Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Stine Forås

Database Systems in Relation to Sports Data

A Performance Test of a Relational Database and Graph Database

Master's thesis in Master of Computer Science
Supervisor: Svein Erik Bratsberg
June 2020

NTNU
Norwegian University of Science and Technology

Stine Forås

Database Systems in Relation to Sports Data

A Performance Test of a Relational Database and Graph Database

Master's thesis in Master of Computer Science
Supervisor: Svein Erik Bratsberg
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

This project aims to find out whether relational databases or graph databases would be the better fit when working with sports data. There are several different systems that could be tested, but for this study, MySQL and Neo4j are the systems chosen to be compared. Whether one of them was significantly better than the other should be determined by running a benchmarking process with several different queries to see how they perform and compute statistical tests on the results to determine how much they differ. The data used are soccer data from Premier League season 19/20. However, due to the spread of Covid-19 and Premier League being postponed from middle of March 2020, there was less data than originally planned. Three different databases was created for both database systems with different structure in each to be able to test their performance on different levels of depth and amount of data. For this study an external server was used for MySQL and a local server for Neo4j, which created an advantage for MySQL. But as the structure of the data was in favor of Neo4j, being a graph database, it was expected that it should not perform much worse and hence an interesting point to look at when comparing the two systems. Due to the choice of two different types of servers, the topic of local vs. external database will be briefly discussed as well, but was not the foundation of the study. The results of this study shows that even with a more powerful server MySQL might not be significantly better than Neo4j and that other factors can be considered when choosing the best system to handle sports data.

Sammendrag

Dette studiet ønsket å finne ut om relasjonelle databaser eller grafdatabaser ville være det beste alternativet når man jobber med sportsdata. Det er flere forskjellige systemer som kunne blitt sammenlignet, men for denne studien ble MySQL og Neo4j valgt. Om én av de var signifikant bedre enn den andre ble bestemt ved å kjøre en "benchmarking" prosess med forskjellige spørringer, og deretter utføre statistiske tester på resultatene for å se hvor forskjellige de var. Dataen brukt for denne studien var data fra Premier League sesong 19/20. Grunnet spredningen av Covid-19 og at Premier League ble utsatt fra midten av mars, var det mindre data enn opprinnelig planlagt. Tre forskjellige databaser ble lagd i begge databasesystemene med forskjellig struktur for å kunne teste ytelse etter forskjellige nivå og mengde data. Denne studien benyttet en ekstern server for MySQL og en lokal server for Neo4j som ga MySQL en fordel. Men siden datastrukturen var til fordel for Neo4j, da den er en grafdatabase, var det forventet at ytelsen ikke ville være mye dårligere og dermed et interessant punkt å se på i sammenligningen. Siden det ble valgt å bruke to forskjellige servere, er også temaet som omhandler ekstern vs. lokal server diskutert kort, men dette var ikke grunnlaget for studiet. Resultatet fra denne studien viser at selv om MySQL hadde en kraftigere server, er den muligens ikke signifikant bedre enn Neo4j og andre faktorer kan vurderes når man velger det beste systemet for håndtering av sportsdata.

Acknowledgment

Thank you to Sportradar for allowing me to use their data in this project and to Martin Folke Emdal for giving me guidance when choosing systems and creating databases and queries. I would also like to thank my advisor Professor Svein Erik Bratsberg from the Norwegian University of Science and Technology (NTNU), for feedback and support.

Contents

List of Figures

List of Tables

Listings

1 Introduction

In the last decade the amount of data collected from various sources have grown enormously due to simpler ways of collecting data and faster ways of sending and distributing it. This is also true for sports data. Today, one is able to collect thousands of data points from one single game using sensors, cameras and observations. The company Sportradar AG collects sports data and analyze it for multiple purposes. In order to be able to do this they need good systems for saving and querying data that is collected. The most common systems are relational databases as they are dependable and relatively fast for querying data that is not too large and does not need to be handled in real time. However, due to the increase of the amount of data, new database systems have emerged in the past years, one of them being graph databases. They have their origin from social networks, where it was discovered that by structuring the database on disk after the graph structure, querying could be made easier and faster. This has led to more applications for the database, and it is now used by many enterprises all over the world.

1.1 Topics to be Studied

This section presents what this case study examined and how the study was performed. First it looks into general background of the technologies and then describe in more detail the chosen systems and databases and how they are implemented and designed.

1.1.1 Database Technologies

The database technologies that this study wants to examine and perform a benchmarking process on are relational databases and graph databases. It will describe how they work and why they can be used for this particular use case and data. This section will also present why Neo4j was the chosen graph database system based on the project work done by the author. Hence section 3 and subsection 2.2 is partly based on the project work done by the author. [For19]

1.1.2 Implementations

The implementation section presents the database systems chosen for the benchmarking process and describes the implementation of each of the systems for this project. It presents the query languages used for each system and subsection 5.5.2 is partly based on the project work done by the author. [For19] Further, there will be a detailed description about the data imported to the databases, the structure of the databases in both systems and the queries to be used in the benchmarking process.

1.1.3 Results from Queries

This section presents all results from queries done on the data in both databases. A description of how the results are reached and how statistical tests are performed is presented as well. This section will be the basis of the discussion and conclusion for this study.

1.2 Sportradar AG

This case study was based on data from the company Sportradar AG. Sportradar is an international company that works with collecting sports data and attain value from it for many different purposes. Their mission is "to empower the broadest range of businesses with state-of-the-art sports data and digital content solutions that fuel the passion of sports fans across the globe."[Spo] Having data and services in-house they provide solutions that their customers need for a complete sporting experience.[Spo] In order to provide the best services, they are ever growing and interested in new insight and thoughts. This project aims to provide insight into database technologies that could be useful for Sportradar.

2 Database Technologies

Today there are several database technologies that can be used for a variety of applications. The most common being the relational database as it has proved to be the best and most stable system for decades. However, as the amount of data accumulated has increased drastically over the years, it has been an increase in the need of scalable systems. Relational databases has a problem when it comes to scalability and NoSQL databases has been proposed as a solution for this. However, there seem to be a drawback of NoSQL databases, that they are not as reliable. Graph databases are NoSQL databases, but in most cases based on native graph storage. They therefore have the same challenge as other NoSQL databases. Some databases have chosen to meet this challenge by ensuring that they are dependable by applying some of the features used in relational databases. This section discusses relational databases and graph databases and their use on sports data.

2.1 Relational Databases

Relational databases has been the most commonly used database technology for decades. It has a wide range of application and many different systems exist that are based on the relational model. This section describes what a relational database is and why it can be used for soccer data.

2.1.1 What is a Relational Database?

Relational Databases are databases that stores data and their relations to each other. It utilizes an intuitive relational model with tables and relationships between them. Data is stored in tables consisting of rows and columns. The rows are records of data and has an ID attribute, referred to as a key. Each record has a unique key, which is the primary key, used when pointing to other records it is related to. Other attributes can be set when creating an record and can be used when querying the database for information. The columns are where the content of each attribute is held and for most attributes this value is set, but it can be empty if this attribute is not relevant for the particular record.[Ora]

Before the Relational Model was created, data was structured in individual ways and if a user wanted to manipulate the data it had to gain knowledge of how this particular data structure. This was time consuming and required a lot of work before an actual application for manipulating data could be made. The Relational Model changed this and created a universal method for structuring data that was intuitive and easy to use due to its utilization of tables as a structure. [Ora]

Oracle lists four major benefits of using Relational Databases: The first one is Data Consistency which makes sure that multiple instances of the database has the same data at all times. The second one is Commitment and Atomicity which means that it has strict rules for commitment and atomicity ensures that data is equal for all instances. It will not update data unless it knows it can be updated for all instances. Stored Procedures and Relational Databases is the third benefit and allows users to store procedures to avoid extra work of writing them repeatedly. Also it helps to ensure that data is stored in a particular way, which creates consistency. The last benefit is Database Locking and Concurrency. This avoids conflicts when several users are accessing the same database simultaneously and to uphold integrity. Locking ensures that users cannot access data while it is being updated and this can be done at a table level or at record level, where doing it at a record level makes it easier to work with the database as it does not lock the entire table for a record update. Concurrency gives permissions according to data control policies when multiple users are querying data simultaneously on the same database. [Ora]

2.1.2 Relational Databases for Sports Data

Sportradar collects millions of data points for many sports throughout a season. This must be stored in order to analyze it and relational databases is a natural choice as the data is connected to each other. For example a game has players, teams and events which are all interconnected. Hence a relational database would be able to display these relationships in a natural way using tables with rows and columns. Additionally, looking at the benefits of a relational database presented in the previous section, the data is ensured to stay consistent even though many employees are accessing data simultaneously.

2.2 Graph Databases

Graph Databases has in recent years grown in popularity for use cases where data that is heavily connected. By structuring the data differently, using nodes and edges, it brings certain advantages for storing and querying the data in addition to an even clearer visual display of relationships between data. This section will describe what a Graph Database is and why it can be used for sports data.

2.2.1 What is a Graph Database?

Since the Internet was launched the world is getting deeply connected. It is based on a network of nodes and edges representing the computers and how they are connected. This principle is easy to understand because it is visual and one can trace the connections through the graph. Today the world is facing a new issue: how to make sense of the large amount of data that is retrieved every day. Graph databases are NOSQL-based and has tables on disk like relational databases, but unlike relational databases, the relationships between the objects are stored in lists related to each node. By using this relationship model instead, the search-and-match computation can be avoided, and the relationships can be treated as equally important as the data. In many cases when analyzing data, the relationship between the data can be of a higher value, and it can be easier to find this using graph databases.[Neoa]

2.2.2 Why can Graph Databases be Used for Sports Data

One of the challenges in today's computer ruled world is the large amount of data that is retrieved and stored for the purpose of being analyzed and used to improve services and benefit the users. To retrieve relevant information is crucial, but even more important is that the relevant information is retrieved and presented at the right time, which puts a lot of pressure on the performance of the systems used for storing and querying data. Sports data is highly related and there is a lot of information in the relationships between the objects. The arguments for using graph databases are based on three categories: performance, flexibility and agility.[RWE15a]

The performance tends to remain relatively constant in a Graph Database even when the data set increases. This is mostly due to the fact that one can perform queries that are only localized to a certain part of the graph. Hence the execution time for each query only depends on the size of the portion of the graph traversed for the given query, rather than the size of the whole graph.[RWE15a]

Graphs in general benefits from the fact that it can grow in the and manner of how the user wants it to. This flexibility is useful because in many cases one does not know the entire complexity of the situation in advance, and to model and build a database step by step is a lot easier. Graphs are also a lot easier to change during the research as one easily can remove nodes and relationships for specific parts, without effecting other nodes and their relationships.[RWE15a]

Developing software systems is often done by incrementally adding new solutions, reviewing them, changing them and then repeating this process. Therefore, it is natural that this incremental and iterative way is used when building databases as well. The agility property lets the user add information as they continue to acquire more information and understanding of the problem. [RWE15a]

3 Choosing the Graph Database System

Choosing the graph database system for this comparison was done on the basis of the project work done by the author. [For19] This section presents the different database systems that was eligible and why Neo4j was chosen.

3.1 Neo4j's Guide to Compare Graph Technologies

Neo4j is the leading provider of graph databases today, and therefore has sat the standards for many of the definitions within the technology and have also laid the foundation for continuous work within the field. However, they have recognized the increasing offer of graph technologies out there and have provided a guide to what should be considered when choosing a graph database system. [Neoc] This guide will be used as a guideline for the comparison in this paper, with some additional elements discovered from trying out the different solutions. The different topics used for the description and comparison of the different systems are as follows: Open Source Foundation Community, Native Graph Storage, ACID Compliance, Graph Query Language, Hybrid Transactional-Analytic Platforms (HTAP), Graph Platform with Tools and
Support for All Types of Users and Business Model, Focus & Staying in Power. [Neoc]

3.2 Neo4j

Neo4j is as mentioned in the previous section, the leading graph database system today. This mainly because it is easy to learn and provides many tools for learning the system properly in a tidy and constructive matter. According to the list from the graph database guide, provided by Neo4j for comparing graph databases, it scores well on most points. For the first point, Open Source Foundation Community, Neo4j is the leading system, it has a well-established user group and has the "Biggest and Most Active Graph Community on the Planet" [Neod] The second point addresses whether it is a native graph storage system, also here Neo4j initiated this term and has set the definition for it. According to themselves "It is the only enterprise-strength graph database that combines native graph storage, scalable architecture optimized for speed, and ACID compliance to ensure predictability of relationship-based queries." [Neod]. This quote also addresses the third point. Neo4j uses Cypher as it's query language which is "one of the most powerful and productive graph query

languages in the world" [Neod] according to themselves This combined with a clear and easy-to-understand interface, Neo4j scores high on being intuitive. The fifth point addresses the issue of evaluating the system based on their own measures. There is a comparison between Neo4j and MySQL as presented in section 4 and this case study will test the same two systems on sports data. However, when other systems compare themselves to others, it is highly common that they compare their system to Neo4j which says something about Neo4j's position. The sixth point checks whether the system that is being tested is easy to use even if you have little to no experience with databases or programming beforehand. As discussed above, Neo4j is highly intuitive and with the numerous resources of learning tools, it is easy for anyone who wants to learn, to master it. The last point checks whether the supplier is eager to stay up to date and support its users. Neo4j has numerous lectures and talks about graph databases, and by having the largest community of users, it is always being tested and improved by them, giving Neo4j a great advantage for future improvements.

3.2.1 OrientDB

One of the biggest opponents of Neo4j is OrientDB, which is a multi-model database. A multi-model database means that it combines different database models such as graph, document, key/value, reactive, object-oriented and geo-spatial into one operational database. This differs from Neo4j that only provides a graph model. According to their own website OrientDB has considered the enterprises needs for more than just one model and states that "in being able to view data in different models it provides more insight in today's age of big data". [Orid] However, as this study looks at comparing graph models, the following comparison will be regarding this model only. [Orie] Following the steps of the guide for choosing a graph database system, OrientDB is also an open source foundation and has a growing community. They provide a online school for learning how to set up and use the system for free and lets users discuss their issues in forums. It is however not as large as Neo4j's community, but it is growing, and they have many resources for users to dive into and get familiar with. [Oria] The second step is regarding native graph storage which OrientDB, like Neo4j, is based on. The relationships between the data are stored in the vertices, and in OrientDB it is stored as documents and as stated on their web page "Native graph databases that apply index-free adjacency report reduced latency in create, read, update and delete (CRUD) operations." [Oric] OrientDB also provide ACID transaction like Neo4j, which means that it preserves the properties of atomicity, consistency, isolation and durability during a transaction and checks of the third point in the list. The fourth point is concerned with the query language used. In OrientDB SQL is used for querying the graph. However, the SQL is modified in order to work with graphs instead of relational databases. This brings the benefit of being able to provide new users, with a background from SQL with an easier transition into the graph database world. OrientDB provides a comparison of itself towards the two other systems, Neo4j and MongoDB, where MongoDB also is a multi-model system. This relates to

the fifth point and in the comparison towards Neo4j, which is performed as an independent benchmark by Tokyo Institute of Technology and IBM Research in 2012 [DS12], however these benchmarks are based on tests in the cloud which differs from the case study presented in this paper, but the results from their own web site is shown in Figure 1, 2, 3 and 4.

