GRAPH DATA MINING NEO4J

Mining Big Datasets
Project 2

Anna Xenaki, Eva Giannatou, Maria Vallila

Contents

1. Introduction 2
2. Dataset 2
3. Model 4
4. Data loading 6
4.1. Load CROSSROAD nodes 6
4.2. Load ROAD edges 6
4.3. Load POIs 7
4.4. Final graph 9
5. Queries 10
5.1. 1st query 10
5.2. 2nd query 11
5.3. 3rd query 11
5.4. 4th query 12
5.5. 5th query 13
5.6. 6th query 14

Figures

Figure 1 Graph database design 5
Figure 2 CROSSROAD nodes and ROAD edges 7
Figure 3 POI nodes and HAS_POI edges 9
Figure 4 Final graph 9
Figure 5 Output of 1st query 11
Figure 6 Output of queries 2 and 3 12
Figure 7 Output of queries 2 and 3 12
Figure 8 Output of 4th query 13
Figure 9 Output of 5th query 14
Figure 10 Output of 6th query 14

1. Introduction

Aim of this project is to get hands on experience on NEO4J and CYPHER.

The dataset used for this assignment contains California's road networks and points of interest. To be more precise the dataset consists of crossroads, roads connecting one crossroad to another and points of interest which are located in the road network. Each road may contain from none to several points of interest.

For this project purposes we transformed the dataset, we designed and created a graph database, we then loaded the dataset into the graph and finally we performed queries in order to answer specific questions. In this report we will describe in more detail the procedure which was mentioned above.

2. Dataset

We used an open source dataset provided by the University of Utah. This dataset consists of 6 separate files.

California Road Network's Nodes

This file contains crossroads which are characterized by their unique id, and their coordinates.

```
nodeID,longitude,latitude
0,-121.904167,41.974556
1,-121.902153,41.974766
2,-121.896790,41.988075
3,-121.889603,41.998032
4,-121.886681,42.008739
```

California Road Network's Edges

This file represents roads connecting one crossroad to another. Each road is described by a unique id, its length and two crossroads, the crossroad from where it begins and the one to which it ends

```
edgeID,startNodeID,endNodeID,distance
0,0,1,0.002025
1,0,6,0.005952
2,1,2,0.014350
3,2,3,0.012279
4,3,4,0.011099
```

California's Points of Interest

This file represents points of interest found in the road network. More specifically each point of interest is described by a unique id and its coordinates.

```
longitude,latitude,categoryID
-114.186394,34.308060,0
-114.430832,34.527500,0
-114.526672,33.869438,0
-114.575279,34.183891,0
-114.601936,34.819439,0
```

California's Points of Interest with Original Category Name

This file contains the name of the category in which each one of the points of interest belong.

```
categoryName,longitude,latitude
airport,-114.18639,34.30806
airport,-114.43083,34.5275
airport,-114.52667,33.86944
airport,-114.57528,34.18389
airport,-114.60194,34.81944
```

Merge Points of Interest with Road Network--Map Format

Finally, this file links the file containing the points of interest to the one containing the roads. It actually maps the location of each point of interest on the network. The first four attributes describe one specific road and the attributes which follow describe the points of interest located in the specified road. Each road is described by its start node, end node, length and its number of POIs. Meanwhile, for roads with number of POIs greater than zero this file contains information about each POI category and its distance from the start node.

File format before transformation

For each edge:
Start Node ID, End Node ID, Number of Points on This Edge, Edge Length.
For each point on this edge:
Category ID, Distance of This Point to the Start Node of This Edge

```
0 1 0.002025 0
0 6 0.005952 0
1 2 0.01435 2
25 0.00537134 55 0.0040002
2 3 0.012279 1
49 0.00474167
3 4 0.011099 3
33 0.000301746 48 0.00971851 54 0.00138718
```

The format of this file was tricky and we had to transform it into two files in order to load it into the graph. Python was used for this purpose. In the resulted format each separate row describes exactly one POI and the road in which this point is located.

Format of the 2 files which were formed after the dataset's transformation

- For each POI:

    - Edge Start Node Id , Category id , Distance
- For each number of Points of interest per road :
    - Edge Start Node Id , Edge End Node Id, Number of Pois

3. Model

The particularity of NEO4J is the fact that the design of the graph database is very flexible and it can completely adapt in the needs of each dataset. Thus, when designing the graph database we should take into consideration the nature of our dataset as well as the queries that we will need to perform.

