ADVANCED DATABASES

LAB 9 – GRAPH DATABASES with NEO4J

When you submit your final labs work, you are required to submit your code only in a text file.

INSTALLING NEO4J

You can install NEO4J (Community Edition) on your laptop by downloading it at https://neo4j.com/download/community-edition/

Then we will work with the web-based interface. Start NEO4J and then open your web-browser to http://localhost:7474/ and login.

REFERENCE MATERIALS

Quick introductory tutorials can be accessed from the browser interface directly by typing:

```
:play concepts  (introductory concepts)
:play cypher (introduction to the query language of NEO4J)
```

One page good tutorial: https://neo4j.com/developer/cypher-query-language/

A more comprehensive good tutorial can be found here:
https://www.tutorialspoint.com/neo4j/neo4j_cql_create_node.htm

MY QUICK REFFERENCE GUIDE FOR THE LAB:

In a nutshell, to do this lab you will need the following:

1. Create a node of a certain type with some attributes and values

Create a node of type "Student" (no attributes)

```
CREATE (s1:Student);
```

Create a node of type student with name Emil, age 29 from Sweden

```
CREATE (s1:Student { name: "Emil", country: "Sweden", age: 29 });
```

Multiple CREATE statement in one query:

```
CREATE (e2:Student { name: "Joe", country: "Ireland", age: 39 }),
 (e3:Student { name: "Mary", country: "Ireland", age: 24 }),
 (e4:Student { name: "Anne", country: "Ireland", age: 26 }),
 (e5:Student { name: "Nick", country: "Ireland", age: 28 });
```

2. MATCH

Match one or more nodes with specific type and specific attributes (remember every match needs a return statement to show what you are returning from the query)

Match all the nodes (any types)

```
MATCH (e) RETURN e;
```

Match all the “students” node

```
MATCH (e:Student) RETURN e;
```

Match all the students with age less than 25 yrs. WHERE is similar to the SQL one.

```
MATCH (e:Student) WHERE e.age < 25 RETURN e;
```

3. CREATE RELATIONSHIPS

Create a relationship (link) among nodes. First match the nodes and then create the relationship

From 1 node to another (from Zoe to Mark):

```
MATCH (e:Student {name:'Zoe'}), (r:Student {name:’Mark’})
CREATE (e)-[:FRIEND_OF]->(r);
```

From 1 node to many nodes with a single query (from Zoe to all the Swedish student):

```
MATCH (e:Student {name:'Zoe'}), (r:Student {country:'Sweden'})
CREATE (e)-[:FRIEND_OF]->(r)
```

From many to many with a single query (from all the Irish students to all the Swedish student):

```
MATCH (e:Student {country:’Ireland’}),(r:Student {country:'Sweden'})
CREATE (e)-[:FRIEND_OF]->(r)
```

Note that a relation can be directed (like -[:FRIEND_OF]->) or undirected (like -[:MARRIED]-). Undirected relationships are symmetrical but in NEO4J the create statement only works with directed relationships. However, when you visit (=query) the graph you can specify if you want to use the link as a directed or an undirected link (see section 5 visiting graphs for an example).

Remember that relationship can have attributes as well. For instance, to create a relationship between two students called “study_with” that has a property “when”:

```
MATCH (e:Student {name:'Zoe'}), (r:Student {name: 'Mark'})
CREATE (e)-[:STUDY_WITH {when: 'Friday'}]->(r)
```

Remember that you can link nodes of any type together (like a student with a subject)

```
MATCH (e:Student {name:'Zoe'}), (r:Subject {name: 'Math'})
CREATE (e)-[:LIKES {rating: 5}]->(r)
```

4. DELETE NODES

To delete nodes – even if they are connected by one or more relationships, use:

```
MATCH (c:City) DETACH
DELETE c
```

Detach is needed to detach the nodes from the relationship. The above command delete all the nodes of types city.

