposed by Neo4j.

For opening JMX to remote monitoring access, please see Adjusting remote JMX access to Neo4j and the JMX documention.
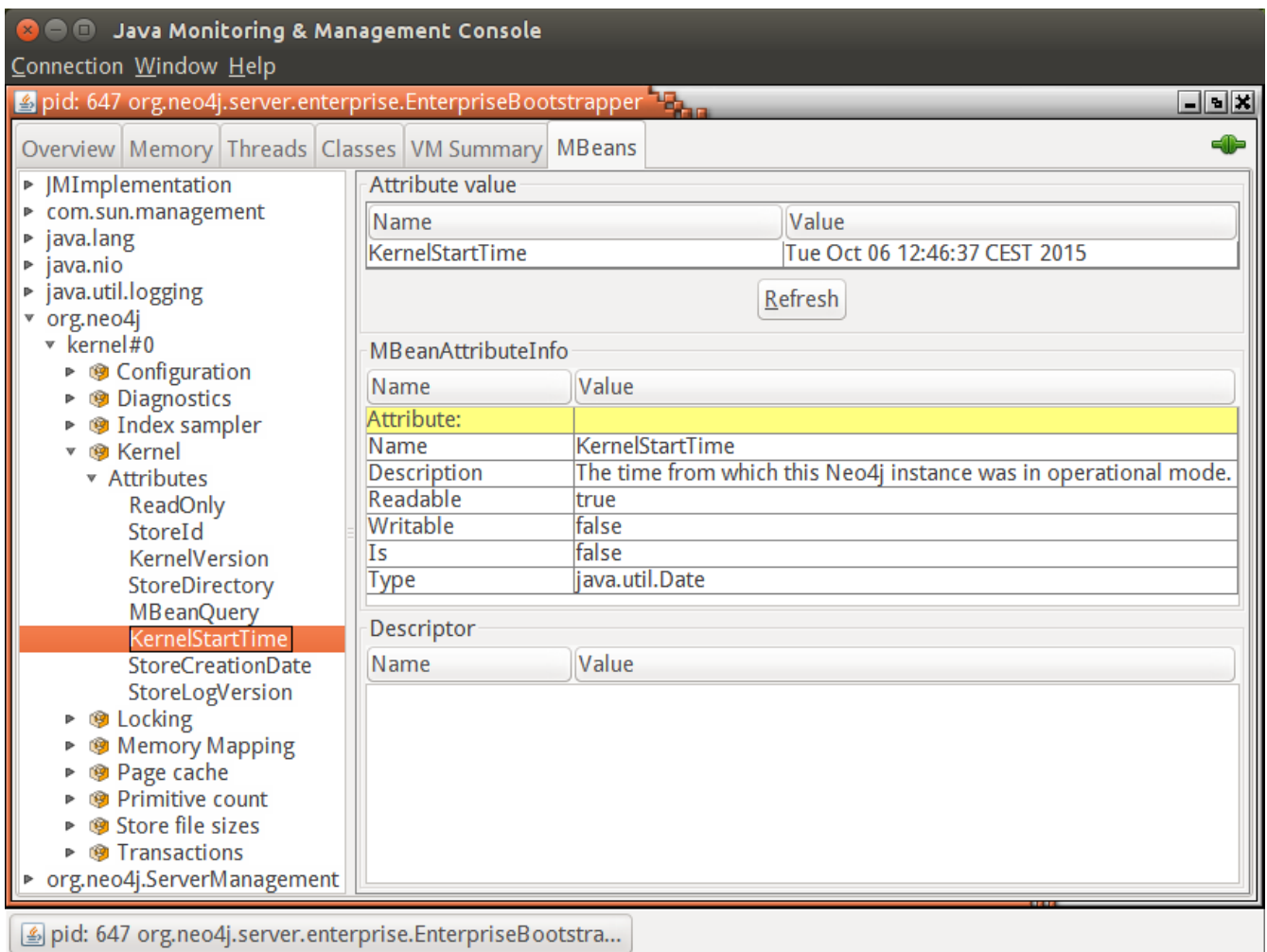
Figure 3. Neo4j MBeans View

How to connect to the JMX monitoring programmatically

In order to programmatically connect to the Neo4j JMX server, there are some convenience methods in the Neo4j Management component to help you find out the most commonly used monitoring attributes of Neo4j. See Reading a management attribute Deprecated for an example.

