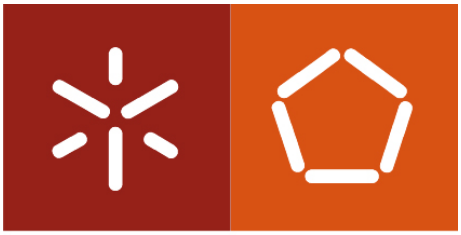


Universidade do Minho

IPLN
Assignment 2 - Neo4j

Márcio Rocha PG41086
Vasco Filipe Figueiredo PG41102

30 November 2019

Introduction

For the class “Introdução a Processamento de Linguagem Natural”, we were tasked with analysing the graph database Neo4j.

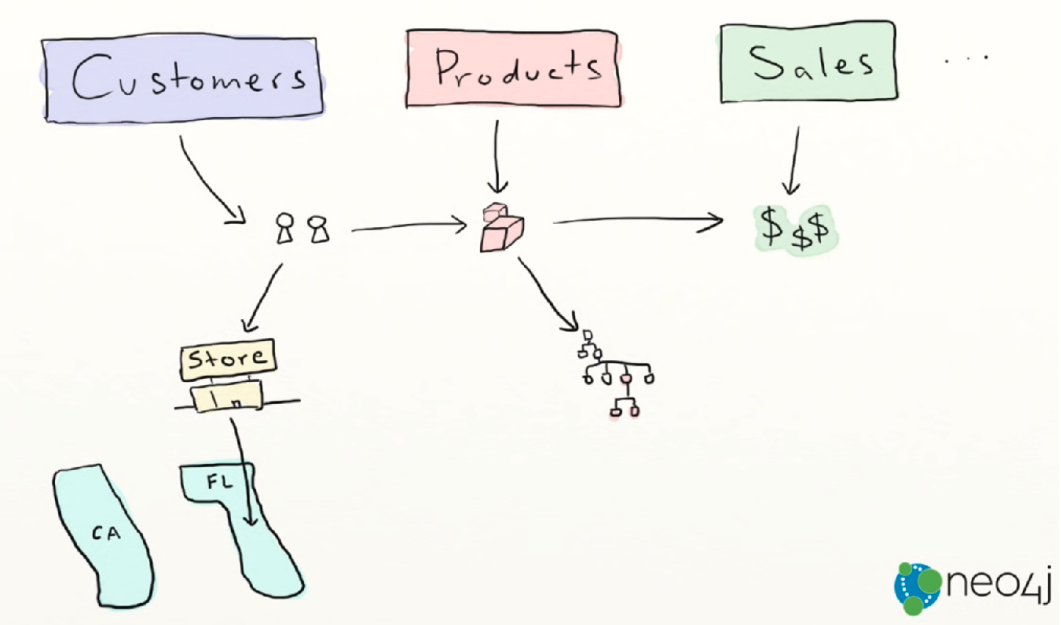
In this document, we will be analyzing Neo4j, it’s python library and show and explain a working example.

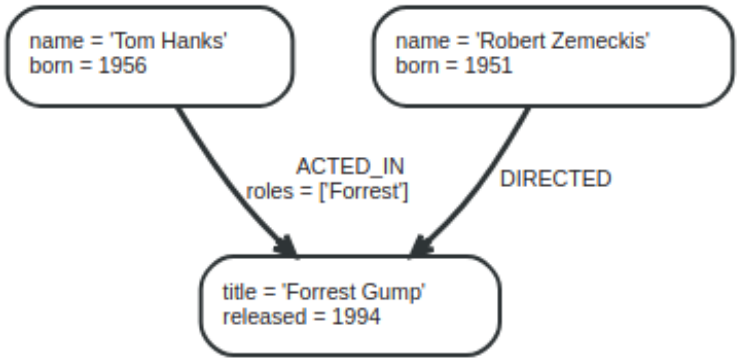
Neo4j

Neo4j is an open-source, NOSQL, native graph database that provides ACID-compliant transactional backend for applications, it's referred as "native graph database" because its native storage layer is implemented like a connected graph model, with this Neo4j avoids the search for relationships through multiple index look ups which are expensive, making it 60% faster than databases with traditional implementations such as MySQL.

Before we can understand more about what Neo4j is we must know what Graph Databases are and what makes it so good for uses such as in companies and organizations around the world.

A Graph Database is a database designed to treat relationships between data as equally import to the data itself. It does not follow the traditional storage databases which store data in rows, columns and tables, instead graph databases has a completely flexible network structure of records or nodes, connected by relationships, representing a real world structure of the information like the image below.





