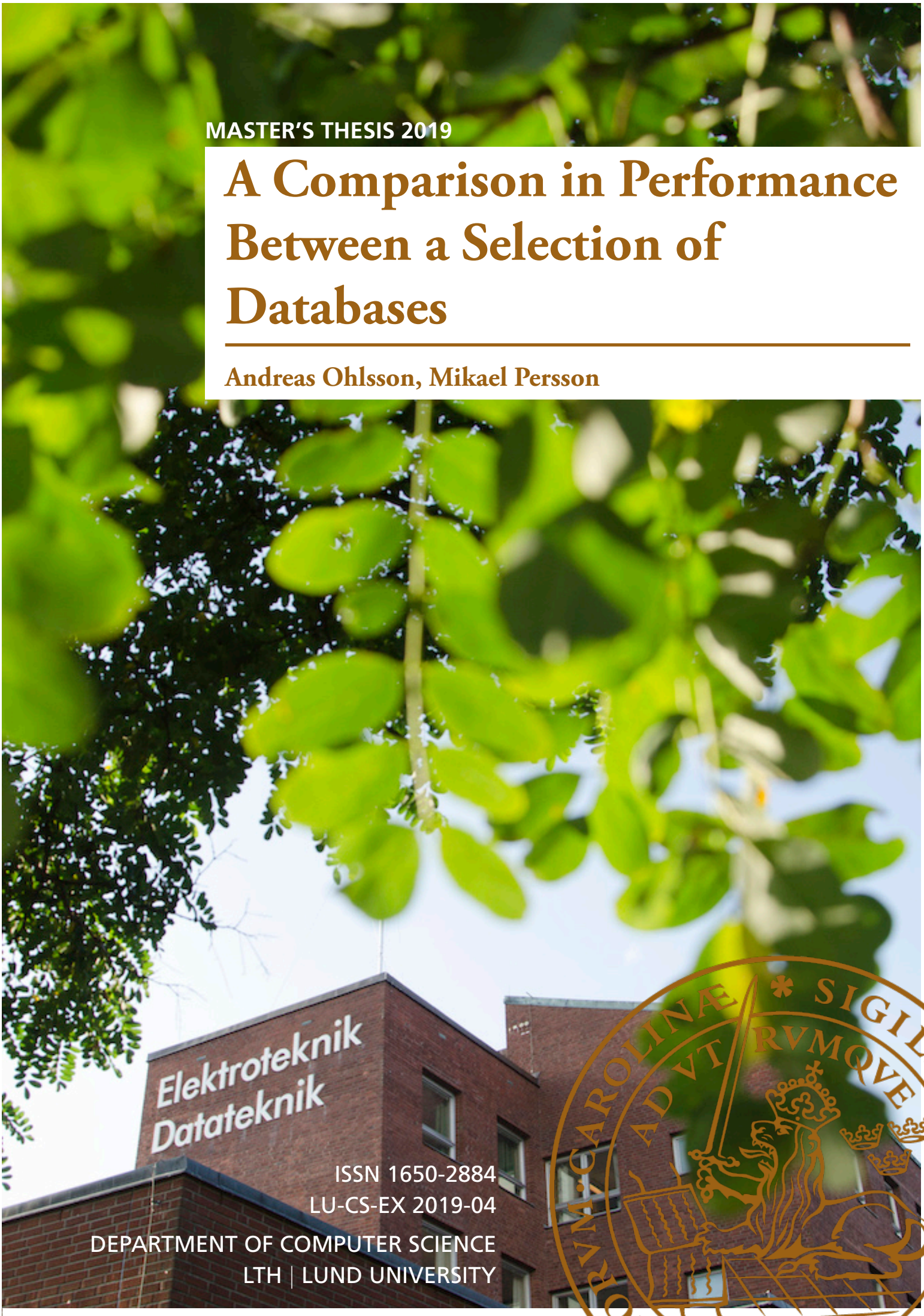


MASTER'S THESIS 2019

A Comparison in Performance Between a Selection of Databases

Andreas Ohlsson, Mikael Persson

ISSN 1650-2884
LU-CS-EX 2019-04

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2019-04

A Comparison in Performance Between a Selection of Databases

Andreas Ohlsson, Mikael Persson

A Comparison in Performance Between a Selection of Databases

(Using the benchmark tool YCSB)

Andreas Ohlsson
dat14aoh@student.lu.se

Mikael Persson
dat14mpe@student.lu.se

May 10, 2019

Master's thesis work carried out at Axis Communications AB.

Supervisors: Fredrik Pihl, fredrik.pihl@axis.com
Jörn Janneck, jorn.janneck@cs.lth.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

Choosing the correct database for an application can be a daunting task for an application owner. There is a vast plethora of different options to choose from, where choosing a NoSQL database or a relational database management system built on SQL is the most defining decision. With the ever-increasing amounts of data that is produced, more and more databases are emerging.

In this report we compare Cassandra and MongoDB, two NoSQL databases, with PostgreSQL, a relational database management system, to see which database is most suited to store semi-structured video metadata. The focus of the benchmarks is on the performance of read operations for each database. The tool we are using for comparison is the popular Yahoo! Cloud Benchmarking System.

The results when running the workloads from the benchmark tool showed that PostgreSQL performed on average better than MongoDB and Cassandra. Therefore, we recommend Axis to use PostgreSQL for storing their video metadata.

Keywords: Databases, NoSQL, SQL, Benchmarking, Metadata

Acknowledgements

We would like to thank our supervisors Fredrik Pihl and Jörn Janneck for their feedback and help throughout this project. Serving as our second supervisor at Axis Communications we would also like to thank Alexandre Martins.

Lastly, we would like to thank our examiner Flavius Gruian and our opponents Anton Danewid and Petter Andersson for their feedback.

Contents

Chapter 1
Introduction

The standard for storing data in computer systems today is with the use of database management systems. The dominating standard among these systems has been relational database management systems, commonly programmed with the SQL language. With the rise of big data, however, different systems have started to take growth and are quite common among today's big companies. These are commonly known as NoSQL databases.

At Axis, a global leading video surveillance company, metadata from the videos are generated several times a day for different purposes. Axis has not used a database system but has instead stored the data in self-composed systems. They realized quickly that this was not sustainable in the long run since extraction of data needed a specified structure, which only one person at Axis knew. A database management system would be ideal for the company. But, since the metadata is vast and varied, research must first be conducted to determine which database is best suited.

In this thesis we have looked at different database management systems, both SQL and NoSQL, and how they store the data. By examining this attribute, which differs from database to database, we could sift through which database suits Axis best. The three databases that we finally decided on were MongoDB, Apache Cassandra and PostgreSQL.

We start by explaining the concepts around database management systems. We will explain the structure of different data, what SQL and NoSQL are, the differences between those two, and how these can be designed. Then we will explain what our methodologies were for conducting the research.

To aid us in deciding the right database system, we used the Yahoo! Cloud Service Benchmarking tool (YCSB) [70] to conduct a benchmark study. The tool is used to benchmark several different types of database systems and is therefore a good choice for this report. The tool has been used in various papers before, with other database systems

Finally, we will give a recommendation for Axis which database to use based on the produced results from the benchmarking study.

Chapter 2
Background

In this chapter we describe the fundamental concepts that this thesis is built upon and aims to describe different types of databases and the theory that comes with it. The following sections start by explaining Axis' current setup, then we describe different categories of data, followed by traditional SQL databases. Finally, we will describe different types of NoSQL databases and compare them to SQL databases.

2.1 Current setup

At Axis, video streams are created every minute for research, testing or production. These video streams contain metadata that describe each frame with several different data types. These data types can vary from what type of frame it is, to peak signal-to-noise ratio (PSNR). The frame types can be either I-frames or P-frames and are denoted with the corresponding alphabetic character. The PSNR number, which is "…the ratio between the maximum possible power of a signal and the power of corrupting noise …" [66], is used in conjunction with YCbCr, which is a kind of color space used in video, and is used to calculate the quality of the current frame. The reader does not need to know exactly what these data types are used for, but it is important to understand that the data stored varies greatly in terms of these data types. Employees at Axis can also add their own data, increasing the variation further. Also, the metadata that was interesting yesterday, might be different today and over time the data that is generated changes. Since the data will vary over time and with the combination of the variation of data in terms of data types, is the take away the variation of the data is quite vast.

The current solution in place at Axis was developed as a filler by one of the employees. The implementation was done by extracting JSON objects from the metadata and parsing these into ordinary text files with no clear thought of usage in mind. Unfortunately, the results were that there was only one person in the department that knew how to use the JSON object to extract the data. This quickly became unsustainable. When someone in the team wanted certain metadata extracted, they needed to go through that employee in the department. The

team then became dependent on this employee to proceed with their work. The best procedure would be for a common solution that everyone in the team knew, so that each employee could get the metadata when it suited that employee. This would increase the work rate.

2.2 Structured data & Non-structured data

To understand databases, it is important to know the difference between common types of data. These are usually categorized in three different kinds of data: structured data, unstructured data and semi-structured data. In this section we will take a closer look at these three categorize and describe common situations where they are found and used.

2.2.1 Structured data

The structured data type is predefined by a schema [55] to which the data conforms to fit the specification [2, p. 14]. The data is therefore predetermined and the relationship between the data is established beforehand [63]. This is done by designing a schema that the resulting database follows. Structured data is one of the most common kinds of data, as it is used in relational database systems [55]. This structure has both its advantages and disadvantages.

The main advantages are what has made it so common. Since the data will not change, unless the schema changes, it will "enable an efficient data processing and an improved storage and navigation of content" [2, p. 122]. The result will be high performance and efficient navigation of the data, which has laid the foundation for the relational database management systems (RDBMS) and the common language SQL that is used to extract and manipulate the database.

The main disadvantages are that a data type still must be present in the structure, even if it is not used for the current application. Also, adding new data types results in updating the whole structure as the schema needs to be redesigned. As the schema gets bigger, the longer it will take for the database to fetch the data [48]. This effectively means that structured data is limited in terms of scalability [55].

Since it is costly to change the schema, it is of great importance that design and plan is setup beforehand. This is usually done by designing an Entity Relationship diagram (ER-diagram). ER-diagrams are used to describe relationships between entities. They can be used for different purposes, but in RDBMS are they used to provide structure and design for the database system before any schemas are put in place [36]. There is no silver bullet in designing the ER-diagram, although there are some design principles that are recommended to follow [67]. This gives us an opportunity to explore what design suites best for the environment we are working with.

2.2.2 Unstructured data

Unstructured data is any form of data stored in an unstructured format, which means that there is no conceptual definition nor any data type definitions [63]. The data can simply not be stored in rows and columns, as in traditional relational databases [13, 55].

Social media posts, tweets and feeds can be seen as unstructured data [52]. The growth of the internet has been a contributing factor in the commonness of unstructured data.

The management of this data cannot be done with the structured data since the data cannot be stored in rows and columns. This has resulted in different solutions, among them Not Only SQL (NoSQL), for storing, processing and retrieval of data [52]. No effort will be needed on the classification, but at the cost of no controlled navigation within the unstructured data [55].

2.2.3 Semi-structured data

The definition of semi-structured data is somewhat conflicted. Examples of semi-structured data by some definitions falls under unstructured data for others [13, 55, 30, 31].

For this master thesis is semi-structured data defined as neither raw data nor strictly constrained, as in structured data [1]. Typical examples would be files in HTML or BibTex format. These files have fields that are not necessary for the format but are a choice by the user. Parts of the semi-structured data may be defined by certain parts of the data, whilst it is still possible that a data instance can have more than one data type [55].

The schema in the structured data can almost be seen as set in stone once it is designed, but for semi-structured data, the schema changes over time, and as such will the schema be flexible [1]. Changes to the schema also needs to be easy and cheap.

Therefore, managing this data with ordinary RDBMS can be challenging as they rely on the schema for their implementation. Using NoSQL databases is simpler in this matter as, more often than not, they do not rely on a schema for their implementation [40, 54].

2.3 Metadata

Metadata is commonly defined as "data about data" [24]. The metadata describes and defines the resource it is linked with. This is an extensive definition of what metadata is, and it seems that the different types of metadata are endless.

What can be said though, is that there are a lot of different forms and categories of metadata. These can be descriptive, structural, administrative, reference, statistical, technical, business, and process metadata. The descriptive and technical metadata are distinguishable from each other. The descriptive metadata is data that is meant to be interpreted by humans, while technical metadata is meant to be interpreted by computer programs [41]. Although there are a lot of different types of metadata, they are typically used for the same thing, to organize and structure the main data.

The metadata can be generated either by automation or by hand. Depending on when it is generated, different data types can be extracted and used for different purposes. For example, an automated process updates the metadata each time a new entity is updated on a website, or a developer can generate metadata from a certain time in a video segment. The format of the metadata can be different depending on the metadata.

