CentraleSupélec

INTRODUCTION TO DATABASES 2019 – 2020

Lab. Assignment 3 : Graph Databases

This assignment aims at introducing and examining the main notions underlying graph databases. We will use **Neo4j** as a database management system.

We are going to learn the language **Cypher** that is used to write queries to obtain data from a graph in a simple, yet expressive, way. We will also see that relatively simple queries can form the basis of a movie recommender system.

1. THE DATASET

The dataset that we are using consists of data obtained from *MovieLens* [1], a recommender system whose users give movies a rate between 1 and 5 based on whether they dislike or love them. MovieLens uses the rates to recommend movies that its users might like. The dataset is modeled as a **directed graph** and consists of 100,004 rates on 9,125 movies across 671 users between January 9th, 1995 and October 16, 2016. The dataset also contains the names of the directors and the actors of each movie.

A directed graph $G = (V, E)$ consists of:

- A set V of **nodes** corresponding to the entities, such as movies, users, actors and directors.
- A set E of **links**, each connecting two nodes and representing a relationship between two entities. For instance, a link between a director and a movie represents the relationship "directed by".

We are now going to explore the structure of this graph by using Neo4j.

[1] https://movielens.org

2. PRELIMINARIES

Download the archive `graph.db.zip` containing the data that you're going to use at the following address:

https://gquercini.github.io/db/lab3/graph.db.zip

Extract the archive, it contains a folder named *graph.db*.

- **Windows users**. When extracting the archive, right-click on the archive, select *Extract all* then as a destination folder specify: **c:\Users\[your-username]\Downloads**.

You can leave the folder in your Downloads folder, but do not rename it.
If you haven't done so yet, you need to download and install **Neo4j Desktop**, at the following address:

https://neo4j.com/download/

Open Neo4j Desktop. The home page of the application should show the panel *Projects* with one already created project named *My Project*. Click on it and, on the panel on the right side of the screen, click on *Add Graph* and follow these instructions:

- Create a Local Graph
- Give the graph a name (e.g., `movies`) and sets a password.
- Click on *Create*. Please wait, this might take a while.
- Once the graph is created, click on *Manage*.
- Click on *Open Folder*. This should open the folder that contains an instance of the Neo4j server that will be launched to manage the new graph. **Windows users:** it might happen that the window with the folder does not pop up. In this case, look at the task bar (the bar at the bottom of the screen) you should see that the File Explorer is flashing, just click on it to make the window pop up.
- In this folder, click on *data* and then *databases*.
- Drag and drop the folder *graph.db* from the Downloads folder to the folder *data/databases*.
- Return to Neo4j Desktop and click on the *Start* button (icon play). The Neo4j server will now start. Wait, as it might take a while.

Once the server is up and running, click on *Open Browser* to open a graphical front-end to the Neo4j database (Figure 1).

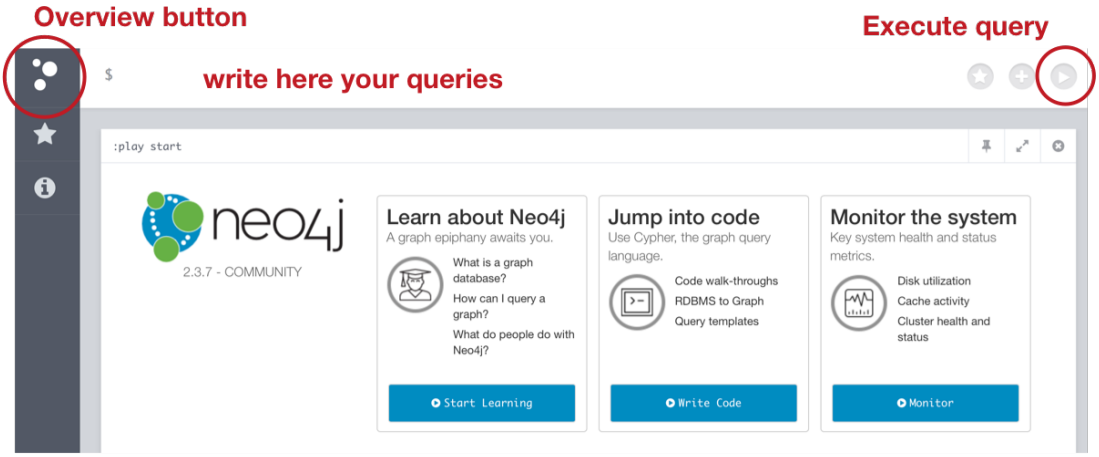

Figure 1: Neo4j Browser

3. Data Exploration

We are now going to explore the dataset by using simple queries expressed in **Cypher**, the query language of Neo4j. For your information, the Cypher reference card is available at the following address:

https://neo4j.com/docs/cypher-refcard/current/

The queries need to be written in the text field that you see on the top of the Neo4j Web interface; this field is indicated by the symbol "$" (Figure 1). Once you've written the query, type the key RETURN to execute it.

Exercise 1

Write and execute the following query:

```
MATCH  (n:Movie {titre:"Toy Story"})
RETURN n
```

The result of this query is the node that corresponds to the movie titled "Toy Story"; it will be visualized in the Neo4j Web interface (Figure 2). In order to visualize the result of the query as a text rather than a graph, click on the button *Rows* (highlighted by a red circle in Figure 2).

Here are the ingredients of this query :

- The clause **MATCH** allows to specify what we are looking for (in this case, a node).
- The element in parentheses "(n:Movie {title:"Toy Story"})" indicates a node named *n* having the **label** *Movie* and the **property** *title* whose value is "Toy Story".
- The clause **RETURN** allows to specify what we want as the result of the query (in this case, the node *n*).

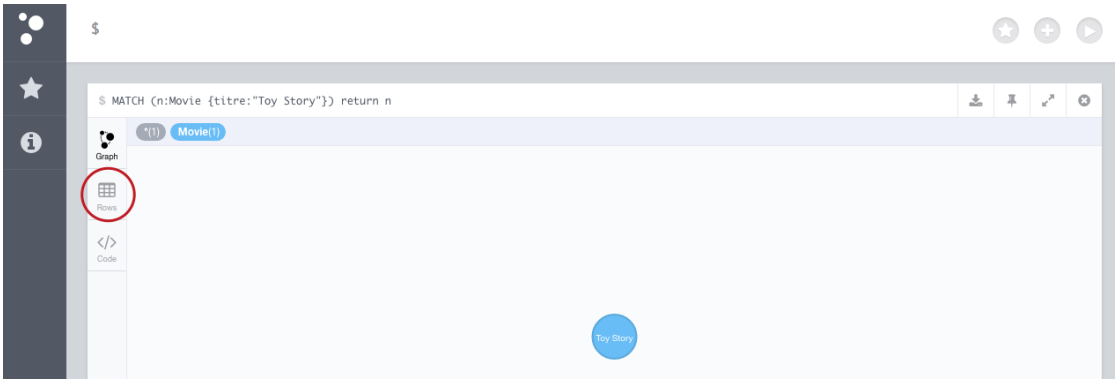

Figure 2: Result of the first query.

Another way to express this query is by using the clause **WHERE** that allows the specification of boolean conditions:

```
MATCH (n:Movie)
WHERE n.title="Toy Story"
RETURN n
```

where *n.title* is the value of the property *title* of node *n*.

All nodes have a **label** that identifies the type of the entity represented by the node (e.g., "Movie", "Actor"...). In order to see the list of all node labels used in the database, click on the button "Overview" (Figure 1).

Exercise 2

Select the node returned by the previous query and observe its properties.

As we can see, a node with label *Movie* corresponds to a movie and has the following properties:

- *id*, a numeric identifier [2].
- *year*, the year of the release of the movie.
- *title*, the title of the movie in English.
- *titre*, the title of the movie in French.

Exercise 3

Write and execute the following query:

```
MATCH (n:Movie {title:"The Strange Love of Martha Ivers"})-[r]-(m)
RETURN n, m;
```

[2] do not confuse it with <*id*> that is the identifier of the node assigned automatically by *Neo4j*

The result of this query (visualized as a star) is the subgraph induced by the node n (the center of the star) representing the movie titled "The Strange Love of Martha Ivers" and all the nodes m that are connected to n through a link r. Note that we did not specify any label or property of m because we want to obtain all the nodes adjacent to n.

