Université libre de Bruxelles

INFO-H-415

Advanced Databases

Graph Databases and Neo4J

Authors:
Anna Turu Pi
Ozge Koroglu

Supervisor:
Esteban Zimányi

December 18, 2017

Contents

List of Tables

List of Figures

1 Introduction

The aim of this project is to compare graph databases to the main DBMSs to pinpoint the use cases it is more suitable for. In order to prove their effectiveness, a database using the same data set has been implemented both in Neo4j, the leading software using graph database technology, and SQL Server, the top-3 DBMS according to DB-Engines.

2 Background

The aim of this project is to prove that graph databases, more specifically Neo4j, was the most performant DBMS for some specific use cases, hence they earned their place in the DBMS's market. The first milestone was to investigate the state of the art of DBMS. Its purpose was to justify the existence of graph databases, showing that it meets some needs not covered by other DBMS.

2.1 State of the art of Databases

Database Systems evolution: Databases and database technology are vital to modern organizations supporting both the daily operations and decision making. Database technology has undergone remarkable evolution over 50 years. Despite dominance to the enterprise DBMS marketplace by Oracle, the industry remains highly competitive with a continued high level of innovation [12].

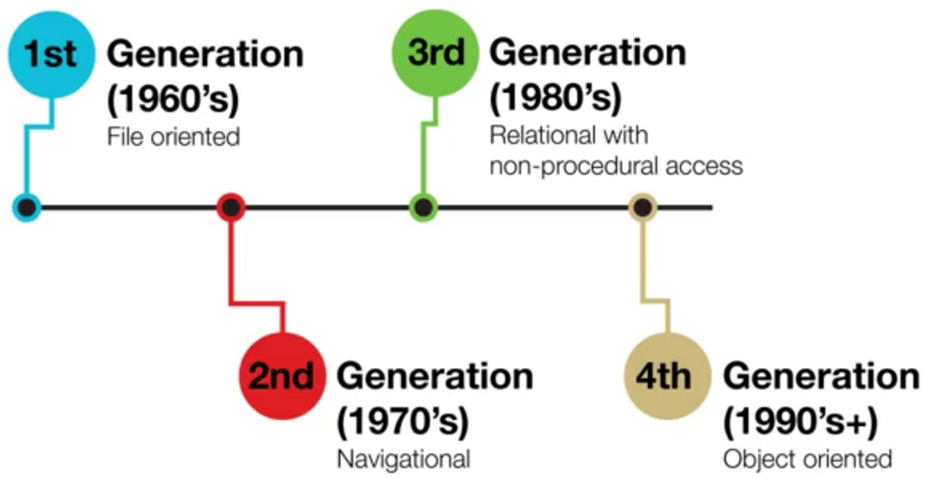


Figure 1: Evolution of database technology

Major periods of database technology evolution [12]:

- **1st Generation (1960's):** File oriented – Supported sequential and random searching of files, but the user was required to write computer programs to access data. The database software industry had little or no standards during this period.

- **2nd Generation (1970's):** Navigational – Could manage multiple entity types and relationships. Computer program still has to be written. Progress on standards.
- **3rd Generation (1980's):** Relational with non-procedural access – Foundation based on mathematical relations and associated operators. Optimization technology was developed. IBM performed pioneering research to enable commercialization of relational database technology.
- **4th Generation (1990's+):** Object oriented – Are extending the boundaries of database technology. New kinds of distributed processing and data warehouse processing. Can store and manipulate unconventional data types. Convenient ways to publish static and dynamic Web data.

DBMS marketplace: Despite dominance to the enterprise DBMS marketplace by Oracle, with more than 40% overall market share, the industry remains highly competitive with a continued high level of innovation. In some environments, its competition is Microsoft SQL Server, IBM DB2, Teradata, SAP Sybase. Open source DBMS products have begun to challenge the commercial DBMS products at the low-end of the enterprise DBMS marketplace. The category of open-source DBMS is leaded by MySQL, followed by MongoDB, PostgreSQL and MariaDB. In the desktop DBMS market, Microsoft Access dominates because of the dominance of Microsoft Office. [12]

Figure 2: DBMS marketplace

Innovation in the industry: The advances in DBMS in recent years support business intelligence processing for data integration and usage of summary data. **NoSQL technology** has been developed to support the needs of Big Data, to be modern web-scale databases. Since 2009, the most accepted definition of NoSQL is next generation databases being *non-relational*, *distributed*, *open-source* and *horizontally scalable*. Other characteristics that usually apply are *schema-free*, *scalability*, *global availability*, *easy replication support*, *simple API*, *eventually consistent/BASE (not ACID)*, and large scale data. [5] [19]

2.2 Types of DBMS

Ranking In this section we observe rankings created by DB-Engines. DB-Engines is an initiative that provides information on the popularity of the DBMS available in the market. They make available different rankings for every DBMS type, which are updated monthly. [3]

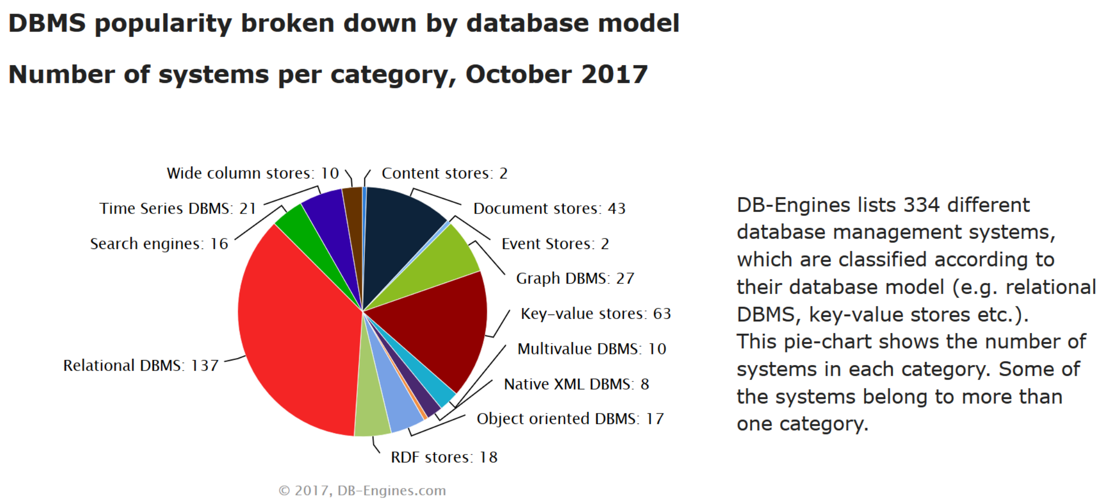


Figure 3: DBMS developed by database model pie chart

Over those lines, a pie chart represents the categories of DBMS that comprise more systems developed. The database model more elaborate is the Relational DBMS, where 137 systems fall under this category. It is followed by Key-value stores, with 63 systems, Document stores, with 43 systems, and Graph DBMS, with 27 systems.

In the overall classification of database models, those DBMS types are distinguished. Types of DBMS:

- Relational DBMS
- Key-value stores
- Document stores
- Graph DBMS
- Time Series DBMS
- RDF stores

- Object oriented DBMS (Atkinson)
- Search engines
- Multivalue DBMS
- Wide column stores
- Native XML DBMS
- Content stores
- Event Stores
- Navigational DBMS

Above these lines, the 14 more developed database models have been listed. If instead of counting the systems developed, the database models are ranked by popularity, the list of models to be considered shrinks. Most of the users work on relational DBMS, the 79.5%, followed by document stores, 7.3%, search engines, 4.3%, key-value stores, 3.5%, wide column stores, 3.1%, and graph DBMS, 1.1%. Below these lines a pie chart represents the most recent popularity rank. [3]

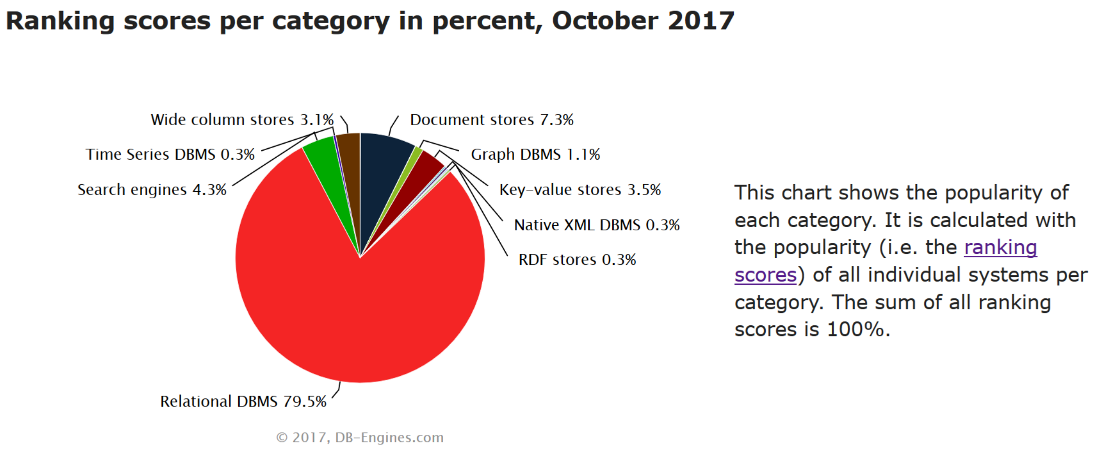


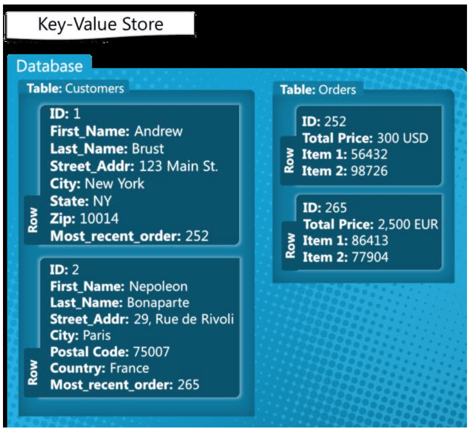
Figure 4: DBMS popularity by database model pie chart

In the pie chart above, it is clear to see that Relational DBMS are the ones used by default. However, the state of the art is changing by the innovations in the database technology. Even though the percentages of popularity of NoSQL databases are minimal compared to Relational DBMS, the fact that they are recent technologies in growth is enough to evaluate them more deeply.

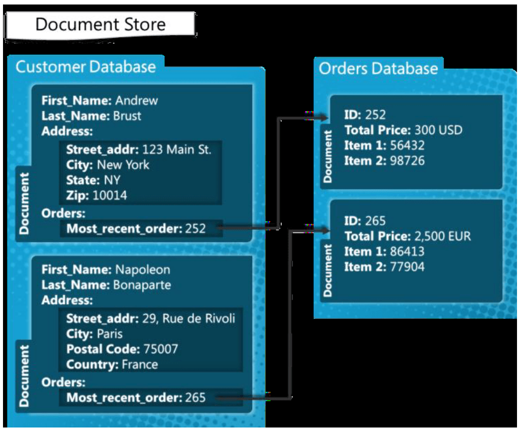
2.2.1 NoSQL DBMS

Many different NoSQL DBMS have been developed, but they are generally classified in four types [5]:

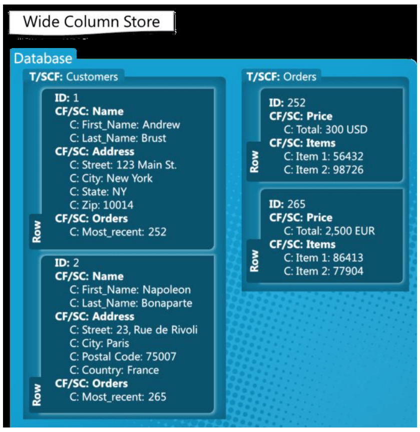
- **Key-value stores:** its structure consists in pairing keys to values. When performing a change in a value, the entire value other than the key must be updated. It scales well because of the simplicity. However, it can limit the complexity of the queries and other advanced features. [18] Examples: *Dynamo, Azure Table Storage, BerkeleyDB*
- **Document Stores:** The records stored are called documents, which consist of grouping of key-value pairs. Values can be nested to arbitrary depths. [18] Examples: *Elastic, MongoDB, Azure DocumentDB*
- **Wide Column Stores:** While RDBMS store all the data in a particular table's rows together on-disk, being able to retrieve a particular row fast, Column-family databases are able to retrieve a large amount of a specific attribute fast by serializing all the values of a particular column together on-disk. This approach is useful for aggregate queries. [18] Examples: *Hadoop/HBase, Cassandra, Amazon Simple DB*
- **Graph Databases:** ideal at dealing with interconnected data. Their structure consist of connections, or edges, between nodes. Both nodes and their edges can store additional properties such as key-value pairs. The strength of a graph database is in traversing the connections between the nodes. Their downside is that they generally require all data to fit on one machine, limiting their scalability. [18] Examples: *Neo4J, InfiniteGraph, TITAN*
- Other types: Multimodel Databases, Object Databases, Grid & Cloud Database Solutions, XML Databases, Multidimensional Databases, Multivalue Databases, Event Sources, Time Series / Streaming Databases



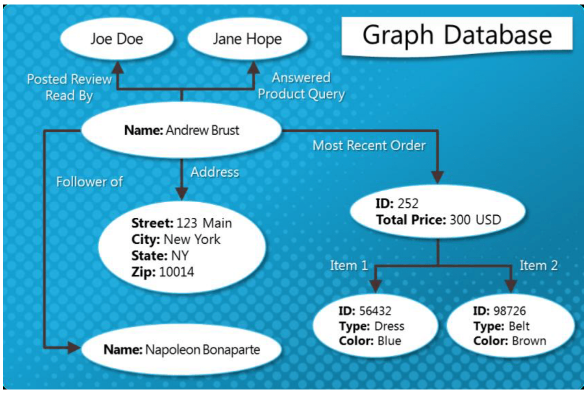
(a) Example of Key-Value Store



(b) Example of Document Store



(c) Example of Wide Column Store



(d) Example of Graph Database

Figure 5: Four main types of NoSQL databases

Consistency Models for NoSQL databases: Before NoSQL, ACID was the quintessential model that databases were meant to follow. Brief reminder of the ACID properties [16]:

- **Atomicity:** All operations in a transaction succeed or every operation is rolled back.
- **Consistent:** On the completion of a transaction, the database is structurally sound.
- **Isolated:** Transactions do not contend with one another. Contentious access to data is moderated by the database so that transactions appear to run sequentially.

- **Durable:** The results of applying a transaction are permanent, even in the presence of failures.

However, NoSQL databases break with the tipicality of SQL models with ACID properties. BASE properties seem to adecuate better to most NoSQL databases, and they are as follows [16]:

- **Basic Availability:** he database appears to work most of the time.
- **Soft-state:** Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
- **Eventual consistency:** Stores exhibit consistency at some later point (e.g., lazily at read time).

ACID transactions can be considered stricter than needed for many NoSQL cases, as they apply many constraints for safety sake. On the other hand, BASE transactions guarantees scale and resilience. The BASE model is used by aggregate stores, such as column family, key-value and document stores. In contrast, graph databases use the ACID model. BASE databases promise availability of the data at the expense of data consistency (the consistency of the data is only assured at concrete snapshots). [16] Graph databases differentiate themselves from other NoSQL databases by focusing more on data consistency. The comparison made in the lines above is shown in a table below:

	ACID	**BASE**
Properties	Atomicity Consistent Isolated Durable	Basic Availability Soft-state Eventual consistency
NoSQL DBMS	Graph Databases	Aggregate stores

Table 1: Comparison of ACID and BASE Consistency Models

2.2.2 Comparison of DBMS

Relational DBMS clearly are the benchmark among database systems. The mass adoption of this DBMS type is an important factor for choosing it as the main system in many companies. However, current trends show that the four main times of NoSQL databases should also be taken into account before installing a DBMS. To have a more objective point of view of the benefits of using each model, the use cases for which they perform better and the ones for which they perform the worst, are listed below.

Use cases for relational databases [17]

- Positive use cases: transaction-oriented databases (banking applications, online reservations), where the concurrency of many transactions must be supported and the integrity of the data must be protected.
- Negative use cases: data warehouses, which are analytically-oriented databases with a large amount of data and infrequent updates. The constraints of the relational database wouldn't support the scalability.

Use cases for key-value stores [19]

- Positive use cases:
  - For storing user session data
  - Maintaining schema-less user profiles
  - Storing user preferences
  - Storing shopping cart data
- Negative use cases:
  - To query the database by specific data value
  - With relationships between data values
  - To operate on multiple unique keys
  - If the business needs updating a part of the value frequently

Use cases for document stores [19]

- Positive use cases:
  - E-commerce platforms
  - Content management systems
  - Analytics platforms
  - Blogging platforms

- Negative use cases:
  - To run complex search queries
  - Application requires complex multiple operation transactions

Use cases for wide-column stores [19]

- Positive use cases:
  - Content management systems
  - Blogging platforms
  - Systems that maintain counters
  - Services that have expiring usage
  - Systems that require heavy write requests (like log aggregators)
- Negative use cases:
  - To use complex querying
  - If the query patterns change frequently
  - Without an established database requirement

Use cases for graph databases [19]

- Positive use cases:
  - Fraud detection
  - Graph based search
  - Network and IT operations
  - Social networks
- Negative use cases:
  - Data Warehouses so big that require BASE model

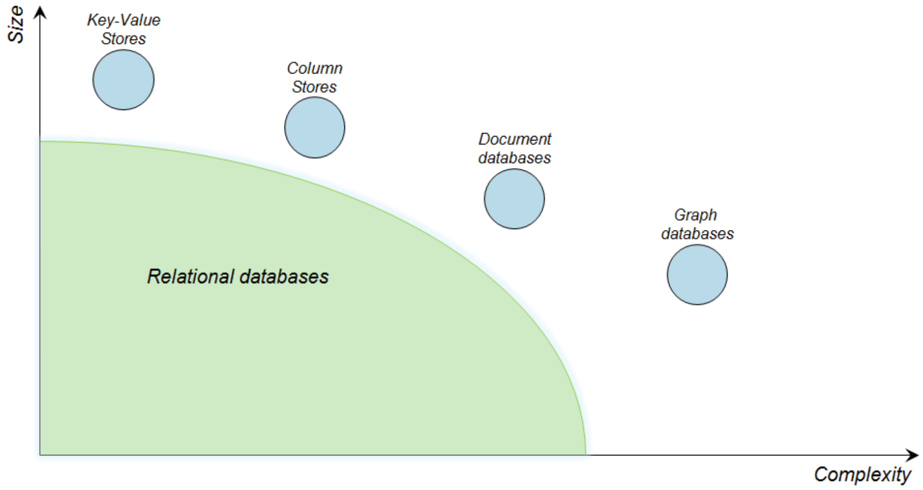


Figure 6: Positions of NoSQL databases (source: Neo4j)

On the figure above, the five types of DBMS that were being compared, are displayed according to the size and complexity of their databases. It can be concluded that each one of those DBMS works for some specific use cases, depending on the amount and complexity of the data that is going to be stored. Their use cases are not overlapped, which justifies that the fifth of them must be considered before implementing a DBMS in a company.

