Large-Scale and Multi-Structured Databases

Neo4J: Java Driver

Prof. Pietro Ducange
Ing. Alessio Schiavo
alessio.schiavo@phd.unipi.it

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
UNIVERSITÀ DI PISA
CROSSLAB Innovation for industry 4.0

Introduction

- Neo4j provides official drivers ***for many programming languages***.
- These drivers are supported by Neo4j and come with ***full cluster*** routing support.
- ***Community drivers*** also exist for many languages, but vary greatly in terms of feature sets, maturity, and support.
- To find out more about community drivers, visit https://neo4j.com/developer/language-guides/

Some Clarifications

- The driver API is intended to ***be topologically agnostic***.
- This means that the underlying database topology — single instance, Causal Cluster, etc. — can be altered without requiring a corresponding alteration to application code.
- In the general case, only the ***connection URI*** needs to be modified when changes are made to the topology.
- The official drivers ***do not support HTTP communication***.
- If we need an HTTP driver, choose one of the ***community drivers***.

Installing Java Driver with Maven

When using Maven, add the following block to your *pom.xml* file:

```
<dependencies>
    <dependency>
        <groupId>org.neo4j.driver</groupId>
        <artifactId>neo4j-java-driver</artifactId>
        <version>5.27.0</version>
    </dependency>
 </dependencies>
```

Here you can find the ***API documentation***:
https://neo4j.com/docs/java-manual/current/

The Driver Object

- A Neo4j client application will require a ***driver instance*** in order to provide access to the database.
- This driver should be made ***available to all parts*** of the application that need to interact with Neo4j.
- The driver can be considered ***thread-safe.***

A note on lifecycle

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

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

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

Construct a Driver Instance

- To construct a driver instance, a **connection *URI*** and ***authentication*** information must be supplied.
- Additional configuration details can be supplied if required.
- All these details are ***immutable*** for the ***lifetime of the driver***.
- Therefore, if ***multiple configurations*** are required (such as when working with multiple database users) then ***multiple driver*** instances must be used.
- When we work with a single server database, we can create a direct driver via ***bolt URI***, for example: bolt://localhost:7687. For a cluster, please use the ***neo4j URI*** neo4j://localhost:7687

Connection URI

```
neo4j://<HOST>:<PORT>[?policy=<POLICY-NAME>]
```

Load balancing policies

A `policy` parameter can be included in the query string of a `neo4j://` URI, to customize the routing table and take advantage of **multi-data center routing settings**.

> A prerequisite for using load balancing policies with the driver is that the database is operated on a **cluster**, and that some **server policies** are set up.

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

In a clustered environment, the URI address will resolve to one of more of the core members; for standalone installations, this will simply point to that server address. The `policy` parameter allows for customization of the routing table and is discussed in more detail in **load balancing policies**.

Authentication

Authentication details are provided as an ***auth token*** which contains the usernames, passwords or other credentials required to access the database.

Neo4j supports multiple authentication standards but uses ***basic authentication*** by default.

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

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

Driver Config

ConnectionAcquisitionTimeout

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

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

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

https://neo4j.com/docs/java-manual/current/client-applications/

Connection Pools

- The driver maintains a ***pool of connections***.
- The pooled connections are reused by ***sessions*** and ***transactions*** to avoid the overhead added by establishing new connections for every query.
- The connection pool always starts up empty. New connections are created on demand by ***sessions*** and ***transactions***.
- When a session or a transaction is done with its execution, the connection will be ***returned*** to the pool to be ***reused***.
- Application users ***can tune connection pool settings*** to configure the driver for different use cases based on client performance requirements and database resource consumption limits. Moreover, also the ***connection timeout*** and the ***max retry time*** can be configured.

