A Comparative Analysis of MongoDB and Cassandra

A thesis presented for the degree of
Master of Science

Kavita Bhamra

Department of Informatics
University of Bergen

November 2017

Abstract

NoSQL is a group of database technologies that emerged due to the limitations of relational databases. The number of NoSQL technologies has increased over time to encompass hundreds of different technologies. The many NoSQL databases are, unlike relational databases, not based on a standardised data model, query language, or a common way of thinking. Also, the NoSQL field brings forth some new concepts and challenges that were not present in the relational context. It can, therefore, be difficult to choose the right NoSQL technology for a particular application. This thesis analyses and compares two specific NoSQL database systems, MongoDB and Apache Cassandra, to simplify the selection process. There are several important factors to consider when deciding on a NoSQL technology. The purpose of this comparison is to outline what factors are useful to consider when selecting a technology. The discussion is mostly based on an extensive literature review. In the end, several factors were stressed, like the potential technology's data model, CAP classification, and available documentation, to name a few.

Acknowledgements

First, I would like to express my gratitude to my supervisor, Kjell J. Hole, for giving me the opportunity to learn so much. His patience and guidance helped me immensely throughout the entire process.

I would also like to thank Simula@UiB for providing me with a workplace and for allowing me to be a part of an inclusive and socially stimulating environment. It has made my last year of higher education enjoyable.

Last but not least, I want to thank my family for their constant support, encouragement and help.

List of Figures

List of Code

Table of contents

Chapter 1

Introduction

Databases are an essential part of most large applications, and relational databases have long been at the forefront of data storage technologies. Over time, several alternative storage solutions have evolved, many of which are referred to as NoSQL technologies [1]. The subject of this thesis is to explore the field of NoSQL further.

This chapter will present the motivations for studying NoSQL databases, explain the overall goal of this thesis, its limitations, and targeted readers. Lastly, an overview of the thesis structure will be given.

1.1 Motivation

Relational databases have long been considered a staple among data storage technologies, but have struggled to meet the demands of modern applications. These applications are consuming ever-growing amounts of data with varying structures. These fast changes in data volumes and variety do not work well with relational databases and SQL.[1] As a result, alternative storage technologies were developed, including NoSQL technologies.

NoSQL solutions are becoming more and more popular, and the term refers to a set of database technologies that break away from relational databases in

[1]SQL stands for Structured Query Language, and is the language associated with relational databases.

several ways. Today, there are hundreds of different NoSQL solutions [2], and in contrast to relational databases, they do not impose a standard data model nor language. Additionally, the technologies are built on different concepts, challenges, and ways of thinking. Furthermore, they are suitable for different use cases. Knowledge of NoSQL means that developers are no longer limited to the traditional way of storing and handling data, and the many NoSQL technologies open up a wide variety of possible storage solutions.

The number of storage options and the amount of available information is overwhelming, which can lead to difficulties in choosing the right technology. One way to understand highly different technologies is by studying a selection of them closely, and that is the subject of this thesis. The overall aim is to explore and understand NoSQL through a comparison of MongoDB and Apache Cassandra, two NoSQL technologies. Most of the existing comparisons are geared towards the needs of larger enterprises with a focus on performance related issues of large sets of data.

1.2 Research objectives and purpose

The main objectives of this thesis are to analyse and compare two specific NoSQL technologies with an emphasis on different use cases of general interest. This entails a top-down study of each technology based on several aspects, as well as a comparison based on the same aspects. The purpose of this approach is not only to provide the reader with a good understanding of the selected technologies and when to use them, but also to outline a framework to simplify the selection process of choosing the right NoSQL technology for a given use case.

1.3 Target group

The main audience of this thesis is other computer science students with little to no knowledge of NoSQL databases, as well as professional developers needing an easy-to-understand introduction to these technologies. It is an advantage if the reader knows relational databases.

1.4 Scope of the study

To ensure that the work in this thesis remains relevant for an extended period of time, it focuses first and foremost on general concepts, rather than the implementations. Furthermore, the thesis does not address the performance and data usage characteristics of the technologies. Though these are important concepts, the former is covered by available documentation, while the latter is geared towards more specific use cases.

1.5 Thesis structure and outline

The rest of the thesis is organized into six chapters which are described as follows.

Chapter 2 starts by reviewing the relevant background materials before it transitions into describing the field of NoSQL. The purpose of Chapter 2 is to set the context and foundation for further discussion. Chapter 3 describes and justifies the chosen research method and its application, along with the selection of NoSQL technologies. Chapters 4 and 5 present the information gathered about Apache Cassandra and MongoDB. Chapter 6 is where the comparison ensues based on the information from the two previous chapters. Chapter 7 concludes the thesis with a summary of the entire research, final remarks, and discussion of possible future work.

Chapter 2

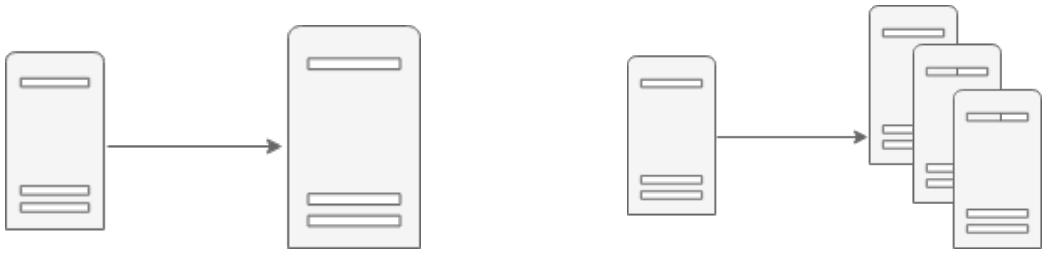
Background and basic concepts

Relational databases have dominated the storage field for many years. In recent years, however, it has become clear that this traditional approach to data storage is facing serious challenges in the world of web, cloud, mobile communication, social media, and Big Data applications. All these applications and their data content change on a frequent basis. In particular, the applications must collect and process ever-growing volumes and varieties of data to extract new information. Because NoSQL technologies are direct results of the limitations companies experienced with SQL, an introduction to these limitations is useful to understand how NoSQL differs from relational databases.

This chapter first presents the problems and challenges related to relational databases that emerged along with the growth and popularity of modern Web 2.0 applications. Further, NoSQL and its related concepts will be introduced. The goal of this chapter is to set the foundation for further discussion throughout the thesis.

2.1 Background

Relational databases have been the de facto standard for data access and storage for several decades [3]. The widespread popularity of relational databases has been due to their general ease of use and the rich set of supported features. The popularity established relational databases as a mature technology that



(a) Scaling up. (b) Scaling out.

Figure 2.1: A visual representation of scalability.

is supported by many vendors and has a big and experienced user-group. However, newer applications have different demands. The ever-increasing data volumes and processing needed to extract and handle more and more information require databases that are highly scalable, that is, able to store and provide rapid access to increasing amounts of data.

Scalability can be handled in two ways; scaling up and scaling out [4, p. 30]. Scaling up, also called vertical scaling, refers to upgrading the resources, by means of, adding more processors, memory, and storage to a single server (see Figure 2.1a). However, this can only be done up to a certain limit due to high costs, as well as hardware limits. On the other hand, scaling out or horizontal scaling refers to increasing the number of resources by adding more servers (see Figure 2.1b). Since only the first option applies to relational databases, they have a restricted ability to scale.

With the increased amount of data and users, it is important that the servers can handle the corresponding traffic load and do so efficiently with minimal latency. Also, many applications are being used 24/7 by users spanning different geographical locations, so constant *availability* and *reliability* are necessary features. Availability refers to the ability to handle all client requests, and reliability to ensure that the data is accessible at all times.

The issue that relational databases pose is that they favour the centralized, monolithic architecture pattern, as they are designed to be deployed to a single server or rely on clustering with shared storage [5], [6]. In this case, shared storage refers to a shared disk architecture [7, p. 16], that is, when multiple nodes are using the same underlying disk. The downfall is if either the server or the shared storage fails, all the data becomes inaccessible. In other words: this is a single point of failure that not able to deliver all-time availability. It is, however, possible for relational databases to become dis-

tributed by adding some enhancements, but these are known to have negative and unpredictable effects [1, p. 8][8].

A second issue is that the data that is generated today comes in all shapes and sizes. A data model is defined as "the model by which the database organizes data," and usually contains a set of fields that describe the content of the data [1]. In the relational data model, everything is viewed through tables of rows and columns which are fixed by a predefined schema. A schema is a skeleton of the content or the format describing the types and names of the data values. This set-up works very well for structured data, even though developers have experienced a problem called "the impedance mismatch" ever since the rise of the object-oriented programming paradigm [1, p. 5]. The impedance mismatch refers to the conceptual (and to some degree technical) difficulties that occur when mapping objects to the relational data model. The problem has led to the development and usage of object-relational-mapping frameworks to address this task. It is an additional overhead on the development process and has been described as "inefficient at best, problematic at worst" [6, p. 5].

A third significant issue is the relational data model; it does not align well with unstructured or semi-structured data [6, p. 3]. That is, data that does not follow a fixed structure with a set number of fields, such as blog posts, surveys with open-ended questions, and social media posts. Additionally, the data may change format repeatedly over time. Essentially this is data that cannot be stored nor presented using the relational model without preprocessing. However, since the relational schema needs to know every aspect of the data upfront, this data model becomes rigid and inflexible to change.

One proposed solution for SQL to handle varying number and types of fields is through the use of "sparse data." The idea is that the potential fields are added, and for those data entries that do not have a value for that field, that column is blank. A problem arises because this column will be allocated disk space. Not only does this take up space, the use of too many blank columns can negatively impact query performance.

Another motivation for NoSQL is that technologies are constantly changing, resulting in more and more demands for shorter development cycles. Since relational databases are based on predefined schemas and resent change, they do not work well with modern agile development approaches [3]. In fact, relational databases have been described as a "roadblock to agility" [6].

To summarise this section: the main challenges are scalability and unstructured data. The limitations of SQL databases are mainly due to their underlying design choices. While the choices made sense when the data sets were structured and were of limited size, the occurrence of Big Data has made it impractical to use relational databases.

