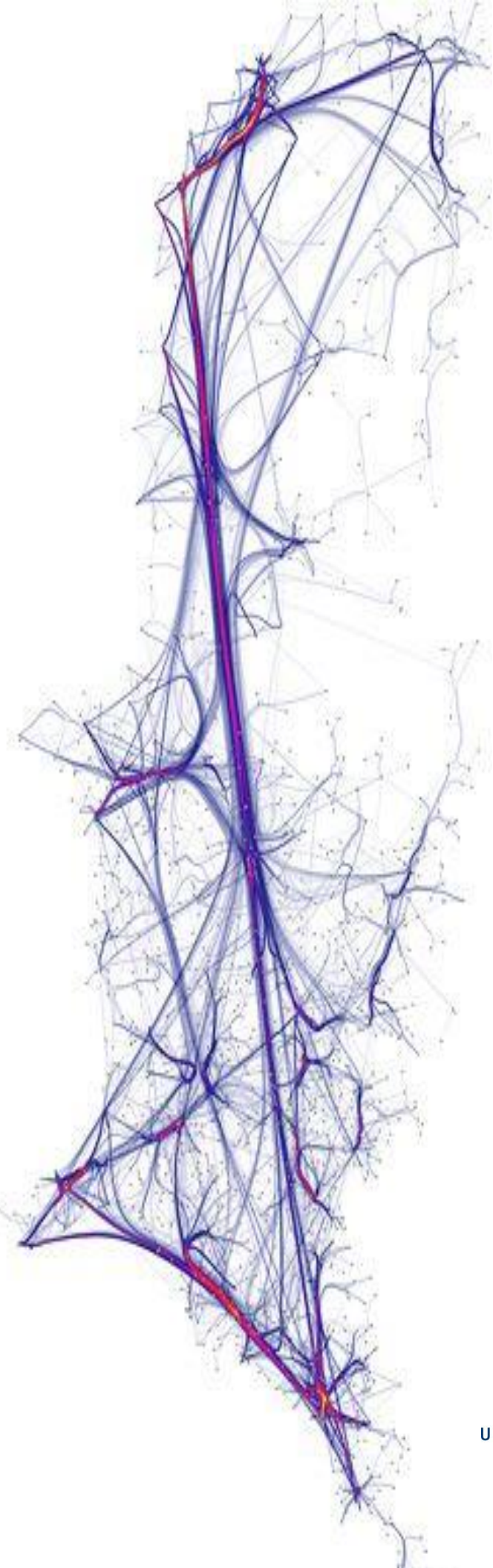


Master in Information Technology for Business Intelligence

Subject: Advanced Databases

Project Name:

(NEO4J)-[:IS A]->(GRAPH DATABASE)

Students:

- Andres Vivanco Villamar
- William Espinoza

12/23/2015

UNIVERSITÉ LIBRE DE BRUXELLES,
UNIVERSITÉ D'EUROPE

ULB

Table de Contents

1) Abstract 2
2) Background 3
2.1) Origin of Graphs 3
2.2) What is a Graph? 4
3) Graph Database 4
3.1) The Property Graph Model 4
3.2) Graph Database 5
3.3) Graph Compute Engine 7
3.4) Graph Database Vs Relational Database 8
4) Neo4J 10
4.1) Neo4J Features 10
4.2) The Cypher Query Languages (CQL) 11
4.3) Performance in NEO4J 19
4.4) Indexing and constraints for faster search 20
4.5) Neo4j Editions 20
4.6) Installation of Neo4j and two ways to use it. 21
5) Building a Graph Database Application 25
5.1) Selection of the topic: Electoral Roll and Friend's Relationship 25
5.2) Conceptual Model 26
5.3) Relational Model 27
5.3) Graph Model 28
5.4) Populating of the Databases 29
5.5) Comparative Queries (Neo4j vs SQL Server 2016) 31
5.6) Analysis of Results (Neo4j vs SQL Server 2016) 39
6) Conclusion 41

1) Abstract

With the needs to manage large and sparse datasets, with many kinds of relationships between them, new kinds of Database have been developed to supply it with a performance and capability better than the traditional databases technologies and queries languages.

Many of these new Kinds of Databases using graph structures like the main engine to allow to user to insert, update, query, delete and apply analysis techniques based in graphs in the networks of graphs.

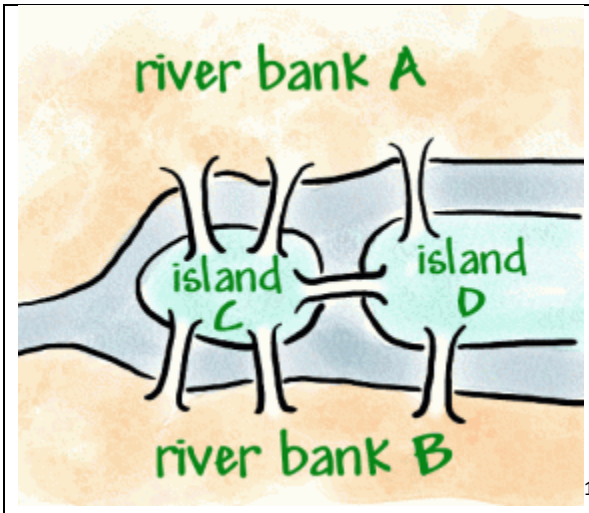
In this report we will look at from the origin of graphs, features of a graph database, passing for a technical comparison between Traditional Database and Graph Databases, a review of a graph database management system, and in the end the results of creating a Graph Database application to analysis advantages, disadvantages and a personal conclusion of this kind of technology using the most leading Graph Database named NEO4J.

Our topic selected to implement in a graph database is about “Electoral Roll” with a “Friend relationship information of the citizens”. This project is focusing in to test out by ourselves the best features of a graph database vs a classical relational database like SQL Server.

2) Background

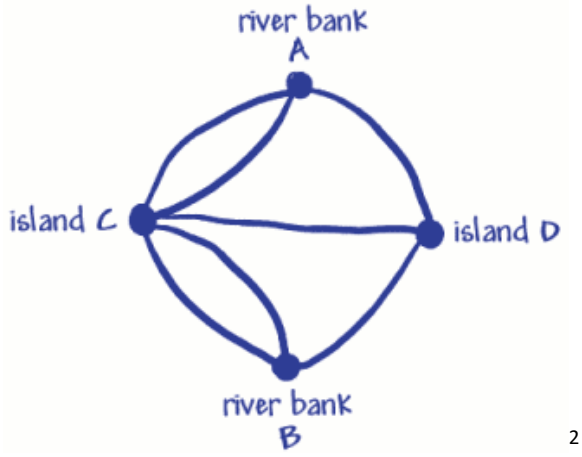
2.1) Origin of Graphs

In the 18th Century, the mathematician Leonhard Euler (1707-1783) could solve one of the most interesting problems in that time named "The Königsberg Bridge Problem".



Picture 1. Königsberg City

Konigsberg is a town on the Preger River, (before it was part of German, but now it is part of Russia).
The city has two river islands (C,D) with seven bridges connected to two bank areas (A,B) (left Picture[1]).
The problem was over the river Preger can all be traversed in a single trip without doubling back, also it needs ends in the same place it start.

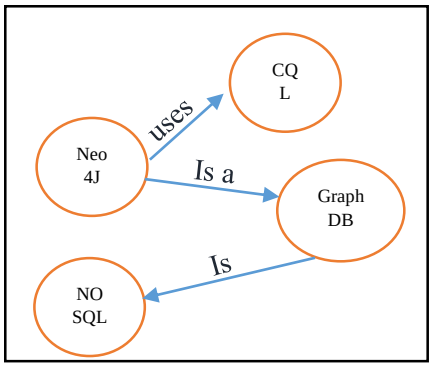


Picture 2. Königsberg in Graph

The problem is similar to asking if the multigraph on four nodes and seven edges (left Picture[2]) has an Eulerian cycle. This problem was solved in the negative by Leonhard Euler (1736), and represented the beginning of graph theory.
Although, Leonard Euler resolved the problem, the most important was the *mathematical basis* that he created to solve it.

[1] The Königsberg Bridge Problem, Königsberg City, NRICH math. Retrieved from https://nrich.maths.org/2484
[2] The Königsberg Bridge Problem, Königsberg in graph, NRICH math. Retrieved from https://nrich.maths.org/2484

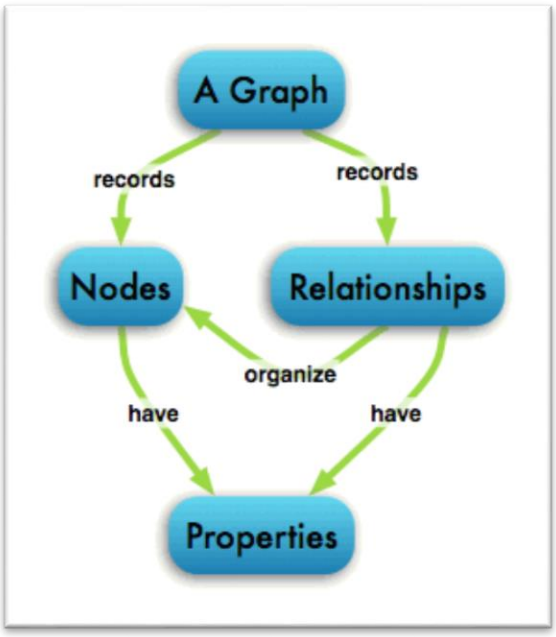
2.2) What is a Graph?



In a mathematical world a graph is a collection of vertices and edges, from the Computer Science and Database perspective a graph is a set of nodes and relationships that connect them. Entities are represented by nodes and the way how these nodes relate to the worlds are relationships. This concept allows to model many kind of scenarios, such as Social Network (friends of friends), Connections between places, etc.

3) Graph Database

3.1) The Property Graph Model



The property Graph Model looks similar to the Object Model or an Entity Relationship diagram. The property graph[3] has entities (nodes) connected between them, it could have some attributes (key-value-pairs). For expressing roles is useful labels tagged to the Nodes. In the same context, for attaching metadata, or indexing, or establishing constraint information could be using Labels too.

The Relationships (edges) represent the name and direction between two nodes (entities). A relationship need to have a direction, type of relationship, and start and end nodes. Also Relationships could have some properties. In a graph database the storing of relationships is stored efficiently, hence many relationships between nodes not will affect the performance.

[3] The Property Graph Model, NEO4. Retrieved from http://neo4j.com/developer/graph-database/

3.2) Graph Database

A Graph Database describes a model of Graph which has the methods: create, read, update and delete (CRUD) as part of the operation support. A Graph Database is an online platform and real time in nature, generally using it in transactional systems (OLTP).

A Graph Database model shows data in a fashion way comparing with others NoSQL models or type of Databases. The Graph Network is represented in the form of tree-structures or graphs that have entities (nodes) what are connected between them with and relationships (edges). This way of representation of the information allow to do operations easier to perform like for example data mining, cascade queries, short path between nodes, etc.

There are two important properties in a graph database:

- The Underlying storage
- The Processing engine

The Underlying storage

Exist Graph Database technologies that using *native graph storage*, which is optimized for managing and storing graphs. However, there are graph databases that storing graph data in a relational database, or in an object-oriented database or another kind of databases.

The processing engine

Generally, a graph database should use *index-free adjacency*, it is means that each node is connected physically to each other in the database. Some databases from User's perspective seems graph databases, because it exposes a graph data model through CRUD operations. However, from a technical view the importance *of index-free-adjacency* is a *native graph processing* is synonym of performance advantage.

