AVAILABILITY AND RELIABILITY STUDY OF NOSQL DATA STORES ON COMMODITY HARDWARE

by

WALDON HENDRICKS

Thesis submitted in fulfilment of the requirements for the degree Master of Technology: Information Technology

in the **Faculty of Informatics and Design**

at the **Cape Peninsula University of Technology**

Supervisor: Dr. Boniface Kabaso

Cape Town
August 2019

DECLARATION

I, Waldon Hendricks, declare that the contents of this thesis represent my own unaided work, and that the thesis has not previously been submitted for academic examination towards any qualification. Furthermore, it represents my own opinions and not necessarily those of the Cape Peninsula University of Technology.

11.11.2019

Signed **Date**

ABSTRACT

Modern application development and the delivery of these applications have changed drastically during the last few years. Applications are deployed on every mobile device to cloud devices hosted on servers and because of this change, users expect much faster response times from servers. To determine which data store, to store information and which data structure to choose, for a high availability and scalable architecture, is still a challenge for developers. Modern applications need to follow the reactive manifesto approach to be more responsive, elastic, resilient and message-driven to be classified as a failure-tolerant system. Four NoSQL categories were chosen to be studied using a common programming language driver. Our research strategy conducted an experiment and this work followed an experimental design approach to send objects using Create, Read Update and Delete (CRUD) operations to measure the read metrics and write metrics per data store. Our research results showed which NoSQL database can be used as read model and which database as write model for a Command Query Responsibility Segregation (CQRS) application, using the reactive manifesto approach.

Key words: developers, data store, scale, NoSQL, JAVA, CQRS, Reactive, CRUD

ACKNOWLEDGEMENTS

I want to thank:

My Heavenly Father for His mercy and grace, this enabled me to complete this study.

My supervisor, Dr Boniface Kabaso for inspiring me to take this kind of research approach.

My reviewers Martin Madioma, Amlan Mukherjee and Janvier Kamanzi for providing the support I needed to formulate this. My editors, Nicole Wessels and Naseema Allie for the editing done. I would like to thank my wife Stephia Hendricks, family and friends for their support while doing this thesis.

DEDICATION

I would like to dedicate this work to my mother that's looking over me every day from heaven. My dad that's always there when I need to tell him something. My grandma, my pillar of strength and my wife for encouraging me to complete the thesis.

GLOSSARY

Computing Cluster	A group of shared individual computers, linked by high-speed communications in a local area network topology using technology such as gigabit network switches and incorporating system software, which provides an integrated parallel processing environment for applications with the capability to divide processing among the nodes in the cluster.
COTS	Commodity off the shelf. Used to describe commodity hardware (personal computers, disks, network) that can be purchased from multiple sources.
Data-Intensive Computing	Used to describe computing applications that are I/O bound or with a need to process large volumes of data. Such applications devote most of their processing time to I/O and movement of data.
A Distributed System	is an application that executes a collection of protocols to coordinate the actions of multiple processes on a network, such that all components cooperate to perform a single or small set of related tasks.
Intrinsic	in computer software, in compiler theory, an intrinsic function (or built-in function) is a function (subroutine) available for use in a given programming language where implementation is handled specially by the compiler.
CQRS	stands for Command Query Responsibility Segregation. It's a pattern that I first heard described by Greg Young. At its heart is the notion that you can use a different model to update information than the model you use to read information**.**
NoSQL	stands for "not only SQL," it is an alternative to traditional relational databases in which data is placed in tables and data schema is carefully designed before the database is built. NoSQL databases are especially useful for working with large sets of distributed data.
CAP Theorem	is a concept that a distributed database system can only have 2 of the 3: Consistency, Availability and Partition Tolerance. CAP Theorem is very important in the Big Data world, especially when we need to make tradeoffs between the three, based on our unique case use.
CALM Theorem	CALM is an acronym for "consistency as logical monotonicity." The CALM Theorem shows that the programs that have consistent, coordination-free distributed implementations are exactly the programs that can be expressed in monotonic logic.
DC/OS	Datacenter Operating System (DC/OS) is an open source operating system based on the Apache Mesos distributed

systems kernel. Developed by Mesosphere, DC/OS is available as both, an open source and a commercial offering.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE
INTRODUCTION TO THE RESEARCH STUDY

1.1 Introduction

Everyday huge amounts of data are generated by researchers and developers, which may be unstructured data, semi-structured and structured (Haseeb & Pattun, 2017). More NoSQL data stores are constantly added to the ecosystem and this brings new challenges to determine which NoSQL database to implement and which data structure and programming language to use with cloud applications. Data generated by researchers exceed their ability to design an appropriate cloud application for data analysis and generating workloads (Haseeb & Pattun, 2017).

Two Internet cloud companies developed their own distributed non-relational systems to help with the scaling of data (Chang et al., 2008). These systems were written from scratch so they created their own unique query language. Relational databases could not scale all the large amounts of data, leading to the rise of NoSQL (DeCandia et al., 2007). Developers had to learn more new languages and connect databases to applications, so most companies had to develop their own visualization tools to interact with the NoSQL databases. There are over 150 NoSQL databases in use and the number continues to increase by the second (Feuerlicht, 2010).

NoSQL data stores provide high concurrent read-writes, database scalability and high availability (Peng Xiang et al., 2010). NoSQL is an alternative, but not to replace relational databases (Abramova et al., 2014). Thus, the NoSQL and relational databases complement each other in database activities and management of data sets (Mackin et al., 2016).

Running web applications and accessing these applications, researchers found that data stores need to provide a more scalable data storage to handle the ever increasing capacity of data storage (Peng Xiang et al., 2010). This led towards a strong interest in NoSQL data stores for researchers. In terms of scalability, speed, cost, non-relational data stores have a stronger advantage of relational data stores (Peng Xiang et al., 2010).

Software engineers and developers should design applications that can operate in the cloud and not just to deploy to the cloud (Grozev & Buyya, 2014). Researchers found that when using multiple cloud environments clients don't need to rely on any interoperability functionalities that's implemented by a provider, this allows the application to be deployed

across different cloud environments (Grozev & Buyya, 2014). Case studies done by IBM and e-Bay demonstrated how three-tier applications utilized multiple data centers to provide better availability and customer Quality of Experience (QoE) and adapt to changes needed (Grozev & Buyya, 2014).

The change to convert a standard application to cloud application is not easy at all. The cloud environment is not the same as a development environment and makes it difficult to just transfer an application to a cloud environment (Grozev & Buyya, 2014). Software applications should be more scalable in design and fault tolerant to dynamically adapt to different workloads and respond in a timely manner.

The most common pattern used in software architecture is the layers' pattern or layered style. The approach to this design was to organize a large-scale logical structure of a system into discrete layers with related responsibilities (Pruijt et al., 2013). The pattern allowed a clean, cohesive separation of concerns, where lower layers are classified as low-level and general services and the higher layers are more application specific (Pruijt et al., 2013). Figure 1.2, represents a strict layered design, the usage relationships are from top to bottom or outside towards the inside layers, described in detail on page 5.

Layered designs are often poorly defined and many violate the key principles for which the layers were designed for. Student projects encounters many layered designs showing only the names of the layers without any specification of the contents and communication rules (Pruijt et al., 2013). Projects designed like this provides no guidance to the developers, therefore a specification of responsibilities of the layers are needed (Pruijt et al., 2013).

Software developers developing software applications needs to understand that architecture design is always necessary and that the well-designed architecture diagrams do not describe the real architecture of an application. Developers should be good at designing their own architecture when they write code. If they don't they will end up with more than one architecture (Kainulainen, 2014). Many developers follow the designs of software architects and believe that software architects are always right and this led developers to follow architecture designs of software architects (Kainulainen, 2014). According to Kainulainen (2014), developers should follow two principles to create their own architecture design.

Principle one: The Separation of concerns (SOC).
This is a design principle for separating an application into distinct sections, which means each section addresses a separate concern. This principle helps developers identify the required

layers and the responsibilities of each layer (Kainulainen, 2014).

Principle two: The keep it simple stupid (KISS) principle.
Systems should be kept simple rather than complicated, simplicity should be kept as a key goal design in developing applications. According to Kainulainen (2014), adding new features takes longer because information has to travel through every layer. Maintaining the application could be impossible if the developers don't understand the architecture.

Developers can adopt these concerns above by using only three layers (Kainulainen, 2014). As shown in Figure 1.1 the web layer of the web application will receive the user's input and return the response back to the user. The web layer should also handle the exceptions thrown by the other layers. This is also the entry point of an application and should take care of authentication and prevent unauthorized users (Kainulainen, 2014).

The service layer resides below the web or presentation layer and contains both the application and the infrastructure services. The application services provide the public Application Programming Interface (API) of the service layer. The infrastructure services contains the code that communicates with external resources such as the file systems, databases or email servers (Kainulainen, 2014).

The repository layer is the lowest layer of the application; this layer is responsible for communicating with the data storage.

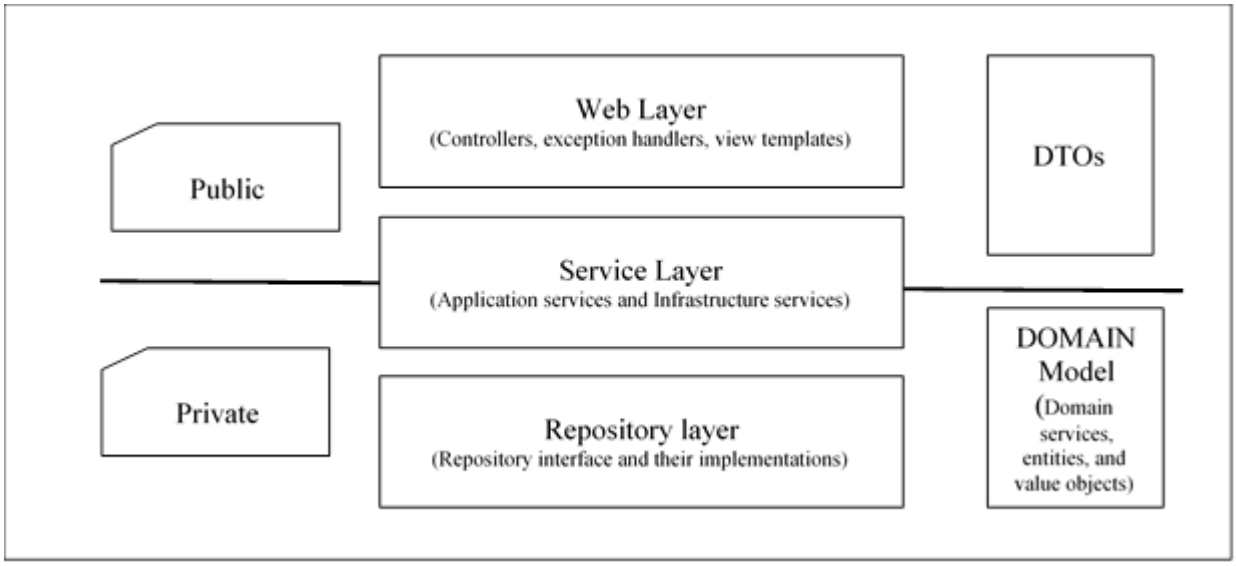


Figure 1.1: The architecture of a classic spring web application

According to Gierke (2013), who studied the importance of architectures in programming code

bases and focuses on Java packages, when splitting up a complex problem into smaller ones, one can approach the smaller ones individually. This is known as a core principle “divide and conquer” and this was done in Java software programming when deploying deployment units (WARs, JARs), packages and classes.

Developers should understand layering is a technical aspect that decomposes the software. A question came to mind that if developers understood the value and benefits of layering software and slicing the code horizontally, developers won’t neglect this approach when they have to vertically decompose business programming functionalities (Gierke, 2013).

Programming language software layers are well understood by developers, but less important to business processes, while slices are new to developers and a key business process requirement. Deployment units are ones managed by either Gradle or Maven, which are a dependency management and build automation tools (Gierke, 2013).

Developers generally skip packages as a means to control the visibility of types, they keep the information in the classes or properties hidden, but not on the classes in a package level. This means if the class is not public they don’t have to manage the dependency on the global level, but within the package only (Gierke, 2013). Packages are created when developers use architecture tools like Sonargraph to help them, by moving the vertical slices into the focus of the package naming and to model the slices in a way that the public API of the slice is as small as possible in the first section or place. Packages can help developers achieve visibility of control when they write their code (Gierke, 2013).