2.2.3 Current trends

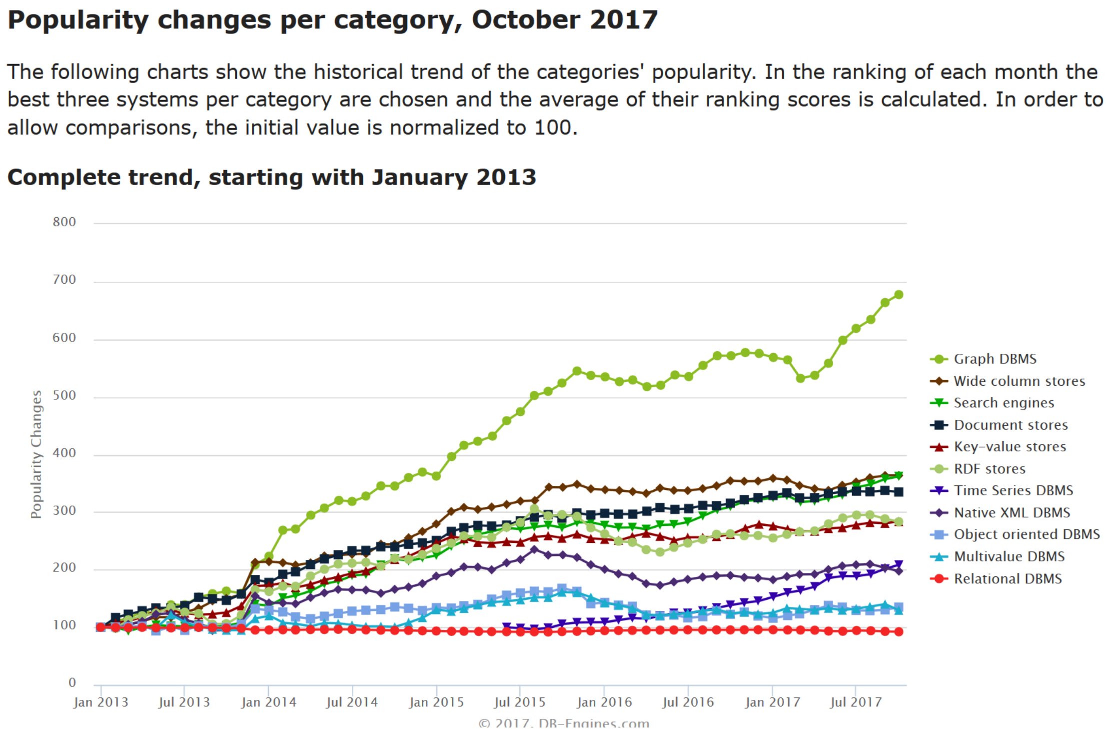


Figure 7: DBMS popularity by database model pie chart

In the previous pie-chart we concluded that the category of relational DBMS comprises most of the DBMS market. However, when looking at the chart of popularity changes per category [3], it is noticed that from 2014, **graph DBMS** differentiated themselves from the rest with a great popularity rise. This project aims to understand the causes of that booming trend.

2.3 Graph Databases

Graph databases are databases whose specific purpose is the storage of graph-oriented data structures [8], therefore an introduction to graph theory to be consistent when using its terminology.

2.3.1 Graph Theory and Its Applications

What is a graph A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges. Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. [14]

Properties [2] [7]

- multigraph: when any two vertices are joined by more than one edge.
- simple graph: a graph without loops and with at most one edge between any two vertices.
- complete graph: when each vertex is connected by an edge to every other vertex.
- directed graph, digraph: when a direction is assigned to each edge.
- The order of a graph is its number of vertices.
- The degree of a vertex in a graph is the number of edges which meet at that vertex.

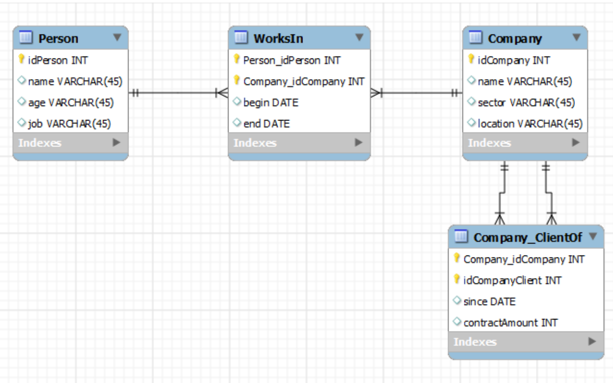
Graph theory applications [7]

- Road and Rail networks
- Integrated circuits
- Supply Chains
- Social networks
- Neural Connections

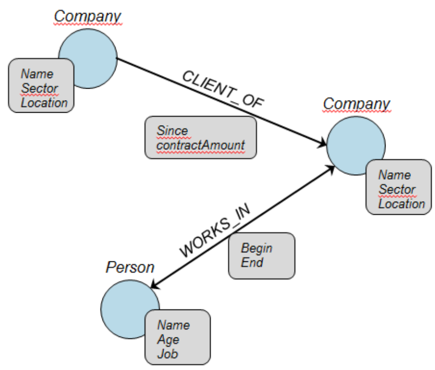
2.3.2 Concepts of Graph Databases

Positioning It has previously been explained that NoSQL databases address several issues that relational databases do not: availability for the processing of large datasets, partitioning, flexibility of the schema and modelling and processing complex structures like trees, graphs, or too many relationships. Graph databases are specialized in processing **highly connected data**, managing **complex and flexible data models** and improving the performance of complex queries by **traversing the graph**. [8]

Model Another quality of graph databases is the simplicity of its model. In the figures below, it can be appreciated the difference in modeling the same use case in a relational database or a graph database. The model of the graph database is more similar to the business model, which makes it more accessible to not-technical profiles. [8]



(a) Relational Database Model



(b) Graph Database Model

Figure 8: Model Comparison

2.3.3 Query performance

Graph databases competitive advantage It has been said that graph databases have a reason to be because they outperform relational databases in complex queries. They are particularly good when the relationships between items are significant. The use case that is better suited for graph databases is **"find all entities of a kind"** (*myEntity.findAll*). The execution of such a query, starts with an index lookup to find the starting node(s) for traversal. Then the relationships in the graph are traversed simultaneously. Because of the concurrence of the traversal, the bigger the volume of data, the more it outperforms relational databases. [8]

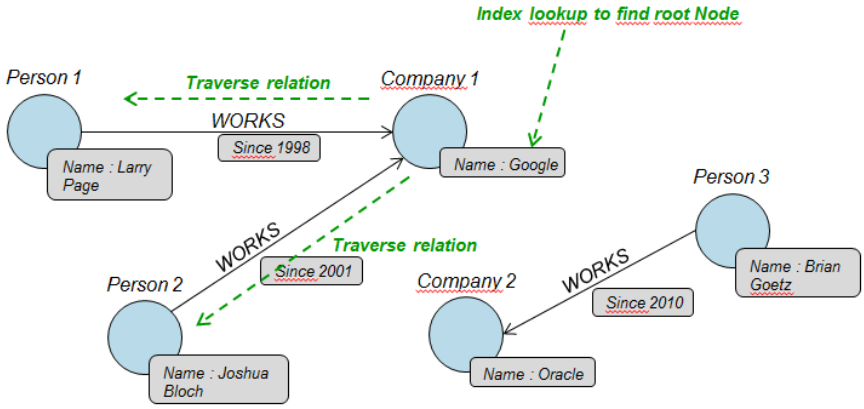


Figure 9: Query execution in graph databases

Relational databases are less adequate to query through relationships. It would mean querying through different tables, following foreign keys and other indexes, and it would considerably increment the performance time. Graph databases traversals are performed by following physical pointers, while foreign keys are logical pointers. [8] The query in the figure, includes the time of each index-scan. The more tables are included in the query, the larger the execution time will become.

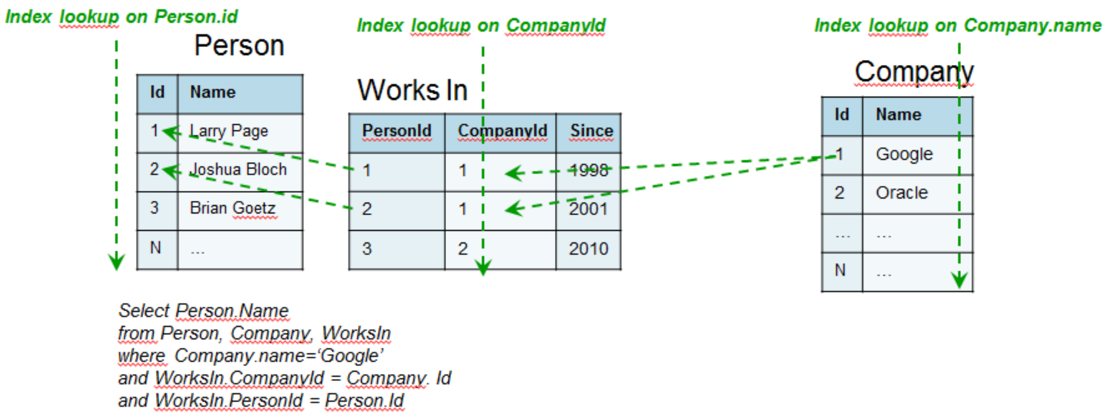


Figure 10: Query execution in relational databases

Relational Databases competitive advantage On the other hand, because of the internal structure of the tables, relational databases would outperform graph databases when the output requires **all the attributes of a table** (*findAll*-like queries). Its ideal use case is to aggregate over a complete dataset. [8]

Graph databases ranking Below those lines, the figure shows the DB-Engines Ranking on Graph DBMS. *Neo4j* leads the ranking, and its score triples the following DBMS, *Microsoft Azure Cosmos DB*. *Neo4j* has been leading the Graph databases sector for some years, as we can see in the trend scatter plot. It must be taken into account that the score is displayed in logarithmic scale, therefore the difference in popularity is really significant.

It can also be seen in the trend scatter plot that *Microsoft Azure Cosmos DB* appeared in the graph database landscape in 2014, and since then its rise in popularity has been quite steep. An argument for that is that Microsoft Azure is well integrated in the software marketplace.

Success factor: It has been stated, when comparing the NoSQL DBMS, that graph databases had a limitation in size. Therefore, it is a competitive advantage to work on facilitate the partitioning of a graph. While *OrientDB* and *InfiniteGraph* state that they accomplished so, *Neo4j* seems to be the DBMS that more successfully is improving graph partitioning. [8]

27 systems in ranking, October 2017

Rank			DBMS	Database Model	Score		
Oct 2017	Sep 2017	Oct 2016			Oct 2017	Sep 2017	Oct 2016
1.	1.	1.	Neo4j	Graph DBMS	37.95	-0.48	+1.50
2.	2.	↑ 4.	Microsoft Azure Cosmos DB	Multi-model	12.63	+1.40	+9.74
3.	3.	↓ 2.	OrientDB	Multi-model	6.13	+0.24	-0.12
4.	4.	↓ 3.	Titan	Graph DBMS	5.47	-0.02	+0.35
5.	5.	↑ 6.	ArangoDB	Multi-model	3.15	+0.15	+1.00
6.	6.	↓ 5.	Virtuoso	Multi-model	1.86	-0.03	-0.83
7.	7.	7.	Giraph	Graph DBMS	1.03	-0.03	+0.08
8.	↑ 9.	↑ 11.	GraphDB	Multi-model	0.64	+0.03	+0.43
9.	↓ 8.	↓ 8.	AllegroGraph	Multi-model	0.61	-0.02	+0.15
10.	10.	↓ 9.	Stardog	Multi-model	0.54	-0.04	+0.11

Figure 11: Graph DBMS Ranking

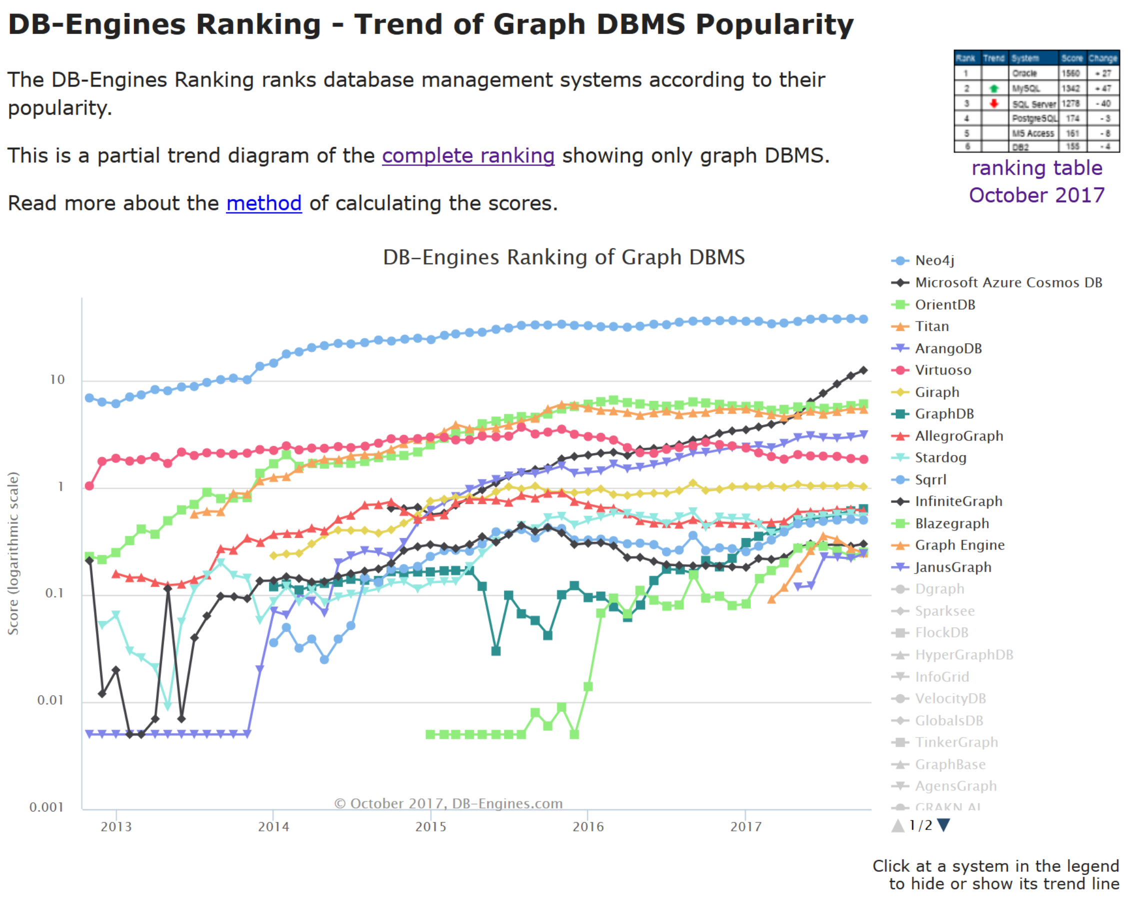


Figure 12: Trend Graph DBMS popularity scatter plot

3 Neo4j

3.1 Justification of Neo4j

Why Neo4j? By using a graph database like Neo4j which focuses on data relationships; patterns and trends can easily be seen unlike to relational databases. Due to today's growing business demands and competitive atmosphere, using the right tool is very important and when it comes to widely connected data Neo4j is the best because it is thousands of times faster than traditional databases. Neo4j analyze and traverse of all data in real time and gives the results very fast. Neo4j is widely used by lots of big companies like eBay,Walmart, Cisco, UBS and many more.

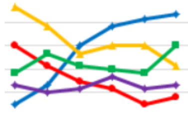
What is Neo4j? Neo4j is an open-source NoSQL graph database written in Java and Scala and According to db-engines.com, **Neo4j is currently world's leading graph database**. This has many reason. First of all Neo4j provides ACID transaction compliance, cluster support, runtime failover, high availability and high speed querying through traversals. It scales to billions of nodes and relationship. It has great user interface and it is easy to learn because there are lots of free online resources on the web. Also it has great community that can help with any problems. In general terms Neo4j is designed for linking relationships and it handles this relationships with speed, ease, and extreme flexibility. With Neo4j, models can easily be converted to database schema. If the data is densely connected or various conceptual model try's is needed for the data then Neo4j is the solution.

DB-Engines Ranking of Graph DBMS

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

This is a partial list of the complete ranking showing only graph DBMS.

Read more about the method of calculating the scores.



trend chart

27 systems in ranking, October 2017

Rank			DBMS	Database Model	Score		
Oct 2017	Sep 2017	Oct 2016			Oct 2017	Sep 2017	Oct 2016
1.	1.	1.	Neo4j	Graph DBMS	37.95	-0.48	+1.50
2.	2.	↑ 4.	Microsoft Azure Cosmos DB	Multi-model	12.63	+1.40	+9.74
3.	3.	↓ 2.	OrientDB	Multi-model	6.13	+0.24	-0.12
4.	4.	↓ 3.	Titan	Graph DBMS	5.47	-0.02	+0.35
5.	5.	↑ 6.	ArangoDB	Multi-model	3.15	+0.15	+1.00
6.	6.	↓ 5.	Virtuoso	Multi-model	1.86	-0.03	-0.83
7.	7.	7.	Giraph	Graph DBMS	1.03	-0.03	+0.08
8.	↑ 9.	↑ 11.	GraphDB	Multi-model	0.64	+0.03	+0.43
9.	↓ 8.	↓ 8.	AllegroGraph	Multi-model	0.61	-0.02	+0.15
10.	10.	↓ 9.	Stardog	Multi-model	0.54	-0.04	+0.11
11.	11.	↓ 10.	Sqrrl	Multi-model	0.50	-0.01	+0.24
12.	12.	12.	InfiniteGraph	Graph DBMS	0.30	+0.01	+0.11
13.	↑ 15.	↑ 18.	Blazegraph	Multi-model	0.25	+0.02	+0.16

Figure 13: Neo4j As a Leading Graph Database

How Neo4j is Different Than Traditional Databases? Graph databases are much different than traditional relational databases like SQL. Instead of using tables with rows and columns, graph databases use a graph with nodes and relationships. Both of these types of databases have their place. Relational database is great for tabular data that is not really closely related. If we have a lot of nested relationships in relational database it can get very complicated with join tables and join queries and we need all kinds of primary and foreign keys and it can be real hard to deal with and even worse than that is it can be really costly on the system so graph databases are built to fix that problem and work with data that is much more closely related and more dynamic.

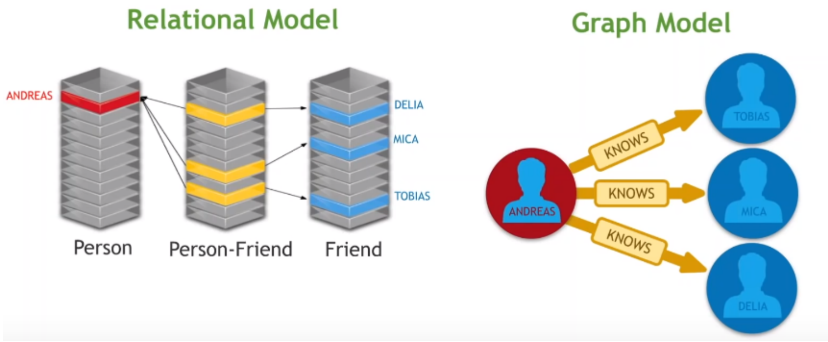


Figure 14: Level Of Complexity of Traditional Databases Comparing to Neo4j

Thus, because of the reasons stated above we choose Neo4j as our database.

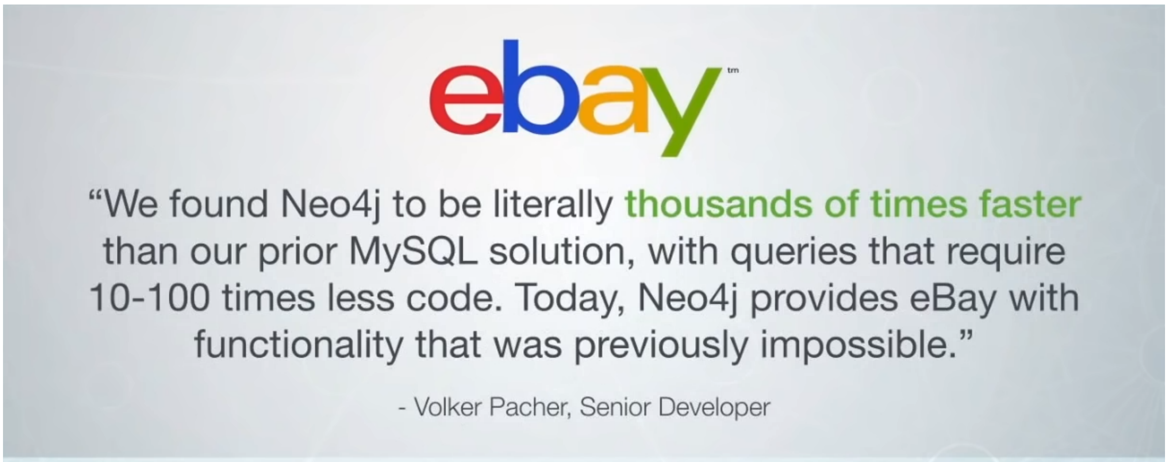


Figure 15: Ebay's comment about Neo4j

3.2 Advantages of Neo4j

Neo4j is very popular in lots of industries and it is a first choice of many companies. Neo4j gives advantage in many points. First of all it is based on handling complex data connections as a result of the increased volume and strength in the data, these companies gain lots of benefits among their competitive. Following are the advantages of Neo4j.