5. VISITNG THE GRAPH and SHOWING PATHS

To traverse (visit) a graph use the relationships defined. To return a path between two nodes use the `path` function matching the first and last node. You can match multiple nodes, in that case path will show all the possible paths
`Shortestpath` returns the shortest path.
Examples:
All the friends of Emil at a distance 3 steps maximum (note the use of the arrow to tell NEO4J that we want to use the direction of the relationship)

```
MATCH (e:Student {name:"Emil"})-[:FRIEND_OF*1..3]->(e2:Student)
RETURN e2
```

Same query but use the relationship as an undirected one (it can be traversed both ways). We do not use the ">" symbol in the quety.

```
MATCH (e:Student {name:"Emil"})-[:FRIEND_OF*1..3]-(e2:Student)
RETURN e2
```

Traverse the path and aggregate.

Find all the friends of Emil at a distance 3 steps maximum, and return the count of friends group by "country" (this is like the SQL: select country, count(*)….. group by country).

```
MATCH (e:Student {name:"Emil"})-[:FRIEND_OF*1..3]->(e2:Student)
RETURN e2.country, COUNT(e2.country) as numfriends
```

Show the path(s) between Alice and Mary:

```
MATCH
path = (e:Student {name:"Alice"})-[:FRIEND_OF*..5]->(m:Student
{name:"Mary"})
RETURN path
```

Shortest path between Alice and Mary:

```
MATCH
path = shortestPath((e:Student {name:"Alice"})-[:FRIEND_OF*..5]-
(m:Student {name:"Mary"}))
RETURN path
```

Show the path(s) from Alice to any other node (maximum distance is 5) and display the path, the name of the target friend, the length of the path, sort by name of the friend:

```
MATCH (e:Student {name:"Alice"})
MATCH path = shortestPath( (e)-[:FRIEND_OF*..5]-(m:Student ) )
RETURN path,m.name,length(path)
ORDER BY m.name
```

6. ACCUMULATOR and the REDUCE function

Use the **REDUCE** function to accumulate a value while visiting a path. The value could be an aggregation of one of the value of the nodes forming the path, or a value of an attribute of the relationship connecting the nodes on the path.

Reduce function syntax:

Syntax: REDUCE(accumulator = initial, variable IN list | expression)

Arguments:

- accumulator: A variable that will hold the result and the partial results as the list is iterated
- initial: An expression that runs once to give a starting value to the accumulator
- variable: a variable used in the loop
- list: An expression that returns a list
- expression: This expression will run once per value in the list, and produces the result value.

Example: supposed that you have a graph of cities, and cities are connected with a property "road" that has an attribute "distance". Each city has also a property "population".

Then to compute the distance from Dublin to Cork (as the sum of the distances of all the cities on the path(s) from Dublin to Cork (max 10 steps) is:

```
MATCH path =  (d:City {name:"Dub"})-[:ROAD*..10]-(c:City
{name:"Cork"})
RETURN
path,  REDUCE (tot = 0, n IN relationships(path) | tot + n.distance)
as tot_distance
```

To compute the total population of all the cities from Dublin to Cork (now we are adding up a property of the node, not of the relationship!)

```
MATCH path =  (d:City {name:"Dub"})-[:ROAD*..10]-(c:City
{name:"Cork"})
RETURN
path,  REDUCE (tot = 0, n IN nodes(path) | tot + n. population)  as
tot_population
```

Note that **relationships(path)** returns the information about all the links in the path, while **nodes(path)** returns all the information about the nodes in the path!

The full "city" example (just four cities and connected with 1 path) is here:

```
//Creation of nodes
CREATE (c:City {name: "Dublin", population: 1500000});
CREATE (c:City {name: "Carlow", population: 20000});
CREATE (c:City {name: "Wexford", population: 60000});
CREATE (c:City {name: "Cork", population: 200000});

//Creation of relationships

MATCH (c:City {name:"Dublin"}),(d:City {name:"Carlow"})
CREATE (c)-[:ROAD {distance:100}]->(d);
```

```
MATCH (c:City {name:"Carlow"}),(d:City {name:"Wexford"})
CREATE (c)-[:ROAD {distance:80}]->(d);

MATCH (c:City {name:"Wexford"}),(d:City {name:"Cork"})
CREATE (c)-[:ROAD {distance:160}]->(d);

//Queries

MATCH path =  (d:City {name:"Dub"})-[:ROAD*..10]-(c:City
{name:"Cork"})
RETURN
path,  REDUCE (tot = 0, n IN relationships(path) | tot + n.distance)
as tot_distance

MATCH path =  (d:City {name:"Dub"})-[:ROAD*..10]-(c:City
{name:"Cork"})
RETURN
path,  REDUCE (tot = 0, n IN nodes(path) | tot + n. population)  as
tot_population
```