There are two tradeoffs in the IT market about Graph Database, one is focused in *native graph storage*, the second is focused in *native graph processing*. Both of them have advantages and disadvantages. For instance, (see tables below)

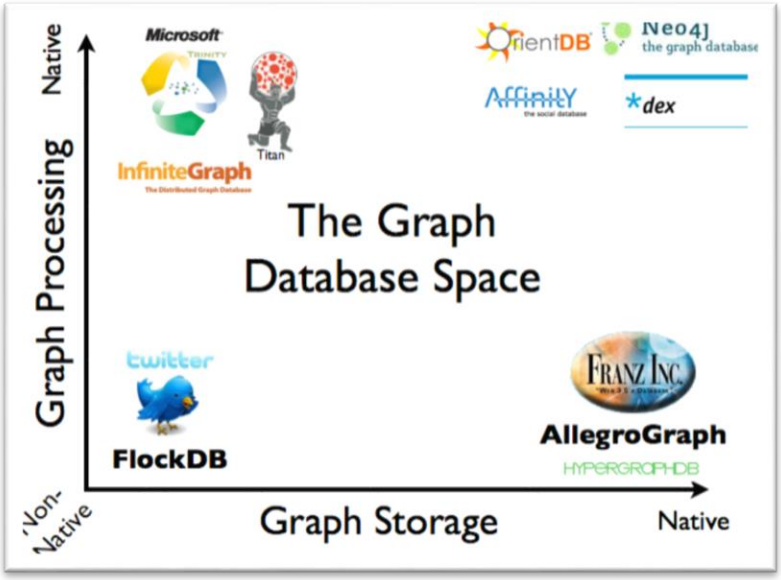
Property of Graph DB	Benefit
Native graph storage	Performance and Scalability
Nonnative graph storage	Possibility to use with a Well Known mature non graph backend (Ex, SQL server , MySQL)

Table 3.1.1 Benefit Native graph storage

Property of Graph DB	Benefit
Native graph processing	Traversal performance
Nonnative graph processing	Easy to make queries with intensive use of memory

Table 3.1.2 Benefit Native graph processing

The next picture 3.2[4] represents an overview of some graph databases on the market based in the storage and processing models

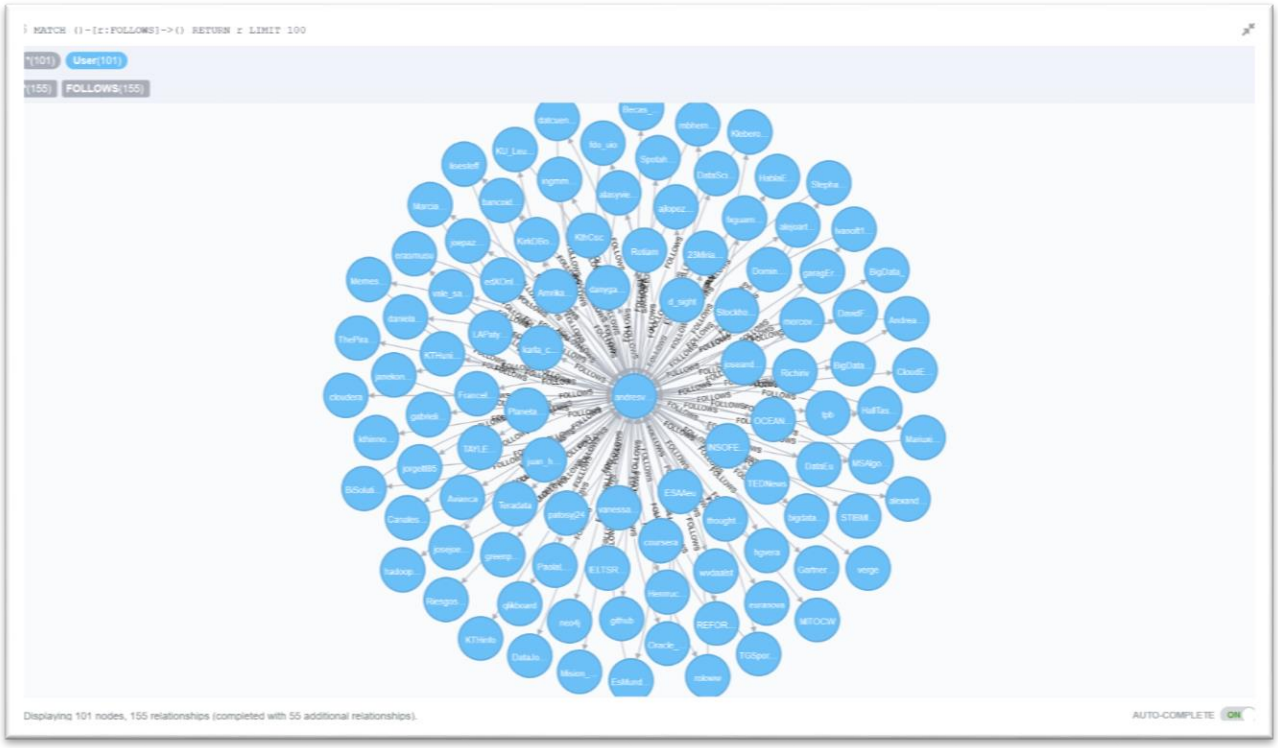


Picture 3.2 An overview of Graph Databases

[4] Ian Robinson, Jim Webber & Emil Eifrem (2015) – 2nd Edition. *Graph Databases*. p. 6. O' Reilly Media Inc., C.A. USA

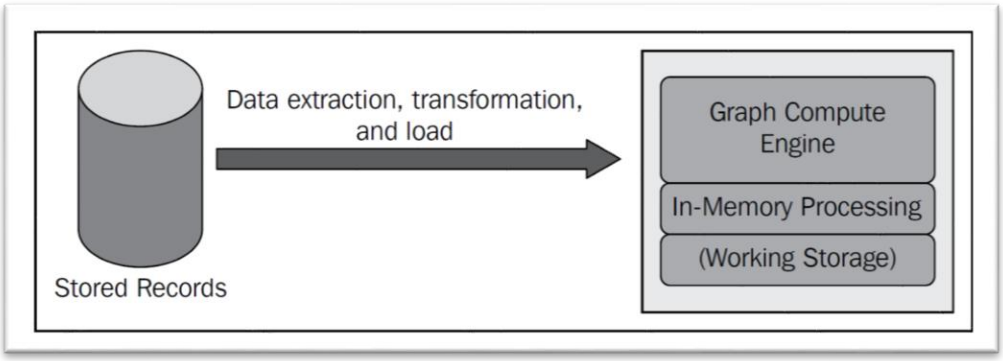
3.3) Graph Compute Engine

A graph compute engine is a technology for running graph computational algorithms in a big dataset. Graph Compute engines are designed mainly to recognize cluster in the data, or to know the numbers of relationships (edges), doing a special emphasis in queries like how many friends do you have, or how many friends or friends in different grades of deep. This is the main reason that Graph Databases are very useful to manage social networks. The next picture 3.3.1 is a Graph using neo4j, with the relationship “Follows” of the twitter account of the student Andres Vivanco limited to one hundred nodes.



Picture 3.3.1 Graph Network of “Follows” relationship of the twitter account of Andres Vivanco.

Global queries in graph compute engines are optimized for scanning and processing large connections of nodes in batches, very similar to another batch analysis technology such as datamining or OLAP. Some graph compute engines[5] (see picture 3.3.2) include a system of record (SOR) database with OLTP properties. Also a layer for processing data with is requested for an external application to respond the query with the results. A high-level overview of a graph computation engine setup



Picture 3.3.2. A high-level overview of a graph computation engine setup

3.4) Graph Database Vs Relational Database

From the 80s, Relational Databases have been the most useful databases of the software applications. A relational database stores structured data in tables with certain types of columns and a lot rows of the same type of information.

For references one table with another tables are necessary to set primary key attributes and foreign keys, to kept the referential integrity is necessary to do constraints. The cost for doing join queries is exponential.

This costly join operation with join tables, are usually focusing by denormalization of the data to decrease the numbers of joins necessary.

Relationships are the strongest point of *the graph database* comparing with another database management system, because each node in a graph database contains directly and physically a list of relationships-records, which represents the relationship with another node.

[5] Sonal Raj (2015). Neo4j High Performance. p.16. Pack Publishing Ltd. Birmingham, UK.

In other words, the Join operation in a Relational Databases is replaced in a graph database by itself, because the graph database just uses the list of relationships of each node in a direct way deleting the need for an expensive search or math computation.

This highlight of pre-materializing relationships, enable to the graph database do join queries with large amount of data from minutes (with relational database) to seconds (with graph database).

In theory, a graph database should be much faster than a relational database in graph traversal. To illustrate it, in a social network, the search friends of friends, while more deeply is the search of friends of friends the time execution of a graph database is better that relational database.

The above table 3.4 Time execution MySQL vs Neo4j [6] is the result of an experiment did for Aleksa Vukotic and Nicki Watt, authors of the book *Neo4J in Action*. This experiment consisted of in a social network, finding all the friends of a user's friend in different grades of depth. They ran queries in MySQL and NEO4J with a database of one millions of users. For it was used a 7–powered commodity laptop with 8 GB of RAM

Depth	Execution Time* – MySQL	Execution Time *– Neo4j
2	0.016	0.010
3	30.267	0.168
4	1,543.505	1.359
5	Not Finished in 1 Hour	2.132

Table 3.4. Time execution MySQL vs Neo4j

*Execution time is in seconds, for 1000 users

Meanwhile in depth 2 and 3 the results are not very surprising, the results of query 4 and 5 are really dramatic with a significant degradation of performance, especially in the depth 5 when MySQL was choked. The reason of it, is that to find friends of friend in a depth 5, the engine of MySQL need to calculate the Cartesian product of the table user_friend five times, for example a table with 50,000 records, the result will be $50{,}000^5$ rows, which is too much time for computing it, also it is necessary to discard more than 99% to return 1,000 records that we request.

[6] Aleksa Vukotic and Nicki Watt (2015). Neo4j in Action. Chapter 1. Manning Publications, USA.

4) Neo4J

4.1) Neo4J Features

Neo4J is the most leading graph database management system, it is implemented in Java and Scala. The source code is available in GitHub[7]. Successful cases of using Neo4j, including different type of industries such as matchmaking, analytic and scientific research, routing, network management, project management, and especially social networks. Etc.

The main feature is that neo4j not depend heavily on index because it supplies a natural adjacency by the graph. Neo4j using this locality to move through the graph. These operations could be kept with an excellent efficiency, crossing millions of nodes per second.

Graph Databases, specially Neo4j, don't depend heavily on indexes because it is supply

Some highlights of Neo4J are:

- ACID transaction compliance
- Materializing of relationships at creation time.
- Constant time for crossing of relationships.
- Developed on top of the Java Virtual Machine
- Memory caching for graphs and compact storage.
- Capability to manage billions of entities in a moderate computer.
- Easy data modeling
- It uses a visualization framework for the representation of data and query results
- Compatible bindings for Python, Java, Ruby and others.
- Disk based storage manager optimized
- It is highly scalable.
- It has a powerful traversal framework for better performance
- It is completely transactional in nature.
- Supporting features as JTA, 2PC, XQ, Transaction Recovery, Deadlock detection
- Neo4J can traverse graph depths of more than 1000 levels in a few seconds
- Neo4j uses Cypher Query Languages
- Easy to write queries about relationships with many types of deep.

[7] Neo4j Source code: https://github.com/neo4j/neo4j

4.2) The Cypher Query Languages (CQL)

The Cypher Query Languages is a 'declarative' language, in another words it means that a user does not need to indicate how to go to a node, just the user needs to ask which is the node to study.

CRUD operations in NEO4J (Create, read, update, delete)

Neo4j stores *entities* (i.e. *Person, City*) in nodes, theses nodes are connected to each other by relationships (edges) (i.e. Person *"is friend of"* Person, or City *"is part of"* State). Nodes and relationships could be defined with properties or metadata with key-value pairs.

The next are the commands to each CRUD operation.

