Université Libre de Bruxelles.
École Polytechnique.

Erasmus Mundus Master's Programme in Information Technologies for Business Intelligence (IT4BI).

Course: Advanced Databases (INFO-H-415).

Lecturer: Esteban Zimányi.
Teaching Assistant: Stefan Eppe.

Neo4j: A Graph Database

Authors:
Maximiliano Ariel López
Miruna Mironescu

First Semester – 2013-2014

Contents

Neo4j: A Graph Database

Introduction

The purpose of this project is to investigate Graph Databases, summarise their key concepts and share a high-level and objective picture about them by means of this report. The ultimate goal is to bring about awareness, reflection and debate about this topic among our colleagues.

After going through some background theoretical information, we will illustrate the technical side of this kind of databases by providing a concrete practical example. In particular, we will work with Neo4j, which is an open-source implementation written in Java.

Beyond any shadow of a doubt, the real world is rich and interrelated. Companies developing Graph Databases take pride in the fact that their products can manage and query highly connected data. They claim that graphs are really useful when it comes to modelling a broad range of business domains, ranging from chain-supply markets to human interconnections –among many others–, thus covering numerous fields (science, government, business, etc.).

That being said, we have started our investigation under the hypothesis that Graph Databases outperform other technologies when relationships are a key component of the model. We would like to test the aforesaid reasoning on the basis of our findings.

Background Information

What is a Graph?

Informally, a graph may be defined as a collection of vertices and edges. In a more formal manner, it can be described as a set of *nodes* and *relationships* that are interconnected. Entities are referred to as nodes. The ways in which nodes relate to each other define relationships.

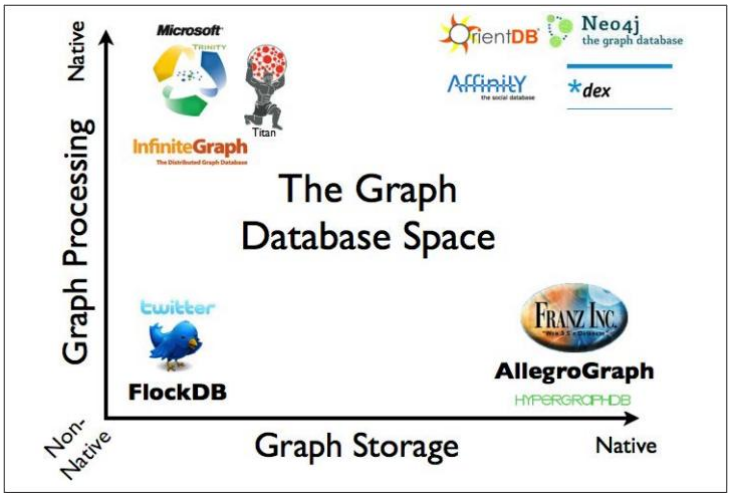
Some Applications of Graph Databases

As it was noted in the previous paragraphs, graphs are particularly ubiquitous in the real world. Both abstract and material objects can be related with each other. For instance, as far as the world of business is concerned, a report published by Gartner states that there are five graphs that need to be leveraged so as to obtain a sustainable competitive advantage: the social graph, the intent graph, the consumption graph, the interest graph and the mobile graph.[1]

Another very good example of data that can be represented via graphs is Twitter's data. Entities (represented by circles) are users, who are connected by "Follows" relationships. Let us not forget that Twitter's real graph is composed of millions of interconnected users.

[1] Ray Valdes (2012). The Competitive Dynamics of the Consumer Web: Five Graphs Deliver a Sustainable Advantage. Gartner Group. Stamford, United States.

The following chart[2] summarises some representative implementations and applications of Graph Databases:



Getting into Graph Databases[3]

A *Graph Database* is an online database management system with Create, Read, Update and Delete (CRUD) methods that expose a graph data model. This kind of database is built to be used with OLTP systems so they are optimised with respect to transactional integrity and availability.

Graph compute engines are a key component of Graph Databases. They are designed to identify clusters in data, or answer questions such as "how many relationships, on average, does everyone in a social network have?". In order to perform these global calculations, the processing of large datasets of information in batch needs to be highly optimized. From this point of view, they are similar to other batch analysis technology –such as data mining or OLAP–, that are found in other databases.

Beyond any shadow of a doubt, there must be a sufficiently good reason to replace a well-established and well-understood data platform (i.e. relational databases). As far as graph databases are concerned, this motivation comes from an important set of use cases and data patterns whose performance improves by one or two orders of magnitude and whose latency is much lower compared to batch processing of aggregates. To sum it up, a performance benefit can be easily seen. Besides this, the technology provides a mode of delivery which is more aligned with agile software delivery practices.

Graph Databases easily deal with connected data and they perform better than relational databases and NOSQL stores. Unlike relational data, where join-intensive queries decrease performance (especially if we are dealing with large data sets) the graph database performance tends to remain relatively constant, despite growing amounts of data. This is due to the fact that queries are restricted to a segment of the graph. The immediate result is that the execution time for each query is only proportional to the size of the graph segment concerned by the query and not to the total size of it.

[2] Ian Robinson, Jim Webber and Emil Eifrem (2013). *Graph Databases*. p. 7. O'Reilly Media Inc., California.
[3] Ibidem. p. 5-9.

Data models need to evolve at the same time as the rest of the application. The technology must be aligned with today's software delivery practices, which are iterative and incremental. There are ways to perform system maintenance, updates and deployments on graph databases with ease and security. We must also remind users of the free-schema nature of the graph data model. The Application Programming Interface (API) and the query language allow users to evolve an application in a controlled manner.

Graph Databases allow schema and structure to come into view together with our growing understanding of the problem space, so they are not forced to be imposed upfront (especially if the data is intricate or the shape of it is rather unknown).