Once you have access to this information, you can use it to for instance expose the values to SNMP or other monitoring systems.

Reference of supported JMX MBeans

Below is a complete reference of Neo4j JMX management beans in two parts. First part shows all beans available in Neo4j Enterprise Edition when the instance is part of a Causal Cluster. The second part shows the management beans that are uniquely available when running in High Availability mode.

> ℹ️ For additional information on the primitive datatypes (`int`, `long` etc.) used in the JMX attributes, please see Getting Started → Introduction: Properties.

JMX MBeans available on instances in a Causal Cluster

The following JMX management beans are available on instances that are part of a Causal Cluster. One JMX Bean not included here is for the current configuration of the Neo4j instance. For configuration details, see Neo4j Operations Manual → Configuration settings.

Table 4. MBeans exposed by Neo4j

Name	Description
Causal Clustering	Information about an instance participating in a causal cluster.
Diagnostics	Diagnostics provided by Neo4j.
Index sampler	Handle index sampling.
Kernel	Information about the Neo4j kernel.
Locking	Information about the Neo4j lock status.
Memory Mapping	The status of Neo4j memory mapping.
Page cache	Information about the Neo4j page cache. All numbers are counts and sums since the Neo4j instance was started.
Primitive count	Estimates of the numbers of different kinds of Neo4j primitives.
Reports	Reports operations.
Store file sizes	This bean is deprecated, use StoreSize bean instead; Information about the sizes of the different parts of the Neo4j graph store.
Store sizes	Information about the disk space used by different parts of the Neo4j graph store.
Transactions	Information about the Neo4j transaction manager.

Table 5. MBean Causal Clustering (org.neo4j.management.CausalClustering) Attributes

Name	Description	Type	Read	Write
	Information about an instance participating in a causal cluster.			
`RaftLogSize`	The total amount of disk space used by the raft log, in bytes.	`long`	yes	no
`ReplicatedStateSize`	The total amount of disk space used by the replicated states, in bytes.	`long`	yes	no
`Role`	The current role this member has in the cluster.	`String`	yes	no

Table 6. MBean Diagnostics (org.neo4j.management.Diagnostics) Attributes

Name	Description	Type	Read	Write
	Diagnostics provided by Neo4j.			
`DiagnosticsProviders`	A list of the ids for the registered diagnostics providers.	`List (java.util.List)`	yes	no

Table 7. MBean Diagnostics (org.neo4j.management.Diagnostics) Operations

Name	Description	ReturnType	Signature
dumpAll	Dump diagnostics information to JMX.	String	(no parameters)
dumpToLog	Dump diagnostics information to the log.	void	(no parameters)
dumpToLog	Dump diagnostics information to the log.	void	java.lang.String
extract	Operation exposed for management.	String	java.lang.String

Table 8. MBean Index sampler (org.neo4j.management.IndexSamplingManager) Operations

Name	Description	ReturnType	Signature
triggerIndexSampling	Trigger index sampling.	void	java.lang.String,java.lang.String,boolean

Table 9. MBean Kernel (org.neo4j.jmx.Kernel) Attributes

Name	Description	Type	Read	Write
	Information about the Neo4j kernel.			
DatabaseName	The name of the mounted database.	String	yes	no
KernelStartTime	The time from which this Neo4j instance was in operational mode.	Date (java.util.Date)	yes	no
KernelVersion	The version of Neo4j.	String	yes	no
MBeanQuery	An ObjectName that can be used as a query for getting all management beans for this Neo4j instance.	javax.management.ObjectName	yes	no
ReadOnly	Whether this is a read only instance.	boolean	yes	no
StoreCreationDate	The time when this Neo4j graph store was created.	Date (java.util.Date)	yes	no
StoreId	An identifier that, together with store creation time, uniquely identifies this Neo4j graph store.	String	yes	no
StoreLogVersion	The current version of the Neo4j store logical log.	long	yes	no

Table 10. MBean Locking (org.neo4j.management.LockManager) Attributes

Name	Description	Type	Read	Write
	Information about the Neo4j lock status.			
Locks	Information about all locks held by Neo4j.	java.util.List<org.neo4j.kernel.info.LockInfo> as CompositeData[]	yes	no
NumberOfAvertedDeadlocks	The number of lock sequences that would have lead to a deadlock situation that Neo4j has detected and averted (by throwing DeadlockDetectedException).	long	yes	no