Create

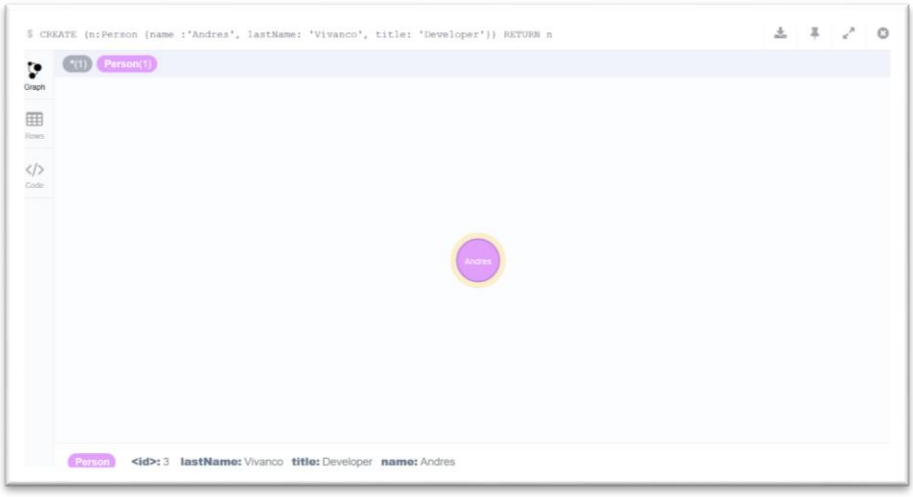
Creating a ***node*** Person with three properties

- name:'Andres'
- lastName: 'Vivanco'
- title: 'Developer'

Code:

```
CREATE (n:Person {name :'Andres', lastName: 'Vivanco',
                  title: 'Developer'}) RETURN n
```

Result in Console:



Creating a ***node*** Person with three properties

- name:'William'
- lastName: 'Esponiza'
- title: 'Engineer'

Code:

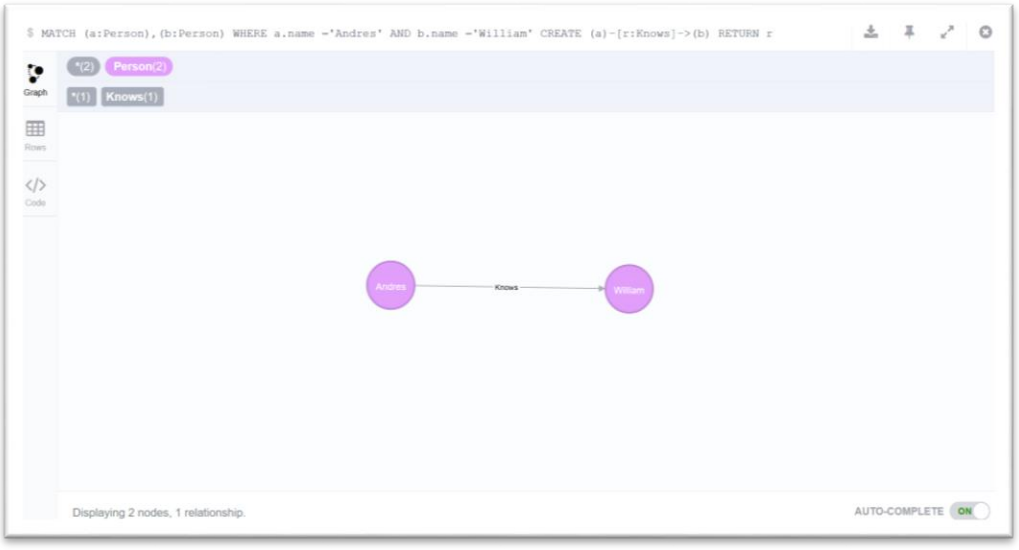
```
CREATE (n:Person {name :'William', lastName: 'Espinoza',
                  title: 'Engineer'})
RETURN N
```

Creating a ***relationship*** named 'knows'

Code:

```
MATCH  (a:Person),(b:Person)
WHERE a.name ='Andres' AND b.name ='William'
CREATE (a) -[r:Knows]->(b)
RETURN r
```

Result in Console:



Read

Read the node named ‘William’ and return the title.

Code:

```
MATCH (n:Person)
WHERE n.name = 'William'  RETURN n.title
```

Result in console:

Update

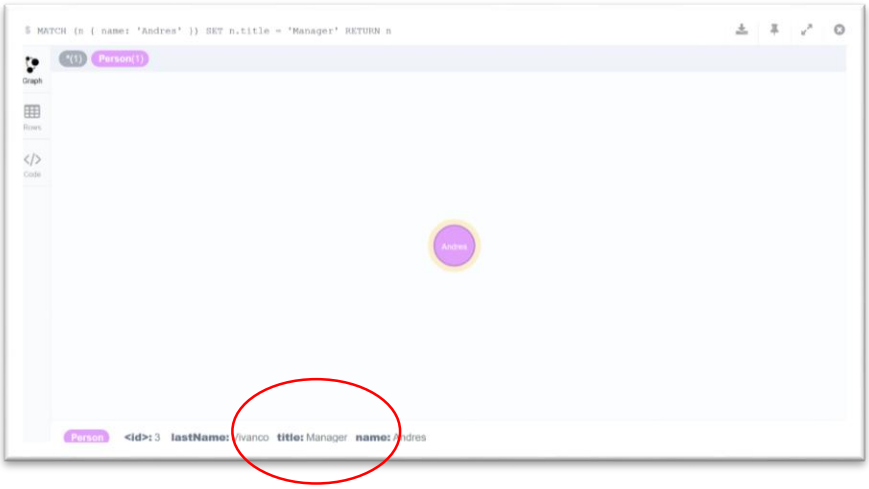
Update the node named ‘Andres’ with the next:

- title: 'Manager'

Code:

```
MATCH (n { name: 'Andres' })
SET n.title = 'Manager' RETURN n
```

Result in console:



Delete

Delete the node (Andres) with all its relationships

Code:

```
MATCH (n { name: 'Andres' })
DETACH DELETE n
```

Result in console:

Outstanding operations, queries and functions in Neo4j.

There are some operations, queries and functions that could be used in Neo4j for doing analysis optimized or loading data in the graph database properly.

A continuation the most relevant:

Importing CSV files with Cypher

CSV files with nodes and relationships could be store on the graph database indicating the Path in the computer or a URL, Neo4j support load csv via https, http, and ftp.

For loading **nodes,** the code is the next:

```
LOAD CSV WITH HEADERS FROM
'http://neo4j.com/docs/2.3.1/csv/artists-with-headers.csv' AS
line CREATE (:Artist { name: line.Name, year: toInt(line.Year)})
```

Result in console:

IMPORTANT for Importing large amounts of data is necessary to write previously "**USING PERIODIC COMMIT** ", it will optimize the loading and doing commit each 1000 rows per default. The numbers of rows could be set, for example do commit after each 500 rows, the command is in this form: **USING PERIODIC COMMIT** 500.

For loading **nodes using periodic commit** the code is the next:

```
USING PERIODIC COMMMIT 500

LOAD CSV WITH HEADERS FROM
'http://neo4j.com/docs/2.3.1/csv/artists.csv' AS line

CREATE (:Artist { name: line.Name, year: toInt(line.Year)})
```

Result in console:

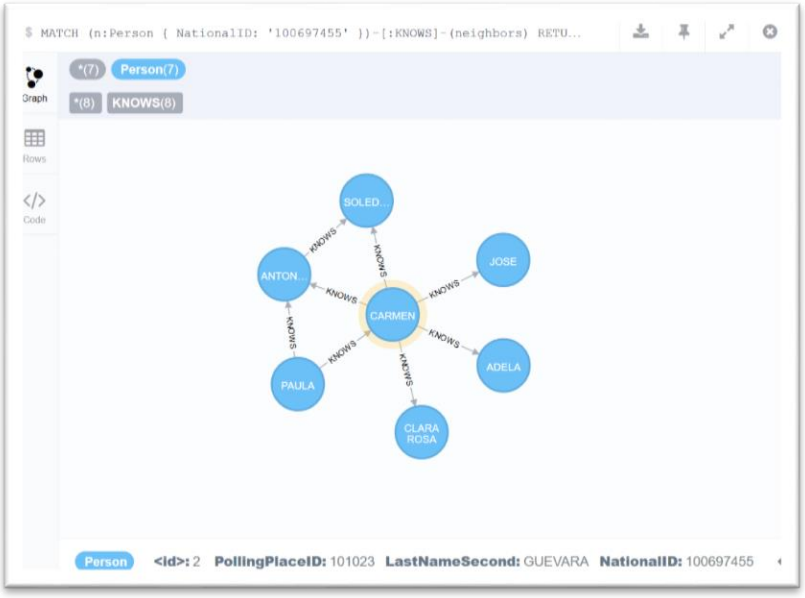
Find related neighbors

This code allows to find “neighbors” of a node, in this case we given the NationalID of a node of Person.

Code:

```
MATCH (n:Person { NationalID: '100697455' })-[:KNOWS]-
(neighbors) RETURN n, neighbors
```

Result in console:



\

Variable length paths

This code allows to find "neighbors" of a node, in different grades of depth for example we can define friends of friends (depth 2), friends of friends of friends (Depth 3), etc. For do this we need to set the level in the next way, we will find friends with the relationship "Knows":

Depth	Code	Explanation
2	[:KNOWS*1..2]	Friends of friends
3	[:KNOWS*1..3]	Friends of friends of friends
*****	[:KNOWS*]	Infinite friends of friends, depends how many friends of friends of friends…. Exists!

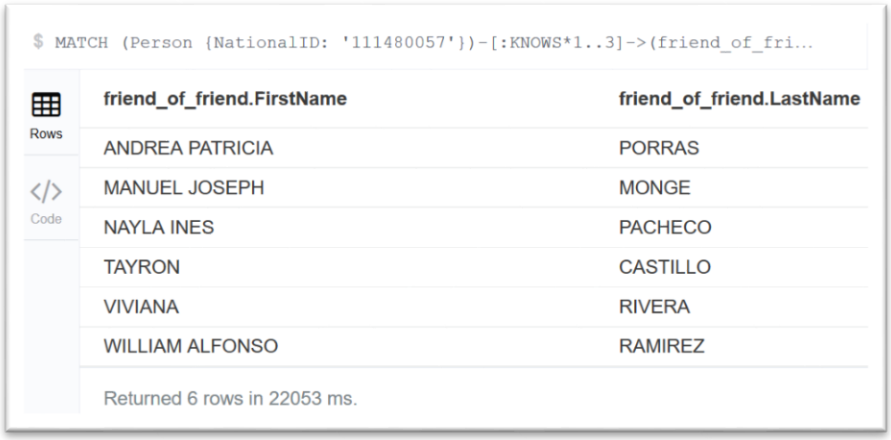
Code:

```
MATCH (n:Person { NationalID: '100697455' })-[:KNOWS*1..3] ]->(friend_of_friend) RETURN DISTINCT
friend_of_friend.FirstName,

friend_of_friend.LastName

ORDER BY friend_of_friend.FirstName ,
friend_of_friend.LastName
```

Result in console:



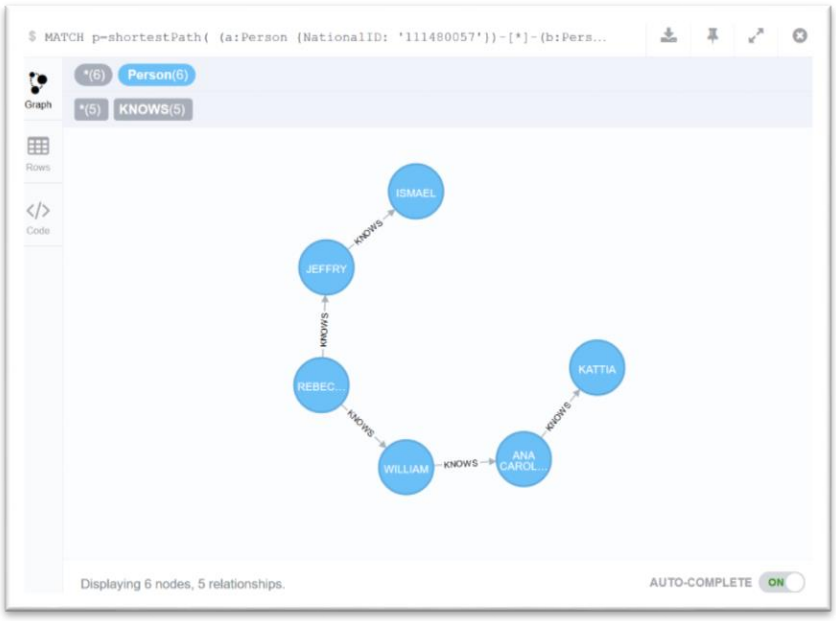
Shortest path

This code shows the shortest path, between two nodes, in another words, the shortest path with the less relationships needs, the way needs to start in one node and finish with the another one.

Code:

```
MATCH p=shortestPath(a:Person {NationalID: '111480057'})-[*]-
(b:Person {NationalID: '111480065'})
)
RETURN p
```

Result in console:



4.3) Performance in NEO4J

These are some tips for ***Tune Neo4j for maximum performance***:

- Ascertain if Neo4J Java process has enough memory. If the JVM heap resident needs more memory, then the OS will swap it out to storage. When occurs a garbage collection, it will be swapped out, and this swap-trashing effect has a negative impact on the performance of Neo4j. In steady-state, a well-tuned Neo4j database does not need to have any swap activity.
- Ascertain if the Java Virtual Machine has enough memory, the next values are recommended. Open the JVM with *–server* flag and *-Xmx<good sized heap>*, for example in good sized heap try with the maximum memory possible, one best one is *Xmx4g* for 4GB (considering that currently a new laptop has 8GB or 12 GB of memory), Sometimes a too large heap could be affect the performance, so try by yourself the best heap sizes in your case.
- Ascertain that neo4j is using a concurrent garbage collector, one of the best values is: *-XX:+UseG1GC*.
- Ascertain that file caching memory is enough to fit the entire store, set in neo4j.properties the values of *dbms.pagecache.memory, it value could be based in the next formula:*
    - *dbms.pagecache.memory = ((totalnodes * 15)+ (totalrelationships * 34) + (number of properties *64))*

- Ascertain if the size of the JVM heap is correct for your database application, it could be set in the file: conf/neo4j-wrapper.conf . The attribute *wrapper.java.maxmemory could be set with the next values of the picture 4.3*[8] Guide Lines for Heap Size*, recommended by Neo Technology*

Guidelines for heap size

Number of entities	RAM size	Heap configuration	Reserved RAM for the OS
10M	2GB	512MB	~1GB
100M	8GB+	1-4GB	1-2GB
1B+	16GB-32GB+	4GB+	1-2GB

Picture 4.3 Guide Lines for Heap Size.