We can observe that:

1. The nodes with the same color have the same label.
2. Each link is **directed** and has a **type** that indicates its semantics, namely :
   - HAS_GENRE is the type of a link from a node with label *Movie* to a node with label *Genre*.
   - RATED is the type of a link from a node labeled *User* to a node labeled *Movie*. A link with type RATED connects a user to a movie that s/he rated. It has a property named *rate* whose value is between 1 and 5 (click on a link with type RATED to see the value of this property).
   - ACTED_IN is the type of a link from a node labeled *Actor* to a node labeled *Movie*. A link with type ACTED_IN connects an actor to a movie s/he played in.
   - DIRECTED is the type of a link from a node labeled *Director* to a node labeled *Movie*. A link with type DIRECTED connects a director to a movie s/he directed.
3. The nodes labeled *User* have a property named *id*, a numeric identifier ; the nodes labeled *Genre*, *Actor* and *Director* have a property named *name*.

Exercise 4

Write and execute the following query:

```
MATCH (n:Movie {title:"The Strange Love of Martha Ivers"})-[r:HAS_GENRE]->(m:Genre)
RETURN n, m;
```

The result of this query is the subgraph induced by n and all the nodes m labeled *Genre* such that there is a link r leaving n with type HAS_GENRE.

Note that the syntax $-[r : HAS\_GENRE] \rightarrow$ indicates a directed link.

Exercise 5

Write and execute the following query:

```
MATCH (n:Movie)<-[:ACTED_IN]-(a:Actor)
RETURN n.title, count(a) AS nbActors
ORDER BY nbActors DESC
LIMIT 5;
```

This query returns the title and the number of actors of the 5 movies having the most actors.

Here is a more detailed explaination of the query:

- **RETURN** n.title, count(a) specifies that the query returns the title of each movie and the number of actors who played a role in the movie.

  Note that this is tantamount to the following SQL query:

  ```
  SELECT m.title, count(*)
  FROM movies_actors ma, movies m
  WHERE ma.movie_id = m.id
  GROUP BY m.title
  ```

  In Cypher, the clause GROUP BY is implicitly specified by the fact the the aggregate function `count(a)` is preceded by `n.title`. If we had only specified `count(a)`, the query would have returned the total number of links with type ACTED_IN across all nodes.

- count(a) **AS** nbActors specifies that the name *nbActors* is assigned to the result of the function *count*. This is very important because this result is used later in the query.

- **ORDER BY** nbActors **DESC** specifies that the list of the movies is sorted (ORDER BY) by the number of actors in descending order (DESC).

- **LIMIT** 5 specifies that the query returns only the first five movies.

3.1. Exercises on Cypher

For the following queries, click on the button *Rows* to see a textual representation of the queries.

Exercise 6

Write and execute the queries to obtain the following results:

1. The genres of the movies in the database. (See the result).
2. The number of movies in the database (See the result).
3. The title of the movies released in 2015. (See the result).
4. The number of directors by movie. Sort in decreasing order. (See the result).
5. The names of the directors and the title of the movies that they directed and in which they also played. (See the result).
6. The genres of the movies in which Tom Hanks played. (See the result).
7. The title and the rate of all the movies that the user number 3 rated. Sort by rate in decreasing order. (See the result).

3.2. Query Chaining

Cypher allows the specification of complex queries composed of several queries that are concatenated with the clause **WITH**. We are now going to see an example to obtain the titles of the movies that have been rated by at least 100 users.

At a first glance, the following query looks like a good solution:

```
MATCH (n:Movie)<-[:RATED]-(u:User)
WHERE count(u) >= 100
RETURN n.titre
LIMIT 5;
```

However, this query cannot be executed because we cannot use an aggregate function in the clause WHERE. The solution consists in chaining two queries with the clause WITH. The first returns the number of users that rated each movie; the second takes as an input the output (i.e., the result of RETURN) of the first and only returns the titles of the movies that had at least 100 rates.

```
MATCH (n:Movie)<-[:RATED]-(u:User)
WITH n, count(u) AS nbRates
WHERE nbRates >= 100
RETURN n.titre
LIMIT 5;
```

In this query, the variables that are the output of the first query (`n` and `count(u)`), and therefore the input of the second, are specified in the clause WITH. Note that we must assign an *alias* (with AS) to the result of an aggregation function specified in the clause WITH.