We designed a directed graph consisting of 2 types of nodes and 2 types of edges. More specifically, we used nodes to represent the crossroads. Each crossroad had 3 properties, the crossroad's unique id and coordinates (longitude and latitude). Roads were represented by directed edges. ROAD edges linked pairs of CROSSROAD nodes and contained 3 properties, the road id, the length of the road and the number of POIs in each road. Nodes named POI_CATEGORY were also used to describe POIs. These nodes contained 2 properties, POI category id and POI category name. POI_CATEGORY nodes were linked to the starting node of the road which they belonged through directed edges named HAS_POI. HAS_POI edges contained information concerning the distance of each poi from the start node of the road where they belong.

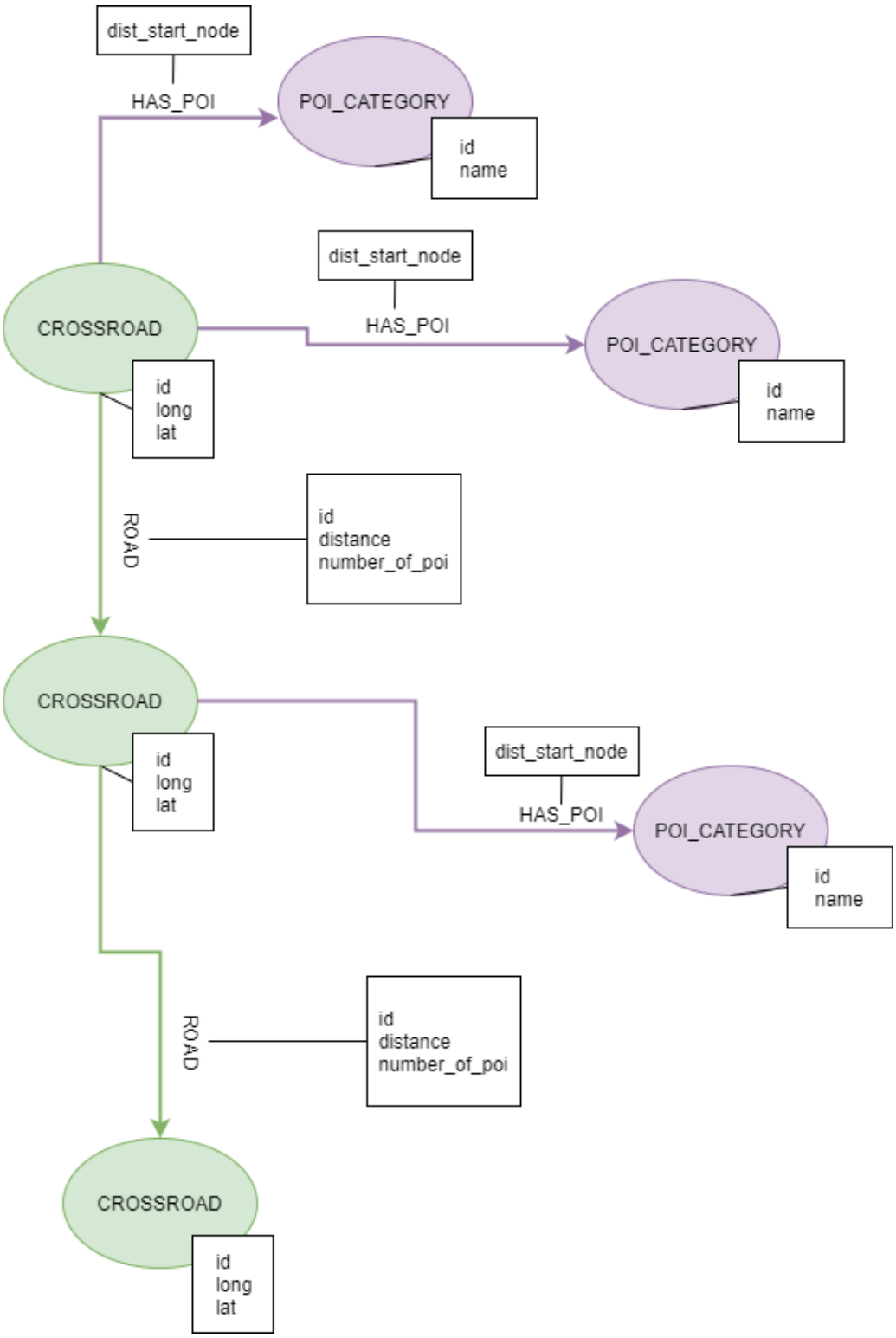

Figure 1 Graph database design

4. Data loading

After the design was finished we actually created the graph database on NEO4J according to what we have planned.