In the first comparison as shown in the figure below the query is "A mix of 50/50 read/update workload. Read operations query a vertex V and reads all its attributes. Update operation changes the last login time." [Orib] One can see that the throughput of operations is quite faster than Neo4j.

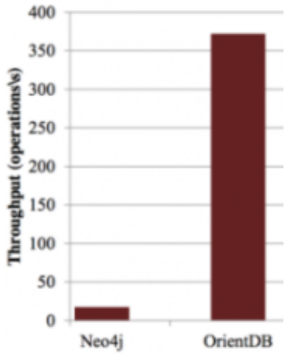


Figure 1: Comparison between Neo4j and OrientDB Workload A - Update heavy [Orib]

For the second workload B, which is read mostly, the query is a mix of 95/5 of read/update. This gives quite similar results as the previous workload.

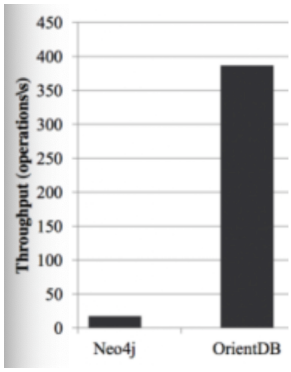


Figure 2: Comparison between Neo4j and OrientDB Workload B - Read mostly [Orib]

The third workload is a read latest and defined as follows: "Inserts new vertices to the graph. The inserts are made in such a way that the power-law

relations of the original graph are preserved." The graph in figure 3 shows that in this case OrientDB also outperforms Neo4j.

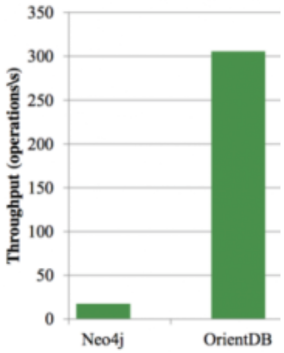


Figure 3: Comparison between Neo4j and OrientDB Workload C - Read latest [Orib]

The last workload tested is short ranges and defined as "Reads all the neighboring vertices and their Vertex attributes. For example, loading the closest friend to a person in a social graph." Figure 4 shows the result and OrientDB does not perform as good in this case, but still better than Neo4j.

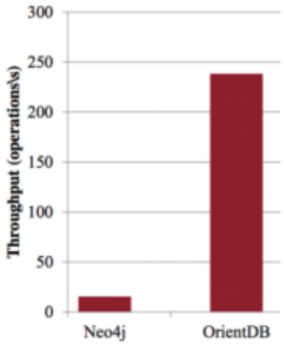


Figure 4: Comparison between Neo4j and OrientDB Workload D - Short Ranges [Orib]

The performance may depend on the type of data that is used and might be different in other cases. For the sixth point of the list OrientDB has chosen to give the users opportunity to choose the language they prefer between SQL and JavaScript which gives a wider range of new users an easy transition. Additionally, there are a lot of sources available on the internet to learn these languages quickly if one is not familiar with it. However, this system might not be as intuitive and easy to start out with and requires some knowledge of

using shells or Docker in order to set it up and the sources for learning is a bit outdated in this regard, which makes it harder for users not familiar with it. But when this hurdle is overcome the graphic interface in their studio is easy to follow and it does not require much knowledge to start creating a graph and querying it. OrientDB is as of the time this paper is written ranked number two of databases that has a community edition available. They are also quite forward leaning on their web page trying to gather more users and doing so by making it easy to migrate from other systems. The main difference from Neo4j is that they are pursuing a multi-model database, but the graph side is quite important, and it shows from their own web page that this is something that they spend a lot of time developing. They are also making it easy to convert a database from Neo4j into their systems, showing that they are interested in taking up the competition with the today's leading system.

3.2.2 Dgraph

Dgraph is a system that was released in 2016 and is now ranked number 11 among graph database systems according to "db-engines.com." [db-] It is a pure graph database that allows the user to create graph databases like the previously two described systems. It is a system that seems to be mostly focused on performance, but it provides an interface for users who prefers this over shell. When it comes to the first step in the guide for choosing graph databases systems, Dgraph is both open source and has a community for its users. Since Dgraph in many ways is made by developers for developers they are focused on letting others bring their thoughts and changes to the system and has a Slack-community where users can interact and contribute to each other. According to their own description of their system, "Dgraph is an open-source, transactional, distributed, native Graph database." [Rao] which addresses both the second and the third point of the list, being native and ACID. The fourth point is regarding the query language, which in Dgraph is GraphQL+-. GraphQL+- is based on Facebook's language GraphQL, which was not originally made for querying graphs, but its structure is very similar to the graph structure which makes it a natural choice when working with graphs. [D-g] GraphQL+- is developed by Dgraph for their solution and is a work in progress constantly being improved to facilitate operations and querying in an even more efficient way. [D-g] Since this language is based on a language that is not used by many, it requires users to learn a new language, making the transition harder, but for new users it should not be harder to learn than any other language used for graph databases. However, they do not provide any learning resources for setting up or learning the language which sets the barrier for learning the language higher. The fifth point addresses how Dgraph performs in comparison to other systems. At their own web page they have made a comparison to Neo4j as it is the leading system today. In their comparison they completed a benchmarking process. They made queries that was based on read and writes and since Dgraph does not do query caching they completed the test with both caching on and off in Neo4j, but standard for Neo4j is that it's turned on. [Raw]

The results from this comparison are presented in figures 5, 6, 7 and 8.

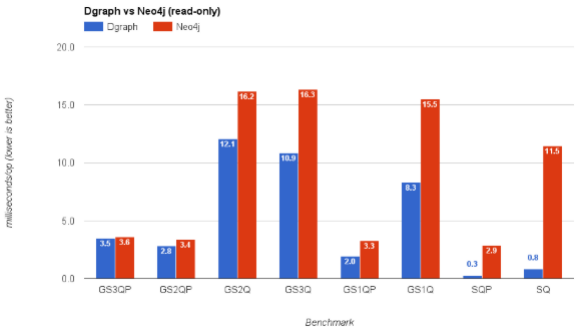


Figure 5: Dgraph vs. Neo4j Cache off read-only [Raw]

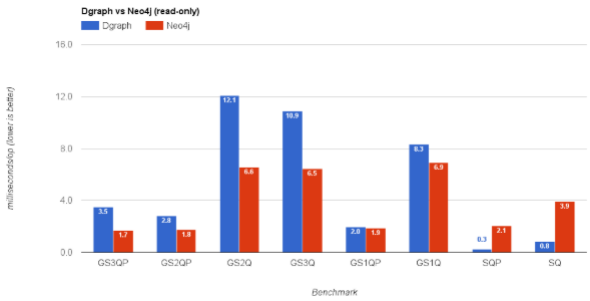


Figure 6: Dgraph vs. Neo4j Cache on read-only [Raw]

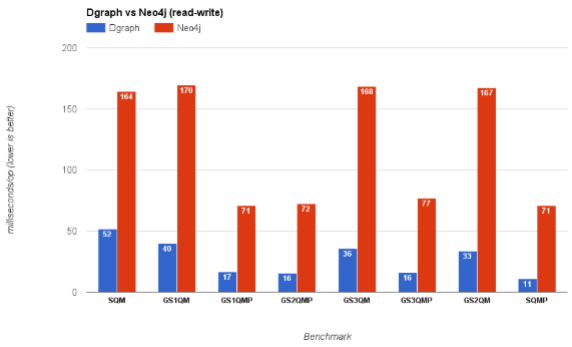


Figure 7: Dgraph vs. Neo4j Cache off read-write [Raw]

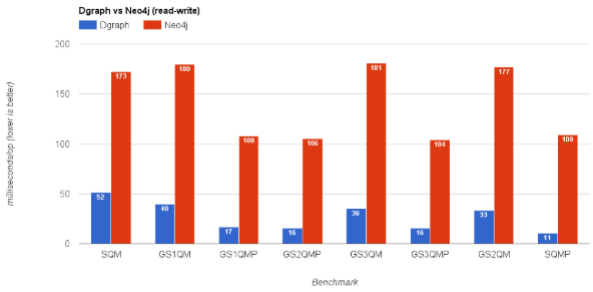


Figure 8: Dgraph vs. Neo4j Cache on read-write [Raw]

As shown in the results for this data, Dgraph performs better in all cases except for read-only when query caching is used in Neo4j. This benchmarking process was done in 2017 so changes to performance might have changed in the time between then and when this paper is written. The sixth point in the list for evaluating graph database systems considers if the system is easy to use for all users. Dgraph does have a user interface that can be used to manage the graph if wanted, however, it is not as intuitive as the other interfaces presented in this paper and therefore not straight forward to understand how to build and manage a graph. Thus, Dgraph might not be the best solution for new users not familiar with coding and databases in general. The last point on the list points to Dgraph's business model and how it tries to stay in power. From their web page and how they present their system they are focused on presenting what their strengths are regarding different topics and guides users directly into the steps of getting started with their system. However, it does not show tendencies of wanting to be the most preferred system by everyone, but to provide the best performance for certain cases as stated on their blog "Dgraph is optimized for high-performance reads and writes. It can serve queries and mutations with low latency and high throughput, even when they involve deep joins and traversals." [Rao]

3.3 Why Neo4j was chosen

For the benchmarking process in this project work Neo4j and OrientDB was chosen. One database was created which contained the data for five games for Manchester United during Premier League 19/20. The total graph can be viewed in figure 10. The results from this benchmarking process are presented in figure 9. This project work concluded that Neo4j was the preferred graph database system for this use case. This lead to choosing Neo4j as the graph database system in this case study.

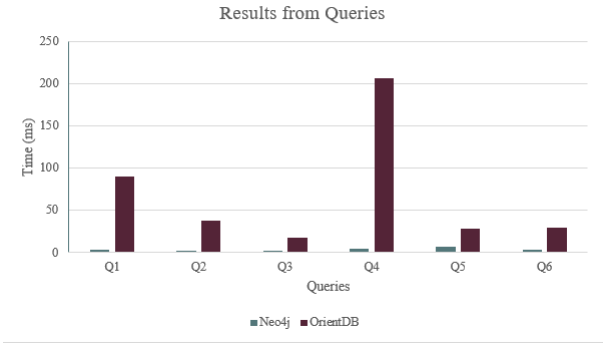


Figure 9: Results from queries in Neo4j and OrientDB [For19]

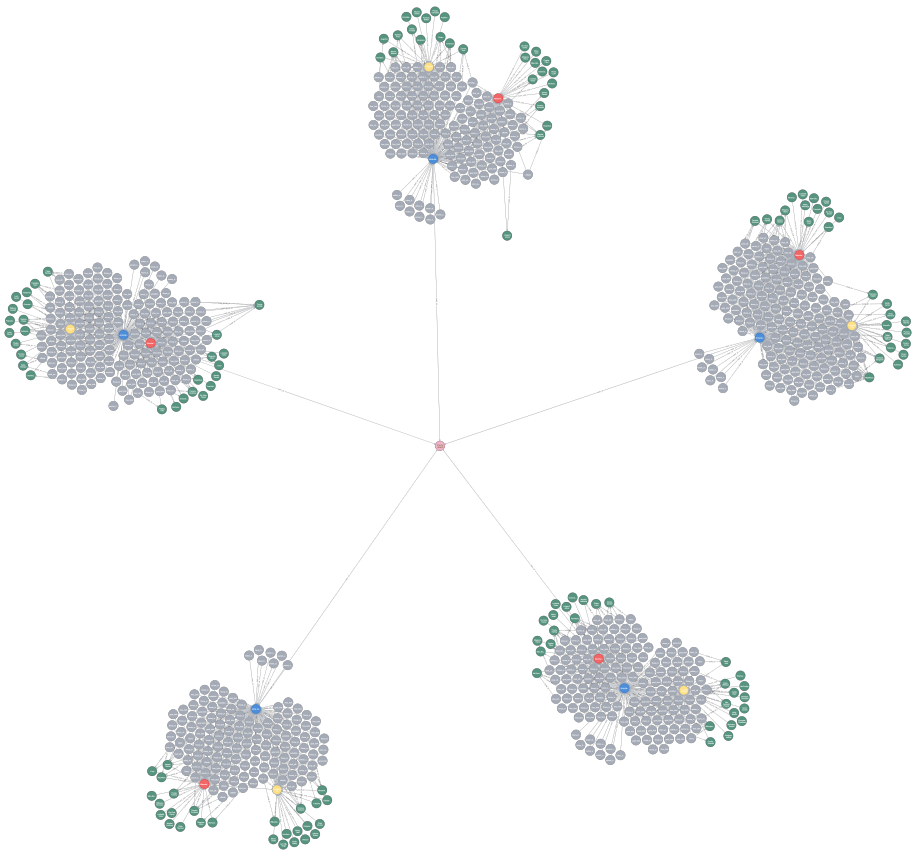


Figure 10: Neo4j Database in project work
[For19]

4 Related Work

There has been several experiments with Neo4j compared to a relational database trying to prove why it is a better choice for heavily connected data. One theory was that Neo4j will outperform MySQL when the depth of the structure increases. This was tested by Jonas Partner and Aleksa Vukotic in the book "Neo4j in action". For this comparison they used a social network graph with 1 million users and intended to test "friends of friends" queries. Which, essentially mean that they tested the graph by depth. The results from their comparison is shown in figure 11. This study shows that for queries written for a limit of 1000 users, Neo4j seems to perform better than MySQL as the depth increases. [new] This is an interesting case study which aims to show the benefits of graph databases and is the basis of what should be investigated in the comparison for this case study.

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

Figure 11: Results from comparison done by Partner and Vukotic [new]

5 Implementation

5.1 MySQL

MySQL was chosen as the relational database for this project and is widely used all over the world. MySQL is developed by Oracle and provides many different services, including community editions, enterprise editions, servers and clusters. It also provides provides a cloud service for handling larger amounts of data.[MySn] The server and storage engine used for this project, and the interface used for handling data will be presented in the following sections.

