Masaryk University
Faculty of Informatics

Suitability of graph data models for different cases of data cohesion

Bachelor's Thesis

Matúš Štovčik

Brno, Spring 2019

Masaryk University
Faculty of Informatics

Suitability of graph data models for different cases of data cohesion

Bachelor's Thesis

Matúš Štovčik

Brno, Spring 2019

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Matúš Štovčik

Advisor: Mgr. Martin Macák

Acknowledgements

I would like to thank my supervisor, Mgr. Martin Macák, for introducing me to new topics and offering me the theme of my thesis. His active attitude and guidance led me to successful results. I would like to express my gratefulness to my parents, my whole family, and friends. Only with their support, I was able to finish every hard task that I came across.

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Abstract

This thesis is focused on finding suitable use cases for which to use graph management systems and for which not. We aimed at the cohesion factor of our data, therefore how much our data are interconnected.

Firstly, we described the theory behind NoSQL and SQL databases. We characterize some other available options and reasons behind selection. Chosen database systems were described more thoroughly, how they work, and how they store data.

Secondly, we designed tests for our data set, each differing in data complexity. Therefore, we focused on defining a threshold of table inter-connection from which is a graph database more suitable.

Keywords

NoSQL, SQL, benchmark, Big Data, Neo4j, PostgreSQL, MongoDB, graph database, relational database, document database …

Contents

List of Tables

Introduction

Recent years showed new ways of handling and processing data. However, a relational database has been default choice for a long period. Relational point of view has been embedded in data handling culture over the last decades. The dominance of relational databases has come to an end with the arrival of NoSQL [1].

Despite the appearance of NoSQL databases, the traditional relational database has still large support among many professionals. We can not claim that NoSQL is better than conventional SQL; each has its advantages and disadvantages. Both NoSQL and SQL have specific use cases. There is not only one right way to handle data storage.

A graph database is one of the NoSQL databases. Graphs proved their value in complex and chaotic data sets with many relationships. Traversal of relationships provides a fast way for data search and offers features that are hardly implementable in other NoSQL databases.

We choose to examine a difference between relational database, graph database, and document database. We want to concentrate on the complexity of data. The aim of the thesis is to provide guidelines for choosing the right type of database for a specific level of data cohesion.

The initial of our thesis is focused on graph databases and its advantages in comparison to traditional SQL databases and document databases. In the next chapters, we describe NoSQL and SQL databases in particular cases, and we examine tools, we used. Afterward, we specify benchmarking, results, and conclusion.

1 Database management systems

In this chapter, the storage was divided into two groups NoSQL and relational database systems. For NoSQL, we described four leading categories. For every category, we provided basic information, inner structure, use cases, and viable representants. Analogously, we described relational databases.

In the last section, we chose three representants of graph, document, and relational databases. For each representant, we presented a description and the reasons for being selected.

1.1 NoSQL management systems

NoSQL database management systems were created as a response to rising demand for more flexible databases, drawbacks, and limitations of SQL. Name NoSQL suggests that it is different from traditional SQL. However, many NoSQL database management systems remained in SQL-like syntax. NoSQL offers fast reading and writing operations, easy to scale and expand in clusters, Big Data capacity, lower cost. NoSQL complies with CAP theorem.

The main difference between SQL and NoSQL is how are data stored, NoSQL systems use more structure-free data, which means that each record can have a different number and types of columns. NoSQL is better scalable than SQL. NoSQL is a solution for the need for storing and querying dynamic data [2].

NoSQL databases are divided into four leading categories: graph databases, document databases, key-value databases, wide-column databases.

1.1.1 Graph databases

Graph databases use graphs of inter-connected nodes. Therefore these stores share similarity with object-oriented databases because of how databases are structured. A graph database is a network of nodes, containing data, linked by edges (relationships). Therefore graphs differ from other NoSQL databases by having implemented pre-made inter-connections between tables or pre-made joins. Graph databases

are useful in use cases for traversing and interpreting various networks, for example, social networks. Traversal between node allows us for faster search and pattern matching inside our networks [3]. Examples of graph databases: Neo4j, JanusGraph, Dgraph, InfiniteGraph.