- **Easy to represent connected data:** It makes both easy and fast to traverse or navigate large amounts of data that has some sort of relationship

- **Can represent semi-structured data easily:** Data that does not fall into natural structure can be easily represented in a graph database
- **Cypher Commands:** Cypher commands are human readable and very easy to learn
- **Simple and Powerful Data Model:** The property graph data model is simple yet still very powerful. The basic building blocks are known to relationships and they can contain data in the form of key value pairs or properties unlike the relational model.
- **Join Aspect:**There's no need for complex and costly joins to retrieve connected or related data. Instead the graph database uses a natural concept of relationships. Relationships in a graph actually formed paths so querying or traversing a graph involves following those pats and because of that path oriented nature of the graph data model, the majority of path based operations are extremely efficient.
- **Performance:** Traversing a relationship is done in constant time so query performance does not decrease when data grows and Cypher is designed for graphs so it is very simple to write graph traversals based on pattern matching. . Neo4j is only graph database that combines native graph storage, scalable architecture optimized for speed, and ACID compliance to ensure predictability of relationship-based queries. [10]
- **Real-time insights:** Neo4j provides results based on real-time data.
- **High availability:** Neo4j is highly available for large enterprise real-time applications with transactional guarantees.[15]
- **Biggest graph community in the world:** Neo4j has the largest and most contributor graph community.
- **Easy to learn:** Mature UI with intuitive interaction and built-in learning.[10]

3.3 Properties of Neo4j

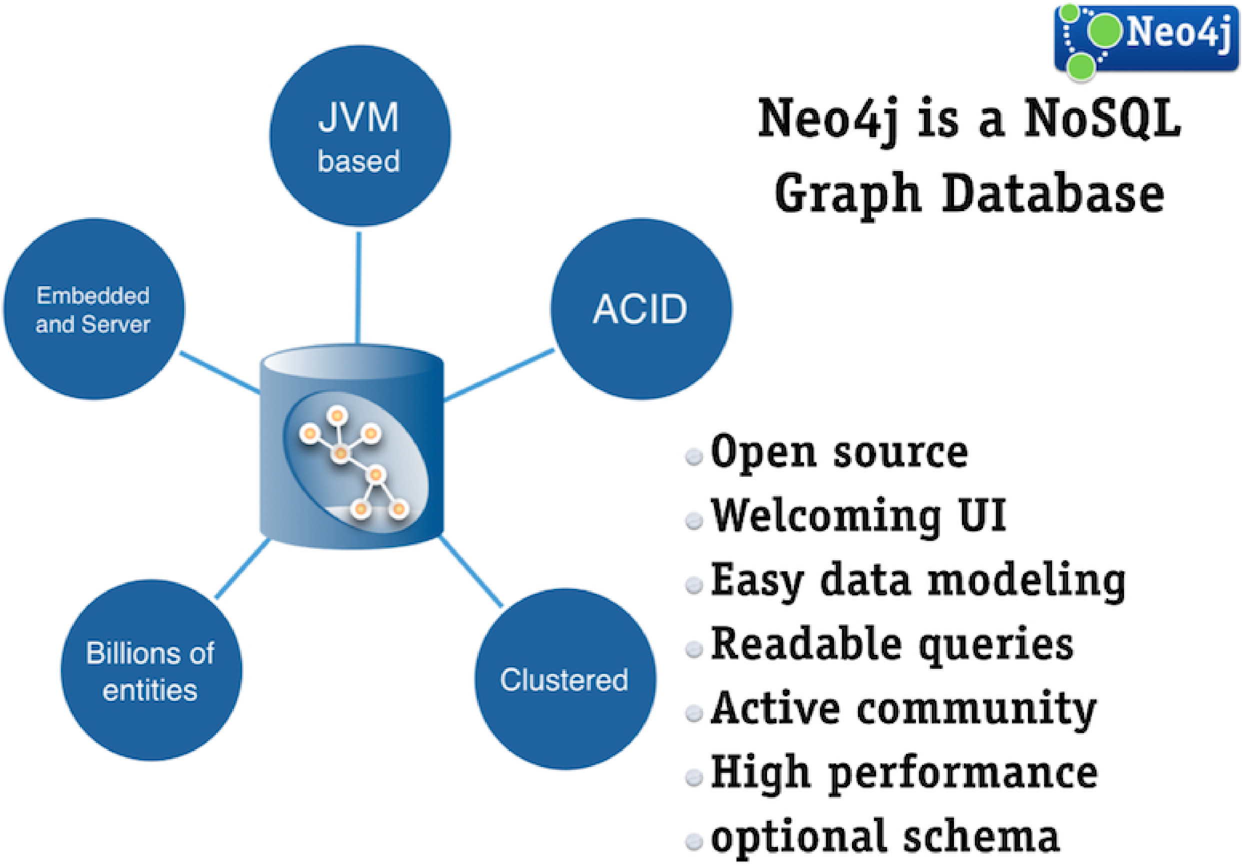


Figure 16: General Look at Neo4j

Following are properties of Neo4j;

- **Data model (flexible schema):** Neo4j has property graph model. It can be explained like graph has nodes and these nodes are connected with each other. Nodes and their relationships store data in key-value pairs known as properties. Neo4j has also flexible schema it means properties can be added or removed when it is necessary.
- **ACID properties:** Neo4j supports full ACID (Atomicity, Consistency, Isolation, and Durability) rules.

- **Scalability and reliability:** Database can be scaled by increasing the number of reads/writes, and the volume without effecting the query processing speed and data integrity. Neo4j also provides support for replication for data safety and reliability.
- **The traversal of the graph:** The traversal is the operation of visiting a set of nodes in the graph by moving between nodes connected with relationships. It's a unique operation to the graph model for data retrieval. Querying the data using a traversal only takes into account the data that's required, therefore it is not needed to query the entire data set in an expensive operation, like is the case with join operations on relational data. [1]
- **Cypher Query Language:** Neo4j provides a powerful declarative query language known as Cypher. It uses ASCII-art for depicting graphs. Cypher is easy to learn and can be used to create and retrieve relations between data without using the complex queries like Joins.[9]
- **Built-in web application:** Neo4j provides a built-in Neo4j Browser web application. Using this, creating and querying graph data can be done.
- **Drivers:** Neo4j can work with
  1. REST API to work with programming languages such as Java, Spring, Scala etc.
  2. Java Script to work with UI MVC frameworks such as Node JS.
  3. It supports two kinds of Java API: Cypher API and Native Java API to develop Java applications.
- **Indexing:** Neo4j supports Indexes by using Apache Lucence.

3.4 Performance In Neo4j

Neo4j provides fast and efficient graph experience and the strongest part of it is; Neo4j can traverse millions of nodes in milliseconds. Also even exponentially increasing data size does not effect the performance of Neo4j unlike relational databases.

Volker Pacher, eBay developer and Neo4j client: "Our Neo4j solution is literally a thousand times faster than the previous MySQL solution, with searches that require between 10 and 100 times less code".

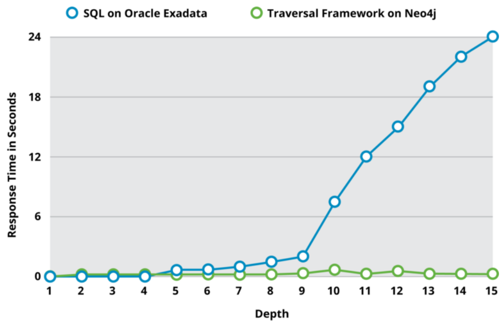


Figure 17: Query times for Oracle Exadata vs Neo4j

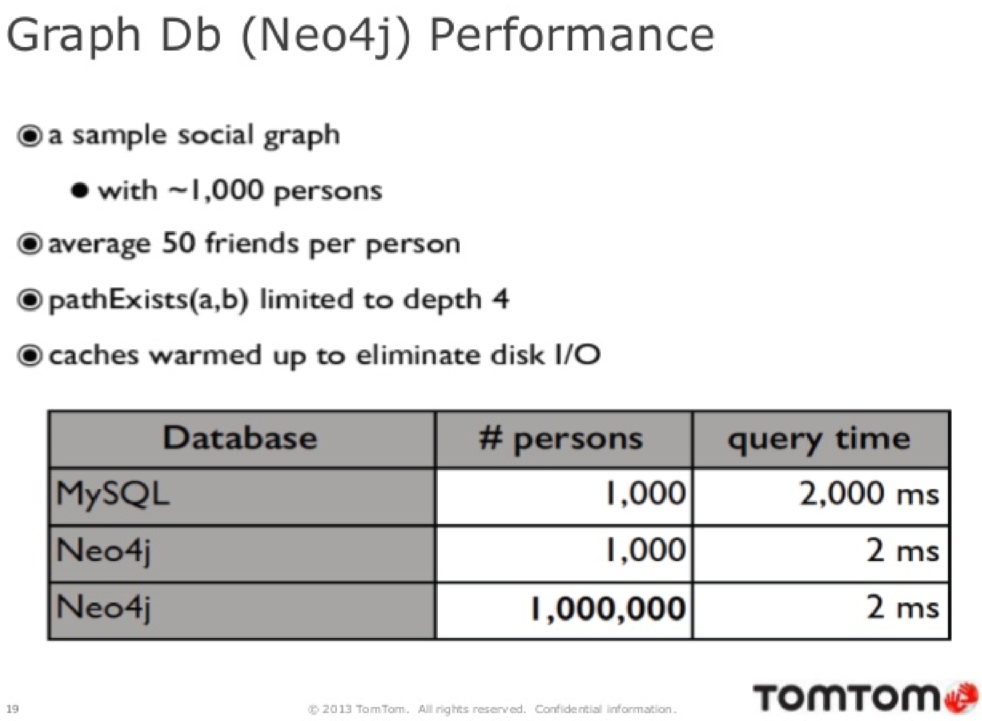


Figure 18: Tomtom's Comparison of Neo4j with MySQL

3.4.1 How To Increase Performance Of Neo4j?

- Increasing the size of available heap memory (Between 8G-16G).
- Increasing open file limit from default 1024 to at least 40000 to be sure.

- In order to avoid costly disk access, making sure of relevant graph data is cached in memory.
- For the non-Neo4j tasks running on the computer a sufficient memory should be reserved.(At least 16G)
- Simple algorithms leads to increased performance.
- All related nodes and edges should be kept in server memory before giving results.
- Traversals should be independent.
- Indexes should be used.

3.5 Cypher Query Language

Cypher is a declarative language for working with graphs and graph data for both reading and writing to the graph and it is very expressive and powerful. Also Cypher defines patterns in the given graph data.

- Cypher is declarative language: This means that we specify the data that we are interested in. We do not specify how to get that data from the database.
- Cypher is very human readable language and it is accessible not just for developers everyone can easily learn and use it.
- Cypher has expressions similar to SQL like WHERE, ORDER BY and simple condition statements like <, =,>. Its difference with sql is; Cypher is designed to represent graph data patterns for example it has MATCH property this property is built on finding and specifying patterns in the data

3.5.1 Structure

Nodes Nodes represents data entities and they can have labels and each node represents different single data entities. It is equivalent to records in a relational database Nodes can also have properties which are basically attributes. Nodes are shown with parentheses like (p:Product).

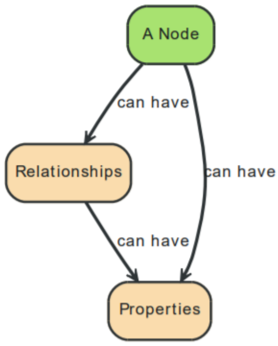


Figure 19: Node Representation

Relationships In Cypher; between the nodes we have lines which represent the relationship between each node. Relationships can also have properties just like nodes which is something that is much different than SQL. Also relationships have directions. Relationship is shown as –> between two nodes.

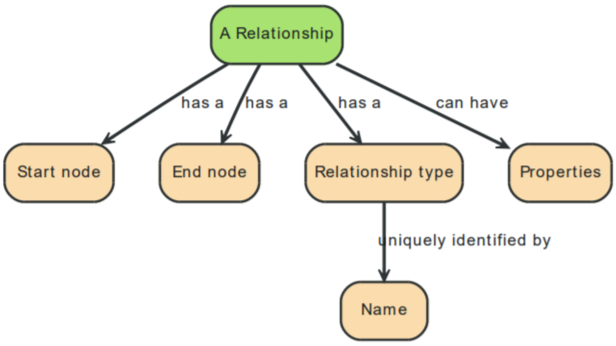


Figure 20: Relationship Representation

3.5.2 Operations In Cypher

Create: It is used to create nodes and relationships between them
We created a node representing us with five properties;

- Name: 'Ozge Koroglu'
- Country: 'Turkey'
- City: 'Istanbul'

- DateOfBirth: '21.05.1994'
- School:'ULB'

  With this Cypher code;

```
CREATE (n:Person {name :'Ozge Koroglu', country: 'Turkey',
city: 'Istanbul', DateOfBirth:'21.05.1994', School:'ULB'}) RE-
TURN n
```

Figure 21: Create Person's Node

- Name: 'Anna Turu Pi'
- Country: 'Spain'
- City: 'Barcelona'
- DateOfBirth: '30.07.1995'
- School:'ULB'

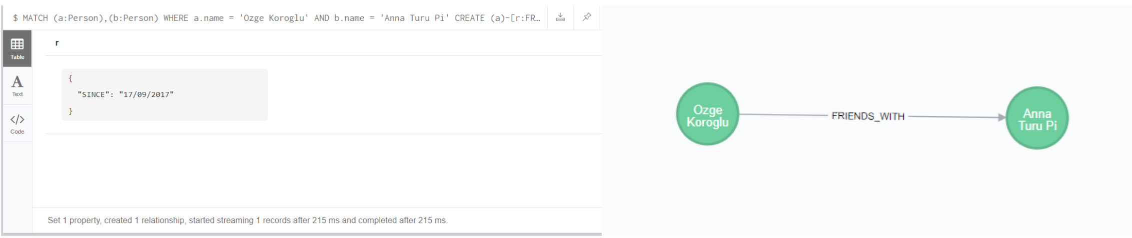
  With this Cypher code;

```
CREATE (n:Person {name :'Anna Turu Pi', country: 'Spain',
city: 'Barcelona', DateOfBirth:'30.07.1995', School:'ULB'})
RETURN n
```

We created a relationship called "FRIENDS_WITH" with the property "SINCE";

With this Cypher code;

```
MATCH (a:Person),(b:Person) WHERE a.name = 'Ozge Koroglu'
AND b.name = 'Anna Turu Pi' CREATE (a)-[r:FRIENDS_WITH
{SINCE:"17/09/2017"}]->(b) RETURN r
```



(a) Result in Console (b) After Creating Relationship

Figure 22: Create Relationship Between Two Nodes

Match: Match finds specified patterns in the data.

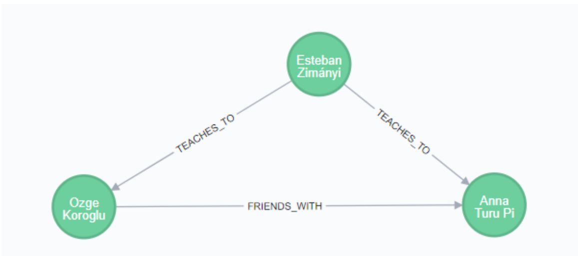


Figure 23: Relationships

With this Cypher code we showed all people whom Esteban Zimányi teaches to;

```
MATCH (a:Person )<-[:TEACHES_TO]-(b:Person{ name: 'Este-
ban Zimányi'}) RETURN a.name
```

Figure 24: Match Result

Set: This is used to update properties in the nodes and relationships.

With this Cypher Code we changed Esteban Zimányi's date of birth to '01.01.1966'

```
MATCH (n { name: 'Esteban Zimányi' }) SET n.DateOfBirth =
'01.01.1966' RETURN n
```

Delete This operator deletes nodes or relationships in the data.

With this Cypher code we deleted Ozge Koroglu

```
MATCH (n:Person { name: 'Ozge Koroglu' }) DELETE n
```

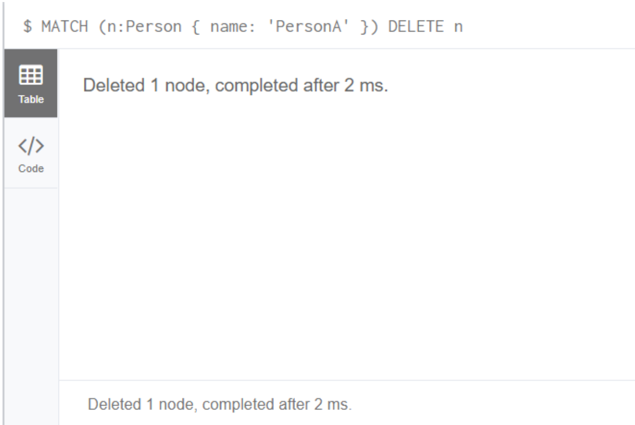


Figure 25: Delete Result

3.5.3 Loading Data With Cypher

There are lots of ways to import data in Neo4j but the most common way is upload it as a csv file. Load CSV operator is built into Neo4j and this operator is used for small or medium size datasets up to 10 million records. If we want to upload data that has more than 10 million records than we should use [USING PERIODIC COMMIT[n]] property. If we dont use this property this means that we are processing whole file in one run and creating everything in one transaction

Load CSV: This operator is used for importing CSV files into Neo4j.

```
[USING PERIODIC COMMIT [1000]]

LOAD CSV WITH HEADERS FROM "(file|http)://" AS row

MATCH  (:Label {property: row.header})
CREATE (:Label {property: row.header})
MERGE  (:Label {property: row.header})
```

Figure 26: Load CSV Operator Structure

3.6 Use Cases of Neo4j

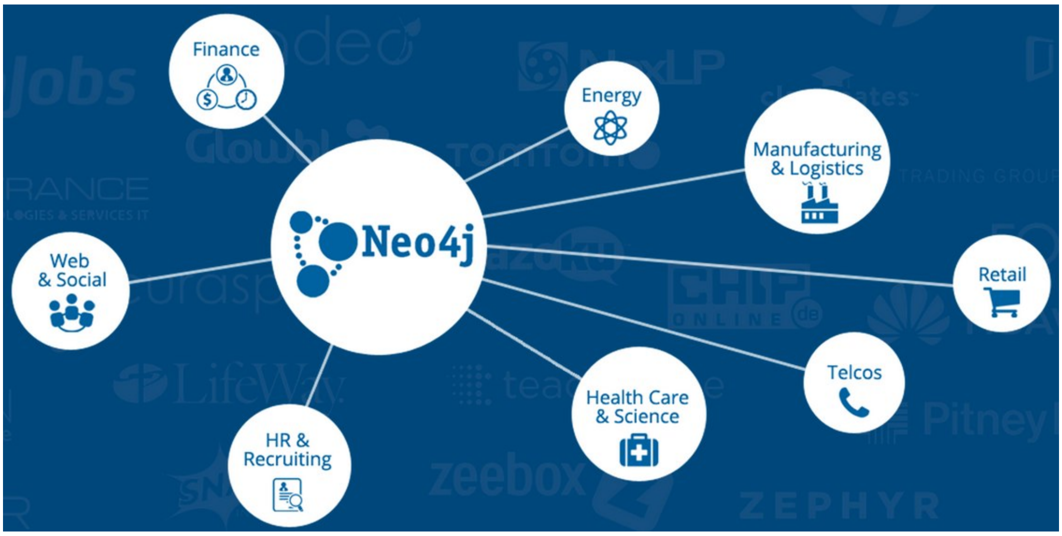


Figure 27: Use Cases Of Neo4j

The common use cases are;

