CentraleSupélec

SACLAY CAMPUS, COMPUTER SCIENCE DEPARTMENT
2019-2020

Models and Systems for Big Data

Graph Database: Neo4j & Cypher

Pr. Nacéra Seghouani

This lab assignment aims at putting into practice the main notions underlying graph databases using Neo4j[1] as graph database and Cypher as query language[2].

The dataset `recommendations.db`, **to be downloaded from Edunao**, describes a set of movies, the artists who acted in and directed these movies, the users who gave a score between 1 and 5 to movies based on whether they dislike or like them.

1 NEO4J DESKTOP & DATA IMPORT

First install Neo4j Desktop on your laptop from official site given in footnote. Launch Neo4j Desktop on your computer, an interface will be displayed in a window (see Figure 1), click on the default project, create a database named `Graph` (remember your login/password if required).

To import data, **copy the content of the repository** `recommendations.db` **to replace the content of the created repository** `graph.db`.

`graph.db` repository is located in `<neo4j-home>/data/databases` or in `<neo4j-home>\data\databases` (if Windows system). This repository depends on the system[3]. For example in MacOS the default repository is:
`user home/Library/Application Support/Neo4j Desktop/Application/neo4jDatabases/database-xxxxx/installation-3.5.12/data/databases/graph.db`.

Click on Start to connect to your database. Open the Neo4j Browser (see Figure 1).

The 'Overview" frame at the left in Figure 1 summarizes the node labels, the relationship types and the key properties. The frame at the top right is used to write Cypher queries. The frame at the down right displays query results.

[1] https://neo4j.com/download/
[2] https://neo4j.com/docs/cypher-refcard/current/
[3] https://neo4j.com/docs/operations-manual/current/configuration/file-locations/

Figure 1: Neo4j Desktop Interface, Graph created and plugins added

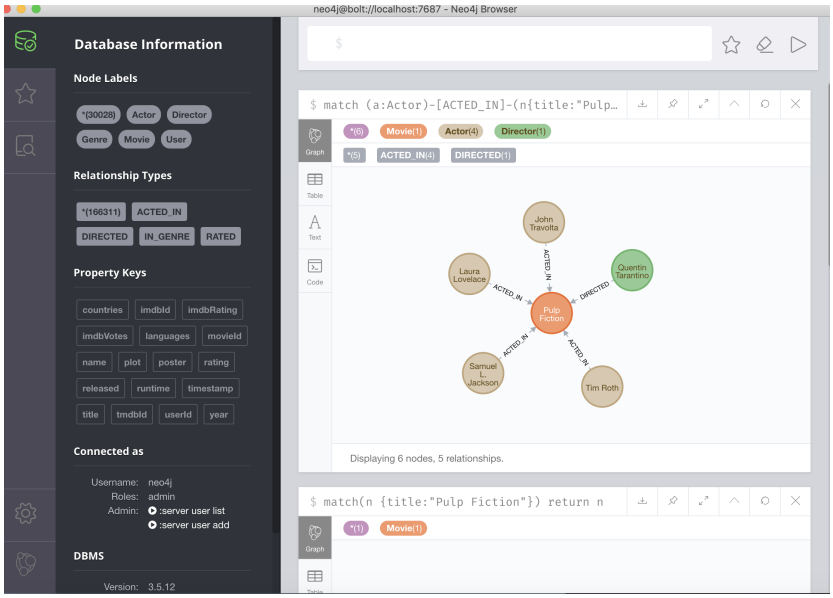

Figure 2: Neo4j Desktop Browser (Client)

Now, you can execute the following queries (**separately**) in the browser, to have the number of nodes, the number of different node labels, the number of relationships and the number of relationship types.