2.4 SQL databases

Traditional SQL databases are what is known as Relationship database management systems (RDBMS). The history of RDBMS spans all the way back to the 1970's when Edgar Codd

proposed a new way of representing data, called the relation data model [18]. The development of RDBMS exploded, and by 1980, RDBMS was the dominant type of database. The standard query language (SQL), developed by IBM (the same company Codd worked at), became the standard language for RDBMS in the 1980's [53]. For this reason, RDBMS is also referred to as "SQL databases". In 1981 Codd received the Turing Award for his work [53].

Fast forward to today and the RDBMS is still used. SQL was updated with new versions over the years [16]. One of the main reasons why it has persisted is because it is easy to learn. Donald D. Chamberlin and Ray Boyce, the authors of the language, wanted SQL to be "…simple enough [so] that ordinary people could "walk up and use it"…" [16].

2.4.1 Difference between SQL and self-composed DBMS

One does not need to have a RDBMS or SQL type database setup to have a database of some sort. However, there exist many benefits of having a SQL database. One of the main reasons is in the name of Relational DBMS. The relational properties pave the way for complex queries and look ups. This is useful as the database grows and becomes more complex. Since querying provides more benefits, such as effectively asking the database questions on the data, is it useful when the database grows. In a database over students, the application could simply ask the database: "Give me all students with a GPA of 3.0 or higher". If the database in this instance is well defined and implemented, then the database would spit out an answer in a relatively short amount of time [61].

To achieve this relational property, there is some planning needed and design implementation. This gives the database a structure to it that is predefined, and not implemented ad-hoc. This benefits into a well-designed and easily implemented database. The database can easily be expanded, compared to a self-composed DBMS.

A SQL database can also handle concurrent access, meaning that several users wants to access the database at the same time, and such are ideal in systems where instances of simultaneously access to a database occurs. The article written by Mischook [38] mentions the database driven web pages as a common type of a dynamic type of web page. If the data in the database changes, the web page changes accordingly, for all users of that web page. Since SQL database can handle concurrent access, it is clear to see that this statement is true, and we can draw the conclusion that databases are well suited for the internet.

The biggest benefits are observed when the data is complex and large, as in the case of our thesis. Taking the time to learn a new skill, that of SQL, might be too time consuming and bothersome for a system that would not benefit from it.

2.4.2 Database design

The design process of a relational database is usually comprised of certain well-known steps and processes. A schema is an essential part of a relational database, which is the core of the whole database. The design process of how this should be implemented is usually done by constructing an ER-diagram. In this section we will look at some common practices when designing a database and the different parts the process covers.

ER-diagram

An Entity Relational diagram is "…an approximate description of the data…" [53] and used to give a conceptual model of the database. It provides an overview of the different components of the final database and the relationship between these components, as well as the attributes for each of the components. ER-diagrams remain as the standard for designing database relations [42], although it has been argued that UML-diagrams are more comprehensible than ER-diagrams [12].

Schema

A schema is a description of the data as a data model [53]. It specifies a certain relation between the data entry's name and the attribute related to that data type, as well as what type the attribute is [53]. An example would be a database of guests at a hotel. A schema could be:

```
Guests(ssn: integer, name: string, room: string,
arrivalDate: string, departureDate: string)
```

Each new guest that is added to the database needs to follow this schema to be able to be added. In simple terms the schema is the template for the entries in the database [53]. It can therefore be seen that changing the schema, as also mentioned earlier, will be costly. Therefore, typically a great deal of labor when designing a database is spent on the decision making of how the schema should be structured.

The schema is for the most part an advantage, as it is almost essential in a structured data environment. As the different data grows though, there is a possibility that more management and time is needed on the schema [32].

Columns and rows

Each data entity specified in the schema is implemented as a table, for example the $Guests$ schema in Schema. The tables contain different columns for the different attributes defined in the schema for the data entry, for example ssn in Schema. Each instance of an entry is stored in rows, meaning the values for a certain data entry. So, in the Schema example, could a row be a new guest with values for each column for that specific guest.

The key takeaway is that the tables follow the associated schema for that data type to the point [53]. Should for some reason a data-field be empty or not used, it is still needed in the table. SQL is using null states to fulfil this purpose [65].

An effective aspect of RDBMS are the relationships that can be formed between various tables. These relationships are formed by associating attributes from one table and with attributes from another table. For relationships to be formed there is a need for planning of the attribute's values in the tables, depending on the relationship type should one table contain the other table's attribute.

Take a database for a school in which students attend. The schema for this may look something like this:

```
Students(studentID: integer, courses: integer,
    birthday: integer)
Teachers(teacherID: integer, birthday: integer)
Courses(courseID: integer)
```

This example provides us an overview of the three different relationships that exist: One-to-one, One-to-Many and Many-to-Many. One-to-One exist between the Teachers and Courses tables. If a teacher at the school are only allowed to teach one course, we can say that, "A teacher teaches one course and a course is only ever taught by one teacher". The tables in a One-to-one relationship is commonly concatenated to one single table, containing the attributes of both [69, Ch. 2.2].

The Many-to-Many relationship exist in the example between the students and the courses. We can say that, "A student attends several courses and a course is attended by several students". This relationship type is usually split up with a table between the student and the course [69, Ch. 2.2]. In the example below, a table TakenCourse would exist:

```
TakenCourse(studentID: integer, CourseID: integer)
```

In this example, every TakenCourse table is related to one student and one course, but each course and student have several TakenCourses. This "middleman" gives us the final relationship.

The One-to-Many relationship is third and most common relationship among the three [69, Ch. 2.2]. In our example with TakenCourse, Students and Courses, we have that, "A student can take several courses, but a specific taken course is only registered to one student". Then we have "A course can be taken by several students, but a taken course corresponds to a specific course". These two relationships are powerful and easy to manage, since TakenCourse only need to know which studentID is associated with a specific courseID. In Figure 2.1 we can see an overview of these two relationships.

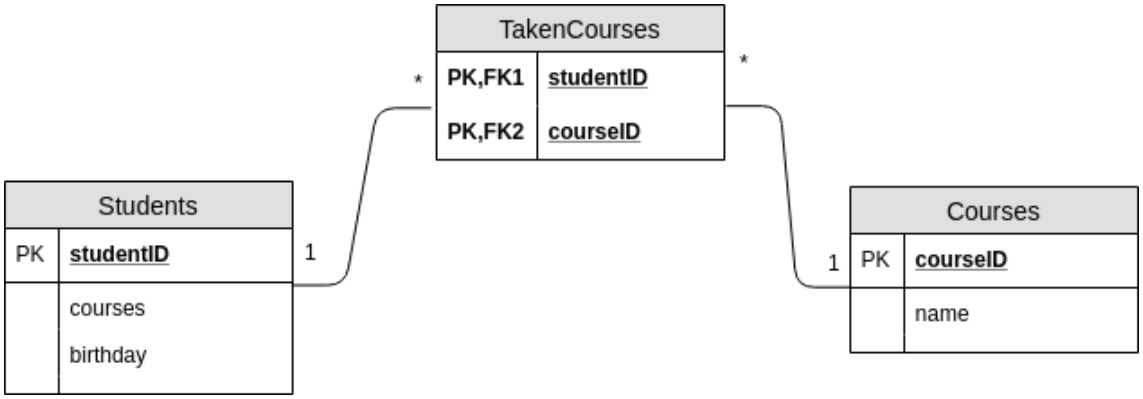


Figure 2.1: An ER-diagram that shows one student can have many taken courses, and each course is taken by several students. A specific taken course is associated with one student and one course.

Join is an important aspect for relational databases and one of many properties that differentiates relational databases from NoSQL databases [33]. Join is a built-in feature in SQL databases that lets the user create an even bigger table out of several smaller ones, so that extensive and necessary queries can be done.

Buckler and Wodehouse [15, 68] both list several differences between the SQL and NoSQL databases. These differences correspond to that of Khan [33]. Buckler also gives small code snippets, which further broadens the difference and highlights the strengthens and weaknesses. Most importantly, both Buckler and Wodehouse mention that join is not an operation available in NoSQL, just as Khan does in his presentation.

2.4.3 Different SQL

There exists a plethora of different SQL extensions. They range from more complex, as PostgreSQL, to more ”lite” functionality, such as SQLite. The licensing also differs from open source, for example, PostgreSQL, to commercial licenses, such as Microsoft SQL Server. In this project we did not have any licensing at our disposal so as a result, the only options were open source software.

The two big open source SQL extension available are MySQL and PostgreSQL [57, 21]. PostgreSQL is an open source object-relational database system. It started as a project in 1986 at the University of California at Berkley and is still used today [49]. MySQL was developed by Michael Widenius, Allan Larsson and David Axmark in 1995 when they did not get better performance with SQL than their current database implementation with mSQL [58]. The company MySQL AB was later founded that year and in 2000, MySQL was turned into an open source software. MySQL AB was later acquired by Oracle, who today, apart from providing the open source part of the software, sell their cloud service together with MySQL enterprise [58].

Since both PostgreSQL and MySQL are built on SQL, they have a lot in common when it comes to default capabilities. Only when more specific features are needed or wanted will the differences be more apparent. As an example, MySQL does not support full outer joins. A common solution is to write a query that has the same functionality, much like Quassnoi does [50].

2.4.4 ACID

The ACID properties guarantee that relational databases provides reliability of transactions [59, 26, 29]. ACID stands for Atomicity, Consistency, Isolation, Durability. RDBMS databases typically follow the ACID properties [54]. Hadjigeorgiou explains each property as follows [26]:

- Atomicity: Either all parts of a transaction must be completed or none.
- Consistency: The integrity of the database is preserved by all transactions. The database is not left in an invalid state after a transaction.
- Isolation: A transaction must be run in isolation to guarantee that any inconsistency in the data involved does not affect other transactions.
- Durability: The changes made by a completed transaction must be preserved, or in other words, be durable.

ACID is important in the aspect of transactions in databases. Transactions are any single operation that is made up of one or many steps [3, 29]. An ACID compliant database will ensure that only successful transactions are completed and this in turn will provide data integrity for the database, as well as enabling recovery from the system failures [37, 29].

2.5 NoSQL Databases

In a response to the realization of the limitations of traditional relation databases, NoSQL (Not-only-SQL) databases started to emerge. NoSQL databases provides a solution for applications that need storage, retrieval and manipulation of non-relational data. The NoSQL databases are horizontally scalable, which means that performance can be increased by adding an extra machine to the server, and most are open source [7]. NoSQL databases have been around since the end of 1960s. However, they did not gain much popularity until Amazon released a paper in 2007 about their Amazon DynamoDB, which is a NoSQL database that they use internally [22].

There are currently over 150 different NoSQL databases today and there are four different types of NoSQL databases: key-value store, column-oriented databases, document-based storage and graph databases, each type serving a different purpose. There are some advantages of choosing a NoSQL database over a relational. For example [43, 35]:

- They are generally faster and more efficient to read and write data.
- There is a wide range of data models to choose from, meaning that the application owner can choose a NoSQL database that fits their application's needs perfectly.
- They are easily scalable horizontally, unlike SQL databases that needs a more powerful and expensive computer to increase the capacity of the database, NoSQL databases work well in a distributed manner, and to increase their capacity one can add more commodity servers instead of buying a bigger and more expensive server. This leads us to be able to easily reduce system capacity when it is no longer needed.
- Some of the NoSQL databases are programmed to handle hardware failures on the servers, so if one node or several go down in the database cluster, the database will still work perfectly, and the clients will never notice. The number of nodes that can go down before the database starts to malfunction can be configured by the owner. This is a trade off as a higher replication rate of data will make writing slower because the data will be written to more nodes, but you will be more resilient to hardware failures.
- Most of the NoSQL databases are open source, which adds more transparency and lets the developer compile from source and provide their own bug fixes.
- They can handle unstructured data, i.e. non-relational data, a lot better because they do not require a pre-defined schema.

Of course, they also have their disadvantages to relational databases. For example [43, 35]:

- Most of the NoSQL databases are relatively new, immature and have a limited ecostructure. Unlike traditional relational databases, many NoSQL databases are missing customer support and management tools.
- There is no standard query language like SQL between NoSQL databases. Each NoSQL database has their own query language, which can be bothersome and not a pleasant experience to learn a new query language for every new database. In addition, NoSQL databases usually do not possess support for complex queries, like join, leading to laying the responsibility on the developer for writing complex queries on the application level.

- Most of them do not support ACID transactions. The developer must implement this by themselves if they find ACID transactions needed for their application.

With these advantages and disadvantages in mind, an application owner must compare the pros and cons to decide which type of database architecture fits their needs best. If that answer would fall on NoSQL, then the next decision would be to decide which type of NoSQL database that they should choose. But first, we need to describe the fundamental principles that all NoSQL databases rely on.