Exercise 7

Write and execute a query to obtain the five movies that obtained the best average rate among the movies that have been rated by at least 100 users. (See the result).

4. Second part : Movie recommendation

We are now going to see how Neo4j can be effectively used in a real application by implementing queries that form the basis of a simple movie recommendation system. This system is based on the notion of **collaborative filtering**. This consists in recommending a user u some films that s/he hasn't rated yet and other users with similar preferences have loved. In our context, we say that a user loves a movie if s/he rated the movie at least 3.

This concept is explained in the example of Figure 3. The user u loves 6 movies, 3 of which are also loved by the user v (the black nodes); it is reasonable to think that u may also love the two movies that v loved and u hasn't rated yet.

The principle of collaborative filtering is based on the computation of a **similarity score** between two users. Several similarity scores are possible in this context; here, we are going

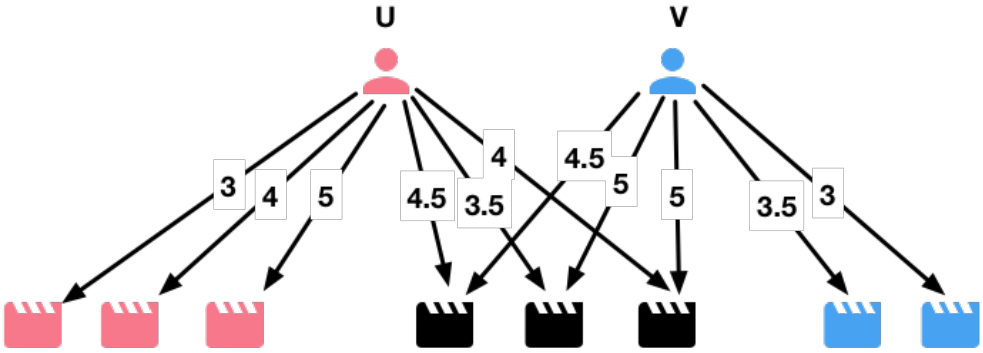

Figure 3: Collaborative filtering.

to focus on the **Jaccard coefficient**. Let $L(u)$ and $L(v)$ be the sets of movies that u and v love respectively; the similarity score $J(u,v)$ between u and v is given by:

$$J(u,v) = \frac{|L(u) \cap L(v)|}{|L(u) \cup L(v)|} \tag{1}$$

In order to recommend movies to a *target user* v, the recommender system computes the similarity score between v and all the other users of the system and proposes to v the movies that s/he hasn't rated yet and that the k most similar users loved.

We are now going to incrementally write a query to recommend some movies to the target user 3.

The first step consists in determining the value $|L(v)|$.

Exercise 8

Write and execute the query to obtain the number of movies that the user 3 loved.
This query must return the target user and the number of movies that s/he loves (See the result).

Next, we are going to determine the value $|L(u)|$, for all users u except v.

Exercise 9

Write and execute the query to obtain the number of movies that each user u loves, except the target user 3.
This query must return each user u and the number of movies that s/he loves (See the result).

We put the two queries together with the clause WITH.

Exercise 10

Compose the two previous queries with the clause WITH.
This query must return the target user 3, the number of movies that the target user 3 loves, the other users u and the number of movies that the users u love (See the result).
Reminder. The output of the first query is the input of the second; therefore, in the first query will need to replace the clause RETURN with the clause WITH.

Now, we need to determine the value $L(u) \cap L(v)$, for each user u, and compute the similarity score of Equation 1.

Exercise 11

Append (by using WITH) to the query written in Exercise 10 a query that obtains the number of movies that any user u loved and that the target user 3 loved too, and computes the similarity score (Equation 1) between the target user 3 and u. This query must return the five most similar users to the target user and the similarity scores (See the result).
N.B. Multiply the numerator of the Equation 1 by 1.0, otherwise Cypher will compute an integer division.