Table 11. MBean Locking (org.neo4j.management.LockManager) Operations

Name	Description	ReturnType	Signature
`getContendedLocks`	getContendedLocks	`java.util.List<org.neo4j.kernel.info.LockInfo> as CompositeData[]`	`long`

Table 12. MBean Memory Mapping (org.neo4j.management.MemoryMapping) Attributes

Name	Description	Type	Read	Write
The status of Neo4j memory mapping.				
`MemoryPools`	Get information about each pool of memory mapped regions from store files with memory mapping enabled.	`org.neo4j.management.WindowPoolInfo[] as CompositeData[]`	yes	no

Table 13. MBean Page cache (org.neo4j.management.PageCache) Attributes

Name	Description	Type	Read	Write
Information about the Neo4j page cache. All numbers are counts and sums since the Neo4j instance was started.				
`BytesRead`	Number of bytes read from durable storage.	`long`	yes	no
`BytesWritten`	Number of bytes written to durable storage.	`long`	yes	no
`EvictionExceptions`	Number of exceptions caught during page eviction. This number should be zero, or at least not growing, in a healthy database. Otherwise it could indicate drive failure, storage space, or permission problems.	`long`	yes	no
`Evictions`	Number of page evictions. How many pages have been removed from memory to make room for other pages.	`long`	yes	no
`Faults`	Number of page faults. How often requested data was not found in memory and had to be loaded.	`long`	yes	no
`FileMappings`	Number of files that have been mapped into the page cache.	`long`	yes	no
`FileUnmappings`	Number of files that have been unmapped from the page cache.	`long`	yes	no
`Flushes`	Number of page flushes. How many dirty pages have been written to durable storage.	`long`	yes	no

Name	Description	Type	Read	Write
HitRatio	Ratio of hits to the total number of lookups in the page cache.	double	yes	no
Hits	Number of page hits. How often requested data was found in memory.	long	yes	no
Pins	Number of page pins. How many pages have been accessed (monitoring must be enabled separately).	long	yes	no
Unpins	Number of page unpins. How many pages have been accessed and are not accessed anymore (monitoring must be enabled separately).	long	yes	no
UsageRatio	The percentage of used pages. Will return NaN if it cannot be determined.	double	yes	no

Table 14. MBean Primitive count (org.neo4j.jmx.Primitives) Attributes

Name	Description	Type	Read	Write
Estimates of the numbers of different kinds of Neo4j primitives.				
NumberOfNodeIdsInUse	An estimation of the number of nodes used in this Neo4j instance.	long	yes	no
NumberOfPropertyIdsInUse	An estimation of the number of properties used in this Neo4j instance.	long	yes	no
NumberOfRelationshipIdsInUse	An estimation of the number of relationships used in this Neo4j instance.	long	yes	no
NumberOfRelationshipTypeIdsInUse	The number of relationship types used in this Neo4j instance.	long	yes	no

Table 15. MBean Reports (org.neo4j.dbms.diagnostics.jmx.Reports) Attributes

Name	Description	Type	Read	Write
Reports operations.				
EnvironmentVariables	Returns a map if the current environment variables.	String	yes	no

Table 16. MBean Reports (org.neo4j.dbms.diagnostics.jmx.Reports) Operations

Name	Description	ReturnType	Signature
listTransactions	List all active transactions.	String	(no parameters)

Table 17. MBean Store file sizes (org.neo4j.jmx.StoreFile) Attributes

Name	Description	Type	Read	Write
colspan this bean is deprecated	This bean is deprecated, use StoreSize bean instead; Information about the sizes of the different parts of the Neo4j graph store.			
ArrayStoreSize	The amount of disk space used to store array properties, in bytes.	long	yes	no
LogicalLogSize	The amount of disk space used by the current Neo4j logical log, in bytes.	long	yes	no
NodeStoreSize	The amount of disk space used to store nodes, in bytes.	long	yes	no
PropertyStoreSize	The amount of disk space used to store properties (excluding string values and array values), in bytes.	long	yes	no
RelationshipStoreSize	The amount of disk space used to store relationships, in bytes.	long	yes	no
StringStoreSize	The amount of disk space used to store string properties, in bytes.	long	yes	no
TotalStoreSize	The total disk space used by this Neo4j instance, in bytes.	long	yes	no