Graphs are additive by nature. In any moment in time we can add new nodes and relationships to our graph model. They are added to an existing structure without disturbing current functional queries or application functionality. Hence, we have positive effect in the developer's performance and in the project risk. Due to graph flexibility, the model does not need to be listed in detail well in advance (which may become problematic when dealing with changing internal conditions like business requirements). This also means that there are few migrations performed. In consequence, a developer deals with less maintenance overhead issues and the risk is reduced.

Comparison with Other Kinds of Databases[4]

Relational databases struggle to make connections between the ad hoc or exceptional relationships that depict the real world. The relationships within a relational database are more likely used in order to make links between tables. Also, relational databases do not disambiguate the semantics of relationships that connect entities, nor do they qualify their weights and strengths. It must also be mentioned that, the relational model becomes burden with large table joins if the dataset becomes more complex and less uniform.

Another problem may also lie in the fact that the relational databases are full of constraints (e.g. null check constraints) thus again burdening large table joins. Sparse tables still undergo check constraints, despite the presence of a schema, before the joins are created. Notice a large amount of wasted space.

The increase in connectedness is reflected into the relational world into a larger number of joins, which may impede performance and might make it difficult for developers to evolve an existing database in response to changing business needs.

Foreign keys were constructed solely with the purpose of making the database work thus adding an additional complexity, when compared to the graph database. Several expensive joins may be needed in order to find the result of a query, hence bringing about high costs. The reciprocal queries are also expensive.

Besides, indexes may also be considered as an alternative, but they do not handle recursive queries that well (e.g. "Which customer bought this product who also bought that product?").

In contrast, most NOSQL databases –document-orientated, column-orientated or key-value oriented– store sets of disconnected documents/values/columns, which make it difficult to

[4] Ibidem 1. p. 10-14.

connect data. A well-known strategy in order to add relationships to such store is to embed an aggregate's identifier inside the field belonging to another aggregate (effectively including foreign keys). But this requires joining aggregates at the application level, which soon becomes particularly expensive.

It is tempting to think that aggregate stores are functionality as good as the graph databases with respect to connected data. Yet, it is not true, because aggregate stores do not maintain consistency of connected data. They neither support what is known as index-free adjacency, whereby elements contain direct links to their neighbours. For connected data problems, as a result, aggregate stores must use inherently latent methods in order to create and query relationships outside the data model.

While previous examples have dealt with implicitly connected data, a user may infer semantic dependencies between entities, but the data models (as well as the databases themselves) are blind to these connections. In order to compensate, our applications has to create a network on the basis of a flat, disconnected data and afterwards deal with any latent writes or slow queries across a denormalized store that arises.

The goal is to obtain a cohesive picture of the whole along with the connection between elements. In contrast to what we have seen so far, connected data is stored as connected data. Whenever there are connections on the domain, there are connections in the data.

A social network is a popular example of a densely connected semi-structured network. The connections (friend, colleague, family, etc.) between the entities (people) do not exhibit uniformity across the domain –the domain is semi-structured–. This example is just one example of data that cannot be captured by a one-size-fits-all schema or conveniently split across disconnected aggregates. The network is very complex; it has grown in size (6 degrees of friends away) and it is very rich in expressiveness. The flexibility of the model allows adding new nodes and new relationships without compromising the existing network or migrating data because the original data and its intent remain the same.

The graph renders a rich picture of the network. Relationships can be easily deduced (see practical section for examples of graphs) as well as the entities that take part in them. Relationships in graphs naturally form paths. Querying the graphs or analysing them in any way implies following paths. The graph model is therefore a fundamentally path-orientated one. The majority of the path-based graphs database operations are highly aligned with the way in which the data is laid out, making them extremely efficient. A study conducted by Partnet and Vukotic, whose results are displayed in *Neo4j in Action* book, states that the graph database is substantially quicker for connected data than a relational store.

Their experiment is based precisely on the social network similar to the one written above. The aforementioned authors seek to find friends-of-friends in a social network, to a maximum depth of 5. The idea is to find a path that connects any randomly chosen 2 persons, a path that is at most 5 persons long. If the social network comprises one million people each having approximately 50 friends, the results strongly suggests that graph databases are the best choice for connected data.

If we were to consider a length of 2 persons between randomly chosen people, then both the relational model and the graph model are good candidates. Despite the fact that Neo4j runs in

2/3 the time of a relational one, an end user would barely see the difference in milliseconds between the two approaches. If the length is increased to 3 persons, then the difference is noticeable. The relational model would need approximately 30 seconds to complete, unacceptable for an online system. At variance with that, Neo4j needs approximately one sixth of a second, which is clearly suitable for an online system.

The depth 4 is problematic, since the database exhibits crippling latency, making it almost useless for an online system (1543 seconds). Neo4j still renders results that are good enough for an online system (1.35 seconds). At depth 5 the relational database is overwhelmed taking it too long to complete (the query was stopped before completion). Neo4j, nevertheless, returns a result in 2 seconds.

The social network is an example to illustrate how different technologies deal with connected data, but can we always use this? In real life, we do not need to compute such remote friends. Nonetheless, we can substitute the social network with any other domain and the results will be similar in terms of time, performance and modelling, among other aspects. Bear in mind the fact that graphs are additive, and you can think about joining other graphs to your already created graph in order to establish connection to other domains.

Querying graphs: Cypher Query Language[5]

Cypher is a database expressive and compact query language. It is primarily used in Neo4j, although it can also be used to programmatically describe graphs in a precise manner due to its close affinity to graphs. It is easy to learn and understand since it follows the way humans intuitively describe graphs using diagrams. In addition, it can be used as a starting point to learn other graph query languages (e.g. SPARQL, Gremlin, etc.).