4.1. Load CROSSROAD nodes

Our first step was to create a unique constraint on the crossroad id. The same will be done for all the ids in the graph in order to make sure that each key will identify a unique record. Due to the fact that importing large amounts of data using LOAD CSV with a single Cypher query may fail due to memory constraints, we decided to use PERIODIC COMMIT. PERIODIC COMMIT processes specific number of rows at a time, while preventing running out of memory. PERIODIC COMMIT will also be used while loading all the other data files.

The first file to be loaded contained information about the crossroads and it was loaded into the CROSSROAD nodes. We used LOAD CSV for loading the nodes.csv file, headers included. The file was read line by line and each row was split into commas. We were able to refer to each column using line. followed by the specified csv column name. MERGE was used for creating the CROSSROAD nodes and the id, long and lat properties.

```
CREATE CONSTRAINT ON (c:CROSSROAD) ASSERT c.id IS UNIQUE;

// Load nodes

USING PERIODIC COMMIT

LOAD CSV WITH HEADERS FROM "file:///nodes.csv" AS line

MERGE(p:CROSSROAD { id: line.id, long: toFloat(line.long),
lat:toFloat(line.lat) })
```

4.2. Load ROAD edges

Like before, we created a unique constraint on the road id and used PERIODIC COMMIT. Then we loaded the csv containing the roads and used it in order to create ROAD edges connecting pairs of CROSSROAD nodes. For each ROAD edge we linked the id of the crossroad from where the road begins to the id of the crossroad to where it ends. Each ROAD edge contained the road id and the length of the road.

```
CREATE CONSTRAINT ON (r:ROAD) ASSERT r.id IS UNIQUE;
// Load edges
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS
FROM "file:///edges.csv" AS line
MATCH (start_node:CROSSROAD { id: line.start_node })
MATCH (end_node:CROSSROAD { id: line.end_node })
CREATE (start_node)-[r:ROAD]->(end_node)
SET r.id =line.id,
r.distance=toFloat(line.distance);
CREATE INDEX on :ROAD(start_node);
CREATE INDEX on :ROAD(end_node);
```

Figure 2 represents an example of how CROSSROAD roads and ROAD edges were formed on NEO4J.

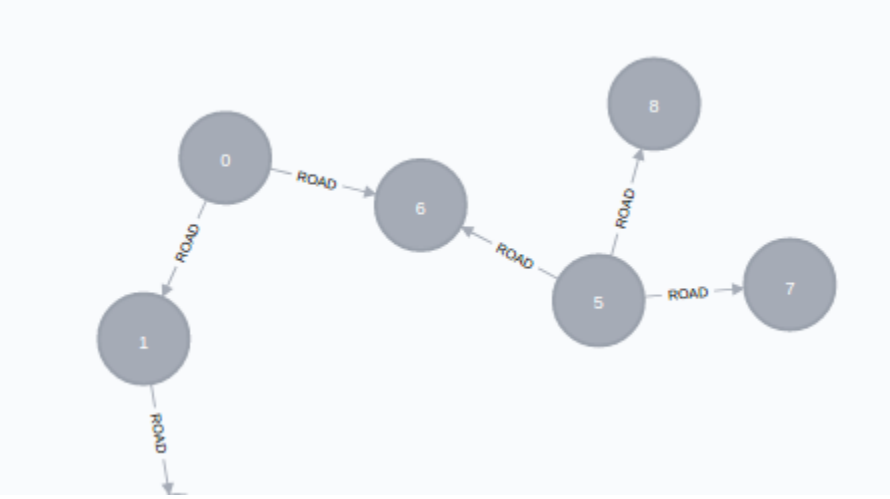

Figure 2 CROSSROAD nodes and ROAD edges

4.3. Load POIs

To load points of interest in our graph database we employ the use of python to convert provided data into a useful format.