Table 18. MBean Store sizes (org.neo4j.jmx.StoreSize) Attributes

Name	Description	Type	Read	Write
colspan info	Information about the disk space used by different parts of the Neo4j graph store.			
ArrayStoreSize	Disk space used to store array properties, in bytes.	long	yes	no
CountStoreSize	Disk space used to store counters, in bytes.	long	yes	no
IndexStoreSize	Disk space used to store all indices, in bytes.	long	yes	no
LabelStoreSize	Disk space used to store labels, in bytes.	long	yes	no
NodeStoreSize	Disk space used to store nodes, in bytes.	long	yes	no
PropertyStoreSize	Disk space used to store properties (excluding string values and array values), in bytes.	long	yes	no
RelationshipStoreSize	Disk space used to store relationships, in bytes.	long	yes	no
SchemaStoreSize	Disk space used to store schemas (index and constrain declarations), in bytes.	long	yes	no
StringStoreSize	Disk space used to store string properties, in bytes.	long	yes	no
TotalStoreSize	Disk space used by whole store, in bytes.	long	yes	no
TransactionLogsSize	Disk space used by the transaction logs, in bytes.	long	yes	no

Table 19. MBean Transactions (org.neo4j.management.TransactionManager) Attributes

Name	Description	Type	Read	Write
	Information about the Neo4j transaction manager.			
LastCommittedTxId	The ID of the latest committed transaction.	long	yes	no
NumberOfCommittedTransactions	The total number of committed transactions.	long	yes	no
NumberOfOpenedTransactions	The total number started transactions.	long	yes	no
NumberOfOpenTransactions	The number of currently open transactions.	long	yes	no
NumberOfRolledBackTransactions	The total number of rolled back transactions.	long	yes	no
PeakNumberOfConcurrentTransactions	The highest number of transactions ever opened concurrently.	long	yes	no

High Availability JMX MBeans

The following JMX management beans are unique to instances that are part of a High Availability cluster.

Table 20. MBeans exposed by Neo4j in High Availability mode

Name	Description
Branched Store	Information about the branched stores present in this HA cluster member.
High Availability	Information about an instance participating in a HA cluster.

Table 21. MBean Branched Store (org.neo4j.management.BranchedStore) Attributes

Name	Description	Type	Read	Write
	Information about the branched stores present in this HA cluster member.			
BranchedStores	A list of the branched stores.	org.neo4j.management.BranchedStoreInfo[] as CompositeData[]	yes	no

Table 22. MBean High Availability (org.neo4j.management.HighAvailability) Attributes

Name	Description	Type	Read	Write
	Information about an instance participating in a HA cluster.			
Alive	Whether this instance is alive or not.	boolean	yes	no
Available	Whether this instance is available or not.	boolean	yes	no
InstanceId	The identifier used to identify this server in the HA cluster.	String	yes	no
InstancesInCluster	Information about all instances in this cluster.	org.neo4j.management.ClusterMemberInfo[] as CompositeData[]	yes	no
LastCommittedTxId	The latest transaction id present in this instance's store.	long	yes	no

Name	Description	Type	Read	Write
LastUpdateTime	The time when the data on this instance was last updated from the master.	String	yes	no
Role	The role this instance has in the cluster.	String	yes	no

Table 23. MBean High Availability (org.neo4j.management.HighAvailability) Operations

Name	Description	ReturnType	Signature
update	(If this is a follower) Update the database on this instance with the latest transactions from the leader.	String	(no parameters)

Explicit indexing Deprecated

⚠️	Explicit indexing in Neo4j is deprecated, and will be removed in the next major release. Consider configuring indexes to support full-text search instead. See Cypher Manual → Indexes to support full-text search.

ℹ️	The functionality described here has been removed in **Neo4j 4.0**.

Explicit indexing operations are part of the Neo4j index API.

Each index is tied to a unique, user-specified name (for example "first_name" or "books") and can index either `org.neo4j.graphdb.Node` or `org.neo4j.graphdb.Relationship`.

The default index implementation is provided by the `neo4j-lucene-index` component, which is included in the standard Neo4j download. It can also be downloaded separately from https://repo1.maven.org/maven2/org/neo4j/neo4j-lucene-index/. For Maven users, the `neo4j-lucene-index` component has the coordinates `org.neo4j:neo4j-lucene-index` and should be used with the same version of `org.neo4j:neo4j-kernel`. Different versions of the index and kernel components are not compatible in the general case. Both components are included transitively by the `org.neo4j:neo4j:pom` artifact, which makes it possible to keep the versions in sync.