There have been a whole range of ideas of architecture of systems, which included the hexagonal architecture adopted by Steve Freeman and Nat Pryce, the onion architecture by Jeffery Palermo and the screaming architectures. They are very similar in their details as they all had the same objective to separate the concerns and dividing software into layers see Figure 1.2 (Martin, 2012).

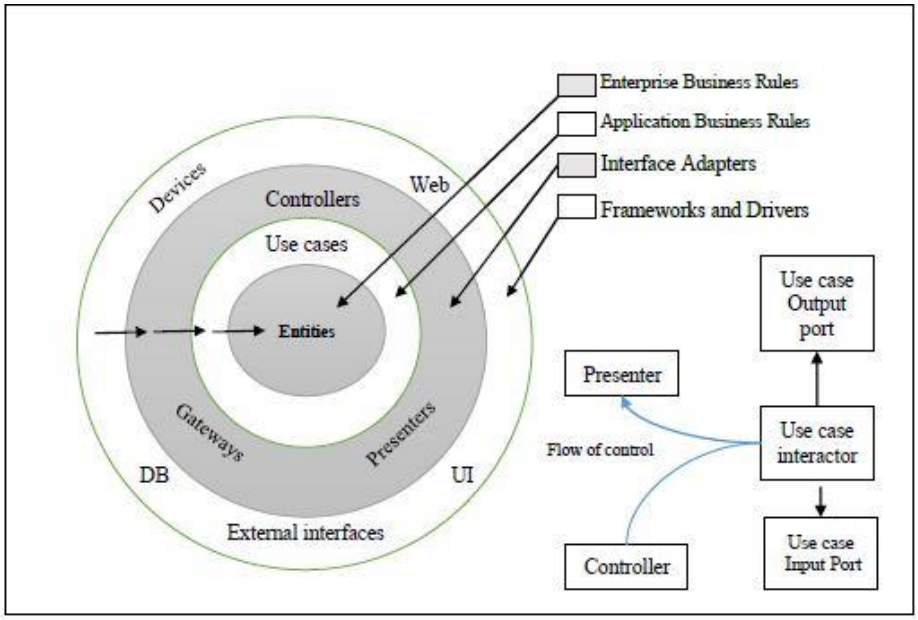


Figure 1.2: The clean architecture

(Martin, 2012)

As shown in *Figure 1.2* systems were produced using ideas as above. They were independent of frameworks and did not depend on the existence of libraries or overloaded software and this allowed the use of frameworks as tools. The business rules could also be tested without the user interface, database and web servers. As being independent of the user interface, the user interface could change easily without changing the rest of the program or affecting the business rules. Developers could swap out the database like structured query language (SQL) servers to another database, like not only Structured (NoSQL) databases as the system was independent of a database (Martin, 2012).

According to Martin (2012), the circles in *Figure 1.2* show all the different areas of software. The outer circles are mechanisms and the inner circles are policies. To make this all work, developers should use the dependency rule that all source code dependencies can only point inwards. The inner circles shouldn't know what's happening in the outer circles. If declaring a name in the outer circle, this shouldn't be mentioned in the inner circle and vice versa, so when all the external parts of the program change or become obsolete, like the database or the web frameworks, it will be easy to replace those external parts (Martin, 2012).

Layered architecture designs used in practice should be designed to meet the specific requirements of a system, as the required layers and responsibilities of each layer may vary amongst different software applications (Pruijt et al., 2013). More literature and case studies are needed on the responsibilities of other types of software applications and other software

architectures.

1.2 Background

Modern application development and the delivery of these applications have changed significantly in the last years. These shifts of applications development generated principles when building, designing and delivering of applications to the end users (Stetson, 2018).

Currently there's two types of three-tier applications in terms of the domain layer design, as mentioned the domain layers are responsible to allow the client to access the data layer where the data gets stored and retrieved (Grozev & Buyya, 2014).

The two types of domain layer designs are stateful and stateless applications. Stateful applications keep the session data in memory to ensure all requests of the session are routed to the same application service. Stateless applications do not keep any data in memory, but data are routed across different application services (Grozev & Buyya, 2014).

Supported by middleware, micro services are for communication and used on low-cost deployments, these are small applications that can be deployed independently and easy to integrate (Esposito et al., 2016). The small applications are designed to achieve simple responsibility tasks.

The awareness of micro services has increased. Regardless of the efforts required to implement micro services (Esposito et al., 2016) container technologies are used to overcome virtualization limitations and this encourages the use of micro services for software deployment and software development (Esposito et al., 2016).

The first principle as shown in *Figure 1.3*, applications should be kept small to lower the cognitive load that developers have to maintain, to focus on solving problems and not to make a complex model of an application and designs. By using micro services developers would also reduce the cognitive load, as each service would be focused on each one's functionality and would communicate using API calls via Representational State Transfer (REST) (Esposito et al., 2016).

The second principle is that the application should be developer-oriented. The developer's environment should be easy to work with and the code should be easy to understand. The architecture and code should have RESTful APIs and these should have endpoints expressed as nouns and Create, Read, Update and Delete (CRUD) operations (Stetson, 2018).

The third principle is that the application should be networked, as applications run on local systems that they are hosted on. As applications become larger, the developments and delivery have become more distributed, this leads to the increase of speed and makes networks more reliable and applications have become more networked. By networking your application, it makes your application architecture more resilient and deployment easier. But since moving from local deployment to network applications have persisted, things slowed down. Applications are getting more networked because the network structure makes the application more resilient and deployment and management easier (Stetson, 2018).

Networking your application provides many benefits over monolithic applications and provides high availability because of the design. Network applications are easier to manage and easier to monitor, when scaling your application to handle more traffic you simply just have to scale an individual service rather than the entire application (Esposito et al., 2016).

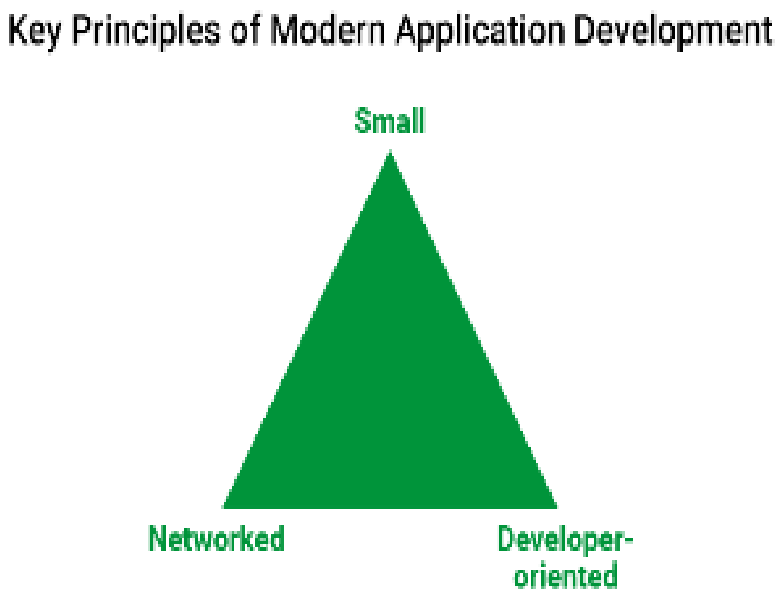


Figure 1.3: Key principles of modern applications development

(Adapted from Stetson, 2018)

If developers implement the principles above, they would take advantage of the modern trends in software development and the delivery of the these applications by using containers like Docker and implementing container orchestration frameworks like kubernetes as well as micro services architectures for applications (Stetson, 2018).

Modern applications support multiple clients, like running on an Android or IOS app for a client, connecting an application through an API or when the client is a UI using the React JavaScript

Library. Modern applications allow users to access the data and services through an API which should be constant as different clients will connect to it through a GUI or CLI interface of HTTP(S).

Modern applications need to follow the goals of the Reactive Manifesto as illustrated in Figure 1.4 to be classified as message-driven, elastic, resilient and responsive (Debski et al., 2018). To make a modern application that is scalable and highly available, applications should be designed with scalability as their first objective, to allow for high volumes of client requests and to easily adjust the resources needed (Debski et al., 2018).

To achieve the Reactive Manifesto approach, developers should use the Command Query Responsibility Segregation (CQRS) design pattern and domain-driven design (Debski et al., 2018). This approach will allow applications to be scalable and more responsive.

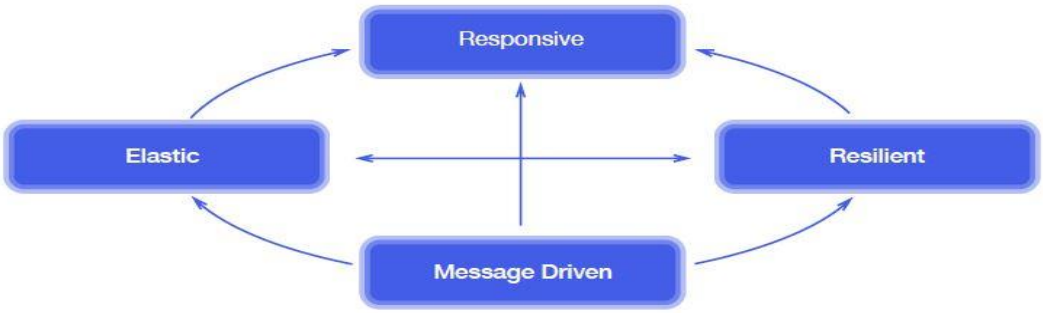


Figure 1.4: The Reactive Manifesto 2.0

(Adopted from Bonér et al., 2014)

So the need to build scalable, responsive systems would be to include a message driven approach. To make sure the system has elasticity and be resilient. This approach would make the system more flexible and scalable as well as to be a failure-tolerant system (Debski et al., 2018).

According to Debski et al., (2018:62) The CQRS principle as shown in Figure 1.5 advised that if developers separate the operations that mutate state (commands) from queries, this would create possibilities to allow developers to choose different databases for write operations and read operations. Developers will be able to select the best-performing alternative database for queries and they would be able to optimize each query separately.

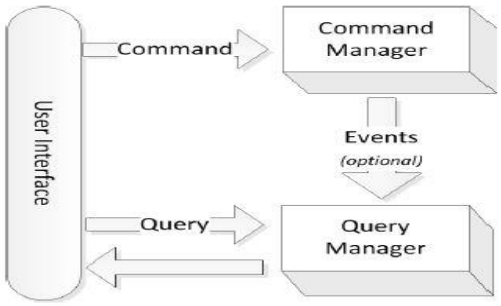


Figure 1.5: CQRS core principle

(Adapted from Debski et al., 2018)

1.3 Research problem

To achieve high availability a system should have a built-in disaster recovery mechanism. RDBMS need to keep multiple replicas consistently seamless on a grid of systems mapped across regions. This would make all resources available during normal operations and if failures occur, this would only affect fewer resources, but this approach makes a machine sit and idle waiting to take over if the primary system fails. The use of vertical fragmentation on relational databases makes the tables split subsequently across multiple workstations (Lourenço, Cabral, et al., 2015). According to Strauch (2014:21), consistency is one of the critical factors and scaling horizontally is a challenging task.

Changes are happening because application requirements have changed in the last few years. Large applications use gigabytes of data, hours of offline maintenance are needed and only one single storage of data. Currently applications are deployed on everything, including mobile devices to cloud-based servers hosted on clusters that run thousands of multi-core processors. Because of these changes users expect a much faster response time and 100% high availability applications.(Bonér et al., 2014).

According to Debski et al., (2018:63) the existing client-server application stacks can't take full advantage of the scalable needs of today's cloud environments. According to Stonebraker et al., (2018:1150), disk volumes have increased, making it impossible to keep everything on just one server, as many technologies are different today than what it was years ago. Within two decades a number of database systems evolved, which includes text management, stream processing and data warehousing. These systems have different requirements than the business data processing used for Structured Query Language (SQL).

As mentioned in the introduction, we discussed the three-tier architecture of an application

where we separate the data, logic and presentation layers. This approach allowed the data store to be seen as one CRUD database which allowed all queries and commands to be performed on one database (Kabbedijk et al., 2014). As the data stores increase and more commands and queries are sent to the data store, many experience performance problems, scalability problems and locking of input and output queries, which will lead to a high probability of data inconsistency (Kabbedijk et al., 2014).

Data layers of applications can cause performance bottlenecks due to the requirements for transactional access and atomicity (Grozev & Buyya, 2014). According to the Consistency, Availability and Partition (CAP) theorem it's impossible for a distributed system to have consistency, availability and partition tolerance all at the same time, this would make it hard to scale horizontally, within a distributed architecture there should be a balance between persistent storage consistency, availability and partition tolerance (Grozev & Buyya, 2014).