1.1.2 Other NoSQL databases

Key-value databases store pairs of value and a unique key. These databases support high concurrency and Big Data like storage. Data can be queried and modified using mentioned unique keys. Values in Key-value pair may be a simple string, bit array or more complex collection like a list or set. Key-value databases can be used almost for storing anything after assigning a unique key []. Examples of key-value databases: GridDB, Riak KV, Aerospike.

Document databases use semi-structure of collections containing records (rows) with a variable amount of key and value pairs (columns). Comparing with key-value databases, we see that document database has better ability to store more queryable data. Various databases may use a different format, for example, JSON (Javascript Option Notation), BSON (Binary JSON) and XML. Document databases are suitable for storing semi-structured Big Data like collections, for example, emails, posts, user profiles. Examples of document databases: MongoDB, RavenDB, RethinkDB.

Wide-column databases, also called column-family databases, use distributed, a column-orientated data structure which assigns multiple attributes to a key. The database consists of column families; each contains rows with stored data. Rows contain unique row key and columns containing names, values, and timestamp. They are used similarly like document databases, with better scalability and worse querying options. Wide-column databases are capable of maintaining high-performance of data analysis [4] . Examples of document databases: Cassandra, HadoopDB, HBase.

1.2 Relational database management systems

Relational databases management systems, also knowns as RDBMS, were proposed around the year 1969 by E. Codd [5]. This concept uses

tables for storing data. Each table contains rows and columns, each row equals one record (entity), and columns represent. The database consists of tables; tables have a fixed structure, which is represented by columns (attributes), changing the mentioned structure in a deployed database requires nontrivial effort. Data are stored in rows (tuples); each row represents one record (entity). The relational aspect is stable, well-known, and embedded in the tech community. RDBMS has been developing for over 40 years. This mature technology supports various complex features such as aggregation, grouping, sorting, views, and joins. RDBMS uses SQL, structured query language, and offers ACID properties of transactions.

1.3 Databases under comparison

Representatives of three different database schemas have been selected for our comparison. All the three databases are prevalent and representative enough so that they can be mapped to other database systems which are currently in use. When choosing these representatives, we have determined the popularity of existing databases based on the DB-Engines website [6] where the popularity score based on multiple factors, like frequency of Google search, relevance in social networks, number of job offers, etc., is given. Naturally, we could only choose from the non-commercial databases. We decided to select a non-graph NoSQL database because in some cases, like when there is a need for a schema-less database, the relational database cannot be chosen. We faced a decision between a wide-column, key-value, and document databases. We decided for document database MongoDB as a suitable example because it is one of the most popular NoSQL databases.

1.3.1 Neo4j

As the representation of graph databases, the Neo4j version 3.5.3[1] was chosen. Neo4j is one of the leading software in graph databases with active support and development. We have found the Neo4j to be the right choice among other graph database software because it performed well in several comparisons with other graph databases [7,

1. `http://neo4j.org`

8, 9]. It uses nodes and relationships to store and navigate through data, which allows for less costly traversing through data than SQL joins. Nodes and relationships are labeled by name and grouped according to sets. Unlike conventional SQL management systems, Neo4j offers structure-free development, which adds to the agility of the whole data storage. This database offers easy replication via core and read-only nodes. In our case, we used three core nodes. Neo4j does not use the traditional concept of master and slave hierarchy; instead, nodes vote leader for every period to maintain freshness and availability. However, there are leaders and followers; only the leader is allowed to write operations.

We also wanted to compare the performance of other graph databases. Especially, we wanted to create a JanusGraph[2] cluster, because it stores the data in the adjacency list format, which is a different method from Neo4j [10]. However, we had many hard problems to run this database, so we do not include it here. The only other popular graph database we managed to run in a cluster was OrientDB. However, we exclude it from this work because it uses a similar method for data storage as Neo4j and is labeled as a multi-model database.

We considered DGraph[3] as other viable choice, but we found Neo4j more adequate option and a more solid choice for benchmarking. Running Dgraph in a cluster was a nontrivial task. Therefore we abandoned this idea.

1.3.2 PostgreSQL

To represent relational databases, PostgreSQL version 11.2[4] has been chosen. PostgreSQL is a modern object-relational database system with over three decades of active development. It ensures robustness, reliability, and performance. PostgreSQL is developed into strong competition with NoSQL management systems, mainly because of its scalability. This thesis tests PostgreSQL capabilities in large inter-table searches.