ℹ️	*Transactions* All modifying index operations must be performed inside a transaction, as with any modifying operation in Neo4j.

Create

An index is created if it does not exist when you ask for it. Unless you give it a custom configuration, it will be created with default configuration and backend.

To set the stage for our examples, we can create some indexes to begin with:

```
IndexManager index = graphDb.index();
Index<Node> actors = index.forNodes( "actors" );
Index<Node> movies = index.forNodes( "movies" );
RelationshipIndex roles = index.forRelationships( "roles" );
```

This will create two node indexes and one relationship index with default configuration. See Relationship indexes for more information specific to relationship indexes.

See Configuration and full-text indexes for how to create *full*-text indexes.

You can also check if an index exists like this:

```
IndexManager index = graphDb.index();
boolean indexExists = index.existsForNodes( "actors" );
```

Delete

Indexes can be deleted. When deleting, the entire contents of the index will be removed as well as its associated configuration. An index can be created with the same name at a later point in time.

```java
IndexManager index = graphDb.index();
Index<Node> actors = index.forNodes( "actors" );
actors.delete();
```

Note that the actual deletion of the index is made during the commit of *the surrounding transaction*. Calls made to such an index instance after `delete()` has been called are invalid inside that transaction as well as outside (if the transaction is successful), but will become valid again if the transaction is rolled back.

Add

Each index supports associating any number of key-value pairs with any number of entities (nodes or relationships), where each association between entity and key-value pair is performed individually. To begin with, we can add a few nodes to the indexes:

```java
// Actors
Node reeves = graphDb.createNode();
reeves.setProperty( "name", "Keanu Reeves" );
actors.add( reeves, "name", reeves.getProperty( "name" ) );
Node bellucci = graphDb.createNode();
bellucci.setProperty( "name", "Monica Bellucci" );
actors.add( bellucci, "name", bellucci.getProperty( "name" ) );
// multiple values for a field, in this case for search only
// and not stored as a property.
actors.add( bellucci, "name", "La Bellucci" );
// Movies
Node theMatrix = graphDb.createNode();
theMatrix.setProperty( "title", "The Matrix" );
theMatrix.setProperty( "year", 1999 );
movies.add( theMatrix, "title", theMatrix.getProperty( "title" ) );
movies.add( theMatrix, "year", theMatrix.getProperty( "year" ) );
Node theMatrixReloaded = graphDb.createNode();
theMatrixReloaded.setProperty( "title", "The Matrix Reloaded" );
theMatrixReloaded.setProperty( "year", 2003 );
movies.add( theMatrixReloaded, "title", theMatrixReloaded.getProperty( "title" ) );
movies.add( theMatrixReloaded, "year", 2003 );
Node malena = graphDb.createNode();
malena.setProperty( "title", "Malèna" );
malena.setProperty( "year", 2000 );
movies.add( malena, "title", malena.getProperty( "title" ) );
movies.add( malena, "year", malena.getProperty( "year" ) );
```

Note that there can be multiple values associated with the same entity and key.

Next up, we will create relationships and index them as well:

```
// we need a relationship type
RelationshipType ACTS_IN = RelationshipType.withName( "ACTS_IN" );
// create relationships
Relationship role1 = reeves.createRelationshipTo( theMatrix, ACTS_IN );
role1.setProperty( "name", "Neo" );
roles.add( role1, "name", role1.getProperty( "name" ) );
Relationship role2 = reeves.createRelationshipTo( theMatrixReloaded, ACTS_IN );
role2.setProperty( "name", "Neo" );
roles.add( role2, "name", role2.getProperty( "name" ) );
Relationship role3 = bellucci.createRelationshipTo( theMatrixReloaded, ACTS_IN );
role3.setProperty( "name", "Persephone" );
roles.add( role3, "name", role3.getProperty( "name" ) );
Relationship role4 = bellucci.createRelationshipTo( malena, ACTS_IN );
role4.setProperty( "name", "Malèna Scordia" );
roles.add( role4, "name", role4.getProperty( "name" ) );
```