2.2 NoSQL

The previous section explained how relational databases struggle to meet the data requirements of modern applications. The frustrations are due to the volume and variety of data. The general solution for many companies has been to make a new database technology that is non-relational. This approach has resulted in many database technologies, collectively known as NoSQL databases.

This section will delve into the world of NoSQL, first by introducing a more concise definition of the term and its origin, describing the core characteristics of NoSQL databases, outlining the different types of databases, and lastly introducing NoSQL related concepts and challenges.

2.2.1 Term and origin

It has been stated what NoSQL refers to, but not where the term came from nor what it means.

NoSQL is a coincidental term with somewhat ambiguous meaning. It can be said that NoSQL is short for "Not only SQL" (and not "No SQL" which it easily can be misconstrued as) [9]. One way to interpret this term in a meaningful way is to view all the NoSQL databases as alternatives to SQL-based databases [10, p. 6]. Furthermore, the term was adopted in 2009 after a developer meetup where existing solutions to "open-source, distributed, non-relational databases" were presented [9].

In the early 2000's, the biggest companies at the time, Google and Amazon, were the first to experience problems relating to database scalability. The companies developed their own storage solutions, first Google with BigTable and then Amazon a couple of years later with DynamoDB. In 2006 and 2007,

papers describing these solutions were published, and the concepts have since influenced and inspired many other NoSQL database systems [1]. As of 2017, there are more than 225 NoSQL database solutions [2].

2.2.2 Core characteristics

So far, NoSQL has been defined as concisely as can be, but we have not addressed what exactly qualifies databases as NoSQL databases.

Fundamental characteristics have been outlined [1, pp. 9–11]. However, they are more general guidelines and may not be present in all NoSQL database systems. The main characteristics are as follows:

- Data models in NoSQL databases are not relational, nor is SQL used. In fact, there is no standard language defined for all database technologies. Instead, different languages are used in different technologies.
- Also, NoSQL databases are schemaless, which results in a more flexible data model. Change is accommodated in an efficient way that supports the storage of any data at any time without explicitly defining a structure beforehand.
- NoSQL databases are designed for horizontal scalability, or a distributed, shared-nothing architecture where the nodes do not share any memory or disk and only communicate through the network [7, p. 19].
- The majority of NoSQL databases are open-source.

2.3 Types of NoSQL databases

There are too many NoSQL database systems to discuss all of them. Fortunately, they can be classified into four broad groups based on their fundamental data model [11]. Each type has a different data model, as well as a different set of advantages and disadvantages.

1. *Document* databases store and retrieve data in the form of a document. Documents are typically in one of the common formats, such as JSON

or XML. The two most popular examples of NoSQL databases that belong to this genre are MongoDB [12] and CouchDB [13].

2. *Key-value* databases have the simplest data model consisting of a key and its corresponding value where the value is the data being stored. The idea is almost the same as a hash table. Popular implementations include Couchbase [14], Riak [15], and Redis [16].

3. *Column-oriented* databases store data in columns (as opposed to the rows from the relational data model). Cassandra [17] and HBase [18] are two well-known implementations using this data model.

4. *Graph* databases are entirely based on graph theory with directed graphs of vertices and edges and are therefore suitable for representing relationships between elements. Neo4J [19] and OrientDB [20] are known examples of graph databases.

This classification is not definitive. While four main types of databases have been presented, there exists some disagreement as to how many types there are. Some claim there are three main types (disregarding key-value based systems as a separate type), while others claim there are five types by considering search-oriented systems as a fifth type. It can also be mentioned that there are hybrid-models that combine the aforementioned data-models, like OrientDB that supports document, graph and key/value systems.

The rest of the chapter introduces some technical terms and concepts that are needed to discuss database technologies later.

2.4 Distributed NoSQL systems

One of the common characteristics of many NoSQL systems is that they are designed for horizontal scalability, which refers to scaling out the database by adding more servers. The system is distributed when the data is spread across multiple servers. Such distributed NoSQL systems bring a different set of challenges and concepts, and these will be presented in the following text.

2.4.1 Consistency and the CAP theorem

The main tasks for all kinds of databases are to store and retrieve correct data efficiently. The ability to do this is related to data consistency, availability, and scalability. Consistency essentially means that after data is written to a database, any subsequent requests for that data will retrieve the newly added value.

In the context of distributed NoSQL databases, consistency has a particular meaning. Traditional relational databases provide *strong consistency*. Strong consistency is equal to ACID-compliant consistency, that is, the properties Atomicity, Consistency, Isolation, and Durability ensure data is guaranteed to be consistent at all times. NoSQL systems are, however, for the most part *eventually consistent*, which means that reads may not reflect the latest update, but will reflect the change sometime in the future. Eventual consistency is a weakness because some systems are dependent on consistency at all times, e.g., banking applications.

This is an example of one of the trade-offs that can arise when adopting a NoSQL database. The trade-offs are nicely encapsulated in the CAP theorem by Eric Brewer [1, p. 53]. CAP stands for Consistency, Availability, and Partition tolerance, and the theorem states that only two of the three features can be obtained at the same time (see Figure 2.2) [1, pp. 53–56]. In a distributed system, only two of the following features can be guaranteed:

- *Consistency* (C) is achieved when the same copy of data is stored across servers.
- *Availability* (A) is achieved when all request will receive a response.

- *Partition tolerance* (P) is achieved when the database system continues to work even when a network partition occurs.

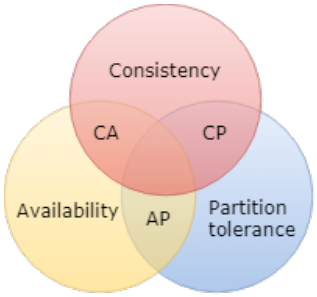


Figure 2.2: CAP theorem

All of the above are desirable properties. The theorem states that both consistency and availability can not be guaranteed simultaneously in a partition-prone network. Since it cannot be assumed that networks are reliable and never fail, not selecting Partition tolerance is not a viable option. That leaves two real options to choose from, Consistency and Availability. Therefore, the possibilities left are availability and partition tolerance (AP), or consistency and partition tolerance (CP). Thus, a sacrifice has to be made depending on the needs of the application. It should also be noted that the CAP theorem is easily misinterpreted and applied wrongly. That is, the CAP theorem is only applicable when a new partition occurs. At all other times, consistency and availability can both be achieved.

Finally, two other terms that frequently show up in the context of NoSQL databases are sharding and replication. Distributing data is accomplished by using these techniques. Sharding is the process of putting data on different servers and replication is the process of duplicating data across multiple nodes.

2.4.2 Replication

Replication is the process of duplicating data in a distributed system so that all the servers are storing and maintaining the same copies of data [1, p. 40]. The purpose of replication is to ensure that data is available in case one or more servers fail. This approach represents a shift in the traditional way

of thinking about databases, as adding redundancy to the dataset is now a way to help ensure availability. This is illustrated in Figure 2.3. The servers involved have different roles, and the replication of data depends on these roles. There are two replication methods: Master-slave and Peer-to-peer [1, p. 46].

- In the *Master-slave* setup, one server is the master, and the rest are slaves. The application uses the master directly, while the slaves just update their contents to the master's content by listening in the background. If the master fails, one of the slaves is usually elected to take over.
- In *Peer-to-peer*, every node is equal and all nodes service requests. They all synchronize with each other continuously.

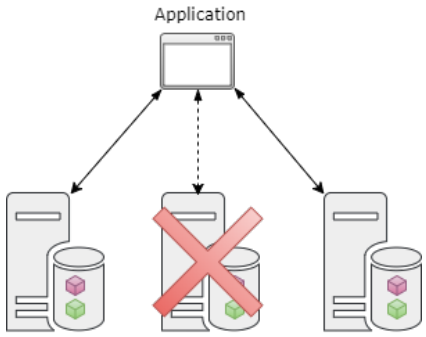


Figure 2.3: Example showing how replication ensures data availability through network failure

2.4.3 Sharding

Sharding is the process of scaling out databases and refers to the method of splitting up and spreading data across servers, illustrated in Figure 2.4. The purpose of sharding, other than expanding the storage capacity, is to improve performance by balancing workload and efficiency to handle more requests [1, pp. 38–39]. There are different strategies for how data is to be distributed, and each strategy has its advantages and disadvantages. Fortunately, most NoSQL databases do support auto-sharding [1, p. 39]. This way the database figures out how and where the data is distributed. Sharding

does add complexity to the architecture, but makes no difference from the application's point of view, and is in most cases straight-forward to use.

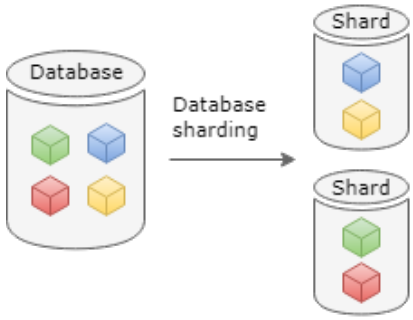


Figure 2.4: A visual representation of sharding

Summary

This chapter has introduced the basic concepts of NoSQL, from where and why the technology started to what it has become. The chapter first described the challenges and limitations relational databases face with the growing amount and variety of data. Some of the identified weaknesses have always been there, and are being amplified by recent demands, while others are rising because of the incompatibility of the traditional data model with newer application requirements. Key points include relational databases inability to scale well and to handle big volumes of data of varying structure in an efficient way.

NoSQL emerged as a set of database technologies that each represents a solution to these problems. The chapter transitioned further into explaining the field of NoSQL regarding definition, origin, the different types of NoSQL databases, and their shared characteristics. The distributed architecture of NoSQL databases makes up an essential part of the technologies, and thus its related concepts were introduced; sharding as a way to achieve data distribution, and replication as a way to achieve high availability by adding redundancy and the notion of consistency. Also, consequences of this distributed nature were summarised by discussing the CAP theorem.

Chapter 3

Methodology

A research method can be defined as a way to study a phenomenon "to discover new information about it, or understand it better"[21]. There are various methods with different purposes, but the one common goal for all of them is to obtain information through some data collection and analysis. The choice of method(s) is, therefore, determined by how the researcher plans to collect, process, and analyse data to answer the stated question(s).