These strategies are application specific, and it's impossible to implement within a general framework containing all the three-tier applications, the balance regarding the CAP theorem requirements is domain inherent (Grozev & Buyya, 2014). Some applications may require data not replicated across different nodes, and others may allow this to achieve availability.

Application engineers should design the data layer in a scalable way to allow all domain layers to access and retrieve data without any time constraints. Database design is the first step in a three-tier system design to serve to other applications.

In a multi-tier application each layer can be the cause of slow performance and the possible solution would be to adopt separate controllers for each layer and tier and use the coordination methods such as message passing techniques as explained in Figure 1.4. So the option to split the parts of the distributed system data layer into multiple, different environments would be to adopt the CQRS pattern for data storage and retrieval (Kabbedijk et al., 2014).

The need to choose the right highly available NoSQL data store for an appropriate system are one of the many challenges for software developers and the research community (Cooper et al., 2010). The NoSQL data models can be documented and compared qualitatively, however comparing the performance of different systems is a harder problem (Cooper et al., 2010).

Understanding the performance of NoSQL data stores and the implications of the type of application needs is a challenging task, developers of these various NoSQL systems report positive performance figures to support the capacity of workloads generated by their systems,

which might not match the workload of a target application (Cooper et al., 2010).

Software development companies make use of NoSQL data stores that need high specifications and should have the ability to handle data in very fast modes using no fixed schemas and unstructured data (Petreley, 2006). Based on the Meta data to achieve a high performance rate, this allowed NoSQL to become more common to use (Sareen et al., 2017).

Many software companies and organisations make use of workstations that use resources like computational power, memory and hard drives for storage. Most times the resources go to waste, as they do not use the workstation's full capacity and electricity is wasted. Finding empirical data on the availability and reliability of the existing data stores and the new ones being created, is adding to the complexity of the choices to be made in the selection process of creating storage clusters on commodity hardware (Adya et al., 2002).

Normal operations would only have half the resources available causing degrade in performance. This approach can be redesigned by implementing peer-to-peer High availability (HA) within RDBMS (Stonebraker et al., 2018).

In this thesis, the study looked at the open source NoSQL data stores to be used across multiple workstations called commodity hardware as a data center cluster.

The research problem addressed, allowed developers to choose the right NoSQL database programming language driver on a particular data structure used during the development of a program or service. The problem used during this research work can be identified as a simple problem. The problem used a recipe which was essential for the research study to test the different variables and provide the results.
It can be divided into:

- NoSQL data store categories as input
- Programing Java Language drivers as inputs
- Data objects used as inputs
- Commodity hardware used as inputs

The first part of the problem referred to the four different NoSQL data store categories. This identified the different data storage methods used per category. The challenge with this part was to identify the different hardware requirements that this can operate on. The four categories determine how data gets stored and retrieved.

Key-Value Stores: Key-Value based are closely related to document stores as they store values against a key and there is no need for schemas to be associated with the values. The programing language used for some NoSQL data stores are written in C (Vaish, 2013).

Wide Column Stores (Extensible Record Stores): Column-oriented databases will store its data in columns instead of rows like in a RDBMS. The implementing programming languages used are JAVA, Python and Go (Vaish, 2013).

Graph Databases: This is a special category of NoSQL databases that characterizes relationships as graphs. This may include social relationships amongst people and many other network topologies. The implementing programming language is JAVA code (Vaish, 2013).

Document Stores: Allows the inserting, retrieving and manipulating of semi-structured data, most databases will use XML, JSON, BSON or YAML where the data access will be over the HTTP protocol by using RESTful APIs, this provides flexibility. Implementing programming languages used are JAVA, Erlang, C++ and C (Vaish, 2013).

,

The second part of this problem looked at the programming language driver used when sending objects for storage and retrieving the objects for validation. The challenge was to find the best driver based on the popularity of language drivers and the use of three-tier applications with NoSQL data stores, through checking the driver usage and ranking to determine the drivers commonly used when implementing Create, Read, Update and Delete (CRUD) operations. The programming language drivers used for this work was the JAVA programming language based on the driver packages available for NoSQL.

The third part of the problem used the second part of the problem to determine how the third part could be implemented for this research work. The study maintained using the same objects when establishing client connections to the NoSQL databases.

This problem allowed the use of commodity hardware nodes that scaled data across the cluster. This hardware was inexpensive workstations and selected to measure the performance, scalability, availability and reliability of NoSQL data stores. The study selected how many instances of the NoSQL data stores can be scaled across the commodity hardware data center.

1.4 Motivation

Software Developers are always looking for scalable, cheap and reliable data stores to distribute data across distributed systems. This research looked at the highly available and reliable NoSQL database per data type category, based on the read and write metrics and to also determine how each NoSQL database performs on inexpensive commodity hardware. As developers use different programming languages with different data types, by undertaking this research they can select which NoSQL database of their chosen programming language are most reliable and highly available. The research aimed to determine and select which NoSQL database can be used as read model and write model for a CQRS application using the Reactive Manifesto approach.

This research problem was to implement and analyse a system to determine which databases are more responsive, reliable, scalable and elastic. The study used experiments to generate and analysed the data to get results.

1.5 Research questions and sub-questions

For the purpose of the study, the question of concern would be, how to identify the best NoSQL data store to use in order to guarantee high availability and reliability of deployed applications. This should look at the current set of NoSQL technologies that exists and how to make developers use and implement these technologies for their applications.

This gave rise to the following research question:
What are the best NoSQL data stores that currently exist, in order to guarantee high availability and reliability of deployed applications?

Research sub-questions

1 What are the different hardware requirements for NoSQL data stores to operate on, to achieve high availability and reliability?
2 What is the best architecture for high availability and reliability?
3 What are the best Java drivers used to persist data in the four types of NoSQL DBs?
4 What is the most common pattern for persisting data objects used by Java developers?

1.6 Research methodology

An experimental method was adopted by using an experimental design methodology. To address the research questions, the study used a conceptual framework model as illustrated in Figure 3.4.

The study wanted to answer the first research sub-question by conducting a systematic literature review to identify which solutions currently exist. This allowed the study to deal with sub-objectives of the study.

To answer sub-objective one, articles identified that researchers have a challenge to determine which data model they should use for applications. Modern applications are required to be highly available to handle big data (Yassien & Desouky, 2016). This allowed the review to determine which architectures can be classified as a highly available architecture. The second sub-objective explored if any of the articles used language drivers that developers use to connect to NoSQL data stores, the focus led to choosing only one language driver per NoSQL category to be explored and studied. The third sub-objective led to the setup of the experiment to find the most common objects used when using Java language drivers and focus on the four NoSQL categories. The third sub-objective used the conceptual framework in Figure 3.4 to implement and study the variables used in the study.

The purpose was to find out what empirical evidence exists by following the guidelines described by Kitchenham et al., (2010). The systematic review helped trim down the work done in this thesis in order to answer the other sub-questions. The output of the systematic review was based on the results of the quality assessment. The quality assessment was used to select the nine articles to the review the literature. The results of the quality assessment articles identified which article performed benchmarks, to select if any client tests were done during the study and if the study used programming language drivers. This allowed the final selection of literature to be analysed and compiled into a table (Appendix A) where experiments performed were classified.

This study's conceptual framework in Figure 3.4 explained how and in what sense these variables have been used in this study. Because the results and outputs would be similar, this would be seen as repeatable research. The study wanted to understand how the variables relate to one another. The study wanted to investigate the effect of changing conditions on the variables by increasing the values and decreasing the values and to establish whether certain conditions produce better results when changing them.

The study chose the quantitative research method approach to study the variables known as NoSQL databases, programming language and data types, to determine the relationships between the variables and then to manipulate them. Based on the theory gathered during the systematic literature review, the study developed a conceptual framework, where the study

added concept and variables to explore and test the relationships between the variables.

The study followed a deductive approach, and was concerned with the generation of new theory emerging from the data collected and made observations from the theories to formulate the results towards the end of the research.

Figure 1.6 shows the outline of our research process that the study followed to gather results and findings. This research process allowed the study to be mapped to ensure that researchers and academia accept the results. The study used the Oates's research model for the purpose of the study and defines the research process as seen in Figure 1.6. The research questions were formulated from the previous relevant research literature and also areas of interest to answer the research questions. The conceptual framework allowed the study to describe the manner in which the researcher's thoughts is structured around the research process (Terblanche et al., 2013). The strategy to conduct the research was selected as an experimental strategy to focus on the cause and effects of an occurrence, in order to prove or disprove the current research questions.

As every academic project required data, this study process used the indirect observation technique to determine the relationships and the effects of the variables, indirect observation was applied to record and observe the data to be analysed at a later stage. The study process evaluated the data obtained and analysed the results using quantitative analysis to present defensible findings.

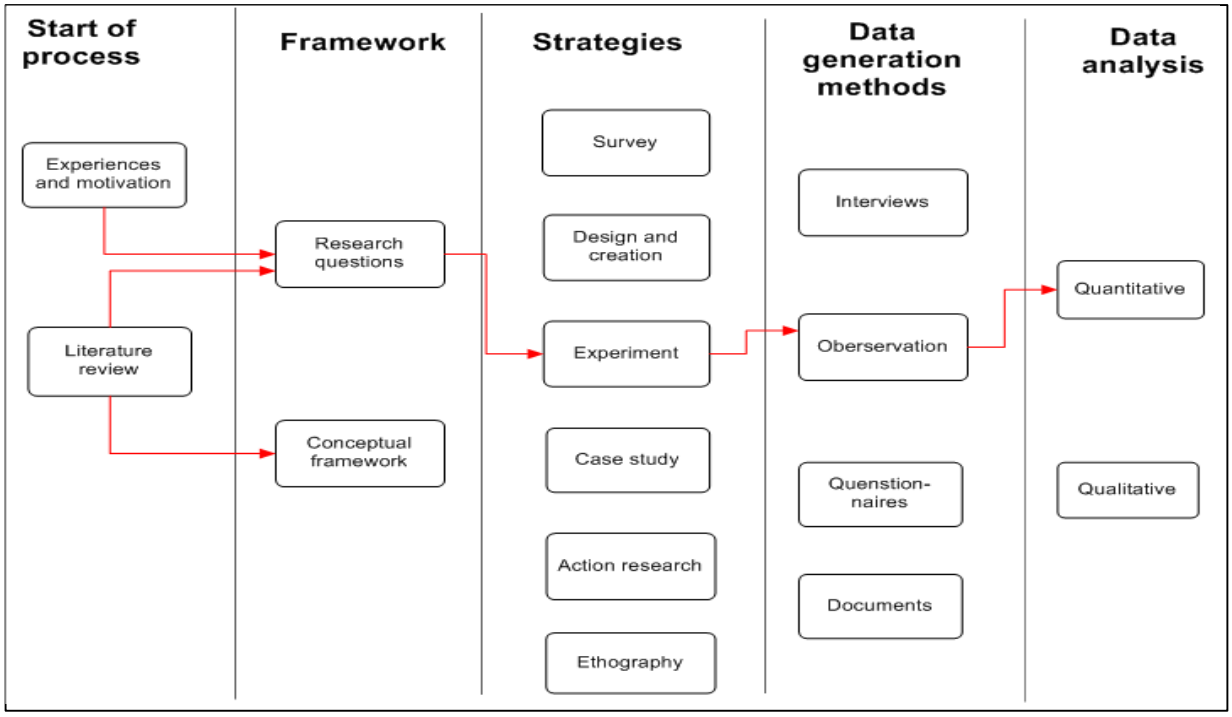


Figure 1.6: The outline of the research process

1.7 Implications of the results of the study

The thesis presented:

- The study of NoSQL databases to determine the best scalable, elastic and reliable data store to select and to implement.
- The highest available open source NoSQL data store was determined by this research
- Which database to use as read model based for a CQRS approach design?
- Which database to use as write model based for a CQRS approach design?

1.8 Thesis outline

The dissertation comprises five chapters. In the first chapter, the background to the research and objectives are described, and the aim of the research is defined.
In the next chapter, the study will focus deeper on the research problem with the use of a literature review.

Chapter 2: Literature review
In this chapter, the study will describe the most relevant discussions surrounding the research study and bring the most important aspects to light and point out how the current work directly affects the study. This research will look at NoSQL distributed systems running on commodity

servers, using distributed consensus on each data store. The study will also look at the programming language Java and reactive systems.

Chapter 3: Research methodology

This section, which is a recipe for the experiment, describes the plan or protocol that is used to perform the experiment and analyze the results. It should provide all information that is necessary to replicate the study and integrate it into the body of knowledge. Further, this section allows readers to evaluate the internal validity of the study, which is an important selection criterion for systematic review.