5.1.1 InnoDB

There are several ways to store data in MySQL depending on how data is handled. The data for this project needed to be updated for each game played, and it was important to be able to insert and query data easily for the benchmarking process. Therefore, the standard storage engine was chosen, namely InnoDB. This is a "general-purpose storage engine that balances high reliability and high performance" [MySl], which is needed to be able to execute queries rapidly and not risking loosing important information. InnoDB supports SQL language and by its Data Manipulation Language (DML) operations it goes along with the Atomicity, Consistency, Isolation and Durability (ACID) model

which supports commit, roll-back and crash-recovery. It supports multi-user concurrency and performance by featuring row-level locking and consistent reads after the Oracle style. [MySl] InnoDB will also store data on disk in a way that optimizes queries. This means that data is stored and structured in a way that reduces input/output (I/O) for lookups and all this is based on primary keys. To support integrity, the primary key has certain constraints, ensuring that when data is inserted, deleted or altered these constraints are checked in order to maintain consistency in the database. [MySl] InnoDB handles Disk I/O by the use of asynchronous disk I/O where it is feasible. This is done by creating threads to manage this operations, while giving other database operations the possibility to continue their operations while I/O is performed. InnoDB uses two major mechanisms to reduce the need for frequent disk access. The first one is Read-Ahead which means that if InnoDB recognize a high probability that some data will be needed in the near future, it executes a read-ahead operations gathering data into the buffer pool to make it available in-memory. This tactic might be useful as it creates less, but larger read request which can be more efficient than making many smaller ones. Read-ahead has two heuristics which are sequential and random. Sequential is used when a pattern is noticed in a segment in the tablespace stating that access is sequential and thus, it can read several pages in advance. Random is used if InnoDB realizes that most of the area in a tablespace is being read into the buffer, then it acquires the remaining area as well. The Doublewrite Buffer is the second mechanism used and is described in the paragraph for System Tablespace, and is used as a safety measure for a crash or any other event that causes the system to cease working. [MySk]

The structure of InnoDB is shown in figure 12. The In-Memory Structure consists of four parts. The largest one is the Buffer Pool which resides in main memory and is where tables are cached and index data obtained. The benefit of this Buffer Pool is that it allows for data that is used often to be processed directly from main memory which reduces processing time. According to the MySQL website, if there are dedicated servers, "up to 80% percent of physical memory is often assigned to the buffer pool."[MySb] This means that processing can be increased by utilizing the Buffer Pool to its full extent. The Buffer Pool is created as a linked list of pages and each page can hold one or several rows. It uses a variation of the Least Recent Used (LRU) algorithm to remove unused data. A figure of how this algorithm works is shown in Figure 13. When a new page is added to the Buffer Pool, it removes least recently used page which is the page at the end of the Old Sublist. The new page is inserted in the middle of the list, between the Old and the New Sublist and if the page is accessed it is moved into the New Sublist. By default the Buffer Pool List is divided as to give 5/8 to the New Sublist and 3/8 to the Old Sublist. [MySb]

The Change Buffer handles the pages that does not reside in the Buffer Pool and therefore has its changes cached to secondary index pages. These changes are buffered and may result from DML operations. When these pages later are loaded into the Buffer Pool the changes are merged. Using Change Buffer allows InnoDB to avoid random access I/O that would have been required to read

secondary index pages from disk. There is a purge operation that periodically writes updated index pages to disk and this happens when the system is mainly idle or during a slow shutdown. This operation speeds up writing to disk in contrast to if the system where to write data to disk immediately after a DML operation. [MySc]

InnoDB does not use hash indexes, but it does use Adaptive Hash Index internally. This feature is used when the system has the suitable combinations of workload and enough memory for the Buffer Pool, without jeopardizing transactional features or reliability. By using these Adaptive Hash Indexes, InnoDB can perform more like an in-memory database. They are created by using a prefix of the index key, which can be of any length. Since only a prefix of the index key is used, only some values in the B-tree may appear in the hash index. In order to work more like an in-memory database only the pages that are in the New Sublist in the Buffer Pool can be used when creating an index. If an entire table fits into main memory these hash indexes can speed up execution of queries by allowing direct lookup of any element and using the index as a pointer. This features does not need to be set by the user as InnoDB has mechanisms that survey index searches and will automatically use hash indexes if it notices that it can speed up queries. [MySa]

The last part of the in-memory structure of InnoDB is the Log Buffer. Data that is to be written to log files on disk resides in this buffer. The Log Buffer flushes its content to disk in a periodically manner. [MySm] This buffer is a useful tool when using many DML operations because it permits "large transactions to run without the need to write redo log data to disk before the transactions commit." [MySm]

On the disk side, InnoDB has several systems for handling tables and a redo log as shown in Figure 12. When creating tables in InnoDB, primary keys has to be defined that are not null, unique and never or very rarely changed after being inserted. This value is used by the most important queries.[MySe] InnoDB uses two types of indexes, clustered and secondary. Clustered indexes are just another name for the primary key and InnoDB uses this index to optimize DML operations and most common lookups. If a primary key is not set for a table, InnoDB will choose the first value that is not null and unique. If this does not exist, it will generate a hidden clustered index which creates a synthetic column with row IDs. These IDs are used to order the rows and for lookups. Using these clustered indexes, will speed up queries by being able to access the page with all data directly. Secondary indexes are indexes that is not a clustered index. Hence, all records in a secondary index has the primary key columns for the row in addition to the columns assigned for the secondary index.[MySd]

The indexes are structured using B-trees, a popular data structure for databases. B-trees are continuously sorted and makes it possible to do fast lookups for exact matches and ranges. B-trees are used for all indexes except the spacial indexes that uses R-trees due to its specialty for indexing multi-dimensional data. InnoDB will try to keep 1/16 of the page free when inserting new records to a clustered index. This is easy to do when records are inserted sequentially, but it gets harder if insertions happens at random which could lead to pages

being 1/2 to 15/16 parts full. InnoDB uses sorted index build, which means that it loads data in bulks when it creates or rebuilds B-tree indexes. [MySq]

One of the major parts of the physical structure in InnoDB is the system tablespace which stores the Data Dictionary, Doublewrite Buffer, Change Buffer and Undo Logs. By default there is one system tablespace data file, but it can have more which can be defined during startup. [MySg] The second part in the On-Disk structure is the File-Per-Table Tablespace which retain data and indexes for a single InnoDB table and "is stored on the file system in its own data file." [MySg] Table and index data can reside in the system tablespace if the tables are created there, in stead of in the file-per-table or general tablespace. The System Tablespace consists of four parts and the first one is the InnoDB Data Dictionary which consists of internal system tables which retains metadata that is utilized to monitor tables, indexes and table columns. [MySj] The second part is the Doublewrite Buffer. When pages are flushed from the Buffer Pool, InnoDB first writes pages to the Doublewrite Buffer before it writes pages to the correct position in the data files. The Doublewrite Buffer is also used to find a copy of a page in case of a crash recovery. It writes data in large consecutive chunks to avoid large I/O overhead or doubling the amount of I/O operations.[MySf] The third part of the System Tablespace is the Change Buffer. The last part is the Undo Logs. They are usually created and held in System Tablespace, but the system storage can be better utilized by creating a separate Undo Tablespace.

The general tablespaces is shared in InnoDB and is created when a tablespace is created. They can store multiple tables, but has the advantage to keep tablespace metadata in memory wile the tablespace is running. Additionally, by keeping several tables in a few general tablespaces, it is possible to use less memory for the metadata than if tables where in different file-per-table tablespaces. It has many of the same capabilities as the file-per-table tablespace and can therefore be wise to use in some cases to speed up processing. [MySh]

On-disk structure also has Undo Tablespaces which holds Undo Logs. Undo Logs consists of Undo Log records that retain information about how to undo the latest change. This is done by performing a transaction to a clustered index record. The Undo Logs resides within Undo Log segments, which again are retained in rollback segments. The Undo tablespace does not exist by default as Undo Logs are usually stored in the system tablespace, but by using this structure one can have undo tablespace in SSD storage while having the remaining system tablespace on hard disk.[MySr] The last tablespace is the Temporary tablespace which contains non-compressed, user-created tables and on-disk internal temporary tables. It is a shared temporary tablespace, and therefore no cost due to performance connected to creating and removing a file-per-table tablespace for every temporary table. Additionally, since there is a temporary table-space there is no need to store temporary table metadata in InnoDB system tables. [MySp]

The last component in the On-disk structure is the Redo Log which is a disk-based data structure and is utilized when a crash has occurred and recovery needed. The most common procedure is that redo logs encodes the requests

made to alter table data, that derive from SQL statements or low-level API-calls. If some alterations did not finish due to a crash in the system, data files are rerun automatically when the system is initialized and before new connections can be accepted. [MySo]

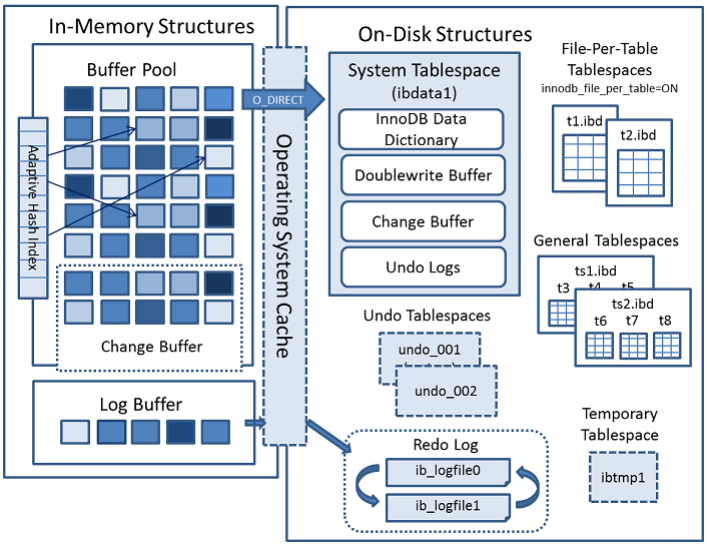


Figure 12: InnoDB Structure [MySi]

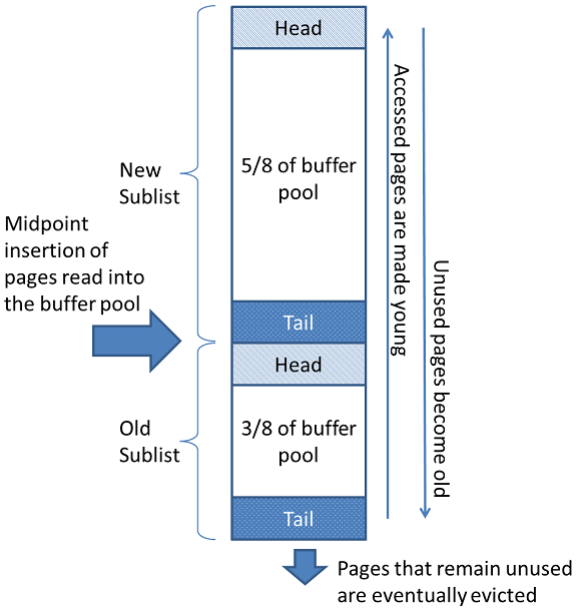


Figure 13: InnoDB Buffer Pool Structure [MySb]

5.1.2 NTNU Student Server

The server chosen for this project was the NTNU Student Server. This is a remote server for students where one can create their own private databases or multi-access databases. The reason for choosing this server was that there was difficulties downloading a local community server and a server with more functionality than the community server was desirable. Since this server was remote some adjustments had to be done to the benchmarking due to network delays when sending the data. Fortunately, the interface provided satisfactory statistics for the network delay, giving the opportunity to eliminate this factor when comparing execution time. The server used was version 5.7.29-0ubuntu 0.16.04.1 and was connected to by using PhpMyAdmin in a browser.

5.1.3 Graphical User Interface: PhpMyAdmin

This graphical user interface has become a frequently used interface for working with a MySQL server and was recommended to use when working with the NTNU MySQL server. It is easy to use and provides features for monitoring, running queries and handle data. Using profiling mechanism in PhpMyAdmin version 4.5.4.1deb2ubuntu2.1, shown in Figure 14, it was possible to determine the seconds spent sending data, which could be subtracted from the execution time. PhpMyAdmin shows all databases created for that user and it was easy to switch between them when querying or handling data. However, this interface

could not be used for crating new databases and setting permissions, this was done by connecting to the server using SSH through X-Win.

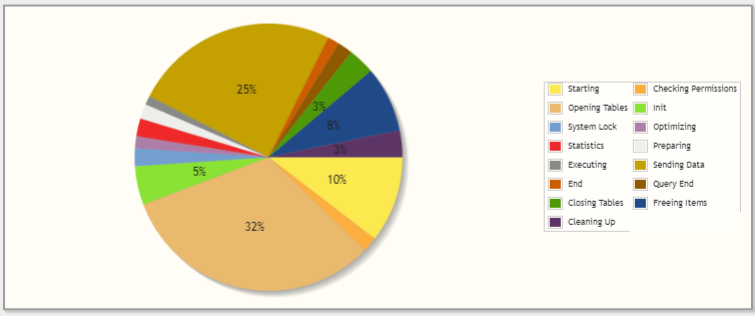
Profiling

Detailed profile

Order	State	Time
1	Starting	29 µs
2	Checking Permissions	5 µs
3	Opening Tables	89 µs
4	Init	13 µs
5	System Lock	6 µs
6	Optimizing	4 µs
7	Statistics	6 µs
8	Preparing	5 µs
9	Executing	3 µs
10	Sending Data	69 µs
11	End	4 µs
12	Query End	5 µs
13	Closing Tables	9 µs
14	Freeing Items	22 µs
15	Cleaning Up	9 µs

Summary by state

State	Total Time	% Time	Calls	ø Time
Opening Tables	89 µs	32.01%	1	89 µs
Sending Data	69 µs	24.82%	1	69 µs
Starting	29 µs	10.43%	1	29 µs
Freeing Items	22 µs	7.91%	1	22 µs
Init	13 µs	4.68%	1	13 µs
Closing Tables	9 µs	3.24%	1	9 µs
Cleaning Up	9 µs	3.24%	1	9 µs
System Lock	6 µs	2.16%	1	6 µs
Statistics	6 µs	2.16%	1	6 µs
Checking Permissions	5 µs	1.80%	1	5 µs
Preparing	5 µs	1.80%	1	5 µs
Query End	5 µs	1.80%	1	5 µs
Optimizing	4 µs	1.44%	1	4 µs
End	4 µs	1.44%	1	4 µs
Executing	3 µs	1.08%	1	3 µs



Showing rows 0 - 24 (383 total, Query took 0.0012 seconds.)

Figure 14: Profiling in PhpMyAdmin

5.2 Neo4j

Neo4j was chosen as the database system to compare MySQL with due to two reasons. It is already used by Sportradar for some use cases and there was an interest in investigating it further as a substitute for relational databases. It was also compared with other commonly used graph database system in the project work done by the author [For19], where Neo4j came out as the best system to use for this particular case.