The current chapter describes the research method for this study and the reasons for choosing it. The content of the chapter is divided into two parts. The first section describes and justifies the research method, and the criteria used to compare the database technologies. The second section provides an in-depth description of the research method.

3.1 Method

As stated in Chapter 1, the aim of this study is to compare two NoSQL technologies. There are three ways how such a comparison can be made. One option is the theoretical approach where the technologies are studied and evaluated based on a literature review. The second option is the practical, experimental approach where performance-related issues like endurance, durability, speed, latency, and operations are tested. A third and final option is some combination of the aforementioned two.

The research method chosen for this study is the theoretical approach, almost

entirely based on a literature study.

A literature review is a research method which is based on existing literature, that is, research publication written by other authors. A review of a collection of articles analyses previously published information in a novel way to gain insights not readily available by reading one or a few of the articles. The resulting findings are a presentation of the collected data in a manner that is different from before [22].

There are three main reasons why a literature review was chosen. Firstly, the experimental approach focuses on large data volumes. The target audience of this thesis is computer science students and novice developers. The practical approach addresses performance issues that are associated with massive data loads (in the context of "Big Data"), which is not likely to be an issue for the target group. That is not to say they are not important aspects, just not of highest priority in this context.

Secondly, while experiments with large data sets would be interesting, it was concluded that it would take too much time to implement such experiments, along with certain requirements for both data loads and multiple machines to ensure validity and reliability. Thus, the available time was better spent doing a thorough literature study.

The third, and probably most important reason is that the practical approach has a rather narrow focus of limited interest to students. The theoretical approach, on the other hand, is more versatile because it makes it possible to address many different aspects, thus, opening up for a broader perspective. This is arguably more favourable to novices as one can provide an all-round presentation of a technology covering aspects such as data models, architectures, languages, and communities.

There is, however, a limit to how much can be understood about a topic from just reading about it. Though this study will not directly include any verifiable implementations, both technologies will be partly implemented and experimented with to obtain a thorough understanding. The examples that are included in the later chapters are sample snippets from the author's testing and implementation.

3.1.1 Choice of technologies

This study targets MongoDB, a document database, and Apache Cassandra, a column-oriented database, as the technologies to be studied and compared. These technologies have been chosen due to a number of reasons.

MongoDB and Apache Cassandra have been the most popular NoSQL database technologies during the last couple of years. The technologies are in the 5th and 8th place on the list of the ten most used database management systems in the world (including relational databases) [23]. This popularity leads to favourable implications for a novice getting started with the technologies, including established online support, expert communities, reading materials and documentation. It also makes the probability of ever using these technologies higher and improves the relevance and the value of the paper.

Another significant factor for choosing the database technologies is that they are different types of NoSQL databases with fundamentally different data models and architectures. The comparison will highlight the large technology differences and their effects, and provide the reader with a solid basis for choosing the right database technology for a particular solution. Other factors that influenced the choice of databases are that both technologies are open-source and support multiple software platforms.

3.1.2 Comparison criteria

There exists an abundance of information about the relevant technologies, from technical system details and specifications to deployed concepts and underlying algorithms. To compare the technologies, the following system aspects were chosen:

- Data model
- Query model
- Consistency model
- CAP classification
- Ease of use
- Available APIs

- Platform dependency
- Available resources such as documentation and community
- Security

The selected aspects were chosen to provide an overall insight into the technologies with an emphasis on aspects that are important to consider when choosing which technology to use for a particular system. The study focuses on information of practical interest when implementing a solution using a database technology. The selection criteria may seem superficial, but they are relevant as they represent the stage most people will be at when considering which database technology to choose. As with most comparisons, parts of this technology comparison will be subjective, especially since it is carried out by a single individual.

3.1.3 Materials and resources

Because this study is heavily based on existing literature, this section provides information about the reading materials and the other resources used.

The literature study in this thesis is based on relevant books, journal articles, white papers, blogs and other web sources. The topical journal articles were obtained through searches in IEEE Xplore's online digital library. Relevant books were, for the most part, accessed through University of Bergen's Library. Since certain parts of the technologies continue to change, it was necessary to read vendor blogs and white papers, and technical documentation found on the Internet to get up-to-date descriptions of the technologies.

The comparison of the database technologies is based on MongoDB version 3.4 and Cassandra version 3.3.

3.2 Process

The components needed to carry out a comparative study have been presented. This section presents the overall method used to compare the two database technologies.

We first study the selected databases, Cassandra and MongoDB, as whole systems, before dividing the technologies into smaller parts. The analysis focuses on the system architecture and the structure of the parts. We will also study the database terminologies, central concepts, and query languages. This is to provide an in-depth explanation and overview of the selected technologies.

Then the comparison will ensue. The criteria will be discussed for each database technology and the technologies will be compared to highlight the factors that should be considered when deciding on a technology. This part is the crux of the thesis.

Summary

This chapter has explained the research method selected to compare the two database technologies Cassandra and MongoDB.

The following two chapters will present the information gathered about the selected technologies, and thus make up the main building blocks for the final comparison.

Chapter 4

Apache Cassandra

Cassandra is the second most used NoSQL database technology today [23], with major companies like Apple with over 75 000 Cassandra nodes and Netflix with 2500+ nodes, in addition to 1500 other companies [17].

Cassandra is a column-oriented, open-source database, with a data model heavily influenced by Google's BigTable, and a distribution model based on Amazon's Dynamo [24, p. 14]. The database technology is written in Java, and Cassandra was initially developed by Facebook for their inbox search, and released in 2008 [24, p. 14]. In 2009, it became an Apache project [24, p. 24], often referred to as Apache Cassandra. The name Cassandra stems from Greek mythology.

This chapter takes a detailed look at the Apache Cassandra with a focus on the data model, architecture, consistency model, and query language.

4.1 Basic data model

In Cassandra, the basic structure used to store data is based on using columns. The overall data structure consists of a keyspace, column families, rows, and columns [24, pp. 46–47].

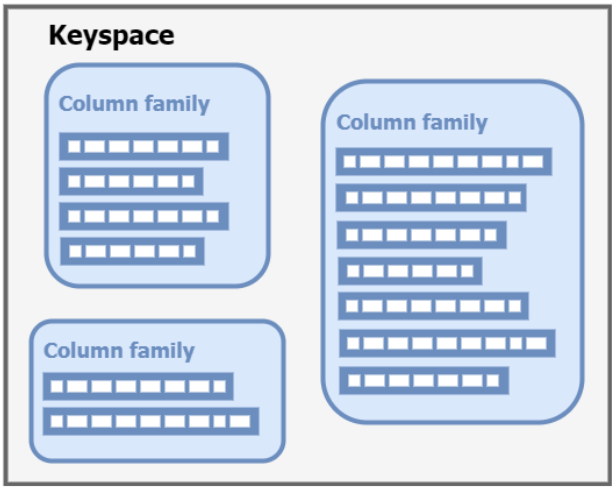


Figure 4.1: Example of a keyspace consisting of column families

- The *keyspace* is the outermost container for the entire dataset, hence, it basically corresponds to the entire database. Keyspaces consist of column-families, see Figure 4.1.
- A *column family* corresponds to a table in relational databases and consists of rows. An example of a column-family is included in Figure 4.2 where the column family contains various data about fruits.

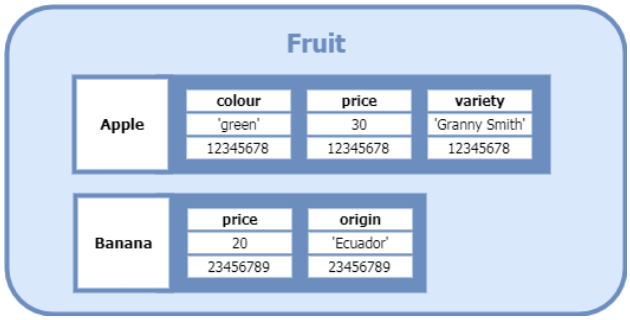


Figure 4.2: An example of a column-family containing two rows

- A *row* typically represents an entity. The row consists of columns, where each column corresponds to a property or attribute of the entity. Since the columns are defined per row, these rows can have different names, changing number of columns, and varying data types. Hence, each column is limited to its row and does not span all rows like in

relational databases. Each row also has a unique identifier, known as a *row key* or *partition key*. See Figure 4.3 for a visual representation.

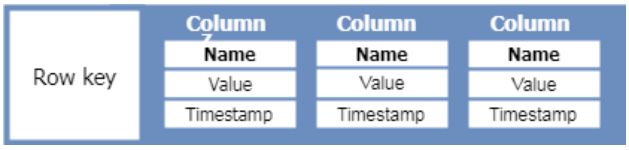


Figure 4.3: The structure of a row in Cassandra's data model

- Each **column** contains a name-value pair together with a timestamp. The timestamp is added by Cassandra, and is used to ensure consistency when there are multiple copies of the same data.

Note that in Figures 4.1 and 4.2 the rows of the column families vary regarding size and content, that is, they contain a different number of columns and different data. This illustrates that Cassandra's data model has a *flexible schema* where the rows within the same column family have different columns.

While the presented data model is quite simple, Cassandra's data model has a rich structure that provides several enhancements. Examples of such extensions are super columns [24, p. 53], various keys, and concepts like collections to store multiple elements within a single column.

It is tempting to draw an analogy to the structure of relational databases as most of the wording is similar, but that would be wrong. A more precise description is obtained by relating a column-family to a nested map-structure, where the outer map's key corresponds to a row key, and the inner map has column-field as its key. Something along the lines of

```
Map<RowKey, Map<Fieldkey, value>>
```

However, this is just a low-level view of how a single unit of data is stored in Cassandra. To understand Cassandra's design as a whole, the architecture must also be examined.

4.2 Architecture

Cassandra is inherently designed to be a distributed storage system. To understand and discuss the distributed structure, some terminology must be defined.

Let a *node* represent a server where data is stored (and for the rest of this chapter; running an instance of Cassandra). A *cluster* is a collection of nodes.