After these operations, our example graph looks like this:
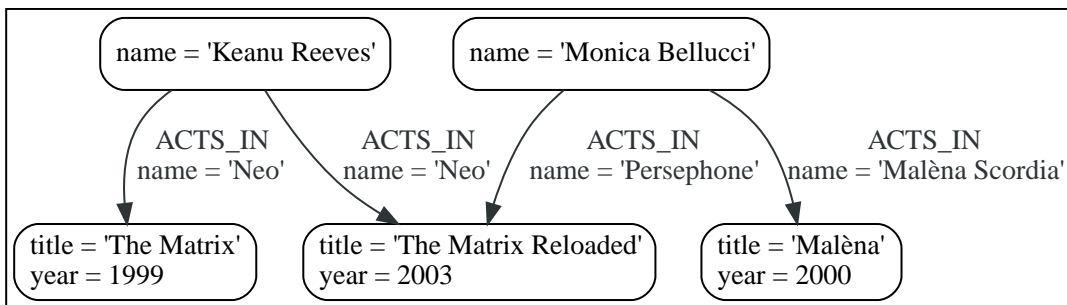
> ℹ The source code for the examples can be found here, `ImdbDocTest.java`

Remove

Removing (`remove()`) from an index is similar to adding, but can be done by supplying one of the following combinations of arguments:

- entity

- entity, key

- entity, key, value

```
// completely remove bellucci from the actors index
actors.remove( bellucci );
// remove any "name" entry of bellucci from the actors index
actors.remove( bellucci, "name" );
// remove the "name" -> "La Bellucci" entry of bellucci
actors.remove( bellucci, "name", "La Bellucci" );
```

> ℹ The source code for the example can be found here, `ImdbDocTest.java`.

Update

> ❗ To update an index entry, the old one must be removed and a new one added. For details on removing index entries, see Remove.

Remember that a node or relationship can be associated with any number of key-value pairs in an index.

This means that you can index a node or relationship with many key-value pairs that have the same key. In the case where a property value changes and you would like to update the index, it is not enough to just index the new value — you will have to remove the old value as well.

Here is a code example that demonstrates how it is done:

```
// create a node with a property
// so we have something to update later on
Node fishburn = graphDb.createNode();
fishburn.setProperty( "name", "Fishburn" );
// index it
actors.add( fishburn, "name", fishburn.getProperty( "name" ) );
// update the index entry
// when the property value changes
actors.remove( fishburn, "name", fishburn.getProperty( "name" ) );
fishburn.setProperty( "name", "Laurence Fishburn" );
actors.add( fishburn, "name", fishburn.getProperty( "name" ) );
```

> The source code for the example can be found here, `ImdbDocTest.java`.

Search

An index can be searched in two ways, `get` and `query`. The `get` method will return exact matches to the given key-value pair, whereas `query` exposes querying capabilities directly from the backend used by the index. For example the Lucene query syntax can be used directly with the default indexing backend.

Get

This is how to search for a single exact match:

```
IndexHits<Node> hits = actors.get( "name", "Keanu Reeves" );
Node reeves = hits.getSingle();
```

`org.neo4j.graphdb.index.IndexHits` is an `Iterable` with some additional useful methods. For example `getSingle()` returns the first and only item from the result iterator, or `null` if there is no hit.

Here is how to get a single relationship by exact matching and retrieve its start and end nodes:

```
Relationship persephone = roles.get( "name", "Persephone" ).getSingle();
Node actor = persephone.getStartNode();
Node movie = persephone.getEndNode();
```

Finally, you can iterate over all exact matches from a relationship index:

```
for ( Relationship role : roles.get( "name", "Neo" ) )
{
    // this will give us Reeves twice
    Node reeves = role.getStartNode();
}
```

> In case you do not iterate through all the hits, `IndexHits.close()` must be called explicitly.

Query

There are two query methods, one which uses a key-value signature where the value represents a query for values with the given key only. The other method is more generic and supports querying for more than one key-value pair in the same query.

Here is an example using the key-query option:

```java
for ( Node actor : actors.query( "name", "*e*" ) )
{
    // This will return Reeves and Bellucci
}
```

In the following example the query uses multiple keys:

```java
for ( Node movie : movies.query( "title:*Matrix* AND year:1999" ) )
{
    // This will return "The Matrix" from 1999 only.
}
```

Beginning a wildcard search with `"*"` or `"?"` is discouraged by Lucene, but will nevertheless work.