Cypher makes it possible for the user (or the application running on the user's behalf) to query the database in order to find specific patterns. The items we wish to find can easily be drawn by hand as they are graphs. For example, if we wish to find a person "a" that knows a person "b" that knows a person "c", we can easily identity the nodes of the graphs as being "a"," b", "c" and the relationship described by "knows".

The query is as follows: (a)-[: knows] -> (b)-[: knows] -> (c)

This pattern describes a path which connects "a" to "b" and "b" to "c". As can be seen, Cypher patterns follow very natural from the way humans draw graphs on the whiteboard. Like most common query languages, Cypher is composed of clauses. A very simple query consists of a START followed by a MATCH and a RETURN clause.

START – specifies one or more starting points – nodes or relationships – in a graph, which are obtained via index lookup (starting points are rarely accessed via IDs).

MATCH – it makes use of the relationships

RETURN – returns nodes and relationships that match the criteria

WHERE – acts as a filter pattern for matching results

[5] Ibidem 1. p. 27-30.

CREATE or CREATE UNIQUE – creates (unique) nodes and relationships

DELETE – removes nodes, relationships or properties

SET – sets property values

UNION – merges results from two or more queries

WITH – chains subsequent query results and pipelines results

Query example: START a = node: user (name= 'Michael')
MATCH (a)-[: knows] -> (b)-[: knows] -> (c)
RETURN b, c

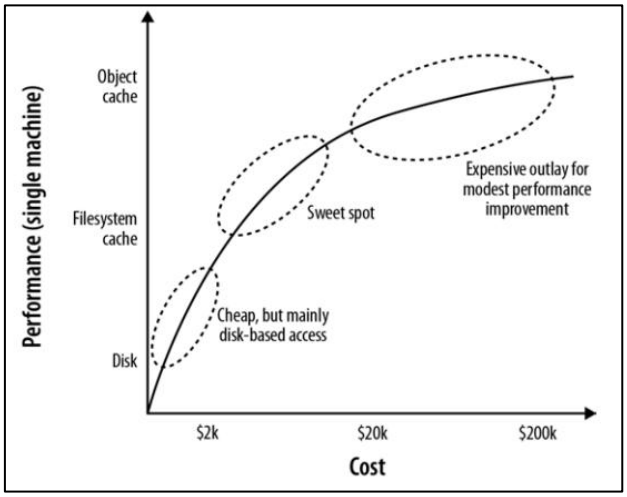
First of all, the starting locations –the anchor points in the real graph to which some parts of the pattern are bound to– can be identified in 2 ways, but one way (namely the index) is the most commonly used. The START clause specifies exactly the starting location using indexes. Here, the index is *user* and it is looking for a node with the property *name* whose value is *Michael.* The return value is bound to the identifier which we use throughout the query. The path described in the MATCH clause comprises 3 nodes, one is bound to the identifier *a.* Therefore, our query is anchored to a specific point in the graph, the real node in the graph named *Michael.* The query looks for all possible matches of *b,c* in the graph which it will finally return in the WHERE clause.

Performance[6]

There are three main methods to improve performance in Graph Databases:

- Increase the percentage of store that is mapped into the file system cache.
- Increase the object cache (the cache has been known to significantly increase over the past years, from 2 GB to 200 GB).

As far as cache is concerned, the user can easily conclude that more RAM is needed. It is always a cost versus performance trade-off. The first criterion to check is whether there is a one-on-one mapping between the size of the store files in the file system cache and the size of the disk. Graph objects in the object cache may be up to 10 times larger than their own on-disk representation. Therefore, we can conclude that allocating RAM to the object cache is far more expensive per graph element than allocating it to the file system cache.



The second criterion is related to the location of the object cache. For example, if the graph database uses an on-heap cache (e.g. Neo4j) it means that increasing the size of the cache requires allocating more heaps. The user must bear in mind the fact that most modern Java

[6] Ibidem 1. p. 94-96.

Virtual Machines do not cope well with heaps larger than 8 GB. Garbage collection may impact the performance of our application if that size is surpassed.

The sweet spot for any performance versus cost trade-off lies around the point where the entire store files can be mapped to RAM while maintaining a moderate size object cache. In many cases, heaps of 4 or 8 GB are used although a smaller heap can actually improve performance by mitigating expensive garbage collection impact.

It is advisable to know the size of the graph beforehand in order to determine how much RAM needs to be allocated to the heap. Therefore, some developers chose to build a representative dataset early in their application so they will gain insight to the resource requirements (e.g. RAM). If the graph cannot fit entirely in memory, the developer might consider cache sharing.

When optimizing a graph database solution for performance, there are a few guidelines that must be considered: 1) the file system cache should be used as much as possible and (if possible) we should entirely map our store files into the cache 2) the Java Virtual Machine heap should be tuned 3) the user might want to consider using fast disks (e.g. enterprise flash disks or SSDs).

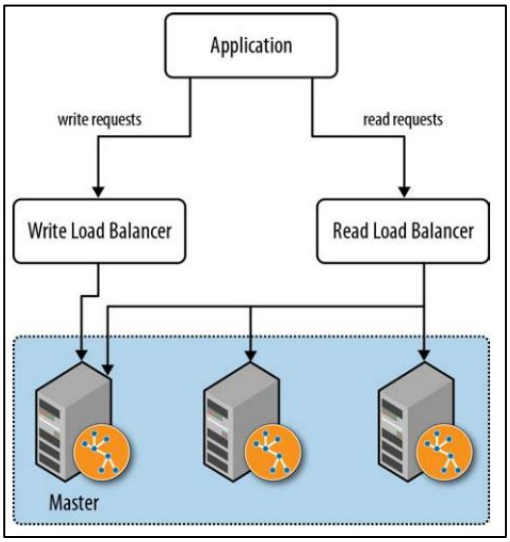
Clustering and Load Balancing[7]

Neo4j clusters for high availability and horizontal read scaling using master-slave replication. Even though it would be possible to perform write operations through slaves, the slave being written would need to synchronise the new information with the master in a synchronous way (i.e. before returning to the client).