Chapter 4: Research findings and discussions

In this section the study sets up the experiment, the requirements and the installation of the distributed system cluster on the commodity hardware. The study explains step by step and provides instructions on how to reproduce this experiment for further research and findings. This section will also show the results produced from the experiments, and how the study analysed the results of the experiment. The study will look at the CRUD operations of objects created per NoSQL databases on the commodity hardware. The study will look at the results when scaling the data stores on the commodity hardware to achieve high availability. From the results the study can determine which data store to select for a read model and which data store to select for the write model. This section would show our discussions applying the CQRS principle to see which selected data store would be used from the findings as a read model for output data and which one data store could be used as a write model for accepting input from users.

Chapter 5: Conclusions and recommendations

This section would include a description of what the study could not achieve during the study period and the limitations. The conclusions of the study would highlight what the study achieved. It will also identify what was left out and recommendations for future research in the field. This would also include the limitations of methodology and research limitations and factors surrounding it. The study would then include recommendations on how to continue the research project

CHAPTER TWO
LITERATURE REVIEW

2.1 Introduction

Based on the Beckman report on database research, a group of researchers found that Big Data aroused due to three major trends (Abadi et al., 2016). Big Data became very cheap to generate large amounts of data because of inexpensive storage. Large amounts of data became cheap due to multicore processors, solid-state storage cloud-computing and open source software. Managing data has evolved to not just database professionals but now application users, journalists, researchers, scientists and even everyday clients manage data. Huge amounts of data will be stored and this captured data gets queried and finally processed to be turned into knowledge (Abadi et al., 2016).

According to Abadi et al. (2013:93), systems created to handle Big Data didn't follow the Database Management System (DBMS) guidelines. A transaction defined as a sequence of operations performed as a single logical unit of work and a logical unit of work must exhibit four properties called the atomicity, consistency, isolation, and durability (ACID) properties, to qualify as a transaction.

When completed, a transaction must leave all data in a consistent state. Big Data systems focused more on scalability and fault tolerance on commodity hardware (Abadi et al., 2016). With these new improvements to handle big data, the design and implementation can bring massive challenges about the volumes, velocity and variety, and this requires drastic reconsidering of the current system design (Abadi et al., 2016).

In this chapter, the study will describe the most relevant discussions surrounding the research study and bring the most important aspects to light and point out the current work, which directly affects the study. This research will look at NoSQL distributed systems running on commodity servers, using distributed consensus on each data store.

The sources of evidence as listed below:

- Commodity Servers
- Distributed Systems
- Distributed Consensus
- High availability and reliability
- Introduction to NoSQL data store

- Programming language Java

2.2 Background

Commodity servers are inexpensive computers that uses low resources, but have high failure rates that uses commercial off-the-shelf computer hardware components (Ngxande & Moorosi, 2014). As stated by Dorband et al. (2013:1), the purpose of commodity cluster computing is to utilize large numbers of readily available computing components for parallel computing. Parallel computing is distributing a large task into several single tasks by using more than one processor to execute the tasks (Ngxande & Moorosi, 2014).

The cluster computing architecture is where a set of loosely connected computers work together to be logically viewed as one computer. This method was adapted from distributed systems. Distributed systems are computers that are connected together that share computing tasks (Ngxande & Moorosi, 2014). Clusters consist of computers and switches. Two types of nodes exist namely, a master node and computing nodes or slaves connected across the network (Ghemawat et al., 2003).

2.2.1 Commodity servers

According to Baker et al. (2018), cluster computing is best characterized by a number of off-the-shelf commodity computers and their resources that are integrated by hardware, networks and software to behave like a single computer. The networks can have high speed or low – latency switches and this could be a single switch or a stack of switches. In essence, a computer cluster is a group of compute nodes working together to act as a single computer. This is to improve the performance and availability of a system other than that of a single computer (Baker et al., 2018).

The expanding of information and data made many organisations adopt technologies to analyse their data. As data grows rapidly, organisations needed to address this by adopting scalable systems that runs on hardware clusters of commodity servers and have specialized software to create distributed file storage systems.

2.2.2 High performance computing

To do research experiments for large data sets, most institutions have to make use of High Performance Computing (HPC) hardware, which can be very expensive. HPC makes use of

high computational power, which researchers use to analyse large data sets as pointed out by Ngxande, (2015:1). Scientific researchers used super computers to compute research problems, and more data intensive applications use specialized multicore processors and large amounts of memory to perform calculations (Middleton & Risk, 2015).

This led to the rise of the new trend of super computer designs using clusters of independent processors connected in parallel. Middleton & Risk (2015:5) found that computing problems using independent compute nodes use parallelism to distribute or divide the computing problem.

2.2.3 Commodity cluster computing

Commodity cluster computing evolved due to the need of high performance computing requirements. As described by Middleton & Risk (2015:6), a computer cluster can be defined as a group of individual computers sharing resources. Cluster computing improves the performance of applications. Cluster computing provides high availability and reliability and are more cost-effective than the single computer with the same performance (Middleton & Risk, 2015). System software and tools that provide parallel job execution environments is the key to capability and performance of the throughput of a computing cluster. Programming languages with parallel processing features that use high-degree of optimizations are needed to insure high-performance results and improves the programmer's productivity (Middleton & Risk, 2015).

The literature reveals that clusters using the available computing resources partition data with available computing resources are able to achieve performance and scalability depending on the amount of data. This approach is referred to as the "shared nothing" approach, since each node in the cluster is using parallel processing that consists of a processor, memory and disk resources that shares nothing with other nodes (Middleton & Risk, 2015). Clusters separate a problem into smaller parallel tasks and this makes them enormously effective, as there is no dependency or communication required between tasks, just the managing of tasks.

Studies that are more recent indicated the sequential workstations cannot provide enough computing power to applications and the only way to overcome this would be to improve the operating speed of the processors and other components to provide more power needed by computationally intensive applications (Baker et al., 2018).

This is evidence that applications have evolved to the use of parallel or distributed platforms

as they depend on the availability of multi core processors and fast networks. Off-the-shelf commodity hardware plays a big role and form part to support the high performance and high availability applications (Baker et al., 2018). As seen in Figure 2.1, independent computer nodes form a unified system with software and networking. When two or more computers solve a problem together, they are a cluster.

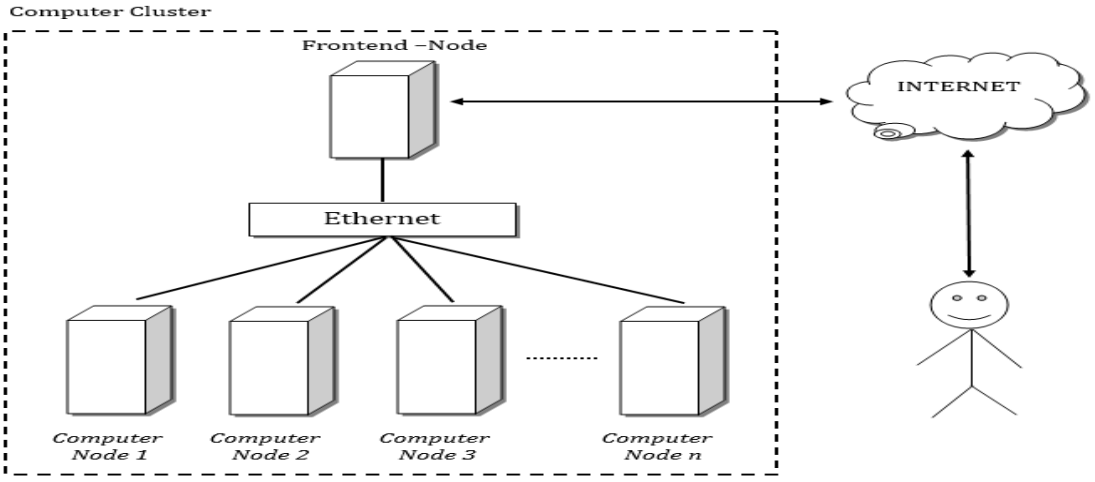


Figure 2.1: Cluster computing demonstrating

(Adopted from Baker et al., 2018)

According to Baker et al. (2018:2), clusters provide great computational power to High Performance Computing (HPC) hardware for High Availability (HA) and greater reliability than what a single computer can deliver. High Performance Computing (HPC) clusters grow in mass, they become very complex and time consuming to manage. That is why you need an automated cluster computing solution for tasks such as deployment, maintenance and monitoring services of the cluster.

2.2.4 Benefits of cluster computing

Clusters have three main benefits namely scalability, availability and performance (Baker et al., 2018). When a cluster runs parallel databases or cluster-enabled applications on compute nodes using their combined processing power, adding more nodes to the cluster achieves scalability. Availability is achieved when the nodes inside the cluster provides a backup to each other in the event of a failure of one of the nodes. In High Availability (HA) clusters, when one service or node fails, the service is redeployed to another node (or nodes) in the cluster. This is a transparent operation for the client as the applications and data running on the failed nodes get carried over to the failover nodes. The user does not know or care if the application is on a single server alternatively, a cluster.

It is clear from the above discussion that clusters provide scalable a capacity for compute data and support of mix workloads. They support horizontal and vertical scalability without downtime and the ability to handle unexpected peaks in workload. They have a central system management of a single systems image and 24x7 availability. They are cost efficient, flexible and have high availability of resources (Baker et al., 2018).

2.2.5 Commercial-off-the-shelf (COTS) hardware

High performance clusters use commercial off-the-shelf (COTS) hardware. This cluster makes use of Linux as their operating system. The COTS hardware used for high performance computing is classified as being homogeneous as each node has the same processor, memory and disk drives.

2.2.6 Parallel computing

Ngxande (2015:29) described parallel computing as using more than one processor to divide one large job into several tasks. Clearly, this showed that parallel computing solves larger problems and have a fast turn-around time. Parallel computing uses cheap inexpensive components to achieve high performance to overcome the limits of serial computing.

2.3 Distributed systems

NoSQL databases are classified as distributed systems because of their horizontal scalability on commodity clusters (Tiwari, 2011). A distributed system is an application that coordinates the actions of multiple processes on a network using a group of protocols, so that all modules cooperate to perform tasks (Ghemawat et al., 2003).

Reliability is one of the functions of distributed systems which suggests that a system should operate continuously, as defined by Shooman, (2002) in terms of a time interval instead of an instant in time. This system should work without pauses during an extended amount of time or periods (Hoda & Azad Kamali, 2014).

It should be highly available and recoverable so that nodes restart after failures (Birman, 2012). NoSQL databases perform better on commodity hardware systems, based on their ability to work in a cluster to gracefully recover from failures (Tiwari, 2011). This type of arrangement works well because of the consensus algorithm or distributed consensus, most NoSQL

databases use either Raft or Paxos consensus algorithms (Helland & South, 2007). Consensus algorithms like Paxos used to manage replication amongst distributed computer systems. These algorithms provide a mechanism to enforce consensus within a cluster (Ongaro & Ousterhout, 2014).

2.3.1 Introduction to distributed system design

A distributed system has the ability to connect remote users with remote resources in an open and scalable way (Sukuba, 2015). According to Sukuba, (2015:2) “open” means the protocols or component have an open interaction with other devices or components. When we say “scalable”, the system can accommodate changes for the number of resources, users and computing needs. Clearly, this showed that a distributed system, if given all the combined capabilities of the distributed components, could be much larger and powerful than the combinations of stand-alone systems (Sukuba, 2015).

For a distributed system to be reliable, it should have the following characteristics:
Fault-tolerant: it should be able to recover from component failures without any system problems when running tasks or processes. When components fail, the system should be able to restore operations permitted. If any failures occur, the failed components should be able to restart themselves and rejoin the system after being repaired. The system can act like a non-distributed system, meaning it can coordinate actions or tasks using multiple components during failures. This means that the system can operate even if the system is increased to a larger size. The system should provide responses in a timely manner. The system should have authentication and access needed to access the services (Sukuba, 2015).

It is clear from the above discussion that these are high standards and challenging to achieve, the most difficult characteristic would be that the distributed system should be able to continue operating even if system components fail.

You have to design a distributed system with the expectation that failures will occur (Sukuba, 2015). Sukuba (2015:2) notes: *“When you design distributed systems you have to expect failures to happen, so you design for failures. This should be the number one concern, for example, design for failure means if I sent a message to you and a network failure occurs, there would be two possible outcomes. One of the possible outcomes would be that the message got to you, the network had a failure and I did not get the response from you. The other concern would be that the message never got to you, because the network had a failure before it arrived.”*

Therefore, we would not know which of the two failures occurred. We can only determine the outcome by finding out from you if the message was delivered. The network has to be repaired or you have to come up online, because another outcome could be that the network was up and running but you died.