You cannot have *any whitespace* in the search term with this syntax. See Querying with Lucene query objects for how to do that.

Relationship indexes

An index for relationships is just like an index for nodes, extended by providing support to constrain a search to relationships with a specific start and/or end node. These extra methods reside in the `org.neo4j.graphdb.index.RelationshipIndex` interface which extends `org.neo4j.graphdb.index.Index<Relationship>`.

Example of querying a relationship index:

```java
// find relationships filtering on start node
// using exact matches
IndexHits<Relationship> reevesAsNeoHits;
reevesAsNeoHits = roles.get( "name", "Neo", reeves, null );
Relationship reevesAsNeo = reevesAsNeoHits.iterator().next();
reevesAsNeoHits.close();
// find relationships filtering on end node
// using a query
IndexHits<Relationship> matrixNeoHits;
matrixNeoHits = roles.query( "name", "*eo", null, theMatrix );
Relationship matrixNeo = matrixNeoHits.iterator().next();
matrixNeoHits.close();
```

And here is an example for the special case of searching for a specific relationship type:

```
// find relationships filtering on end node
// using a relationship type.
// this is how to add it to the index:
roles.add( reevesAsNeo, "type", reevesAsNeo.getType().name() );
// Note that to use a compound query, we can't combine committed
// and uncommitted index entries, so we'll commit before querying:
tx.success();
tx.close();

// and now we can search for it:
try ( Transaction tx = graphDb.beginTx() )
{
    IndexHits<Relationship> typeHits = roles.query( "type:ACTS_IN AND name:Neo", null, theMatrix );
    Relationship typeNeo = typeHits.iterator().next();
    typeHits.close();
```

Such an index can be useful if your domain has nodes with a very large number of relationships between them, since it reduces the search time for a relationship between two nodes. A good example where this approach pays dividends is in time series data, where we have readings represented as a relationship per occurrence.

The source code for the examples can be found here, `ImdbDocTest.java`.

Scores

The `IndexHits` interface exposes scoring (`org.neo4j.graphdb.index.IndexHits.currentScore()`) so that the index can communicate scores for the hits.

The result is not sorted by the score unless you explicitly specify that. See Sorting for how to sort by score.

```
IndexHits<Node> hits = movies.query( "title", "The*" );
for ( Node movie : hits )
{
    System.out.println( movie.getProperty( "title" ) + " " + hits.currentScore() );
}
```

The source code for the example can be found here, `ImdbDocTest.java`.

Configuration and full-text indexes

At the time of creation extra configuration can be specified to control the behavior of the index and which backend to use. For example to create a Lucene full-text index:

```
IndexManager index = graphDb.index();
Index<Node> fulltextMovies = index.forNodes( "movies-fulltext",
        MapUtil.stringMap( IndexManager.PROVIDER, "lucene", "type", "fulltext" ) );
fulltextMovies.add( theMatrix, "title", "The Matrix" );
fulltextMovies.add( theMatrixReloaded, "title", "The Matrix Reloaded" );
// search in the full-text index
Node found = fulltextMovies.query( "title", "reloAdEd" ).getSingle();
```

	The source code for the example can be found here, `ImdbDocTest.java`.

Here is an example of how to create an exact index which is case insensitive:

```java
Index<Node> index = graphDb.index().forNodes( "exact-case-insensitive",
        MapUtil.stringMap( "type", "exact", "to_lower_case", "true" ) );
Node node = graphDb.createNode();
index.add( node, "name", "Thomas Anderson" );
assertContains( index.query( "name", "\"Thomas Anderson\"" ), node );
assertContains( index.query( "name", "\"thoMas ANDerson\"" ), node );
```

	In order to search for tokenized words, the `query` method has to be used. The `get` method will only match the full string value, not the tokens.

	The source code for the example can be found here, `TestLuceneIndex.java`.

The configuration of the index is persisted once the index has been created. The `provider` configuration key is interpreted by Neo4j, but any other configuration is passed onto the backend index (e.g. Lucene) to interpret.