Reduce function quick reference: https://neo4j.com/docs/developer-manual/current/cypher/#functions-reduce

A quick good reference guide on NEO4J queries: https://neo4j.com/docs/cypher-refcard/current/

LAB Exercise 1

Download the file ***friends.json*** and execute it in NEO4J web interface. The query will create a friends graph. Each node is a person and each person is described by a name (*name* field) age (*age* field), country of origin (*country* filed) and the sport they do (*sport* field). The nodes are connected by the FRIEND_OF relationship.

Check if everything was created correctly by running this query to show all the nodes of type Person:

```
MATCH (p:Person) return p;
```

You are required to:

- Add a person Tom (age: 28 from: Spain like: football). Make Tom a friend of Mary.
- Insert a new person (name: Bill, age: 23, country: Ireland) and make Bill a friend of Mary and Denis

Write the following query:

- Show the age of Denis and his friends
- Show all the person from Scotland
- Show all the person with age less or equal than 20 from Ireland
- Show all the person with age less or equal 30 playing football
- Count the person by country
- Show the average age of the person group by sport
- Show all the direct friends of Mary
- Show all the friends of Paul with a maximum distance of 5 steps
- Count all the friends of Paul with maximum distance 5 steps by nationality
- Show the path(s) between Paul and Lisa. For each path show the length. How many paths are there?
- Show the shortest path between Paul and Lisa.
- Show (if exists) a connection between Mary and all her friends, where the path can only contain persons that play football

  Remember that the friend relationship has a direction!

LAB Exercise 2

For this exercise you will need the **reduce** function of NEO4J used to accumulate an aggregated value (like a total, avg..) when visiting a path.

Frist, create a graph showing connections between airports.

Each node of type Airport has a field “country” , a 3 digits code (like “DUB”) and the name of the city.

Connect the airports with a relationship “**connected_to**”. The relationship has the following properties: time (in minutes) and price (in euro).

connected_to is an UNDIRECTED relationship, it can be traversed both ways.

This is the list of airports with city, country and code is the following:

(dublin,ireland,dub).

(cork,ireland,ork).

(london,uk,lhr).

(rome,italy,fco).

(moscow,russia,dme).

(hongkong,china,hkg).

(amsterdam,holland,ams).

(berlin,germany,txl).

(paris,france,cdg).

(newyork,usa,jfk).

(chicago,usa,ord).

(sao_paulo,brazil,gru).

(rio,brazil,gig)

The list of connections is the following (node_from,node_to,time,price):

(london,dublin,45,150)

(rome,london,150,400)

(rome,paris,120,500)

(paris,dublin,60,200)

(berlin,moscow,240,900)

(paris,amsterdam,30,100)

(berlin,dublin,120,900)

(london,newyork,700,1100)

(dublin,newyork,360,800)

(dublin,cork,50,50)

(dublin,rome,150,70)

(dublin,chicago,480,890)

(amsterdam,hongkong,660,750)

(london,hongkong,700,1000)

(dublin,amsterdam,90,60)

(moscow,newyork,720,1000)

(moscow,hongkong,420,500)

(newyork,chicago,240,430)

(dublin,sao_paulo,900,800)

(sao_paulo,newyork,840,650)

(rio,berlin,1200,1100)

Check that the graph has been created correctly.

You are required to:

- Submit the code (list of queries) to create the above graph

Execute the following queries:

- Find the total time from Moscow to Rio. Show also the path (airport connections)
- Show all the flights from Dublin to any destination and sort them by price (from the most expensive)
- Show what can be reached from Chicago in one or two steps (= direct flight or 1 change only)
- Show what can be reached from London in less than 240 minutes (4 hours).