2.3.2 Challenges for a distributed system

Bouchrika (2018) described the major challenges of distributed systems as listed below:

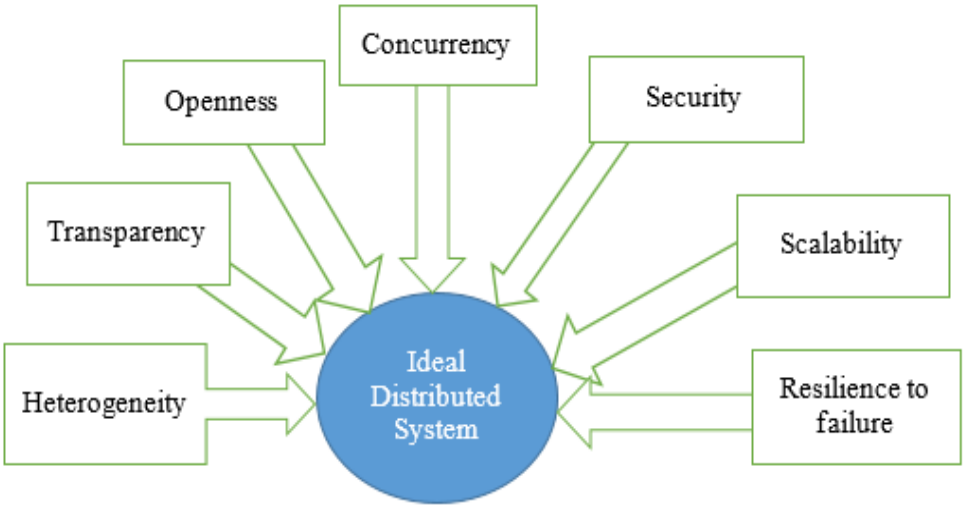


Figure 2.2: The major challenges in distributed systems

(Adopted from *Bouchrika, 2018)*

2.3.3 Heterogeneity

Heterogeneity, as described by Bouchrika (2018), is the collection of computers and networks that runs applications and services for the users to access over the internet. These collections of computers and networks include the following:

Hardware devices: computers, tablets, mobile phones, embedded devices, etc.
Operating System: MS Windows, Linux, Mac, UNIX, etc.
Network: Local network, the Internet, wireless network, satellite links, etc.
Programming languages: Java, C/C++, Python, PHP, etc.

Different roles of software developers, designers, system managers.

The different programing languages present different characters and data objects such as arrays and records. If developers use different programming languages and write programs, they should be able to communicate with each other (Bouchrika, 2018).

Middleware: this term applies to the software layer that provides a programming abstraction and masks the heterogeneity of the layers below like the networks, hardware, operating systems (OS) and programming languages. Middleware deals with the differences in operating systems and hardware (Bouchrika, 2018).

Figure 2.3 below illustrates a distributed system for several applications running on different operating systems where the middleware are responsible for the heterogeneity of the communications.

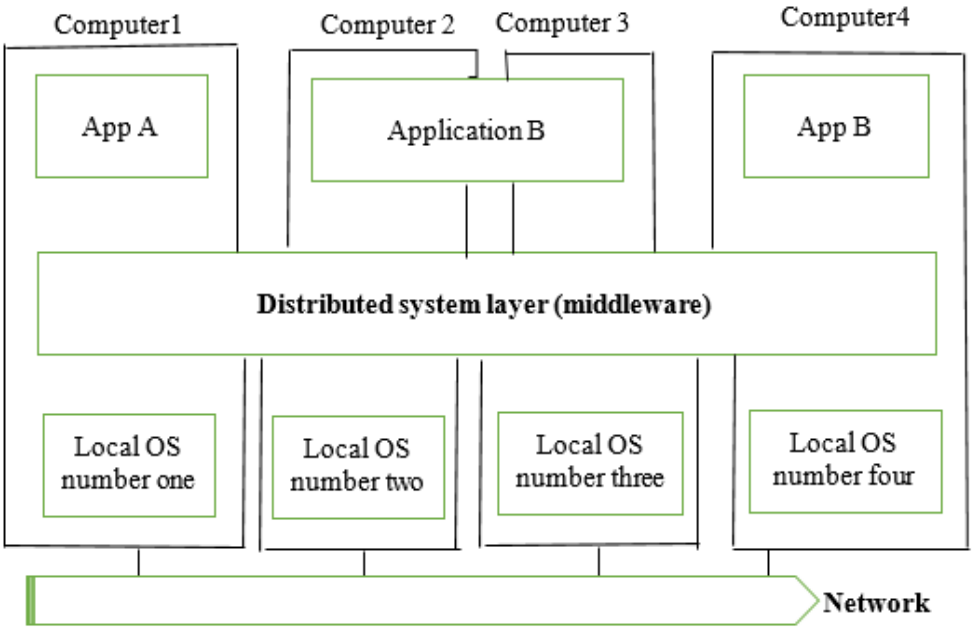


Figure 2.3: A distributed system for several applications running on different operating systems

(Adopted from Bouchrika, 2018)

2.3.4 Transparency

According to Bouchrika (2018), transparency is defined as to conceal the separation of components that make up the distributed system, so the user and the developer sees the system as one system. Some terms of transparency in distributed systems are:

Access	Hide differences in data representation and how a resource is accessed.
Location	Hide where a resource is located.
Migration	Hide that a resource may move to another location.
Relocation	Hide that a resource may be moved to another location while in use.
Replication	Hide that a resource may be copied in several places.
Concurrency	Hide that a resource may be shared by several competitive users.
Failure	Hide the failure and recovery of a resource.
Persistence	Hide whether a (software) resource is in memory or a disk.

2.3.5 Openness

The openness of a distributed system determines if the system can be re-implemented. The point of when new resource sharing services can be added determines this. It should be easy for developers to add new features or replace sub systems in the future. For example, Twitter and Facebook have an Application Programming Interface (API) that allows developers to develop their own software interactively (Bouchrika, 2018).

2.3.6 Concurrency

For an object to be safe in a concurrent system, the operations should be coordinated. This is so that it remains consistent when several clients attempt to access a shared resource as a data structure that records bids for an auction that is accessed (Bouchrika, 2018). This can be achieved by semaphore used in operating systems.

2.3.7 Security

Information resources in distributed systems have a high essential value to their users. Their security is therefore of substantial importance. Security for information resources has three components namely, confidentiality (protection against disclosure to unauthorized individuals), integrity (protection against alteration or corruption) and availability for the authorized (protection against interference with the means to access the resources) (Bouchrika, 2018).

2.3.8 Scalability

Distributed systems must be scalable as the number of users increase. Neuman (1994), defines the scalability as the following: “A system is said to be scalable if it can handle the

addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity.”

Scalability has three dimensions:

- Size
  Number of users and resources to be processed. Problem associated is overloading.
- Geography
  Distance between users and resources. Problem associated is communication reliability.
- Administration
  As the size of distributed systems increases, many of the systems needs to be controlled. Problem associated is an administrative mess (Bouchrika, 2018).

2.3.9 Failure handling

Computer systems fail from time to time. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation. The handling of failures is particularly difficult (Bouchrika, 2018).

2.3.10 Gray failures

Huang et al. (2017:4), described the gray failures as limping around in a degraded mode, you trying to keep the system at your best and mask the problems, this in the end are one of the main causes of availability breakdowns and performance anomalies in distributed systems. The larger you scale, the more common gray failures become.

Gray failure characterized in an abstract model as shown in Figure 2.4 below:

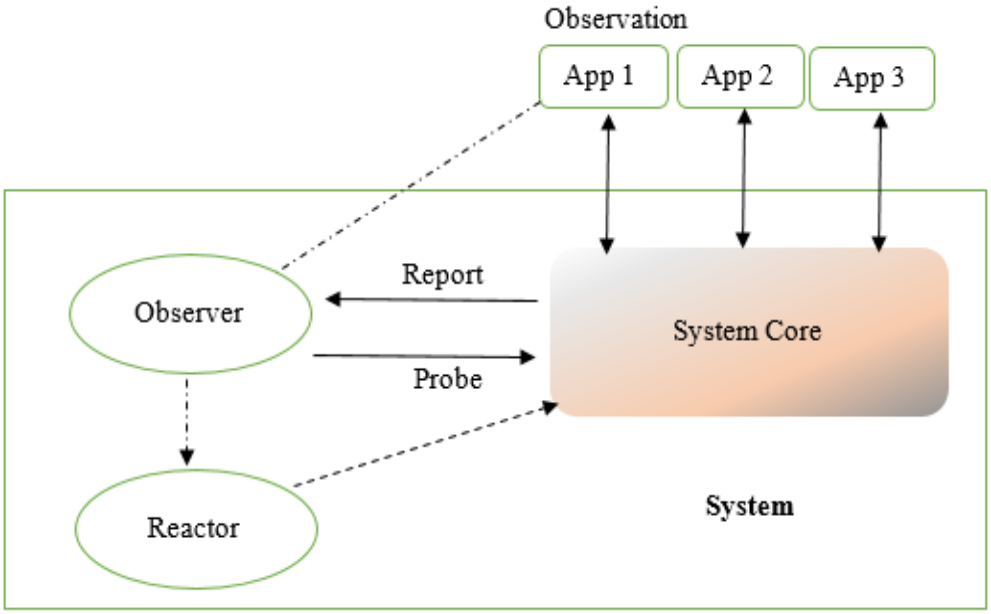


Figure 2.4: An abstract model to characterize gray failure

(Adopted from Huang et al., 2017)

From Figure 2.4 above we can observe that a failure detector (Observer) is monitoring the system. If the observer detects a fault, the reactor takes action (for example restarting components) and this happens while the user accesses applications. These applications make their own observations regarding the health of the system like are responses slow, errors reported, etcetera.

Huang et al. (2017:3), defined: *"gray failure as a form of differential observability. More precisely, a system is defined to experience gray failure when at least one app makes the observation that the system is unhealthy, but the observer observes that the system is healthy."*

Figure 2.5: Quadrant of gray failure

(Adopted from Colyer, 2017)

Huang et al. (2017:3), argued that,

> *"Initially the system experiences minor faults (latent failure) that it tends to suppress. Gradually,* seen in Figure 2.6 *the system transits into a degraded mode (gray failure) that is externally visible, but which the observer does not see. Eventually the degradation may reach a point that takes the system down (complete failure), at which point the observer also realizes the problem. A typical example is a memory leak."*

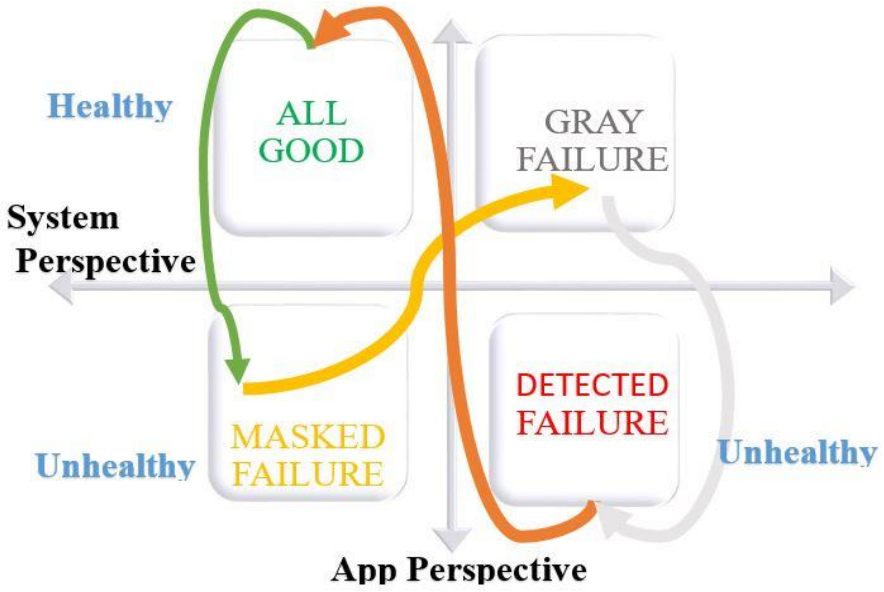


Figure 2.6: Gray failure cycle

(Adopted from Colyer, 2017)

2.4 Basics of a distributed system

According to Takada (2018) there are two consequences of distribution when dealing with distributed programming. The first one would be that information travels at the speed of light and the second would be that independent mechanisms fail independently. Distributed programing is about dealing with the distance and having more than one of the same mechanisms.