2.5.1 The CAP theorem

NoSQL databases are affected by the CAP theorem [25]. CAP is short for:

- **Consistency:** All clients will always, instantly see the latest data.

- **Availability:** All clients will always be able to fetch data, even when some node in the cluster is down.

- **Partition tolerance:** The system will always continue to operate regardless if a node of the cluster experience a message failure.

Now, what the CAP theorem says, is that it is impossible for any distributed system to support all these three attributes, which means that the developer must prioritize. In turn, many NoSQL databases adapt to the eventual consistency model, which means that all data in the database will eventually be correct. In conclusion, when choosing a NoSQL database, one must consider the theory behind the CAP theorem. A visual diagram of this theorem can be seen in Figure 2.2, showing an example of what different databases prioritize. Where two circles overlap, we can see a choice of databases that has these two attributes. No database is available for the area where all three circles overlap.

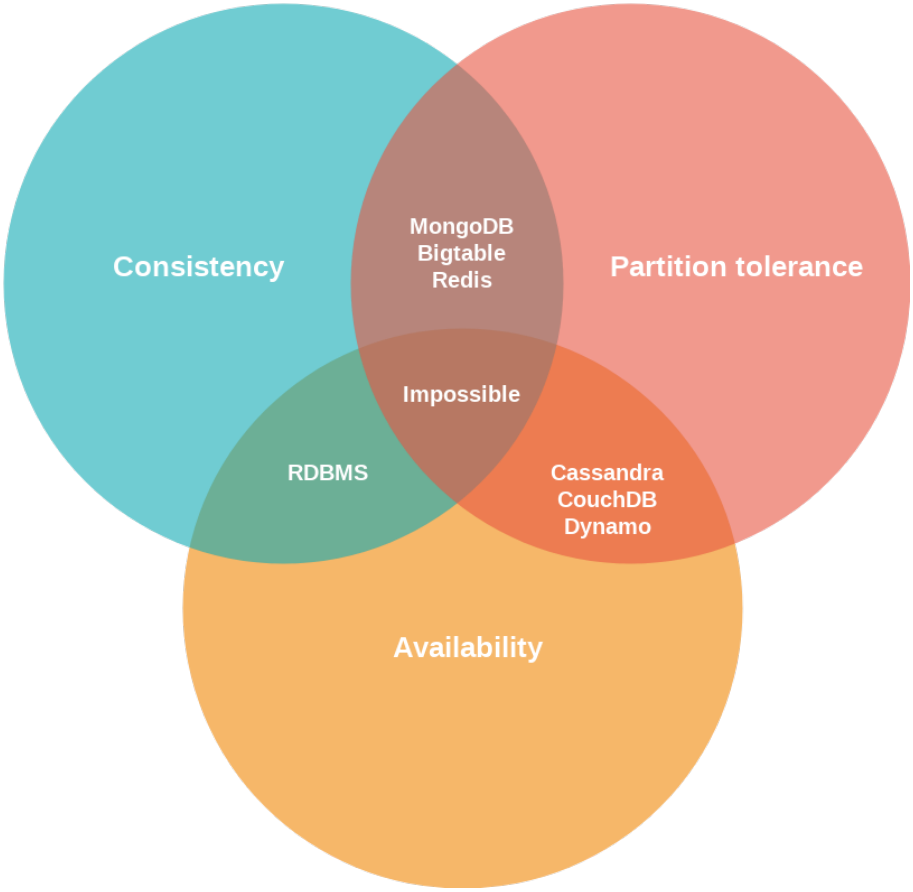


Figure 2.2: A Venn diagram illustrating the CAP theorem, which stands for Consistency, Partition tolerance and Availability, showing that no database today can support all three properties. Between each two properties there are some examples of databases who fulfils those said properties.

2.5.2 BASE

NoSQL databases were created to provide high performance, both in terms of size and speed, and a high availability - in exchange for losing the ACID trait. Instead they hold to a weaker standard, which is the feature of BASE (Basically Available, Soft state, Eventual consistency) [60]. The acronym for BASE was purposely chosen to show that it is the opposite of ACID. To give a better explanation of BASE we will describe what each term means:

- **Basically available:** The database should be up and available as much as possible.
- **Soft state:** The consistency of a transaction in the database is a soft state where the state does not need to update directly after the transaction, but rather in sometime within a threshold.
- **Eventual consistency:** The procedure described in the soft state leads to eventual consistency.

2.5.3 Key-value stores

Key-value stores use a hash map as their fundamental data model [43]. The data in this model is represented as a collection of key-value pairs, where each key in the collection must be unique. This simple approach to store data is faster than the traditional RDBMS and is highly scalable. Complex querying like join and aggregate operations are not available for this kind of data structure. The previously mentioned NoSQL database, DynamoDB is a key-value store and fits very well for its purpose, where you want to store a user's session or a user's shopping cart, which is classified as changing, unstructured data. The development of many key-value stores was sparked by Amazon's dynamo, some notable key-value stores that are worth mentioning are Amazon DynamoDB, RIAK, Redis and Voldemort.

2.5.4 Document storage

These types of databases store and organize data as collections of documents. The documents inside the database share some common properties with records in relational databases, the difference is that the documents are a lot more adaptable since they do not need to confine to a defined schema. The formats that are used in the document store are known standard formats, for example, XML, JSON, BSON, and PDF. With a structure like this, a user can have a different number of fields for the same type of data inside a document. To understand the terminology, a collection can be considered as tables in a relational database, and documents can be considered as records. However, they are different, in a relational database, a record will have the same fields for every record, and if a record does not need all fields, then the unused fields in the record will remain empty, while documents in a collection may have fields that are entirely dissimilar. A document store is like a key-value store, with the difference that a document store enables the user to encapsulate key-value pairs in documents, also known as key-document pairs.

Document store databases are a good fit when the data is not structured, the domain model can be separated over several documents, and the data between the documents will not have a lot of relations or a need to be normalized [43]. Notable Document stores are MongoDB, CouchDB, CouchBase, and DocumentDB.

2.5.5 Column stores

Column stores databases define the database as a predefined set of columns, where a column can contain other columns, which is then called a super-column. They are like traditional relational databases that store columns and rows. However, in their data structure, the keys and values of the columns can be varied from row to row in the same table. This is illustrated in Figure 2.3.

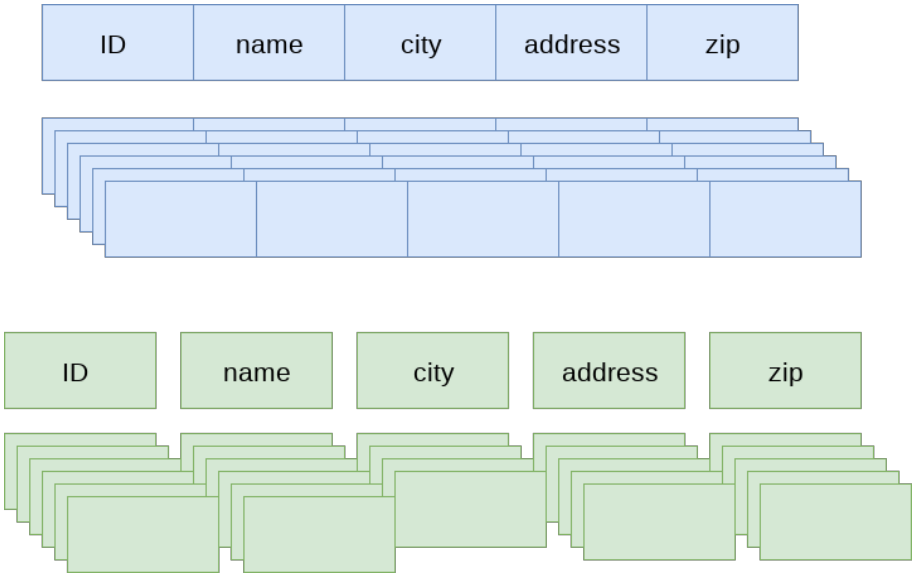


Figure 2.3: Showing the differences between a standard column-row architecture, and a column store

Now, as we can see, the data is stored column-by-column in a column store, which makes it easier to query on a field since they are stored together. This in turn can improve the performance of the queries depending on what kinds of queries your application is usually doing. To illustrate this in an example, we have first made a representation of a data set in a standard relation data table, which can be seen in Figure 2.4. As you can see the unused fields in a row is filled with a null value. If we would do a migration and add another column, say for example *nbr_members*, which in this case only makes sense for the last row in the figure, then the rest of the rows would still need to have that field filled with a value null.

performer	born	country	instrument	founded	type
James Hetfield	1963	USA	vocals, guitar	null	artist
Lars Ulrich	1963	Denmark	drums	null	artist
Metallica	null	USA	null	1981	band

Figure 2.4: A representation of a data set in a relation data table. Note that the rows also have fields that are not filled with any data, instead they have a value of null.

Next, is the same data set, but this time represented in a column store, which can be seen in Figure 2.5. The big difference here is that the fields are represented as a key-value, where the partition key acts as a key, and the value for the partition key is the super-column. Inside the super-column we have smaller columns holding a key-value each. Adding a new key-value to a column is trivial without needing to alter the rest of the columns. Notable column stores are HBase [10], Cassandra [34], and Amazon SimpleDB [8].

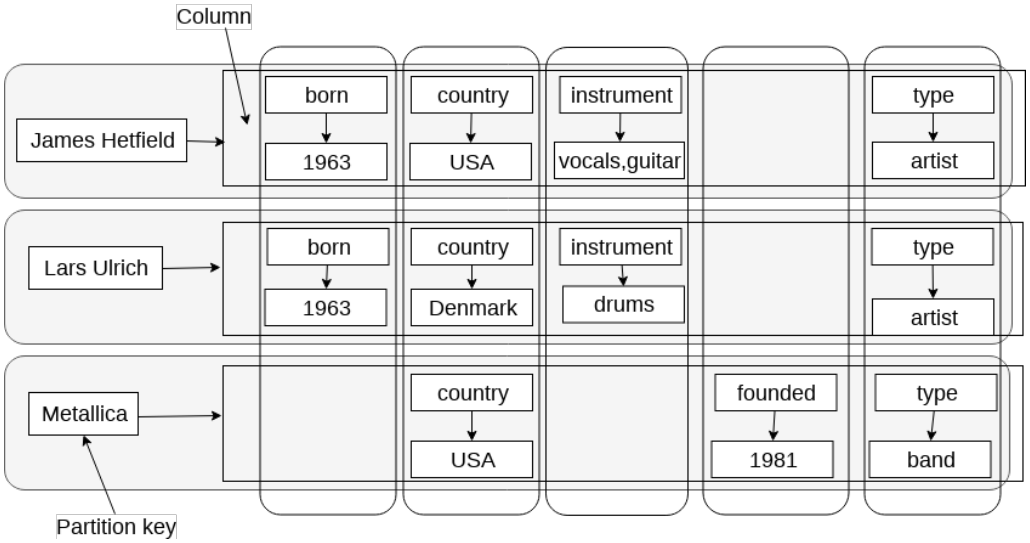


Figure 2.5: A representation of how the previous data set in Figure 2.4 can be visualized in a column store. Each field in this case is represented as key-value. The partition key is the main key and the value for the partition key is the whole column.

2.5.6 Graph based

Databases that use a graph architecture are designed for applications who have data that can be represented using a graph consisting of nodes/elements connected between each other, with a finite number of relations. This type of data could, for example, be relations between users on a social media platform, road maps in navigation systems, links in public transportation system, etc. The focus of graph databases is on the relation between data. They are schema less and provides a powerful storage for semi-structured data. Notable graph databases are Neo4j [45], InfiniteGraph [46], and Sparksee [56].

Chapter 3
Approach

In this section we will describe our approach to determine which database is most suitable for the situation that we and Axis are in. First, we will characterize what kind of data that we are working with, how it is extracted and the overall size of it. Next, we will discuss Axis current implementation and why this cannot be a long-term solution. Then, we will walk through the requirements that Axis has on a database. From this we will explain why we have chosen a specific set of databases and dig deeper into that set of databases, describing each one and the implementation we have thought of using to fulfil these requirements. Finally, we will specify how we will measure which database is most suitable and what kind of benchmarks we will be using.

3.1 The metadata

The data we will be using is metadata generated by video cameras from Axis. Metadata is not the video stream, but rather everything else that the video camera will generate. We have categorized the metadata into two categories, metadata from a whole sequence and metadata from frame-by-frame. Examples of metadata that is frame-by-frame that the video cameras could generate is: PSNR, QP, SSIMY, frame type, bytes per frame, etc. The details of what these parameters are, are not that important, what is more important is that we have a diverse set of metadata and a mix of strings and integers. There is also metadata that is manually created, e.g. the name, the location of the camera and the codec that is used. This kind of metadata is typically ordered by sequence, since they will not change from frame to frame.