Some Analysis Queries

```
MATCH (n) RETURN COUNT(n)
COUNT(n)
30028
MATCH (n) RETURN LABELS(n), COUNT(*)
LABELS(n) COUNT(*)
["Movie"] 9125
["Genre"] 20
["User"] 671
["Actor"] 15873
["Director"] 4339
MATCH ()-[r]-() RETURN COUNT(r)
COUNT(r)
332622
MATCH ()-[r]-() RETURN TYPE(r), COUNT(*)
TYPE(r) COUNT(*)
"IN_GENRE" 40680
"RATED" 200008
"ACTED_IN"  71858
"DIRECTED" 20076
```

2 Querying Data Preliminaries

To explore the graph, we will write Cypher[4] queries in the top frame starting with prompt symbol $. What do you observe when you execute the following queries?

Example 1

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

- `MATCH` clause specifies a node pattern: nodes labelled `Movie` and having the property `title` with `"Toy Story"` value.
- `RETURN` clause specifies the matched nodes stored in `n` variable to be returned.
- `WHERE` clause specifies the conditions on `title` property value of `n`.

The result can be visualized as a text rather than a graph, as shown in Figure 3.

Movie node has the following properties:


```
{
  "languages": [
    "English"
  ],
  "year": 1995,
  "imdbId": "0114709",
  "runtime": 81,
```


[4] https://neo4j.com/docs/cypher-refcard/current/

Figure 3: Toy Story Query Results

```
  "imdbRating": 8.3,
  "movieId": "1",
  "countries": [
    "USA"
  ],
  "imdbVotes": 591836,
  "title": "Toy Story",
  "tmdbId": "862",
  "plot": "A cowboy doll is profoundly threatened and jealous when a new spaceman
   figure supplants him as top toy in a boy's room.",
  "poster": "http://ia.media-imdb.com/images/M/...",
  "released": "1995-11-22"
}
In this graph database, all nodes have a label: Movie, Actor, User, Genre, Director.
To see all  labels, click on Overview button.
```

Example 2

```
MATCH (n:Movie {title:"Pulp Fiction"})-[r]-(m)
RETURN n, m;
```

In this query the `MATCH` clause specifies a 2-length path pattern composed of a noden labelled Movie, `"Pulp Fiction"` as title value and any node `m` related by any relationship `r`. Note that the database stores the following directed relationship types:

- `IN_GENRE` relationship relates nodes with label `Movie` to nodes with label `Genre`.
- `RATED` relationship relates nodes with label `User` to nodes with label `Movie`. It has a property rating whose value is between 1 and 5 (click on a `RATED` to see the value of this property).
- `ACTED_IN` relationship relates nodes with label Actor to nodes with label Movie.
- DIRECTED relationship relates nodes with label Director to nodes with label Movie.

User nodes have a property userId; Genre, Actor and Director nodes have a property name.

Example 3

```
MATCH (n:Movie {title:"Pulp Fiction"})-[r:IN_GENRE]->(m:Genre)
RETURN n, m;
```

In this query the `MATCH` clause specifies a 2-length path pattern where `IN_GENRE` relationship relates a node `n` labelled `Movie` having `"Pulp Fiction"` as `title` value to a node `m` labelled `Genre`.

As Neo4j follows Property Graph Data Model, it should support only directional relationships (uni- or bi-directional). A pattern without direction means that the search engine will search the bi-directional relationships.

Example 4

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

In this query:

- `RETURN` clause specifies `n.title` followed by `count(a)`. The result is the title of the 5 movies having the most number of actors and the number of actors of each title. In SQL language we would write:

  ```
  SELECT m.title, count(*) AS nbActors
  FROM movies_actors ma, movies m WHERE ma.movie_id = m.id
  GROUP BY m.title
  ORDER BY nbActors DESC
  LIMIT 5;
  ```

  In Cypher, `GROUP BY` clause is implicitly specified in the aggregate function `count(a)`. As this function follows `n.title`, it is applied to each `n.title` value. The query without `n.title` would return the number of links of type `ACTED_IN` relating `Actor` nodes to `Movie` nodes.

- `AS nbActors` assigns a name to `count(a)` result.

- `ORDER BY nbActors DESC LIMIT 5` specifies that the number of actors are sorted in descending order and limited to 5.

Example 5

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

`WITH` clause specifies a pipe (chaining) between the `Match` output and the input of the next clause. In this case the input of `Where` clause is `Movie` nodes having incoming `RATED` links from `User` nodes and the number of links of each `Movie` node. The query returns the titles of movies that have been rated at least 100 times.