[8] Guide Lines for Heap Size. http://neo4j.com/docs/stable/performance-guide.html

4.4) Indexing and constraints for faster search

Queries in Neo4j could be optimized if the data is indexed, and also applying some constraints. With this trick we will avoid redundant matches and does directly to the desired index location.

For applying index on a label the code is the next:

- CREATE INDEX ON: Person(NationalID)

On the another hand, to create constraints for example unique values is with the next code:

- CREATE CONTRASTRAINT ON n:Person

  ASSERT n.NationalID is UNIQUE

Managing Index and constraints will be more efficient the queries, especially search large amount of data.

4.5) Neo4j Editions

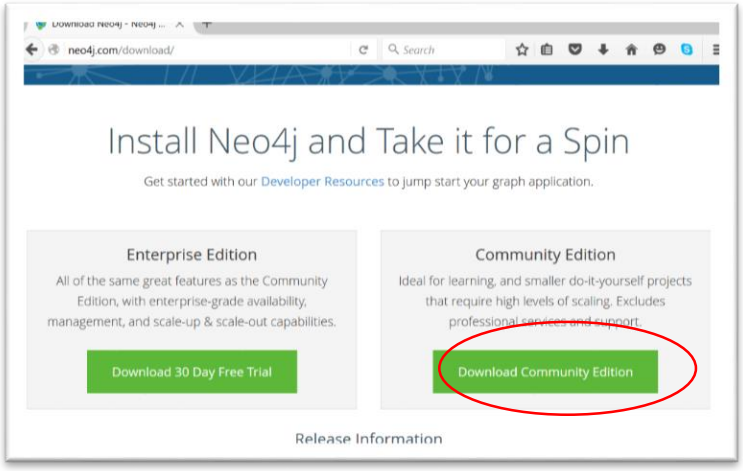
Neo4j has 2 types of licenses:

- **Community Edition.** It is free and open source, is a high performance with whole features described in the chapter 4.1.
- **Enterprise Edition.** Include all features of chapter 4.1 and also include scalable clustering, fail-over, high-availability, cache sharding, live backups, and comprehensive monitoring.

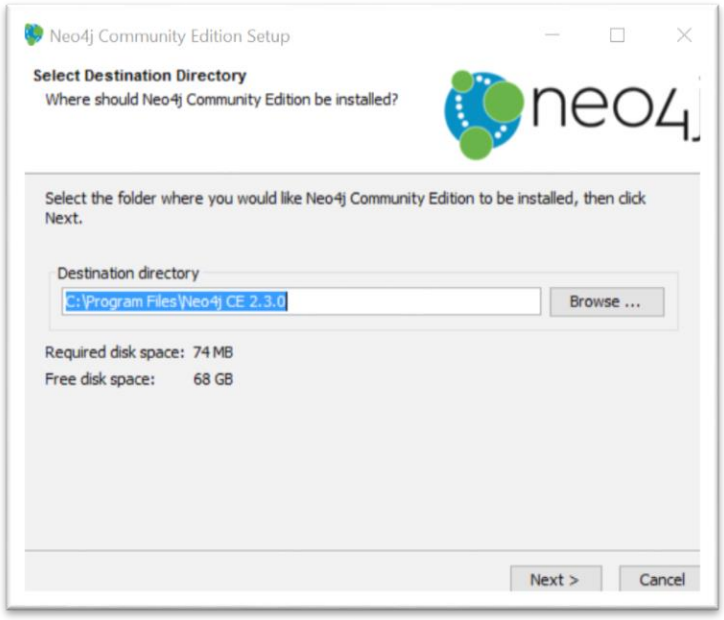
4.6) Installation of Neo4j and two ways to use it.

For our experiment we used Neo4j Community Edition v.2.3[9]:

1.) Download the last version available of Neo4j from http://neo4j.com/download/

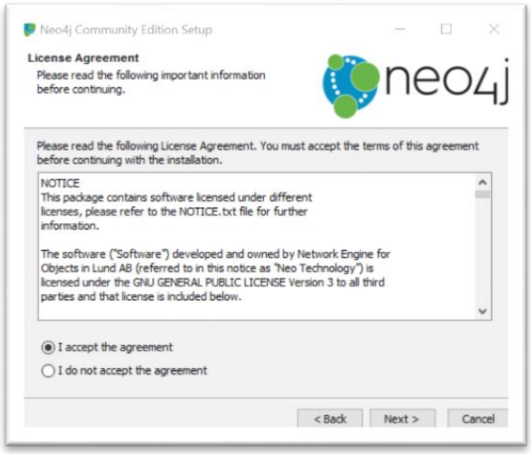


2.) Open the Installer and select the folder where it will be installed

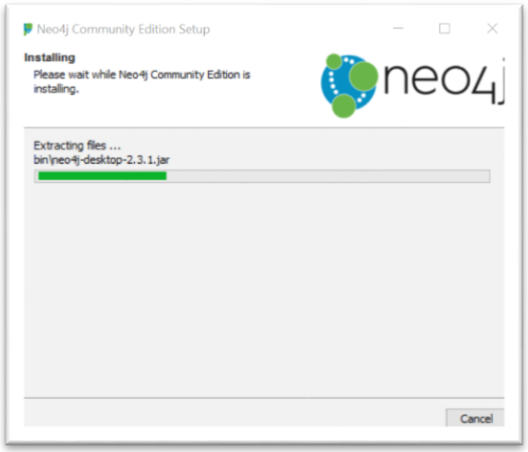


[9] Neo4j Community Edition v.2.3 http://neo4j.com/download/

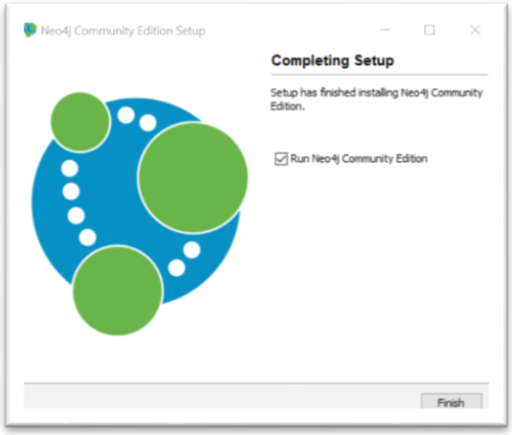
3.) Accept the agreement, and next, next



4.) Wait until it finishes to install all components



5.) Click in Finish and open Neo4j

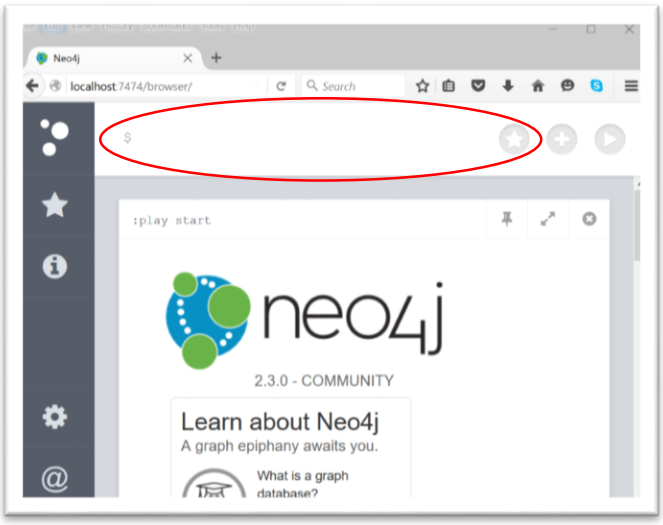


6.) When you open neo4j, please select in the bottom “choose”, the folder when you want to work. Each different folder, is like a different database

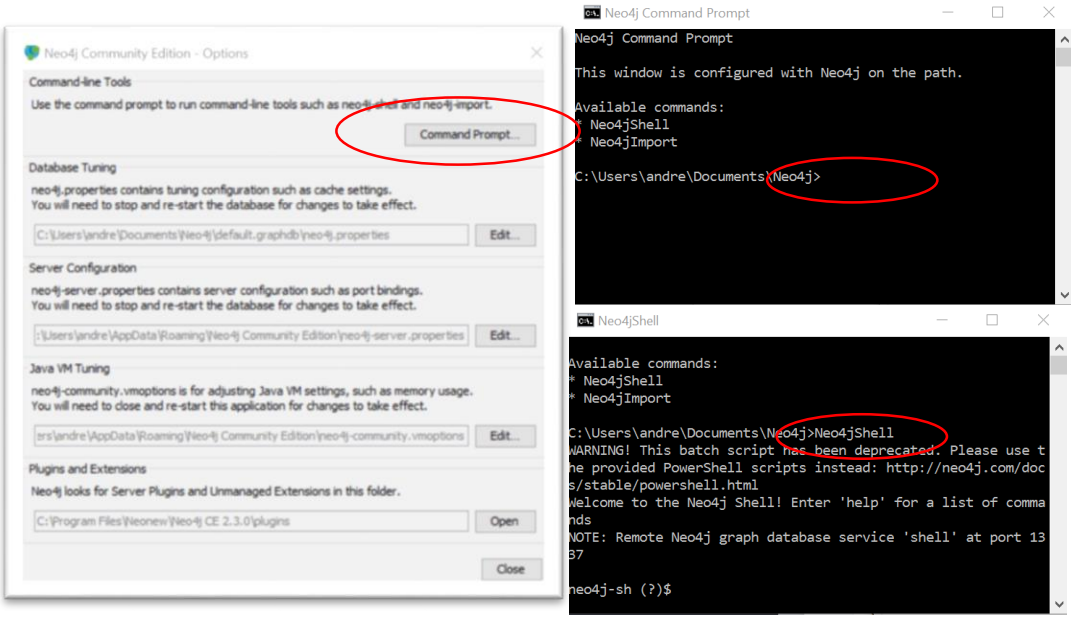
7.) Do click in Start , now is running

For working in neo4j, there are two ways one is in the browser, the another one is in shell console:

1.) For using in a browser, do click in the link in the green box or open in a browser with the next url: http://localhost:7474/browser/ . For writing queries is in the red circle



2.) For using from a shell console, to do click in the bottom Options, after do click in bottom Command Prompt , in the shell write “Neo4jShell” and Enter.

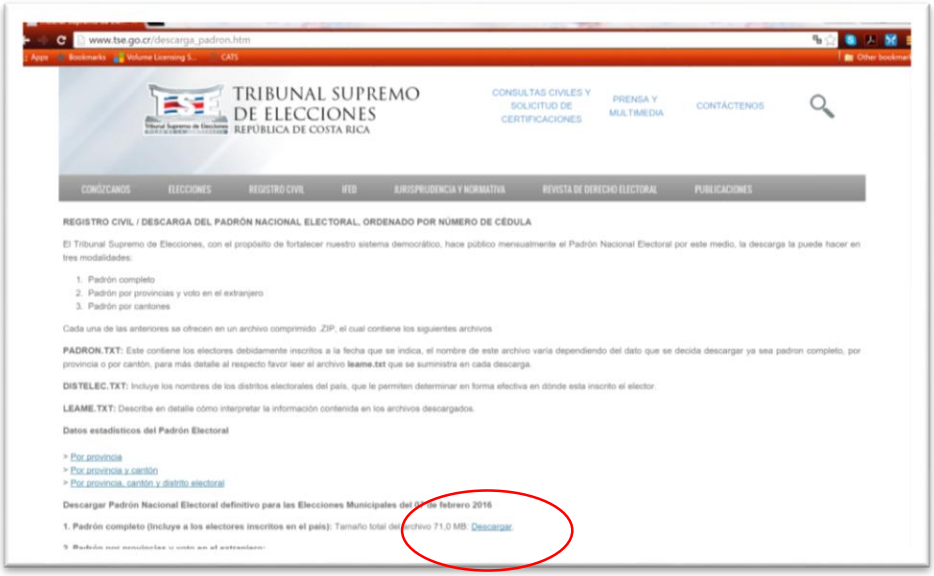


5) Building a Graph Database Application

5.1) Selection of the topic: Electoral Roll and Friend's Relationship

The Database application selected to work with Neo4J is about **"Electoral Roll"** adding by ourselves manually information of **friend's relationship** between citizens to evaluate main features of graph databases and comparing with a classic relational database like SQL Server 2016.