4.2.1 Data distribution

A cluster of nodes in Cassandra is usually referred to and visualized as, a ring (see Figure 4.4). Cassandra works by distributing all the data from the keyspace evenly across all the nodes. Each Cassandra node in the cluster stores a subset of the data and is assigned a range of hashes for which it is responsible. When data is inserted into the database, the row key of each row is sent to the partitioner. The *partitioner* is a function that computes the hash value of the row key. This resulting hash determines which node is to contain that row. The row is then stored into a *partition* on that node. Randomness is obtained by using hashing, which in turn ensures fair load balancing across the nodes.

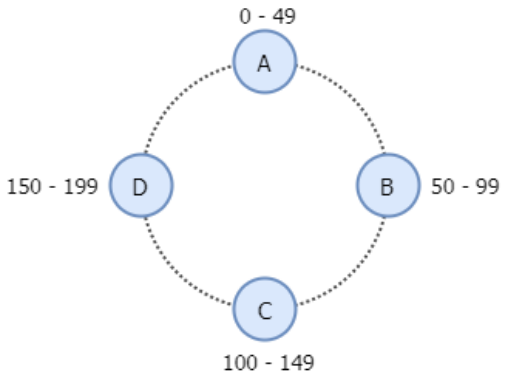


Figure 4.4: Example of a Cassandra ring

In Figure 4.4, the four nodes (A–D) are each given a key value range, which is marked beside each node. For example, rows with hashed key values in

the range of 0-49 will reside in node A, and similarly for the rest. Hence, a row with a hashed key value of 135 will be stored in node C.

All of the nodes in the ring are equal, and can, therefore, receive all types of requests. The client will issue his request to an arbitrary node without knowing if that node contains the data it is seeking. Cassandra uses the idea of a *coordinator* node. The node that receives a query from a client is the coordinator for that specific operation and is responsible for facilitating the exchange between the node containing the response to the query, and send the result to the client. From the client's point of view, the interface looks the same whether there is one single node or a cluster of nodes because the client always issues the request to a single node. Thus, the coordinator simplifies the retrieval process.

4.2.2 Replication

Cassandra uses peer-to-peer replication to ensure high availability. Replication in a cluster is defined per keyspace and defined by two parameters; the *replication strategy* and *replication factor*. The replication strategy refers to the algorithm that decides at which nodes to store copies of rows. The replication factor states the number of copies of each row to be stored in the cluster. The copies are stored following the clockwise direction, and count the initial data to be written as a copy. So, a replication factor of N will write the data to one node, and replicate to the $N - 1$ next neighbouring nodes.

These replicas are especially useful if a node goes down, or if it is unable to address the request in any other way. Then the other nodes will be able to step in to ensure that no loss of data occurs in the first case, or respond to the request. To be able to detect a failure, there needs to be interaction between the nodes. Cassandra has solved this challenge with the *gossip protocol* through which the nodes in the cluster communicate with each other, and that runs every second [25]. Each node exchanges information about itself and all the information it has regarding the other nodes in the cluster with at most three other nodes. The name of the protocol represents the protocol's inner workings as each node learns about every other node in the cluster through communicating with only a small subset of them. Not only is the protocol efficient in terms of information gathering purposes, but it is also used to detect failures and load balance requests. A failure is typically

detected when a node is not partaking in the gossip a fixed number of times.

Peer-to-peer replication has an added advantage of no single point of failure. Another favourable consequence of the peer-to-peer structure is that it is decentralised since all nodes are equal, all of the nodes can handle requests.

Now that we have covered Cassandra's overall architecture, we can focus on the important implications of the architecture.

4.3 CAP classification and consistency

Cassandra's default configuration classifies it as AP [26], available and partition-tolerant. According to the CAP theorem, consistency is then sacrificed for guaranteed high availability (partition tolerance is a non-selectable option for distributed databases). As seen in the previous section, the distributed, peer-to-peer structure ensures that any node can handle requests and no data loss will occur if a node goes down, hence providing availability at all times.

However, this does not mean that there is no consistency at all. Cassandra has a special feature regarding consistency, that is, Cassandra has tuneable consistency [26]. Replication factor and consistency level can be set in such a way that the user can choose between strong consistency and eventual consistency. Due to the databases distributed structure, the latter can affect the average response time negatively. Therefore, the trade-offs must be considered. If we take a read-operation as an example, with a consistency level specified to be `ONE`, then the response to the request will be retrieved from the closest node. This option provides high availability at the cost of possible stale data.

4.4 Query model and language

So far, a theoretical overview of two of the most important components of Cassandra has been introduced. In the following section, the query model and language will be presented through a practical approach. A query language refers to the language that facilitates storing, changing, and retrieving data, while a query model is a conceptual model related to querying. The intention is to get an understanding of how the data model and queries are

implemented.

4.4.1 CQL

CQL stands for Cassandra Query Language [27], and is the standardised language for interacting with data stored on one or more Cassandra nodes.

This language is accessible through `cqlsh`, a command-line tool included by default in all Cassandra distributions and application language drivers. Drivers are available for 13 languages [28], and Cassandra supports Linux and macOS (formerly OS X) platforms [29]. The APIs provide means for applications to connect to the database and issue CQL queries to it, and receiving result sets.

CQLs syntax is very similar to SQL both in terms of terminology and statement structure [27]. This similarity may provide a sense of familiarity and possibly reduce the learning curve, but it is important to note that the internal storage mechanisms underlying CQL and SQL are completely different. Thus, the languages operate differently internally.

4.4.2 Terminology

CQL terminology that differs from Cassandra's initial data model, is the use of the terms *table* and *primary key*, where tables represent column families and primary keys refer to row keys. Not only does a table replace a column-family in the setting of CQL, but it also has a different meaning. A table refers to a cell-based structure, while a column-family is as mentioned more of a map-like structure. Thus, this new terminology imposes an abstraction layer on top of Cassandra's data structure. The ideas of the keyspace, columns, and rows are the same as before but can look different.

4.4.3 Example

An example will be used to show how Cassandra can be used by a supermarket to store fruits in a Cassandra database. Only fruits will be used in the example, and they can have a variable number of properties. For example,

some fruits come in different colours, sizes, and variants, while others differ in their land of origin and the quantity they come in. Since these fruits are products belonging to the supermarket, they are each registered with a unique product-id.

In CQL this fruit column family is implemented through the statement in Listing 4.1. Lines 2–4 define the name and type of the columns in the table, that is, the attributes describing the table elements. In line 5 the primary key, the row key in Cassandra, is set.

```
1 CREATE TABLE fruit (
2 id int,
3 name text,
4 price double,
5 PRIMARY KEY (id) );
```

Listing 4.1: Creating a fruit table

To populate this table, rows will be inserted using the `INSERT`-statement. An example of this is included in listing 4.2 below.

```
1 INSERT INTO fruit (id, name, price) VALUES (210,'pear'
    ,22.0);
2 INSERT INTO fruit (id, name, price) VALUES (67,'apple'
    ,32.0);
```

Listing 4.2: Inserting elements into the fruit table

Two fruits have been added, apples and pears, but now fruits that come in different colours have to be added as well. Cassandra supports addition and deletion of columns. Listing 4.3 is an example of the statement for adding another colour column to our table.

```
1 ALTER TABLE fruit ADD colour text;
```

Listing 4.3: Addition of a column to the table

Now that fruit can be stored with colours, we can add such fruits to the table. Lines 1–3 of Listing 4.4 fill up the table with rows of plums, orange, and blueberry. The statement in line 5 lists the contents of the fruit-table, and its result will show how the data is actually stored. Its resulting output is included in Listing 4.5.

```
1 INSERT INTO fruit (id, name,price,colour) VALUES (542,'
    plums',45.0,'blue');
2 INSERT INTO fruit (id, name,price,colour) VALUES (32,'
    orange',35.0,'red');
3 insert into fruit (id, name) values (12,'blueberry');
4
5 SELECT * FROM fruit;
```

Listing 4.4: Inserting more elements into the fruit table

```
1  id  | colour | name       | price
2 -----+--------+------------+-------
3   67 |  null  |      apple |  null
4  210 |  null  |       pear |    22
5  542 |  blue  |      plums |    45
6   32 |   red  |     orange |    35
7   12 |  null  | blueberry  |  null
```

Listing 4.5: Resulting output from line 5 in Figure 4.4

When studying this output-table, there are two points to note. First of all, at this layer, CQL stores the data in a traditional table-structure which is different from Cassandra's map-like data model. In the latter, the storage structure would correspond to

```
FruitMap<id,FruitAttributeMap<Name,Value>>
```

in which each column is only defined with a value (and never is null). A more representative listing is shown below. This is an example that clearly illustrates the mismatch mentioned above.

```
Fruit :        {
                        "id":67, {"name":"apple"},
                        "id":210, {"name":"pear", price:22.0}
                        ...
                }
```

Secondly, as seen in Listing 4.1, Cassandra does enforce a schema as the column names and data types have to be defined upfront. However, it was also shown later that the table is easily altered to accommodate another column. Hence, Cassandra supports dynamic changes and is therefore referred

to as having a flexible schema. This flexibility is also shown at the row-level. When inserting rows, Cassandra allows rows to omit column values, as shown in the example. In other words, the structure of the content may vary over time. Another aspect to mention is that the null-values displayed are defined implicitly and do not have real values.

Tables and column families exist within keyspaces. In the supermarket example, a Products keyspace could be a viable keyspace. Listing 4.6 below shows the statements for creating a keyspace. Both the replication factor and the replication class are defined here. In this case, the replication factor is set to 1, which means there is one copy of each data row in the keyspace. Though these properties are set early, they are, like columns, modifiable through simple commands.

```
1 CREATE KEYSPACE products WITH REPLICATION = { 'class' :
    'SimpleStrategy' , 'replication_factor' : 1 };
```

Listing 4.6: CQL creating a keyspace

4.4.4 Querying

Cassandra's distributed design requires that we consider some aspects regarding querying. Firstly, data is organized into partitions and partitions are stored at nodes. Cassandra requires data to be stored in a way that facilitates short response times to queries, which means that queries have to be limited to partitions, and not span across nodes [30]. The latter is slow and therefore prohibited. This restriction has consequences for the design of Cassandra's data model and CQL query possibilities.