In high write load scenarios, writes may be buffered by using queues. This regulates traffic and reduces contention at the same time that it allows maintenance without refusing client requests.

It is also possible to install multi-region clusters in multiple data centres and on cloud platforms such as Amazon Web Services. Therefore, client requests can be serviced by the portion of the cluster that is geographically closer. This may sometimes introduce latency and disrupt the coordination protocol but it is possible to counteract this effect by configuring slave-only databases (e.g. disabling the master re-election process).

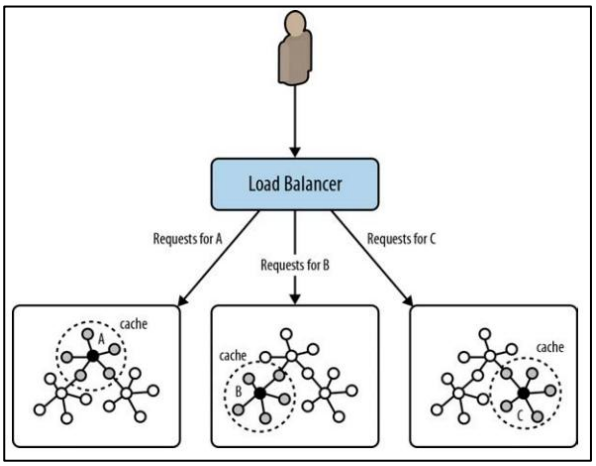
As far as Load Balancing is concerned, we should consider that Neo4j does not include a native load balancer so it relies on the load-balancing capabilities of the network infrastructure so as to help maximise throughput and reduce latency.



As writes need to pass through the master instance, load balancers need to be configured to direct write traffic to the master, whereas read operations can be balanced across the entire cluster. In a web-based application, the HTTP method is often sufficient to distinguish writes (POST, PUT or DELETE) from reads (GET).

[7] Ibidem 1. p. 78-82.

When running in server mode, Neo4j either exposes a Uniform Resource Identifier (URI) that indicates whether that instance is currently the master or determines what the master is. Load balancers can poll this URI at intervals of time to determine where to route traffic.



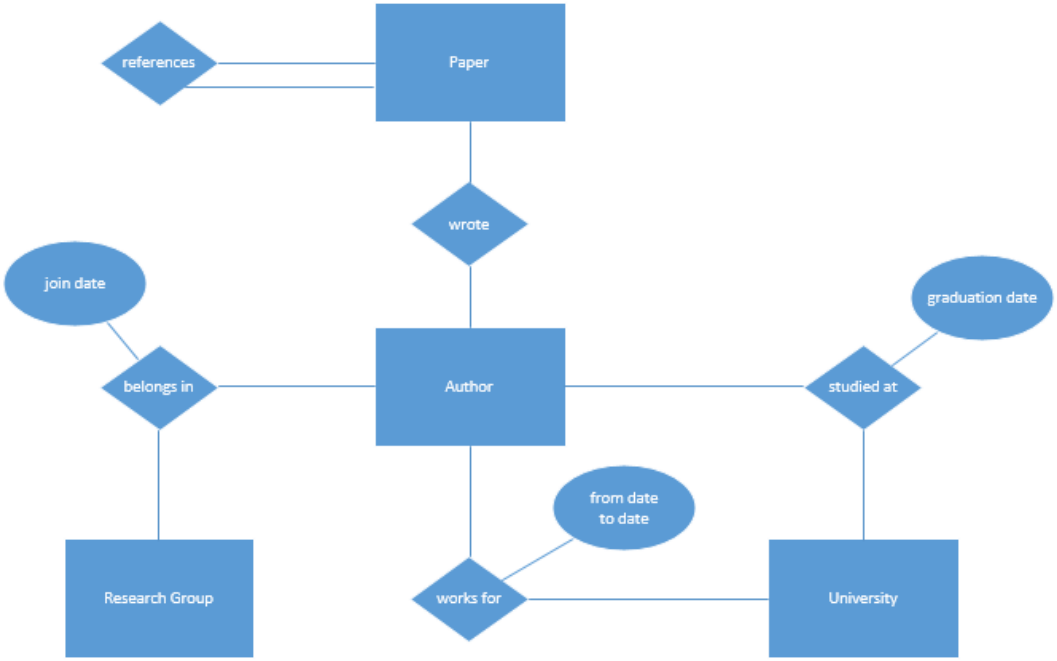
From a cache perspective, queries will naturally run faster when the portions of the graph needed to satisfy them reside in the filesystem cache or in the object cache, namely when they are stored in main memory. Cache sharding consists of routing each request to a database instance in a cluster where the portion of the graph necessary to satisfy that request is likely already in main memory. If most queries are graph-local queries, meaning they start from one or more specific points in the graph, and traverse the surrounding subgraphs, then a mechanism that consistently routes queries beginning from the same set of start points to the same database instance will increase the likelihood of each query hitting a warm cache. The strategy used to implement consistent routing will vary according to domain.