Neo4j is the leading graph database system according to their website. [Neod] It is a native graph storage system and "is the only enterprise-strength graph database that combines native graph storage, scalable architecture optimized for speed, and ACID compliance to ensure predictability of relationship-based queries." [Neod] It uses its own developed query language, Cypher for mani-pulating data and has many users world wide, both enterprise and community. [Neod]

5.2.1 Native Graph Storage

Native Graph Storage is made in order to be able to handle graph data better than the relational model. This means that the storage is built in a way that optimizes for graphs and makes sure that nodes and relationships are written adjacent on disk. Graph databases can be based on a non-native graph storage, but this will lead to poorer performance as the storage is not optimized for it, having nodes and relation-ships stored far off from each other on disk.

Native graph processing includes storage and queries that optimize the graph structure and obtains optimal performance of the graph database. What separates native graph storage from other common storage engines used, as InnoDB described in Section 5.1.1, is that it does not use indexes. It uses index-free adjacency. This means that when writing data, the process is made faster by making sure that the node is stored directly near its neighboring nodes and relationships. Index-free adjacency removes the need for indexes when reading because it retrieves only adjacent nodes and relationships. Since there is no need for indexes there is a high need for ACID writes. This ensures a higher data integrity than in other NoSQL databases. When a relationship record is to be inserted into the graph it cannot only insert the record, but updates to the graph at both its end, thus at two nodes must occur. If one of these operations fails, the graph will be corrupted. This is only possible to avoid in total by using fully ACID compliant transactions. [Cha]

Neo4j's architecture is shown in Figure 16. This lays the foundation for native graph storage where data is kept in store files made up of data for a particular part of the graph, being nodes, relationships, labels or properties. Since storage is divided in such a manner different responsibilities are set for each part. This helps separating the graph structure from property data and makes it easy to perform graph traversals. [RWE15b]

The structure of a store file record is depicted in Figure 16 and shows how a node and a relationship is stored on disk. Every node store is of a fixed-size and

every record is nine bytes long. Since they are fixed size it is possible to perform rapid lookups for nodes. The first byte in the file is called the in-use flag and tells the system whether this record is in use or not. The next four bytes is the ID of the first relationship linked to the node and the next four is the ID of the nodes first property. There are five bytes for labels which are used for pointing to the label store for this particular node. One extra byte is used for flags. There are several types of flags, one is used to identify if nodes are connected to many other nodes. The rest of the space in the store file is left to be used in the future.The relationships are also stored in a store file of fixed-size records. These records reside the IDs of nodes at both ends of the relationship, several pointers, one to the relationship type and others for next and previous relationships of the nodes at both ends of this relationship. It also has a flag which states whether this relationship is the first one in a so called relationship chain. Figure 17 shows the physical storage in Neo4j. [RWE15b] The use of record IDs in a pointer-like way and fixed-size records makes it possible to implement traversals which can be performed at high speed. Traversals is done by tracing these pointers trough the data structure. In order to traverse a relationship from one node to another one, the system executes ID computations that can be done cheaply in contrast to using global indexes which would have had to be done in a non-native database. From the first node connected with the particular relationship, the first record in the relationship chain is found by computing its offset into the relationship store which leads straight to the correct record. Using this relationship record, it is possible to find the ID of the second node by looking into the second node field in this record. This ID leads to the right node record. Constraints according to relationship type or label can be added by using lookups in the relationship type store or label store, again using the corresponding pointer. Nodes and relationships also have properties stored as key-value pairs in a property store file and can be referenced from both nodes and relationship store files.[RWE15b]

Properties are physically stored and are records of fixed-size. Depending on the size of the property it is stored in a dynamic store or inlined value. Inline values are preferable as it provides faster lookups, but if the properties are too large it does not fit in an inlined value. Here either a dynamic string store or array store is used. These dynamic records will contain linked lists of fixed-size records and can therefore take up more than one dynamic record.[RWE15b]

To increase performance, having the entire graph in main memory is preferable, but as graphs get larger this is not possible in most cases. Therefore Neo4j uses in-memory caching to boost the performance of the database. The cache used in Neo4j is an Least Recently Used-K page cache. This cache divides the stores into discrete regions and keeps a fixed number of regions for each store file. Evictions are based on Last Frequently Used (LFU) cache policy, with a variation based on page popularity. If a page becomes unpopular it will be removed to let more popular pages in, even if the more popular pages has not been accessed lately.[RWE15b]

Another important part of the database is ensuring that it is dependable. It needs to be able to access data when needed and recover from potential crashes. In relational databases it is highly common that they are fully ACID, but for

many graph databases this is not always the case. Neo4j however, ensures that it is fully ACID by being a transactional system. Transactions in graph databases are by definition the same as traditional transactions, except that it handles nodes and relationships. Transactions are implemented by representing each transaction as an in-memory object supported by a lock manager. The lock manager gives write locks to nodes and relationships when they are either created, updated or deleted. When trans-action roll-back occurs the object is abandoned and the write locks released. If this is successful the changes will be committed to disk. This is done by a write ahead log, where changes are added to an active transaction log. When a transaction commit is called, it will flush the log's content to disk. Only then will the changes be applied to the graph and all write locks connected to the transaction will be released. Recoverability is handled by checking the transaction log and replaying transactions if they are in the log. [RWE15b]

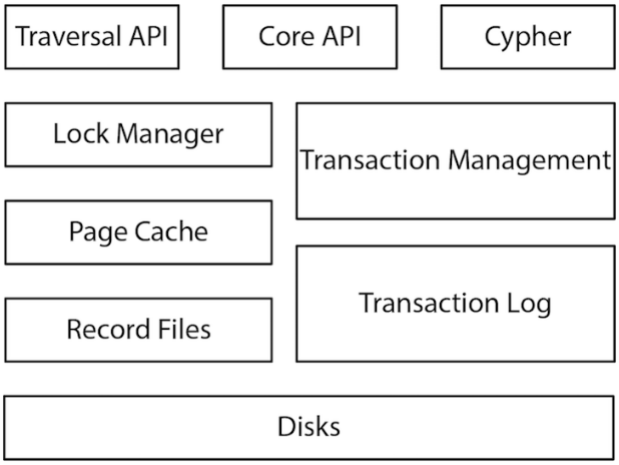


Figure 15: Neo4j Architecture [Cha]

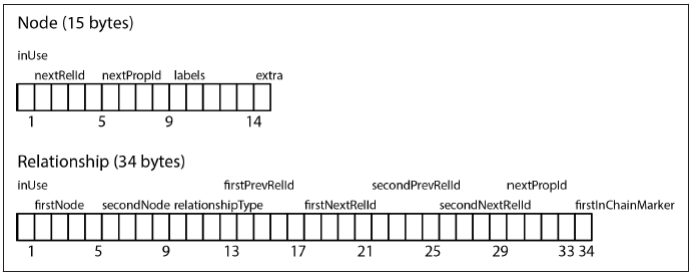


Figure 16: Neo4j Store File Record Structure [RWE15b]

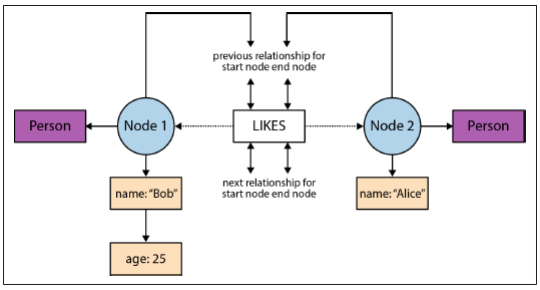


Figure 17: Neo4j Physical Storage [RWE15b]

5.2.2 Neo4j Desktop

The Desktop version used for this study comes with different licenses, but for this project a license for on local database was used. This gave access to the enterprise version which provided more features than the community one. But as this was a local database, it was mostly features for profiling that was of use to this project. Neo4j Desktop Version 3.5.6 was downloaded to a stationary computer from Norwegian University of Science and Technology (NTNU) with Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz processor and 32GB RAM.

5.2.3 Neo4j Browser

The Neo4j Browser is a graphical interface used for handling data and querying in Neo4j. The browser is available from Neo4j Desktop and can be started after a database is running. The browser provides features for querying, profiling and other features for handling data. Bookmarks for queries can be created and it displays the result of the queries as either both a graph and a table or only tables if the result cannot be represented as a graph.

5.3 Data Import

The data used for the project came from Sportradar's web API for Soccer. The version used was Soccer v3 for Premier League 19/20 and the JSON-files retrieved was related to the tournament, matches and players respectively. A list of JSON-files retrieved is provided in Section 9 Appendix.

5.3.1 MySQL

In order to import large amounts of data to all MySQL databases, several methods in Python converted JSON-files to tuples to be inserted into the database. One method retrieved data from JSON-files from Sportradars web API and the second method connected to the database and inserted each element in a set created in the first method. An example of how this was done is shown in Listing 1.

```
def get_teams(url):
    response = requests.get(url)
    data = response.json()
    teams = set()
    for event in data["sport_events"]:
        for team in event["competitors"]:
            id = team["id"]
            name = team["name"]
            val = (id, name)
            teams.add(val)
    return teams


def insert_team(team_set):
    query = "INSERT INTO Team (team_id, name) VALUES (%s, %s)"
    try:
        db = mysql.connector.connect(host="mysql.stud.ntnu.no",
    user=user, passwd=password, database=db)
        cursor = db.cursor()
        cursor.executemany(query, team_set)
        db.commit()
    except Error as e:
        print('Error:', e)
    finally:
        print(cursor.rowcount, "records inserted")
        cursor.close()
        db.close()
```

Listing 1: Python MySQL import

5.3.2 Neo4j

Neo4j gives the user the option to add a library called APOC. APOC stands for Awesome Procedures On Cypher and consists of many procedures and functions to make handling data easier in Neo4j. [Neob] It is mostly used for data integration, graph algorithms and data conversion. The APOC library brings more functionality to the user and makes it easier to work with the data. This project uses APOC for data import from Sportradar's web API, regarding a converter from a JSON-file, defines what data should be imported as nodes and where the edges should be created. APOC makes the import intuitive and it is effective for one JSON-file. However, when many JSON-files needs to be loaded in sequence, there is not a obvious way to do this in APOC for Neo4j version 3.5.6. Therefore, a Python script was created for this task in order to run the APOC function for several URLs in a loop. The code used for this is shown in Listing 2 and the query part of the code shows how APOC is used for retrieving data from each JSON-file.

```
def add_statistics(player_list):
    tournament_name = "Premier League 19/20"
    for player in player_list:
        query = "call apoc.load.json($url) yield value" \
                " unwind value.statistics.seasons as s" \
                " unwind value.player as pl" \
                " MATCH (p:Player {player_id: pl.id})" \
                " MATCH (t:Tournament {name: s.name})" \
                " WITH * WHERE s.name = $name" \
                " MERGE (p)-[:HAS_STATS {team_name: s.team.name," \
                "matches_played: s.statistics.matches_played, " \
                "substituted_in: s.statistics.substituted_in, "\
                "substituted_out: s.statistics.substituted_out, " \
                "goals_scored: s.statistics.goals_scored, "\
                "assists: s.statistics.assists, " \
                "own_goals: s.statistics.own_goals, "\
                "yellow_cards: s.statistics.yellow_cards, " \
                "yellow_red_cards: s.statistics.yellow_red_cards, "
    \
                "red_cards: s.statistics.red_cards}]->(t)"
        session.run(query, parameters={
            "url": url_part1 + player +
                   url_part2, "name": tournament_name})
```

Listing 2: Python Neo4j Import

5.4 Database Structure

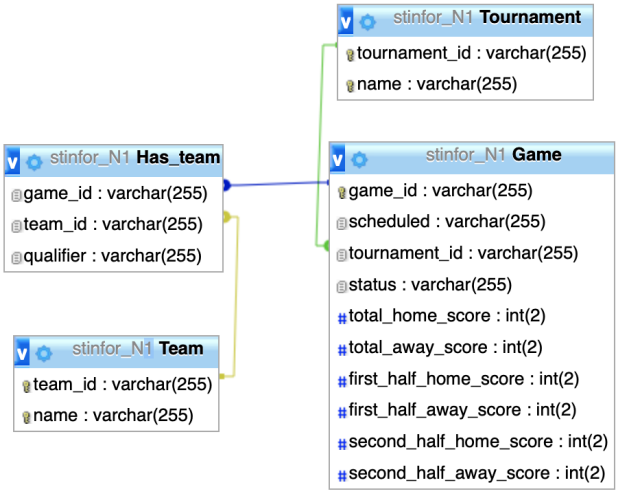
In order to test different queries on different depths and sizes of a database, three databases was created for this study. This sections describes each database and shows their structure.

5.4.1 Database 1

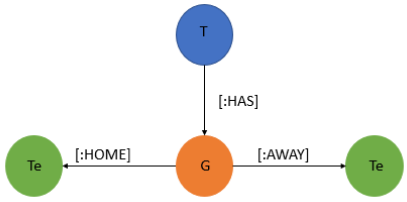
Database 1, was the simplest database with the least amount of data. It consisted of one Tournament instance which was Premier League 19/20, all the games played this season and the home and away teams for each game.

The MySQL structure had a **tournament** table with an id and a name. The **game** table had an id as primary key, tournament id as foreign key and attributes stating when the game was scheduled and the status of the tournament, being either closed or postponed. It also had attributes for the scores of each team in the first and second half, and the final scores for each team. To link the games to home and away-teams, a **has game** table was used that had a foreign key to the game id and the team id in addition to an attribute stating whether it was a home or away-team in that particular game. The **team** table consists of a team id and the name of the team. The table structure is shown in Figure 18a.

The Neo4j structure had a **tournament** node with an id and a name like the MySQL structure, it had an edge directed to a **game** node named **HAS** to represent the relationship between tournament and game. The **game** node had the attributes id, scheduled and status and the scores like described for MySQL. It had the edges **HOME** and **AWAY** directed to a **team** node. This relationship was used to represent whether this team was the home or away-team for this game. The **team** node had the same attributes as in MySQL. The graph structure is shown in Figure 18b.