To match category names with category ids we used excel feature of vlookup to match each category name to the respective category id. This file we named POI-name-id.csv and was imported in neo4j after the addition of points of interest in our graph database.

Next, we created nodes named POI and then we loaded has_poi.csv. POI nodes contained the category name and id in which the specified poi belongs. For matching POIs with start CROSSROADS we used the cat_id and start_node columns accordingly which were available in the file has_poi.csv. POI nodes were linked to the start nodes of the road in which they were located. This link was named HAS_POI and contained a property with the distance of the POI from the start node. We also added a new property to the ROAD relationship named number_of_poi, containing the number of POIs located in road of the relationship.

```
//load poi for csv load
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS
FROM "file:///has_poi.csv" AS line
MATCH (start_node:CROSSROAD { id:line.start_node})
CREATE (p:POI { id:line.cat_id})
CREATE (start_node)-[h:HAS_POI]->(p)
SET h.dist_start_node =toFloat(line.distance)
// Load poi category names
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:///POI-name-id.csv" AS line
MATCH(p:POI { id: line.cat_id})
SET  p.name=UPPER(line.cat_name);
CREATE INDEX on :POI(id);
CREATE INDEX on :POI(name);
//add number of poi
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:///has_number_of_poi.csv" AS
line
MATCH (start_node:CROSSROAD { id: line.start_node })
MATCH (end_node:CROSSROAD { id: line.end_node })
MATCH (start_node)-[r:ROAD]->(end_node)
SET r.number_of_poi=toInteger(line.number_of_poi)
```

Figure 3 represents an example of how POI nodes were linked to the start CROSSROAD node of the ROAD relationship where they belong.

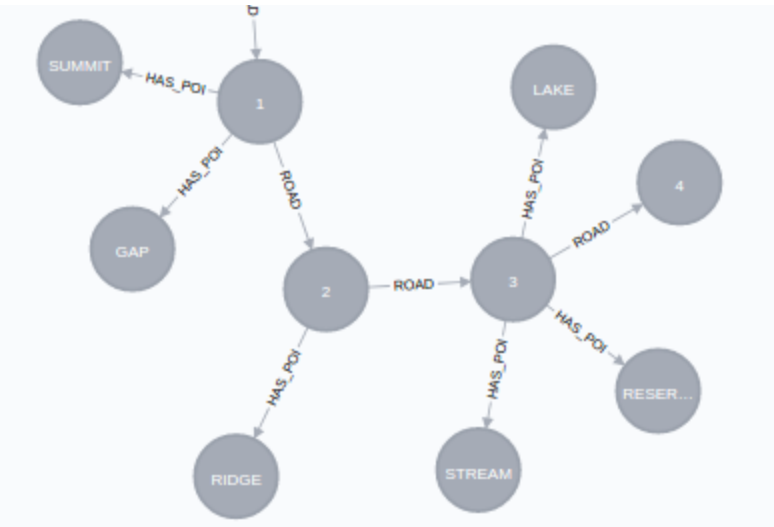

Figure 3 POI nodes and HAS_POI edges

4.4. Final graph

Figure 3 represents a small part of the final graph which was created in NEO4J.