Real Time Recommendations: Recommendation algorithms finds relationships between people, products and other services related to purpose based on user's previous behaviors. Neo4j is able to store interconnected data about customers and products and since Neo4j doesn't need indexing at every suggestion it provides very fast and effective algorithm to deal with real time data. Walmart uses Neo4j for this purpose

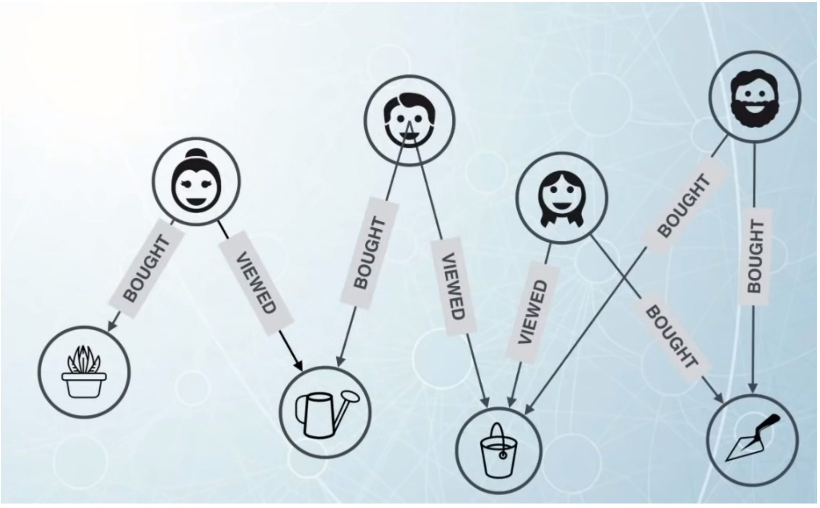


Figure 28: Real Time Recommendations Graph Design

Master Data Management: In large organizations, different systems stores information about customers, employees, titles and supply chain. With the graph model it is easy to bring data from different systems create views about customers or can keep track of all the information about the organizational system itself. Cisco uses Neo4j for this purpose and the company also uses Neo4j for their help desk solution

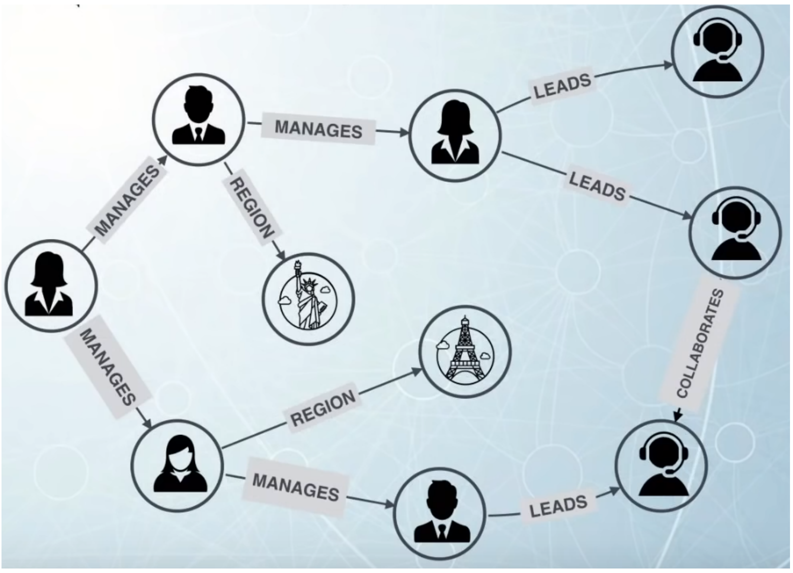


Figure 29: Master Data Management Graph Design

Fraud Detection: Fraud detection is very important in finance industry. Nowadays in order not to be detected by bank's fraud algorithms people use different approaches like open several bank accounts with valid information and do normal transactions without being an outlier. So people open false bank accounts with the same identity token and withdraw all the money in all bank accounts. It is hard to detect that behavior but it is very easy to see that with graph because the pattern of the people opening bank accounts using the same identity token can be easily detected as a pattern in a graph

Graph Based Search: Metadata is available for things like products, articles etc. And being able to model metadata as a graph allows to enhance search meaning users are able to find more relevant things for them. For example LinkedIn; When search is executed we don't see random or alphabetical sorted results we first see the relevant ones. Lufthansa uses Neo4j for this matter.

Network & IT Operations: If data center is modelled as a graph then dependency analysis can easily be applied on network systems to get conclusions like if one virtual machine goes down how many applications will be affected. Hp uses Neo4j to model their network for some large telecommunication providers.

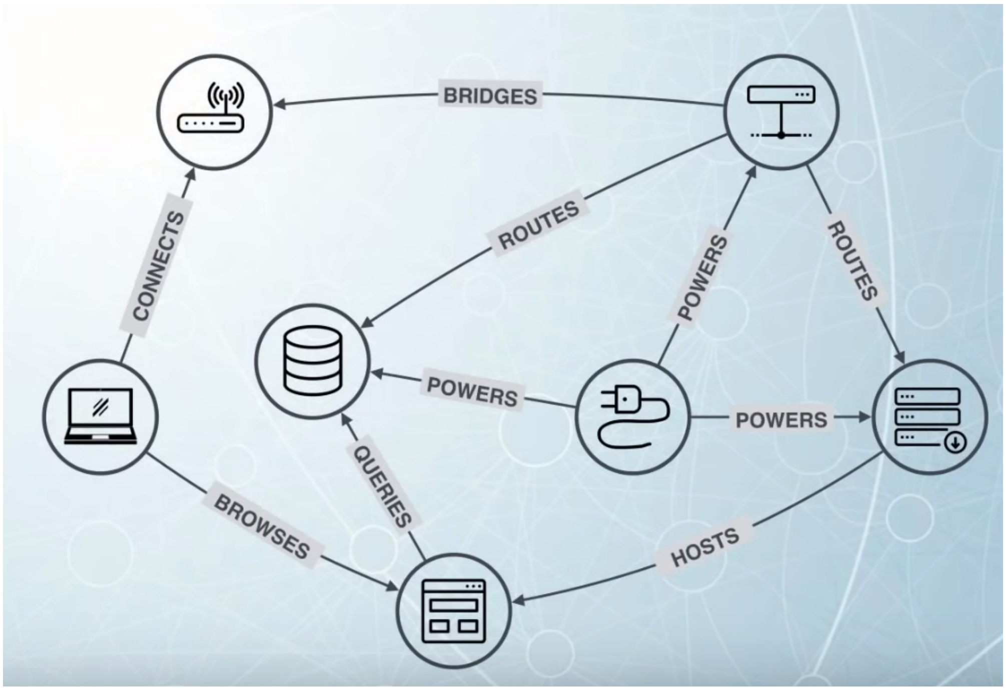


Figure 30: Network IT Operations Graph Design

Identity & Access Management: Within large organizations there are hundreds of users and controlling who can access to which information is crucial for security reasons. So creating groups and roles for each user comes in handy in this situation. This kind of data is very rich and connected and can be easily handled by Neo4j. UPC London uses Neo4j for that and it received 2014 Graphic awards for "Best İdentity and access management app"

4 Neo4j Application

Software For the graph database, *Neo4j Community Edition 3.2.5* has been used, and for the relational database, SQL Server 2017.

4.1 Use Case Selected

As proposed in graph database benchmark guidelines [4], the best tests to benchmark a graph database are: traversal (which includes the calculation of the shortest path), graph analysis, connected components, communities, centrality measures, pattern matching and graph anonymisation. It is also commented that among the domains where graph databases prove to be more beneficial are the shortest path graph analysis and real time analysis of traffic networks. In our implementation, we are going to model flight routes, as they have the ideal properties to benchmark a graph database. Airports and airlines are elements where the information lies on the their inter communications.

4.2 Data

The data set selected to perform the benchmark was a data set of flight routes provided by OpenFlights.org [13]. It provided three flat files, *airlines.dat*, *airports.dat*, *routes.dat*.
Because of the size concerns we created synthetic data in addition to our existing data tables. Before creating new data we had 67663 different routes and now we have 1193413 different routes. The rows we created have dummy variables, they do not have any connection with the existing data except their types. So our queries mostly resulted in initial data results. This data creation process was applied because the more data we have, the more accurate bench-marking results we get. Also unlike traditional databases, adding more data to Neo4j does not effect its performance.

4.2.1 Implementing Data

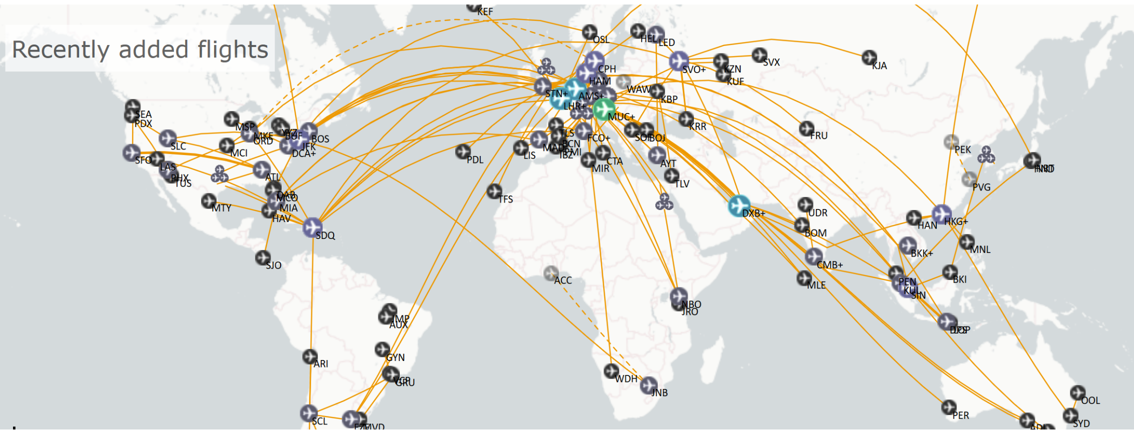


Figure 31: OpenFlights.org

Neo4j: To create the Neo4j database we developed a python code. This code uses py2neo library to access Neo4j database and it reads our data (external source) to create nodes, relationships, properties and indexes

*make_graph

```python
from __future__ import print_function
import csv
import sys

from math import radians, cos, sin, asin, sqrt
from py2neo import Graph, Node, Relationship, authenticate
authenticate("localhost:7474", "user", "pass")

def create_nodes(graph, label, sourcefile, fieldnames):

def create_airline_nodes(graph, sourcefile):

def create_airport_nodes(graph, sourcefile):

known_distances = {}

def haversine(lat1, lon1, lat2, lon2):

def get_distance(source_airport_node, destination_airport_node):


def create_route_nodes(graph, sourcefile, airline_nodes, airport_nodes):

def create_schema(graph):

def main():

if __name__ == '__main__':
    sys.exit(main())
```

Figure 32: Structure of the python code

The original airport data had latitude and longitude attributes. In order to present better visualization we created a function that calculates the distance between two connected airports. Route data has *source_ airport* and *destination_ airport* So we created a route node and we assigned the distance between *source_ airport* and *destination_ airport* as a name attribute to route node. In the end four types of nodes are Airlines, Airports and Routes, and they have the following communications:

Route	→	TO	→	Airport
Route	→	FROM	→	Airport
Route	→	OF	→	Airline

Table 2: Graph database schema

We implemented our data to Neo4j with this schema;

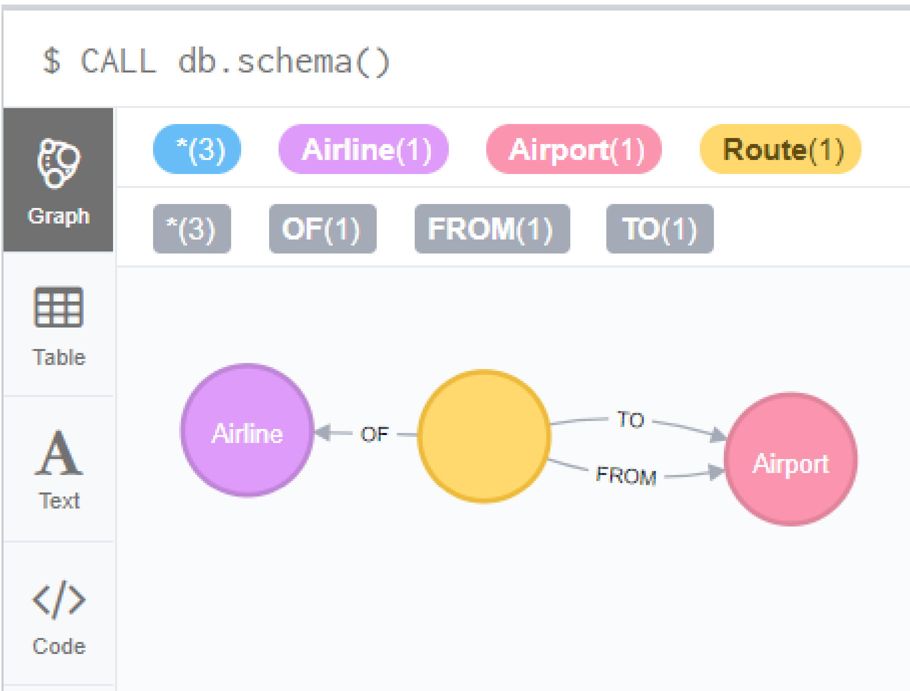


Figure 33: Initial Schema

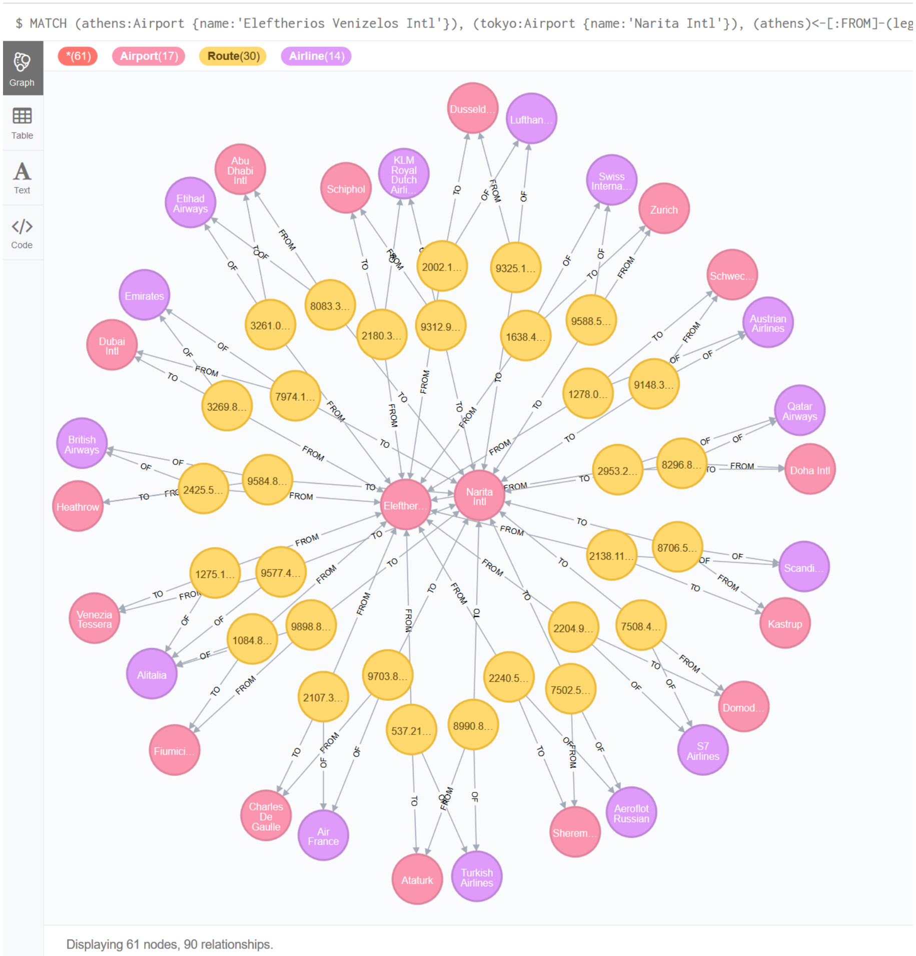


Figure 34: Example of a query in Neo4j

SQL: A relational database was created importing each flat file as a table and then we created foreign key references between tables.

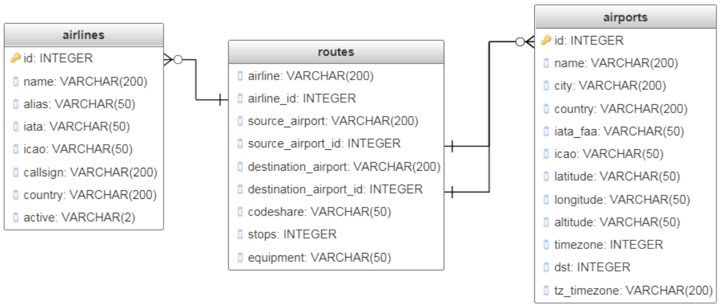


Figure 35: Relational database diagram

4.2.2 Export data

To export the Neo4j, we chose to use the apoc library. It is needed to authorize Neo4j to run the plugins. For that, this line of code has to be added in *neo4j.conf*: `apoc.export.file.enabled=true`.

Export to CSV

`apoc.export.csv.query(query,file,config)`: exports results from the Cypher statement as CSV to the provided file
`apoc.export.csv.all(file,config)`: exports whole database as CSV to the provided file
`apoc.export.csv.data(nodes,rels,file,config)`: exports given nodes and relationships as CSV to the provided file
`apoc.export.csv.graph(graph,file,config)`: exports given graph object as CSV to the provided file

We exported the entire database executing the following command in cypher:

```
CALL apoc.export.csv.all("/temp/neo4j_database_csv_file.csv",
{batchSize:10}) YIELD file, source, format, nodes, relationships,
properties, time, rows
```

```
$ CALL apoc.export.csv.all("/temp/neo4j_database_csv_file2.csv",{batchSize:10}) YIELD file, source, fo...
```

file	source	format	nodes	relationships	properties	time	rows
"/temp /neo4j_database_csv_file2.csv"	"database: nodes(305914), rels(926225)"	"csv"	305914	926225	1604692	7882	0

Started streaming 1 records after 7897 ms and completed after 7897 ms.