2. https://janusgraph.org/
3. https://dgraph.io/
4. https://www.postgresql.org/

1.3.3 MongoDB

MongoDB version 4.0.6[5] was chosen as a representative of document store databases. MongoDB is classified as a NoSQL database. Data are managed in structure-free storage with the capability of every collection to be different. Therefore, it stores data in a more flexible way than PostgreSQL. Its flexibility is one of the reasons for being chosen in our work. MongoDB uses collections instead of tables, and it uses a JSON-like document schema. Instead of a SQL table, MongoDB has collections, and it also uses references. MongoDB offers scalability in clusters, stability, and the great prospect of future development [11].

We faced a decision about whether to include a key-value database or wide column database as well. However, we found document databases very similar to both, but with better options for querying with more use cases. For our purposes, choosing the document database meant choosing a representative among the document database, key-value database, and wide column database as they share the same advantages and we found a document database more reasonable choice.

5. `https://www.mongodb.com/`

2 Design of experiments

This chapter contains basic information about our data set. We created the ERD model to better illustrate tables and relationships in our database. This chapter also describes the configuration of our clusters and importing of data. Lastly, we grouped our queries into three sections and provided a description of each query.

2.1 Description of our data set

To correctly determine for which level of data complexity are the publicly available graph databases more suitable, we decided to use a large data set named Microsoft Academic Graph [12] (Figure 2.1).

Our data set contains information about research papers presented on conferences. Furthermore, it consists of conference instances of mentioned conferences, authors, affiliations, journals, keywords, URLs, and references connected to research papers. The chosen data set contains over 1.7 billion rows, 14 tables, and 13 relationships between the mentioned tables. The number of rows provided us desirable volume, one of the aspects of Big Data. A number of relationships contributed to a series of distinct queries on our data set and allowed us to show the difference between references of SQL and traversal relationship of a graph database. Therefore the data set was the right choice for this comparative study.

2.2 Setup

Each node was configured in a cluster with an Intel Skylake (2.2GHz, 4 cores), 16GB RAM running on Ubuntu 16.04 LTS 64bit Linux 4.4.0 kernel.

Then, the first step in our research was to pre-process our data. We needed to parse data to load it properly. Afterward, we modeled the data to fit our specification. We had to load and model each database differently. Neo4j needed relationships for fast traversal between nodes. PostgreSQL required references, and foreign keys and MongoDB needed references to objects.