2.4.1 Distributed programming

Takada (2018) emphasizes that distributed programming is about solving the same problem that one would use on a single computer, but using multiple computers due to the problem not residing on the same single computer. Computation and storage would be very fast if done on a single, fast reliable system hosted in the cloud by a cloud service provider.

But most communities don't have these resources, they would try to upgrade the hardware, but as the problem they are trying to solve increases, you may not solve the problem by just using one computer. Figure 2.7 illustrates how the performance gap between high-end and commodity hardware decreases with the cluster size. Adding a new computer to a cluster would increase the performance and the capacity of a system, but this would not be possible, as we need to take in to account that the computers are all separate from each other. Data sent across the nodes and the commutation tasks must be coordinated.

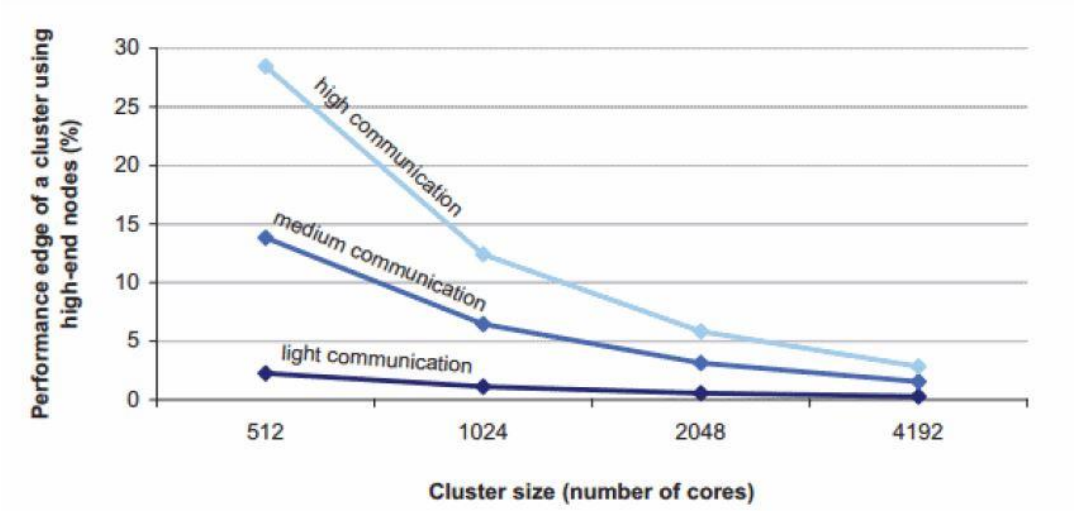


Figure 2.7: Performance advantage of a cluster built with high-end server nodes over a cluster with low-end server nodes

(Adapted from Barroso et al., 2013)

Ideally, adding a new machine would increase the performance and capacity of the system linearly. However, of course this is not possible, because there is some overhead that arises due to having separate computers. Data needs to be copied around, computation tasks have to be coordinated and so on (Barroso et al., 2013).

2.4.2 Scalability

According to Takada, (2018) a problem becomes harder once you pass a certain volume in size and then reach a size limit. The scalability of a system should be to handle the increase of volumes in a capable manner and be able to handle the given load. Adding more nodes to a system should make the system operate faster. Multiple data centers spread across the globe should reduce the time it would take to respond to user queries. A system should have the ability to add more nodes to the system without causing any administrative costs.

2.4.3 Performance and latency

According to Takada (2018), performance is the amount of work done by a computer system compared to the time and resources used. Performance can be achieved by a short response time or low latency. The high throughput using low utilization of computing resources would also increase the performance of the system.

Latency is about the delay measured between the initiation phase and the event when something happened (Takada, 2018). Based on Takada (2018), latency is not about the amount of old data, but the speed by which new data gets generated. To measure latency would be to measure how long it writes new data to make it visible to the users.

2.4.4 Availability

If a system cannot be accessed by a user, it would be classified as unavailable. As pointed out by Takada (2018), a distributed system can tolerate failures whereas a single system cannot. Clearly this showed that distributed systems can be built with unreliable components together and still build a reliable system layer on top of the system. Therefore, systems that have no redundancy can only be available as their underlying components. And systems built with redundancy can be tolerant of partial failures and be more available (Takada, 2018).

The formula for availability can be described as: Availability = uptime / (uptime + downtime)

Availability is about being fault tolerant. The probability of failures occurring would increase the number of components. When the number of components, servers and datacenters increases, the system should not become less reliable (Takada, 2018).
For example:

Availability %	How much downtime is allowed per year?
90% ("one nine")	More than a month
99% ("two nines")	Less than 4 days
99.9% ("three nines")	Less than 9 hours
99.99% ("four nines")	Less than an hour
99.999% ("five nines")	~ 5 minutes
99.9999% ("six nines")	~ 31 seconds

2.4.5 Replication

According to Takada (2018), replication is one of many problems in distributed systems. Replication provides a context or many sub-problems such as leader election, failure detection and consensus. Replication allows the system to achieve scalability, performance and fault tolerance (Takada, 2018).

For example, let's say there is a database that clients make requests that would change the state of the database. This arrangement and communication pattern has several stages as shown in Figure 2.8. Stage one shows the request that the client sends to the server, then stage two synchronisation takes place. At stage three a response gets returned to the client and at phase four asynchronous replication takes place.

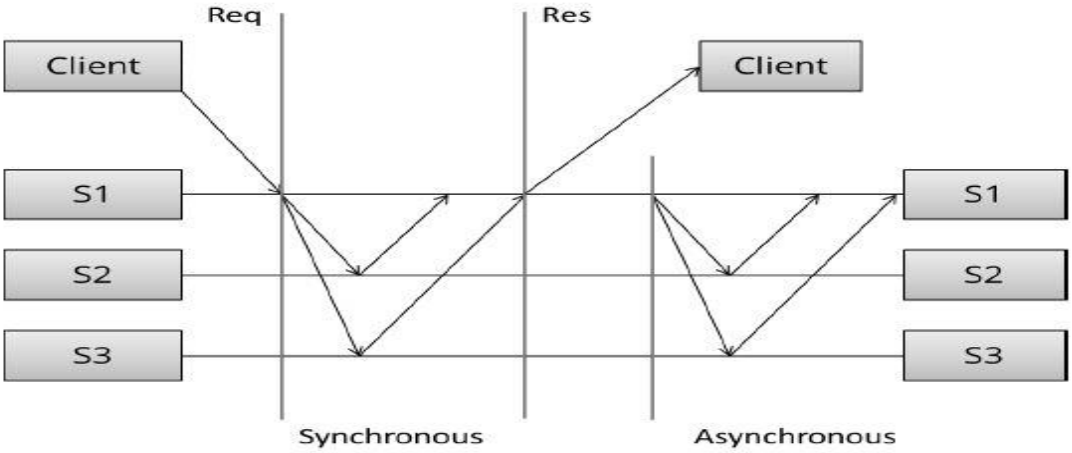


Figure 2.8: Understanding replication in databases and distributed systems ((Adopted from Wiesmann et al., 2002)

2.4.6 Synchronous replication

During the synchronous phase the first server contacts the other servers and waits for replies from the other servers before sending a response to the client. The client is blocked because it has to wait for replies from the system. This type of system cannot tolerate the loss of any servers. If and when a server is lost, the system won't be able to write to all the nodes, such a system can allow read-only access, but won't allow write-access to the data (Takada, 2018).

2.4.7 Asynchronous replication

Asynchronous replication can be seen as a passive replication. The master or leader sends the response back to the client immediately, so the client is not forced to wait for the communication to occur between the servers. The master server contacts the other servers to update their copies of the data (Takada, 2018).

From a performance perspective this type of system is fast and this system is also more tolerant of network latency. If nothing goes wrong the data is replicated to all the servers, but if the server containing the data is lost, all the data will be permanently lost (Takada, 2018).

2.5 Replication algorithms

Distributed systems should behave like a single system, this ensures that only a single copy of the system is active and that the replicas are always in agreement. This is known as the

consensus problem (Takada, 2018).

The term consensus is a collective decision process where a group of workstations must all agree in order to operate as a group. The master node will send broadcasts to the rest of the network and the various different computers making up the network (called ‘nodes’) come to an agreement to make a collective decision.

Consensus algorithms allow a cluster of servers to act as a coherent group. They should be able to continue functioning even though some servers experience hardware failures (Ongaro & Ousterhout, 2014).

According to Rao et al. (2011), 2-way replication (master–slave) can lead to a disaster and huge data loss. This resulted in introducing a 3-way replication (master-master-slave) to protect against data loss or disk failures running on commodity servers. Consensus gives the assurance that the drive will be available for all reads and writes during 3-way replication.

2.5.1 Partition tolerant consensus algorithms

> *“Raft is a consensus algorithm for managing a replicated log. Its structure is different from Paxos; this makes Raft more understandable than Paxos. It also provides a better foundation for building practical systems. Raft separates the key elements of consensus, such as leader election and log replication. Raft enforces a stronger degree of coherency to reduce the number of states. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety”* (Ongaro & Ousterhout, 2014).

2.5.2 CRDTs: convergent replicated data types

For a set of operations to converge on the same value in an environment, the operations need to be order-independent and insensitive to duplication or redelivery (Takada, 2018). So their operations need to be:

- Associative (a+(b+c)=(a+b)+c), so that grouping doesn't matter.
- Commutative (a+b=b+a), so that order of application doesn't matter.
- Idempotent (a+a=a), so that duplication does not matter.

These data objects are known as semi lattices in mathematics. A partially ordered set known

as the lattice has a distinct top and a distinct bottom, a semi lattice is like a lattice, but only has a distinct bottom. To guarantee convergence, a data type should be expressed as a semi lattice data structure (Takada, 2018).

For example, if you calculate the max( ) of a set of values, this will always return the same results regardless of order in which the values were received, because max( ) operation is associate, cummutative and idempotent (Takada, 2018).

Another example by Takada (2018), two lattices: one drawn for a set, where the merge operator is union (items) and one drawn for a strictly increasing integer counter, where the merge operator is max (values).

The data indicated that data types can be expressed as semi lattices where you can have replicas that communicate in any patterns and receive updates in any order, they will eventually agree on the end result as long as they see the same information (Takada, 2018).

The limitation of expressing a data type as a semi lattice requires a level of interpretation, as many data types have operations that are not order dependent. When adding items to a set, its associative, cummutative and idempotent. But if items are removed from a set then they would need to resolve the conflicting operations like add (A) and remove (A).

This means that data types have implementations such as CRDTs that makes it a different tradeoff by resolving conflicts and also doing it in an order-independent manner. NoSQL Key-value data store deals with registers and for a developer to know which data store to choose he needs to make use of the right data type to avoid anomalies (Takada, 2018).

> According to Takada (2018), *"A lattice is a partially ordered set with a distinct top (least upper bound) and a distinct bottom (greatest lower bound). A semi lattice is like a lattice, but one that only has a distinct top or bottom. A join semi lattice is one with a distinct top (least upper bound) and a meet semi lattice is one with a distinct bottom (greatest lower bound)."*

Some examples of the different data types specified as CRDT's include (Takada, 2018):

- Counters
    - Grow-only counter (merge = max (values); payload = single integer)

    - Positive-negative counter (consists of two grow counters, one for increments and another for decrements)
- Registers (Key-value store)
    - Last Write Wins -register (timestamps or version numbers; merge = max (ts); payload = blob)
    - Multi-valued -register (vector clocks; merge = take both)
- Sets
    - Grow-only set (merge = union (items); payload = set; no removal)
    - Two-phase set (consists of two sets, one for adding, and another for removing; elements can be added once and removed once)
    - Unique set (an optimized version of the two-phase set)
    - Last write wins set (merge = max (ts); payload = set)
    - Positive-negative set (consists of one PN-counter per set item) observed-remove set
- Graphs and text sequences

To ensure an anomaly-free operation, developers should use the right data type for their application. For example, if you only going to remove an item once, then a two-phase set would be used. If you need to add items to a set and never remove them then a grow-only set works (Takada, 2018).

Clearly, this showed *that "Not all data objects have known implementations as CRDTs, but there are CRDT implementations for Booleans, counters, sets, registers and graphs"* (Takada, 2018).

2.5.3 The CALM theorem

CALM tells programmers which operations and programs can guarantee safety when used in an eventually consistent system. Any code that fails CALM tests is a candidate for stronger coordination mechanisms.