Figure 36: Exporting Neo4j database to CSV file

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	_id	_labels	active	alias	id	iata	icao	name	callsign	country	betweenne	partition	community	altitude	longitude	tz_timezon	iata_faa	latitude	city	dst	timezone	stops
8498	8838	:Airport			2517		SAZY	Aviador C Campos		Argentina	0	8885	8885	2569	########	America/Cc	CPC	########	San Martin	N		-3
8499	8839	:Airport			2518		SBAA	Conceicao Do Araguai:		Brazil	3232	12334	13201	653	########	America/Bc	CDJ	########	Conceicao	S		-4
8500	8840	:Airport			2519		SBAF	Campo Delio Jardim D		Brazil	0			110	########	America/Sao_Paulo		########	Rio De Jane	S		-3
8501	8841	:Airport			2520		SBAM	Amapa		Brazil	0			45	########	America/Fortaleza		2.077.511	Amapa	S		-3
8502	8842	:Airport			2521		SBAQ	Araraquara		Brazil	0	8885	9867	2334	########	America/Sa	AQA	-21.812	Araracuara	S		-3
8503	8843	:Airport			2522		SBAR	Santa Maria		Brazil	########	8885	9867	23	########	America/Fc	AJU	-10.984	Aracaju	S		-3
8504	8844	:Airport			2523		SBAS	Assis		Brazil	0			1850	########	America/Sao_Paulo		########	Assis	S		-3
8505	8845	:Airport			2524		SBAT	Alta Floresta		Brazil	0	8885	9867	947	########	America/Ca	AFL	########	Alta Florest	S		-4
8506	8846	:Airport			2525		SBAU	Aracatuba		Brazil	########	8885	9867	1361	########	America/Sa	ARU	########	Aracatuba	S		-3
8507	8847	:Airport			2526		SBBE	Val De Cans Intl		Brazil	########	8885	10373	54	########	America/Bc	BEL	-137.925	Belem	S		-4
8508	8848	:Airport			2527		SBBG	Comandante Gustavo		Brazil	0			600	########	America/Sa	BGX	########	Bage	S		-3
8509	8849	:Airport			2528		SBBH	Pampulha Carlos Drum		Brazil	########	8885	9867	2589	########	America/Sa	PLU	########	Belo Horizc	S		-3
8510	8850	:Airport			2529		SBBI	Bacacheri		Brazil	0			3057	########	America/Sa	BFH	########	Curitiba	S		-3
8511	8851	:Airport			2530		SBBQ	Major Brigadeiro Door		Brazil	0			3657	########	America/Sao_Paulo		########	Barbacena	S		-3
8512	8852	:Airport			2531		SBBR	Presidente Juscelino K		Brazil	########	8885	9867	3479	########	America/Sa	BSB	-158.711	Brasilia	S		-3
8513	8853	:Airport			2532		SBBU	Bauru		Brazil	0			2025	-490.538	America/Sa	BAU	########	Bauru	S		-3
8514	8854	:Airport			2533		SBBV	Boa Vista		Brazil	0	8885	10373	276	########	America/Bc	BVB	2.846.311	Boa Vista	S		-4
8515	8855	:Airport			2534		SBBW	Barra Do Garcas		Brazil	0			1147	########	America/Campo_Gran		########	Barra Do G	S		-4
8516	8856	:Airport			2535		SBCA	Cascavel		Brazil	########	8885	9867	2473	########	America/Sa	CAC	########	Cascavel	S		-3
8517	8857	:Airport			2536		SBCC	Cachimbo		Brazil	0			1762	########	America/Boa_Vista		########	Itaituba	S		-4
8518	8858	:Airport			2537		SBCF	Tancredo Neves Intl		Brazil	########	8885	9867	2715	########	America/Sa	CNF	########	Belo Horizc	S		-3
8519	8859	:Airport			2538		SBCG	Campo Grande		Brazil	########	8885	9867	1833	-546.725	America/Ca	CGR	########	Campo Gra	S		-4

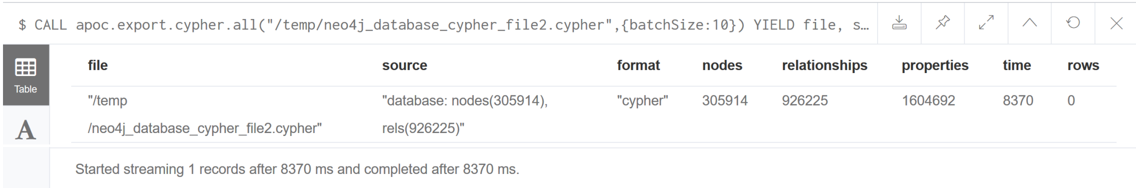
Figure 37: CSV file containing Neo4j database

Export to cypher script

`apoc.export.cypher.all(file,config)`: exports whole database incl. indexes as Cypher statements to the provided file
`apoc.export.cypher.data(nodes,rels,file,config)`: exports given nodes and relationships incl. indexes as Cypher statements to the provided file
`apoc.export.cypher.graph(graph,file,config)` exports given graph object incl. indexes as Cypher statements to the provided file
`apoc.export.cypher.query(query,file,config)`: exports nodes and relationships from the Cypher statement incl. indexes as Cypher statements to the provided file
`apoc.export.cypher.schema(file,config)`: exports all schema indexes and constraints to cypher

The database was also exported to cypher a cypher script:

```
CALL apoc.export.cypher.all("/temp/neo4j_database_cypher_file.cypher",
{batchSize:10}) YIELD file, source, format, nodes, relationships,
properties, time, rows
```

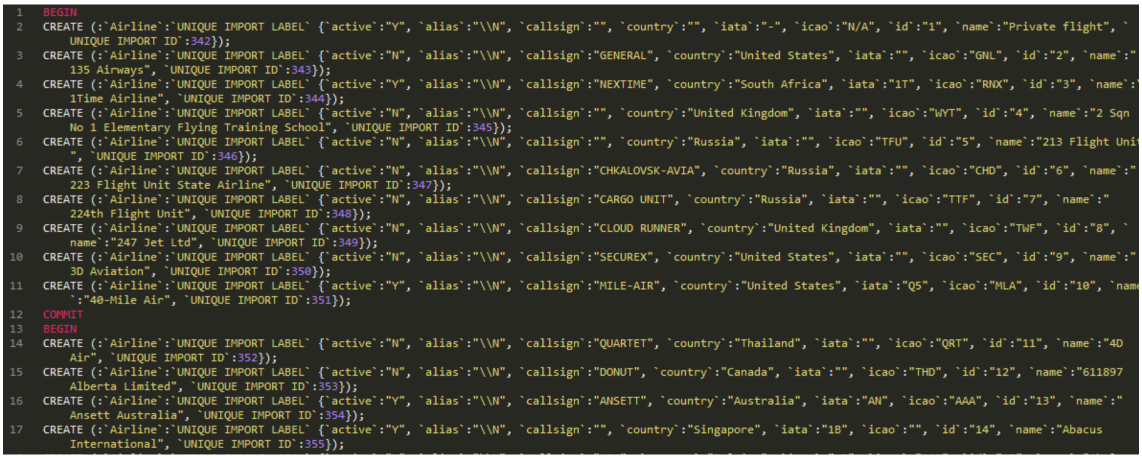


$ CALL apoc.export.cypher.all("/temp/neo4j_database_cypher_file2.cypher",{batchSize:10}) YIELD file, s…

file	source	format	nodes	relationships	properties	time	rows
"/temp /neo4j_database_cypher_file2.cypher"	"database: nodes(305914), rels(926225)"	"cypher"	305914	926225	1604692	8370	0

Started streaming 1 records after 8370 ms and completed after 8370 ms.

Figure 38: Exporting Neo4j database to cypher script



```
BEGIN
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"Y", `alias`:"\\N", `callsign`:"", `country`:"", `iata`:"-", `icao`:"N/A", `id`:"1", `name`:"Private flight", `
    UNIQUE IMPORT ID`:342});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"N", `alias`:"\\N", `callsign`:"GENERAL", `country`:"United States", `iata`:"", `icao`:"GNL", `id`:"2", `name`:"
    135 Airways", `UNIQUE IMPORT ID`:343});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"Y", `alias`:"\\N", `callsign`:"NEXTIME", `country`:"South Africa", `iata`:"1T", `icao`:"RNX", `id`:"3", `name`:"
    1Time Airline", `UNIQUE IMPORT ID`:344});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"N", `alias`:"\\N", `callsign`:"", `country`:"United Kingdom", `iata`:"", `icao`:"WYT", `id`:"4", `name`:"2 Sqn
    No 1 Elementary Flying Training School", `UNIQUE IMPORT ID`:345});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"N", `alias`:"\\N", `callsign`:"", `country`:"Russia", `iata`:"", `icao`:"TFU", `id`:"5", `name`:"213 Flight Unit
    ", `UNIQUE IMPORT ID`:346});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"N", `alias`:"\\N", `callsign`:"CHKALOVSK-AVIA", `country`:"Russia", `iata`:"", `icao`:"CHD", `id`:"6", `name`:"
    223 Flight Unit State Airline", `UNIQUE IMPORT ID`:347});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"N", `alias`:"\\N", `callsign`:"CARGO UNIT", `country`:"Russia", `iata`:"", `icao`:"TTF", `id`:"7", `name`:"
    224th Flight Unit", `UNIQUE IMPORT ID`:348});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"N", `alias`:"\\N", `callsign`:"CLOUD RUNNER", `country`:"United Kingdom", `iata`:"", `icao`:"TWF", `id`:"8", `
    name`:"247 Jet Ltd", `UNIQUE IMPORT ID`:349});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"N", `alias`:"\\N", `callsign`:"SECUREX", `country`:"United States", `iata`:"", `icao`:"SEC", `id`:"9", `name`:"
    3D Aviation", `UNIQUE IMPORT ID`:350});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"Y", `alias`:"\\N", `callsign`:"MILE-AIR", `country`:"United States", `iata`:"Q5", `icao`:"MLA", `id`:"10", `name
    `:"40-Mile Air", `UNIQUE IMPORT ID`:351});
COMMIT
BEGIN
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"N", `alias`:"\\N", `callsign`:"QUARTET", `country`:"Thailand", `iata`:"", `icao`:"QRT", `id`:"11", `name`:"4D
    Air", `UNIQUE IMPORT ID`:352});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"N", `alias`:"\\N", `callsign`:"DONUT", `country`:"Canada", `iata`:"", `icao`:"THD", `id`:"12", `name`:"611897
    Alberta Limited", `UNIQUE IMPORT ID`:353});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"Y", `alias`:"\\N", `callsign`:"ANSETT", `country`:"Australia", `iata`:"AN", `icao`:"AAA", `id`:"13", `name`:"
    Ansett Australia", `UNIQUE IMPORT ID`:354});
CREATE (:`Airline`:`UNIQUE IMPORT LABEL` {`active`:"Y", `alias`:"\\N", `callsign`:"", `country`:"Singapore", `iata`:"1B", `icao`:"", `id`:"14", `name`:"Abacus
    International", `UNIQUE IMPORT ID`:355});
```

Figure 39: Cypher script containing Neo4j database

4.3 Query Examples (Neo4j-SQL)

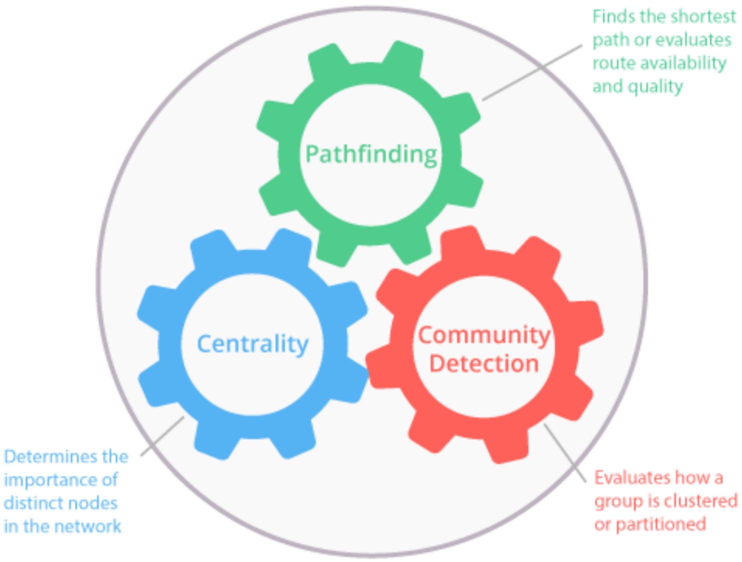


Figure 40: Algorithms for graph databases

Add libraries: It has been commented that Neo4j includes graph algorithms that allow us to perform queries that would be impossible to perform in SQL. Libraries of algorithms can be downloaded and added in Neo4j as plugins.

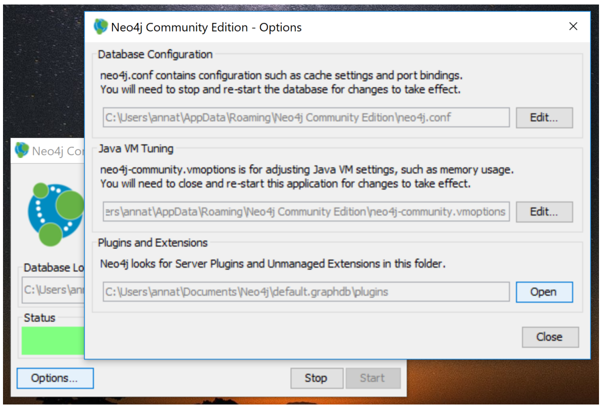


Figure 41: Add jar files in plugin folder

It is needed to authorize Neo4j to run the plugins. For that, this line of code has to be added in *neo4j.conf*: `dbms.security.procedures.unrestricted=apoc.*` (e.g., apoc library).

After that, Neo4j needs to be restarted, and it can be verified that the plugin is working by writing the following command in Neo4j browser:

```
CALL dbms.procedures() YIELD name, signature, description
WHERE name starts with "apoc"
RETURN name, signature, description
```

4.3.1 Shortest Path

This algorithm is the one that better justifies the existence of graph databases. Its calculation is impossible with SQL. In SQL it is needed to specify the number of layers the route has.

First query example: find the shortest path to go from an airport in Madrid to an airport in Seoul.

```
MATCH p=shortestpath((src:Airportcity: 'Madrid')-[r:FROM|TO*..15]-
(dest:Airportcity: 'Seoul')) RETURN p
```

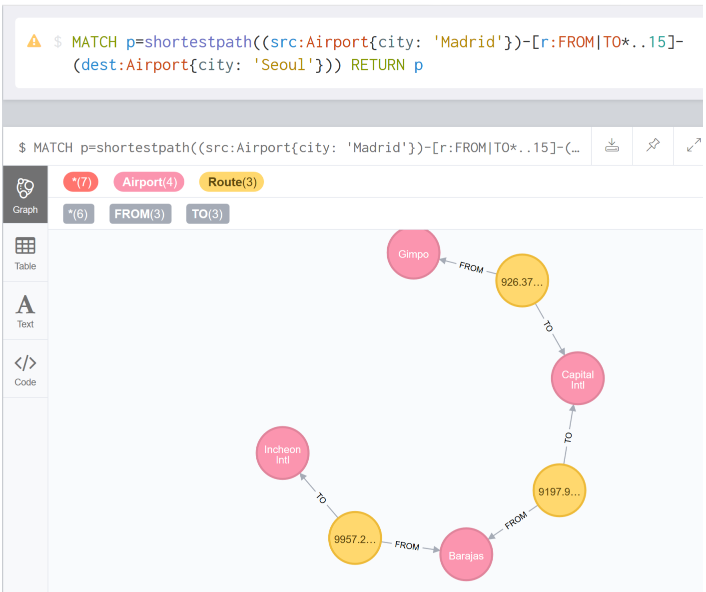


Figure 42: Shortest path query from Madrid to Seoul

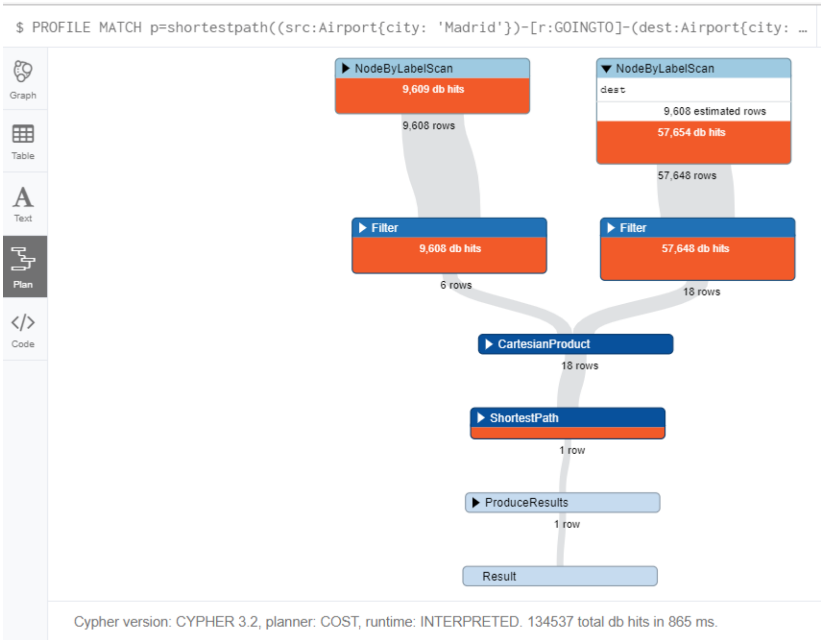


Figure 43: Pipeline of the shortest path query

The nodes can be expanded, and we see the airline to which each route belongs.

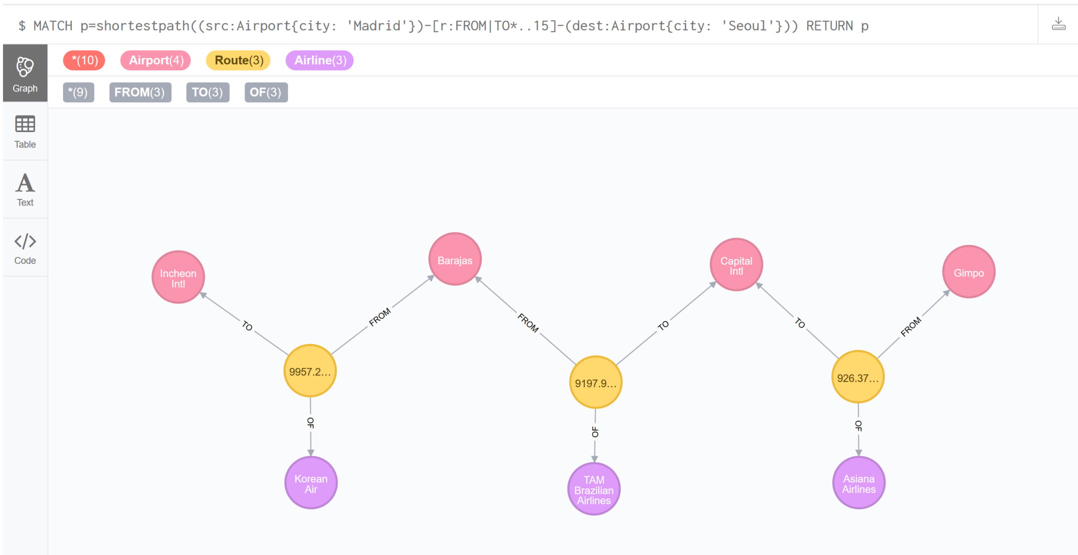


Figure 44: Expanded shortest path query

Second query example: find the shortest path between an airport in Seoul and an airport in Antwerp.

```
MATCH p=shortestpath((src:Airport{city:  'Seoul'})-[r:FROM|TO*..15]-
(dest:Airport{city:  'Antwerp'})) RETURN p
```

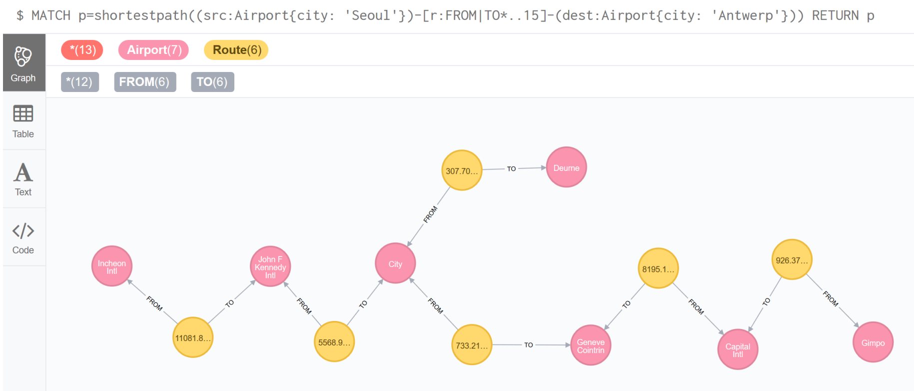


Figure 45: Shortest path query from Seoul to Antwerp

Paying attention to the relationships, it can be seen that the query doesn't output a physically possible travelling route from the origin city to the origin city. In the first query, one of the paths ends up in Seoul, but the other has two sources, Madrid and Seoul, and they both end up in Beijing. The second query has three origin airports, one in Antwerp and two in Seoul, and all the routes finish in Geneve.