To optimize querying, Cassandra's data model has to fit the possible queries it needs to support. The query fitting is referred to as query-driven design [30], where the table or column family is built to answer queries, rather than representing relations or objects. Thus, the first step in Cassandra data modelling is to figure out what queries are to be supported. The second step is then to create a table that will satisfy the queries. Typically, the query -driven approach results in one table created for each query pattern. The consequence of these steps is to anticipate the needs of the application in advance. Naturally, this will possibly lead to numerous tables, potentially with duplication of data, or denormalization as it is called. Cassandra is

designed to handle the resulting redundancy, and its efficiency is built atop of this.

While CQL, for the most part, shares its syntax and statement structure with SQL, it differs notably in regards to querying possibilities. When querying, one often wants data that is related to one or more of the columns. Cassandra does not support sudden queries on fields that are not part of the primary key. This restriction does make sense in regards to the underlying map-structure. To run a query that results in the price of an apple, one either needs to make use of indexes (similar to indexes in books) or create a price_of_fruit table with the same fields with a primary key consisting of the price. The latter method is an example of denormalization.

4.5 Other features

4.5.1 Security

Client-server and inter-node communication can be encrypted using SSL (Secure Socket Layer). Thus, data transmission can be secured [31]. Authentication and authorization (through role-based access control) are other available options. Observe that none of these options are enabled in the default installation of Cassandra, they need to be configured.

4.5.2 Documentation, community and support

Apache Cassandra's official documentation is surprisingly poor considering its popularity and wide usage. The best Cassandra documentation is one provided by Datastax, a vendor that specializes in solutions built on Cassandra.

The community is strong, which makes sense since Cassandra is deployed by more than 1500 companies. Support is equally strong especially since there are several companies offering commercial support including courses and training. User groups are especially visible and helpful through technology and company blogs and in online communities.

Summary

This chapter introduced the basic concepts related to Apache Cassandra with an emphasis on its data, architecture, and query model. It started by describing how data is stored in Cassandra's version of the column-oriented structure, before it delved into Cassandra's architectural components and how they enable scalability through even data distribution. Furthermore, a description of how Cassandra applies replication to show how the database technology ensures fault detection and continuous availability without any single point of failure. Building on these notions, the CAP classification and consistency levels were discussed. Querying and CQL were presented to illustrate how Cassandra approaches queries. Through CQL it was shown how changes could be accommodated through dynamic evolution of the schema and flexibility of the data model. Further discussion highlighted the limitations Cassandra's distributed design imposes on querying to ensure quick responses. Lastly, other factors discussed were security features, available documentation and resources, and the overall availability regarding the supported platforms and language APIs.

Chapter 5

MongoDB

MongoDB is the leading NoSQL technology today [23] with more than 4300 customers and 30 million downloads [32]. It is the subject of the current chapter.

The storage system is written in C++ and is developed by the company 10gen, currently known as MongoDB, Inc., in 2007 to handle issues related to scalability [32]. The name stems from *humongous* and reflects its scalable nature [33]. MongoDB is described as a free, open-source database technology that promises a flexible data structure, broad querying capabilities, and an architecture model that ensures scalability and high availability [34].

This chapter is organized in a similar way to the previous chapter with an emphasis on introducing MongoDB's data, architecture and query model, along with some other features.

5.1 Data model

MongoDB is a document-oriented database [34], and therefore documents are used as the primary structure for storing data. Other concepts MongoDB uses for data modelling are collections and databases. These concepts are illustrated in Figure 5.1.

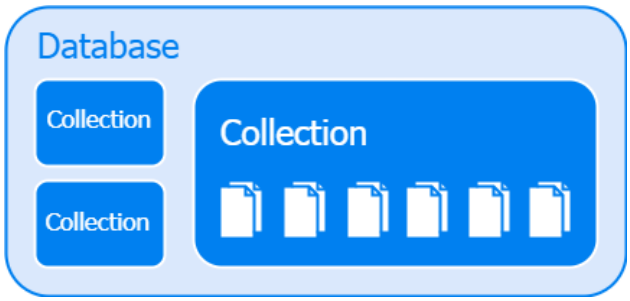


Figure 5.1: MongoDB overall data structure

- A *database*, in the context of MongoDB, is the outermost container consisting of collections. Databases are the MongoDB equivalents of keyspaces, and so encompass the entire application data.
- A *collection* is a grouping of MongoDB documents.
- A *document* is the basic structure for storing a single unit of data. MongoDB documents use the BSON, binary JSON (JavaScript Object Notation), format that is made up of sets of `field:value`-pairs. Additionally, each document is required to have a unique identifier. Basic data types, arrays, and even objects are examples of possible values that can be stored in a document. An example is included in Figure 5.2, where the data being stored is information about a video for a video-sharing site.

```
{
   _id : <ObjectIdX>,                          <---- field : value
   title : "MongoDB document model",           <---- field : value
   size :  "3.4MB",
   duation : "00:03:12",                            :
   details :                                        :
       {
          text : "Some descriptive text",
          contact : "contact@mail.com"
       },
    related : [<ObjectIdY>,<ObjectIdZ>],
    tags : ["mongodb","document","tutorial"]
}
```

Figure 5.2: Document example

There are a few points to be made about MongoDB's data model. Firstly, note the lack of schema that dictates the structure of documents within a collection. There is a general guideline to store documents of the same structure

and intent in one collection, but nothing hinders the storage of documents with different structures and contents in the same collection. The second point is that the document structure enables hierarchical data through nesting capabilities. Also, based on its full form, JSON is meant to describe objects, and therefore this model aligns well with object-oriented programming. Finally, the model requires relations to be explicitly defined, either through embedded documents (subdocuments) or through referencing other documents using their unique identifiers. Both methods are used in the example above, through the related videos and the details for each video. Each method has its own set of pros and cons, but in general, embedding works better for small documents of stable nature, while references are better for larger documents with dynamic data, and more likely to be an isolated query.

5.2 Architecture

MongoDB supports scalability by implementing sharding and replication. These techniques allow MongoDB to distribute and duplicate data as needed.

5.2.1 Sharding

A distributed MongoDB database consists of three main components; shards, query routers, and config servers [35, Ch 13, p.232]. *Shards* refer to the servers that store the subsets of data. *Query routers* are the interface between the client application and the shards. Such routers are responsible for routing the read and write requests to the appropriate shard(s). To access and query the right shard, the query uses information provided by the config servers. *Config servers* only store metadata about the shards, such as where data is located and other configuration settings. An illustration is shown in Figure 5.3.

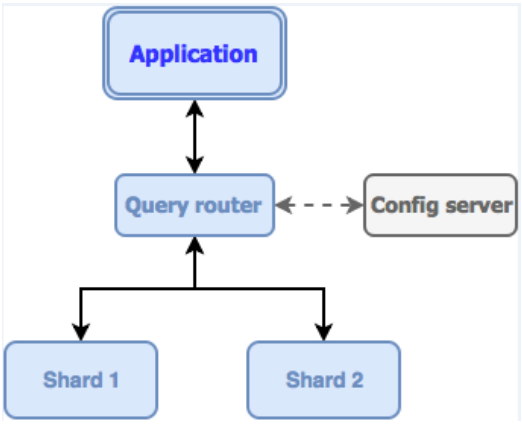


Figure 5.3: Example of a sharded cluster

The components address how data is stored and retrieved, but not how the distribution process itself is done. The distribution of data is performed at the collection level, rather than at database level, giving the user more control of the data. Thus, sharded and unsharded collections are supported within the same database. Data distribution is based on the chosen sharding key(s) and strategy [35, Ch 13]. Each document in the sharded collection has a field that corresponds to the shard key. *The shard key* is the value that decides how partitioning will happen, and the *sharding strategy* refers to the algorithm that determines which shard the document will be placed in. There are currently two strategies available: range-based and hash-based. Range-based sharding splits the data up into ranges derived by the sharding key, while hash-based sharding sends the sharding key into a hash function, and the output then determines the location of the data. Each option has its own set of advantages and disadvantages, and a choice between these must carefully balance these according to the use case. Lastly, a balancer [35, Ch 13, p. 253] is a process that runs in the background and evens out the distribution across the shards. Though, data will not be divided until the first shard has reached its maximum capacity.

5.2.2 Replication

MongoDB's replication is based on a master/slave setup with some additional options. In MongoDB, a set of different servers that are responsible for keeping the same data is called a *replica set* [35, Ch 9, p. 169] . The servers within a replica set have delegated roles; there is one *primary node* (which is

the master) and the rest are *secondary nodes* (slaves). The primary node is the recipient of all write requests, and stores all of its changes in its operations log. The secondaries use this log to replicate the changes onto their own copies of the data set to ensure consistency. The components of a replica set are depicted in Figure 5.4.

In a master/slave scheme, if the primary dies, the system can potentially have a single point of failure. MongoDB handles this problem through a communication protocol, known as the *heartbeat* and *elections* [35, pp. 191–192]. The members of a replica set ping, or send heartbeats to, each other every two seconds. If a member does not answer within ten seconds, it is considered dead. Through this communication, secondaries are informed of a primary's death, and will then hold an election to become the new primary. The members of the replica set vote and the secondary that gets the majority vote wins the election. In addition, there is a third kind of a node, known as the arbiter node. There can be no more than one arbiter in each replica set, and an arbiter's only purpose is to partake in elections in case of ties during elections. Thus, MongoDB ensures high failover and availability.

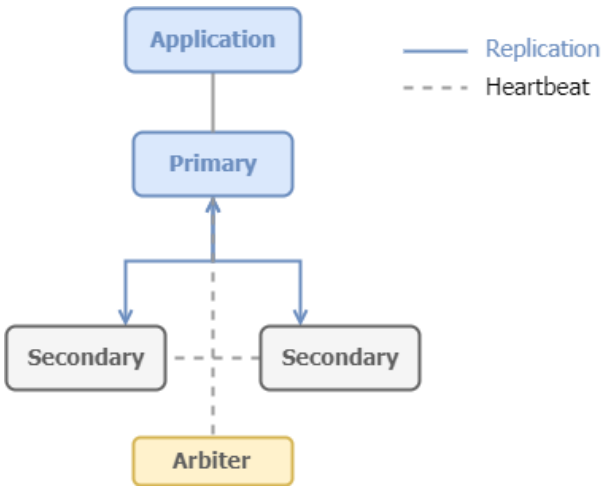


Figure 5.4: Example of a replica set

Sharding and replication have been presented separately, but it should be mentioned that they are usually used together as it is the only way to achieve high availability.