(a) Table structure 1 MySQL from PhpMyAdmin



(b) Graph Structure 1 Neo4j

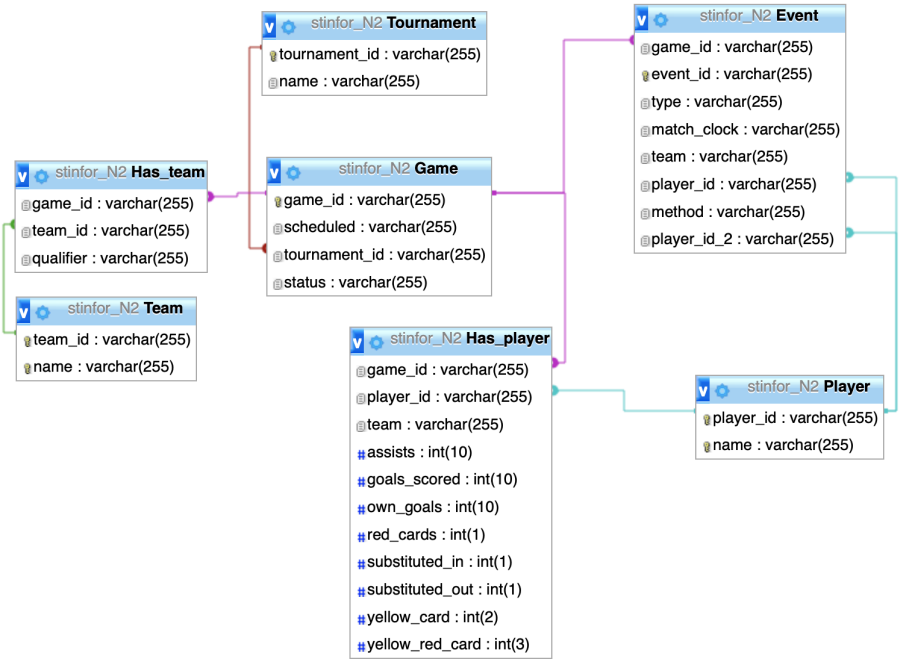
Figure 18: Structure Database 1

5.4.2 Database 2

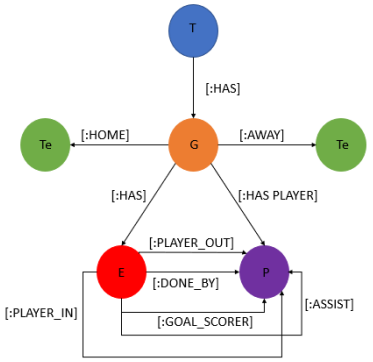
The second database had the same basic foundation as described above with tournament, games and teams. This database had more data added to it for more complex querying.

The MySQL structure is shown in Figure 19a. A **event** table was added with a primary key id and foreign keys game id and two player ids. The first player id represents the primary player for this event or the only player related to the event, depending on the nature of the event. For example if the event was a corner kick, only the first attribute was set as it was the only player kicking. If the event was a substitution the first player would be the player in, the second player attribute would be the player out. The remaining attributes are type, match clock, team and method. Here team was represented as home or away to represent which of the participating teams who's event this was. A **player** table was also created, this was linked directly to a **event** tabled as described above and linked to a game through a **has player** table which had the foreign keys game id and player id. This table also contains the attributes stating the team the player plays for being home or away and summary of events related to the player, for example goals scored, assists and yellow cards. The **player** table itself contains a primary key player id and an attribute name.

For Neo4j the basic structure was also identical to the first database and the graph structure is shown in Figure 19b. This structure also had a **event** and **player** node added with the same attributes as in MySQL. The edges created was **HAS EVENT** and **HAS PLAYER**, the first one between **game** and **event**, the second one between **game** and **player**. The **HAS PLAYER** edge contains the same attributes as in the **has player** table in MySQL. The difference for Neo4j was that it had several edges from **event** to **player**. These edges are **GOAL SCORER, ASSIST, PLAYER IN, PLAYER OUT**, and represent the context of which the player was related to that particular event.



(a) Table structure 2 MySQL from PhpMyAdmin



(b) Graph Structure 2 Neo4j

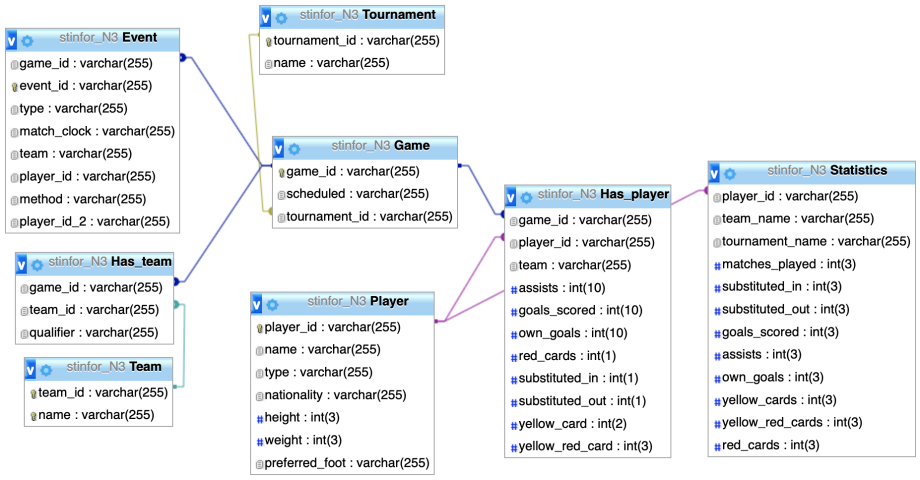
Figure 19: Structure Database 2

5.4.3 Database 3

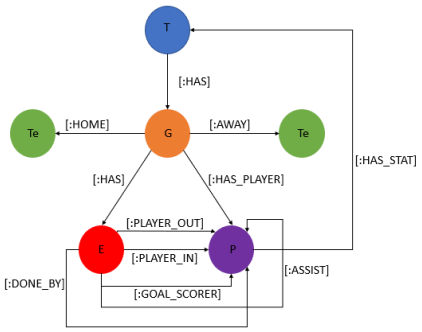
The final and third database had the same structure as the second one, but an additional **statistics** element was added for more query options. The structures for both databases is shown in figure 20.

For MySQL the **statistics** table was added having a foreign key to the player id from the **player** table. The attributes for this table was a summary of all events related to the player for the entire season. One player could have more than one table related to it if the player had changed teams during the season.

For Neo4j the statistics where represented as a an edge between the **player** node and the **tournament** node. This edge had the same attributes as described for the **statistics** table in MySQL.



(a) Table structure 3 MySQL from PhpMyAdmin



(b) Graph Structure 2 Neo4j

Figure 20: Structure Database 3

5.5 Query Languages

There are several query languages that can be used when working with both relational databases and graph databases. SQL is the most common basis for query languages for relational databases, but are adapted to the system they are used for. For graph databases, several different query languages exist as well, however a standard query language, GQL, is in the progress of being made, but is not yet commonly used. This section introduces the query languages used for this project.

5.5.1 SQL

SQL stands for Structured Query language and is the most common language for querying databases. There are many different versions of the language, but it is a ANSI and ISO standard and hence it has to support the main commands such as SELECT, WHERE and INSERT. This study used MySQL and therefore their version of SQL is presented. The execution of the queries was done in PhpMyAdmin as described in Section 5.1.3.

SQL syntax

In order to be able to write queries in SQL there has to be a database with tables to query. Tables has rows and columns where the columns are the attributes and the rows are the records.[W3Ca] When writing a SQL query it is the records that are the result, the SELECT statement defines the columns to be retrieved and the FROM statement declares which table or tables to get them from.

The SQL language is not case sensitive so there is no difference in writing the commands in upper or lower case letters. Additionally, some SQL version requires a semi-colon behind the statements.[W3Cb] This has to be done when writing MySQL queries in a shell, however in PhpMyAdmin used in this case study it is not necessary.

Listing 3 shows a basic SQL query in MySQL using SELECT, FROM, WHERE and ORDER BY. Which commands used depends on the query, but every query needs to have a SELECT and FROM command stating what is to be retrieved and where it should be retrieved from.

```
1 SELECT * FROM A
2 WHERE A.attribute = value
3 ORDER BY A.other_attribute
```

Listing 3: Genaral SQL

5.5.2 Cypher

Cypher is the query language used in Neo4j. It is open source and supports operations like insert, update and delete. One of the main goals for the language is that it should be simple to learn and use, even for user with little or no experience with coding or databases. [Neoa]

Cypher is a declarative language that is inspired by SQL and in order to achieve the previous mentioned goal, it strives to be as visual and logical as possible when querying the database. This is done by letting the user write a query in the natural way one would think nodes are related, as shown in Figure 21. [Neoa]

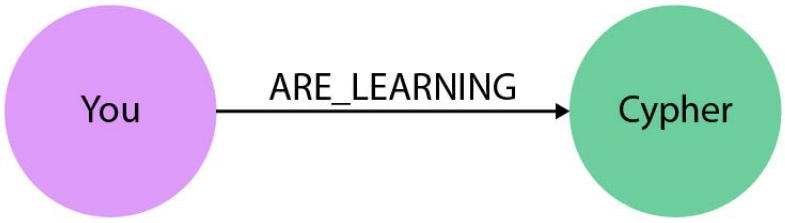


Figure 21: Cypher Query Visually [Neoa]

Neo4j provides a developer guide for new users to learn Cypher which takes the user through the basics step by step.

Cypher Syntax
The main part of the Cypher syntax is to handle and query data as nodes and relationships. Nodes are represented with a label that represents the nodes type. Attributes can be added to the nodes to define differences between the nodes, for example the node Player has a property name, and player id, to identify the player. Attributes are used to show the individuality between the nodes and used when querying the graph. The relationships between the nodes are represented with an arrow that points in the direction the relationship goes. Relationships can in some cases be bidirectional, this is not possible to show with Cypher, but as it is bidirectional, one can choose which direction to set the relationship. Attributes can also be added to the relationship, providing more detailed information about the relationship. [Neoa]

5.6 Queries

The queries are divided into two categories. The first examine the performance of each database in regards to three different depths. The second category wants to obtain information related to soccer using aggregations. The questions that is to be answered using queries are listed in Table 1 and 2.

Depth	Question
1	Get all games for the tournament
2	Get all events of type corner kick for all games in the tournament
3	Get all players that have done a corner kick for all games in the tournament

Table 1: Questions by depth

Database	Query	Question
1	1	Find average of goals scored in first and second half at home for Manchester United
1	2	Which teams have Manchester United won over at home this season?
2	1	How many goals has Manchester United scored for each game at home this season? Sort the result after number of goals scored
2	2	How many goals has each player playing for Manchester United scored this season?
3	1	Count how many players that has left, right or both feet as preferred foot for each team
3	2	How many goals has each player scored this season and how many shots on target do they have?

Table 2: Questions for each database

5.6.1 Depth

Three different levels of depth was queried for this case study. The queries were written to find all information regarding the nodes/tables for each level, hence a lot of data was retrieved. All queries for this category was performed in Database 2. The queries are shown in Listing 4 and 5

```
//Depth 1
MATCH (t:Tournament)-->(g:Game)
RETURN t,g

//Depth 2
MATCH (t:Tournament)-->(g:Game)-->(e:Event {type="corner_kick"})
RETURN t,g,e

//Depth 3
MATCH (t:Tournament)-->(g:Game)-->(e:Event {type="corner_kick"})
    -->(p:Player)
RETURN t,g,e,p
```

Listing 4: Cypher Queries Depth

```
#Depth 1
SELECT * FROM Game INNER JOIN Tournament on Tournament.
    tournament_id = Game.tournament_id
LIMIT 500

#Depth 2
SELECT * FROM Event INNER JOIN Game on Game.game_id = Event.game_id
INNER JOIN Tournament on Tournament.tournament_id = Game.
    tournament_id
WHERE Event.type = "corner_kick"
LIMIT 10000

#Depth 3
SELECT * FROM Event INNER JOIN Game on Game.game_id = Event.game_id
INNER JOIN Tournament on Tournament.tournament_id = Game.
    tournament_id
INNER JOIN Player on Player.player_id = Event.player_id
WHERE Event.type = "corner_kick"
LIMIT 10000
```

Listing 5: SQL Queries Depth

5.6.2 Soccer

Two queries for each database was made with the intention to find out more about the soccer tournament using aggregations and subqueries. As these queries are written to acquire information about the tournament they are not written to explicitly measure execution time. However, they are used for comparing execution time as they indicate performance in common queries for Sportradar and would indicate if one was to prefer over the other for common use. The queries for the two different systems is shown in Listing 6 and 7.

```
//Database 1
//Query 1
MATCH (g:Game { status: "closed" })-[r: HOME]->(t:Team {name: "
    Manchester United"})
RETURN avg(g.first_period_home_score) as avg_first_period, avg(g.
    second_period_home_score) as avg_second_period

//Query 2
MATCH (te)<-[:AWAY]-(g:Game {status: "closed"})-[r: HOME]->(t:Team
    {name: "Manchester United"})
WHERE g.final_home_score > g.final_away_score
RETURN te.name


//Database 2
//Query 1
MATCH (e:Event {team: "home", type: "score_change"})<-[:HAS_EVENT-(
    g:Game)-[:HOME]->(t:team{name: "Manchester United"})
WHERE toInt(split(e.match_clock,":")[0] <= 45
RETURN g.id, e.type, count(e.type)

//Query 2
MATCH (p)<--[r:GOAL_SCORER]-(e:Event)<-[:HAS_EVENT]-(g:Game)-->(t:
    Team {name: "Manchester United"})
RETURN p.name as player_name, count(r) as goals_scored
ORDER BY goals_scored DESC


//Database 3
//Query 1
MATCH (p:Player)-[r:HAS_STATS]->(t)
WHERE p.preferred_foot IS NOT NULL
RETURN r.team_name, p.preferred_foot, count(p.preferred_foot)
ORDER BY r.team_name

//Query 2
MATCH (e:Event {type: "shots_on_target"})-[DONE_BY]->(p)-[r:
    HAS_STATS]->(t)
RETURN p.name as player_name, count(e) as shots_on_target, r.
    goals_scored as goals_scored
```

Listing 6: Cypher Queries for Soccer

```
#Database 1
#Query 1
SELECT avg(g.first_half_home_score) as avg_first_home, avg(g.
    second_half_home_score) as avg_second_home
FROM Game g
    INNER JOIN Has_team ht ON g.game_id = ht.game_id
    INNER JOIN Team t ON ht.team_id  = t.team_id
WHERE ht.qualifier = 'home' and t.name = 'Manchester United' and g.
    status = 'closed'

#Query 2
SELECT t.name FROM Game g
    INNER JOIN Has_team ht ON g.game_id = ht.game_id
    INNER JOIN Team t ON ht.team_id = t.team_id
    WHERE g.game_id IN
        (SELECT g.game_id FROM Game g
            INNER JOIN Has_team ht ON g.game_id = ht.game_id
            INNER JOIN Team t ON ht.team_id = t.team_id
            WHERE ht.qualifier = 'home' and t.name = 'Manchester
    United' AND g.status = 'closed' AND g.total_home_score > g.
    total_away_score)
    AND ht.qualifier = "away"


#Database 2
#Query 1
SELECT  e.game_id, e.type as type, COUNT(e.type) as
    number_of_occurences
FROM Event e WHERE e.game_id IN
    (SELECT g.game_id FROM Game g INNER JOIN Has_team ht ON g.
    game_id = ht.game_id
    INNER JOIN Team t ON ht.team_id = t.team_id
    WHERE ht.qualifier = 'home' and t.name = 'Manchester United'
    and g.status = 'closed') and e.team = 'home' and (CONVERT(
    SUBSTRING(e.match_clock,1), SIGNED INTEGER) <= 45)
    AND e.type = 'score_change'
GROUP by e.game_id

#Query 2
SELECT p.name as player_name, count(p.name) as goals_scored FROM
    Game g
    INNER JOIN Has_team ht on ht.game_id = g.game_id
    INNER JOIN Team te on te.team_id = ht.team_id
    INNER JOIN Event e on e.game_id = g.game_id
    INNER JOIN Player p on p.player_id = e.player_id
WHERE e.type = 'score_change' and te.name = 'Manchester United'
GROUP BY player_name
ORDER BY goals_scored DESC


#Database 3
#Query 1
SELECT s.team_name as team_name, count(p.preferred_foot),
    preferred_foot
FROM Player p
    INNER JOIN Statistics s on s.player_id = p.player_id where
    preferred_foot IS NOT NULL
```

```
47 GROUP BY preferred_foot, team_name
48 ORDER BY team_name
49
50 #Query 2
51 SELECT pl.name, s.goals_scored as goals_scored, shots_on_target
       from Statistics s        INNER JOIN (SELECT p.player_id as
       player_id, count(e.type) as shots_on_target
52     FROM Player p INNER JOIN Event e on e.player_id = p.player_id
53     WHERE e.type = "shot_on_target" GROUP BY p.player_id) as A
54     ON A.player_id = s.player_id
55     INNER JOIN Player pl on pl.player_id = s.player_id
```

Listing 7: SQL Queries for Soccer

6 Results

The results for this study came from running the queries in 25 iterations through the graphical interfaces for each of the database technologies. The execution time presented in the results, that will be used for the discussion, are execution times without considering the time it takes to present the data. In the case of the MySQL Server, the time it takes to send the data was also not considered when comparing the results. However, the time it takes to send data is presented in the result to give a complete overview of how the execution time was calculated.