The purpose of the algorithm is to find the shortest path to connect two nodes, independently of the physical meaning, but real routes can be created with the following modification:

Persistent inferred relationships: For each route going from an airport to another, a relationship connecting both airports has been added. This way, the shortest path query can look for only one type of relationship. If the objective is to find physically possible paths between two airports (e.g., not stepping into an airline) it will be assured looking for that inferred relationship that airports are being connected to airports.

Relationship *CONNECTED*. This relationship has the property `weight`, and is proportional to the number of routes between two airports. It is being used in the shortest path queries and community detection queries.

Cypher code to create the relationship:

```
MATCH (ap1:Airport)<-[:FROM]-(r:Route)-[:TO]->(ap2:Airport)
```

```
WHERE id(ap1) <> id(ap2)
WITH ap1, ap2, COUNT(*) AS weight
CREATE (ap1)-[c:CONNECTED]->(ap2)
SET c.weight = weight
```
In the figure below the database schema after adding the inferred relationship is displayed:

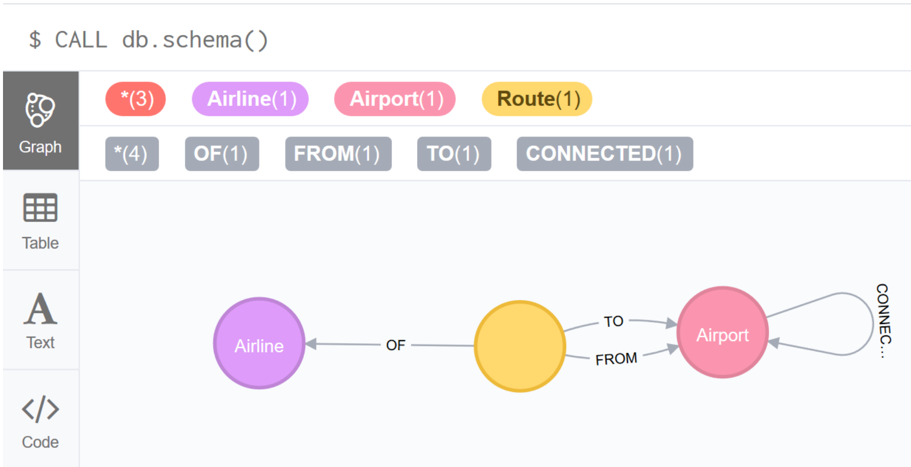


Figure 46: Neo4j DB schema after adding Connected relationships

Cypher code to delete the relationship:
```
MATCH (ap1:Airport)-[r:CONNECTED]->(ap2:Airport) DELETE r
```

Relationship *GOINGTO*. This relationship saves the route and airline information in its properties. It is being used in the shortest path queries and community detection queries.

Cypher code to create the relationship:
```
MATCH (ap1:Airport)<-[:FROM]-(r:Route)-[:TO]->(ap2:Airport)
WHERE id(ap1) <> id(ap2)
WITH ap1, ap2, r
MATCH (r)-[:OF]->(al:Airline)
CREATE (ap1)-[g:GOINGTO]->(ap2)
SET g.distance = r.distance
SET g.route = id(r)
SET g.airline = al.name
```

In the figure below the database schema after adding the inferred relationship is displayed:

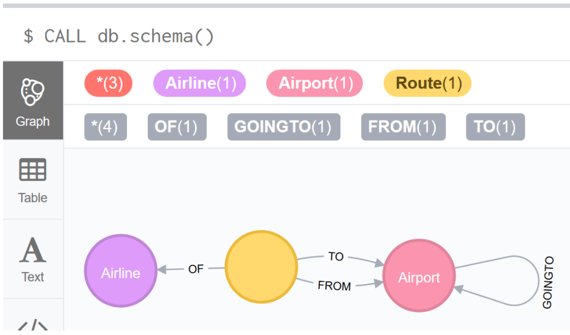


Figure 47: Neo4j DB schema after adding Goingto relationships

Cypher code to delete the relationship:
`MATCH (Airport)-[r:GOINGTO]->(Airport) DELETE r`

The first shortest path query is run again now with the inferred relationships:

```
MATCH p=shortestpath((src:Airport{city: 'Madrid'})-[r:GOINGTO]-
(dest:Airport{city: 'Seoul'})) RETURN p
```

Figure 48: Shortest path between Madrid and Seoul

Now the airports are directly connected to each other. The route node cannot be seen, but its identifier is saved as one of the relationship properties. With the follwoing query it can be verified if the route matches the requisites:
`MATCH (r:Route) WHERE id(r)=50276 RETURN r`

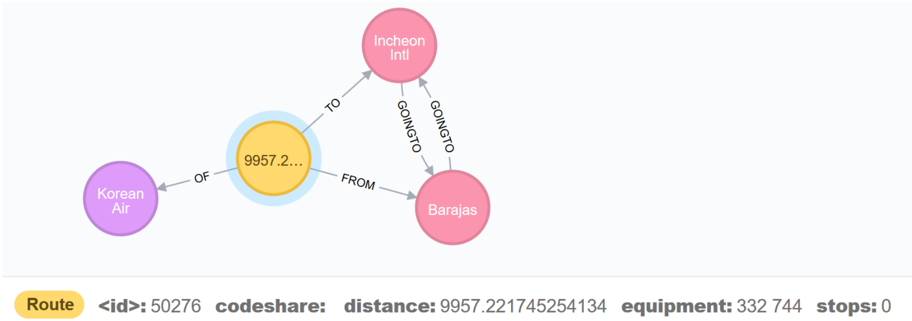


Figure 49: Shortest path outbound route output

It is verified that the relationship *GOINGTO* was equivalent to a real outbound route between Madrid and Seoul. The return rout is also verified:
`MATCH (r:Route) WHERE id(r)=50205 RETURN r`

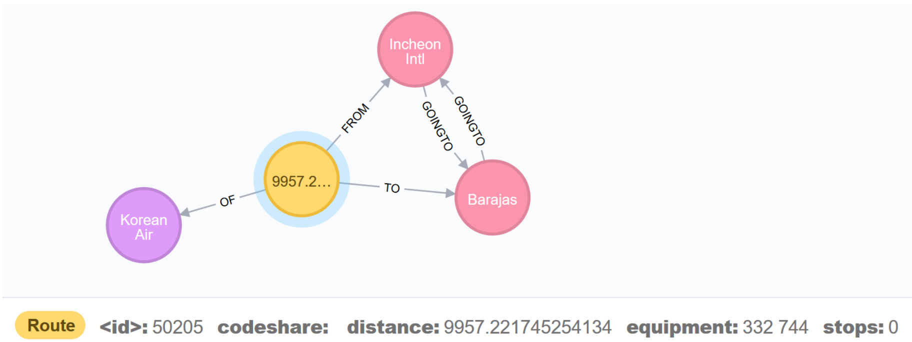
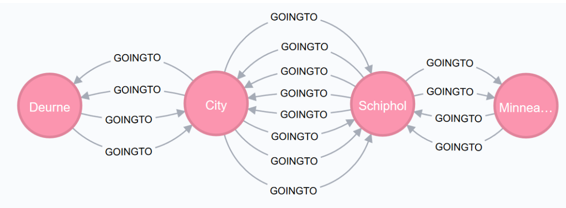
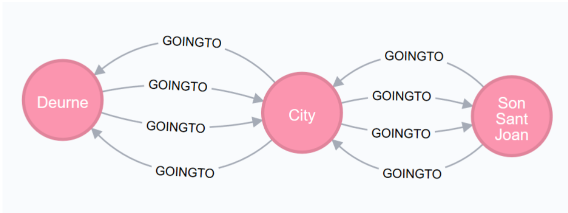


Figure 50: Shortest path return route output

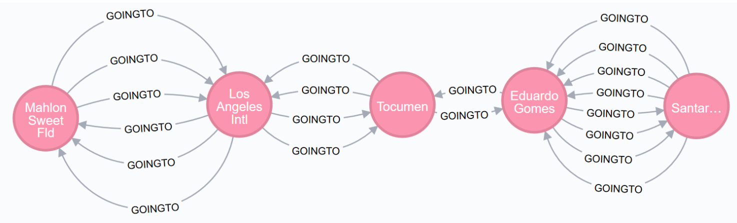
Other examples:



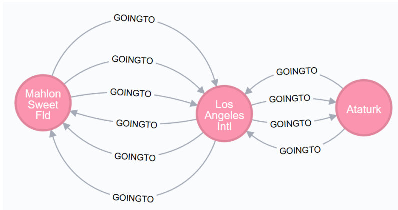
(a) From Antwerp to Minneapolis



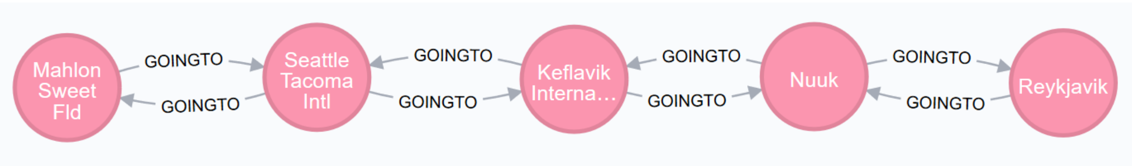
(b) From Antwerp to Mallorca



(c) From Santarem to Eugene



(d) From Istanbul to Eugene



(e) From Reykjavik to Eugene

Figure 51: Other shortest path examples

Shortest path in SQL Server: SQL Server has the limitation that it need to be specified the number of layers in the path. An alternative is to use a recursive query, but from our experience, it was not effective.

When executing the query, we obtain the following message: "The statement terminated. The maximum recursion 100 exhausted before statement completion."

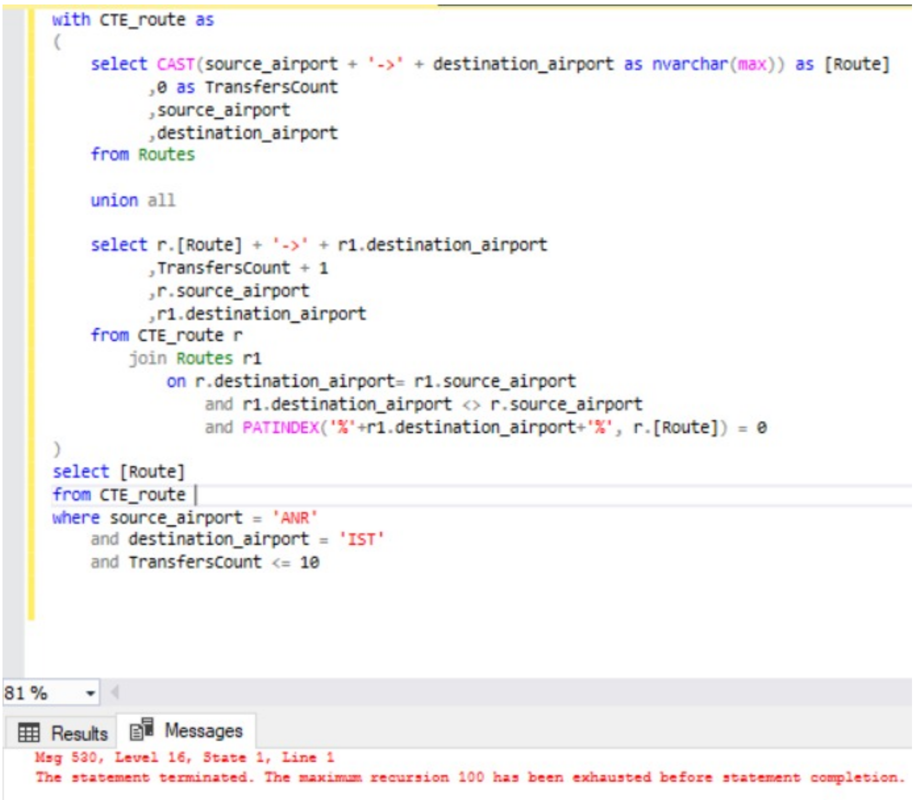


Figure 52: SQL Server recursive query output

For the same query, in Neo4j it only needs a few lines and the result is output in 794ms.

```
MATCH p=shortestpath((src:Airport{city: 'Antwerp'})-
[r:GOINGTO*..30]-(dest:Airport{city: 'Istanbul'})) RETURN p
```

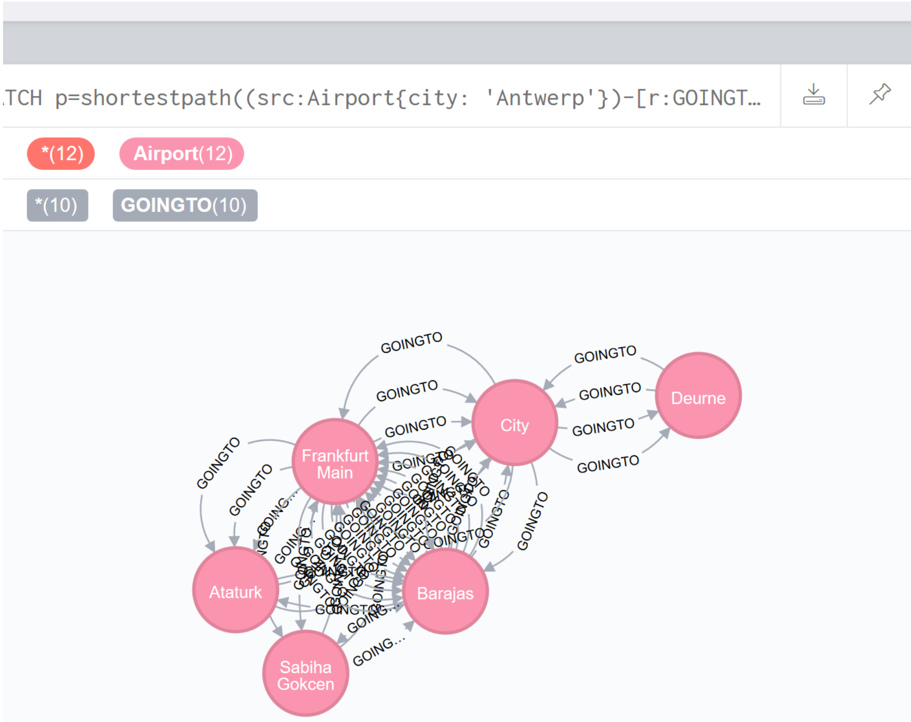


Figure 53: Neo4j query on Antwerp-Istanbul shortest path

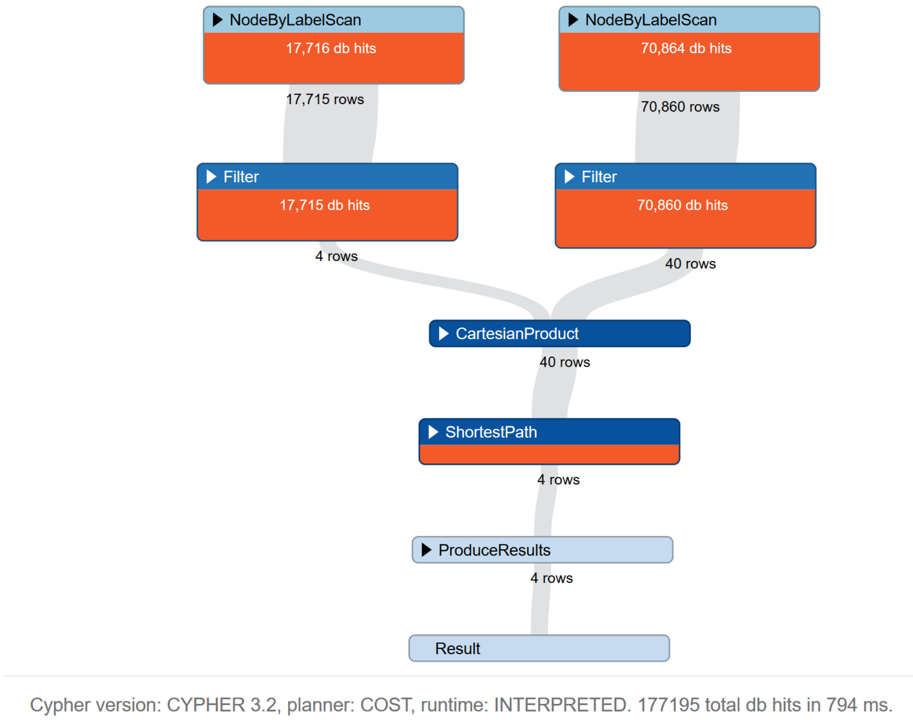


Figure 54: Pipeline of Neo4j query on Antwerp-Istanbul shortest path

4.3.2 Betweenness centrality:

The betweenness centrality of a node in a network is the number of shortest paths between two other members in the network on which a given node appears. Betweenness centality is an important metric because it can be used to identify "brokers of information" in the network or nodes that connect disparate clusters. [6]
This query shows the airports that have to be crossed more often by routes to go from one airport to another. In other worlds, the airports where more transfers take place. As it is displayed in the figure below, the airports highlighted are like bottlenecks that connect clusters of airports.

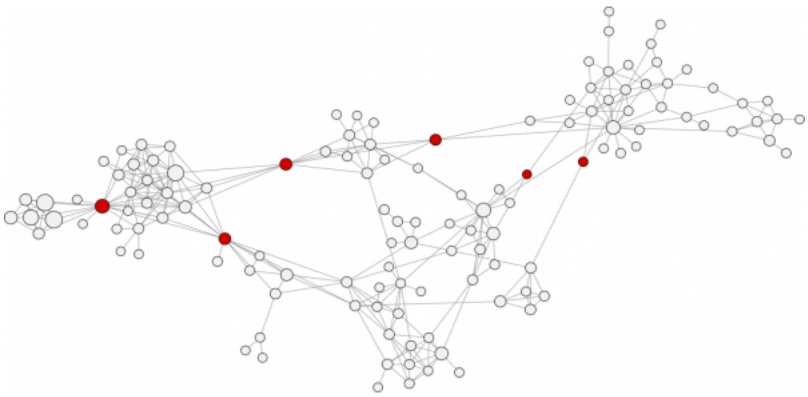


Figure 55: Concept of betweenness centrality

```
MATCH (ap:Airport)
WITH collect(ap) AS airports
CALL apoc.algo.betweenness(['CONNECTED'], airports, 'OUTGOING')
YIELD node, score
SET node.betweenness = score
RETURN node AS Airport, score ORDER BY score DESC LIMIT 25
```

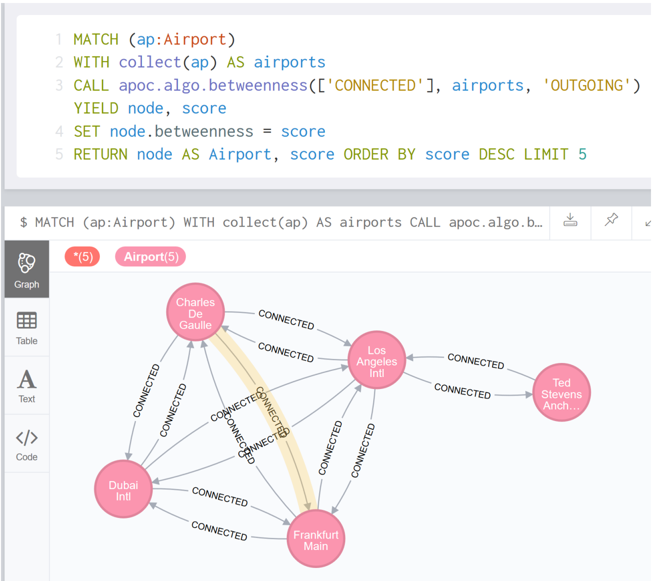


Figure 56: Betweenness centrality query result

The query outputs five big airports, which are commonly used to transfer during intercontinental journeys. It makes sense that they have the highest betwenness centrality.