The input data downloaded is public information of Costa Rica[10] about Electoral Roll of 2015.



The present information was downloaded of the website of the Supreme Electoral Tribunal of Costa Rica (*or in Spanish Tribunal Supremo de Elecciones de la Republica de Costa Rica*) contains:

- Citizens (People in our model): 3.198.597,00
- Polling Places (Number of Districts): 2.123,00
- Cities: 124,00
- Provinces: 8,00
- Relationship (People VOTING ON districts) 3.198.597,00

[10] Electoral Roll of Costa Rica (2015) http://www.tse.go.cr/descarga_padron.htm

It's important to considerer that in the data downloaded also exist information about citizens who voting in different embassies of Costa Rica around the world, and the information about the place where they voting is:

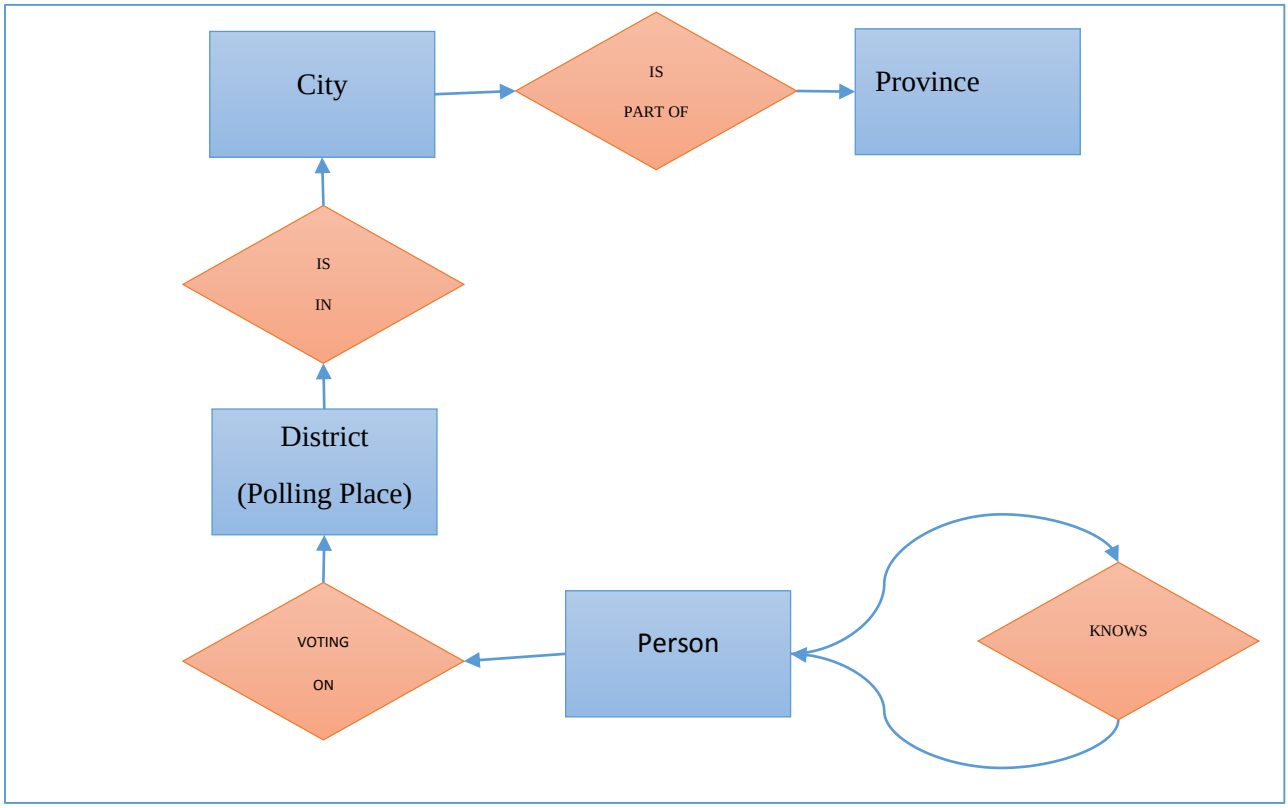
Tables or nodes of Places	For people living abroad
District	City of the Embassy
City	Country of the Embassy
Province	Static value 'CONSULADO'

Like we said previously, we also created "invented" data about "Friend's relationship" with the name "Knows" to simulate that one person "Knows" to another person. The numbers of this relationships are:

- Relationships ('Knows'): 3.764.822,00

5.2) Conceptual Model

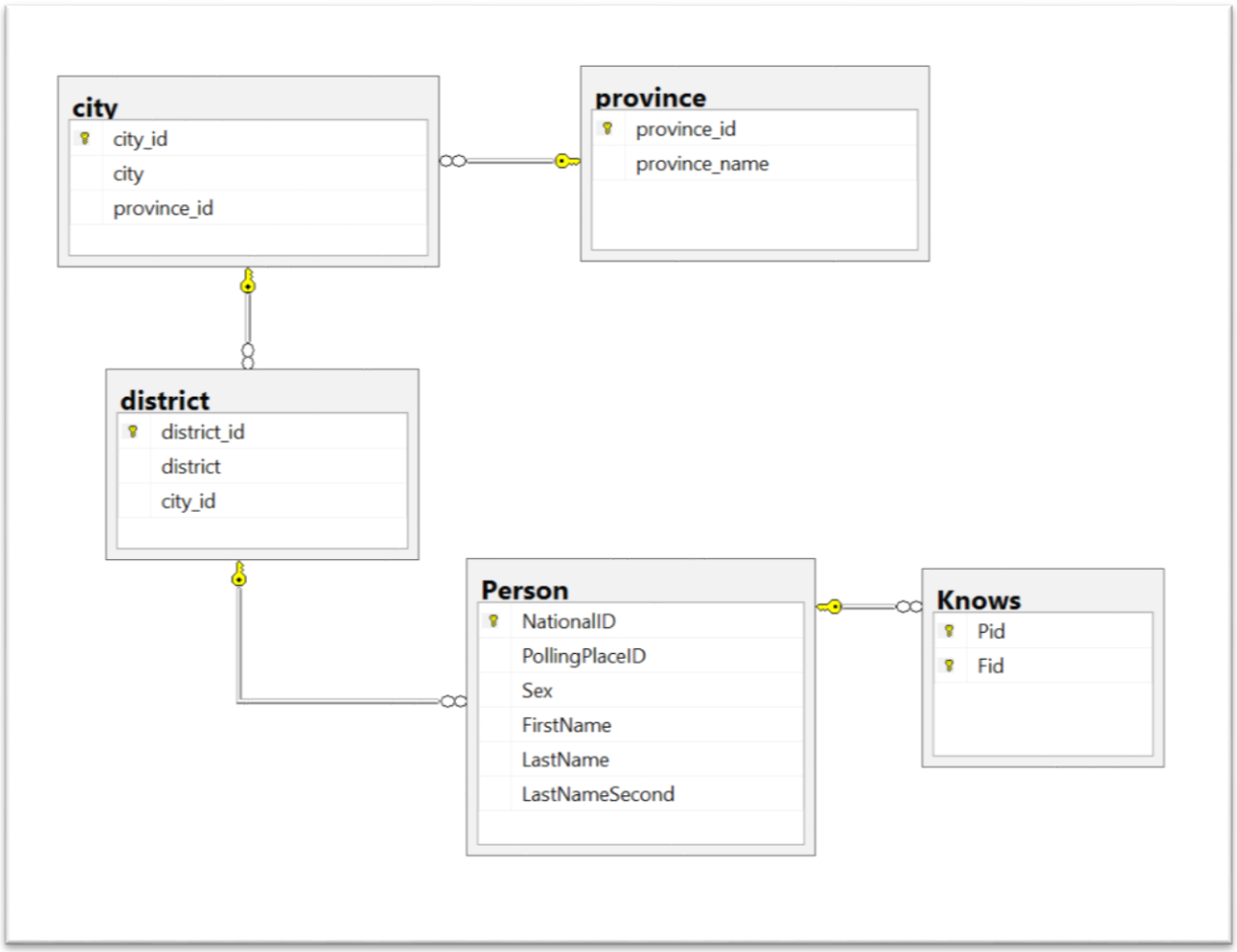
The next picture 5.2.1 is the Conceptual Model of our Database Application



Picture 5.2.1 Conceptual Model

5.3) Relational Model

The next picture 5.2.2 is the Relational Model of our Database Application

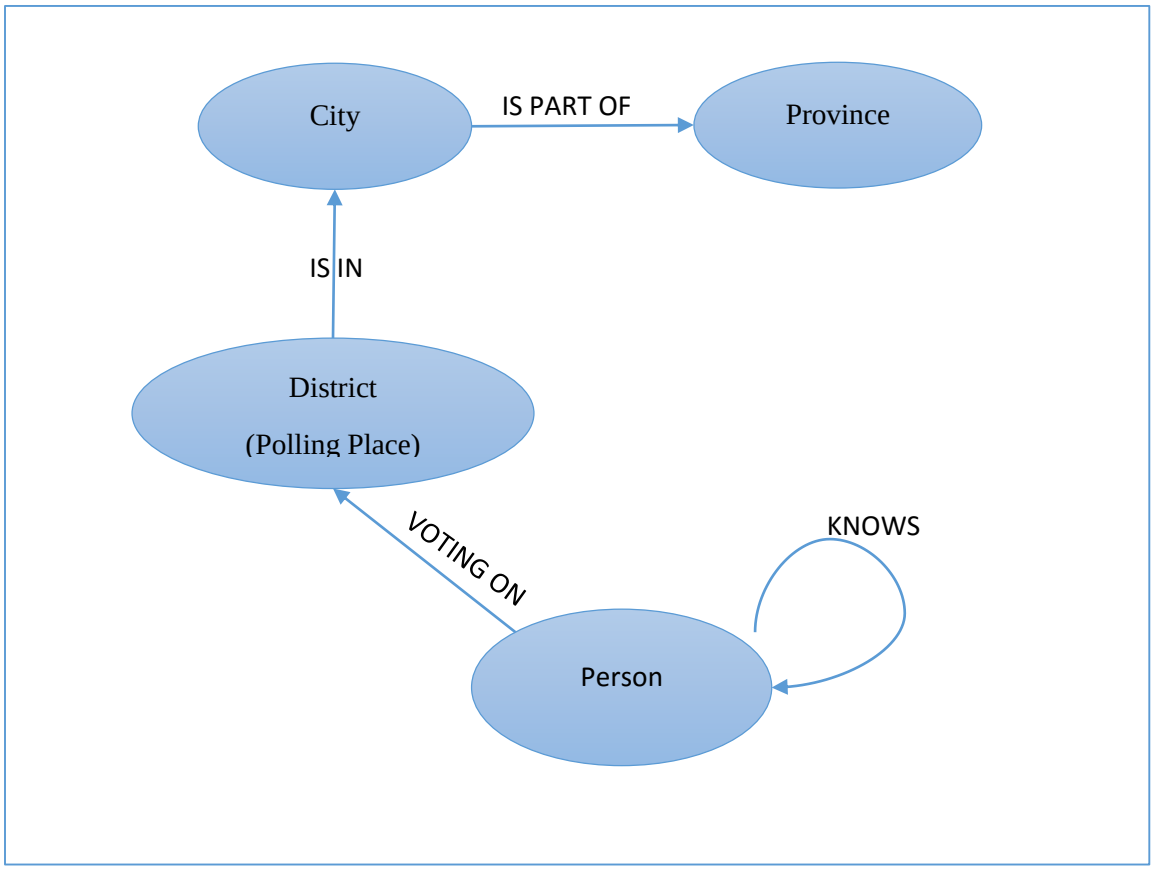


Picture 5.2.2 Relational Model

5.3) Graph Model

The next picture 5.2.3 is a graph which represents the nodes and how these are connected I our graph database in neo4j

.