Structure

Neo4j has two main components: the desktop graph manager, that allows the user to create and manage multiple database; and the drivers, which are libraries written for various languages, that allow direct communication with the database. The database is written in Cypher, which is a declarative graph query language and was created specifically for Neo4j and optimized for graphs.

In this image, we can see a Neo4j graph and its structure. In the image we can see 3 rectangles with text inside and 2 lines connecting the rectangles. Each rectangle is a Node. Nodes can be seen as entities with properties. The simplest possible graph is a single node. A node can also have multiple properties. Properties are name-value pairs that are used to add qualities to nodes and relationships.

In the image above, we see that the nodes have names ("Person" or "Movie"). This names are known as Labels. Labels are used to classify and group nodes into sets with nodes with the same labels. So if we have a bunch of nodes representing different animals, and we want to group the mammals, we can give them a label :Mammal. With that in place, we can perform operations only on those nodes.

The lines that connect the nodes are called Relationships. Each relationship need obligatorily a relationship type. In the image above its ACTED IN and DIRECTED. Relationships also have a source node and a target node. In the example above, "Tom Hanks"is the source node, while "Forrest Gump"is the target node. Relationships always have a direction, and a node can have a relationship to itself.

Code

The class Driver is the base for all types of instances of which are used as the primary access point to Neo4j's server. This class has two parameters, the uri parameter, which is used to identify the graph database service and

the config parameter, which is used for configuration and authentication details, setting up connection with the desired graph database through a Bolt protocol.

When we want to run a Cypher query on our database we create a session. This session is a class within the Driver class and it's a logical context for transactional units of work, with read or write access mode. As needed connections will be drawn from the class Driver connection pool and used for transactions.

In general, sessions will be created and destroyed within a 'with' context. For example:

```
with driver.session() as session:
    result = session.run("MATCH (a:Person) RETURN
        a.name")
    # do something with the result...
```

The reason for this is because sessions are lightweight operations and not thread safe, therefore its imperative that they are short-lived and don't span multiple threads.

The run method in class Session runs a Cypher statement within an auto-commit transaction, the statement is sent and the result header is received immediately, but the class StatementResult content is fetched lazily as consumed by the client application.

The class transaction, within the run method, is a container for one or multiple Cypher queries to be executed within a single context session. Cypher is typically expressed as a statement template plus a set of named parameters. In python they are expressed through a dictionary of parameters. For example:

```
>>> statement = "CREATE (a:Person {name:{name},
    age:{age}})"
>>> tx.run(statement, {"name": "Alice", "age": 33})
>>> tx.run(statement, {"name": "Alice"}, age=33)
>>> tx.run(statement, name="Alice", age=33)
```

Class StatementResult and BoltStatementResult are handlers for the result of Cypher statement execution. Instances of this class are typically constructed and returned by Session.run and Transaction. Through this handler's records method, we have access to the records returned by a query, which are an iterable list of Records, if the query doesn't return anything these records are empty.

The class Records is a immutable ordered collection of key-value pairs.

Using Neo4j

Neo4j is available for both standalone server and embeddable component(Neo4j Desktop), however for this project we will only cover how to use Neo4j Desktop in order to demonstrate the examples generated graph database.

The Neo4j Desktop application can be downloaded here: `https://neo4j.com/download/` . An account will have to be created in order to use the application.

Instructions:

- When inside create a database on Projects section;
- Click on the rectangular "Add Graph", press "Create a Local Graph and click on "Start" button;
- After that click on "Manage" button and then on "Open browser".

Small Example

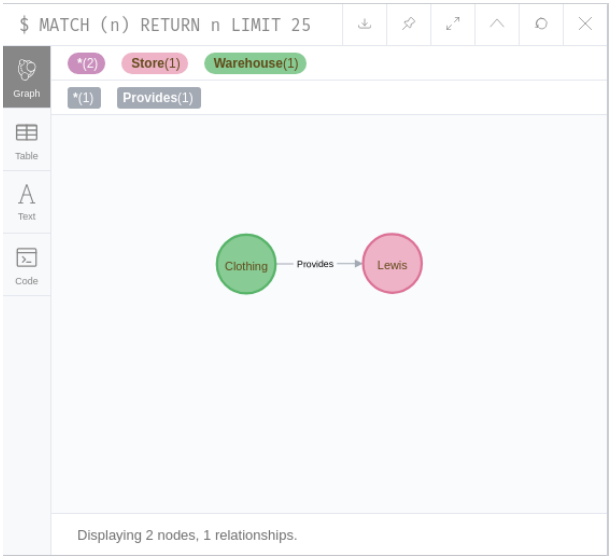
```
#!/usr/bin/env python3
from neo4j import GraphDatabase #import of neo4j

def main():
    driver = GraphDatabase.driver(uri="bolt://localhost:
    7687" , auth=("neo4j", "eheh"))

    warehouse = "Clothing"
    store = "Lewis"
    relation = "Provides"

    with driver.session() as session:
        session.run("CREATE (node1:Warehouse {name:
        $warehouse})-[r:"+relation+"]->(node2:Store
        {name: $store})",warehouse=warehouse,store=store,
        relation=relation)

if __name__ == "__main__":
    main()
```