5.3 CAP classification and consistency

The information presented can be applied to the CAP theorem, which in turn can be used to understand MongoDB's capabilities better. As established above, MongoDB uses a master/slave setup for data replication. The default configuration defines that every read and write request goes to the primary node, while the secondary nodes duplicate all of the primary's data changes. Seeing that a single node handles all of the requests, the system becomes strongly consistent. However, it is possible to tune this consistency level by allowing secondaries to handle read requests, but this comes with the added risk of stale data. The system then provides eventual consistency.

If a partition occurs, that is, a hardware or network failure, there are two worst-case scenarios for the primary node. Either the primary node fails, and an election has to occur, or the secondary nodes can not connect to the primary. In both cases availability is affected; in the former, the system is unavailable to handle requests during the duration of the node failure and election. In the latter, the data itself is unavailable. Thus, availability is sacrificed in light of CAP (in spite of the availability the replica sets provide). Hence, MongoDB is referred to as a consistent and partition (CP) tolerant storage system.

5.4 Query language and model

Another way to understand MongoDB better is to study its query language concerning its syntax and general build.

5.4.1 Query language

MongoDB can be accessed through the `mongo shell` or application language drivers. The shell is included in all MongoDB distributions, and its interface is interactive and entirely JavaScript-based. The shell handles all aspects from low-level tasks such as collection creation and data insertion to higher level administrative tasks. Currently, MongoDB officially supports drivers for 11 programming languages along with several community-supported drivers [36]. MongoDB is, also, available for download on various Linux, macOS,

and Windows distributions [37].

MongoDB does not have an official query language but delivers its own JSON-structure based syntax resembling the dot notation for accessing JavaScript object properties. Line 2 in Listing 5.1 illustrates this notion. The language is for the most part used in the shell, while the drivers handle all the database interaction through the means of the appropriate language.

```
1 var doc = {name:"lemon", weight:"335g"}
2 doc.name              //returns "lemon"
3 doc.colour = "yellow"        //{"name":"lemon","weight
    ":"132g","colour":"yellow" }
```

Listing 5.1: Example illustrating JavaScript dot notation

The similarity to JavaScript objects for its document data model, and, as we will see later, to general object-oriented languages for its query model, can lower the learning curve. Another favourable point is that it is possible to run JavaScript code directly in the shell, including almost all of the language components such as loops and functions.

5.4.2 Query model

The MongoDB query model is implemented *in* the APIs of the supported programming languages, though they are very similar to the syntax mentioned in the previous section. Thus, MongoDB querying does not require the user to learn a completely different language. For the rest of this section, the shell-based notation will be used.

MongoDB maps its data model directly to its query model, so there is no excess terminology nor mappings that need to be defined. We can therefore delve straight into querying in MongoDB.

There are two important insights to understand queries in MongoDB; queries consist of certain components and revolve around specifying (and receiving) documents. The latter refers to the fact that when retrieving a document from the database, one also sends in a document consisting of some property that one specifies. In other words, the queries consist of documents. This will become more clear in the following section.

The general structure of any MongoDB query consists of the following components:

- A *collection name* so that the shell knows where the desired documents are located
- A *collection method* that specifies what method will be used for investigating. Examples include `find()`, to find documents fulfilling some criteria, and `count()`, to count the number of documents within some context.
- A *query document* that is optional and consists of the fields one wants to filter a search on.
- A *projection document* is also optional and describes the desired format of the results of the query by specifying the fields one wants to include or exclude.

Figure 5.5 illustrates this structure with a sample query that retrieves all stored fruits red colour. The figure contains the generic form of MongoDB queries and a specific instance. In the example, the projection has been omitted because it is optional. The example also shows that the query document and eventual projection document are used in conjunction with the method as the query document is passed to it as a function argument. The structure of the query document highlights that even queries are directly document oriented. Another point to note is that the query is prefixed by `db` that specifies which collection is chosen within the current database.

```
CollectionName.CollectionMethod(QueryDocument,ProjectionDocument)

db.fruit.find({'color':'red'});
```

Figure 5.5: MongoDB query structure and an example of a query

In general, queries in MongoDB can be applied to all fields in a document. Indexing for fast retrieval is also available alongside with numerous other options [38].

5.4.3 Example

To have a look at how some of the discussed concepts actually work in MongoDB, the fruit example from Chapter 4 will be used with the same set of description as then; the fruits vary in terms of price, land of origin, and other fields.

Similar to the earlier example, the storage system keeps track of the products belonging to a supermarket. Fruits belong to a collection within a products database. Listing 5.2 shows how a database and collection is created and used along with a couple of documents insertions.

```
1  use products          // creates a products database
2
3  db.fruit.insert(
4     {
5     name: "apple",
6     price: 3.45,
7     colour: ["red","yellow","green"]
8    }
9  );
10
11 db.fruit.insert(
12    {
13    name: 'watermelon',
14    origin: "Brazil"
15    }
16 );
17
18 db.fruit.insert(
19   {
20    name: 'pear',
21    price: 1.25,
22    colour:"green"
23   }
24 );
```

Listing 5.2: Creating and populating a fruit collection

There are a couple of notable points here. Line 1 creates and switches to the products database. Line 3 and onwards insert an apple and a pear document into a fruit collection that is in no way or form declared beforehand. This is the first way of illustrating the lack of schema in MongoDB and its dynamic nature. None of the fields, nor their types are defined before insertion. Fruits also have varying numbers, names, and types of fields. For example, the wa-

termelon has an origin-field while the pear's colour-field is not an array. Thus, this is the second way that confirms the flexible schema. Another thing that can be mentioned is that the shell returns `WriteResult("nInserted":1)` after each successful insertion. The output is a document consisting of a boolean field that indicates whether the operation succeeded.

We can now look at how data is stored, see listing 5.3 line 1 for the query and the output below for its results. The `find()`-method has in this instance no arguments, which corresponds to the empty document `{}`, hence all the fruits in the collection are returned. A shortened version of the returned output is included below the query.

```
1 db.fruit.find().pretty()
```

Listing 5.3: MongoDB query

```
{
        "_id" : ObjectId("59f0b31c1edbbbc61f92bf65"),
        "name" : "apple",
        "price" : 3.45,
        "colour" : [
                "red",
                "green",
                "yellow"
        ]
}
...
{
        "_id" : ObjectId("59f0b6051edbbbc61f92bf67"),
        "name" : "watermelon",
        "origin" : "Brazil"
}
```

We can now see that the documents are stored in the exact same format as they were inserted in with the exception of the id. An unique `_id` field was added by MongoDB. Lastly, the sample query from earlier on can serve as a concrete example for understanding MongoDB's query structure. Listing 5.4 contains a query for red fruit, with its response below.

```
1 db.fruit.find({"colour":"red"}, {"colour":0});
```

Listing 5.4: Query for red fruit

```
{
        "_id" : ObjectId("59f0b31c1edbbbc61f92bf65"),
        "name" : "apple",
        "price" : 3.45
}
```

The query lists the collection name and method, along with query and projection document. The query document specifies that we wish to retrieve the fruits of colour red, while the projection document specifies that the colour-attribute is to be excluded from the response. This is consistent with the resulting output. The example also shows MongoDB's ability to query all field values including nested ones, as the apple's red colour is nested as it is assigned as an element of an array.

5.5 Other aspects

5.5.1 Security

MongoDB's security has been in the spotlight several times. Most recently in September [39] and January [40] of 2017 when many tens of thousands of MongoDB databases were the target of ransomware attacks. The attackers were exploiting data that was publicly accessible, which is, in fact, *not* due to MongoDB lacking security features.

MongoDB offers an extensive set of security features spanning authentication, authorization, role-based access control, SSL encryption, and auditing. However, the default configuration does not enable any of these security measures. Common for all of the victims of the mentioned attacks, is that they were running the default configuration. In fact, all of the victims were using default MongoDB ports that were open to the public internet (which allows everyone to connect to the database), and without having authentication activated (which in turn leads to the intruder to have administrator rights on

the system) [41]. These are obvious vulnerabilities that the provided security features can protect against. In other words, the users of the database technology bear the responsibility of implementing a secure configuration.

5.5.2 Documentation, communities and support

MongoDB's documentation is extensive, consisting of up-to-date descriptions of seemingly every feature of the storage system along with examples and tutorials. The community is big with several yearly events, user groups, free online courses and blogs [42]. Additionally, various support resources are listed on the official website including multiple community-supported forums [43].

Summary

The chapter gave an overview of MongoDB as a distributed storage system. It first described MongoDB's document model by looking at its structure and how it accommodates various kinds of data. The same section also described how flexible and dynamic the document model is due to its non-existent schema. The next section proceeded to describe how MongoDB implements a scalable structure, that is, MongoDB's take on sharding and replication. Lastly, the section concluded how the technology is categorized as per the CAP theorem, heavily based on its replication strategy. The second half of the chapter focused on the practical applicability of MongoDB regarding query language and model, security settings and available resources such as documentation, communities and support. MongoDB's shell-based interface was used to define collections and documents, as well as conduct queries. The examples created illustrated MongoDB's flexible data model, how it can handle all kinds of data, and how it's query model and language work.

Chapter 6

Comparison of Cassandra and MongoDB

Chapters 4 and 5 introduced Cassandra and MongoDB as distributed storage systems along with their concepts and features. Each technology has its own set of strengths and weaknesses and therefore may be suited for various use cases. The varying data models lead to different ways to work with data and meet different application needs.

The purpose of this chapter is to conduct a side by side comparison of Cassandra and MongoDB using a set of selected criteria. The chapter starts by comparing the data and query models to explain their fit for application data, followed by a discussion about their CAP classifications and consistency levels. Next, the chapter discusses accessibility, available documentation, support and communities. Lastly, a comparison of the security features and overall ease of use is presented before a final summary of the chapter.