Query performance: Writing *PROFILE* before the cypher query, outputs the pipeline of the query execution.

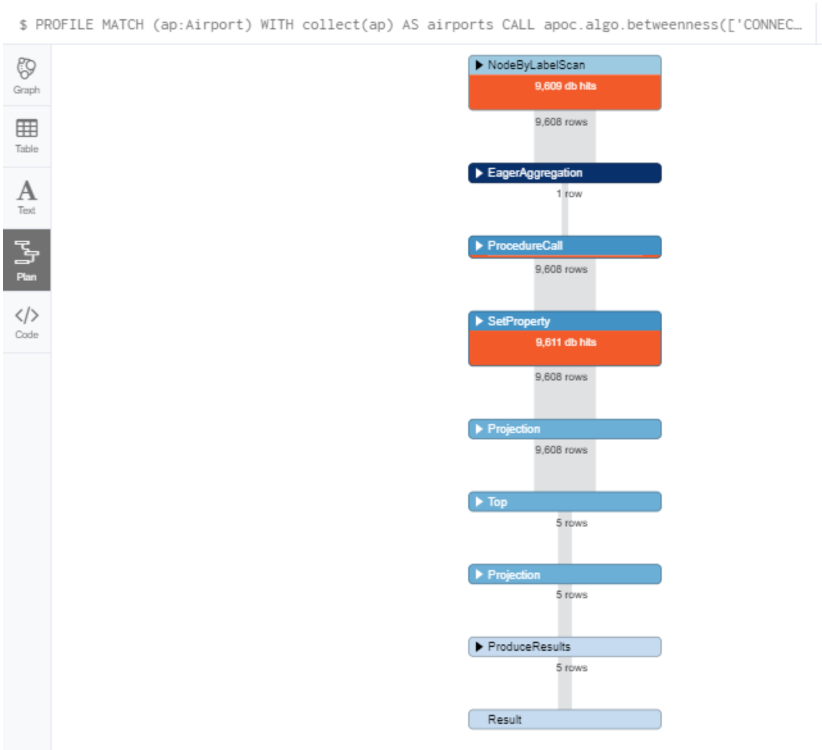


Figure 57: Pipeline of the betweenness centrality query

4.3.3 Closeness centrality:

Closeness centrality is the inverse of the average distance to all other characters in the network. Nodes with high closeness centality are often highly connected within clusters in the graph, but not necessarily highly connected outside of the cluster. [6]

This query outputs the airports that have more connections to different airports. In other words, it shows the locations that are more geographically isolated to be reached by other means of transport (e.g. islands). It can output the airports with more direct flights from different locations or the airlines that perform more routes.

Figure 58: Concept of closeness centrality

Query example: output the five airports with a higher closeness centrality:

```
MATCH (ap:Airport)
WITH collect(ap) AS airports
CALL apoc.algo.closeness(['CONNECTED'], airports, 'OUTGOING')
YIELD node, score
RETURN node AS Airport, score ORDER BY score DESC LIMIT 5
```

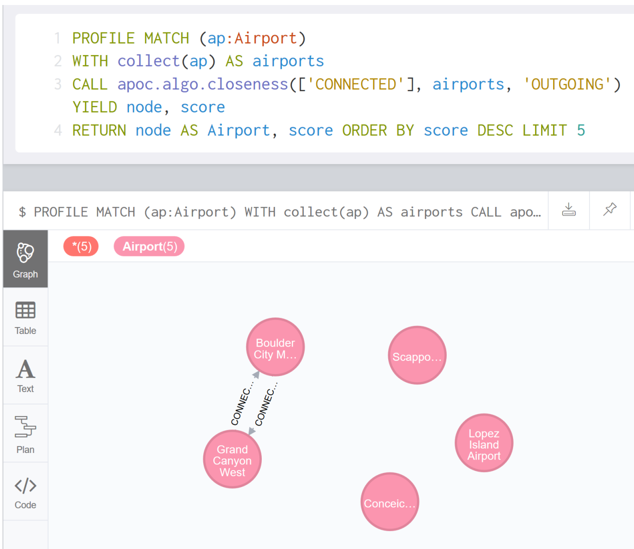


Figure 59: Closeness centrality query result

As predicted, the query outputs airports that are in highly touristic but geographically isolated locations: Lopez Island near Seattle, the river Araguaia in the middle of Brazil, the Grand Canyon of Colorado...

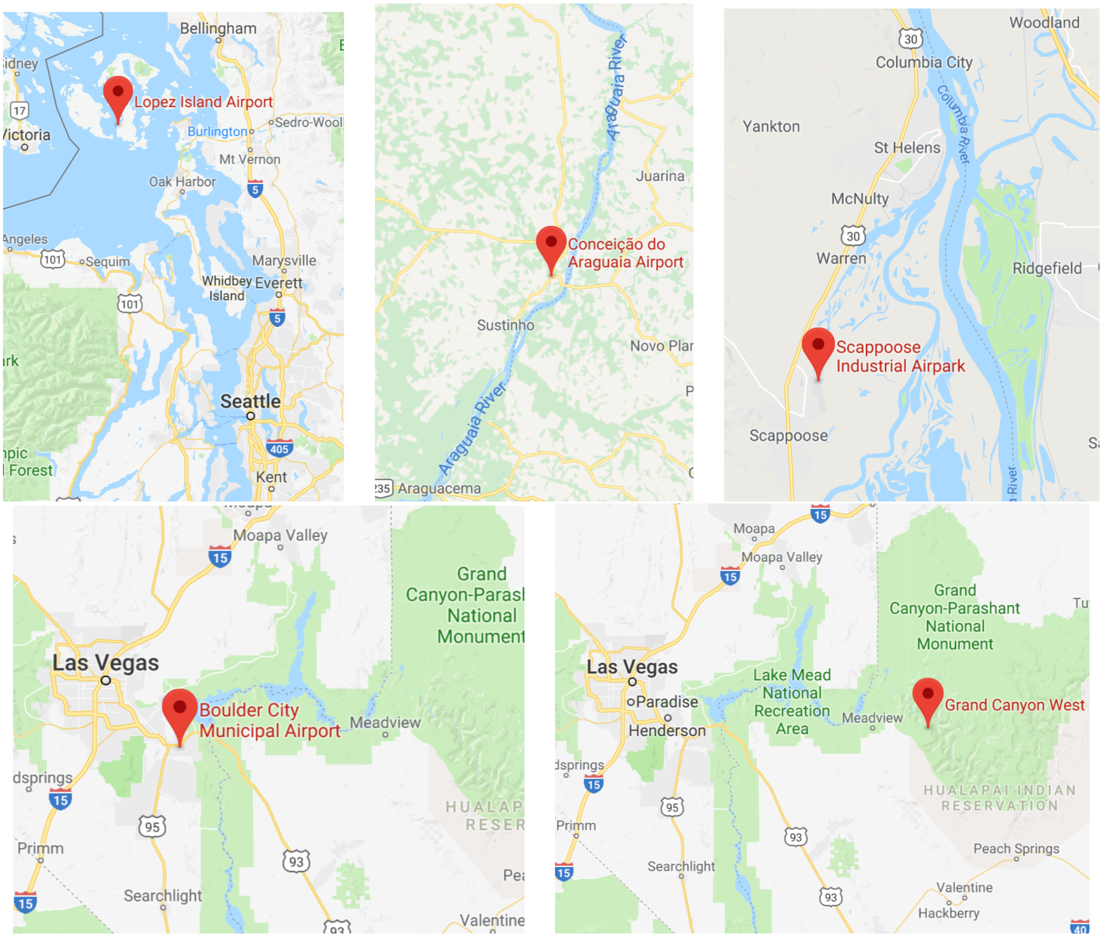


Figure 60: Location of the airports with highest closeness centrality

Query performance: Writing *PROFILE* before the cypher query, outputs the pipeline of the query execution.

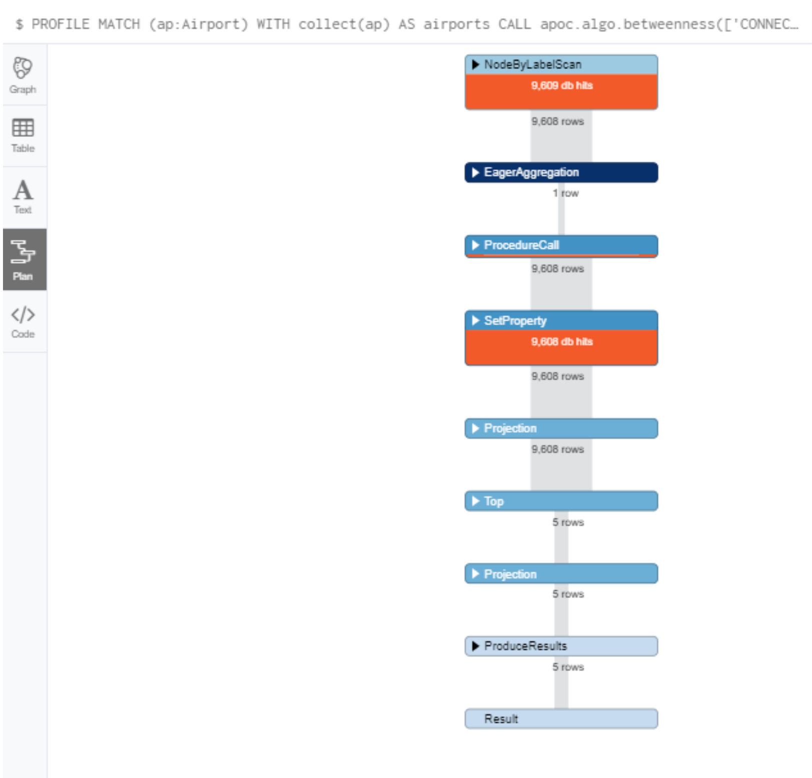


Figure 61: Pipeline of the closeness centrality query

4.3.4 PageRank:

The secret of Google's success was its search algorithm, PageRank. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites [11]. This algorithm can output the most connected airport or the most powerful airline (the node connected to more routes).

First query: output the most important airports

```
MATCH (ap:Airport) WITH collect(ap) AS airports
CALL apoc.algo.pageRank(airports) YIELD node, score
RETURN node, score ORDER BY score DESC LIMIT 10
```

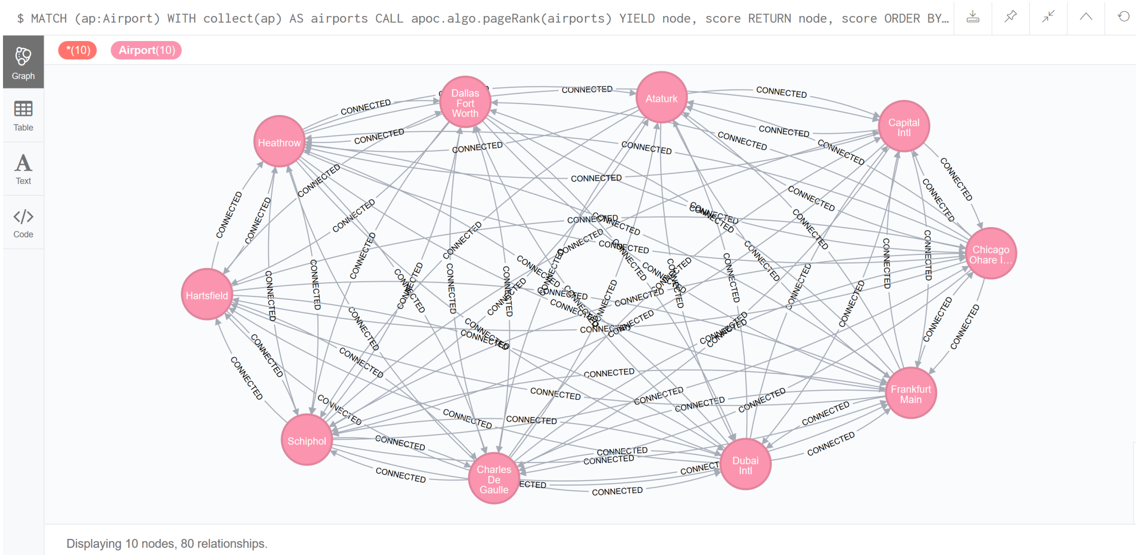


Figure 62: Airports pagerank result

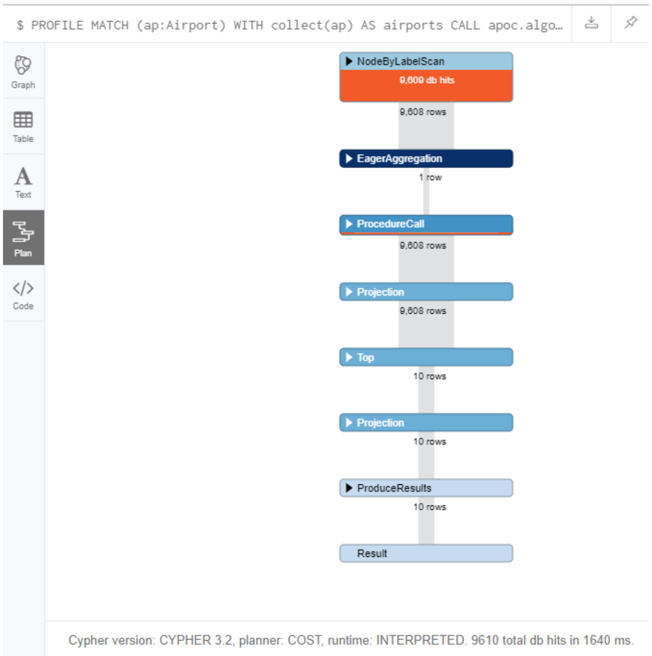


Figure 63: Pipeline of the airports pagerank query

The most important airports are from London, Paris, Frankfurt, Istanbul, Dubai, Beijing and the USA. The output is not surprising.

Second query: Output the most popular airlines.

```
MATCH (node:Airline) WITH collect(node) AS airlines
CALL apoc.algo.pageRank(airlines) YIELD node, score
RETURN node, score ORDER BY score DESC LIMIT 10
```

Figure 64: Airlines pagerank result

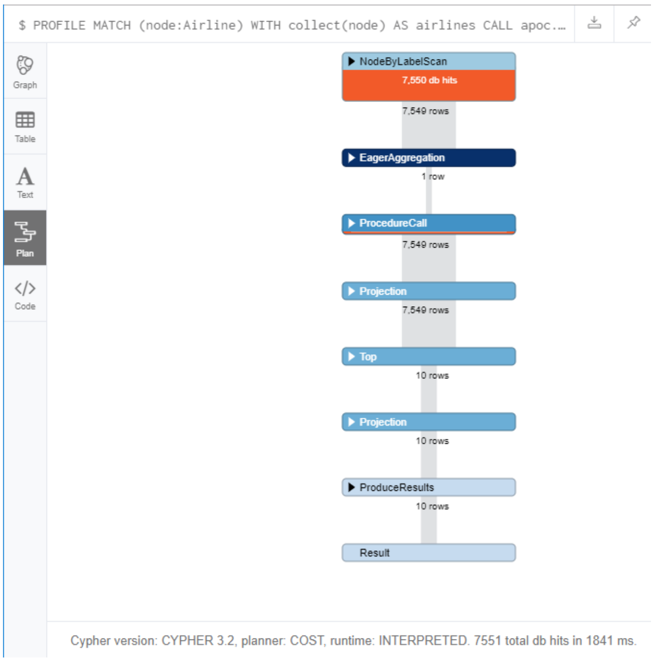


Figure 65: Pipeline of the airlines pagerank query

As a result we can see that Ryanair is the leading airline, followed by four companies from the USA and three from China.

4.3.5 Community Detection:

There are many algorithms for community detection: triangle counting, strongly connected components, ... This algorithms cluster together the nodes more related with each other. We have chosen an algorithm from the library APOC, and what the code below does, is classify the airport nodes in 40 partitions. The classification is determined on the weight of the connected relationships (the number of routes between each pair of airports).

Seeing as airports are geographical location, and routs are physical journeys between them, it is expected that geographically neighbouring airports will be clustered together. That hipothesis is verified below.

```
CALL apoc.algo.community(40,['Airport'],'partition',
'CONNECTED','OUTGOING','weight',10000)
MATCH (ap:Airport) WHERE exists(ap.partition) RETURN ap
```

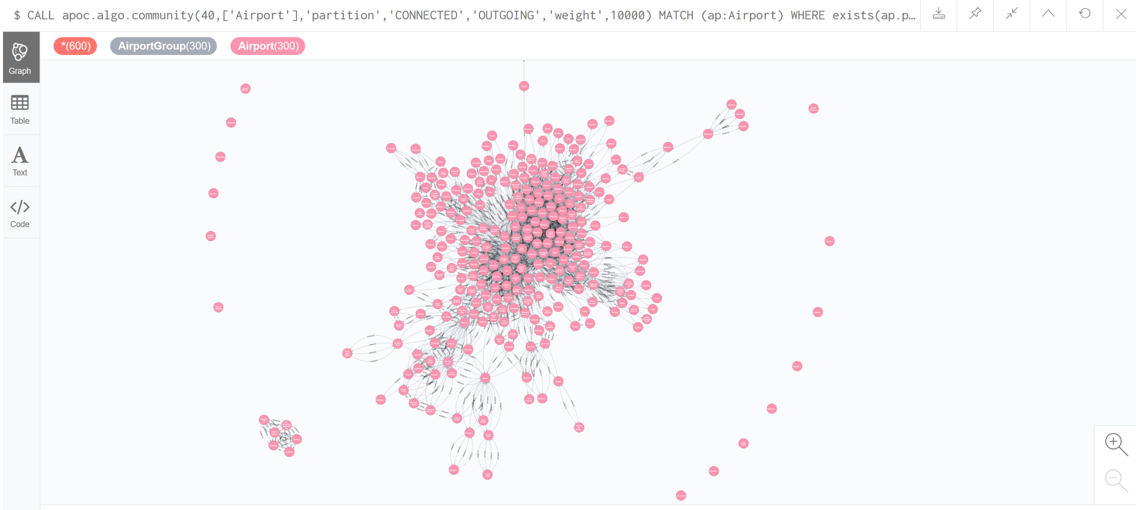


Figure 66: Community detection graph

The figure over these lines shows the shape of the graph after the nodes have been classified in partitions. To see which nodes belong to each partition, the partition number must be returned as output:

```
CALL apoc.algo.community(40,['Airport'],'partition',
'CONNECTED','OUTGOING','weight',10000)
MATCH (ap:Airport) WHERE exists(ap.partition)
RETURN ap.partition, ap.country, COUNT(*) AS num
ORDER BY ap.partition, num DESC
```

$ CALL apoc.algo.community(…

ap.partition	ap.country	num
6394	"Papua New Guinea"	23
6407	"Iceland"	4
6464	"Canada"	2
6520	"Canada"	5
6531	"Canada"	5
6544	"Canada"	2
6577	"Canada"	6
6584	"Canada"	17
6590	"Canada"	1
6624	"Algeria"	4
6640	"Nigeria"	14
6640	"Congo (Kinshasa)"	13
6640	"Ethiopia"	6
6640	"Cameroon"	5
6640	"Ghana"	5
6640	"Equatorial Guinea"	2

Figure 67: Community detection table

Figure 68: Pipeline of community detection query

Going back to the visualization of the community detection for airports, the partitions can be recognized and verified by looking at the table. The cluster of six nodes disconnected from the rest of airports is comprised of Papua New Guinea airports (the country can be seen by hovering over the nodes). They belong to the first partition in the table, 6394.

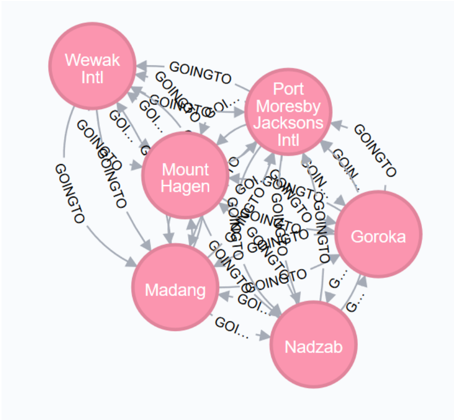


Figure 69: Papua New Guinea partition

The following part of the graph is a bit scattered, but it can be seen that they are all communicated to the central nodes. Hovering over them, we see that they all belong to Canada, and we can suppose that the more separated nodes are regional airports connected to bigger more important airports. That part of the graph is equivalent to seven partitions in the table.