"Consider building a database for queries on stock trades. Once completed, trades cannot change, so any answers that are based solely on the immutable historical data will remain true. However, if your database keeps track of the value of the latest trade, then new information such as new stock prices might retract old information, as new stock prices overwrite the latest ones in the database. Without coordination between replica copies, the second database might return inconsistent data" (Bailis & Ghodsi, 2013).

Clearly, order-independence is an important property of any computation that converges: if the order in which data items are received influences the result of the computation, then there is no way to execute a computation without guaranteeing order. However, the order of statements does not play a significant role in many programming models. For example, in the Map Reduce model, both the Map and the Reduce tasks are specified as stateless tuple-processing tasks that need to be run on a dataset. Concrete decisions about how and in what order data is routed to the tasks is not specified explicitly, instead, the batch job scheduler is responsible for scheduling the tasks to run on the cluster.

Similarly, Structured Query Language (SQL) specifies the query, but not how the query is executed. The query is simply a declarative description of the task, and it is the job of the query optimizer to figure out an efficient way to execute the query (across multiple machines, databases and tables). Of course, these programming models are not as permissive as a general-purpose programming language. Map Reduce tasks need to be expressible as stateless tasks in an acyclic dataflow program. SQL statements can execute fairly sophisticated computations, but many things are hard to express in it (Takada, 2018).

"Programming models which express a desired result, while leaving the exact order of statements up to an optimizer to decide, often have semantics that are order-independent. This means that such programs may be possible to execute without coordination, since they depend on the inputs they receive, but not necessarily the specific order in which the inputs are received" (Takada, 2018).

2.5.4 Partition and replicate

There is a manner in which a data set gets distributed between multiple nodes. This is very important for any computation to occur, the data need to be located to act on it. Two basic techniques can be applied to a data set. The first one is to split the data set over multiple nodes (partitioning) to allow for more parallel processing. The second one would be to copy or cache on different nodes to reduce the distance between the client and the server to allow for greater fault tolerance (replication) (Takada, 2018).

We can illustrate the difference between the two techniques below:

Figure 2.9 illustrates the partitioned data (A and B) divided into independent sets. The replicated data (C) is copied into multiple locations.

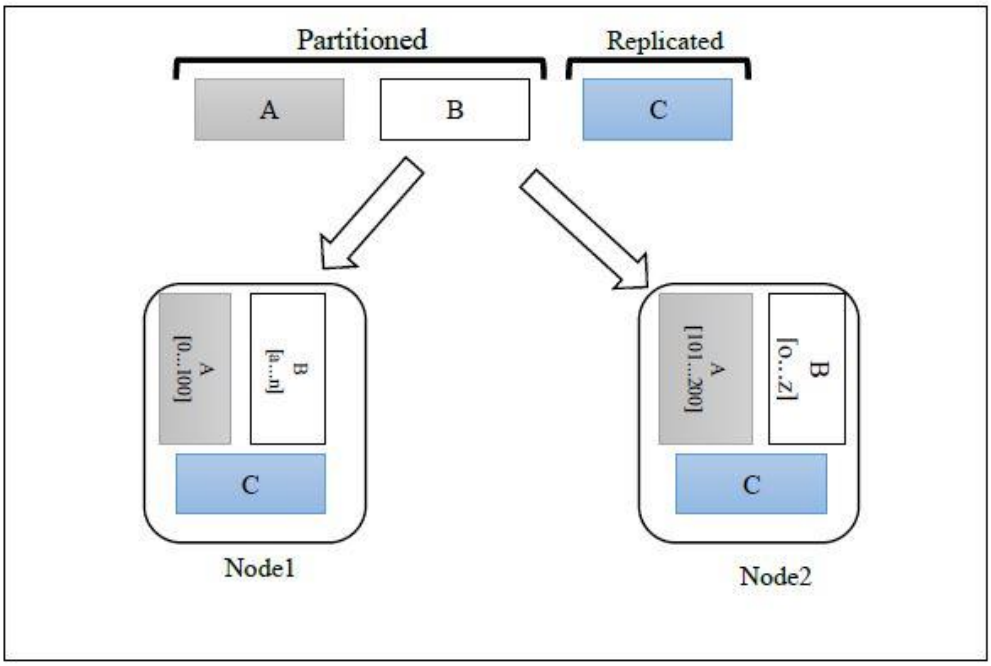


Figure 2.9: Splitting of partitioned data and replicated data

(Adopted by Takada, 2018)

2.5.5 Partitioning

Partitioning is dividing the dataset into smaller distinct independent sets; this is used to reduce the impact of dataset growth since each partition is a subset of the data. Partitioning improves performance by limiting the amount of data to be examined and by locating related data in the same partition.

Partitioning improves availability by allowing partitions to fail independently, increasing the number of nodes that need to fail before availability is sacrificed. Partitioning is also very much application specific, so it is hard to say much about it without knowing the specifics. Partitioning is about defining partitions based on the primary access pattern and dealing with the limitations that come from having independent partitions (e.g. inefficient access across partitions, different rate of growth etc.).

2.5.6 Replication

Replication is making copies of the same data on multiple machines; this allows more servers to take part in the computation. Replication improves performance by making additional computing power and bandwidth applicable to a new copy of the data. Replication improves

availability by creating additional copies of the data and increasing the number of nodes that need to fail before availability is sacrificed. Replication is about providing extra bandwidth and caching where it counts.

2.6 Introduction to NoSQL data store

The term Structured Query Language (SQL) evolved in the nineteen seventies to store structured data in relational data stores (Lourenço, Abramova, et al., 2015). Most NoSQL databases are designed to scale well in the horizontal direction and these data stores are key-value, wide column, graph and document stores described below in detail.

2.6.1 Key-value stores

Key-Value based are closely related to document stores as they store values against a key and there is no need for schemas to be associated with the values. The programing language used for some NoSQL data stores are written in C (Vaish, 2013). Key-value stores have the ability to process the data in real time. Key-value stores provide horizontal scalability through nodes in clusters and used in applications where results are rapidly required (Zafar et al., 2017).

The key-value store database is used in web applications for session management. The database uses a key-value pair scheme to store data to provide support for interaction with social media applications (Zafar et al., 2017). Data stored on key-value data stores are shared between nodes. The data stores include Redis, Riak, Kyoto, Cabinet, Amazon Dynamo Database (DB), Couch DB, Berkeley DB, Memcached, Aerospike, EHCache, Voldermot and Cassandra (Zafar et al., 2017).

As much as 300K to 400K read/write operation can be achieved with an Intel Core 2 Duo, 2.4 GHz using key-value stores for application development when operated on custom built computers. Figure 2.10 shows the structure of a key value based database in which there is one key and many values related to that key (Zafar et al., 2017). Table 2.1 shows the summary of key-value stores and features of each one.

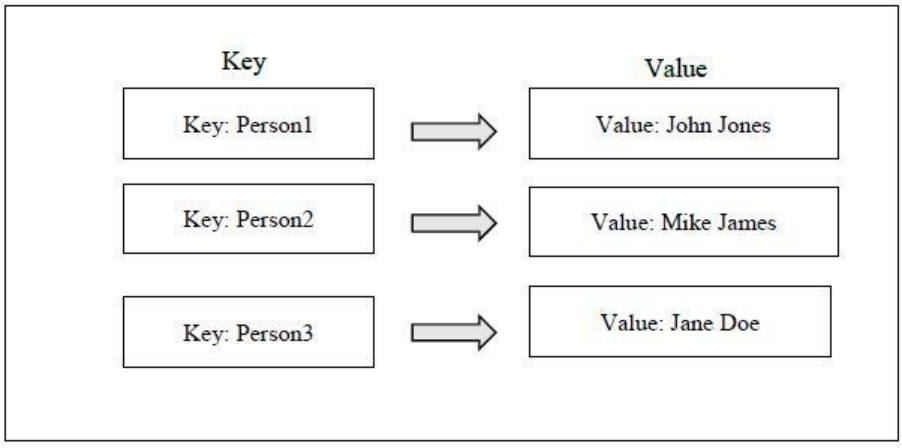


Figure 2.10: Key-value data store

(Adapted from Zafar et al., 2017)

Table 2.1:Summary of key-value data store features (adapted from Zafar et al., 2017)

Data store name	Features
Memcached	Shared-nothing architecture, in-memory object caching systems with no disk persistence. Automatic sharding but no replication. Client libraries for popular programming languages including Java, .Net, PHP, Python, and Ruby.
Cassandra	Shared-nothing, master-master architecture, in-memory database with disk persistence. Key range based automatic data partitioning. Synchronous and asynchronous replication across multiple data centers. High availability. Client interfaces include Cassandra Query Language (CQL), Thrift, and MapReduce. Largest known Cassandra cluster has over 300 TB of data in over 400-node cluster.
Redis	Shared-nothing architecture, in-memory database with disk persistence, ACID transactions. Supports several data Redis structures including sets, sorted sets, hashes, strings, and blocking queues. Backup and recovery. High availability. Client interface through C and Lua.
Voldemort	Shared-nothing architecture, in-memory database with disk persistence, automatic data partitioning and replication, versioning, map and list data structures, ACID with relax option, backup and recovery, high availability. Protocol Buffers, Thrift, Avro and Java serialization options. Client access through Java API
Riak	Shared-nothing architecture, in-memory database with disk persistence, data treated as BLOBs, automatic data partitioning, eventually consistency, backup and recovery, and high availability through multi data center replication. Client API includes Erlang, JavaScript, MapReduce queries, full text search, and REST.
Aerospike	Shared-nothing architecture, in-memory database with disk persistence. Automatic data partitioning and synchronous replication. Data structures support for string, integer, BLOB, map, and list. ACID with relax option, backup and recovery, high availability. Cross data center replication. Client access through Java, Lua, and REST.

2.6.2 Wide column stores (extensible record stores)

Column-oriented databases store data in columns instead of rows like in a RDBMS. The implementing programming languages used are JAVA, Python and Go (Vaish, 2013). In RDBMS, the data is processed based on row-major order. The rows are exclusively recognized by the auto generated IDs for each row. On the contrary, the Column based database follows the column-major order (Zafar et al., 2017).

A column database is similar to relational database. The column based databases are similar to the key value structure. Database applications is categorized by short variations and easy to use schema. If the data is changed, it is stored with a changed version of same column data by the addition of a timestamp. Fractional data access may increase the performance of some applications (Zafar et al., 2017).

The data store performs different operations such as the computation of datasets. The performance is enhanced by gathering the columns with same features in a family. The concept of column family is similar to the column notion in the relational databases (Zafar et al., 2017). The data stores include Apache Cassandra, Google Big Table, HBase, Hypertable, Cloudata, Oracle RDBMS Columnar Expression, and Microsoft SQL Server 2012 Enterprise Edition.

Table 2.2 contains the name of NoSQL databases and features for a better understanding and selection of database system for user. Figure 2.11 shows an example of column store and a row store. The database sorts the data by columns to ensure rapid retrieval (Zafar et al., 2017).

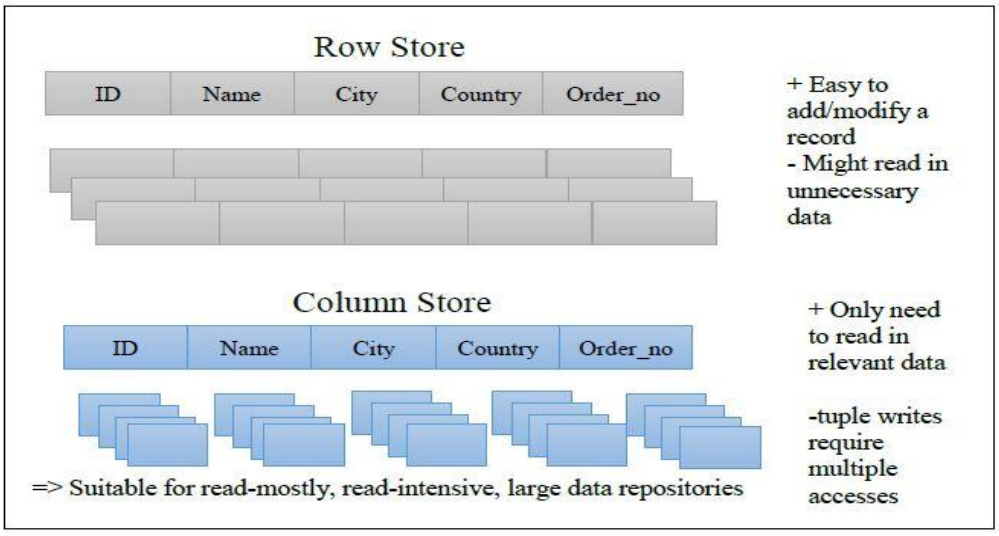


Figure 2.11: Column store and row store tables

Adapted from Zafar et al., 2017)

Table 2.2: Summary of column data store features (adapted from *Zafar et al., 2017)*

Data store name	Features
BigTable	A sparse, persistent, distributed, multi-dimensional-sorted map. Features strict consistency and runs on distributed commodity for data that is in the order of billions of rows with millions of columns.
HBase	Open Source Java implementation of BigTable. No data types and everything is a byte array. Client access tools: shell (i.e., command line), Java APT, Thrift, REST, and Avo (binary protocol). Row HBase keys are typically 64-bytes long. Rows are byte-ordered by their row keys. Users distributed deployment model. Works with Hadoop Distributed File System (HDFS), but uses file system APT to avoid strong coupling. HBase can also be used with CloudStore.
Cassandra	Provides eventual consistency. Client interfaces: phpcasa (a PHP wrapper), pycassa (Python binding), command line/shell, Thrift, and Cassandra Query Language (CQL). Popular for developing financial services applications.