6.1 Data model

The data model defines how the application (or use case) data will be stored and retrieved. As the main medium for organizing data, it is important that the model fits the application data. A bad fit can at worst lead to increased development overheads such as extra logic and mappings, difficult maintenance, and the added risk of misunderstandings and inaccuracies. On

the other hand, a good fit simplifies development, maintenance and increases the efficiency by accelerating the programming process.

To ensure a good fit, it is important to scrutinise the characteristics of the data to be stored. By looking at the build and nature of the data, one can narrow down the possibilities. The build of the data refers to whether the data is structured, unstructured, or is a combination of the two (semi-structured). Another consideration is the possibility of nested structures. The nature of the data refers to whether the data is volatile, that is, is the data likely to change or will it stay the same.

In Chapter 4, it was shown that Cassandra is a column-family based data model where data is stored in rows consisting of columns, and the rows are contained within column-families. Though the terminology is similar to the one used for relational databases, the data model itself differs as it is much more capable of storing a wider range of data types. Cassandra requires a schema with the option of adding and removing columns to fit one's need. Thus, Cassandra is flexible as it can accept changes in data and accommodate rows with varying content.

MongoDB, however, stores data as JSON documents in collections. Similarly to Cassandra's data model, documents can store nested data, arrays, objects, and other data types. As shown in the previous chapter, MongoDB is entirely schemaless. There is no point at which the structure of a document is defined, and therefore each document within a collection can contain different fields and values without further ado. This facilitates smooth development when used with unstructured data, as opposed to Cassandra where changes need to be reflected in a schema before inserting the data. Another point in MongoDB's favour is that JSON is a known format for most developers. This fact along with the natural mapping from objects to documents make it an intuitive data model to work with, thus simplifying the learning curve and reducing the need for mappings compared to Cassandra.

6.2 Query model and language

Since one of the main objectives of databases is to respond to queries, it is important to consider the querying capabilities and constraints of the database alternatives one is considering. Applications have different query requirements. Some applications require no more than basic query support, while

other applications may demand queries to be supported on more fields and even combined fields.

Cassandra only allows data to be queried through the row key. The supported indexing features are additions to the row key and are of limited nature. Cassandra's data model stems from query-driven design which makes unpredictable queries difficult to support without modifications to the defined data model. This constraint is something the application must account for. Therefore, querying in Cassandra can be seen as cumbersome. MongoDB accepts queries based on any fields in a document and supports a wide range of indexes. Thus, MongoDB has a more flexible query model than Cassandra.

Any query language is closely related to a query model. Chapter 4 introduced CQL, Cassandra's query language, which essentially looks like a limited version of SQL and works atop an abstraction layer. MongoDB, on the other hand, has no query specific language but instead uses a syntax built on JSON documents that map directly to the data model. Both technologies provide a native shell-based interface to interact with the databases with one distinction being the JavaScript functionality and capabilities provided by MongoDB's shell.

6.3 CAP classification and consistency model

There are especially two characteristics that need to be considered when comparing two distributed storage systems; the technologies' CAP classifications and consistency models. The CAP classification is important because it says something about how the database system behaves when facing network instability. MongoDB is classified as consistent and partition-tolerant due to its default configuration of master-slave replication with all operations routed to the primary node. Cassandra is available and partition-tolerant due to its peer-to-peer structure where all nodes can accept requests.

In distributed databases, there is likely to be several copies of the same data. It is preferable that the data is consistent across these copies. Therefore, consistency is a desirable trait and important to discuss. In light of the description above, MongoDB is defined as consistent, while Cassandra is eventually consistent. Eventual consistency is different and requires that the application code handles possible issues that may occur like conflicts.

This means that the developer has to decide about trading availability for consistency, or vice versa, obtaining either consistency or eventual consistency. The decision should be based on the relationship between the application and the data, in addition to the trade-offs being acceptable. The use case will typically help narrowing the choice down. For example, in banking and other financial systems it is critical that the data is consistent. Social media on the other, like Facebook, prioritises responding to requests tolerating potentially stale data rather than returning no data at all. Though these notions are for the most part theoretical and highly tuneable in implementation, they are important to consider because they represent a change of mindset.

6.4 OS and programming language accessibility

Operating systems support and programming language support are two vital aspects to consider. To be able to work with a technology at all, the platform of choice has to be supported. Similarly, technologies are not used in isolation. To incorporate a technology into an application, the application's programming language must be supported. These aspects are important to consider because drivers tend to simplify the learning process as they are in a language already known to the developer, and are the interfaces between the applications and the databases.

To summarize the information presented in chapters 4 and 5, Cassandra supports various Linux and macOS distributions. Currently, MongoDB supports several Linux, macOS, and Windows platforms. Cassandra offers around 13 language drivers where the majority are community supported drivers, so developers have to take the responsibility of ensuring compatibility with the Cassandra version in use and its functionality. MongoDB officially supports 11 languages along with several community-supported drivers.

The technologies are almost equal regarding supported operating systems except for Cassandra's lack of support for Windows. On the topic of supported programming language drivers, both technologies offer drivers for many languages. However, MongoDB officially supports the most drivers, which in turn lessens the burden on the developer(s).

6.5 Security

By definition, databases contain data and the data can be of different types. In some cases, this data can be of a sensitive nature, which is attractive to criminals. In other cases, the data itself is less significant but vital to ensure continuity of the system. In both cases databases become interesting targets for hackers, emphasising that database security is necessary to consider. Vendors and developers share the responsibility to introduce safety measures — vendors, by offering the features and users by implementing the features. Thus, it is important to be aware of the available safety measures.

In Chapter 4, Cassandra's security features were introduced. Cassandra supports encryption for data in transmission along with authentication and authorization. Despite MongoDB's bad reputation regarding security, the technology offers the same set of security features as Cassandra with the extension of auditing. The technologies have in common that none of these security features are enabled in the default installation, which is therefore the responsibility of the developer(s).

6.6 Documentation, communities and support

When exploring new technology, good documentation is essential. MongoDB's documentation is seemingly impeccable with concise information and numerous examples. Cassandra's official documentation is lacking in both explanations of general concepts and CQL syntax and is therefore considered disappointing.

Similarly, the communities and available support facilitate knowledge sharing in an easily accessible way. The latter can make learning easier by introducing examples, tutorials, and interesting discussions of concepts related to the technology. Both Cassandra and MongoDB have strong communities and support, but as the most popular NoSQL technology, MongoDB visibly has the bigger community and support.

6.7 Ease of use

It is vital to consider ease of use in any technology as it can facilitate learning and implementation. Each technology's data model, query language syntax, and terminology will be used to discuss the level of difficulty or ease of use.

Both technologies are easy to use. Though, MongoDB's ease of use is superior due to several reasons. First, the document model is in JSON format, a format many developers already have used. The format is a natural model for objects, thus providing a structure that aligns well will the object-oriented paradigm. This results in a logical and intuitive data model. Secondly, the supported JavaScript in the native interface makes it easy to experiment with possible solutions at an early stage. Finally, the documentation is very comprehensive and beginner-friendly.

Cassandra's query language is similar to SQL. For those who have experience with relational databases, CQL then is quick and easy to learn. The column-family data model is fairly simple, but the similar terminology to relational databases can lead to confusion.

Summary

The chapter compared two NoSQL database technologies, MongoDB and Cassandra, by studying differences and similarities between the technologies and their implications.

The first section in this chapter compared and described the main points of each technology's data model, and related the insights to the needs of the application data. Concepts such as the native structure and volatility of data and the resulting requirements for schema flexibility were used to evaluate the data models. Both models are fairly flexible, but MongoDB comes across as the most versatile data model of the two.

The following section discussed the query models and languages, and highlighted the importance of mapping the needs of the application to a suitable query model to ensure efficient queries. In comparison to MongoDB, Cassandra seems to have a limited query model. The query languages are very different. Cassandra's CQL uses a SQL-like syntax, while MongoDB uses a

syntax based on fragments of JSON documents.

Distributed storage systems can constrain some of the guarantees related to availability and consistency, based on their architecture. The next section, therefore, compared the CAP classification and consistency models of MongoDB and Cassandra. As both availability and consistency are desirable traits, developers have to decide which one is of higher importance to the actual use case. Similarly, there is a distinction between using a database that promises consistency vs. eventual consistency, which also should be carefully considered according to the use case.

The remaining sections compared other factors that are worth considering when adopting a new database technology; each technology's accessibility, security, available documentation and other supportive resources, and overall ease of use. As leading contenders in their fields, both database technologies offer almost the same platform and programming language support with minor differences. The main distinction being that Cassandra no longer supports Windows and that most of Cassandra's drivers are community driven. One big distinction between the technologies is the officially supported documentation. Cassandra lacks in that aspect, which more or less forces one to use the documentation provided by third-parties. Nevertheless, the communities and support mechanisms for these two technologies are seemingly equal. Another similarity between MongoDB and Cassandra is that they offer almost the same set of security features. Finally, both technologies are considered fairly easy to use. However, MongoDB's ease of use stands out due to how its data and query model naturally align with object orientation.

Chapter 7

Conclusion

The aim of this thesis was to explore and compare two NoSQL databases in an effort to understand how one proceeds to choose a technology for a use case. This chapter will in the following section summarise the results of the thesis, and in the subsequent section discuss the prospects for future work.

7.1 Summary

Beginning with Chapter 1, the *introduction*, the work of this thesis was motivated by the large number of available NoSQL databases and the difficult choice of deciding which technology to use. To solve this problem, the author decided to compare two very different NoSQL databases, namely Cassandra and MongoDB.

In the following chapter, the *background and basic concepts* of distributed NoSQL databases were introduced. The chapter started by explaining some of the problems relational databases were experiencing, and how it led to NoSQL data storage technologies being developed. Afterwards,the chapter explained the main features and the various types of NoSQL databases. Lastly, a few important concepts within distributed databases were explained; sharding, a technique for distributing data across several servers, replication, a way of duplicating data across servers, and the CAP theorem, which is a classification of some of the characteristics in distributed systems.