The last step consists in recommending some movies to the target user.

Exercise 12

From the previous query, take the identifier of the user w with the highest similarity to the target user. **You are going to use this identifier directly in the new query.** Write and execute the query to obtain the list of the movies that the user w loved and that the target user hasn't rated yet. Sort this list by decreasing rate (See the result).
Hint.

- First, write a query to obtain the list of the movies that the target user rated. In the MATCH clause, use the variable m to indicate a movie that the target user rated. Conclude the query with:

  RETURN collect(m.title) AS movies

  The function *collect* creates a list called *movies*.

- Replace RETURN with WITH in the previous query and add a second query to select the titles of the movies m that the user w loved and the target user did not rate. In order to exclude the films that the target user did not rate, use the following predicate

  none(x in movies where x=m.title)

  in the WHERE clause.

Query 6.1

romance
fantasy
horror
musical
comedy
war
imax
thriller
animation
adventure
.... (19 results)

Return

Query 6.2

9125

Return

Query 6.3

The Atticus Institute
Manson Family Vacation
Last Knights
The Jinx: The Life and Deaths of Robert Durst
Woman in Gold
A Very Murray Christmas
Schneider vs. Bax
Tomorrowland
Ex Machina
Truth
.... (183 results)

Return

Query 6.4

To Each His Own Cinema — 36
Life in a Day — 31
Paris, I Love You — 22
Movie 43 — 13
Fantasia — 11
New York, I Love You — 11
11’09"01 - September 11 — 11
Aria — 10
V/H/S — 10
Hamlet — 8
.... (8841 results)

Return

Query 6.5

Rick Moranis — Strange Brew
Jennifer Jason Leigh — The Anniversary Party
Steve Oedekerk — Kung Pow: Enter the Fist
Tony Jaa — Ong-Bak 2: The Beginning
Ralph Eggleston — For the Birds
Doug Sweetland — Presto
Ida Lupino — The Bigamist
Ida Lupino — On Dangerous Ground
Art Clokey — Gumby: The Movie
Paul Mazursky — Faithful
.... (470 results)

Return

Query 6.6

imax
sci-fi
drama
adventure
animation
children
comedy
fantasy
thriller
mystery
documentary
action
romance
war
crime

Return

Query 6.7

The Shawshank Redemption — 5
Forrest Gump — 5
Requiem for a Dream — 5
Fight Club — 5
The Princess Bride — 5
Pulp Fiction — 4.5
Flags of Our Fathers — 4.5
Titanic — 4.5
Letters from Iwo Jima — 4.5
The White Stripes Under Great White Northern Lights — 4
.... (151 results)

Return

Query 7

The Godfather — 4.4875
The Shawshank Redemption — 4.487138263665597
The Godfather: Part II — 4.385185185185182
The Usual Suspects — 4.370646766169154
Schindler's List — 4.303278688524591

Return

B. Second Part — Results

Query 8

{id:3} — 47

Return

Query 9

{id:138} — 60
{id:551} — 82
{id:147} — 35
{id:560} — 100
{id:219} — 125
{id:569} — 80
{id:228} — 55
{id:434} — 161
{id:93} — 151
{id:443} — 40
.... (670 results)

Return

Query 10

{id:3} — 47 — {id:294} — 875
{id:3} — 47 — {id:366} — 26
{id:3} — 47 — {id:375} — 21
{id:3} — 47 — {id:581} — 13
{id:3} — 47 — {id:617} — 57
{id:3} — 47 — {id:590} — 89
{id:3} — 47 — {id:249} — 19
{id:3} — 47 — {id:276} — 14
{id:3} — 47 — {id:25} — 19
{id:3} — 47 — {id:572} — 102
.... (670 results)

Return

Query 11

{id:146} — 0.1702127659574468
{id:379} — 0.15625
{id:623} — 0.1328125
{id:42} — 0.13
{id:658} — 0.12631578947368421

Return

Query 12

Toy Story
Star Wars: Episode IV - A New Hope
The Lion King
The Incredibles
Back to the Future
Animal House
Memento
Shrek
The Three Musketeers
WALL·E
.... (47 results)

Return