A Practical Example

Conceptual Model

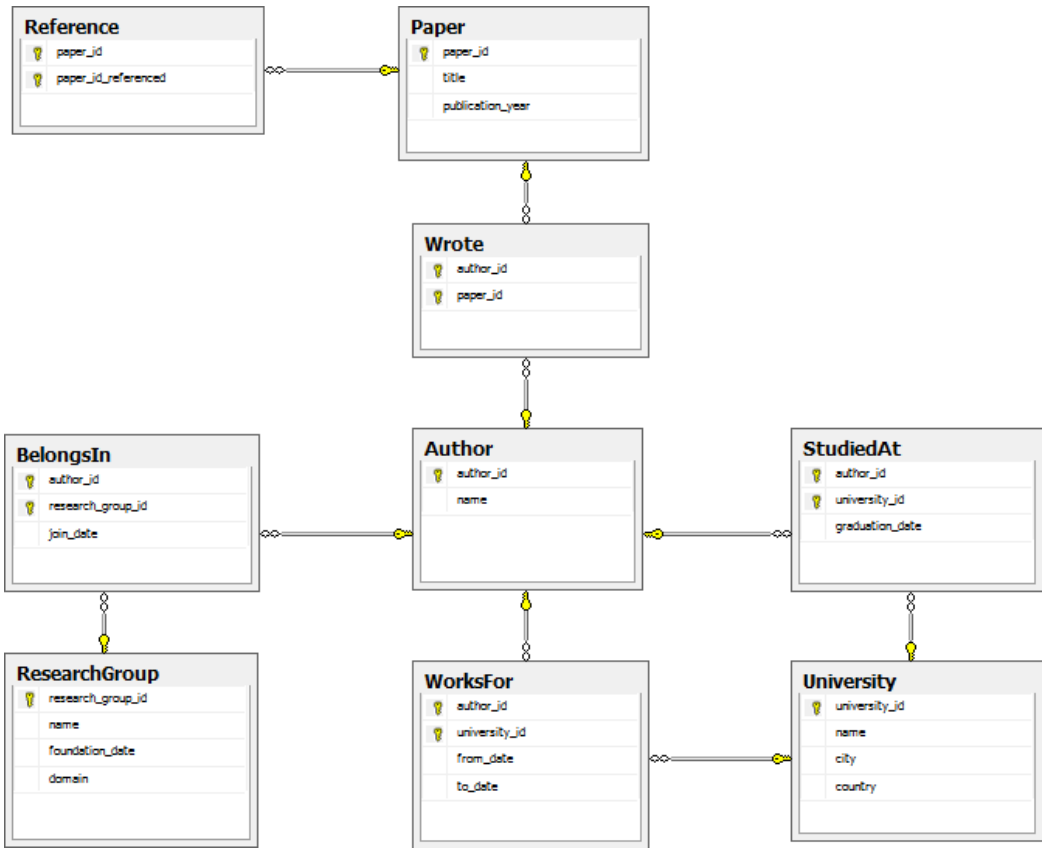
Broadly speaking, we will work on a database that models some features of the world of academic writing.

We will therefore collect information about papers, their authors (including their affiliation to universities and research groups) along with the references among the different papers, as the following diagram depicts:



Relational Logical Model

On account of the fact that we would like to compare *query expressivity power* between Graph Databases and the traditional Relational Databases, we have also implemented the conceptual model described above in Microsoft SQL Server. The underlying logical model is depicted in the following diagram:



Our Test Dataset

To build our example database, we extracted from DBLP[8] a sample of ***25.000 papers*** representing more than ***18.000 authors***. We also manually created ***5 universities*** and ***10 research groups***.

Afterwards, relationships were randomly created according to the cardinalities described as follows:

- An author belongs in 1 or 2 research groups (representing about **27.000 "belongs in"** relationships).
- An author studied at 1 or 2 universities (representing about **27.000 "studied at"** relationships).
- An author works at most in 2 universities (representing about **18.000 "works for"** relationships).

[8] *"The DBLP Computer Science Bibliography"*. University of Trier, Germany. Version retrieved on 01/12/2013 from http://dblp.uni-trier.de/xml/dblp.xml.

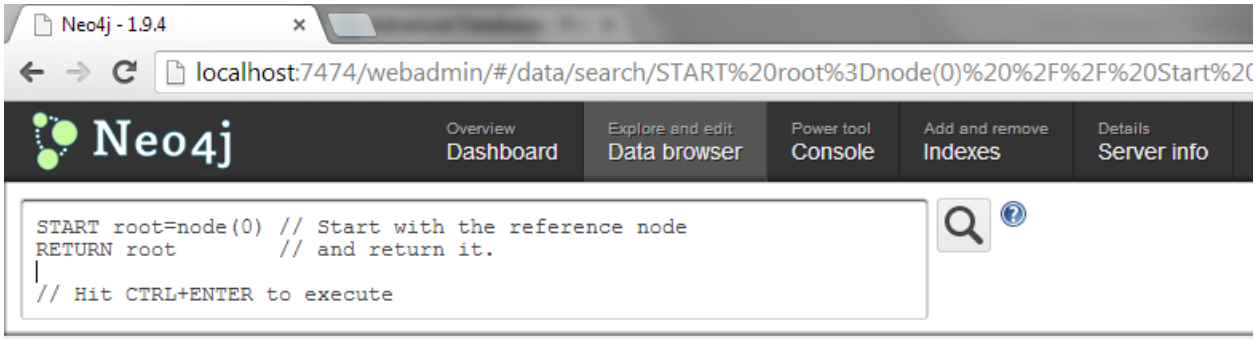
- A paper references 10 to 30 other papers (representing about **500.000 "references"** relationships).

In order to import this information into the database, we programmed a custom batch import application in Java. Even though we initially analysed the chance of reusing the generic Batch Import Program developed by Max de Marzi[9], we eventually decided to create our own version in order to allow random relationships creation. In addition, our custom code does not only take care of creating nodes and relationships in Neo4j database but it also inserts the relevant records to a relational database in Microsoft SQL Server for comparison purposes.