In this example, we can see the most basic structure of a Neo4j code, where we create 2 nodes and a relation. Since the Neo4j's databases are written in Cypher, in order to create nodes or relations, we need to create queries in Cypher. The 5 principle clauses are CREATE (creates something), MATCH (checks if exists), MERGE (merges object with existing object, if there is none creates a new object), RETURN, DETACH (deletes something).

In the main, we create a driver, which allows us to connect to a database. The "ip"we use is generated by Neo4j Desktop. The ip can also be generated by the Neo4j website (https://neo4j.com/sandbox-v2/) or it can even be created by hosting it in a cloud (check https://neo4j.com/developer/guide-cloud-deployment/ for more information).

After connecting to the database, we generate 2 nodes and a relationship between them.

- (node1:Warehouse name: $warehouse): Creates a node of type "Warehouse"with one parameter, "name";
- (node2:Store name: $store): Creates a node of type "Store"with one parameter, "name";
- -[r:"+relation+"]-¿: Creates a relation of type "Provides", the direction this relations points is related how the query is written, in this case the starting point is "Warehouse"node and the ending point is "Store"node.



In the image above we can see displayed the result graph on Neo4j Desktop, with the 2 nodes and their relationship.

Relationship Finder

This example implements an application in the context of IPLN, it's intended to read a text and return all, love and family, relationships with the correspondent people attached, nodes, creating a vast network depending on the complexity of the relations and text.

Firstly, we look for a potential relationship in a phrase, this is achieved by finding a "relation verb"such as married, boyfriend,etc., followed by the people that are attached to both ends of the relationship, it is needed at least two people to work,otherwise nothing will be created.

When both relationship and people are identified we proceed to implement this data into neo4j's graph database. When the program is first executed we create a connection with the database through a driver, now we only create sessions every time we want to execute a transaction with a Cypher query.

The main Cypher queries are:

```
#creates a node if it does not exist
def createNode(driver,person):
    with driver.session() as session:
        session.run("MERGE (p:Person {surename: $person}) RETURN p.surename", person = person)
```

This query checks if there is a existing node "Person"with its surname equals to "person", if this node is not found it creates a new one with this parameters.

```
#adds a relatioship to two existing nodes, from person1 to person2
def addRelation(driver,person1, person2, relation):
    with driver.session() as session:
        session.run("MATCH (p1:Person { surename: $person1 }),(p2:Person { surename: $person2 })
        MERGE (p1)-[r:"+relation +"]->(p2) RETURN p1.surename, type(r), p2.surename", person1 = person1, person2 = person2, relation = relation)
```

This query is executed after the nodes "Person"are created, however we do not have access to the nodes themselves we only have their names, so we first use the "MATCH"clause to find them in our database and afterwards to check if our wanted relationship already exists, if not "MERGE"clause will create it for us.

Conclusion

To summarize, Neo4j is a innovative way to work with databases. By switching to a graph system, Neo4j is able to create a simpler and faster system than a traditional implementation such as MySQL. Coding with Cypher is also very straightforward, as the language was created to be simple and clear. Although, Neo4j is limited by the fact that in order to create a database, it needs a connection to either the Neo4j Online Sandbox or to the Neo4j Desktop application. Also, to communicate with the database, it needs "drivers"(libraries to different languages) and not all of them are heavily supported with some of them being outdated (for example, some of the drivers for python only support python 2.8).

The project we developed using Neo4j is a simple "relationship detector", that tries to find in a text, all people mentioned and the relationships between them. Although it works to some extent, creating a software like this is extremely hard, since the portuguese language has lot of different rules, which makes it harder to create something like this from scratch. But we were still able to create a decent proof of concept that also utilizes Neo4j capabilities and demonstrates some of the potential of this graph database.