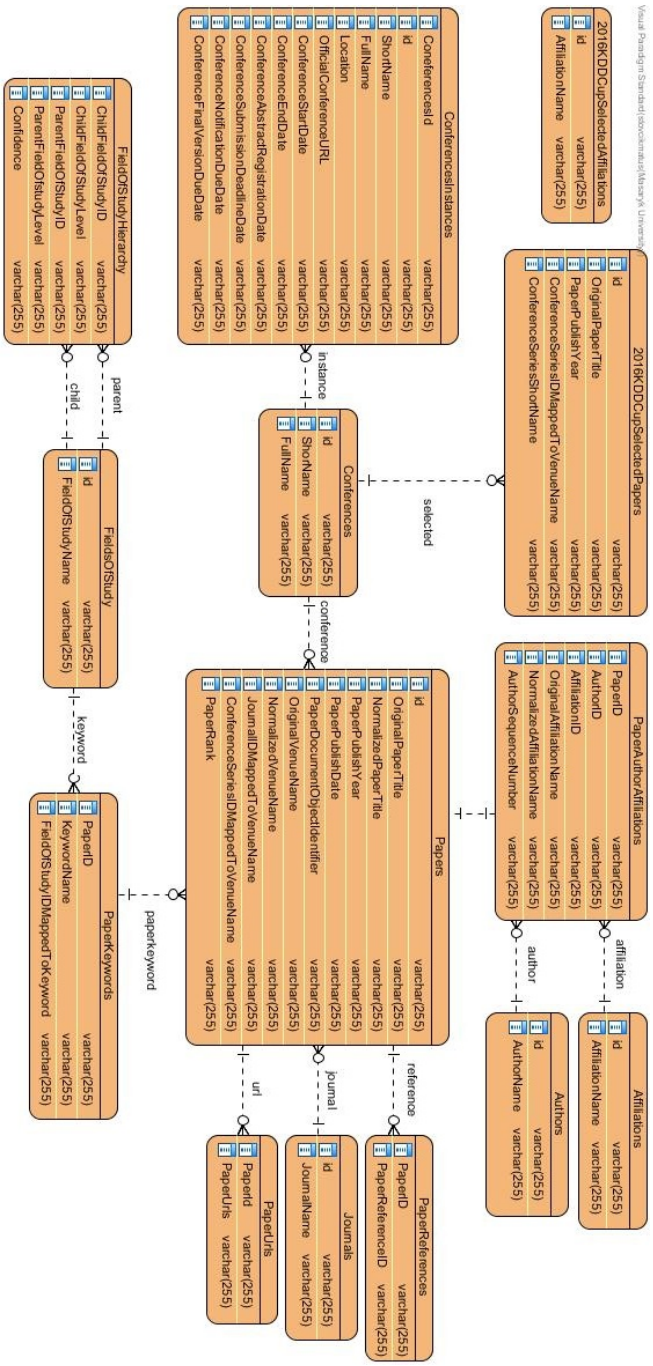

Figure 2.1: ERD of Microsoft Academic Graph data set

We decided to go with three node clusters with configuration focused on reading and high availability. Each of the chosen databases was configured to clusters in the most suitable way for the program and our use cases. We focused on memory and processing equality for our configurations. However, there are some differences between configurations of Neo4j, PostgreSQL, and MongoDB because each one of them has different needs, and our goal was to find the best settings for each mentioned database.

All three databases required CSV files for loading. However, each had different syntax and mechanics of this process. The best and fastest way of loading the data into Neo4j was to use 'neo4j-admin import' tool with a configuration and sequence. Next, PostgreSQL was loaded using its console. First, we had to use 'CREATE' for creating a table, and afterward, we used 'COPY' to copy the data set from a CSV file to the table. Lastly, MongoDB required us to use 'mongoimport' tool. Every data loading was done on master nodes, and then the masters propagated data to slaves or read-only nodes.

2.3 Queries

We aimed for queries to be in groups of a certain level of data cohesion. We created sets of queries depending on the inter-table connection. We focused on identifying the specific number of joins or traversals needed for a query to be effective in the graph database. We were continuously creating more complex queries.

For the correct way to compare our query results, we used built-in tool's timers. Each query was run five times. We removed the biggest and the lowest value from the results, and then we averaged the remaining three values for the final result.

We wanted to highlight the power of relationships in the graph database. The first set of queries named J1, J2, J3 were simple join across multiple tables. This queries aimed at determining the threshold of data-level complexity. We wanted to test real-world heterogeneous data. Therefore each table may have varied in several rows and level of interconnection.

The second batch of queries contained the same joins as the previous with the addition of filters and simple conditions. We wanted to

perform real-world use cases by using simple where statements. We assigned names W1, W2, W3 for these queries.

The third collection of queries focused on showing string operations, specifically contains a substring. By doing this, we wanted to test how database tools handle working with values by filtering values based on contents. We named these queries C1, C2, C3.

- J1: Counts how many papers have been presented at a conference.

```
PostgreSQL:
SELECT count(*)
FROM (conferences
JOIN papers
ON papers.conferenceseriesidmappedtovenuename
    = conferences.id);

Neo4j:
MATCH (b:Conferences)<-[q:Conference]-(c:Papers)
RETURN count(q) as count;

MongoDB:
db.Papers.aggregate([
{ $lookup:
{
    from: 'Conferences',
    localField: 'ConferenceSeriesIDMappedToVenueName',
    foreignField: 'id',
    as: 'ConferenceSeriesIDMappedToVenueName'
}
},
{
    $match:
        {
                ConferencesId: {$not: {$size: 0}}
        }
}
]).toArray().length
```

- J2: Counts number of papers presented at conference instances.

```
PostgreSQL:
SELECT count(*)
FROM ((SELECT conferences.id AS cid FROM
    (conferences JOIN conferenceinstances ON
    conferenceinstances.conferencesid = conferences.id))
    AS dist1
    JOIN
    papers ON papers.conferenceseriesidmappedtovenuename
        = dist1.cid);
```

```
Neo4j:
match (a:ConferenceInstances)-[r:Instance]->(b:Conferences)
    <-[q:Conference]-(c:Papers)
return count(q) as count;

MongoDB:
conference= db.ConferenceInstances.aggregate([
{ $lookup:
{
    from: 'Conferences',
    localField: 'ConferencesId',
    foreignField: 'id',
    as: 'ConferencesId2'
}
},
{
    $match:
    {
            ConferencesId2: {$not: {$size: 0}}
    }
}
])

db.Papers.aggregate([
{ $lookup:
{
    from: 'conference',
    localField: 'ConferenceSeriesIDMappedToVenueName',
    foreignField: 'ConferencesId',
    as: 'ConferenceSeriesIDMappedToVenueName2'
}
},
{
    $match:
    {
        ConferenceSeriesIDMappedToVenueName2:
            {
                $not: {$size: 0}
            }
    }
}
]).toArray().length
```