Some code excerpts will be illustrated in ***Code samples*** section of this document but the full source code can be downloaded from https://code.google.com/p/neo4j-batch-insert-example. It uses **Neo4j**[10] and **Log4j**[11] JAR libraries, as well as **Microsoft SQL Server JDBC driver**[12].

Running Queries

In order to run queries, we need to open a web browser and go to http://localhost:7474/webadmin/. Then, the "Data browser" tab allows us to write Cypher queries and see results:



Query examples

We have written 7 different queries in Cypher and SQL in order to compare the expressivity power of both platforms under different scenarios. Queries and results are shown as follows:

1) Count the number of authors that studied at each university:

Neo4j
the graph database

```
START u=node(*)
MATCH (u)<-[:STUDIED_AT]-(a)
RETURN u.name, count(*)
```

Microsoft SQL Server

```
SELECT u.name, count(*)
FROM University u, StudiedAt s
WHERE u.university_id = s.university_id
GROUP BY u.name
```

[9] "Batch Importer – Part 1". Max de Marzi. Retrieved from http://maxdemarzi.com/2012/02/28/batch-importer-part-1 on October 23, 2013.
[10] Jar files included in "lib" folder of file "neo4j-community-2.0.0-windows.zip", downloaded from http://www.neo4j.org/download.
[11] Jar file "log4j-1.2.17.jar", downloaded from http://logging.apache.org/log4j/1.2/download.html.
[12] Jar file "sqljdbc4.jar", downloaded from http://msdn.microsoft.com/en-us/sqlserver/aa937724.aspx.

Returned **5 rows.** Query took **3200ms**

u.name	count(*)
"Universidad de Buenos Aires"	5507
"Technische Universität Berlin"	5339
"Ecole Centrale Paris"	5384
"Université Libre de Bruxelles"	5395
"Politehnica University Bucharest"	5531

	name	(No colu
1	Ecole Centrale Paris	5384
2	Politehnica University Bucharest	5531
3	Technische Universität Berlin	5339
4	Universidad de Buenos Aires	5507
5	Université Libre de Bruxelles	5395

2) Get the name of authors that studied at “Université Libre de Bruxelles”:

Neo4j the graph database

```
START u=node:universities(name="Université Libre de Bruxelles")
MATCH (u)<-[:STUDIED_AT]-(a)
RETURN a.name
ORDER BY a.name
```

Microsoft SQL Server

```
SELECT a.name
FROM University u, StudiedAt s, Author a
WHERE u.university_id = s.university_id
AND s.author_id = a.author_id
AND u.name = 'Université Libre de Bruxelles'
ORDER BY a.name
```

Returned **5395 rows.** Query took **4102ms**

a.name
"A. A. Majid"
"A. A. Rodon"
"A. A. Zaidee"
"A. Akbarzadeh"
"A. Atarashi"

	name
1	A. A. Majid
2	A. A. Rodon
3	A. A. Zaidee
4	A. Akbarzadeh
5	A. Atarashi
6	A. B. M. Shawkat Ali
7	A. Chris Long
8	A. Doi

adb_project | 00:00:00 | 5395 rows

3) Count how many papers were written by each author working at “Politehnica University Bucharest”:

Neo4j the graph database

```
START u=node:universities(name="Politehnica University Bucharest")
MATCH (u)<-[:WORKS_FOR]-(a)-[:WROTE]->(p)
RETURN a.name, count(p.title)
ORDER BY a.name
```

Microsoft SQL Server

```
SELECT a.name, count(*)
FROM University u, WorksFor w,
        Author a, Wrote wr
WHERE u.university_id = w.university_id
AND w.author_id = a.author_id
AND u.name = 'Politehnica University Bucharest'
AND wr.author_id = a.author_id
GROUP BY a.name
ORDER BY a.name
```

Returned **3622 rows.** Query took **3604ms**

a.name
"A. A. Azeezunnisa"
"A. Biavasco"
"A. Gomes"

	name	(No col
5	A. H. Wadaa	1
6	A. J. Mendes	1
7	A. Paviotti	1
8	A. Saremi	1
9	A. Taleb-Bendiab	2
10	A. V. Nath	1
11	A. Wayne Madison	1

adb_project | 00:00:00 | 3622 rows

4) Get the number of references to papers written by someone who studied at ULB grouped by the year of publication of the article that references the paper:

Neo4j the graph database

```
START u1=node:universities(name='Université
Libre de Bruxelles')
MATCH a1-[:STUDIED_AT]->u1,
        a1-[:WROTE]->p1,
        p2-[:REFERENCES]->p1
RETURN p2.publication_year, count(p2.title)
ORDER BY p2.publication_year
```

Returned 45 rows. Query took 7940ms

p2.publication_year	count(p2.title)
"1949"	4
"1950"	4
"1953"	3
"1963"	17
"1967"	16
"1969"	6
"1970"	9

Microsoft SQL Server

```
SELECT p.publication_year, count(*)
FROM University u, StudiedAt s, Wrote wr,
        Reference r, Paper p
WHERE u.university_id = s.university_id
AND u.name = 'Université Libre de Bruxelles'
AND s.author_id = wr.author_id
AND r.paper_id_referenced = wr.paper_id
AND p.paper_id = r.paper_id
GROUP BY p.publication_year
```

	publication_year	(No colum
1	1949	4
2	1950	4
3	1953	3
4	1963	17
5	1967	16
6	1969	6
7	1970	9

adb_project | 00:00:01 | 45 rows

5) For all papers written by people who studied at "Universidad de Buenos Aires", count the number of references to articles from authors who studied at ULB:

Neo4j the graph database

```
START u1=node:universities(name='Universidad de
Buenos Aires'),
u2=node:universities(name='Université Libre de
Bruxelles')
MATCH a1-[:STUDIED_AT]->u1,
    a2-[:STUDIED_AT]->u2,
    a1-[:WROTE]->p1,
    a2-[:WROTE]->p2,
    p1-[:REFERENCES]->p2
RETURN count(*)
```