A typical video sequence is approximately 5 minutes to 1 hour, a video camera has 30/60 fps (frames per second), which means that in the max case we will have $60 * 60 * 60 = 216000$ frames in a video sequence, where each frame holds unique metadata that needs to be saved in a database, and this is just one video sequence, where it would roughly be 24 MB if stored as a JSON-file. Now, imagine that we want to save several of these every day without deleting the old sequences. The data will accumulate to be relatively high numbers after some time.

Type of data

We have chosen to categorize this type of metadata as semi-structured data. The reason for this is:

1. The metadata will not contain the same type of fields for every sequence.

2. The nature of the metadata will change and it is impossible to anticipate what kinds of fields it will hold in the future.

Now, if we would just have the first case then this problem would be a lot easier. If we can accept that some fields in the database would have a null value for some sequences, then this problem could be modeled as relational data problem. However, since we cannot anticipate what kind of fields we will need in future to represent this data, it is much easier to treat the data as semi-structured and make up a model that fits that narrative.

3.2 Current implementation

The current implementation to handle metadata at Axis is not a viable long-term solution. We will describe it in this subsection to motivate why. As of now, the metadata is being stored on a user's file system. In the organization, they have come up with a directory-template that everyone should follow when they save metadata on their file system. An example of such template can be seen in Figure 3.1. By using this structure, a user can run a previously written python script that will parse this folder structure into a JSON file. Next step is to load that JSON file into another python script and query on it by writing the keys in correct order for a sequence that they want to extract metadata from.

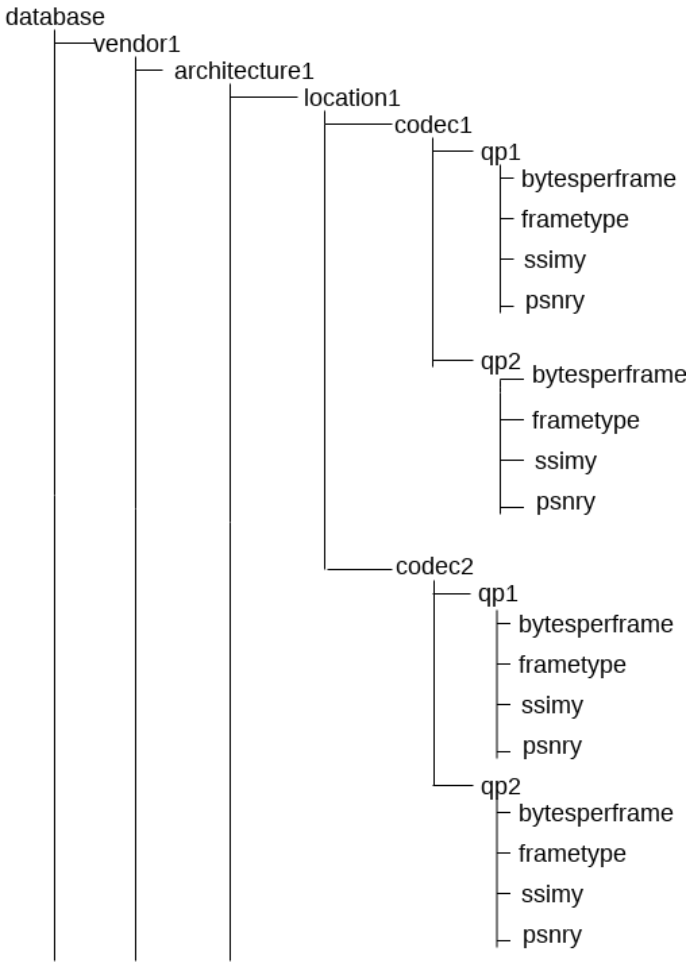


Figure 3.1: An illustration of the directory structure that Axis use to store metadata.

There are several limitations with this current solution:

- Everyone that uses this system needs to know the exact directory template, otherwise the scripts will fail.
- Retrieving and inserting data is a bothersome process that takes up a lot of time and is prone to errors, since it is done manually.
- A user needs to know the exact directory structure, all the values of the keys and in what order they should be to be able to retrieve the data.

These limitations were identified together with the employees at Axis, who uses the current implementation today. The implementation has some positives to it:

- Retrieval of data is instant, when the user knows all the keys in the JSON structure.
- The data is in text files, which is easy to manipulate if needed.

The goal is to come up with a new solution which involves a database that will erase all the negative aspects, whilst keeping the positive ones.

3.3 Use cases

When finding a suitable database management system (DBMS), it is important to understand the main use cases for the DBMS. For Axis there would be several different scenarios, and the database should have support and provide good performance for all of them. These use cases have been constructed in cooperation with Axis, by talking to their employees to understand how they are using the current implementation today and what they want to be able to do with the new solution.

First one is insertion of metadata. Every day, database entries should be created automatically with the metadata extracted with the latest software. The common term for this is "builds". This happens for several different cameras every day, where the number of frames is ranging from approximately 100-216000. A user should also be able to manually insert metadata from a video sequence that was not generated during the automated builds.

The second one is updates of metadata. Users might want to replace the metadata of a video sequence with new metadata, such as, where the users realize they uploaded incorrect metadata for a video sequence.

The third one being retrieval of metadata. For a DBMS, this operation is called reads. Reads will be done by users, where they want to retrieve metadata from one or several video sequences. The benefits for this are that developers can clearly see the improvements or failures for new builds. They can do this by extracting a previous video sequence's metadata and compare it to the metadata of one more recent video sequence. By analyzing the result, the developers can see if the changes they made to the recent builds has improved the quality of the metadata.

If a user wants to retrieve metadata of several video sequences, then the amount of data can grow quite large. For example, a user wants to retrieve metadata from five specific video sequences, and each video sequence holds 100000 frames, the total number of frames would then be 500000, which is a lot bigger than the largest insertion. We will focus mostly on reads, because it is done by a user, not an automated script, leading to us wanting to minimize the time a user must wait for the retrieval of data.

The fourth use case is retrieval of metadata from several video sequences that fulfils a specific condition, for example, every video sequence from a camera that was taken today, ranging to a week ago. A situation like this is called a scan, where a range of records are being queried, instead of individual records. This is also a heavy operation done by a user, with a lot of data involved.

After a discussion with the employees of Axis, it was clear that we should put our focus on the third and fourth use case, i.e. retrieval of data. These are the use cases that will be used most by the users, leading to us wanting to minimize the time it takes for those use cases. The other use cases will still be considered, just not on the same level.

What is common for all these use cases is that there will be very little chance of concurrent requests to the database. The team at Axis that will use this DBMS will consist of around ten employees, where each employee might use the database a couple of times per day.

These use cases will later be reflected in our benchmarking study.

3.4 Requirements

We have in cooperation with Axis defined a set of software requirements to be able to choose a set of databases that we find suitable for Axis and their needs. These requirements were collected by discussing with employees of Axis of what their needs are, and then analyzing how these needs can be structured as requirements. These software requirements are defined as:

1. The database should be open source.
2. The database should be able to handle semi-structured data.
3. Migrations of the database should not be needed.

From these requirements we can find different databases that we think has potential to fulfil these, and if there are several, measure which database has the best performance with regards to the use cases.

3.5 Database selection

In this section we will list the databases that we think have potential to be a part of our solution and motivate our decision. We will also explain why some types of databases did not make the cut. The databases were found by extensive research on the web, with googling and reading different papers related to our field of work. An important metric for us was to have different types of databases management systems, for example, not only compare relational database management systems, or only compare document storage databases.

The DBMSs that we have decided to use are PostgreSQL [49], Cassandra [34] and MongoDB [39]. In this set of DBMSs, we have one RDBMS and two NoSQL databases, where Cassandra is a column store and MongoDB is a document storage database. We will dig deeper into these databases and describe them and list some of their features, but first, we will explain why a key-value store and graph databases are not suitable.

A key-value store has a relatively simple architecture where querying is fast, but to be able to query, a user needs to know every value of the keys in the querying tree. This is a limitation in the implementation that exists today at Axis and by using a key-value store we have not removed a limitation. Moving on to graph databases, graph databases as previously said, is a perfect fit for representing relations between data, for instance a user's followers in a social network. We do not have a lot of relations between our data, except between frames and sequences, which makes graph databases not suitable for our problem set.

3.5.1 PostgreSQL

The first database that we chose was PostgreSQL, an RDBMS that is open source with immense support from different communities. We had several different reasons to choose PostgreSQL. The first one was out of pure research stand point, where we wanted to have at least one database based on SQL. MySQL was also in the discussion, however, we found that comparisons between MySQL and NoSQL databases had already been done on several occasions

with a lot of different NoSQL databases, for example in Tudorica and Bucur's paper [60]. We did not find as many papers comparing PostgreSQL with NoSQL databases. Also, since version 9.4 of PostgreSQL, a JSONB structure is supported [64]. According to the documentation it says [64]:

> "With the new JSONB data type for PostgreSQL, users no longer have to choose between relational and non-relational data stores: they can have both at the same time."

This new functionality fits our need perfectly. What this new type does, is that instead of using strings or integers as a datatype for representing the metadata, we can use a JSON object. A JSON object is dynamic and lets us have different properties inside the metadata JSON object for each frame, without having to deal with any migrations. In Figure 3.2 is an illustration of an ER-diagram that represents the schema that we will use to store a sequence. As we can see, a sequence and a frame impose one-to-many relationship, where every frame has a foreign key, which is the primary key of the sequence to which it belongs.

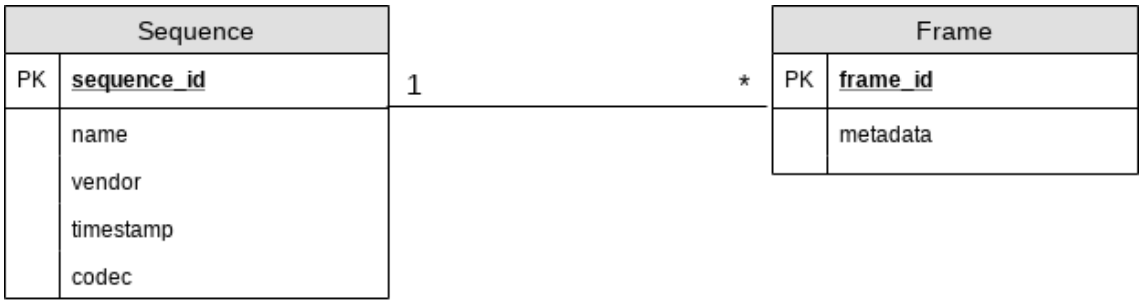


Figure 3.2: An ER-diagram of a schema that is possible to use for the PostgreSQL database. In this schema a sequence and a frame would impose a one-to-many relationship.

3.5.2 MongoDB

MongoDB belongs to the category of document storage databases and is the most popular NoSQL database on stackoverflow, meaning that it is the most commonly used NoSQL database by users participating in their survey [57]. From the same survey, we can gather that MongoDB is also the most wanted database to learn [57]. This is one of the reasons for us choosing MongoDB. There are several others as well. The second one, is that document databases are in general not much concerned about concurrency with read and write, but rather to ensure a reliable storage for large data sets and a rich querying experience with good performance [27]. MongoDB supports a data structure called BSON [14], which is the binary encoding of JSON documents (not to be confused with JSONB [64]). BSON also adds additional data types such as binary and Date. A large benefit of using BSON is that it provides the ability to add indexes to values that resides inside the BSON structure. This can improve the performance when running queries.

When structuring a one-to-many relationship in MongoDB, the team behind it will give you two suggestions [47]. The first one is to store all the data inside the same document, where the data with the many-relationship is nested inside the JSON structure, this is called

denormalized data. The other suggestion is to store it as normalized data, which is usually what you would do in an RDBMS. To store it as normalized data, one would store sequences and frames in separate documents, with a reference stored inside every frame, pointing to the sequence, much like we would do in the case the PostgreSQL database. Now, the recommendation on how you should store the data depends on what queries the users would most often do. If users would mostly be interested in fetching the data frame-by-frame, then normalized data approach would be better to avoid querying on unwanted data. However, in our case the users are mostly interested in retrieving whole sequences at a time, with all the frames included. In this situation storing the data as denormalized is a lot more efficient and resource-friendly. Although, there is one problem in MongoDB when storing sequence by sequence, and that is that the file size of a document can be quite large in the maximum case of 216000 frames. MongoDB has a document file size limit of 16 MB, which the maximum case will exceed. This requires some work on the client side and can be handled by splitting the document into several, if the file size exceeds the maximum limit.