- J3: Counts number of journals presented at conference instances.

```
PostgreSQL:
SELECT count(*)
FROM (SELECT papers.journalidmappedtovenuename AS jid FROM
    ((SELECT conferences.id AS cid FROM (conferences
    JOIN
    conferenceinstances ON conferenceinstances.conferencesid
        = conferences.id))
    AS dist1
    JOIN
    papers ON papers.conferenceseriesidmappedtovenuename
        = dist1.cid))
```

```
    AS dist2
    JOIN
    journals on journals.id = dist2.jid;

Neo4j:
match (a:ConferenceInstances)-[r:Instance]->(b:Conferences)
    <-[q:Conference]-(c:Papers)-[o:Journal]->(d:Journals)
return count(r) as count;

MongoDB:
conference= db.ConferenceInstances.aggregate([
{ $lookup:
{
    from: 'Conferences',
    localField: 'ConferencesId',
    foreignField: 'id',
    as: 'ConferencesId2'
}
},
{
    $match:
    {
        ConferencesId2: {$not: {$size: 0}}
    }
}
])

db.Papers.aggregate([
{ $lookup:
{
from: 'conference',
localField: 'ConferenceSeriesIDMappedToVenueName',
foreignField: 'ConferencesId',
as: 'ConferenceSeriesIDMappedToVenueName2'
}
},
{
    $match:
    {
      ConferenceSeriesIDMappedToVenueName2: {$not: {$size: 0}}
    }
},
{ $lookup:
{
    from: 'Journals',
    localField: 'JournalIDMappedToVenueName',
    foreignField: 'id',
    as: 'JournalIDMappedToVenueName2'
}
},
{
    $match:
    {
        JournalIDMappedToVenueName2: {$not: {$size: 0}}
    }
}
])
```

We provide the code of J queries solely because C and W queries used the same joins. They only differ in operations performed on data, so we provide only the textual description.

- W1: Counts how many papers have been presented on conferences with specified 'short name'.
- W2: Counts the number of papers, linked through a conference with specified 'short name', that have been presented at conference instances.
- W3: Counts the number of journals, linked through a conference with specified 'short name', that have been presented at conference instances.
- W4: Counts how many papers with specified 'original paper title' have been presented at conferences.
- C1: Counts the number of papers, linked to a conference with 'short name' containing specified sub-string, that have been presented at conference instances.
- C2: Counts the number of papers, linked through a conference with 'short name' containing specified sub-string, that have been presented at conference instances.
- C3: Counts the number of journals, linked through a conference with 'short name' containing specified sub-string, that have been presented at conference instances.

3 Results of experiments

This chapter contains tables with results for each query. We provide interpretations and evaluations of results as well as explanations for unexpected outcomes.

3.1 Measurements

3.1.1 J queries

Table 3.1: Measurements of J1 query

#	Neo4j	PostgreSQL	MongoDB
1.	524 ms	22561 ms	> 180 s
2.	571 ms	22538 ms	> 180 s
3.	614 ms	22440 ms	> 180 s
4.	565 ms	20456 ms	> 180 s
5.	587 ms	21458 ms	> 180 s

Table 3.2: Measurements of J2 query

#	Neo4j	PostgreSQL	MongoDB
1.	3112 ms	1104 ms	> 180 s
2.	3023 ms	1127 ms	> 180 s
3.	3178 ms	1159 ms	> 180 s
4.	3301 ms	1009 ms	> 180 s
5.	3103 ms	1113 ms	> 180 s

Table 3.3: Measurements of J3 query

#	Neo4j	PostgreSQL	MongoDB
1.	1732 ms	1355 ms	> 180 s
2.	1760 ms	1185 ms	> 180 s
3.	1785 ms	1186 ms	> 180 s
4.	1760 ms	1204 ms	> 180 s
5.	1715 ms	1175 ms	> 180 s

We expected that PostgreSQL will dominate less complex inter-connections. However, in J1 we found that Neo4j handled values counting of the join between the enormous size of Papers and Conferences better. PostgreSQL did not have a space for any optimizations. Therefore Neo4j performed better with pre-made relationships.