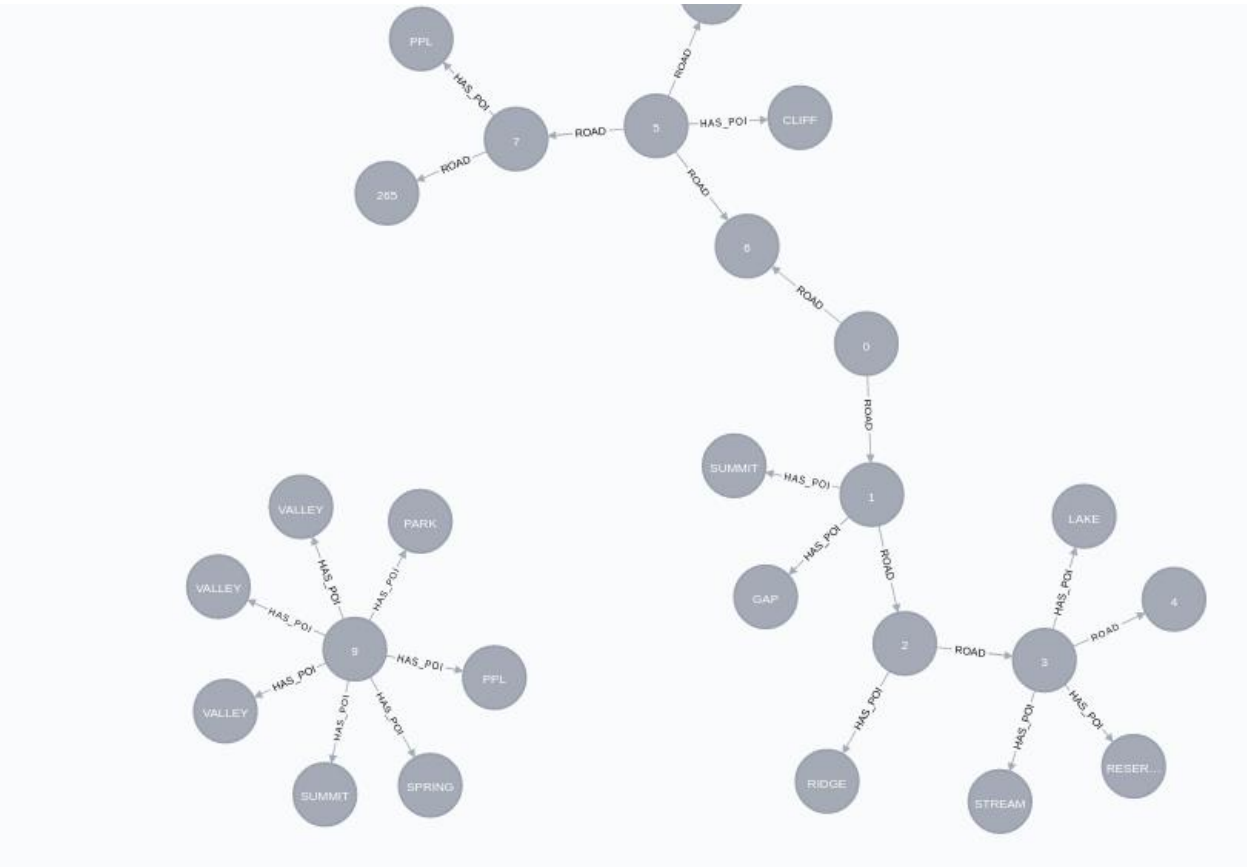

Figure 4 Final graph

5. Queries

5.1. 1st query

Which crossroad has the most points of interest and in which category do they belong (category name and number of pois for every category)? Sort them in descending order?

During the first part of the query, for each CROSSROAD node we counted the number of HAS_POI edges starting from this CROSSROAD node. Then in the second part for each HAS_POI edge of every node we collected the category names of the connected POI nodes.

We used COLLECT and UNWIND clauses in order to perform the union between the property category_name of POI node and the degree, or number of HAS_POI edges starting from each node. The 1st part returned a sorted list in descending order containing the node id and number of pois of this node. In the second part of the query we added the category name in the already sorted list of nodes with pois.

```
match (a:CROSSROAD)-[h:HAS_POI]-(c:POI)

with  a.id as id ,count(h) as degree ,collect(c.name) as
nameList

unwind nameList AS name

WITH id,degree,name, count(*) AS count

ORDER BY count DESC

with
id,degree,collect({category_name:name,number_of_pois:count}) as
list

return id,degree,list

ORDER BY degree DESC
```

Figure 5 Output of 1st query

5.2. 2nd query

Which is the shortest path between crossroads with id:10 and id:16?