3.5.3 Cassandra

Cassandra was originally developed by Facebook with the purpose to handle when users search through their inbox. Back then, they were using MySQL which was not able to handle the immense throughput of data that was required for a function like that. The engineers at Facebook then started to work on a storage system where the focus was to provide availability and scalability, which would put Cassandra between availability and partition tolerance in the CAP theorem, whilst not prioritizing consistency. The result was a distributed storage system that can easily scale horizontally by adding more servers, referred to as nodes, to the cluster, with a large focus on providing high write performance, whilst still maintaining decent read performance. The Cassandra project proved to be a huge success for Facebook and is now deployed as the backend storage system for multiple services at Facebook. Later, Cassandra was migrated to the Apache Foundation and is now maintained by Apache as open source.

Cassandra is a NoSQL database which can be categorized as a column store. With this as our third database, we have a variance of different types of databases when benchmarking the performance of them. We have decided to use Cassandra in our project mainly because the paper by Abramova and Bernardino [4] and the paper by Tudorica and Bucur [60] shows very promising numbers for Cassandra when working with a large data set. As said, Cassandra was developed by Facebook and was meant to run on tens of thousands of servers around the world, serving a billion users, which might seem a bit excessive for Axis to have that kind of infrastructure.

As previously stated, NoSQL databases do not have centralized language for querying like SQL, however, Cassandra has its very own query language called CQL, which shares a lot of similarities with SQL, making it easier to get accustomed to CQL if the developer knows SQL. Cassandra does not use SQL, since Cassandra is not a relational database and does not support ACID properties.

To know how data should be structured and saved in Cassandra to optimize performance, one must understand the fundamental architecture of Cassandra. When creating a Cassandra database, a cluster will be created. In this cluster there are nodes which the data is stored on. If a node is added or removed by the application owner, all data will automatically be distributed over other nodes. Using this technique, it is not necessary to calculate which data

should go to which node. The data is also replicated over multiple nodes in a cluster. This can be configured by changing the parameter for the replication strategy. If the replication strategy is set to 3, then every data written to a node will also be replicated to 2 other nodes, meaning that the data will exist on a total of 3 nodes. This will help to mitigate the damage if a node will go offline. On the other hand, if the replication strategy is set to a high number, then the write process will take longer and the size of data will grow faster as well, since it will exist multiple copies of the same data.

Cassandra uses a feature called partition keys. Partition keys are used to help Cassandra partition the data across the nodes. Imagine the node being a physical disc where Cassandra fetches the data. Data that has the same partition key will get saved next each other on the disc. This will aid Cassandra into faster reads and writes if the data that is being handled in each request has the same partition key. In our case, since we would be fetching the video metadata sequence by sequence, we would use the sequence as partition key and save all its frames next to each other using that partition key.

3.6 Benchmark

In our efforts to find a suitable DBMS, we decided that a benchmark study would be good aid in the decision making of the final database. Ideally, a benchmark that covers different properties of database operations would be the best choice for us, according to our use cases. The benchmark can be done by two different approaches, either an already existing tool could be used, or a self-composed one.

As most of our operations are reads, we had a first thought of only measuring the access speed of the databases. This could have been done by using simple code that measures the start time and the end time, then outputs the delta of the two times. However, after more studying of what the actual purpose of the database was, it was clear that full neglection of insert and update operations was not ideal. The two extra operations increased the complexity of implementing a suitable tool for doing the benchmark.

In the research we discovered several different benchmarking tools suitable for testing the performance of different DBMSs. The alternative of using an existing tool, as compared to a self-composed tool was an interesting alternative. One of the most documented tools, was Yahoo! Cloud Service Benchmarking (YCSB). YCSB is an open source software, available to be used by anyone, developed by some employees at Yahoo! Research [19].

YCSB has been used in various papers when comparing different database systems to one another [6, 51, 4]. This makes YCSB a well validated tool for us to use, but also questions whether what we are researching is unique. The research of using the tool to compare between SQL and NoSQL has not been documented. This will be the biggest difference between this report and other papers.

The tool only has support for NoSQL databases, which is a problem, as we have a SQL database that also needs to be benchmarked. Fortunately, a Go port of the YCSB benchmark has been developed by some developers at PingCAP. The port has support for some SQL databases, among them PostgreSQL. It uses the same commands as the regular benchmark, and as such, is a great tool to have for our benchmarking. Since the documentation for the port functions is the same way as the original, we will henceforth refer to the original documentation in this report.

The benchmark uses different workloads on the database systems during tests [19]. The core package has several different performance measurements. The following workloads are in the core package [19, 11]:

Workload A: Update heavy workload
This workload has 50% reads and 50% writes.

Workload B: Read mostly workload
This workload has 95% reads and 5% writes.

Workload C: Read only
This workload has 100% reads.

Workload D: Read latest workload
This workload has 95% reads and 5% inserts. The latest inserts are the most popular when reading.

Workload E: Short ranges
This work load has 95% scans and 5% inserts. Ranges of records are queried.

Workload F: Read-modify-write
This workload has 50% reads and 50% writes. The record is read, modified and then written with the new changes.

Benchmarking with YCSB is two phased. The first phase is called the load phase, in this phase, the data that will be inserted into the database is defined. The user can also change, among other variables, the record count to operate on. The second phase is called the transaction phase. In this phase the loaded workload is executed. Even in the transaction phase the user can change the operations count for the phase. In this, the user can even specify the number of threads and how many operations per second that should be done by each thread.

The load phase is the same for all the workloads and as such one load can be done before running several workloads. However, since workload D and E uses inserts in their transaction phase, it is recommended to delete the database and reload the data before executing either one after the other. The benchmark's wikipage have a recommended step-by-step work procedure when running all the workloads in the core package, which we tend to follow [11]:

"1. Load the database, using workload A's parameter file (workloads/workloada) and the "-load" switch to the client.

2. Run workload A (using workloads/workloada and "-t") for a variety of throughputs.

3. Run workload B (using workloads/workloadb and "-t") for a variety of throughputs.

4. Run workload C (using workloads/workloadc and "-t") for a variety of throughputs.

5. Run workload F (using workloads/workloadf and "-t") for a variety of throughputs.

6. Run workload D (using workloads/workloadd and "-t") for a variety of throughputs. This workload inserts records, increasing the size of the database.

7. Delete the data in the database.

8. Reload the database, using workload E's parameter file (workloads/workloade) and the "-load switch to the client.

9. Run workload E (using workloads/workloade and "-t") for a variety of throughputs. This workload inserts records, increasing the size of the database."

The workloads consist of different commands and contains a pre-determined set of tables of records. The commands are implemented with a distribution which, among other things, decides which command to run, which record to use, and how many records to use [19]. The commands implemented are [19]:

Insert:
Inserts a new record to the database.

Update:
Replaces the value of a field.

Read:
Reads either a field or all fields of a record

Scan:
"Scan[s] records in order, starting at a randomly chosen record key. The number of records to scan is randomly chosen."

Cooper et. al. [19] are aware that the core package tools do not cover all of the performance space. They have therefore implemented a feature that allows users of the tool to implement their own workloads and modify existing ones. We do not plan on implementing our own workloads, since the current workloads cover our use cases, as in section 3.3.

The loading phase will cover the use case of insertion of metadata. Workload A and workload F will reflect the use case of when a user reads the metadata from a video sequence, and then wants to update it. Workload B, C and D, will cover the use case of when a user wants to retrieve metadata from one or several video sequences. Lastly, workload E will reflect when a user wants to search for several video sequences, using specific conditions.

The user can modify the pre-determined workloads with different fields. Among the modifications is the ability to change the record and operation count, to achieve heavier loads on the workloads. This can be useful for us, as we can set the workloads' load to closer resemble that of the final prototype.

3.7 Experimental setup

To run the benchmarks, we used a Debian GNU/Linux 9 (stretch) 64-bit machine, with 16 GB of RAM available, with an Intel Core i7-7700K CPU @ 4.20GHz as CPU.

As our focus for the study we measure the execution time for each workload to evaluate the performance for each database system, where each data set is measured three times, as Abramova and Bernardino did in their paper [4], with a reset of data between each, then the average is taken between those three times to provide a result for a data set in a workload.

By default, the YCSB client uses a single client thread to handle the requests to the databases. There is a possibility to configure and increase the number of threads by changing a

runtime parameter when running a workload, to simulate when several clients simultaneously interact with the database. For our case, as stated in section 3.3, usually only one client will interact with the database at any given time, which is why we have decided to not to tweak any runtime parameters concerning threads.

Chapter 4

Evaluation

In this chapter we will present and discuss the results from our benchmarks. The results from the load phase and the workloads are presented in diagrams for an easy overview. The discussion will provide our thoughts on the results, as well as our final recommendation to Axis.

4.1 Results

We present the results from the benchmarking tests where we expose the databases to every workload as previously described in section 3.6. The results are presented as graphs with a graph for each workload.

In the following figures, we show the execution time for the loading phase, and the execution time for each of the workloads, A, B, C, D, E, and F. The data points that have been used for every graph is when running the measurements with 10K, 100K, 250K, 500K, 750K, and 1000K of operation, where an operation in this case is a database operation. The range of operations were chosen in consultation with the employees at Axis to reflect how they will use the database system in the future.

Loading phase The graph in Figure 4.1, shows that the databases are mostly linear when inserting new data into the database, where Cassandra has the highest slope, and PostgreSQL has the lowest. There are no significant differences between the databases at the lower levels of operations.

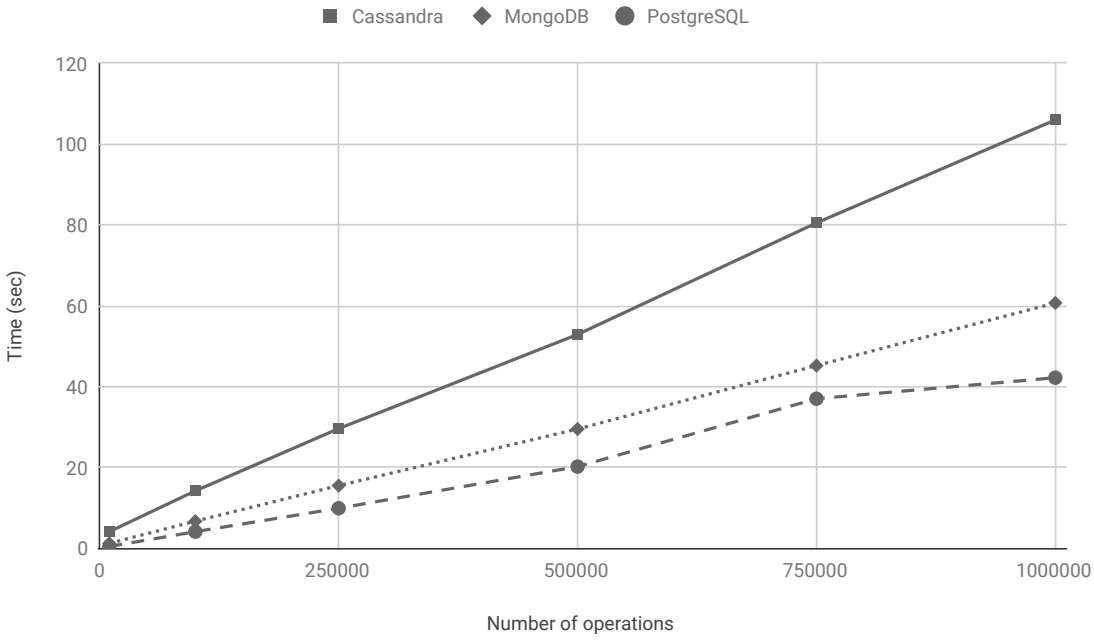


Figure 4.1: The execution time for the databases when running the loading phase with different number of operations.

Workload A (50/50 reads and updates) Cassandra is having a linear development throughout the entire experiment as we can see in Figure 4.2. What is more interesting is seeing that both MongoDB and PostgreSQL will increase their execution time rapidly in the end, where MongoDB even surpasses Cassandra. MongoDB has a bit of an edge when it comes to smaller numbers of operations compared to the others. PostgreSQL is the best one for 1000K operations.