Note that we must assign a name to `count(u)` result to keep this result as input of the next clause.

Someone might try the following query. The use of an aggregate function in `WHERE` clause **is not allowed**. Indeed, the `WHERE` condition is applied to each matched result whereas the aggregate function needs to be applied to all the results; in this case all the `User` nodes found (matched).

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

3 First Part to Do: Data Exploration Queries

Write and execute the following queries:

Exercise 1. The genres of movies in the database

```
match (g:Genre) return g.name;
//distinct(g.name) if two Genre nodes have the same name
```

Exercise 2. The number of movies in the database

```
match (m:Movie) return count(m);
OR
match (m:Movie) return count(*);
```

Exercise 3. The title of movies released in 1995

```
match (m:Movie {year:1995}) return m.title;
OR
match (m:Movie) where m.year = 1995 return m.title;
```

Exercise 4. The number of directors by movie sorted in decreasing order

```
match (m:Movie)<-[:DIRECTED]-(d:Director) return m.title, count(*) as nbDirectors
order by nbDirectors desc ;

Or,
but the result could be different if it exists  movies with the same title

match (m:Movie)<-[:DIRECTED]-(d:Director) return m, count(*) as nbDirectors
order by nbDirectors desc ;
```

Exercise 5. The names of the directors and the title of the movies that they directed and they also played in

```
match (a:Actor)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Director)
where a.name=d.name return a.name, m.title;
```

```
the following query doesn't have answer because in this graph database
the same person is represented with two different nodes with
different label: Director, Actor

match (a:Actor)-[:ACTED_IN]->(m:Movie)<-[:DIRECTED]-(d:Director)
where a=d return a.name, m.title;
```

Exercise 6. The genres of movies that Tom Hanks played in

```
match (:Actor {name:"Tom Hanks"})-[:ACTED_IN]->(:Movie)-[:IN_GENRE]->(g:Genre)
return distinct(g.name);
```

Exercise 7. The title and the rate of movies rated by the user number 46 sorted by rate in decreasing order

```
match (:User {userId:"46}")-[r:RATED]->(m:Movie)
return m.title, r.rating order by r.rating desc;
```

Exercise 8. At most 5 movies that obtained the highest average ratings and that have been rated by at least 100 rates

```
MATCH (m:Movie)<-[:RATED]-(u:User)
WITH  m, count(u) as nbRates
MATCH (m)<-[r:RATED]-(:User) WHERE nbRates >= 100
RETURN m.title, avg(r.rating) as avgRate order by avgRate desc LIMIT 5;

The input of the second MATCH is a list of row, each row is (m, nbRates)
The second MATCH is applied to each (m, nbRates)
```

4 Second Part to Do : Recommendation System Queries

In this part, we want to write queries for a movie collaborative-based recommendation system. This consists in recommending a user u movies s/he hasn't rated yet that are liked by a similar user v. In this context, we consider that a user liked a movie if s/he rated it at least 3. Figure 4 shows that the user u liked 6 movies and 3 of them are also liked by the user v. On the basis of collaborative reasoning, the user u might like the other movies liked by the user v.

In order to compute a similarity score between two users, we can use the Jaccard index. Let $L(u)$ and $L(v)$ be the sets of movies that u and v liked respectively. The similarity score is defined as follows:

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

The recommendation system will propose to a target user the movies liked by the k most similar users according to Jaccard index.