Moving to J2 and J3, PostgreSQL performed better than Neo4j. Despite the pre-made relationship, Neo4j performed worse. We accredit this behavior to PostgreSQL optimizations of joins where it did not have to use every row in both the joined tables. Instead, PostgreSQL chose a subset of rows which led to improved performance. However, Neo4j searched for raw matches inside the database.

MongoDB performed very poorly with time exceeding 3 minutes.

3.1.2 W queries

Table 3.4: Measurements of W1 query

#	Neo4j	PostgreSQL	MongoDB
1.	23 ms	3.621 ms	> 180 s
2.	27 ms	3.737 ms	> 180 s
3.	47 ms	3.753 ms	> 180 s
4.	23 ms	3.664 ms	> 180 s
5.	25 ms	3.831 ms	> 180 s

Table 3.5: Measurements of W2 query

#	Neo4j	PostgreSQL	MongoDB
1.	30 ms	24.684 ms	> 180 s
2.	44 ms	23,895 ms	> 180 s
3.	43 ms	22,352 ms	> 180 s
4.	31 ms	22.104 ms	> 180 s
5.	30 ms	23,455 ms	> 180 s

Table 3.6: Measurements of W3 query

#	Neo4j	PostgreSQL	MongoDB
1.	73 ms	35.498 ms	> 180 s
2.	69 ms	33.987 ms	> 180 s
3.	68 ms	33.254 ms	> 180 s
4.	75 ms	34.477 ms	> 180 s
5.	69 ms	34.664 ms	> 180 s

The used data set contained only string values, so we decided to go with a basic string search. We searched for a specific conference across one (W1), two (W2), or three (W3) tables. We found that PostgreSQL managed to perform better. The difference was significant; Neo4j needed almost eight times more time. MongoDB was greatly outperformed, with every query taking longer than 3 minutes.

Table 3.7: Measurements of W4 query

#	Neo4j	PostgreSQL	MongoDB
1.	2256 ms	22559 ms	> 180 s
2.	2481 ms	22116 ms	> 180 s
3.	2509 ms	22504 ms	> 180 s
4.	2486 ms	22415 ms	> 180 s
5.	2500 ms	22496 ms	> 180 s

Query W4 was very similar to query W1. The difference was between the direction of traversal. Neo4j can match relationship even if it is coming into a node, not only when it comes from the node. But there is performance difference. We showed that searching for value in node from which starts a relationship is faster than the opposite. These results showed us that Neo4j might perform nine times better if relationships are in the right order.

3.1.3 C queries

Table 3.8: Measurements of C1 query

#	Neo4j	PostgreSQL	MongoDB
1.	118 ms	86.785 ms	> 180 s
2.	129 ms	88.536 ms	> 180 s
3.	123 ms	86.334 ms	> 180 s
4.	152 ms	86.595 ms	> 180 s
5.	131 ms	88.160 ms	> 180 s

Table 3.9: Measurements of C2 query

#	Neo4j	PostgreSQL	MongoDB
1.	2301 ms	1108 ms	> 180 s
2.	2317 ms	996 ms	> 180s
3.	2309 ms	1001 ms	> 180 s
4.	2298 ms	1051 ms	> 180 s
5.	2281 ms	1064 ms	> 180 s

Table 3.10: Measurements of C3 query

#	Neo4j	PostgreSQL	MongoDB
1.	51 ms	21.158 ms	> 180 s
2.	47 ms	21.790 ms	> 180 s
3.	57 ms	22.480 ms	> 180 s
4.	46 ms	21.297 ms	> 180 s
5.	48 ms	23.329 ms	> 180 s

We decided to use the contains function in values. The queries were modeled as one (C1), two (C2), or three (C3) joined tables as we wanted to show how each database performs on different inter-connection levels. PostgreSQL performed better than Neo4j. However, the difference between Neo4j and PostgreSQL was smaller than in the W queries.

3.2 Interpretations of results and recommendations

For each query measurement, we excluded the maximum and minimum value, and from the rest, we computed an average.