Logging

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

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

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

CRUD Operations with the Neo4j Java driver

Write to the database

- To create a node representing a person named Alice, use the Cypher clause CREATE:

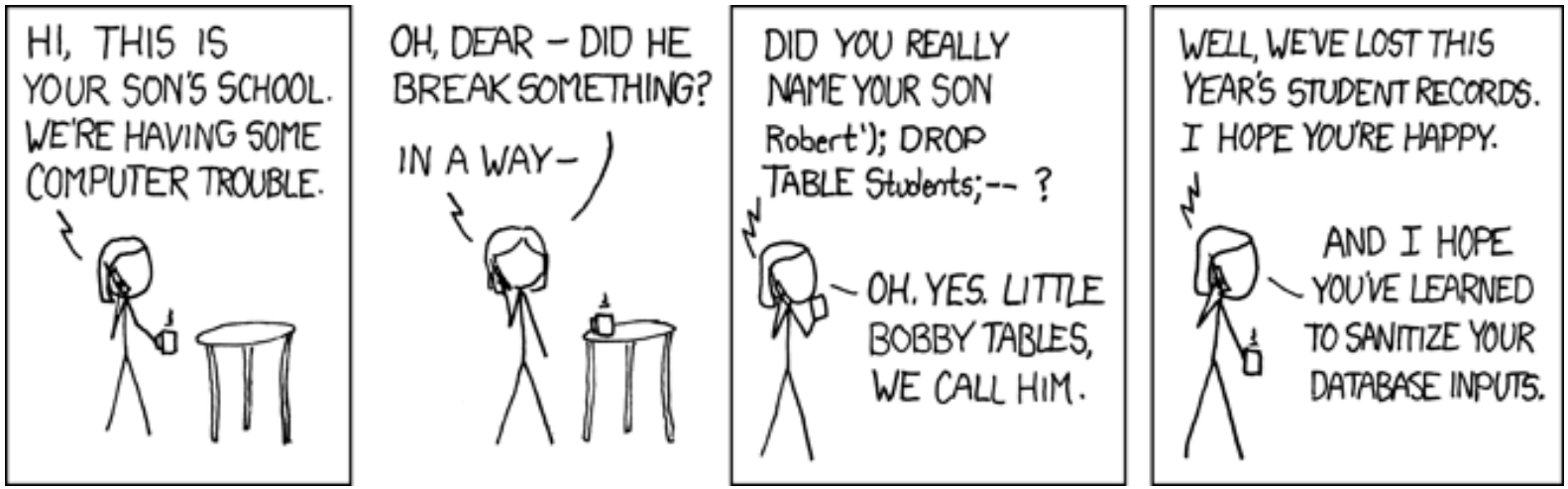
Create a node representing a person named Alice

```
// import java.util.Map;
// import java.util.concurrent.TimeUnit;
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("CREATE (:Person {name: $name})")  ①
    .withParameters(Map.of("name", "Alice"))  ②
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())  ③
    .execute();
var summary = result.summary();  ④
System.out.printf("Created %d records in %d ms.%n",
    summary.counters().nodesCreated(),
    summary.resultAvailableAfter(TimeUnit.MILLISECONDS));
```

Query parameters (I)

- **Do not hardcode or concatenate parameters directly into queries.** Instead, always use placeholders and specify the Cypher parameters, as shown in the previous examples. This is for:
  - **Performance benefits**: Neo4j complies and caches queries, but can only do so if the query structure is unchanged;
  - **Security reasons**: protecting against Cypher injection



Query parameters (II)

- Given that Bobby's mother would likely have continued her crusade, her next child might be named to target Cypher, used by the leading graph database Neo4j. Let's call them Little Robby Labels.

```
"Robby' WITH true as ignored MATCH (s:Student) DETACH DELETE s; //"
```

- This is an attempt at Cypher injection, the equivalent of the SQL injection attack, but with the deletion of all :Student nodes instead. If Robby's school used string concatenation to build their queris, this attack might succeed:

```
String queryString = "CREATE (s:Student) SET s.name = '" + studentName + "'";

Result result = session.run(queryString);
```

Query parameters (II)

- Given that Bobby's mother would likely have continued her crusade, her next child might be named to target Cypher, used by the leading graph database Neo4j. Let's call them Little Robby Labels.

```
"Robby' WITH true as ignored MATCH (s:Student) DETACH DELETE s; //"
```

- This is an attempt at Cypher injection, the equivalent of the SQL injection attack, but with the deletion of all :Student nodes instead. If Robby's school used string concatenation to build their queris, this attack might succeed.