Returned 1 row. Query took 3198ms

count(*)
43705

Microsoft SQL Server

```
SELECT count(*)
FROM University u, University u2,
    StudiedAt s, StudiedAt s2,
    Wrote wr, Wrote wr2,
    Reference r
WHERE u.university_id = s.university_id
AND u.name = 'Universidad de Buenos Aires'
AND u2.name = 'Université Libre de Bruxelles'
AND s.author_id = wr.author_id
AND wr.paper_id = r.paper_id
AND u2.university_id = s2.university_id
AND s2.author_id = wr2.author_id
AND wr2.paper_id = r.paper_id_referenced
```

	(No column name)
1	43705

6) Author David Robson has referenced other authors. Those referenced authors, in turn, have referenced third parties. The papers written by such third parties might be of interest for David Robson for further reading, so he would like to retrieve them as well as the name of their respective authors.

Neo4j the graph database

```
START a1=node:authors(name='David Robson')
MATCH a1-[:WROTE]->p1,
    a2-[:WROTE]->p2,
    a3-[:WROTE]->p3,
    p1-[:REFERENCES]->p2,
    p2-[:REFERENCES]->p3
WHERE a3.name <> a1.name
RETURN distinct p3.title, a3.name
ORDER BY p3.title
```

Microsoft SQL Server

```
SELECT distinct p3.title, a3.name
FROM Author a1, Author a3,
    Wrote w1, Wrote w3,
    Reference r1, Reference r2,
    Paper p3
WHERE a1.name = 'David Robson'
AND a1.author_id = w1.author_id
AND w1.paper_id = r1.paper_id
AND r1.paper_id_referenced = r2.paper_id
AND r2.paper_id_referenced = w3.paper_id
AND w3.author_id = a3.author_id
AND a1.author_id <> a3.author_id
AND w3.paper_id = p3.paper_id
ORDER BY p3.title
```

Returned **276 rows.** Query took **749ms**

p3.title
"3D Object Reconstruction from a Sequence of Images Using
"3D Range Optical Sensor: analysis of the Measurement Error
"3D Shape Recovery and Registration based on the Projection

	title	name
1	3D Object Re...	Elsayed E.
2	3D Range Op...	Sara Lazz
3	3D Shape Re...	Roberto R
4	A case study i...	George Go
5	A Case Study ...	Jong Sze
6	A conceptual ...	Hemant Pa
7	A Conversion ...	Weidong

adb_project | 00:00:00 | 276 rows

7) For all papers written by author Michael L. Heytens, we will obtain first and second level outgoing references:

Neo4j the graph database

```
START a1=node:authors(name='Michael L. Heytens')
MATCH a1-[:WROTE]->p1,
m=p1-[:REFERENCES*1..2]->p2
RETURN length(m), count(*)
```

Microsoft SQL Server

```
SELECT 1, COUNT(*)
FROM Author a, Wrote w, Reference r
WHERE a.author_id = w.author_id
AND w.paper_id = r.paper_id
AND a.name = 'Michael L. Heytens'
  UNION
SELECT 2, COUNT(*)
FROM Reference r
WHERE r.paper_id IN
( SELECT r.paper_id_referenced
FROM Author a, Wrote w, Reference r
WHERE a.author_id = w.author_id
AND w.paper_id = r.paper_id
AND a.name = 'Michael L. Heytens' )
```

Returned **2 rows.** Query took **702ms**

length(m)	count(*)
2	447
1	21

	(No column name)	(No column name)
1	1	21
2	2	447

Experimenting with Neo4j

Installing Neo4J[13]

Neo4j can be run on any desktop or laptop computer. This option is available not only for Windows but also for Linux and Mac.

A normal user would need a Java Virtual Machine installed on the computer before installing Neo4j (with the exception of Windows OS). In addition, Java Development Kit (JDK) is recommended.

We will further explain how to install Neo4j for each of the main operating systems.

Windows

Provided the user has the corresponding administrative rights, Neo4j can be easily installed as a windows service by following the steps described below:

1. Click on Start -> All programs -> Accessories
2. Right click on Command Prompt and chose "Run as Administrator"
3. Provide the necessary authorization and/or the Administrator password
4. Navigate to %NEO4J_HOME% (in other words, the directory where Neo4j was installed).
5. Run **bin\Neo4j.bat install**

In order to uninstall Neo4j, you need to run: **bin\Neo4j.bat remove**

Also, to query the status of the service, you need to run: **bin\Neo4j.bat status**

Run **bin\Neo4j.bat start** to start the service from the command prompt. Also, to stop the service, run: **bin\Neo4j.bat stop**

Linux

Neo4j may participate in the normal startup/shutdown process of the computer. On most popular Linux distributions, the following procedure is suggested:

1. **cd $NEO4J_HOME** (change directory to corresponding folder)
2. **sudo ./bin/neo4j install** (enter the corresponding password to gain super-user privileges)
3. **service neo4j-service status** (this command will indicate if the server is running)
4. **service neo4j-service start** (this command will start the server)

During installation, the option (to select the user Neo4j will run as) will eventually appear. A username (by default: neo4j) must be provided. If the user account does not exist, it will be created as a system account. The ***$NEO4J_HOME/data*** directory will be owned by that user.

There is a specific command to remove the server from the set of startup services. The corresponding command is: **service neo4j-service remove**

[13] *"Neo4j. The Graph Database"*. Neo Technology, Inc., San Mateo, California. Version retrieved on 01/12/2013 from http://docs.neo4j.org/chunked/stable/server-installation.html

It will also stop the server (if the server is running) and eventually remove it. Remember that, if a new user account was previously created, upon uninstall you will be asked to remove it.