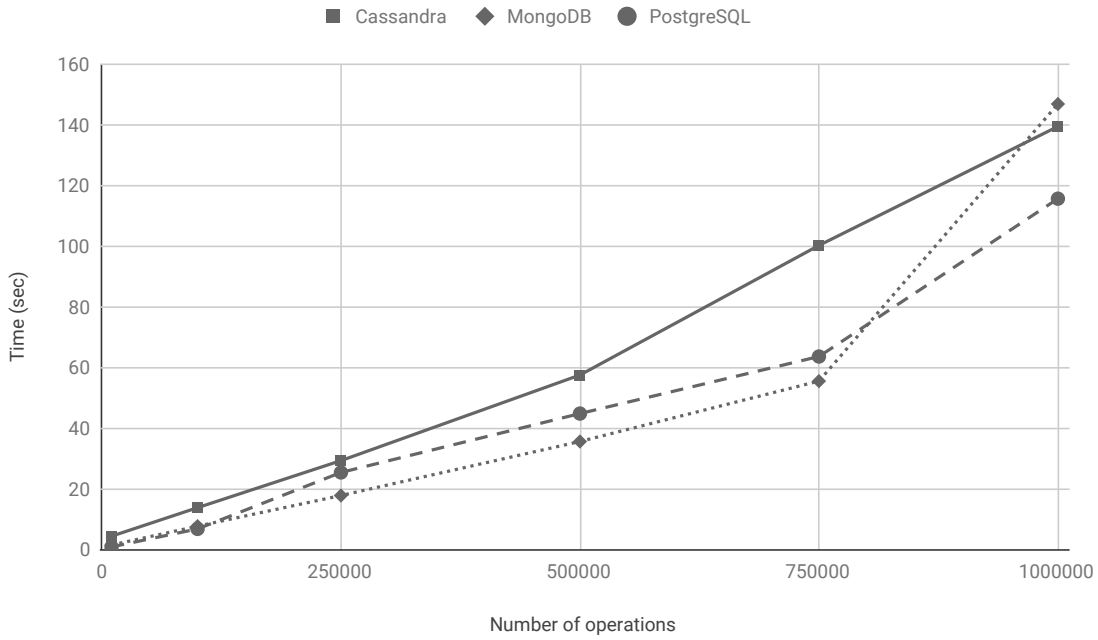


Figure 4.2: The execution time for the databases when running workload A with different number of operations.

Workload B (95/5 reads and updates) In the graph in Figure 4.3, we can see that all databases have a linear development where PostgreSQL is the clear winner in all categories.

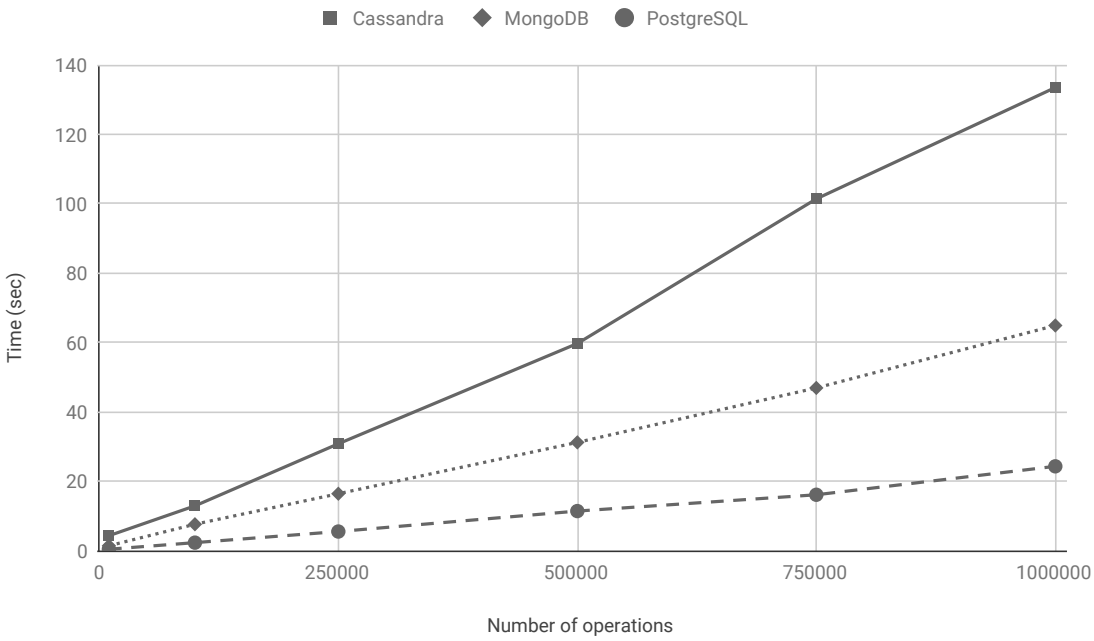


Figure 4.3: The execution time for the databases when running workload B with different number of operations.

Workload C (100% reads) The graph in Figure 4.4, shows very similar result to workload B, and PostgreSQL is the clear winner again.

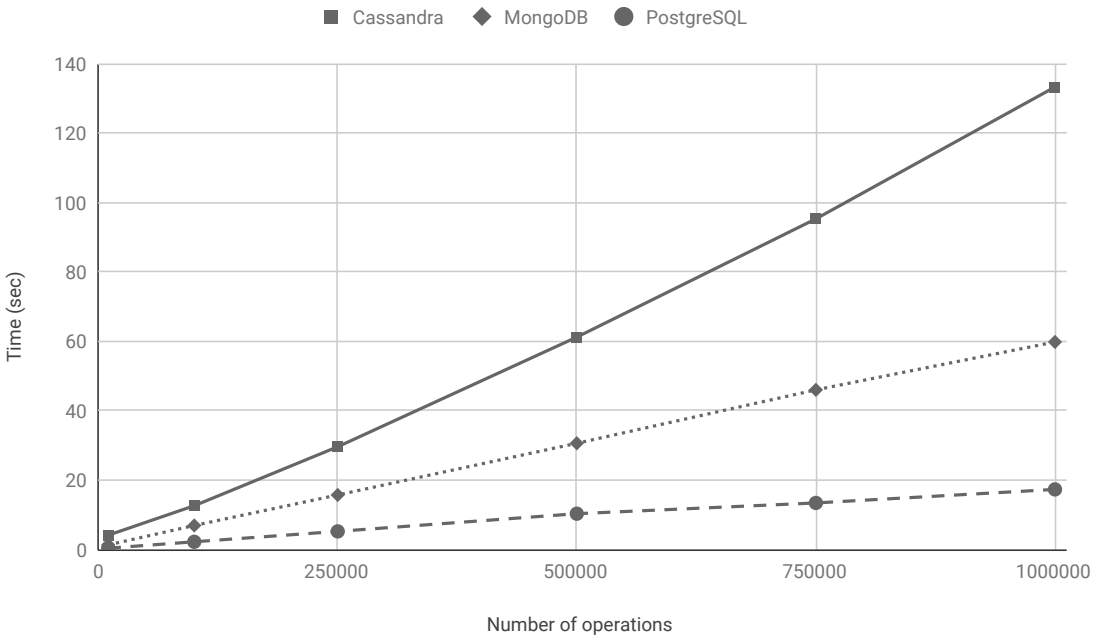


Figure 4.4: The execution time for the databases when running workload C with different number of operations.

Workload D (95/5 reads and inserts) Again, since we are mostly measuring reads, the results in Figure 4.5, will be like the two previous workloads.

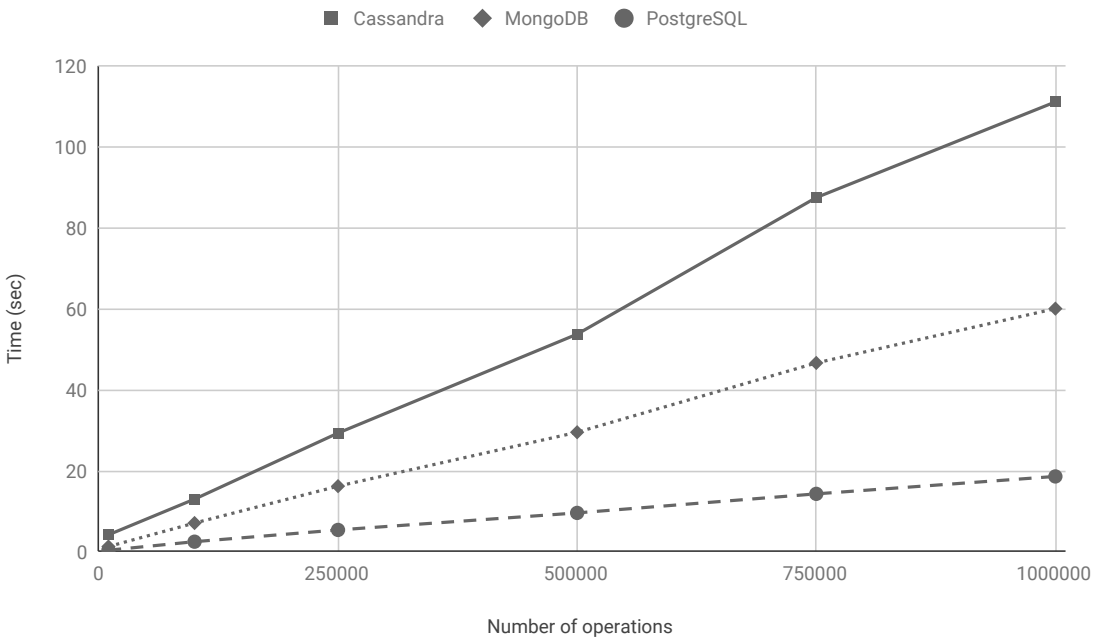


Figure 4.5: The execution time for the databases when running workload D with different number of operations.

Workload E: Short ranges (95/5 scans and inserts) As we can see in Figure 4.6, the time for this workload has increased dramatically compared to the other workloads, for all databases, where Cassandra is taking off to embarrassing numbers in the end. Again, PostgreSQL is the winner.

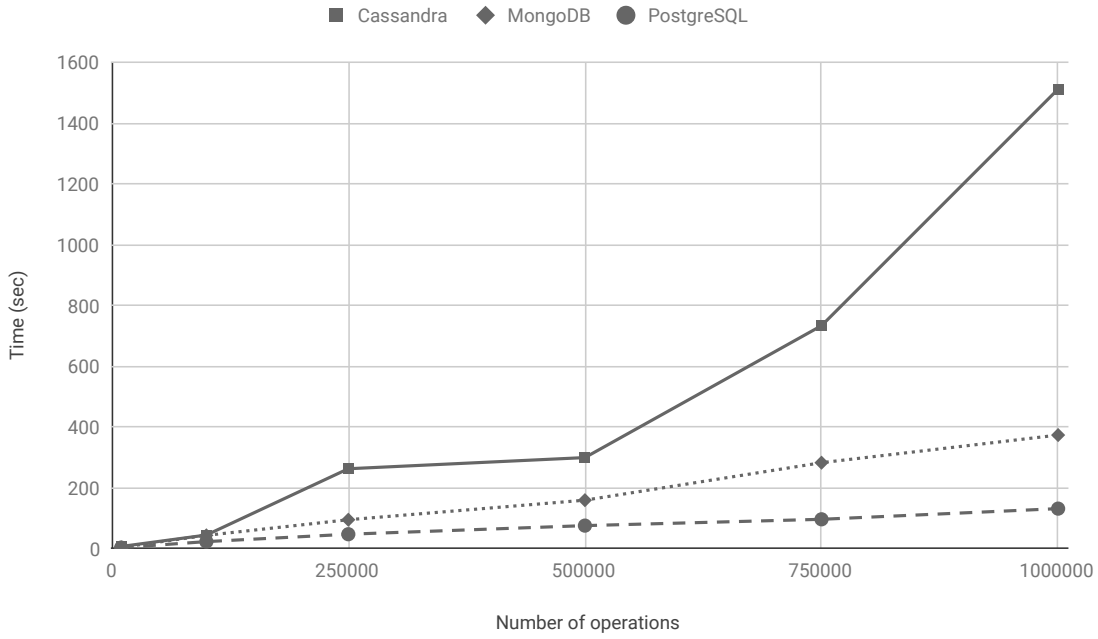


Figure 4.6: The execution time for the databases when running workload E with different number of operations.

Workload F: Read-modify-write (50/50 reads and updates) The results for workload F can be seen in Figure 4.7. All databases have a similar execution time. It is interesting to note that the execution time for MongoDB almost doubled from 750K operations to 1000K operations.