A T-test was performed for all query execution times in order to determine if one of the two database solutions have results that are dissimilar enough to state that one was better than the other. This T-test was a paired two sample for means. This was chosen to be able to compare each iteration, taking into account the mean value for all iterations. The results are shown in Table 6

6.1 Results from Depth Queries

This section presents the results from the Depth Queries for each query and depth. The results are shown in table 3, 4, 5 and as graphs in figure 22, 23, 24.

Iteration	Depth	Execution Time MySQL (ms)	Sending Data MySQL (ms)	Final Execution Time MySQL (ms)	Execution Time Neo4j (ms)
1	1	3,6	0,361	3,239	62,484
2	1	5,5	1,100	4,400	0,000
3	1	3,5	0,359	3,141	0,000
4	1	2,8	0,365	2,435	15,625
5	1	2,8	0,349	2,451	0,000
6	1	3,5	0,865	2,635	15,652
7	1	3,7	1,000	2,700	0,000
8	1	4,2	1,000	3,200	5,196
9	1	3,6	0,375	3,225	0,000
10	1	1,7	0,360	1,340	0,000
11	1	3,1	0,357	2,743	0,000
12	1	3,7	0,341	3,359	15,626
13	1	3,7	0,353	3,347	0,000
14	1	3,6	0,369	3,231	0,000
15	1	3,3	0,353	2,947	0,000
16	1	4,3	1,100	3,200	0,000
17	1	3,5	0,348	3,152	0,000
18	1	3,4	0,355	3,045	0,000
19	1	5,5	1,100	4,400	0,000
20	1	4,2	0,340	3,860	15,621
21	1	3,5	0,339	3,161	0,000
22	1	3,5	0,330	3,170	0,000
23	1	3,1	0,339	2,761	0,000
24	1	3,6	0,358	3,242	0,000
25	1	3,8	0,349	3,451	0,000

Table 3: Results Depth 1

Iteration	Depth	Execution Time MySQL (ms)	Sending Data MySQL (ms)	Final Execution Time MySQL (ms)	Execution Time Neo4j (ms)
1	2	24,2	21,400	2,800	93,729
2	2	31,1	23,100	8,000	0,000
3	2	31,9	21,600	10,300	0,000
4	2	33,0	22,700	10,300	15,624
5	2	27,7	21,700	6,000	0,000
6	2	31,8	21,800	10,000	0,000
7	2	25,9	22,000	3,900	0,000
8	2	33,2	22,000	11,200	14,123
9	2	32,1	21,800	10,300	0,000
10	2	29,5	22,100	7,400	0,000
11	2	32,3	22,300	10,000	0,000
12	2	22,5	21,200	1,300	0,000
13	2	30,7	21,500	9,200	0,000
14	2	28,4	21,600	6,800	15,627
15	2	30,3	22,200	8,100	0,000
16	2	23,2	21,600	1,600	0,000
17	2	27,5	20,900	6,600	0,000
18	2	31,1	21,400	9,700	15,621
19	2	23,2	22,100	1,100	0,000
20	2	22,5	21,100	1,400	0,000
21	2	30,3	22,100	8,200	0,000
22	2	31,1	22,400	8,700	15,625
23	2	25,1	21,100	4,000	0,000
24	2	27,3	21,400	5,900	0,000
25	2	29,9	21,100	8,800	0,000

Table 4: Results Depth 2

Iteration	Depth	Execution Time MySQL (ms)	Sending Data MySQL (ms)	Final Execution Time MySQL (ms)	Execution Time Neo4j (ms)
1	3	26,5	25,200	1,300	62,479
2	3	29,2	27,500	1,700	6,850
3	3	31,5	24,700	6,800	0,000
4	3	29,0	24,100	4,900	0,000
5	3	30,4	25,900	4,500	0,000
6	3	29,4	25,200	4,200	15,627
7	3	34,5	27,900	6,600	0,000
8	3	30,7	24,700	6,000	15,956
9	3	25,4	24,600	0,800	0,999
10	3	27,8	24,400	3,400	7,049
11	3	26,1	24,900	1,200	0,000
12	3	29,7	24,600	5,100	0,000
13	3	27,4	25,000	2,400	0,000
14	3	42,1	24,800	17,300	15,625
15	3	30,8	24,800	6,000	0,000
16	3	33,7	24,500	9,200	15,621
17	3	25,6	24,700	0,900	0,000
18	3	30,5	24,000	6,500	0,000
19	3	26,3	24,600	1,700	0,000
20	3	26,0	24,239	1,761	0,000
21	3	25,8	24,157	1,643	0,000
22	3	32,0	24,075	7,925	15,622
23	3	27,6	23,993	3,607	0,000
24	3	25,9	23,911	1,989	0,000
25	3	30,6	23,829	6,771	0,000

Table 5: Results Depth 3

Query	Average MySQL (ms)	Average Neo4j (ms)	T-test (P=t) two-tail
1	3,113	5,208	0,438
2	6,864	6,814	0,99003
3	4,568	6,233	0,543

Table 6: Average Execution Time and T-test

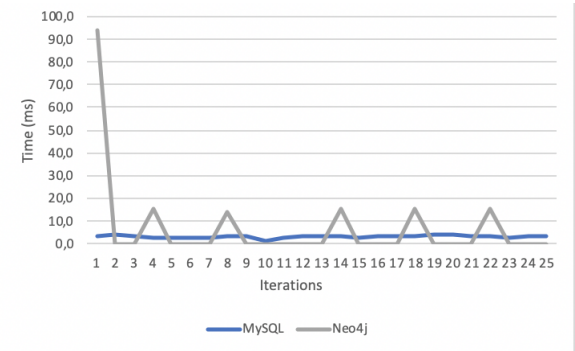


Figure 22: Graph Depth 1

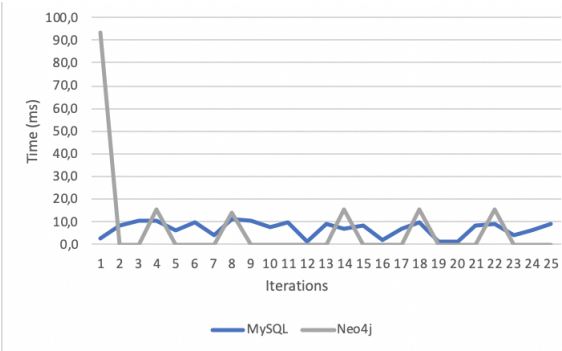


Figure 23: Graph Depth 2

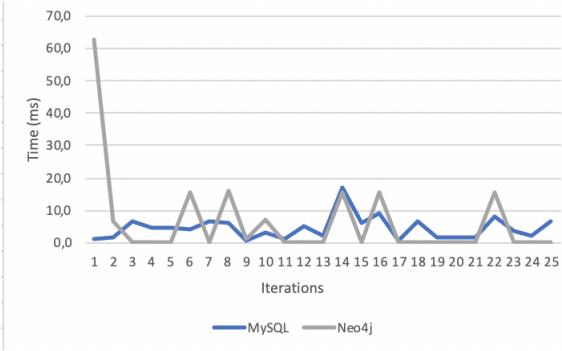


Figure 24: Graph Depth 3

6.2 Results from Soccer Queries

There was performed two queries per database related to soccer. These are relevant queries for Sportradar, and are used in this case study to establish whether either of the systems should be preferred over the other. The structure for all databases both MySQL and Neo4j are shown in Figure 18, 19 and 20 and the queries are shown in Listing 6 and 7.

6.2.1 Database 1

This section presents the results done for the first database.

Iteration	Execution Time MySQL (ms)	Sending Data MySQL (ms)	Final Execution Time MySQL (ms)	Execution Time Neo4j (ms)
1	3,200	0,383	2,817	265,6
2	1,100	0,395	0,705	0,000
3	1,900	0,424	1,476	0,000
4	1,500	0,375	1,125	0,000
5	1,500	0,386	1,114	0,000
6	1,400	0,385	1,015	0,000
7	1,400	0,370	1,030	15,59
8	1,600	0,366	1,234	0,000
9	1,500	0,403	1,097	0,000
10	1,400	0,383	1,017	0,000
11	1,600	0,379	1,221	0,000
12	1,400	0,363	1,037	0,000
13	1,400	0,367	1,033	0,000
14	1,400	0,396	1,004	0,000
15	1,500	0,366	1,134	15,62
16	1,600	0,365	1,235	0,000
17	1,700	0,394	1,306	0,000
18	1,600	0,401	1,199	0,000
19	1,500	0,367	1,133	0,000
20	1,800	0,410	1,390	0,000
21	1,700	0,374	1,326	0,000
22	1,400	0,378	1,022	0,000
23	1,600	0,396	1,204	0,000
24	1,400	0,365	1,035	0,000
25	1,500	0,367	1,133	0,000

Table 7: Results Database 1 Query 1

Iteration	Execution Time MySQL (ms)	Sending Data MySQL (ms)	Final Execution Time MySQL (ms)	Execution Time Neo4j (ms)
1	10,70	9,000	1,700	131,5
2	3,200	1,300	1,900	0
3	3,200	1,400	1,800	15,62
4	3,100	1,300	1,800	0
5	3,200	1,400	1,800	0
6	3,000	1,300	1,700	0
7	3,000	1,300	1,700	6,505
8	3,200	1,400	1,800	0
9	2,900	1,400	1,500	0
10	2,800	1,300	1,500	0
11	3,000	1,300	1,700	0
12	3,100	1,400	1,700	0
13	2,800	1,300	1,500	0
14	3,200	1,300	1,900	0
15	3,200	1,300	1,900	0
16	3,200	1,300	1,900	0
17	2,900	1,300	1,600	0
18	2,800	1,300	1,500	15,63
19	2,800	1,300	1,500	0
20	3,000	1,300	1,700	0
21	2,900	1,300	1,600	0
22	3,200	1,400	1,800	0
23	3,000	1,300	1,700	0
24	2,800	1,300	1,500	0
25	2,300	1,400	0,9000	0

Table 8: Results Database 1 Query 2

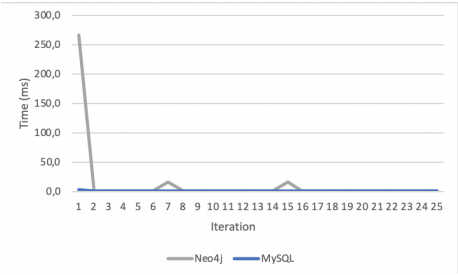


Figure 25: Database 1 Query 1

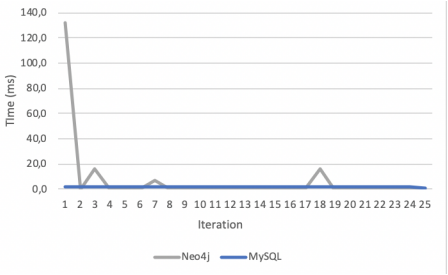


Figure 26: Database 1 Query 2

Query	Average MySQL (ms)	Average Neo4j (ms)	T-test (P=t) two-tail
1	1,202	11,87	0,3215
2	1,664	6,770	0,3423

Table 9: Database 1 Average Execution Time and T-test

6.2.2 Database 2

This sections presents the results for Database 2.

Iteration	Execution Time MySQL (ms)	Sending Data MySQL (ms)	Final Execution Time MySQL (ms)	Execution Time Neo4j (ms)
1	16,400	14,000	2,400	359,7
2	15,500	13,800	1,700	0
3	18,000	13,900	4,100	21,66
4	15,900	14,000	1,900	0
5	15,600	13,900	1,700	0
6	16,200	14,400	1,800	0
7	16,200	14,500	1,700	30,18
8	16,200	14,200	2,000	0
9	16,400	14,100	2,300	0
10	16,700	13,900	2,800	0
11	15,300	13,800	1,500	0
12	15,700	14,000	1,700	8,052
13	15,600	14,000	1,600	0
14	15,700	14,100	1,600	0
15	15,500	13,900	1,600	42,32
16	15,800	14,100	1,700	0
17	15,700	13,800	1,900	0
18	15,400	13,800	1,600	0
19	15,600	13,900	1,700	0
20	15,700	13,900	1,800	0
21	15,700	14,100	1,600	0
22	16,100	14,400	1,700	0
23	15,700	14,000	1,700	0
24	15,800	13,800	2,000	8,016
25	15,700	13,800	1,900	0

Table 10: Results Database 2 Query 1

Iteration	Execution Time MySQL (ms)	Sending Data MySQL (ms)	Final Execution Time MySQL (ms)	Execution Time Neo4j (ms)
1	26,600	24,900	1,700	78,10
2	18,200	16,200	2,000	0
3	18,600	15,800	2,800	15,65
4	17,800	16,100	1,700	0
5	17,500	16,100	1,400	0
6	17,800	16,100	1,700	0
7	17,400	15,900	1,500	15,59
8	17,400	15,700	1,700	0
9	17,600	16,000	1,600	0
10	17,300	15,900	1,400	0
11	18,900	16,100	2,800	0
12	17,200	15,800	1,400	15,62
13	17,400	16,000	1,400	0
14	17,500	15,900	1,600	0
15	17,400	15,900	1,500	15,62
16	17,700	16,000	1,700	0
17	17,200	15,800	1,400	0
18	17,400	15,900	1,500	0
19	17,200	15,800	1,400	0
20	17,700	16,000	1,700	0
21	17,500	15,900	1,600	0
22	17,400	15,900	1,500	0
23	17,800	16,100	1,700	0
24	17,400	15,900	1,500	14,44
25	18,200	15,900	2,300	0

Table 11: Results Database 2 Query 2

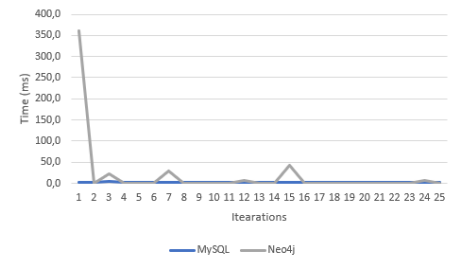


Figure 27: Database 2 Query 1

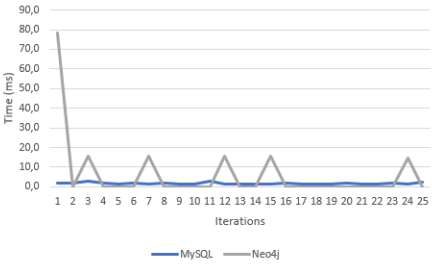


Figure 28: Database 2 Query 2

Query	Average MySQL (ms)	Average Neo4j (ms)	T-test (P=t) two-tail
1	1,92	18,80	0,251
2	1,7	6,201	0,18

Table 12: Database 2 Average Execution Time and T-test

6.2.3 Database 3

This section presents the results for Database 3.