Table 24. Lucene indexing configuration parameters

Parameter	Possible values	Effect
`type`	`exact`, `fulltext`.	`exact` is the default and uses a Lucene keyword analyzer. `fulltext` uses a white-space tokenizer in its analyzer.
`to_lower_case`	`true`, `false`.	This parameter goes together with `type`: `fulltext` and converts values to lower case during both additions and querying, making the index case insensitive. Defaults to `true`.
`analyzer`	The full class name of an Analyzer.	Overrides the `type` so that a custom analyzer can be used. ⓘ `to_lower_case` still affects lowercasing of string queries. If the custom analyzer uppercases the indexed tokens, string queries will not match as expected.

Extra features for Lucene indexes

Numeric ranges

Lucene supports smart indexing of numbers, querying for ranges and sorting such results, and so does its

backend for Neo4j. To mark a value so that it is indexed as a numeric value, we can make use of the `org.neo4j.index.lucene.ValueContext` class, like this:

```
movies.add( theMatrix, "year-numeric", new ValueContext( 1999 ).indexNumeric() );
movies.add( theMatrixReloaded, "year-numeric", new ValueContext( 2003 ).indexNumeric() );
movies.add( malena, "year-numeric", new ValueContext( 2000 ).indexNumeric() );

int from = 1997;
int to = 1999;
hits = movies.query( QueryContext.numericRange( "year-numeric", from, to ) );
```

> ℹ The same type must be used for indexing and querying. That is, you cannot index a value as a Long and then query the index using an Integer.

By giving `null` as from/to argument, an open ended query is created. In the following example we are doing that, and have added sorting to the query as well:

```
hits = movies.query(
        QueryContext.numericRange( "year-numeric", from, null )
                .sortNumeric( "year-numeric", false ) );
```

From/to in the ranges defaults to be *inclusive*, but you can change this behavior by using two extra parameters:

```
movies.add( theMatrix, "score", new ValueContext( 8.7 ).indexNumeric() );
movies.add( theMatrixReloaded, "score", new ValueContext( 7.1 ).indexNumeric() );
movies.add( malena, "score", new ValueContext( 7.4 ).indexNumeric() );

// include 8.0, exclude 9.0
hits = movies.query( QueryContext.numericRange( "score", 8.0, 9.0, true, false ) );
```

> ℹ The source code for the examples can be found here, `ImdbDocTest.java`.

Sorting

Lucene performs sorting very well, and that is also exposed in the index backend, through the `org.neo4j.index.lucene.QueryContext` class:

```
hits = movies.query( "title", new QueryContext( "*" ).sort( "title" ) );
for ( Node hit : hits )
{
    // all movies with a title in the index, ordered by title
}
// or
hits = movies.query( new QueryContext( "title:*" ).sort( "year", "title" ) );
for ( Node hit : hits )
{
    // all movies with a title in the index, ordered by year, then title
}
```

You can sort the results by relevance (score) like this:

```
hits = movies.query( "title", new QueryContext( "The*" ).sortByScore() );
for ( Node movie : hits )
{
    // hits sorted by relevance (score)
}
```

ℹ️ The source code for the examples can be found here, `ImdbDocTest.java`.

Querying with Lucene query objects

Instead of passing in Lucene query syntax queries, you can instantiate such queries programmatically and pass in as argument, for example:

```
Node actor = actors.query( new TermQuery( new Term( "name", "Keanu Reeves" ) ) ).getSingle();
```

ℹ️ The `TermQuery` is basically the same thing as using the `get` method on the index.

This is how to perform *wildcard* searches using Lucene query objects:

```
hits = movies.query( new WildcardQuery( new Term( "title", "The Matrix*" ) ) );
for ( Node movie : hits )
{
    System.out.println( movie.getProperty( "title" ) );
}
```

Note that this allows for whitespace in the search string.

ℹ️ The source code for the examples can be found here, `ImdbDocTest.java`.

Compound queries

Lucene supports querying for multiple terms in the same query, like so:

```
hits = movies.query( "title:*Matrix* AND year:1999" );
```

🔥 Compound queries cannot search across committed index entries and those who have not got committed yet at the same time.

ℹ️ The source code for the example can be found here, `ImdbDocTest.java`.

Default operator

The default operator (that is whether `AND` or `OR` is used in between different terms) in a query is `OR`. Changing that behavior is also done via the `org.neo4j.index.lucene.QueryContext` class:

```
QueryContext query = new QueryContext( "title:*Matrix* year:1999" )
        .defaultOperator( Operator.AND );
hits = movies.query( query );
```

> ℹ️ The source code for the example can be found here, `ImdbDocTest.java`.

License