This query calculates and returns the shortest path and the length of the shortest path between the 2 specified nodes. This query also answers the 3rd query, which asks 'Find the total distance of the shortest path from query 2.'

```
MATCH path=allShortestPaths((start_node:CROSSROAD {id:'10'})-
[ROAD*]-(end_node:CROSSROAD {id:'16'}))

RETURN length(path) as length,path
```

5.3. 3rd query

Find the total distance of the shortest path from query 2.

```
MATCH path=allShortestPaths((start_node:CROSSROAD {id:'10'})-
[ROAD*]-(end_node:CROSSROAD {id:'16'}))

RETURN length(path) as length,path
```

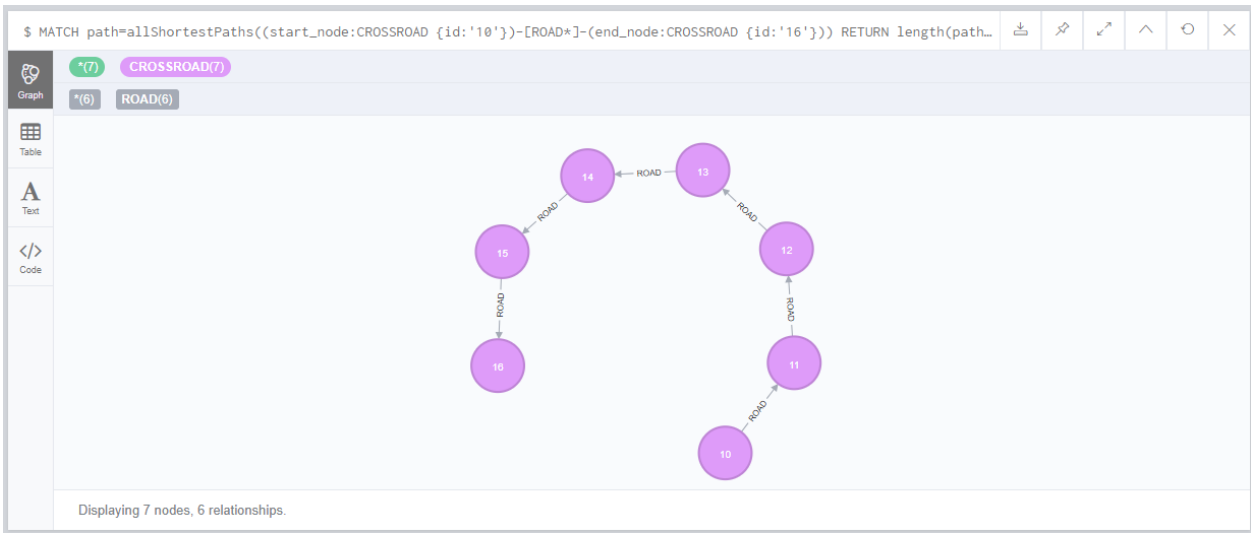

Figure 6 Output of queries 2 and 3

Figure 7 Output of queries 2 and 3

5.4. 4th query

Find the top three crossroads with the most bars.

We first had to find all the CROSSROAD nodes which were connected to at least one POI with category name 'BAR'. For each of these CROSSROADS we counted the number of HAS_POI edges which lead to BAR POI. Finally, the query returned a list with the ids of these CROSSROAD node followed the count of BAR POIs. This list was ordered by the number of bars in descending order and only the first 3 rows were kept.

```
match (a:CROSSROAD)-[h:HAS_POI]-(c:POI{name:'BAR'})
RETURN a.id,count(h) as degree
ORDER BY degree DESC
limit 3
```

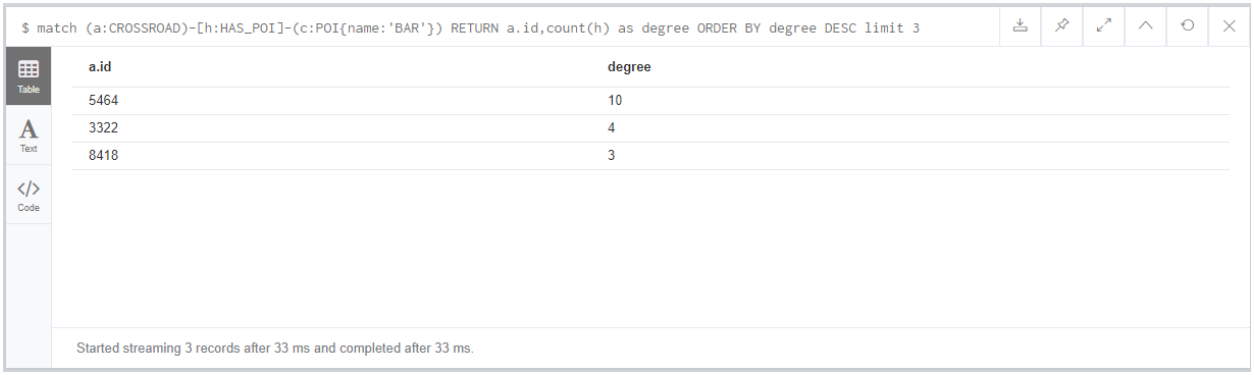

Figure 8 Output of 4th query

5.5. 5th query

Which is the shortest path between crossroads with id:10 and id:21 which passes through crossroad with id:17?