Iteration	Execution Time MySQL (ms)	Sending Data MySQL (ms)	Final Execution Time MySQL (ms)	Exectuion Time Neo4j (ms)
1	2,300	1,200	1,100	242,03
2	1,800	1,000	0,800	10,05
3	2,500	1,000	1,500	30,16
4	1,800	1,000	0,800	0
5	2,100	1,000	1,100	0
6	1,900	1,000	0,900	0
7	3,300	1,000	2,300	30,15
8	2,800	1,000	1,800	0
9	2,200	0,997	1,203	0
10	1,900	1,000	0,900	0
11	2,100	1,000	1,100	0
12	1,700	1,100	0,600	10,09
13	3,100	1,000	2,100	0
14	2,000	1,000	1,000	0
15	2,000	1,000	1,000	28,11
16	2,200	1,000	1,200	2,014
17	2,000	1,000	1,000	0
18	3,100	1,000	2,100	0
19	2,100	1,000	1,100	0
20	2,200	1,000	1,200	0
21	2,600	1,000	1,600	0
22	2,400	0,991	1,409	0
23	10,600	1,000	9,600	0
24	2,700	1,000	1,700	0
25	3,100	1,000	2,100	0

Table 13: Results Database 3 Query 1

Iteration	Execution Time MySQL (ms)	Sending Data MySQL (ms)	Final Execution Time MySQL (ms)	Execution Time Neo4j (ms)
1	15,700	13,733	1,967	140,2
2	15,700	13,659	2,041	15,65
3	15,400	13,945	1,455	31,24
4	15,100	13,744	1,356	8,939
5	15,700	13,748	1,952	0
6	15,700	13,841	1,859	0
7	16,000	13,741	2,259	0
8	15,700	13,739	1,961	0
9	15,800	13,94	1,860	31,25
10	17,200	13,743	3,457	0
11	16,300	13,648	2,652	0
12	15,700	13,74	1,960	0
13	15,100	13,64	1,460	0
14	16,100	13,746	2,354	15,65
15	15,600	13,651	1,949	0
16	15,600	13,738	1,862	0
17	17,600	13,94	3,660	0
18	17,500	13,649	3,851	15,62
19	15,600	13,646	1,954	0
20	15,400	13,746	1,654	0
21	15,700	13,735	1,965	0
22	15,300	13,74	1,560	15,62
23	16,100	14,14	1,960	0
24	16,400	14,052	2,348	0
25	15,800	13,852	1,948	0

Table 14: Results Database 3 Query 2

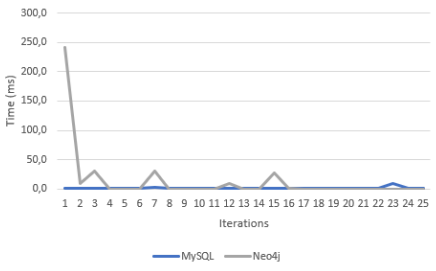


Figure 29: Database 3 Query 1

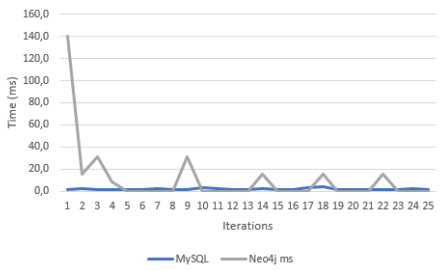


Figure 30: Database 3 Query 2

Query	Average MySQL (ms)	Average Neo4j (ms)	T-test (P=t) two-tail
1	1,649	14,10	0,2127
2	2,132	10,97	0,1367

Table 15: Database 3 Average Execution Time and T-test

7 Discussion

This case study examines how well each system performs on soccer data and tries to indicate whether either of the systems should be the preferred choice over the other for storing and querying data. This section will discuss the systems chosen for this study, the type of server used and the results from the benchmarking process.

7.1 Database Systems

This study aimed to see if there was a system that performed better for sports data and if so how much better. The reason this study compared MySQL and Neo4j was that the system mostly used in Sportradar today is MySQL, but they have started using Neo4j for some use cases and was interested in looking into whether there are significant differences in performance of the two systems. Choosing Neo4j was also based on the study performed by the author as presented in section 3. This study compared Neo4j to several other graph databases using sports data and found that Neo4j was the preferred system. [For19] Relational Databases has been the most used system for decades and this is not without reason. It is a well defined database solution with many different systems based on it, MySQL being one of the most used ones. This study shows that it can be used in a wide variety of ways, based on the users background and knowledge and it can be used with different systems on top. This system used PhpMyAdmin as the graphical interface which was very easy to understand and operate. This is beneficial for MySQL, that the user can choose between different system and deploy the system that works best for its purposes. Neo4j is a Graph Database which has grown in popularity after social networks became an interesting subject to study and analyze. Graph Databases has for cases where data is heavily connected a better visual model and can hence be easier to visualize and work with for the right applications. However, it has the same structure on disk, but it's the way data is stored in this tables and how they are accessed that are different. Neo4j uses native graph storage which is beneficial storing deeply connected data close to each other on disk, reducing accessing time. Sportradar employees with programming background and knowledge about query languages and databases, might have enough skill to use shell-solutions for both technologies to handle data import, export and alterations. However, displaying the results in a way that is easy to understand

for customers that may not have the same knowledge can be challenging. In this case the graph model is easier to understand when looking at results and this is also the case for soccer data.

Summing up the points discussed in this section, MySQL is the natural choice based on amount of users and stability over many years. Neo4j, being a newer system, does not have the advantage of being improved over many years and can in some cases be more unstable. Since this case study looks at the application for soccer data, the graph model fits well with how the data is connected. This brings many benefits, being able to build the graph as the tournament continues, show data in a neat visual way and only query certain parts of the graph. Looking at these factors Neo4j has a few more advantages than MySQL and can be a good choice when working with this kind of data.

7.2 Servers

This study compared an external server for MySQL and a local server for Neo4j. The reason for choosing external MySQL server was due to unforeseen challenges downloading the local community server and the external server had features that was useful for the benchmarking process. Following this change in server, the hypotheses was that the external MySQL server should perform better than the local Neo4j server as it most likely had better main memory and a more powerful Central Processing Unit (CPU) than the local machine from Norwegian University of Science and Technology (NTNU). The external server was the NTNU server for students and was a MySQL server with more functionality than the local community server and therefore it was easier to go into more detail of how queries was processed. This was helpful when looking into time spent sending data over the internet and gave the opportunity to eliminate this factor when comparing execution time of queries. The Neo4j local server used, was the enterprise version of their server for one local machine that gives more options for tuning the database than the community version. By looking at the servers and their main memory and CPU capabilities, MySQL should perform perform better having more advantages in the means of computational power.

Due to the fact that MySQL used an external server for this project, data had to be sent over the internet to be displayed. Download time, as shown in the results, was not a part of the comparison. However, it has an impact for the user. It could take several minutes to view large results in a graphical interface, making it impractical to use when handling queries with thousands of rows. This was also the case with a local server, as displaying results takes a lot of time, even though it was shorter when the data was local. This should be considered when choosing a database system handling large amounts of data. The drawbacks sending data can be substantial, but in this case few queries analyzing soccer data results in thousands of records. Data transmission time can in this case therefore be ignored. If the server is not in the control of the company, security can be an issue and therefore must be considered. This could be of interest when working with sports data as it has great value for analyzing. However, if several people are to access the same database it is impractical

to have a local single user database. Sportradar handles millions of data and need to have multi-user databases. Thus, might benefit from having an external server. The server could be in-house to eliminate security risks. On the other hand, a local database can be used for smaller investigations of the data, hence both servers could be beneficial for Sportradar for different use cases.

7.3 Results

MySQL and Neo4j had pros and cons in relation to working with soccer data. For this study MySQL had the advantage of a more powerful server which leads to the assumption that it would outperform Neo4j. However, as the structure of the data was beneficial for the graph database, Neo4j may not be as inferior to MySQL. This section discuss the results from the two different benchmarking processes, the first being queries by depth, and the second soccer queries.

7.3.1 Queries by depth

This study compared the two systems in regards of processing time of different kinds of queries. The first part handled queries that tested the systems with different levels of depth. One of the points made by Neo4j is that it is faster when querying a graph of "friends of friends", or "friends of friends of friends" and this section intended to test if this applied to the structure for soccer data. Since MySQL had a more powerful server than Neo4j, it was expected that MySQL would perform better. At the same time, benefits of the graph structure would give Neo4j advantages to compete with the execution time in MySQL. Looking at the results for the query executed for depth 1, Neo4j had a larger execution time for the first iteration, but decreases drastically for the second iteration. In MySQL however it increases from the first to the second iteration, but not with as large of a variation as Neo4j. Neo4j, likely suffer from larger cold start problems than MySQL as it needs to load data into main memory in the first iteration. Having less computer power this takes longer time than for the MySQL server. However, after the first query it was able to keep data in main memory for the second query and uses much less time the second time. MySQL had an increase in execution time from the first to the second iteration. This could be due to the fact that the server after certain intervals of time flushes data from main memory and data had to be looked up again from disk for the second iteration, since the execution time reduces for the third iteration again. Since MySQL uses an external server, it was not possible to run and terminate it as one would with a local server. Hence, there might be data in the cache from previous queries that are reused for the first iteration. MySQL uses the Least Recently Used (LRU) algorithm which removes the last recently used page when new ones are loaded. Resulting in a higher execution time in the second iteration. Both Neo4j and MySQL had fluctuations in their execution time, this could be due to the use of cache when running queries. Since the iterations are repeated 25 times, data had to be loaded from disk several times, therefore there was variations in execution time. In Neo4j it takes longer to

retrieve data from disk and the execution time increases. However, looking at the results from iterations between points of retrieving data from disk, the execution time was negligible. This might be because data was loaded fully into main memory and using native graph storage, traversing the graph was performed rapidly. Since Neo4j uses Least Frequently Used (LFU) for updating the cache, it removes unpopular pages, keeping statistically popular pages in memory longer. However, in certain iterations it had to load parts of the data into main memory, this could be due to either a interval flush or that the LFU algorithm had removed data. The latter being highly plausible as the intervals between the higher execution times increases. Indicating that the algorithm predicts better which pages are popular. The average execution time for Neo4j was larger than the average execution time for MySQL which was expected. This study wanted to investigate whether either of the systems where significantly better than the other. Therefore a T-test was performed as described in Section 6. Table 6 shows the results for all depths. Interestingly, the T-test shows no significant difference in performance for this depth. MySQL had the advantage of being an external server with more computer power than the Neo4j server based on 7.2. Therefore it might be interesting for a future benchmarking to test an external server for Neo4j, to see if this can improve the execution time for Neo4j.

Depth 2 had more data processed in the query. Neo4j was expected to compete better with MySQL due to the increase in depth based on the "friends of friends" experiment in section 4. The results for MySQL shows that for the the first iteration most of the data was in the InnoDB Buffer Pool and the execution time was low compared to other iterations. Much data seems to be flushed after the first iteration as the second iteration takes much longer time. Since a great amount of data was to be loaded into main memory this execution took longer time than the previous iteration. This query suffered from higher fluctuations than the first one. This is probably due to fact that there were more joins which requires more lookups and even though data is kept in main memory it still had to use indexes to locate the pages resulting in longer execution time. If whole tables could fit into main memory the use of Adaptive Hash Index could speed up query time by permitting direct access to pages, resulting in execution times that were much smaller than others. This method was controlled by InnoDB, resulting in lower execution times appearing sporadically. For Neo4j the cold start problem was still present and was larger than for the query of depth 1 due to larger amount of data being loaded into main memory. The amount of time loading data from disk in later operations did not seem to change from the previous query, indicating that it did not remove more data even though the data amount had increased. Neo4j was able to keep a considerable amount of data in main memory to reduce execution time in between the iteration where data was fetched from disk. Because MySQL spent longer time executing queries at this depth, and Neo4j was able to run queries quite rapidly when not retrieving data from disk, the average execution time for this query was quite similar for both systems, which could also be viewed from the result of the T-test. Hence for this depth they performed almost equal,

again raising the interest for testing Neo4j with a more powerful server to see if it performs better than MySQL.

The query for depth equal to three processed less data, due to the filtering. MySQL did not have the cold start issue in first iteration, seeming to benefit from having data in main memory for the first query. Looking at iteration 14 for MySQL, the server might have had a restart and all data retrieved from disk as the execution time is a lot higher than the rest. The reason for fluctuating results in this part could be due to Adaptive Hash Index. Neo4j had a similar result as in the query for depth one. It had a cold start problem, but running queries faster after loading data into main memory. MySQL was faster in average than Neo4j, but not significantly based on T-test. Suggesting again that Neo4j might be able to perform better if provided with a more powerful server.