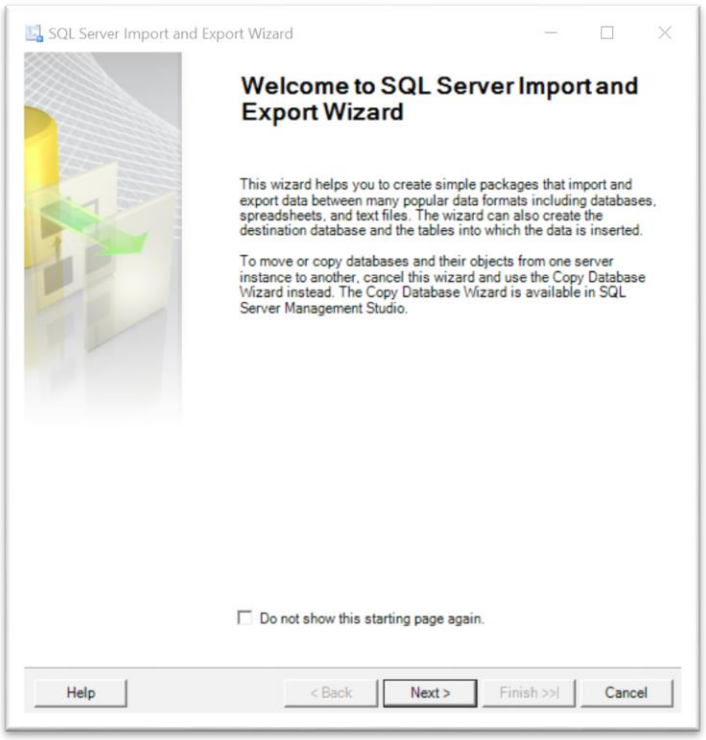
Picture 5.2.3 Graph Model

5.4) Populating of the Databases

For loading data in the database, we use some tools depending the technology. Also previously we changed the headers of the files with names more readable.

Loading data in SQL Server 2016

For loading data in SQL server, we used the tool of the SQL Server "Import and Export Data" which is included in the SQL Server 2016.



Picture 5.4.1 Tool for Import in SQL Server

Loading data in NEO4J

For loading data in NEO4J, we used the tool showed in the chapter 4.2, in the sub charter: ***Importing CSV files with Cypher.***

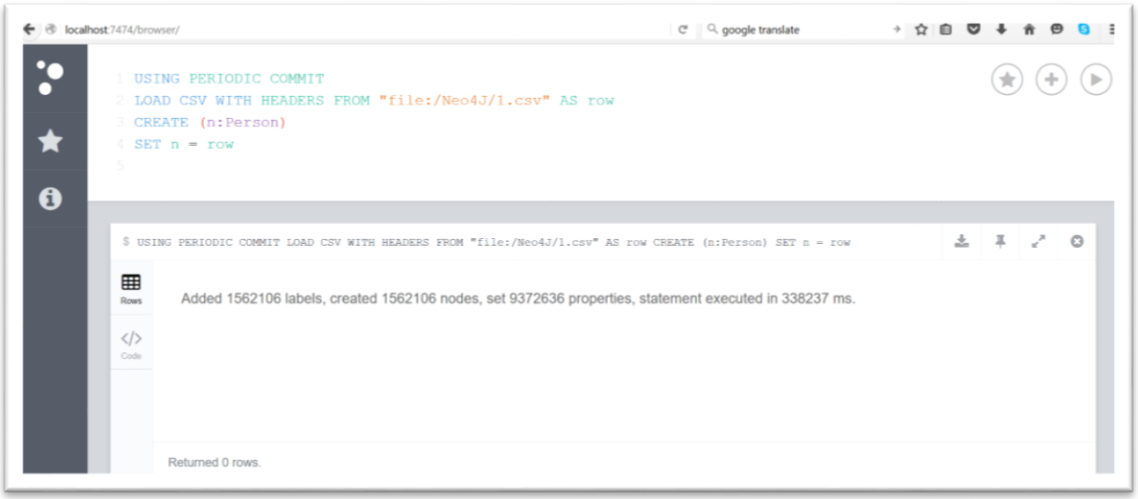
Example code to upload a csv with headers with data of People:

```
USING PERIODIC COMMIT
LOAD CSV WITH HEADERS FROM "file:/Neo4J/ELECTORAL_ROLL.csv" AS row
CREATE (n:Person)
SET n = row
```

IMPORTANT, for an optimized work we considered these important points.

- For importing we used always "**USING PERIODIC COMMIT** ".
- We split files greater than 3 million of rows or more, in files of 1.5 million maximum in one load.
- Always create index with the nodes and properties more usables. Example the index for person for search by National ID is:

  `CREATE INDEX ON :People(NationalID)`



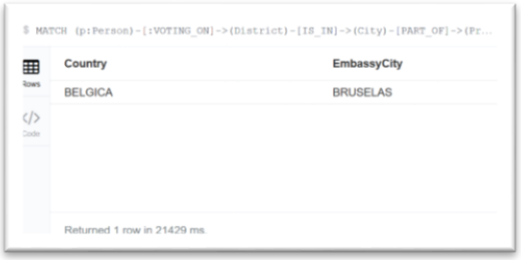
Picture 5.4.2 Tool for Import in Neo4J

5.5) Comparative Queries (Neo4j vs SQL Server 2016)

We will run X number of queries to compare the performance. It is will be running in Cypher Query Language and Structure Query Language for comparing the expressivity of both technologies.

1) Search the country and the city where a Citizen of Costa Rica, living abroad could vote. The name of the citizen is "Esteban Zimanyi":

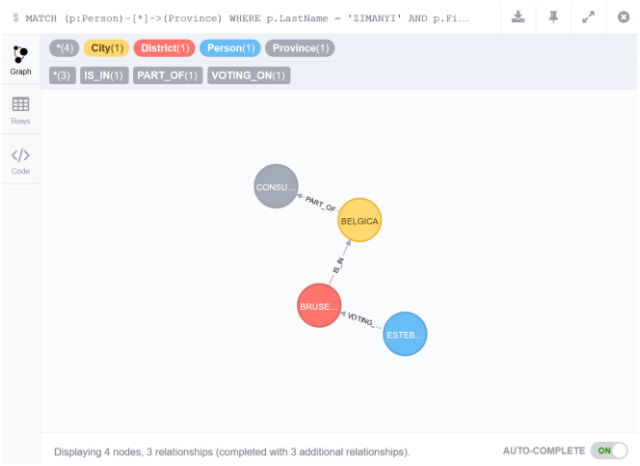
```
MATCH (p:Person)-[:VOTING_ON]->(District)-
[IS_IN]->(City)-[PART_OF]->(Province)
WHERE p.LastName = 'ZIMANYI' AND p.FirstName
= 'ESTEBAN'
RETURN City.city_name as Country ,
District.district_name as EmbassyCity
```




```
SELECT c.city as Country, d.district as
EmbassyCity
FROM  District d, Person p , City c, Province pr
WHERE d.district_id = p.PollingPlaceId and
                d.city_id = c.city_id and
c.province_id = pr.province_id
                and p.LastName = 'ZIMANYI' AND
P.FirstName = 'ESTEBAN'
```

1.1 In the same way of this query, showing graphically the connections of the graph

```
MATCH (p:Person)-[*]->(Province)
WHERE p.LastName = 'ZIMANYI' AND p.FirstName =
'ESTEBAN'
RETURN *
```

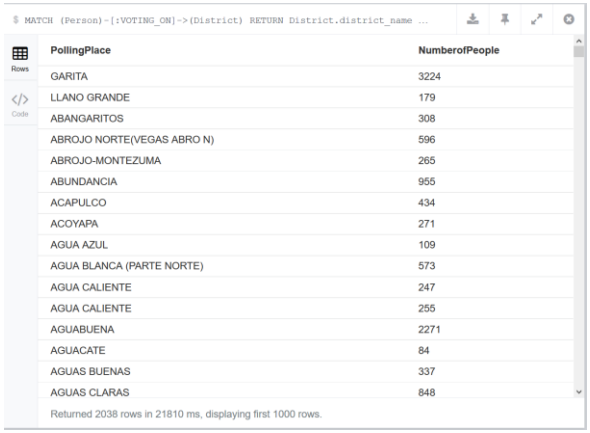


It is Not Possible to do it in Sql Server

2) Count the number of citizens (People) who voting in each district (Polling Place):

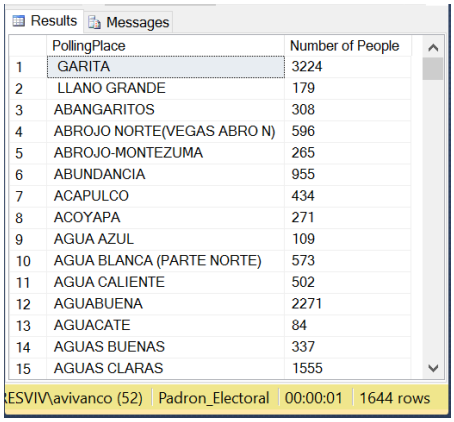
```
MATCH (Person)-[:VOTING_ON]->(District)
RETURN District.district_name as
PollingPlace, count(*) as NumberofPeople
order by District.district_name
```

```
SELECT d.district as 'PollingPlace', count(*) as
'Number of People'
FROM  District d, Person p
WHERE d.district_id = p.PollingPlaceId
GROUP BY d.district order by PollingPlace
```



PollingPlace	NumberofPeople
GARITA	3224
LLANO GRANDE	179
ABANGARITOS	308
ABROJO NORTE(VEGAS ABRO N)	596
ABROJO-MONTEZUMA	265
ABUNDANCIA	955
ACAPULCO	434
ACOYAPA	271
AGUA AZUL	109
AGUA BLANCA (PARTE NORTE)	573
AGUA CALIENTE	247
AGUA CALIENTE	255
AGUABUENA	2271
AGUACATE	84
AGUAS BUENAS	337
AGUAS CLARAS	848

Returned 2038 rows in 21810 ms, displaying first 1000 rows.



Results | Messages

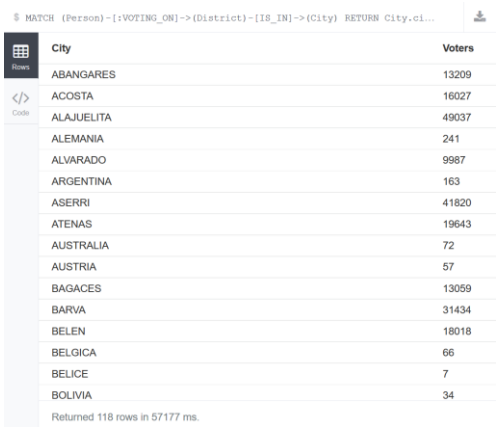
	PollingPlace	Number of People
1	GARITA	3224
2	LLANO GRANDE	179
3	ABANGARITOS	308
4	ABROJO NORTE(VEGAS ABRO N)	596
5	ABROJO-MONTEZUMA	265
6	ABUNDANCIA	955
7	ACAPULCO	434
8	ACOYAPA	271
9	AGUA AZUL	109
10	AGUA BLANCA (PARTE NORTE)	573
11	AGUA CALIENTE	502
12	AGUABUENA	2271
13	AGUACATE	84
14	AGUAS BUENAS	337
15	AGUAS CLARAS	1555

ESVIV\avivanco (52) | Padron_Electoral | 00:00:01 | 1644 rows

3) Count the number of citizens (People) who voting in each City:

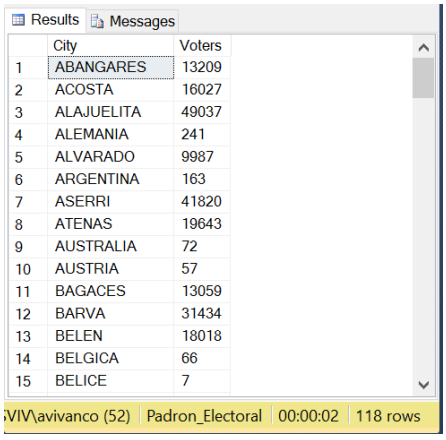
```
MATCH (Person)-[:VOTING_ON]->(District)-
[IS_IN]->(City)
RETURN City.city_name as City, count(*) as
Voters order by City.city_name
```

```
SELECT c.city as 'City', count(*) as 'Voters'
FROM  District d, Person p , City c
WHERE d.district_id = p.PollingPlaceId and
                d.city_id = c.city_id
GROUP BY c.city order by City
```



City	Voters
ABANGARES	13209
ACOSTA	16027
ALAJUELITA	49037
ALEMANIA	241
ALVARADO	9987
ARGENTINA	163
ASERRI	41820
ATENAS	19643
AUSTRALIA	72
AUSTRIA	57
BAGACES	13059
BARVA	31434
BELEN	18018
BELGICA	66
BELICE	7
BOLIVIA	34

Returned 118 rows in 57177 ms.