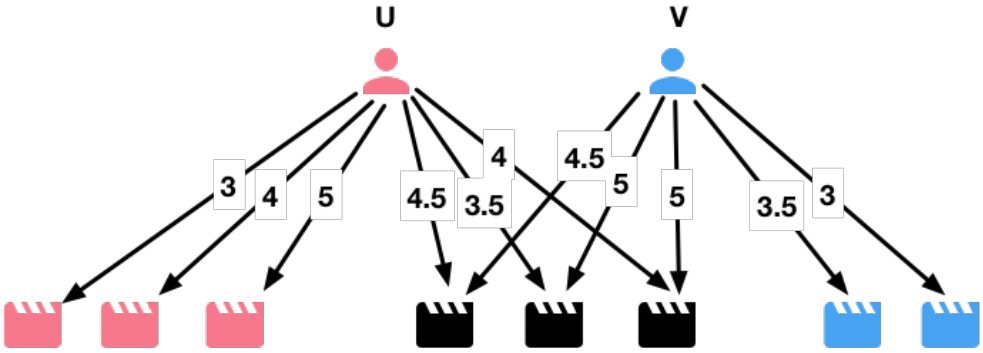

Figure 4: Collaborative Filtering

Exercise 1. To compute $|L(target)|$, write and execute the query which returns the target user (`userId:"3"`) and the number of movies s/he liked

```
match (target:User {userId:"3"})-[r:RATED]->(m:Movie)
where r.rating >= 3
return target, count(m);
```

Exercise 2. To compute $|L(other)|$, write and execute the query which returns each other user (except the target user 3) and the number of movies that each one liked

```
match (u:User)-[r:RATED]->(m:Movie) where u.userId <> "3" and r.rating >= 3
return u, count(m)
```

Exercise 3. To compute $|L(target) \cup L(other)|$, write and execute the query which returns the target user "3", the number of movies that the target liked, the other users and the number of movies that each one liked

☛ Use the `WITH` clause to chain the previous queries (See Example 2).

```
match (target:User {userId:"3"})-[r:RATED]->(m:Movie)
where r.rating >= 3
with  target, count(m) as lTarget
match (u:User)-[r:RATED]->(m:Movie)
where u.userId <> "3" and r.rating >= 3
return target, lTarget, u, count(m)

The input of the second MATCH is one row (target, lTarget)
```

```
each row of the second MATCH is joined with the input row
the variable m of the second MATCH could be named n this doesn't
change the result

This one is more performant
match ((target:User {userId:"3"})-[r1:RATED]->(m1:Movie)),
((m2:Movie)<-[r2:RATED]->(other:User))
where r1.rating >= 3 and r2.rating >=3 and other.userId<>"3"
with  target, count(m1) as lTarget, m2, other
return target, lTarget, other, count(m2)
```

Exercise 4. To compute $|L(target) \cap L(other)|$, write and execute the query which returns each user (except the target) and the number of movies s/he and the target user liked too

```
match (target:User {userId:"3"})-[r:RATED]->(m:Movie)<-[r1:RATED]-(u:User)
where r.rating >= 3 and r1.rating >= 3
return u, count(m)
```

Exercise 5. Write and execute the query which returns: (i) the five first users that are the most similar to the target user and (ii) their similarity score with the target user using $J(target, other)$ formula 1

- ☛ Use the `WITH` clause to chain the previous queries (Example 2).
- ☛ Multiply first the numerator before dividing, otherwise Cypher computes an integer division.

```
match (target:User {userId:"3"})-[r:RATED]->(m:Movie) where r.rating >= 3
with  target, count(m) as lTarget
match (other:User)-[r:RATED]->(m:Movie) where other.userId <> "3" and r.rating >= 3
with target, lTarget, other, count(m) as lOther
match (target:User {userId:"3"})-[r:RATED]->(m:Movie)<-[r1:RATED]-(other:User)
where r.rating >= 3 and r1.rating >= 3
 return other, count(m)*1.0 / (lTarget + lOther - count(m)) as sim
 order by sim desc limit 5
```

Exercise 6. Write and execute the query to obtain the list of the movies: (i) liked by the user with the highest similarity with the target (using the identifier value found in the previous query); (ii) that the target user hasn't rated yet. Sort this list by decreasing rate

- ☛ As `count()`, `collect()` is an aggregation function which returns a list of values.

☛ The expressions `x in list` and `not(x in list)` return a boolean value.

```
match (target:User {userId:"3"})-[r:RATED]->(m:Movie)
with collect(m) as movies
match (u:User {userId:"146"})-[r:RATED]->(m:Movie)
where r.rating >=3 and not (m in movies)
return m.title order by r.rating desc;
```