The *methodology* chapter explained and justified the entire research proce-

dure and the chosen materials for the comparison. In short, a literature review was chosen as the research method, while MongoDB and Cassandra were chosen mainly due to their popularity. Finally, the comparison criteria were also outlined.

In Chapter 4, *Apache Cassandra* was introduced. At first, Cassandra's column family oriented data model was explained with its accompanying terminology. Afterwards, the architecture model and its consequences in the form of the technology's CAP classification were explained. Subsequently, the query model and language were presented by defining the terminology the model uses along with the syntax of CQL. Through examples it was shown how the query model and language work together. Lastly, other important features and resources were evaluated; available documentation, community and support, security and programming language support. The following chapter *MongoDB* presented information in likewise fashion, starting with its document-oriented model and ending with listing the available resources and features.

Chapter 6 conducted the *comparison of Cassandra and MongoDB* by observing the differences between the two technologies and with their practical effects. The chapter starts by comparing Cassandra's column oriented data model with MongoDB's document data model. The section explained the importance of choosing the right data model to fit one's needs, as well as which data characteristics to look at for doing so. The next section examined each technology's querying materials and capabilities, and related their characteristics to the needs of the application. Two other important aspects that affect the behaviour of the application, the CAP classification and consistency model, were then discussed. Finally, the remaining factors that were compared are the security features, language and platform availability, documentation and support, and ease of use. The significance of these factors comes from their role in the general development point of view. The central findings of the comparison are listed in Table 7.1 along with other information that may be of interest.

On a general note, it is important to fit the needs of the use case to the specific database technology. Based on this comparison and the author's own experience, however, MongoDB appears to be the stronger contender due to its highly versatile data model and query model, along with the numerous factors contributing to its high ease of use.

	Cassandra	MongoDB
Written in	Java	C++
Development model	Open-source	Open-source
Release year	2008	2009
NoSQL type	Column-oriented	Document
Data storage model	A keyspace consisting of column-families, and column-families consisting of rows.	Databases consisting of collections, and collections containing documents.
Schema	Flexible schema	No schema
Replication	Peer-to-peer	Master-slave
Sharding	Each server is a shard with its data replicated across other servers in the ring.	Each shard is a replica set. Thus, replicated data is stored on the same shard.
Consistency model	Eventual consistency	Strong consistency
CAP classification	AP	CP
Query language	CQL (similar to SQL)	JS-like syntax
Query model	Differs slightly compared to the data model (abstraction layer)	Identical to data model
Query abilities	Supports predictable queries, and are restricted to row keys.	Rich structure, supports many indexing options and dynamic queries.
Supporting resources	Official documentation is lacking, but has a large community and support.	Comprehensive documentation, community, and support.
Security features	Authentication, authorization, and encryption.	Authentication, authorization, auditing, and encryption.
Language and platform compatibility	There are currently 13 community-driven language drivers available, and Cassandra distributions only support Linux and macOS.	MongoDB officially offers 11 language drivers, and is available for all major platforms (Linux, macOS and Windows).

Table 7.1: A summary of the comparison

7.2 Further work

The scope of the thesis was limited to investigate two types of NoSQL databases according to selected parameters. The limited scope opens up for a couple of extensions to this work that would provide further insight into the various NoSQL systems and an understanding of when and how to use a particular database technology.

While this study used a document database and a column database, there are also other types of NoSQL databases, namely graph and key-value systems. Extending the comparison to include all the types of NoSQL systems is one area of future work that could be beneficial as it broadens the perspective.

Another possible area of future work is performance related experiments with the various types of NoSQL databases. Such experiments would provide information about how different types of databases behave in real applications.

A final suggestion for further work is to extend the comparisons to include NewSQL.[1]

[1] https://en.wikipedia.org/wiki/NewSQL

Bibliography

[1] P. J. Sadalage and M. Fowler, *NoSQL distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Crawfordswille, Indiana: Addison-Wesley, 2013.

[2] *Nosql databases*.
URL: http://nosql-database.org/ (visited on 26 Sep. 2017).

[3] R. Hecht and S. Jablonski, "Nosql evaluation: A use case oriented survey," in *2011 International Conference on Cloud and Service Computing*, 2011, pp. 336–341. DOI: 10.1109/CSC.2011.6138544.

[4] D. Sullivan, *NoSQL For Mere Mortals*. Ann Arbor, Michigan: Pearson Education, 2015.

[5] M. Simborg, *The Relational Conundrum*, 2017.
URL: https://www.datastax.com/2017/08/the-relational-conundrum (visited on 25 Sep. 2017).

[6] Couchbase, "Why NoSQL?" Couchbase, Tech. Rep.
URL: https://www.couchbase.com/binaries/content/assets/website/docs/whitepapers/why-nosql.pdf (visited on 25 Sep. 2017).

[7] I. Coric and D. Gaspar, *Bridging Relational and Nosql Databases*. IGI Global, 2017.

[8] M. Allen, *Relational Databases Are Not Designed For Scale*.
URL: http://www.marklogic.com/blog/relational-databases-scale/ (visited on 26 Sep. 2017).

[9] M. Fowler, *NosqlDefinition*.
URL: https://martinfowler.com/bliki/NosqlDefinition.html.

[10] B. Dayley, *Sams Teach Yourself NoSQL with MongoDB in 24 hours*. Indianapolis, Indiana: Pearson Education, 2015.

[11] B. Jose and S. Abraham, "Exploring the merits of nosql: A study based on mongodb," in *2017 International Conference on Networks Advances in Computational Technologies (NetACT)*, 2017, pp. 266–271. DOI: 10.1109/NETACT.2017.8076778.

[12] MongoDB, official website.
URL: http://www.mongodb.com.

[13] CouchDB, a document-oriented database.
URL: http://couchdb.apache.org/.

[14] Couchbase, a document-oriented database.
URL: https://www.couchbase.com/.

[15] Riak, a key-value database.
URL: http://basho.com/.

[16] Redis, official website of the key-value database.
URL: https://redis.io/.

[17] Apache Cassandra, official website.
URL: http://cassandra.apache.org/.

[18] HBase, a column-family oriented database.
URL: https://hbase.apache.org/.

[19] Neo4J, official website of the graph database.
URL: https://neo4j.com/.

[20] OrientDB, a graph database.
URL: http://orientdb.com/.

[21] Cambridge Business English Dictionary, *Research method*.
URL: http://dictionary.cambridge.org/dictionary/english/research-method (visited on 23 Sep. 2017).

[22] K. H. A. Rasch, *Litteraturgjennomgang*, 2017.
URL: https://www2.hiof.no/nor/hogskolen-i-ostfold/studiestart/oss---studie-spesifikk-informasjon/olm/organisasjon-og-ledelse-2/litteraturgjennomgang?lang=nor (visited on 23 Sep. 2017).

[23] *DB-Engines Ranking*.
URL: https://db-engines.com/en/ranking (visited on 25 Sep. 2017).

[24] E. Hewitt and J. Carpenter, *Cassandra: The Definitive Guide, Distributed Data at Web Scale*, second. O'Reilly Media, 2016.

[25] Datastax, Inc., *Internode communication*.
URL: https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archGossipAbout.html (visited on 11 Oct. 2017).

[26] Datastax, Inc., *How are consistent read and write operations handled?* URL: https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutDataConsistency.html (visited on 15 Oct. 2017).

[27] Apache Software Foundation, *The Cassandra Query Language (CQL).* URL: http://cassandra.apache.org/doc/latest/cql/index.html (visited on 15 Oct. 2017).

[28] Apache Software Foundation, *Client drivers.* URL: http://cassandra.apache.org/doc/latest/getting_started/drivers.html (visited on 11 Oct. 2017).

[29] Datastax, Inc., *Supported platforms.* URL: https://docs.datastax.com/en/landing_page/doc/landing_page/supportedPlatforms.html (visited on 15 Oct. 2017).

[30] Datastax, Inc., *Data modeling concepts.* URL: https://docs.datastax.com/en/cql/3.3/cql/ddl/dataModelingApproach.html (visited on 15 Oct. 2017).

[31] Apache Software Foundation, *Security.* URL: http://cassandra.apache.org/doc/latest/operating/security.html (visited on 17 Oct. 2017).

[32] MongoDB Inc., *Our story.* URL: https://www.mongodb.com/company (visited on 17 Oct. 2017).

[33] MongoDB Inc., *Glossary.* URL: https://docs.mongodb.com/manual/reference/glossary/ (visited on 17 Oct. 2017).

[34] MongoDB Inc., *What is MongoDB?* URL: https://www.mongodb.com/what-is-mongodb (visited on 17 Oct. 2017).

[35] K. Chodorow, *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage.* O'Reilly Media, 2013.

[36] MongoDB Inc., *MongoDB Drivers.* URL: https://docs.mongodb.com/ecosystem/drivers/ (visited on 22 Oct. 2017).

[37] MongoDB Inc., *Supported platforms.* URL: https://docs.mongodb.com/manual/installation/ (visited on 03 Nov. 2017).

[38] MongoDB Inc., *Indexes.* URL: https://docs.mongodb.com/manual/indexes/ (visited on 03 Nov. 2017).

[39] P. Muncaster, *MongoDB Customers Held to Ransom Again*, 2017.
URL: https://www.infosecurity-magazine.com/news/mongodb-installations-held-to/ (visited on 19 Oct. 2017).

[40] T. Claburn, *How to secure MongoDB because it isn't by default and thousands of DBs are being hacked*, 2017.
URL: https://www.theregister.co.uk/2017/01/11/mongodb_ransomware_followup/ (visited on 19 Oct. 2017).

[41] A. van Scheppingen, *Secure MongoDB and Protect Yourself from the Ransom Hack*, 2017.
URL: https://severalnines.com/blog/secure-mongodb-and-protect-yourself-ransom-hack (visited on 20 Oct. 2017).

[42] MongoDB Inc.
URL: https://www.mongodb.com/community (visited on 20 Oct. 2017).

[43] MongoDB Inc., *Technical Support*.
URL: https://docs.mongodb.com/v3.4/support/ (visited on 20 Oct. 2017).