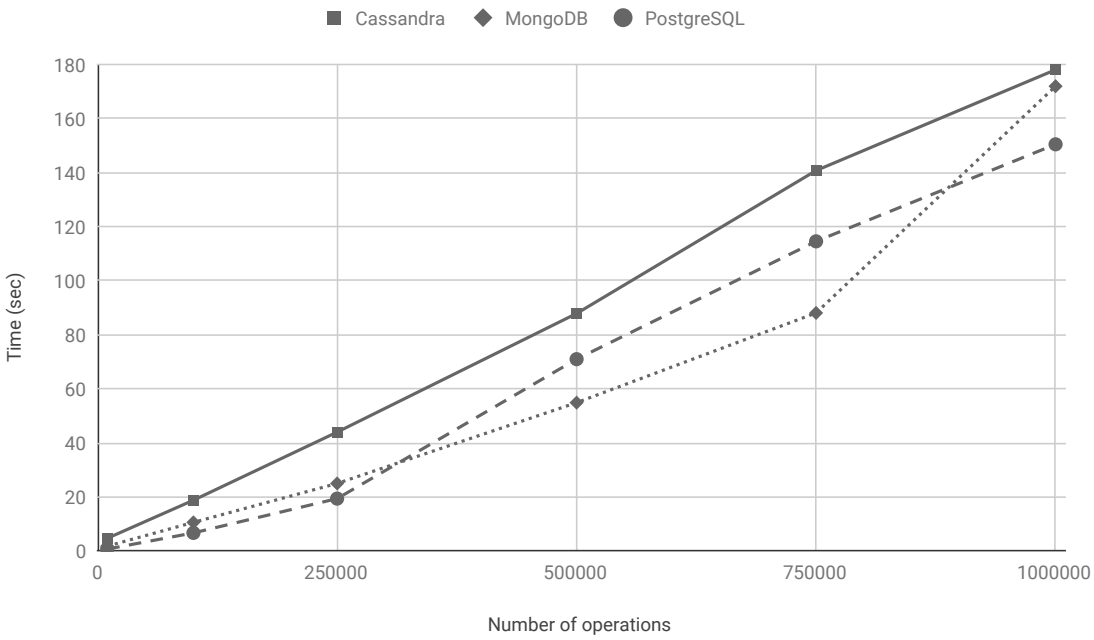


Figure 4.7: The execution time for the databases when running workload F with different number of operations.

The results from the measurements when running the workloads shows that PostgreSQL had the lowest execution time in 37/42 measurements. Cassandra scored the shortest execution time in 0/42 measurements, and MongoDB was in the middle with the lowest execution time in 5/42 measurements. At the other end of the spectrum, we have Cassandra who achieved the longest execution time in 41/42 measurements, MongoDB scored the highest execution time in 1/42 measurements, and PostgreSQL had the longest execution time in 0/42 measurements.

4.2 Discussion

The workloads are mostly focused on reads, where workload C has the most reads with a 100%. This fits well with our use case, which is why we decided not to implement any workloads of our own, as for example, as Abramova and Bernardino did in their paper [4], where they added two more workloads *H* and *G*, with 95/5 reads and updates for workload H, and 100% updates for workload G. If we, for example, would have a larger focus for updates, then we could have used those workloads as well. However, that is not the case. We will probably have more write operations to the database than read operations. However, most of those writes will be made by an automated script each day where the number of operations will be relatively small. Read operations are usually handled by a client request that is initiated by a user. In that situation the amount of data to read are often a lot larger than the scheduled inserts, which leads to us being more concerned about the performance of read operations, than the performance of write operations.

PostgreSQL is ACID-compliant. ACID is used to define properties for databases where concurrent access is some of the main use cases. We do not have these in our use cases,

and as such, we did not have ACID as a part of our scope. Even though it is not part of our scope, it is important to have this if the system is ever to expand. Since SQL databases are only type of databases that can follow the ACID properties, as per section 2.5, they do not describe Cassandra or MongoDB. However, they are defined by a different property, BASE. We did not take into consideration the BASE properties. The BASE properties describe features that are useful to have when a database needs to be expanded. Also, when the use cases for the system are depicting concurrent access to the database, then, databases described by these properties are useful to have. Even though expansion of the database could be a useful feature, it is not a part of our requirements. We therefore did not consider BASE as a measurement point for our scope.

Notably, our selection of databases consisted of one of each one from the three database categories of the CAP-theorem. Since PostgreSQL is a RDBMS, we can see that it is a combination of consistency and availability. Cassandra's and MongoDB's position can be seen in Figure 2.2. We chose the different databases partly because they were different in the CAP-diagram. This gave us the benefit of benchmarking a database from each category. We did not know which category that would suit our requirements, as such, the best procedure was to cover all categories.

In the results we can clearly see that Cassandra has the worst performance overall. Increasing the performance could have been done in different ways. We observed this in several papers that benchmarked Cassandra. These papers gave us an insight to how Cassandra's performance behavior as it did for us.

In the paper by Haughian et al. [28] is Cassandra performing better than MongoDB. However, YCSB's client was set up with eight threads, meaning that the client side of the operations could be done concurrently. If we would had used the same setup in our environment, the 1 million operations would be split up into smaller parts. This would increase the throughput and, in the end, would Cassandra have better performance. However, our use cases did not specify that we will have concurrent client access to the database, and therefore we did not configure YCSB's client to more than one thread.

This does not correspond to the result of similar papers where Cassandra has been benchmarked with the same tool. In the paper by Abramova and Bernardino [4], Cassandra is outperforming MongoDB. The setup that Abramova and Bernardino use is a virtual environment where the database is set up. Further configuration is not explicitly stated. Their results show that Cassandras execution time decreased as the number of operations got larger. The reason behind this behavior is not stated, but it is contrary to what we experienced in our results. Our results were that Cassandra linearly increased its execution time. For our experiments we did not alter or add any configurations to the YCSB's Cassandra benchmark other than the operationscount. This meant that we were using the standard configuration that YCSB provided. Therefore, we can make an educated guess, that the Cassandra configuration in the experiments done by Abramova and Bernadino, had been altered.

By changing the configuration of Cassandra, one could have seen better results. However, this was something we did not do. Increasing the possible number of threads would make Cassandra able to do parallelism on the operations, increasing the performance. This can be seen in the results of the paper by Abramova et al. [5], where the increase in threads resulted in faster execution times. Most importantly, the execution time got significantly faster, both with one single node and a six nodes cluster, when the threads increased from 1 to 6 threads. Therefore, we could have increased our number of threads to increase the performance of

Cassandra. This could have made Cassandra sound for the future, but doing this would have altered the results in unfair favor towards Cassandra. The other two databases would be needed to be altered in the same way. However, since the use cases did not specify scenarios where parallel access occurs, we did not experiment with this any further. Thus, we would not experience an increase in execution time if the implementation from the paper by Abramova et al. [5] would have been used.

We configured Cassandra to use one node. Since we ran the database locally this was the most logical setup for us. Each machine that is part of a server is usually set up with one node. The nodes will then use the IP addresses of the machines to communicate with one another. Setting up a node cluster would decrease the execution time of Cassandra. However, the DBMS in the final product will be run on a single machine. As such, the results would be skewed, as the results from the benchmark would show that Cassandra had the fastest execution time, even though that configuration of Cassandra would never be used by Axis. Another benefit from increasing the number of nodes would show in the performance of Cassandra, more so in the write operations. This is due to how clusters are configured in Cassandra, and Cassandras replication strategy. The use cases do not have a significant number of writes, the writes are few and sparse. The read performance for single separated reads, is of bigger importance, and these do not benefit from extra nodes.

It is important for the reader to understand that configuration of Cassandra is heavily relying on the use cases. Our purpose is not to discredit or diminish Cassandra in any way, Cassandra is a powerful system, that with the right configurations, can be a great database for a system. Since Netflix and Facebook, two global companies, famous for their amount of processing data, have Cassandra as their main database, we can claim that Cassandra is a great DBMS. Netflix can even achieve more than 1 million writes per second when using Cassandra [17]. During this paper we have not optimized Cassandra, which needs to be considered as the final recommendation is taken.

In Workloads that were update heavy, we could see that MongoDB outperformed the other DBMSs, if the number or operations where around 750 000 or lower. As the number of operations got bigger, so did the execution time. The reason behind this is not clear. According to Haughian et al. [28], MongoDB uses single threaded locking mechanism for all writes. This means that the document is locked from all operations until the write operation is done, to ensure atomicity. Suppose this is true, then the more operations that is executing, the longer will the execution time be for writes. In our results we would have seen an increase in execution times where writes are involved. This we did, but only for operations larger than 750 000. For this reason, we cannot solely say that this is the reason we are observing that behavior from MongoDB. Another property of MongoDB is sharding. Among the benefits of sharding, is the scaling of write operations. By using shards, one can scale the write operations between several machines, increasing the total write throughput [9, Ch. Sharding]. We did not configure sharding for MongoDB. This means that the write operations would have been affected for larger operations. This we did observe for operations that were bigger than 750 000. An educated guess would therefore be that the single server was on the verge of not handling the write operation. This is resolved by using sharding [9, Ch. Sharding], which we did not do. However, the final setup for the prototype will never be on several machines. As such we did not experiment with sharding in MongoDB.

MongoDB's reads were faster than Cassandra's and slower than PostgreSQL's. We were confounded by this result. We expected MongoDB to be slower, as we had not configured

MongoDB to use shards and since MongoDB used single threaded locking mechanism. However, the paper by Jan Sipke et al. [61], showed that PostgreSQL outperformed MongoDB, when it comes to multiple reads by a single client. The paper also showed that MongoDB outperformed Cassandra in the same category, which corresponded to that of our result. Unexpectedly, the remainder of the results did not correspond to that of ours. An explanation is not stated either to why one database would outperform one of the others. The paper does however state that "PostgreSQL is the best choice when … read performance is important" [61], which we can confirm with our results.

PostgreSQL overall performed better than the other two databases. When in regard to workloads with updates, however, PostgreSQL performed worse than MongoDB when the number of operations where less than a million. A first thought would be that MongoDB uses the RAM. MongoDB do use RAM for caching and keeps the most recent used data there. It does not, however, use the RAM for storing the query results.

PostgreSQL would be our final recommendation to Axis. We based our recommendation on the performance of PostgreSQL and PostgreSQL's compatibility to the requirements. We did not choose Cassandra because of the performance. Although MongoDB had a faster execution time than PostgreSQL for updates, we still did not choose MongoDB since the write operations from the use cases were few and sparse. PostgreSQL suited our use cases and requirements, which added to the reasoning behind the recommendation to Axis.

4.3 Related Work

The YCSB benchmarking tool has been used in several other papers and has become a very popular tool for comparing NoSQL databases and finding the database that is most suitable for their needs. The tool has been used by several companies [20, 23, 44], and in several academic research papers [4, 6, 51]. It is important to note that most of these companies have some sort of bias, since they can be affiliated with different NoSQL databases. For example, Datastax is one of the providers for Cassandra, where they have Cassandra as the clear winner in their benchmarking tests [20]. Similar, Thumbtack Technologies, the authors of [44] has strategic and/or commercial relationships with Aerospike, Couchbase, and 10gen (the makers of MongoDB).

Throughout the report we have compared our results to those of Abramova and Bernardino [4], where they compare MongoDB with Cassandra. In their paper, they did not use the default workloads, *D* and *E*, but instead implemented their own two workloads as a replacement that better fit their needs. They set up their benchmarking experiments on a Virtual Machine with different hardware compared to us.

In the original YCSB paper, Cooper et. al. [19], the authors compared four different NoSQL databases to illustrate the purpose and value of the implemented tool and highlight the tradeoffs between choosing different NoSQL databases. The databases they used were Cassandra, HBase, PNUTS, and a sharded MySQL database. The results that they provided were not the time it took to execute a workload for each database, as we did, but rather the latency as a function of the throughput (operations/sec).

There is another paper that uses the Go extension of YCSB (Go-YCSB). In their paper [62] they use Go-YCSB to determine if their new database scheme called Adaptive Layout Optimization of Raft groups (ALOR) increases the performance of distributed key-value stores,

and if so, how much more throughput in percentage that is, compared to the Raft protocol.

All these studies do not address our problem domain, which is to examine which database, NoSQL database or RDBMS, that is best suited for the needs of Axis to store and retrieve their video metadata efficiently.

Chapter 5
Conclusions

This report has investigated different types of databases, both NoSQL databases and RDBMS, to decide which database is most suited for the needs of Axis to store their video metadata. To determine this, we have used the popular benchmarking tool YCSB, where we used their predefined workloads to compare the execution time of the databases with a different number of operations, ranging from 10K to 1000K. The databases that we compared was Cassandra, MongoDB and PostgreSQL. The criteria for choosing a database was that it should be able to handle the requirements that was setup in cooperation with Axis at the beginning of the study, amongst those requirements were that the database should be able to handle semi-structured data.

The benchmark tests were run on the same computer that the databases were configured on. We only used one client thread to simulate how the database will be used.

We found that PostgreSQL performed, on average, best compared to the other databases, which leads to us recommending PostgreSQL to Axis for storing their video metadata. Axis has accepted our recommendation and will use PostgreSQL in the future.

Bibliography

[1] Serge Abiteboul. Querying semi-structured data. In *International Conference on Database Theory*, pages 1–18. Springer, 1997.

[2] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: from relations to semistructured data and XML*. Morgan Kaufmann, 2000.

[3] Serge Abiteboul and Victor Vianu. Transactions in relational databases (preliminary report). In *VLDB*, 1984.

[4] Veronika Abramova and Jorge Bernardino. Nosql databases: Mongodb vs cassandra. In *Proceedings of the international C* conference on computer science and software engineering*, pages 14–22. ACM, 2013.