Results | Messages

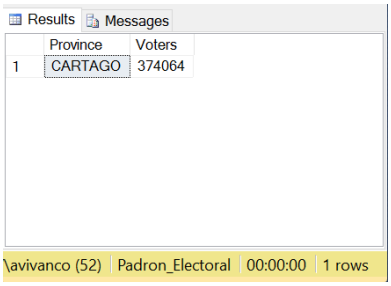
	City	Voters
1	ABANGARES	13209
2	ACOSTA	16027
3	ALAJUELITA	49037
4	ALEMANIA	241
5	ALVARADO	9987
6	ARGENTINA	163
7	ASERRI	41820
8	ATENAS	19643
9	AUSTRALIA	72
10	AUSTRIA	57
11	BAGACES	13059
12	BARVA	31434
13	BELEN	18018
14	BELGICA	66
15	BELICE	7

VIV\avivanco (52) | Padron_Electoral | 00:00:02 | 118 rows

4) Count the number of citizens (People) who voting in the province of "Cartago"

```
MATCH (Person)-[:VOTING_ON]->(District)-
[IS_IN]->(City)-[PART_OF]-
>(Province{province_name: 'CARTAGO'})
RETURN Province.province_name as Province,
count(*) as Voters order by
Province.province_name
```

```
SELECT pr.province_name as 'Province', count(*) as
'Voters'
FROM  District d, Person p , City c, Province pr
WHERE d.district_id = p.PollingPlaceId and
                d.city_id = c.city_id and
c.province_id = pr.province_id
                and pr.province_name = 'CARTAGO'
GROUP BY pr.province_name order by
pr.province_name
```



5) Considering that in the data, exist information about people who living abroad but they are voting in the embassies of Costa Rica over the world, count the number of citizens (People) who voting in each Country, in this case the province is with the value of "CONSULADO" and order the results in descendent order.

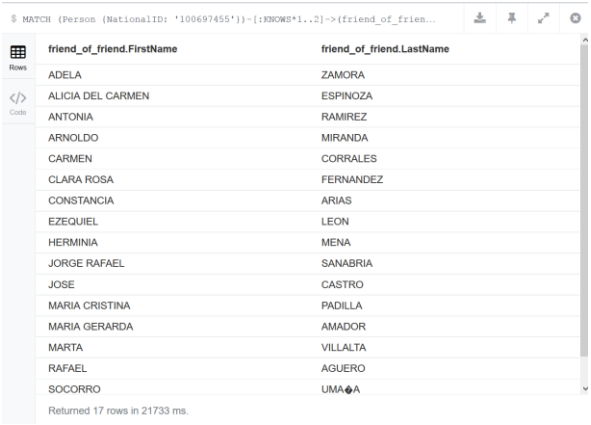
```
MATCH (Person)-[:VOTING_ON]->(District)-
[IS_IN]->(City)-[PART_OF]-
>(Province{province_name: 'CONSULADO'})
RETURN City.city_name as Country, count(*) as
Voters order by Voters desc
```

```
SELECT  c.city as 'Country' ,count(*) as 'Voters'
FROM  District d, Person p , City c, Province pr
WHERE d.district_id = p.PollingPlaceId and
                d.city_id = c.city_id and
c.province_id = pr.province_id
                and pr.province_name = 'CONSULADO'
GROUP BY  c.city    order by  Voters desc;
```

6) Search friends of friends of the Person with National ID 100697455:

neo4j

```
MATCH (Person {NationalID: '100697455'})-
[:KNOWS*1..2]->(friend_of_friend)
RETURN DISTINCT  friend_of_friend.FirstName,
friend_of_friend.LastName
ORDER BY friend_of_friend.FirstName ,
friend_of_friend.LastName
```

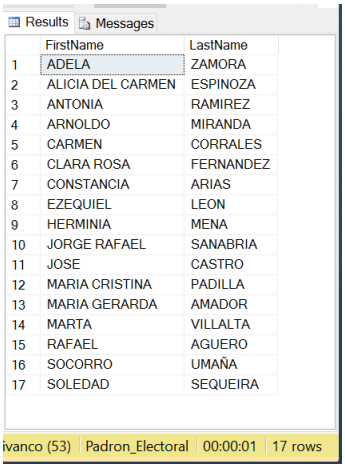


friend_of_friend.FirstName	friend_of_friend.LastName
ADELA	ZAMORA
ALICIA DEL CARMEN	ESPINOZA
ANTONIA	RAMIREZ
ARNOLDO	MIRANDA
CARMEN	CORRALES
CLARA ROSA	FERNANDEZ
CONSTANCIA	ARIAS
EZEQUIEL	LEON
HERMINIA	MENA
JORGE RAFAEL	SANABRIA
JOSE	CASTRO
MARIA CRISTINA	PADILLA
MARIA GERARDA	AMADOR
MARTA	VILLALTA
RAFAEL	AGUERO
SOCORRO	UMA◆A

Returned 17 rows in 21733 ms.

Microsoft SQL Server 2016

```
select distinct P.FirstName, P.LastName
from Knows , Person P
where Fid = P.NationalID  and Pid IN (
Select distinct Fid from Knows
where Pid = 100697455 ) --order by P.FirstName,
P.LastName)
Union select distinct P.FirstName, P.LastName from
Knows , Person P where Fid = P.NationalID  and Fid
IN ( Select distinct Fid from Knows
where Pid = 100697455 ) order by P.FirstName,
P.LastName
```



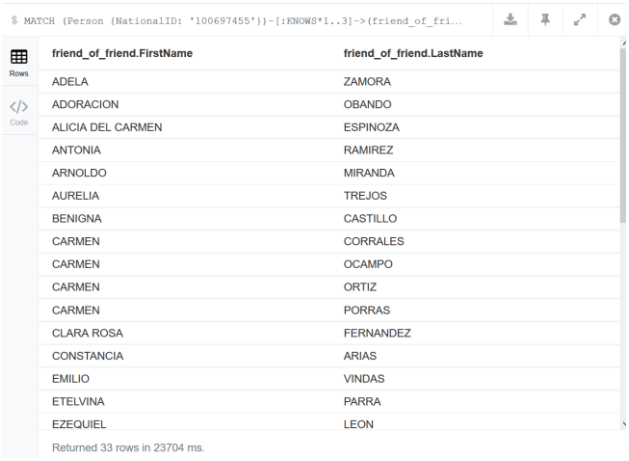
Results | Messages

	FirstName	LastName
1	ADELA	ZAMORA
2	ALICIA DEL CARMEN	ESPINOZA
3	ANTONIA	RAMIREZ
4	ARNOLDO	MIRANDA
5	CARMEN	CORRALES
6	CLARA ROSA	FERNANDEZ
7	CONSTANCIA	ARIAS
8	EZEQUIEL	LEON
9	HERMINIA	MENA
10	JORGE RAFAEL	SANABRIA
11	JOSE	CASTRO
12	MARIA CRISTINA	PADILLA
13	MARIA GERARDA	AMADOR
14	MARTA	VILLALTA
15	RAFAEL	AGUERO
16	SOCORRO	UMAÑA
17	SOLEDAD	SEQUEIRA

ivanco (53) | Padron_Electoral | 00:00:01 | 17 rows

7) Search friends of friends of friends (3rd Grade of Depth) of the Person with National ID 100697455:

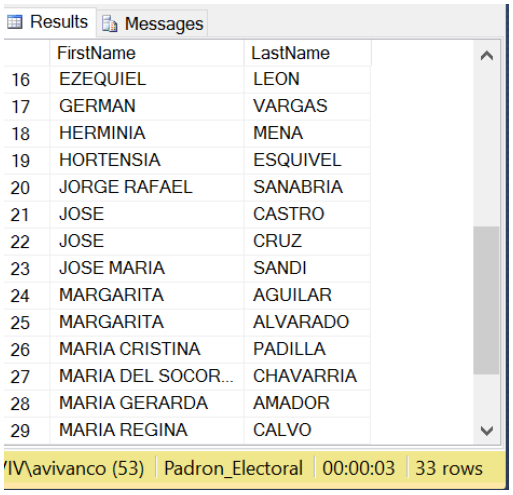
```
MATCH (Person {NationalID: '100697455'})-
[:KNOWS*1..3]->(friend_of_friend)
RETURN DISTINCT  friend_of_friend.FirstName,
friend_of_friend.LastName
ORDER BY friend_of_friend.FirstName ,
friend_of_friend.LastName
```



friend_of_friend.FirstName	friend_of_friend.LastName
ADELA	ZAMORA
ADORACION	OBANDO
ALICIA DEL CARMEN	ESPINOZA
ANTONIA	RAMIREZ
ARNOLDO	MIRANDA
AURELIA	TREJOS
BENIGNA	CASTILLO
CARMEN	CORRALES
CARMEN	OCAMPO
CARMEN	ORTIZ
CARMEN	PORRAS
CLARA ROSA	FERNANDEZ
CONSTANCIA	ARIAS
EMILIO	VINDAS
ETELVINA	PARRA
EZEQUIEL	LEON

Returned 33 rows in 23704 ms.

```
select distinct P.FirstName, P.LastName
from Knows , Person P
where Fid = P.NationalID  and Pid IN (
                Select  Fid
                from Knows
where Pid IN ( Select  Fid
                from Knows
where Pid = 100697455 ) )
union
select distinct P.FirstName, P.LastName
from Knows , Person P
where Fid = P.NationalID  and Pid IN (
                Select  Fid
                from Knows
where Pid IN ( Select  Fid
                from Knows
where Fid = 100697455 ) )
union
select distinct P.FirstName, P.LastName
from Knows , Person P
where Fid = P.NationalID  and Pid IN (
                Select  Fid
                from Knows
where Fid IN ( Select  Fid
                from Knows
where Fid = 100697455 ) ) order by
P.FirstName, P.LastName
```



Results | Messages

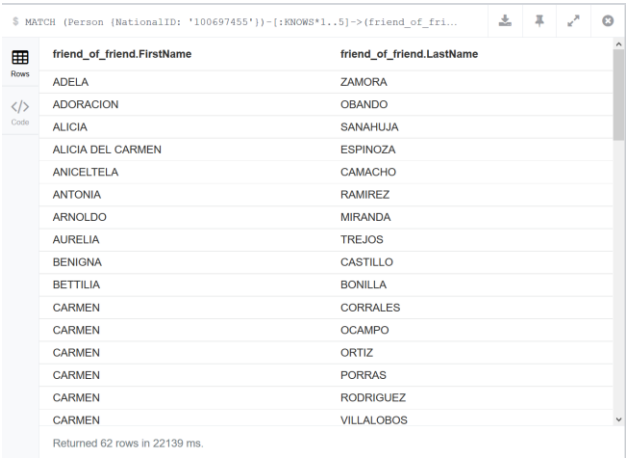
	FirstName	LastName
16	EZEQUIEL	LEON
17	GERMAN	VARGAS
18	HERMINIA	MENA
19	HORTENSIA	ESQUIVEL
20	JORGE RAFAEL	SANABRIA
21	JOSE	CASTRO
22	JOSE	CRUZ
23	JOSE MARIA	SANDI
24	MARGARITA	AGUILAR
25	MARGARITA	ALVARADO
26	MARIA CRISTINA	PADILLA
27	MARIA DEL SOCOR...	CHAVARRIA
28	MARIA GERARDA	AMADOR
29	MARIA REGINA	CALVO

IV\avivanco (53) | Padron_Electoral | 00:00:03 | 33 rows

8) Search friends of friends of friends (5th Grade of Depth) of the Person with National ID 100697455:

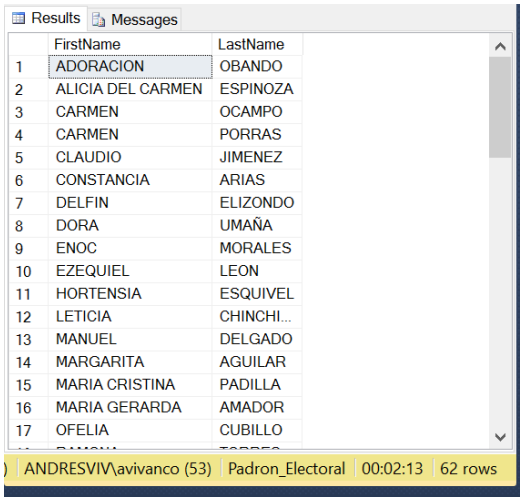
```
MATCH (Person {NationalID: '100697455'})-
[:KNOWS*1..5]->(friend_of_friend)
RETURN DISTINCT  friend_of_friend.FirstName,
friend_of_friend.LastName
ORDER BY friend_of_friend.FirstName ,
friend_of_friend.LastName
```

```
select distinct P.FirstName, P.LastName
from Knows , Person P
where Fid = P.NationalID  and Pid IN (
Select  Fid                 from Knows
where Pid IN ( Select  Fid  from Knows
where Pid IN (Select  Fid   from Knows
where Pid IN (Select  Fid   from Knows
where Pid = 100697455 ))))
UNION
select distinct P.FirstName, P.LastName
from Knows , Person P
where Fid = P.NationalID  and Pid IN (
Select  Fid                 from Knows
where Pid IN ( Select  Fid  from Knows
where Pid IN (Select  Fid   from Knows
where Pid IN (Select  Fid   from Knows
where Fid = 100697455 ))))
UNION
select distinct P.FirstName, P.LastName
from Knows , Person P
where Fid = P.NationalID  and Pid IN (
Select  Fid                 from Knows
where Pid IN ( Select  Fid  from Knows
where Pid IN (Select  Fid   from Knows
where Fid IN (Select  Fid   from Knows
where Fid = 100697455 ))))
UNION
select distinct P.FirstName, P.LastName
from Knows , Person P
where Fid = P.NationalID  and Pid IN (
Select  Fid                 from Knows
where Pid IN ( Select  Fid  from Knows
where Fid IN (Select  Fid   from Knows
where Fid IN (Select  Fid   from Knows
where Fid = 100697455 ))))
UNION
select distinct P.FirstName, P.LastName
from Knows , Person P
where Fid = P.NationalID  and Pid IN (
Select  Fid                 from Knows
where fid IN ( Select  Fid  from Knows
where Fid IN (Select  Fid   from Knows
where Fid IN (Select  Fid   from Knows
where Fid = 100697455 ))))
```



friend_of_friend.FirstName	friend_of_friend.LastName
ADELA	ZAMORA
ADORACION	OBANDO
ALICIA	SANAHUJA
ALICIA DEL CARMEN	ESPINOZA
ANICELTELA	CAMACHO
ANTONIA	RAMIREZ
ARNOLDO	MIRANDA
AURELIA	TREJOS
BENIGNA	CASTILLO
BETTILIA	BONILLA
CARMEN	CORRALES
CARMEN	OCAMPO
CARMEN	ORTIZ
CARMEN	PORRAS
CARMEN	RODRIGUEZ
CARMEN	VILLALOBOS

Returned 62 rows in 22139 ms.



Results | Messages

	FirstName	LastName
1	ADORACION	OBANDO
2	ALICIA DEL CARMEN	ESPINOZA
3	CARMEN	OCAMPO
4	CARMEN	PORRAS
5	CLAUDIO	JIMENEZ
6	CONSTANCIA	ARIAS
7	DELFIN	ELIZONDO
8	DORA	UMAÑA
9	ENOC	MORALES
10	EZEQUIEL	LEON
11	HORTENSIA	ESQUIVEL
12	LETICIA	CHINCHI...
13	MANUEL	DELGADO
14	MARGARITA	AGUILAR
15	MARIA CRISTINA	PADILLA
16	MARIA GERARDA	AMADOR
17	OFELIA	CUBILLO

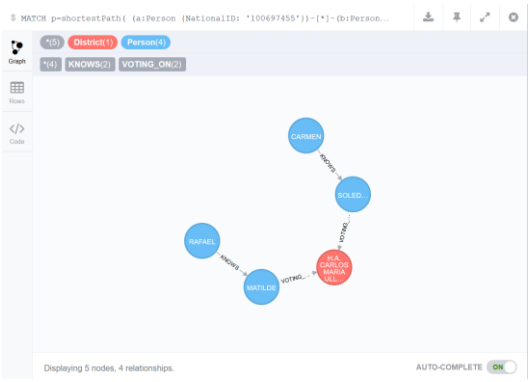
ANDRESVIV\avivanco (53) | Padron_Electoral | 00:02:13 | 62 rows

9) Let's suppose, that we want to connect or introduce two people between them, we don't know the name of the relationships or nothing about how they can be connected, the unique data that we have is just the two National Ids. We want to know how can connect them in a short way possible.

Search the Shortest Path between Two People with NationalIDs "100697455 "and "101018697"

```
MATCH p=shortestPath(
  (a:Person {NationalID: '100697455'})-[*]-
(b:Person {NationalID: '101018697'})
)
RETURN p
```

It is Not Possible to do it in Sql Server because for create a query, is necessary to know previously which relationships are in the middle, but it could violate the constraint that we stablished for this query.

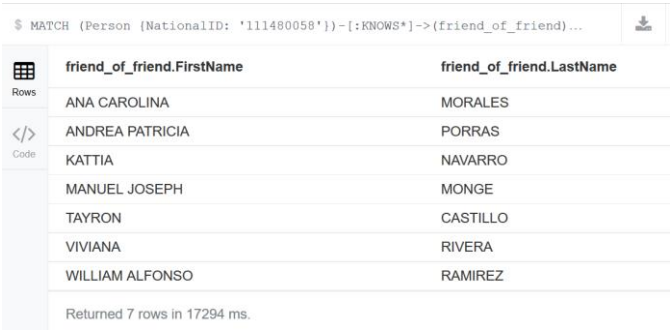


10) We want to know all the circle of friends of friends. given a NationalID of a person.

Find all friends of friends possible for one person, in another words, search friend of friends of friends... until the last one that exist, to do it for the Person with the NationalID "111480058"

```
MATCH (Person {NationalID: '111480058'})-
[:KNOWS*]->(friend_of_friend)
RETURN DISTINCT  friend_of_friend.FirstName,
friend_of_friend.LastName
ORDER BY friend_of_friend.FirstName ,
friend_of_friend.LastName
```

It is Not Possible to do it in Sql Server because for Sql always is needed to stablish the grade of depth of friendship



```
$ MATCH (Person {NationalID: '111480058'})-[:KNOWS*]->(friend_of_friend)...
```

Rows | Code

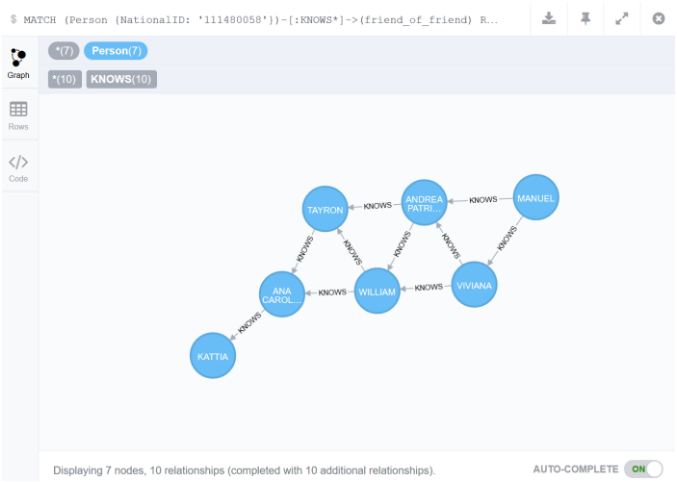
friend_of_friend.FirstName	friend_of_friend.LastName
ANA CAROLINA	MORALES
ANDREA PATRICIA	PORRAS
KATTIA	NAVARRO
MANUEL JOSEPH	MONGE
TAYRON	CASTILLO
VIVIANA	RIVERA
WILLIAM ALFONSO	RAMIREZ

Returned 7 rows in 17294 ms.

10.1In the same way, we want to watch graphically the circle of friends of the query 9.

```
MATCH (Person {NationalID: '111480058'})-
[:KNOWS*]->(friend_of_friend)
RETURN DISTINCT  friend_of_friend
```

It is Not Possible to do it in Sql Server



5.6) Analysis of Results (Neo4j vs SQL Server 2016)

We worked in a Laptop (Microsoft Surface Pro 3) with the next features:

- Processor: Intel Pentium 5
- RAM: 8GB

We started from cero, in both databases, creating it, loading the data, creating the index in Neo4jand primary and foreign keys in SQL Server.

We created different types of queries for evaluate time execution and the expressivity.

Our conclusion per groups of Query is the next:

- **From Query 1 to Query 5:** We can notice that these queries are simples query without not too much data to search, and with information in 4 different tables in SQL server, and in neo4j with 4 different types of nodes.
  In this case SQL server had the best performance with the best time execution, for each query, about the expressivity we can conclude that it could be similar, in both cases for specific information we need to write each relationship or how the nodes are connected to search the result.
  But, for showing graphically the connections of one person like in the query 1.1, neo4j has an incredible functionality, and it is very friendly for the users, also the expressivity of the query is significant, with less lines of codes.
  Just we need to write the node to find, addressed to with node we want to know the information, like for Example:

```
MATCH (p:Person)-[*]->(Province)
WHERE p.LastName = 'ZIMANYI' AND p.FirstName = 'ESTEBAN'
RETURN *
```

  In this case only knowing the input data (Name of the citizen) and indicate the last node that we want to know, in this case Province, we can obtain nodes and relationships between both. Like this:

This important feature is named Traversal, and this is unique of graph model

- **From Query 6 to Query 8:** We did queries about friends of friends, in different grades of depth, in SQL server we used the table “knows” meanwhile in Neo4j the relationship “knows”.
  We can notice for this type of queries SQL was superior in time execution until depth 3th, in another words “friends of friends of friends”. But our surprised was when we run friends of friend with depth 5(Query #8), SQL server did it in more than 2 minutes, whereas in Neo4j was less of one minute. The theory of the differences of these types of queries was proven. Another significant point and not less important is the expressivity, practically in neo4j for this type of query is the same query just with different parameters, but do this types of queries in SQL server, to major grade of depth, is major the number of code lines.

- **Query 9:** For this type of queries, about the shortest path, the unique solution was in Neo4J, neo4j like is a graph, using the mathematical algorithm for do it, it could be very helpful for example to know the shortest way between two places. For doing it in SQL server is necessary to know the relationships and the result could be a complex query, but we supposed that we don’t need to know information about relationships or how the tables are connected.

- **Query 10:** For this type of queries, the unique solution was in Neo4J, because in a graph database we don’t need to indicate how to go to the information, just we need to write what patrons or nodes we want to find. And the graph database does it by itself. It is a powerful tool that could help in many circumstances to explore information or inclusive in datamining, for example, to see connection between people to avoid money laundering.

6) Conclusion

The next are some important conclusions, to summarize whole work in the passionate world of graph databases that we researched.

- Leonard Euler resolved “The Königsberg Bridge Problem”, but also create a math basis to solve it
- A Graph has Entities (nodes) and relationships (edges)
- The nodes, and relationships could possess properties
- The relationships have a named and direction to connect nodes.
- A graph could be modeled with almost any technology, i.e. relational, but the main differences is the performance, for example execution time
- Linear cost to retrieve adjacent nodes: depends on the number of local neighbors
- Graphs are whiteboard friendly in comparative with a RDBMS
- Doing join queries in a graph database is more efficient and more expressive than a relational database.
- Traversal, is the operation of visiting a set of nodes in a graph, going between nodes connected with the relationships, this operation is unique of a graph mode.
- This is very powerful when the user wants to explore a set of data, because given double click in the node, the user could watch more nodes related to it, and discovering information, a little like datamining
- Shortest path, is a function very helpful, for example to find the shortest path between two places, or search the shortest path to introduce one person to another one like in the social network LinkedIn.

Although, In the storing computing world, Graphs databases seems like the next step of the relational databases, we can notice some important points to consider it.

- Is not necessary leave to work with Relational DBMS, and just focus to work with a graph database or just neo4j, but perhaps, you can combine to work the graph database with your traditional database, and use neo4j specially to find relationships in your database, because the engine of the graph database is focusing in optimize search or relationships. See a graph database like a search engine for relations
- Due to the advantages of graph database we recommend to use it for:
    - Fraud Detection, Money laundering
    - Social Network
    - Managing of relationship with good performance
    - Exploring Data that you don’t know, just doing “Double click” you can discover many things
    - Recommendations Systems (I.e. Amazon recommends to buy something, Netflix)
    - Route Planning Systems