Mac

Using Homebrew, the following command is required in order to install Neo4j Server:

brew install neo4j && neo4j start

This will get a Neo4j instance running on http://localhost:7474. The installation files will be found in /usr/local/Cellar/neo4j/community-{NEO4J_VERSION}/libexec/.

As a service, Neo4j can be installed as a Mac launched job by running these commands:

1. **cd $NEO4J_HOME**
2. **./bin/neo4j install**
3. **launchctl list | grep neo**
   (this will show the launched "org.neo4j.server.7474" job to run the Neo4j Server)
4. **./bin/neo4j status**
   (this will indicate whether the server is running or not)
5. **launchctl stop org.neo4j.server.7474**
   (this command should stop the server)
6. **launchctl start org.neo4j.server.7474**
   (this command should start the server again)

In order to remove the launchctl service, the user must issue the command: **./bin/neo4j remove**

Populating the Database

Code Samples

- Connecting to the Database:

```
import org.neo4j.graphdb.GraphDatabaseService;

import org.neo4j.graphdb.factory.GraphDatabaseFactory;

[…]

String filePath = "c:\\users\\user1\\documents\\neo4j\\default.graphdb";


GraphDatabaseService graphDb = new GraphDatabaseFactory()

  .newEmbeddedDatabaseBuilder(filePath).newGraphDatabase();
```

- Creating Nodes, Relationships and Indexes:

```
import org.neo4j.graphdb.Transaction;

import org.neo4j.graphdb.Node;

import org.neo4j.graphdb.Relationship;

import org.neo4j.graphdb.DynamicRelationshipType;
```

```
import org.neo4j.graphdb.index.Index;

[…]

Transaction tx = graphDb.beginTx();


Node n1 = graphDb.createNode();

n1.setProperty("name", "Université Libre de Bruxelles");

n1.setProperty("city", "Bruxelles");

n1.setProperty("country", "Belgique");

n1.setProperty("type", "university");

Node n2 = graphDb.createNode();

n2.setProperty("name", "François Englert");

n2.setProperty("type", "author");


Relationship r1 = n2.createRelationshipTo( n1,

DynamicRelationshipType.withName("STUDIED_AT") );

r1.setProperty("graduation_date", "1955-06-30");

Relationship r2 = n2.createRelationshipTo( n1,

DynamicRelationshipType.withName("WORKS_FOR") );

r2.setProperty("from_date", "1961-09-01");

r2.setProperty("to_date", "1998-06-30");


Index<Node> i1 = graphDb.index().forNodes("indexOnAuthors");

i1.add(n, "name", n.getProperty("name"));

Index<Node> i2 = graphDb.index().forNodes("indexOnUniversities");

i2.add(n, "name", n.getProperty("name"));

i2.add(n, "city", n.getProperty("city"));

i2.add(n, "country", n.getProperty("country"));

tx.success();

tx.finish();
```

Discussion

In the same vein as what happens with the transition between procedural programming and object oriented languages, developing graph databases indisputably requires a mind-set change. In fact, if graph databases were used according to the relational paradigm (e.g. modelling relations as attributes of nodes) we would not be benefiting from their advantages.

Nevertheless, provided proper modelling takes place, great gains in terms of performance may be obtained. As previously discussed, some studies –like the one performed by Partnet and Vukotic– have shown that, when multiple-level indirect relationships are involved, relational databases are not a viable solution because of their protracted runtimes. Under those circumstances, there are compelling reasons to use Graph Databases.

Query expressivity may also turn out to be much better in Graph Databases. For instance, the seventh query depicted in the examples section, clearly shows that a fairly short and intelligible Cypher query can obtain the same results that a longer and slightly more cryptic query in SQL. It might be worth mentioning that such case is only exploring a two-level relationship, whereas this advantage would be considerably magnified if the number of levels rose.

That being said, we consider that there might be some aspects that still need to be refined. For example, as far as stability or fault tolerance are concerned, we noticed that some queries can easily make the database management system unstable. The only way to recover from this scenario proved to be restarting the database service.

Furthermore, let us consider a query with a START clause that does not use indexes. In a relational database, a lack of indexes brings about a full table scan. However, in a graph database, there would be a full scan of the entire database because nodes are not grouped by entity type. Ideally, the database management system should ensure stability by foreseeing and successfully dealing with scenarios like these by offering, for instance, an option to cancel long running queries or make them timeout.

On the other hand, we also noticed that coding is almost a must when it comes to batch inserting records in a graph database. Even though there are other alternatives such as REST or Console commands, the truth is that the latter can only be run one at a time whereas the former needs to be restricted in terms of length (command sets should be 2 KB or 5 KB long at most). There is a chance of running multiple Cypher insert commands in Linux by using the pipelining feature but that does not work in Windows.

Therefore, at variance with SQL insertion scripts, which may sometimes be written by functional analysts with no technical background, Graph Databases might pose other needs from a skillset perspective.

Conclusion

As a result of this project, we had our first exposure to the world of Graph Databases, which differs significantly from what we were used to seeing in the traditional relational databases.

Particularly, the example we have built gave us the chance to explore and acquire hands-on experience in some of the most important features of this type of databases. This will hopefully provide with a good starting point to someone who is performing an initial evaluation of this technology.

Over the course of this report, we discussed and illustrated some of the compelling reasons that might drive a company to use Graph Databases. However, for the sake of objectiveness, we also examined some of the improvement opportunities that this developing technology might have.

Taking everything into account, we consider that there are no significant reasons to contradict the hypothesis we had at the time we started with our project. Indeed, Graph Databases seem to outperform other technologies when relationships are a key component of the model.

§§§