Table 3.11: Mean values of J queries

J	Neo4j	PostgreSQL	MongoDB
1.	574 ms	22145 ms	> 180 s
2.	3140 ms	1115 ms	> 180 s
3.	1751 ms	1191 ms	> 180 s

Neo4j managed to vastly outperform PostgreSQL and MongoDB in J1 query. It shows that the relationships of the graph have great potential. Despite having worse disk space demands, it allowed Neo4j to perform 20 times better than PostgreSQL. Moving to more complex joins, we found that PostgreSQL achieved better results. We wanted to preserve the direction of the relationship in Neo4j based on data set. Therefore we had some relationships in a different direction across our path. We concluded that this inconsistency in path direction led to slower performance.

Table 3.12: Mean values of W queries

W	Neo4j	PostgreSQL	MongoDB
1.	25 ms	3.718 ms	> 180 s
2.	34,667 ms	23.234 ms	> 180 s
3.	70.333 ms	34.376 ms	> 180 s
4.	2498 ms	22471 ms	> 180 s

This set of queries was focused on the ability of effective search. PostgreSQL performance was overall better. However, Neo4j was outperformed only by 36 ms. Neo4j performed W4 query in ninth of PostgreSQL time. These results showed that the direction of relationships, whether it comes from node A to node B or vice-versa, has a large impact on the performance of Neo4j. We would recommend carefully consider the creating directions of relationships based on the expected usage of this graph database. There is also an option of creating two-way directions of relationships, but in that case, there would be a greater sacrifice of disk space.

Table 3.13: Mean values of C queries

C	Neo4j	PostgreSQL	MongoDB
1.	128 ms	87.18 ms	> 180 s
2.	2303 ms	1039 ms	> 180 s
3.	48.333 ms	21.856 ms	> 180 s

Relation database performed better in the C queries. Results have shown that the contains function was used faster by PostgreSQL. Neo4j needed twice the amount of time to finish the tasks comparing to PostgreSQL. However, in the C1 and C3 queries, Neo4j lost by 40 ms and 20 ms, which again proved that Neo4j performs decently.

Despite Neo4j having slightly worse performance, we find our results interesting. This data set was vast but not chaotic enough for Neo4j to thrive. We concluded that PostgreSQL performed better in our benchmarks than Neo4j or MongoDB. However, differences of 20-40 ms between Neo4j and PostgreSQL times proved that Neo4j

could perform almost as well as PostgreSQL on more relational-based data.

MongoDB performed poorly. As mentioned, the scheme of data was more relational-based, with a fixed structure and normalized tables. The normalized data in this database caused the low performance. MongoDB is not well-equipped for join operation; it offers 'lookup' aggregate function, which allows us to only left outer join. This join was not a desirable outcome, because to get the right data, we had to filter empty joins, which cause another blow for performance.

4 Conclusion

In this thesis, we perform tests that compare the Neo4j graph database to the non-graph databases, PostgreSQL, and MongoDB. Based on our knowledge, this is the first tries to compare the performance of graph and non-graph databases in a cluster. We compared the execution times of these databases for several non-trivial queries on a real data set. These queries were performed on multiple levels of joins so we could determine in which cases are more efficient to use a graph database rather than a non-graph database.

Our tests have shown that for several cases, a graph database can have similar performance as a relational database in a cluster of three machines. We also found that there is a significant difference of graph database performance between filtering values on nodes with direction heading out instead of in. We provided recommendations for when to use the graph database and show the results of performance measurements of the chosen graph database and non-graph databases. We found that MongoDB was not a good choice for our tests, because it was outperformed in every query due to normalized data structure and its implementation of joins.

Generally, graph databases perform better on complex data when traversing in the direction of relationships. We must point out the difference between graph databases and relational databases. RDBMS performed better more times than graph databases. However, the maximum difference was about 100 ms. On the other hand, graph databases outperformed RDBMS by multiple times in suitable use cases. We can recommend using graph databases for more complex and inter-connected data. Meaning, if we are expecting queries that have to use data from multiple tables, we recommend using graph databases with the right orientation of edges.

4.1 Future directions

We think that results were affected by the configuration of nodes in our clusters. We suggest using more nodes with greater computational power in clusters and different data set. Furthermore, we propose

using other database tools to see the difference between the same type of databases, but with a different implementation.