The query that will be run is:

```
CREATE (s:Student)
SET s.name = 'Robby' WITH true as ignored MATCH (s:Student) DETACH DELETE s; //';
```

Protecting against Cypher Injection using Parameters

- When receiving user input, it is possible to **prevent Cypher Injection by using parameters**. In the previous example, instead of concatenating the students name with the CREATE query, one should use parameters instead:

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

String query =
"CREATE (s:Student)" + "\n" +
"SET s.name = $studentName";

Result result = transaction.execute( query, params );
```

- The parametrized query would look like this:

```
CREATE (s:Student)
SET s.name = $studentName
```

Query configuration

- You can supply further configuration parameters to alter the default behavior of **.executableQuery()**. You do so through the method **.withConfig(),** which takes a **QueryConfig** object.

- **DATABASE SELECTION:** it is recommended to **always specify the database explicitly** with the **.withDatabase("<dbName>")** method, even on single-database instances. This allows the driver to work more efficiently, as it saves a network round-trip to the server to resolve the home database.

```
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("MATCH (p:Person) RETURN p.name")
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();
```

Read from the database

- To retrieve information from the database, use the Cypher clause MATCH:

Retrieve all Person nodes

```
// import java.util.concurrent.TimeUnit;
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("MATCH (p:Person) RETURN p.name AS name")
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .execute();

// Loop through results and do something with them
var records = result.records(); ①
records.forEach(r -> {
    System.out.println(r);  // or r.get("name").asString()
});

// Summary information
var summary = result.summary(); ②
System.out.printf("The query %s returned %d records in %d ms.%n",
    summary.query(), records.size(),
    summary.resultAvailableAfter(TimeUnit.MILLISECONDS));
```

Update the database (I)

- To update a node's information in the database, use the Cypher clauses MATCH and SET:

Update node Alice to add an age property

```
// import java.util.Map;
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("""
    MATCH (p:Person {name: $name})
    SET p.age = $age
    """)
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .withParameters(Map.of("name", "Alice", "age", 42))
    .execute();
var summary = result.summary();
System.out.println("Query updated the database?");
System.out.println(summary.counters().containsUpdates());
```

Update the database (II)

- **To create a new relationship, linking it to two already existing nodes**, use a combination of the Cypher clauses MATCH and CREATE:

Create a relationship :KNOWS between Alice and Bob

```
// import java.util.Map;
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("""
    MATCH (alice:Person {name: $name})  ①
    MATCH (bob:Person {name: $friend})  ②
    CREATE (alice)-[:KNOWS]->(bob)  ③
    """)
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .withParameters(Map.of("name", "Alice", "friend", "Bob"))
    .execute();
var summary = result.summary();
System.out.println("Query updated the database?");
System.out.println(summary.counters().containsUpdates());
```

Delete from the database

- **To remove a node and any relationship attached to it,** use the Cypher clause DETACH DELETE:

Remove the Alice node and all its relationships

```
// import java.util.Map;
// import org.neo4j.driver.QueryConfig;

// This does not delete _only_ p, but also all its relationships!
var result = driver.executableQuery("""
    MATCH (p:Person {name: $name})
    DETACH DELETE p
    """)
    .withConfig(QueryConfig.builder().withDatabase("neo4j").build())
    .withParameters(Map.of("name", "Alice"))
    .execute();
var summary = result.summary();
System.out.println("Query updated the database?");
System.out.println(summary.counters().containsUpdates());
```

Run queries as a different user

- You can execute a query through a different used with the method **.withAuthToken()** . Switching user at the query level is cheaper than creating a new **Driver** object. The query is then run within the security context of the given user (i.e., home database, permissions, etc.).