7.3.2 Soccer queries

The second part of the test ran queries to find information about the Premier League tournament, its teams and players. For this part, three different databases were used with different structures and sizes. Two queries were written for each database and compared using a T-test. For the first database the first query was a aggregate. It wanted to get the average score for the first and second half for a team, in this case, Manchester United. This involved several joins of tables or traversal of nodes to find the data needed to perform the calculation. MySQL performed all iterations in this query in very similar time, except for the first one where data was loaded into the Buffer Pool. Since MySQL probably managed to keep a large portion of the data in the Buffer Pool, only some data was flushed and retrieved again from disk which resulted in less fluctuation. Neo4j on the other hand had a large value for the first iteration. This could be due to the fact that much data had to be retrieved. It had a low execution time for the following iterations until data had to be loaded from disk again. MySQL did not have any significant cold start issue and did not require much additional time when retrieving data from disk the average value was much lower than for Neo4j. Neo4j, however performed better in all cases where data was not retrieved from disk. The T-test presents no significant differences between the databases. The second query wanted to list all names of teams Manchester United had won over at home this season. This required several steps of filtering and aggregating data and MySQL executed this query rapidly in all iterations without any highly differing values. Neo4j had the same issues as in all previous queries, but again performed fast inn all iterations where data was not retrieved from disk. There was less difference between the average values than the previous query and the T-test again showed no significant difference.

The second database had more data and a more complex structure than the first. This required more joins for MySQL and traversals in the graph for Neo4j. The first query retrieved the amount of goals scored in the first half for each game at home for Manchester United. Neo4j had the longest execution time running first iteration, due to retrieving large amounts of data from disk.

Having more data stored in main memory results in updating the cache more frequent in addition to retrieving parts of data from disk. Since Neo4j uses the LFU algorithm for evictions, there could be different amounts of data evicted from the cache from iteration to iteration, leading to different execution times. Neo4j did not have a dedicated server, meaning that the allocated space for Neo4j's cache could vary from iteration to iteration leading to different amounts of data removed from main memory. The average value was a lot higher for Neo4j than for MySQL, but it was still executing faster when data was in main memory. The T-test showed no significant difference between MySQL and Neo4j for this query. The second query retrieved a count of all goals scored per player that plays for Manchester United. This aggregation required approximately the same amount of data as in query 1 and the results for MySQL were also quite similar as the previous query. However, it had higher fluctuations, indicating that more data had to be retrieved from disk after a few iterations. Iteration one for Neo4j had less execution time than the previous query for this database even though it was still high. This indicates that the first iteration had to collect much data from disk. Neo4j had the advantage of running queries fast when data was in main memory, but disadvantage of fetching data from disk more often. The T-test showed that the databases were different, but not significantly.

Database 3 was an extension of database 2 with an additional table for MySQL and a relationship for Neo4j with statistics for each player for the entire season. The first query for this database retrieved a count of how many right foots, left foots or both feet there was for each team in Premier League. This query targeted only two tables in MySQL leading to the assumption that MySQL had the advantage of few joins in addition to having a better server than Neo4j. Looking at the results for MySQL it showed that it performed this query fast except for iteration 23 where it seamed like it had to retrieve more data from disk. Neo4j, however spent more time executing this query due to the retrieval of data from disk. MySQL had a lower average execution time than Neo4j, but as for all previous queries the difference was not significant. The second query for this database and the last query performed in this study, retrieved a count of shots on target and goals scored for each player to see how many shots actually resulted in a goal. This query joined three tables for MySQL which lead to the assumption that Neo4j should perform well in comparison to MySQL. The results for this query shows that MySQL spent more time executing this query due to more join-operations and had a few iterations that were longer than others where data probably was retrieved from disk. Neo4j performed this query similar to the past one, indicating that it did not suffer particularly from having to execute more traversals in the graph. The iterations with high values, where data was retrieved from disk, was not higher than in the previous query even though it happened more frequent due to more data handled. The difference in average execution time for both databases was less than for the previous query, as MySQL performed poorer and Neo4j a little better. MySQL was as assumed better than Neo4j, but for this query as well it was not a significant difference between the databases.

8 Conclusion

This case study compared MySQL and Neo4j when using soccer data. MySQL had a more powerful basis than Neo4j, as an external server was used providing more computational power, while Neo4j used a local server. The hypothesis was that Neo4j would not perform much worse in comparison due to its graph structure and native graph storage. Since there was a unfair basis for the comparison, it was expected that MySQL was better. In eight out of nine queries MySQL had in average a better performance than Neo4j, but T-tests determining difference, showed that it was no significant difference between the two databases. The results indicated some of the expected behaviors of the database systems. MySQL performed well when there was one join operation, but spent more time as depth and data amount increased. However, MySQL reduced execution time for more joins if the amount of data was reduced. Neo4j spent more time handling queries with more depth and more data, but reduced execution time when data amount was reduced even though depth increased. All these results lead to the conclusion that neither of the systems can be chosen based purely on performance. Factors as user experience and use cases needs to be considered when choosing the system that is best for Sportradar.

8.1 Future Work

This section introduces some interesting views gathered from the case study that could be of interest to investigate further in other case studies.

8.1.1 Servers

This case study used an external server for MySQL and a local server for Neo4j and showed that MySQL had a lower average execution time, but was not significantly better than Neo4j. This indicates that a comparison with an external server for both database systems could show in more extent whether one is significantly better than the other. This case study briefly discusses the use of a local vs. external server. This could be investigated further. Especially, looking into security and different use cases for different types of data used by Sportradar.

9 Appendix

9.1 Data Import

- JSON-files from Sportradar API
  - Tournament schedule
  - Tournament result
  - Match timeline
  - Match summary
  - Player profile

Bibliography

[DS12] Miyuru Dayarathma and Toyotaro Suzumura. *XGDBench: A Benchmarking Platform for Graph Stores in Exascale Clouds*. 2012. DOI: 10.1109/CloudCom.2012.6427516. URL: https://orientdb.com/wp-content/uploads/xgdbench_cloudcom2012.pdf. (accessed: 07.11.19).

[RWE15a] Ian Robbinson, Jim Webber, and Emil Eifrem. *Graph Databases: New oppurtunities for connected data*. OReilly Media, Inc, 2015.

[RWE15b] Ian Robbinson, Jim Webber, and Emil Eifrem. "Graph Databases: New oppurtunities for connected data". In: OReilly Media, Inc, 2015. Chap. 6.

[For19] Stine Forås. *Graph Databases for Sportsdata (Report available from the author)*. 2019.

[Cha] Joy Chao. *Graph Databases for Beginners: Native vs. Non-Native Graph Technology*. URL: https://neo4j.com/blog/native-vs-non-native-graph-technology/. (accessed: 11.05.20).

[D-g] D-graph. *Dgraph Documentation*. URL: https://docs.dgraph.io/. (accessed: 01.12.19).

[db-] db-engines. *DB-Engines Ranking of Graph DBMS*. URL: https://db-engines.com/en/ranking/graph+dbms. (accessed: 02.12.19).

[MySa] MySQL. *Adaptive Hash Index*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-adaptive-hash.html. (accessed: 11.05.20).

[MySb] MySQL. *Buffer Pool*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool.html. (accessed: 11.05.20).

[MySc] MySQL. *Change Buffer*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-change-buffer.html. (accessed: 11.05.20).

[MySd] MySQL. *Clustered and Secondary Indexes*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-index-types.html. (accessed: 11.05.20).

[MySe] MySQL. *Creating InnoDB Tables*. URL: https://dev.mysql.com/doc/refman/5.7/en/using-innodb-tables.html. (accessed: 11.05.20).

[MySf] MySQL. *Doublewrite Buffer*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-doublewrite-buffer.html. (accessed: 11.05.20).

[MySg] MySQL. *File-Per-Table Tablespaces*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-file-per-table-tablespaces.html. (accessed: 11.05.20).

[MySh] MySQL. *General Tablespaces*. URL: https://dev.mysql.com/doc/refman/5.7/en/general-tablespaces.html. (accessed: 11.05.20).

[MySi] MySQL. *InnoDB Architecture*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-architecture.html. (accessed: 11.05.20).

[MySj] MySQL. *InnoDB Data Dictionary*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-data-dictionary.html. (accessed: 11.05.20).

[MySk] MySQL. *InnoDB Disk I/O*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-disk-io.html. (accessed: 11.05.20).

[MySl] MySQL. *Introduction to InnoDB*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-introduction.html. (accessed: 11.05.20).

[MySm] MySQL. *Log Buffer*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-redo-log-buffer.html. (accessed: 11.05.20).

[MySn] MySQL. *MySQL Products*. URL: https://www.mysql.com/products/. (accessed: 10.05.2020).

[MySo] MySQL. *Redo Log*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-redo-log.html. (accessed: 11.05.20).

[MySp] MySQL. *Temporary Tablespaces*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-temporary-tablespace.html. (accessed: 11.05.20).

[MySq] MySQL. *The Physical Structure of an InnoDB Index*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-physical-structure.html. (accessed: 11.05.20).

[MySr] MySQL. *Undo Tablespaces*. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-undo-tablespaces.html. (accessed: 11.05.20).

[Neoa] Neo4j. *Concepts: Relational to Graph*. URL: https://neo4j.com/developer/graph-db-vs-rdbms/. (accessed: 04.12.2019).

[Neob] Neo4j. *Neo4j APOC Library*. URL: https://neo4j.com/developer/neo4j-apoc/. (accessed: 05.06.20).

[Neoc] Neo4j. *Neo4j resources*. URL: https://neo4j.com/whitepapers/graph-technology-buyers-guide/. (accessed: 06.11.19).

[Neod] Neo4j. *Why Neo4j? Top Ten Reasons*. URL: https://neo4j.com/top-ten-reasons/. (accessed: 29.05.20).

[new] Neo4j news. *How much faster is a graph database, really?* URL: https://neo4j.com/news/how-much-faster-is-a-graph-database-really/. (accessed: 02.06.20).

[Ora] Oracle. *What Is a Relational Database?* URL: https://www.oracle.com/database/what-is-a-relational-database/. (accessed: 08.05.2020).

[Oria] OrientDB. *Getting Started*. URL: https://orientdb.org/getting-started. (accessed: 08.11.19).

[Orib] OrientDB. *OrientDB vs Neo4j*. URL: https://orientdb.com/orientdb-vs-neo4j/. (accessed: 03.12.19).

[Oric] OrientDB. *What is a Graph Database?* URL: https://orientdb.com/graph-database/. (accessed: 07.11.19).

[Orid] OrientDB. *Why a Multi-Model Database?* URL: https://orientdb.com/multi-model-database/. (accessed: 07.11.19).

[Orie] OrientDB. *Why OrientDB*. URL: https://orientdb.com/why-orientdb/. (accessed: 07.11.19).

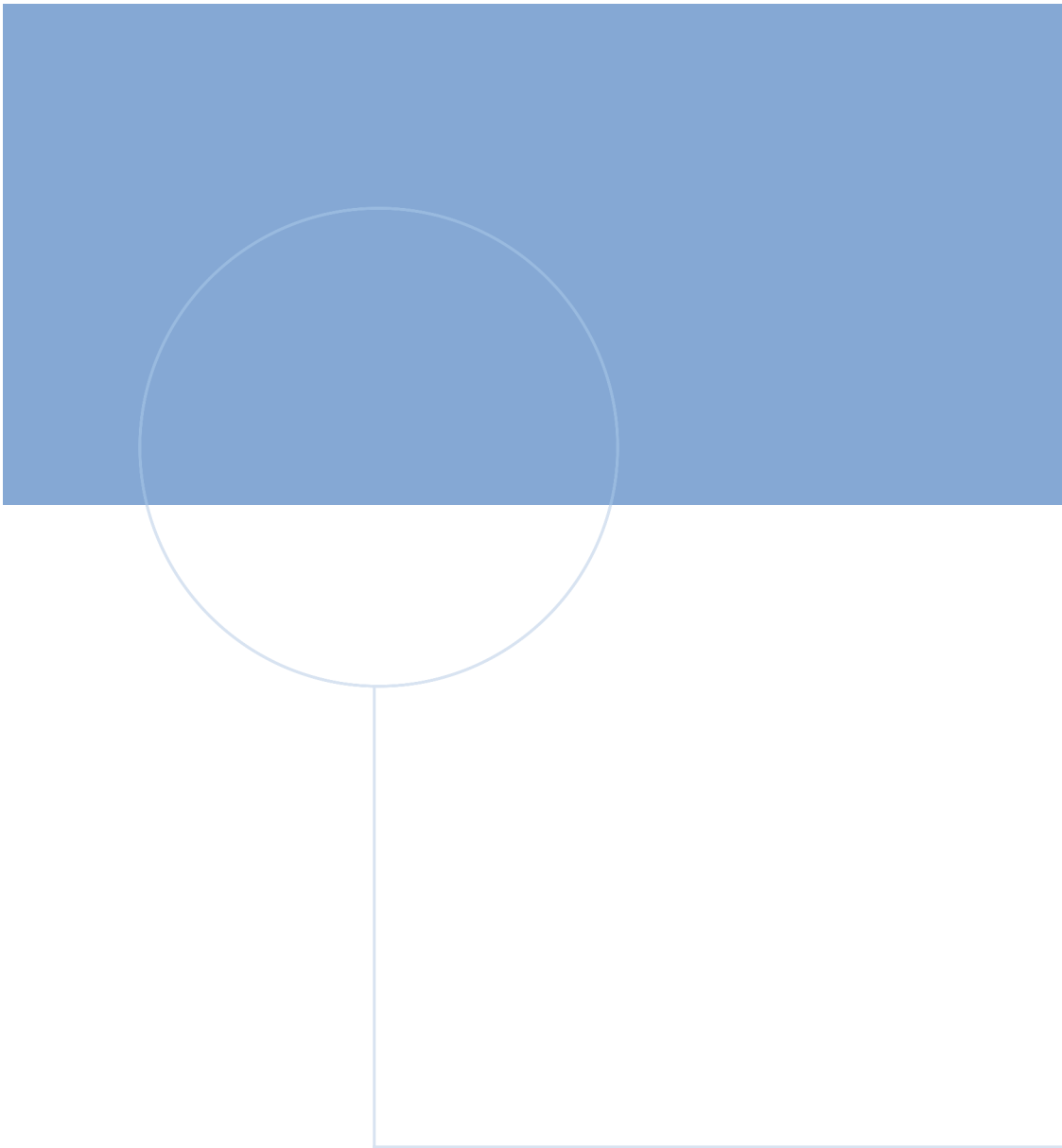
[Rao] Karthic Rao. *Releasing BadgerDB v2.0*. URL: https://blog.dgraph.io/post/releasing-badger-v2/. (accessed: 02.12.19).

[Raw] Pawan Rawal. *Neo4j vs Dgraph - The numbers speak for themselves*. URL: https://blog.dgraph.io/post/benchmark-neo4j/. (accessed: 02.12.19).

[Spo] Sportradar. *Mission & Vision*. URL: https://sportradar.com/about-us/vision/. (accessed: 02.06.20).

[W3Ca] W3C. *Introduction to SQL*. URL: https://www.w3schools.com/sql/sql_intro.asp. (accessed: 11.05.20).

[W3Cb] W3C. *SQL Syntax*. URL: https://www.w3schools.com/sql/sql_syntax.asp. (accessed: 11.05.20).



NTNU

Norwegian University of
Science and Technology