We learned that the execution time of queries depends on edges directions. We suggest implementing bi-directional edges. Despite bigger disk space usage, we expect a significant improvement in execution times.

Implementing more complex relationships would affect results. In our work, we managed to join four tables in one query because of technical difficulties. To prove which level of data complexity is better to use graphs, we recommend generating Big Data with a more suitable structure and more complex network of nodes for implementing into graph database tools.

We had limitations concerning RAM; we recommend to upgrade specification to at least 32GB of RAM. We had to carefully configure our clusters to run at the most efficient setup for our RAM. We think that using more RAM and better CPU can improve user experience, speed of importing data and will allow us to do more complex queries on data. Running databases in more clusters can affect the results of experiments. We advise considering adding more nodes to cluster setups.

We found out that MongoDB performs worse on normalized data than Neo4j or PostgreSQL. Using data set with the less normalized database could benefit MongoDB performance.

Bibliography

1. MAKRIS, Antonios; TSERPES, Konstantinos; ANDRONIKOU, Vassiliki; ANAGNOSTOPOULOS, Dimosthenis. A Classification of NoSQL Data Stores Based on Key Design Characteristics. *Procedia Computer Science*. 2016, vol. 97, pp. 94–103. ISSN 1877-0509. Available from DOI: https://doi.org/10.1016/j.procs.2016.08.284. 2nd International Conference on Cloud Forward: From Distributed to Complete Computing.
2. JING HAN; HAIHONG E; GUAN LE; JIAN DU. Survey on NoSQL database. In: *2011 6th International Conference on Pervasive Computing and Applications*. 2011, pp. 363–366. Available from DOI: 10.1109/ICPCA.2011.6106531.
3. MONIRUZZAMAN, A. B. M.; HOSSAIN, Syed Akhter. NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. *CoRR*. 2013, vol. abs/1307.0191. Available from arXiv: 1307.0191.
4. MACÁK, Martin. *Tools for big data analysis*. 2018 [cit. 2019-05-19]. Available also from: https://is.muni.cz/th/po5g7/. Master's thesis. Masaryk University, Faculty of Informatics, Brno. Supervised by Barbora BÜHNOVÁ.
5. CODD, E. F. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*. 1970, vol. 13, no. 6, pp. 377–387. ISSN 0001-0782. Available from DOI: 10.1145/362384.362685.
6. *DB Engines Ranking*. Available also from: https://db-engines.com/en/ranking.
7. DOMINGUEZ-SAL, D.; URBÓN-BAYES, P.; GIMÉNEZ-VAÑÓ, A.; GÓMEZ-VILLAMOR, S.; MARTÍNEZ-BAZÁN, N.; LARRIBA-PEY, J. L. Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark. In: SHEN, Heng Tao; PEI, Jian; ÖZSU, M. Tamer; ZOU, Lei; LU, Jiaheng; LING, Tok-Wang; YU, Ge; ZHUANG, Yi; SHAO, Jie (eds.). *Web-Age Information Management*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 37–48. ISBN 978-3-642-16720-1.

8. JOUILI, S.; VANSTEENBERGHE, V. An Empirical Comparison of Graph Databases. In: *2013 International Conference on Social Computing*. 2013, pp. 708–715. Available from DOI: 10.1109/SocialCom.2013.106.

9. CIGLAN, M.; AVERBUCH, A.; HLUCHY, L. Benchmarking Traversal Operations over Graph Databases. In: *2012 IEEE 28th International Conference on Data Engineering Workshops*. 2012, pp. 186–189. Available from DOI: 10.1109/ICDEW.2012.47.

10. *JanusGraph*. Available also from: https://janusgraph.org/.

11. *The rise of MongoDB*. Available also from: https://medium.com/@joerezendes_91375/the-rise-of-mongodb-8ac6f7cb8a95.

12. SINHA, Arnab; SHEN, Zhihong; SONG, Yang; MA, Hao; EIDE, Darrin; HSU, Bo-June (Paul); WANG, Kuansan. An Overview of Microsoft Academic Service (MAS) and Applications. In: *Proceedings of the 24th International Conference on World Wide Web*. Florence, Italy: ACM, 2015, pp. 243–246. WWW '15 Companion. ISBN 978-1-4503-3473-0. Available from DOI: 10.1145/2740908.2742839.