[5] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. Evaluating cassandra scalability with ycsb. In Hendrik Decker, Lenka Lhotská, Sebastian Link, Marcus Spies, and Roland R. Wagner, editors, *Database and Expert Systems Applications*, pages 199–207, Cham, 2014. Springer International Publishing.

[6] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. Experimental evaluation of nosql databases. *International Journal of Database Management Systems*, 6(3):1, 2014.

[7] Rakesh Agrawal, Anastasia Ailamaki, Philip A Bernstein, Eric A Brewer, Michael J Carey, Surajit Chaudhuri, AnHai Doan, Daniela Florescu, Michael J Franklin, Hector Garcia-Molina, et al. The claremont report on database research. *ACM Sigmod Record*, 37(3):9–19, 2008.

[8] Amazon. Amazon simpledb. `https://aws.amazon.com/simpledb/`. (Accessed on 05/02/2019).

[9] Christian Amor Kvalheim. *The Little MongoDB Schema Design Book*. Learnpub, 1st edition, 2015. (Accessed on 03/22/2019).

[10] Apache. Apache hbase. `https://hbase.apache.org/`. (Accessed on 05/02/2019).

[11] Nick Bailey. Ycsb wiki - core workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads, 10 2010. (Accessed on 02/18/2019).

[12] Gabriele Bavota, Carmine Gravino, Rocco Oliveto, Andrea De Lucia, Genoveffa Tortora, Marcela Genero, and José A Cruz-Lemus. A fine-grained analysis of the support provided by uml class diagrams and er diagrams during data model maintenance. *Software & Systems Modeling*, 14(1):287–306, 2015.

[13] Robert Blumberg and Shaku Atre. The problem with unstructured data. *Dm Review*, 13(42-49):62, 2003.

[14] Bson specification. http://bsonspec.org/. Accessed: 2019-02-18.

[15] Craig Buckler. Sql vs nosql: The differences. https://www.sitepoint.com/sql-vs-nosql-differences/, 5 2017. (Accessed on 02/11/2019).

[16] Donald D Chamberlin. Early history of sql. *IEEE Annals of the History of Computing*, 34(4):78–82, 2012.

[17] Adrian Cockcroft and Denis Sheahan. Benchmarking cassandra scalability on aws. https://medium.com/netflix-techblog/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e, 11 2011. (Accessed on 03/11/2019).

[18] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[19] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[20] Datastax Cooperation. Benchmarking top nosql databases. https://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf.

[21] Db-engines ranking. https://db-engines.com/en/ranking/relationaldbms, 2 2019. (Accessed on 02/11/2019).

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[23] Diomin and Grigorchuk: Altoros Systems Inc. Benchmarking couchbase server for interactive applications.

[24] Erik Duval. Metadata standards: What, who & why. *Journal of Universal Computer Science*, 7(7):591–601, 2001.

[25] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[26] Christoforos Hadjigeorgiou et al. Rdbms vs nosql: Performance and scaling comparison. *EPCC, The University of Edinburgh*, 2013.

[27] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th international conference on pervasive computing and applications*, pages 363–366. IEEE, 2011.

[28] Gerard Haughian, Rasha Osman, and William J Knottenbelt. Benchmarking replication in cassandra and mongodb nosql datastores. In *International Conference on Database and Expert Systems Applications*, pages 152–166. Springer, 2016.

[29] Ian. What does acid mean in database systems? `https://database.guide/what-is-acid-in-databases/`, 6 2016. (Accessed on 02/13/2019).

[30] ihritik. What is semi-structured data? `https://www.geeksforgeeks.org/what-is-semi-structured-data/`. (Accessed on 05/02/2019).

[31] ihritik. What is unstructured data? `https://www.geeksforgeeks.org/what-is-unstructured-data/`. (Accessed on 05/02/2019).

[32] HV Jagadish and Frank Olken. Database management for life sciences research. *ACM SIGMOD Record*, 33(2):15–20, 2004.

[33] Mohammed-Ali Khan. Sql vs. nosql. *Retrieved February*, 5:2015, 2012.

[34] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 5–5. ACM, 2009.

[35] Neal Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2), 2010.

[36] Victor M Markowitz. Representing processes in the extended entity-relationship model. In *Data Engineering, 1990. Proceedings. Sixth International Conference on*, pages 103–110. IEEE, 1990.

[37] Microsoft. What is a transaction? - windows applications | microsoft docs. `https://docs.microsoft.com/en-us/windows/desktop/Ktm/what-is-a-transaction`, 5 2018. (Accessed on 02/28/2019).

[38] Stefan Mischook. Database driven websites: what are they and how are they built? `https://www.killersites.com/articles/articles_databaseDrivenSites.htm`. (Accessed on 02/22/2019).

[39] Mondodb documentation. `https://docs.mongodb.com`. Accessed: 2019-02-15.

[40] ABM Moniruzzaman and Syed Akhter Hossain. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*, 2013.

[41] Prakash M Nadkarni. What is metadata? In *Metadata-driven Software Systems in Biomedicine*, pages 1–16. Springer, 2011.

[42] Shamkant B Navathe. Evolution of data modeling for databases. *Communications of the ACM*, 35(9):112–123, 1992.

[43] Ameya Nayak, Anil Poriya, and Dikshay Poojary. Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*, 5(4):16–19, 2013.

[44] Nelubin and Engber: Thumbtack Technology Inc. Ultra-high performance nosql benchmarking.

[45] neo4j. Neo4j graph platform. `https://neo4j.com/`. (Accessed on 05/02/2019).

[46] Objectivity. Infinitegraph. `https://www.objectivity.com/products/infinitegraph/`. (Accessed on 05/02/2019).

[47] One-to-many relationship in mongodb. `https://docs.mongodb.com/manual/tutorial/model-referenced-one-to-many-relationships-between-documents/`. Accessed: 2019-02-18.

[48] Zachary Parker, Scott Poe, and Susan V Vrbsky. Comparing nosql mongodb to an sql db. In *Proceedings of the 51st ACM Southeast Conference*, page 5. ACM, 2013.

[49] Postgresql: About. `https://www.postgresql.org/about/`. (Accessed on 02/11/2019).

[50] Quassnoi. Emulating full outer join in mysql. `https://explainextended.com/2009/04/06/emulating-full-outer-join-in-mysql/`, 4 2009. (Accessed on 02/11/2019).

[51] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, 2012.

[52] Rajeshwari K. Rai. Intricacies of unstructured data. *EAI Endorsed Trans. Scalable Information Systems*, 4(14):e6, 2017.

[53] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw Hill, 2000.

[54] Vatika Sharma and Meenu Dave. Sql and nosql databases. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(8), 2012.

[55] Rolf Sint, Sebastian Schaffert, Stephanie Stroka, and Roland Ferstl. Combining unstructured, fully structured and semi-structured information in semantic wikis. In *CEUR Workshop Proceedings*, volume 464, pages 73–87, 2009.

[56] Sparsity-technologies. Sparksee high-performance graph database. http://www.sparsity-technologies.com/#sparksee. (Accessed on 05/02/2019).

[57] Stack overflow developer survey 2018. https://insights.stackoverflow.com/survey/2018/#technology-databases, 2018. (Accessed on 02/11/2019).

[58] Exadel Innovations Team. Old reliable: A history of mysql database. https://exadel.com/news/old-reliable-mysql-history/, 10 2017. (Accessed on 02/11/2019).

[59] Techopedia. Atomicity consistency isolation durability (acid). https://www.techopedia.com/definition/23949/atomicity-consistency-isolation-durability-acid. (Accessed on 03/22/2019).

[60] Bogdan George Tudorica and Cristian Bucur. A comparison between several nosql databases with comments and notes. In *Roedunet International Conference (RoEduNet), 2011 10th*, pages 1–5. IEEE, 2011.

[61] Jan Sipke Van der Veen, Bram Van der Waaij, and Robert J Meijer. Sensor data storage performance: Sql or nosql, physical or virtual. In *Cloud computing (CLOUD), 2012 IEEE 5th international conference on*, pages 431–438. IEEE, 2012.

[62] Yangyang Wang, Yunpeng Chai, and Xin Wang. Alor: Adaptive layout optimization of raft groups for heterogeneous distributed key-value stores. In *IFIP International Conference on Network and Parallel Computing*, pages 13–26. Springer, 2018.

[63] Geoffrey Weglarz. Two worlds data-unstructured and structured. *DM REVIEW*, 14:19–23, 2004.

[64] What's new in postgresql 9.4. https://wiki.postgresql.org/wiki/What%27s_new_in_PostgreSQL_9.4. Accessed: 2019-02-18.

[65] Wikipedia. Null (sql). https://en.wikipedia.org/wiki/Null_(SQL). (Accessed on 05/02/2019).

[66] Wikipedia. Peak signal-to-noise ratio. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio. (Accessed on 03/01/2019).

[67] Hugh E. Williams and Saied M.M. Tahaghoghi. Entity relationship modeling examples - learning mysql [book]. https://www.oreilly.com/library/view/learning-mysql/0596008643/ch04s04.html, 11 2006. (Accessed on 03/04/2019).

[68] Carey Wodehouse. Sql vs. nosql: What's the difference? https://www.upwork.com/hiring/data/sql-vs-nosql-databases-whats-the-difference/, 5 2016. (Accessed on 02/11/2019).

[69] Randy Jay Yarger, George Reese, and Tim King. Mysql & msql. https://docstore.mik.ua/orelly/linux/sql/index.htm, 1999. (Accessed on 02/11/2019).

[70] Ycsb. https://github.com/brianfrankcooper/YCSB. Accessed: 2019-05-02.

Appendices

EXAMENSARBETE A Comparison in Performance Between a Selection of Databases
STUDENTER Andreas Ohlsson, Mikael Persson
HANDLEDARE Jörn Janneck (LTH)
EXAMINATOR Flavius Gruian (LTH)

Val av databas - en inte så enkel uppgift

POPULÄRVETENSKAPLIG SAMMANFATTNING **Andreas Ohlsson, Mikael Persson**

Stora mängder metadata måste sparas för att kunna användas i utvecklingssyfte. Genom att använda sig av databaser, kan metadatan effektivt sparas. Men vilken databas passar bäst för vårt system? Det är vad vår uppsats har svarat på.

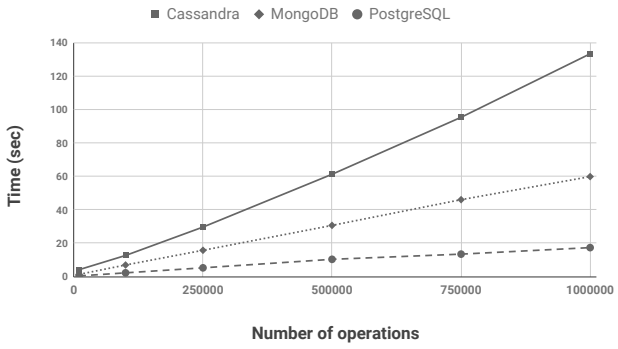
Varje bild tagen med en kamera innehåller mycket information. Upplösning, bildkvalité och typ av bild kan vara sådan information. Eftersom en video består av flera bilder per sekund, kan vi förstå att en video sekvens kan innehålla enorma mängder information. Denna information, eller ofta kallad metadata, måste sparas och hämtas ut enkelt så att en jämförelse kan ske.

För oss var lösningen att använda en databas. Databaser har använts i många stora företag, så som Facebook, Instagram och Google. Detta för att det går snabbt och effektivt att ta ut information. En användare vill inte sitta och vänta på att Google ska ladda klart en sökning. Den stora frågan för vår masteruppsats var: Vilken databas passar den metadata vi har?

Vi kom fram till att databasen PostgreSQL passade bäst för våra kriterium. PostgreSQL är öppen källkod, vilket betyder att vem som helst kan skriva kod till denna databas. Vi tittade inte bara på en databas, utan två andra. Dessa två andra databaser, Cassandra och MongoDB, passande mindre bra för vårat system.

I framtiden kan vår studie användas av företag eller personer som behöver välja en databas för deras metadata. Det är också intressant ur ett forskningsperspektiv, då vår kombination av jämförelse mellan PostgreSQL, Cassandra och MongoDB är speciell.

I figuren kan vi se att PostgreSQL hade den snabbast tiden för en av de olika belastningarna. Vi kan se att Cassandra, vilket är en databas som Facebook använder sig av, inte presterade så bra.



Figur: Visar att PostgreSQL hade den bästa körningstiden i det flesta fall av alla databaser

Vi använde oss av ett jämförelseverktyg kallad Yahoo! Cloud Service Benchmark (YCSB). Ett jämförelseverktyg används för att på ett säkert och beprövat sätt få en så jämn jämförelse mellan databaser som möjligt. YCSB var också enkelt att använda.