This query calculates the shortest path between crossroads with id:10 and id:17 as path1 and crossroads with id:17 and id:21 as path2. It then calculates the sum of the lengths of path1 and path2 and names it as length. It sorts the results in length's descending order and only keeps the first row which contains the shortest path between crossroads with id:10 and id:21 which passes through crossroad with id:17. Finally it prints the shortest path and its length.

```
MATCH path1=shortestPath( (a:CROSSROAD{id:'10'})-[r1:ROAD*]-
(c:CROSSROAD{id:'17'}))
MATCH path2=shortestPath( (c:CROSSROAD{id:'17'})-[r2:ROAD*]-
(b:CROSSROAD{id:'21'}))
RETURN path1, path2, LENGTH(path1) + LENGTH(path2) AS length
ORDER BY length desc
LIMIT 1
```

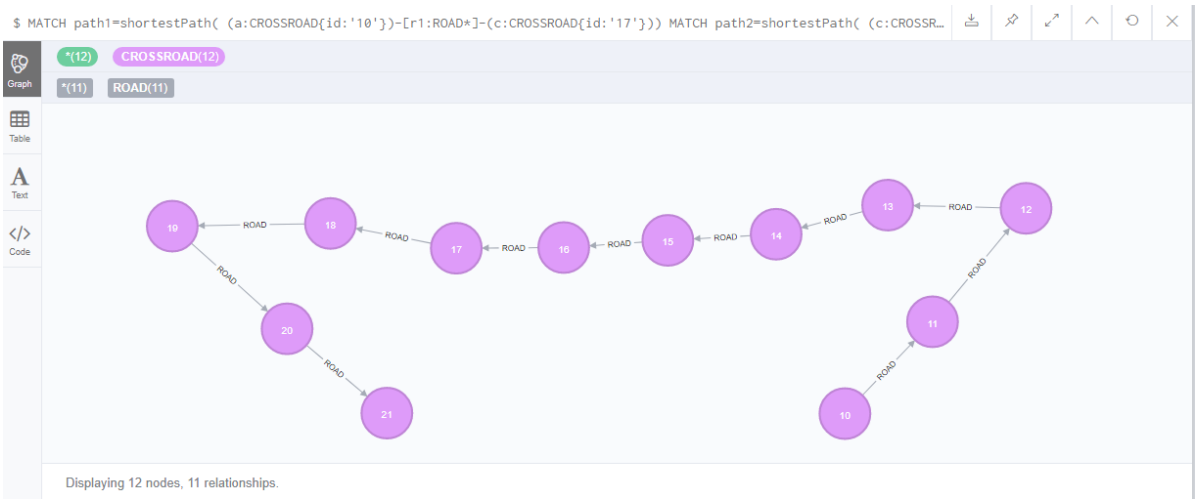

Figure 9 Output of 5th query

5.6. 6th query

Find the five closest crossroads to the point with Latitude: 39 and Longitude: -123.

We first defined a POINT with the given coordinates. Then we used the coordinates in order to calculate the Euclidean distance between all the CROSSROAD nodes and the given point with coordinates (39,-123). We then sorted the results by Euclidean distance and kept the top 5 rows which contained the five closest nodes to the point with Latitude: 39 and Longitude: -123.

```
MATCH (o:CROSSROAD)

WITH ({ id: 'POINT', long: -123, lat: 39 }) AS p1, ({ id: o.id,
long: o.long, lat: o.lat }) AS p2

RETURN  p1 as point, p2 as nearestCrossroads, sqrt((p1.long-
p2.long)^2 + abs(p1.lat-p2.lat)^2) AS dist

ORDER BY dist

LIMIT 5
```

Figure 10 Output of 6th query