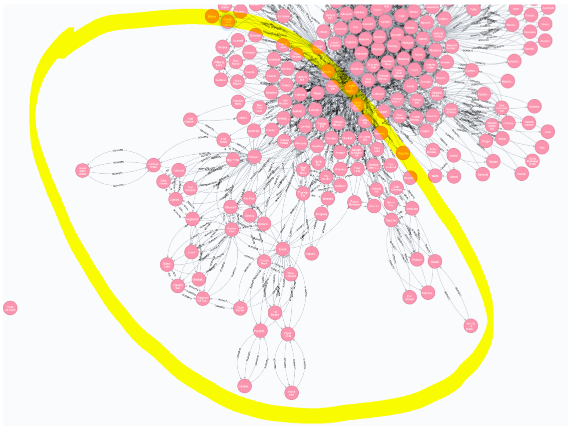


Figure 70: Canada partitions

Next to Canada, a group of nodes are separated, and those airports are all from Algeria. They must belong to partition 6624.

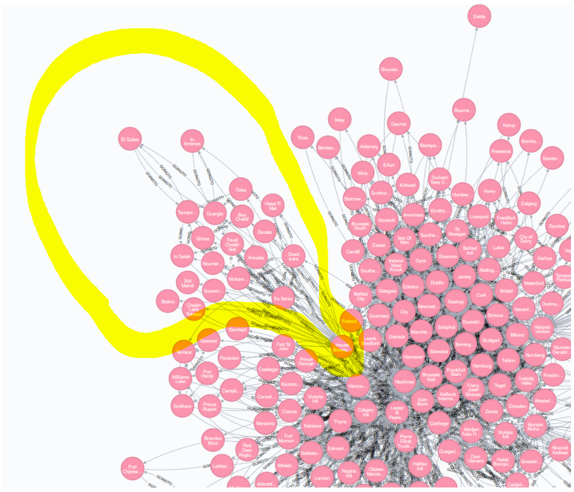


Figure 71: Algeria partition

The more centralized part of this subgraph are the airports from Finland. Some of those are connected with a Greenland's airport, which connects with other Greenland and Iceland airports.

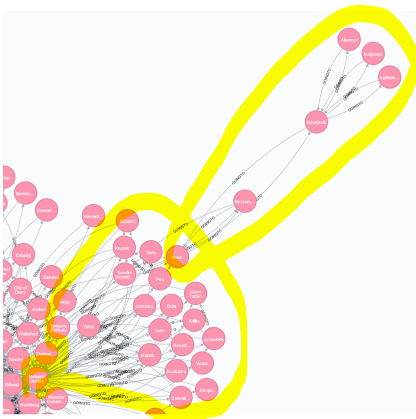
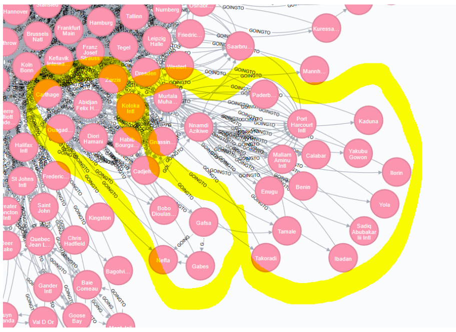


Figure 72: Finland, Greenland, Iceland partitions

The next subgraph shouws airports from different african countries interconnected with each other. On the left side, there are airports, and airports from african countries highly connected to them, and on the right side there are mainly nigerian airports, among other african aiports too.

6640	"Nigeria"	14
6640	"Congo (Kinshasa)"	13
6640	"Ethiopia"	6
6640	"Cameroon"	5
6640	"Ghana"	5
6640	"Equatorial Guinea"	2
6640	"Gabon"	2
6640	"Congo (Brazzaville)"	2
6640	"Burkina Faso"	2
6640	"Liberia"	2
6640	"Sierra Leone"	1
6640	"Togo"	1
6640	"Niger"	1
6640	"Chad"	1
6640	"Cote d'Ivoire"	1
6640	"Central African Republic"	1
6640	"Benin"	1

(a) Partition table



(b) Partition graph

Figure 73: Africa partition

Going back to the center of the graph, it is hard to recognize more than one partition, as it shows the central european airports, which are highly interconnected.

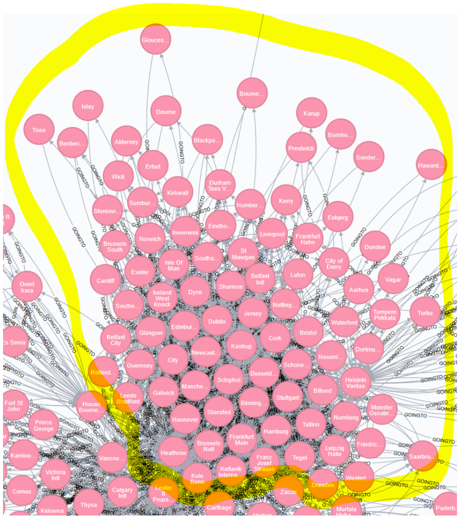


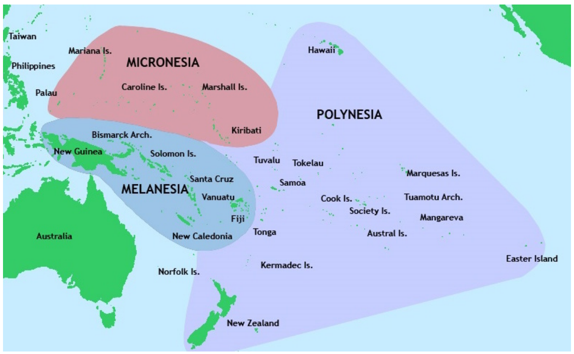
Figure 74: Europe partition

At last, a partition was detected in the table, 8355. Checking if those airports are geographically related, it has been determined that those are islands between

Polynesia, Micronesia and Melanesia. that

8353	"New Caledonia"	10
8355	"Vanuatu"	23
8355	"Solomon Islands"	17
8355	"Australia"	9
8355	"Fiji"	8
8355	"Marshall Islands"	1
8355	"Tuvalu"	1
8355	"Nauru"	1
8355	"Kiribati"	1

(a) Partition table



(b) Geographical location

Figure 75: Australasia partition

4.3.6 Possible queries on SQL

The previous section showed operations that cannot be done with SQL. Now we will present operations applicable to both;

1. **Finding flights between two airports that have no direct route between them:**

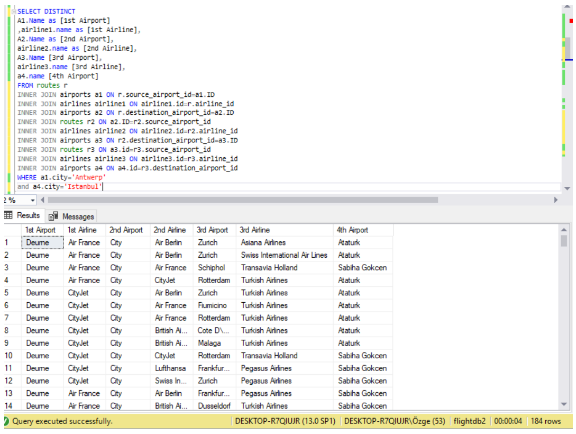
neo4j

```
MATCH
p=allShortestPaths((ap1:Airport
{city:'Antwerp'})-[*]->(ap2:Airport
{city:'Istanbul'}))
WITH extract(node in
nodes(p)|node.name) as
cities,
extract(rel in
relationships(p)|rel.airline)as
airlines
RETURN cities,airlines
```

Microsoft SQL Server

```
select distinct A1.Name as
[1st Airport]
,airline1.name as [1st
Airline],
A2.Name as [2nd Airport],
airline2.name as [2nd
Airline],
A3.Name [3rd Airport],
airline3.name [3rd Airline],
a4.name [4th Airport]
FROM routes r INNER JOIN
airports a1
ON r.source_airport_id=a1.ID
INNER JOIN airlines airline1
ON airline1.id=r.airline_id
INNER JOIN airports a2
ON
r.destination_airport_id=a2.ID
INNER JOIN routes r2
on a2.ID=r2.source_airport_id
INNER JOIN airlines airline2
on airline2.id=r2.airline_id
INNER JOIN airports a3
ON
r2.destination_airport_id=a3.ID
INNER JOIN routes r3
on a3.id=r3.source_airport_id
INNER JOIN airlines airline3
on airline3.id=r3.airline_id
INNER JOIN airports a4
on
a4.id=r3.destination_airport_id
WHERE a1.city='Antwerp' and
a4.city='Istanbul'
```

(a) Neo4j Result



(b) SQL Result

Figure 76: Comparison of Queries - first query

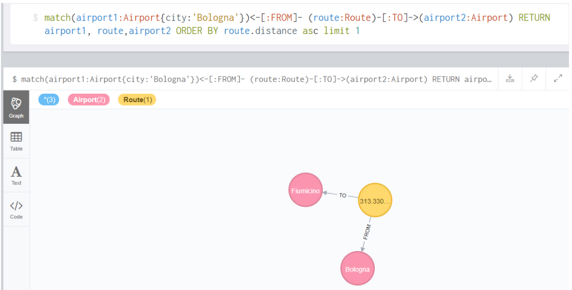
As it can be seen from here finding all possible routes between two airports is easy in Neo4j. Besides that Neo4j gives visualization.

There is one important point here; In SQL we have to specify level of depth to find results. For example in this query we searched 3-level flights between Antwerp and Istanbul. If we searched 1 or 2 level then the query would have returned no result. But in Neo4j we don't have to specify level, it finds all routes between two airports and even calculates the shortest route. Therefore this is one of the drawbacks of using SQL in data that has levels.

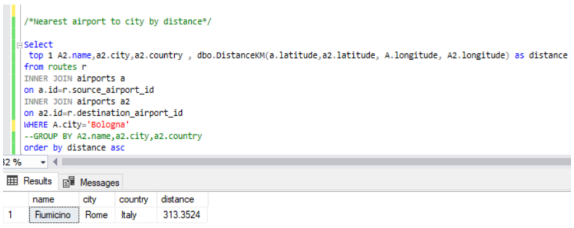
2. **Nearest airport to city by distance**

```
match(airport1:Airport{city:'Bologna'}
)<-[:FROM]- (route:Route)
-[:TO]->(airport2:Airport)
RETURN airport1,
route,airport2
ORDER BY route.distance
asc limit 1
```

```
Select
top 1
A2.name,a2.city,a2.country
,dbo.DistanceKM(a.latitude,a2.latitude,
A.longitude, A2.longitude) as
distance
from routes r
INNER JOIN airports a
on a.id=r.source_airport_id
INNER JOIN airports a2
on
a2.id=r.destination_airport_id
WHERE A.city='Bologna'
order by distance asc
```



(a) Neo4j Query



(b) SQL query

Figure 77: Comparison of queries - second query

While we were uploading our data into Neo4j we created a node called route and this node has three relationships; TO, FROM, OF and as a descriptive property we assigned calculated distance property into route node. To be in the same page we created a function in SQL that calculates distances between airports given latitude and longitude attributes of airports which already exists in our data. Both approaches give the same result but Neo4j also provides visualization.

3. **Most connected airports**

neo4j

```
MATCH
(airport:Airport)<-[:FROM]-(r:Route)
WITH airport, count(r) as
departures
MATCH
(r2:Route)-[:TO]->(airport)
RETURN airport.name as
airport_name, departures
, count(r2) as arrivals
order by
departures+arrivals desc
```

SQL Server

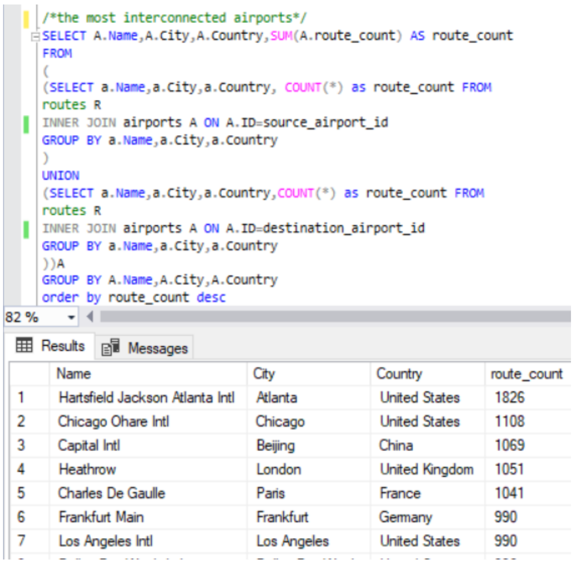
```
SELECT
A.Name,A.City,A.Country,SUM(A.route_count)
AS route_count
FROM(
SELECT
a.Name,a.City,a.Country,
COUNT(*) as route_count FROM
routes R
INNER JOIN airports A ON
A.ID=source_airport_id
GROUP BY
a.Name,a.City,a.Country
)
UNION(
SELECT
a.Name,a.City,a.Country,COUNT(*)
as route_count FROM
routes R
INNER JOIN airports A ON
A.ID=destination_airport_id
GROUP BY
a.Name,a.City,a.Country ))A
GROUP BY
A.Name,A.City,A.Country ORDER
BY route_count desc
```

```
$ MATCH (airport:Airport)<-[:FROM]-(r:Route) WITH airport, count(r) as departures MATCH (r2:Route)-[:TO]->(airport) RETURN…
```

airport_name	departures	arrivals
"Hartsfield Jackson Atlanta Intl"	915	911
"Chicago Ohare Intl"	558	550
"Capital Intl"	527	526
"Heathrow"	527	524
"Charles De Gaulle"	524	517
"Los Angeles Intl"	492	498
"Frankfurt Main"	497	493
"Dallas Fort Worth Intl"	469	467
"John F Kennedy Intl"	456	455
"Schiphol"	453	450
"Pudong"	409	412
"Changi Intl"	408	412
"Barcelona"	391	392
"Incheon Intl"	370	370
"Denver Intl"	361	374
"Miami Intl"	368	366
"Franz Josef Strauss"	368	360

Started streaming 4652 records after 988 ms and completed after 999 ms, displaying first 1000 rows.

(a) Neo4j Query



```sql
/*the most interconnected airports*/
SELECT A.Name,A.City,A.Country,SUM(A.route_count) AS route_count
FROM
(
(SELECT a.Name,a.City,a.Country, COUNT(*) as route_count FROM
routes R
INNER JOIN airports A ON A.ID=source_airport_id
GROUP BY a.Name,a.City,a.Country
)
UNION
(SELECT a.Name,a.City,a.Country,COUNT(*) as route_count FROM
routes R
INNER JOIN airports A ON A.ID=destination_airport_id
GROUP BY a.Name,a.City,a.Country
))A
GROUP BY A.Name,A.City,A.Country
order by route_count desc
```

	Name	City	Country	route_count
1	Hartsfield Jackson Atlanta Intl	Atlanta	United States	1826
2	Chicago Ohare Intl	Chicago	United States	1108
3	Capital Intl	Beijing	China	1069
4	Heathrow	London	United Kingdom	1051
5	Charles De Gaulle	Paris	France	1041
6	Frankfurt Main	Frankfurt	Germany	990
7	Los Angeles Intl	Los Angeles	United States	990

(b) SQL query

Figure 78: Comparison of queries - third query

With these queries we found the most interconnected airport by counting number of incoming and outcoming flights. As it seems it is very easy to write in Neo4j.

5 Conclusion

In conclusion, graph databases are necessary for a very concrete data sets: huge amounts of data of high complexity, where entities are very related to one another. That is because, they efficiently query through the relationships among entities, in contrast to relational databases.

Graph databases support algorithms to perform concrete queries that are out of reach to relational databases, for their tabular structure and static schema. Also, the bigger the volume of data, the slower the queries would be in SQL, because they would require to lookup joined tables with a great number of tuples. Graph databases allow to traverse through the graph and reach a high level of depth, without having to read all the data stored.

Neo4j is, by far, the leading technology of graph databases. It analyze and traverse of all data in real time and gives the results very fast. It has great user interface and support. But the greatest feature of it is; even data size grow exponentially, performance of Neo4j does not affected by it.

In our hands on research, we have stored a graph database about flight routes in Neo4j. The same data has been stored in a SQL Server database, in order to proof that some queries are more efficient in Neo4j, and some are even not possible to execute in SQL. We have queried the Shortest Path, PageRank, Betweenness and Closeness Centrality, and Partition for Community Detection.

For that, Neo4j offers algorithms easy to implement, and the results are the values expected. To evaluate its execution, the pipeline of the execution of the queries is shown. In contrast, the queries that SQL manages to perform, require complex code, and some queries, like the shortest path, are impossible to replicate.

Bibliography

[1] Tareq Abedrabbo Dominic Fox Jonas Partner Aleksa Vukotic, Nicki Watt. *Neo4j in Action*. Manning Publications, 2015.

[2] Stephan C. Carlson. Graph theory. encyclopædia britannica. *Available at* `https://www.britannica.com/topic/graph-theory`, May 2013. Accessed: 2017-11-30.

[3] DB-Engines. Knowledge base of relational and nosql database management systems. *Available at* `https://db-engines.com/en/`, 2017. Accessed: 2017-10-20.

[4] Martinez-Bazan N. Muntes-Mulero V. Baleta P. Larriba-Pay J.L. Dominguez-Sal, D. A discussion on the design of graph database benchmarks. September 2010.

[5] Stefan Edlich. Nosql archive. *Available at* `http://nosql-database.org/`. Accessed: 2017-11-20.

[6] William Lyon. Analyzing the graph of thrones. *Available at* `http://www.lyonwj.com/2016/06/26/graph-of-thrones-neo4j-social-network-analysis//`, June 2016. Accessed: 2017-12-3.

[7] Mathigon. Graphs and networks. Accessed: 2017-11-30.

[8] Thomas Vial Michel Domenjoud. Graph databases: an overview. OctoTalks, July 2012. Accessed: 2017-11-30.

[9] Neo4j. Intro to cypher.

[10] Neo4j. Top ten reasons for choosing neo4j. *Available at* `https://neo4j.com/top-ten-reasons/`.

[11] Neo4j. Neo4j graph algorithms. Github, October 2017. Accessed: 2017-12-8.

[12] University of Colorado. Database management essentials. *Available at* `https://www.youtube.com/playlist?list=PL73oFZbnYuixa9w-dL-EsM7Vy5BQGBIeO`. Accessed: 2017-10-21.

[13] OpenFlights.org. Airport, airline and route data. *Available at* `https://openflights.org/data.html`. Accessed: 2017-11-3.

[14] Tutorials Point. Graph theory: Introduction. *Available at* *https://www.tutorialspoint.com/graph_theory/graph_theory_introduction.htm*. Accessed: 2017-11-30.

[15] Tutorials Point. Neo4j - overview. *Available at* *https://www.tutorialspoint.com/neo4j/neo4j_overview.htm*. Accessed: 2017-11-30.

[16] Bryce Merkl Sasaki. Graph databases for beginners: Acid vs. base explained. *Available at* *https://neo4j.com/blog/acid-vs-base-consistency-models-explained/*, September 2015. Accessed: 2017-11-20.

[17] James Serra. Relational databases vs non-relational databases. Big Data and Data Warehousing. James Serra's Blog, August 2015. Accessed: 2017-11-29.

[18] James Serra. Types of nosql databases. Big Data and Data Warehousing. James Serra's Blog, April 2015. Accessed: 2017-11-29.

[19] Roopendra Vishwakarma. The different types of nosql databases. Open Source For U, May 2017. Accessed: 2017-11-29.