```
// import org.neo4j.driver.AuthTokens;
// import org.neo4j.driver.QueryConfig;

var authToken = AuthTokens.basic("somebodyElse", "theirPassword");
var result = driver.executableQuery("MATCH (p:Person) RETURN p.name")
    .withConfig(QueryConfig.builder()
        .withDatabase("neo4j")
        .withAuthToken(authToken)
        .build())
    .execute();
```

Run queries as a different user

- You can execute a query through a different used with the method **.withAuthToken()** . Switching user at the query level is cheaper than creating a new **Driver** object. The query is then run within the security context of the given user (i.e., home database, permissions, etc.).

```
// import org.neo4j.driver.QueryConfig;

var result = driver.executableQuery("MATCH (p:Person) RETURN p.name")
    .withConfig(QueryConfig.builder()
        .withDatabase("neo4j")
        .withImpersonatedUser("somebodyElse")
        .build())
    .execute();
```

Run your own transactions

- When querying the database with **executableQuery()**, the driver automatically creates a *transaction*.

- You can include multiple Cypher statements in a single query, as for example when using MATCH and CREATE is sequence to **update the database**, but you cannot have multiple queries and interleave some client-logic in between them.

- For more advanced use-cases, the driver provides functions to take full control over the transaction lifecycle. These are called **managed transactions**, and you can think of them as a way of unwrapping the flow of **executableQuery()** and being able to specify its desired behavior in more places.

Create a session

- Before running a transaction, **you need to obtain a *session*.**
- Sessions act as concrete query channels between the driver and the server, and ensure **causal consistency** is enforced.
- Sessions are created with the Driver.session() method.

```
// import org.neo4j.driver.SessionConfig

try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {
    // session usage
}
```

https://neo4j.com/docs/java-manual/current/transactions/

Run a managed transaction

- A transaction **can contain any number of queries**. As Neo4j is **ACID compliant**, **queries within a transaction will either be executed as a whole or not at all**. Use transactions **to group related queries** which work together to achieve a single *logical* database operation.

Retrieve people whose name starts with Al.

```
// import java.util.Map
// import org.neo4j.driver.SessionConfig

try (var session = driver.session(SessionConfig.builder().withDatabase("neo4j").build())) {  ①
    var people = session.executeRead(tx -> {  ②
        var result = tx.run("""
        MATCH (p:Person) WHERE p.name STARTS WITH $filter  ③
        RETURN p.name AS name ORDER BY name
        """, Map.of("filter", "Al"));
        return result.list();  // return a list of Record objects ④
    });
    people.forEach(person -> {
        System.out.println(person);
    });
    // further tx.run() calls will execute within the same transaction
}
```

Process query results

- The driver's output of a query is a Result object, which encapsulates the Cypher result in a rich data structure that requires some parsing on the client side. There are two main points to be aware of:
  - **The result records are not immediately and entirely fetched and returned by the server.**
  - **The result acts as a cursor.** This means there is no way to retrieve a previous record from the stream, unless you saved it in an auxiliary data structure.

The easiest way of processing a result is by calling .list() on it, which yields a list of Record objects.

https://neo4j.com/docs/api/java-driver/current/org.neo4j.driver/org/neo4j/driver/Result.html

How the driver works with results:

Sessions

- A session is a ***container*** for a sequence of transactions.
- Sessions ***borrow connections*** from a pool as required and so should be considered ***lightweight*** and ***disposable***.
- Sessions are usually ***scoped*** within a ***context block***.
- This ensures that they are ***properly closed*** and that any underlying connections are ***released*** and ***not leaked***.

https://neo4j.com/docs/java-manual/5/session-api/

Cypher types vs Java types

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

Exercises

1. **Implement the following queries in Neo4J Java, using the movies graph database**

   a) Find a title which contains a certain string
   b) Find a movie by title and display the cast
   c) Find the director that managed the least number of distinct actors
   d) Find the top 3 movies with the highest ratio between number of actors and number of producers
   e) Find the shortest path from the youngest actor and the oldest director

Most of the information included in this presentation have been retrieved from:

https://neo4j.com/docs/java-manual/

Students are invited to use the previous source, along with other the community support, as references for developing their applications.