2.6.3 Graph databases

This is a special category of NoSQL databases that characterizes relationships as graphs. This may include social relationships amongst people and many other network topologies, the implementing programming language are JAVA code (Vaish, 2013). The core algorithm is the graph data model. In some applications, the associations between entities are more important than the entities, this can be dynamic or fixed. The social media applications data are modeled with graphs. Graph data models are used in many industries e.g., oil and gas and airlines. (Zafar et al., 2017).

The database that is built using graph data models include Infinite Graph, Titan, Microsoft Trinity, Flock Database (DB), Orient DB, DEX, Facebook open graph, Google knowledge graph and Neo4J.

Figure 2.12 shows the example of graph based database that shows the data is linked and the correlation of the data.

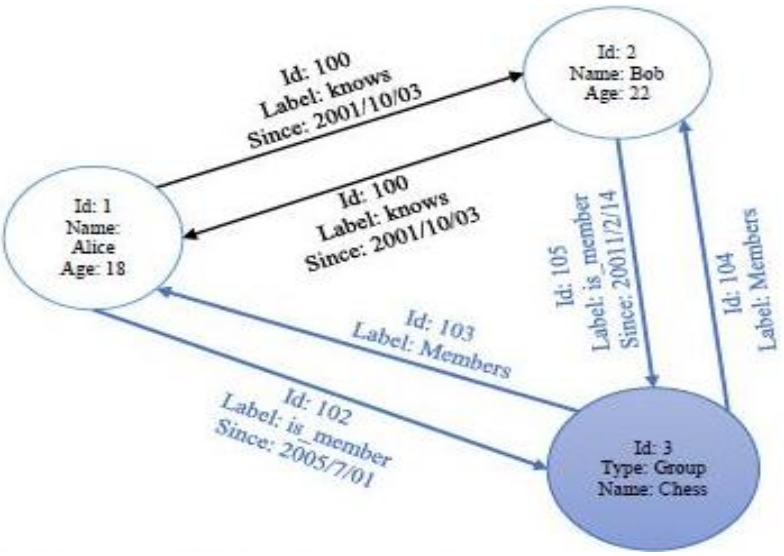


Figure 2.12: Graph data store

(Adopted from Zafar et al., 2017)

Table 2.3: Graph data store features (adapted from *Zafar et al., 2017)*

Data store name	Features
Neo4J	In-memory or in-memory with persistence. Full support for transactions. Nodes/vertices in the graph are described using properties and the relationships between nodes are typed and relationships can have their own properties. Deployed on compute clusters in a single data center or across multiple geographically distributed data centers. Highly scalable and existing applications have 32 billion nodes, 32 billion relationships, and 64 billion properties. Client interfaces: REST, Cypher (SQL-like), Java, and Gremlin.
AliegroGraph	In-memory or in-memory with persistence. Full support for transactions. Nodes/vertices in the graph are described using properties and the relationships between nodes are typed and relationships can have their own properties. Deployed on compute clusters in a single data center or across multiple geographically distributed data centers. Highly scalable and existing applications have 32 billion nodes, 32 billion relationships, and 64 billion properties. Client interfaces: REST, Cypher (SQL-like), Java, and Gremlin

2.6.4 Document stores

Document stores allow the inserting, retrieving and manipulating of semi-structured data. These databases will use XML, JSON, BSON or YAML where the data access will be over the HTTP protocol by using RESTful API's and this provides flexibility. The Implementing programing languages used are JAVA, Erlang, C++, and C (Vaish, 2013).

According to Zafar et al. (2017:6), the document based database relies on documents or texts.

The system is used to manage non-structured data sets as key-value data types. These databases are identified by their key identifier. Using non-structured document, main values are efficient for applications used for document groups. The databases which are designed using this technique include Mongo DB, Couch DB, CouchBase, Raven DB and Fat DB. Figure 2.13 shows the example of a document based database operation and Table 2.4 shows the document data store features (Zafar et al., 2017).

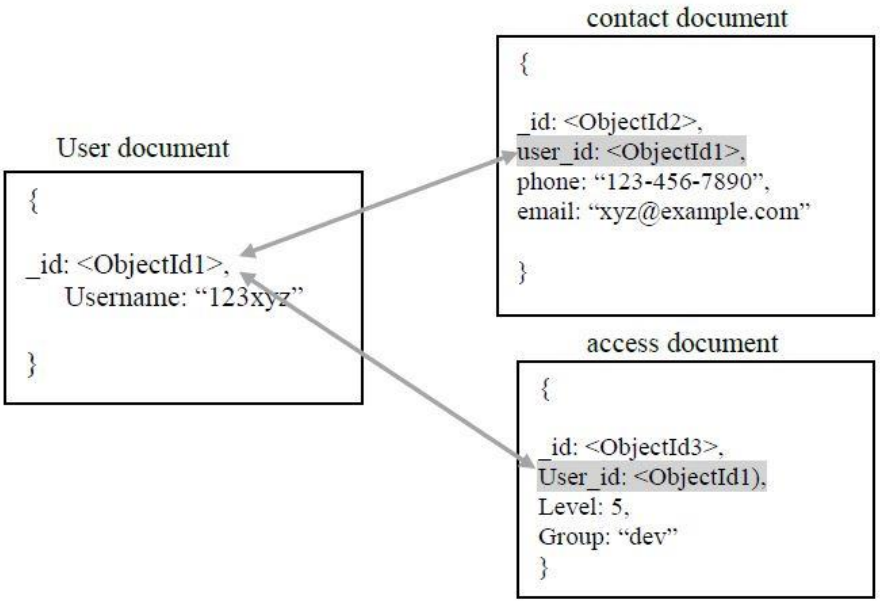


Figure 2.13: Document store

(Adapted from Zafar et al., 2017)

Table 2.4: Document data store features (adapted from *Zafar et al., 2017)*

Data store name	Features
MongoDB	No transaction support. Only modifier operations offer atomic consistency. Lack of isolation levels may result in phantom reads. Uses memory-mapped files storage. Support is available for geospatial processing and Map Reduce framework. Indexing, replication, GridFS, and aggregation pipeline. JavaScript expressions as queries. Client access tools: JS Shell (command line tool), and drivers for most programming languages. Suitable for applications that require auto-sharding, high horizontal scalability for managing schema-less semi-structured documents.
CouchDB	Open Source database written in Erlang. JSON format for documents. Client access tools: REST API, CouchApps (an application server), and MapReduce. JavaScript is used for writing Map Reduce functions.
Couchbase	Incorporates functionality of CouchDB and Membase. Data is automatically partitioned across cluster nodes. All nodes can do both reads and writes. Used in many commercial high availability applications and games.

According to Zafar et al. (2017:7), There are many applications for NoSQL and each application has a unique architecture and structure. To date, the size of data in NoSQL has greatly increased. To better understand the NoSQL data stores, four categories of NoSQL data stores have been discussed.

2.6.5 Features of NoSQL data stores

According to Hu et al. (2016:2), NoSQL data stores can handle structured, semi-structured and unstructured data. NoSQL data stores are simple and faster. NoSQL data stores allow object-oriented programming. NoSQL data stores have the ability to connect multiple NoSQL data stores so that they can work as a single logical unit together using horizontal scalability. Modern applications like mobile applications desire efficient and scalable databases, NoSQL data stores meet these requirements, NoSQL do not intend to replace relational databases, but they complement each other (Hu et al., 2016).

2.7 Programming language Java

According to Dwarampudi et al.(2010), it's difficult to determine the scope of one programming language over another one. Java is an object-oriented language that is similar to C++. Scala is a language that addresses the needs of a modern developer, it reduces the code by two or three times compared to Java. C++ is a middle-level language as it has both high and low level language features (Dwarampudi et al., 2010).

Based on criteria for each programming language used during the study Dwarampudi et al.

(2010:2), claims that Java is regarded amongst others as one of the best default security programming languages. Java provides web services and have outstanding XML (Extensible Markup Language) support with libraries, APIs (Application Programming Interface) and many frameworks (Dwarampudi et al., 2010). Java also provides aspect oriented programming using extensions. Java can also perform automation, macros and shell scripting on different platforms (Dwarampudi et al., 2010).

In addition to the studies described above, Dwarampudi et al. (2010) conclude that every language has its ups and downs. Every language has its own specialty and better programing practices made it popular to revolutionize the computing industry. Both Java and PHP languages are two very good languages for developing web-applications. However, Java has better performance and is more scalable than PHP. Both languages confidently address the most common website attacks of cross-side-scripting, path manipulation and SQL injection. However, Java is much more secure than PHP. The present paper is a fraction of the study conducted to compare Java and PHP languages. The aim would be to provide an answer to the question of which of the two languages is the best for web programming (Dwarampudi et al., 2010).

2.7.1 The reactive manifesto

Reactive systems deal with problems effectively without affecting the users using the application, this approach simplifies error handling (Bonér et al., 2014). Reactive systems stay responsive and highly-available during failures. If a system is not resilient it will not be responsive after failures. To achieve resilience, the application should be able to replicate, be contained, isolated and delegated. If failures occur this would be contained within each component thus isolating the components from each other (Bonér et al., 2014). This ensures that if certain parts of a system fail it can be recovered without compromising the system or application. The recovery of each component would then be delegated to another component and high availability will be ensured by replicating where necessary without affecting the client (Bonér et al., 2014).

Reactive systems handle the workload of the end-user by increasing or decreasing the resources needed for an application. This gets achieved by predictive live performance measures and reactive scaling algorithms. This makes the system more elastic to run on commodity hardware (Bonér et al., 2014).
Reactive systems rely on asynchronous message-passing to get a boundary established between the components that ensures loose coupling. Using the message-passing enables

load management, elasticity and flow control by the shaping and monitoring of message queues in the systems. The non-blocking communication allows recipients to make use of resources only when being active, to allow for less system overhead (Bonér et al., 2014).

The reactive manifesto describes how to build modern architecture that can scale as more resources are needed (Sachdeva, 2019). Reactive systems should follow the following principles:

1. The system should be able to respond in a timely manner and be responsive.
2. The system should be able to handle workload under high load to be classified as elastic.
3. The system should be able to handle failures of components and be resilient.
4. The system should use asynchronous message passing between its components and be message driven.

The first three principles listed above, relates to the architectures choices of design. Micro services architecture and technologies like Docker containerization are important aspects of Reactive systems (Sachdeva, 2019). Running a single LAMP (Linux Apache MySQL PHP) stack does not meet the requirements of the Reactive Manifesto.

There are a few attributes for Java developers based on the last principle. When failures happen the program should gracefully handle the situation, than to throw out an exception. Back pressure is an important attribute of reactive programming. This happens when a database query returns thousand records queried. But when the client cannot accept any more records then the client goes into a blocked state. This will keep the database in a waiting state and continue with other operations (Sachdeva, 2019). The last attribute is non-blocking, where the program uses multiple CPU threads and consumes more resources and to be able to send requests in a non-blocking manner and less switching of threads.

2.7.2 Reactive programming in Java

Reactive programing was first introduced in 1960, and became very popular during the last few years (Sachdeva, 2019). Reactive programming, like functional programming is just another programming paradigm. Reactive programming deals with asynchronous data streams and change propagation in an ordered manner. Reactive programming provides simplistic solutions for high-load applications. Social networks, chat clients, proxy servers, load balancers and real-time data streaming.

Reactive programing and reactive systems can be used together interchangeably, but they are

not the same. Reactive systems are the design and architecture that allows the developer to build a responsive application. The combination of the two gives an application more benefits to make them more loosely coupled and efficient use of resources (Sachdeva, 2019).

The next phase of the literature review included a detailed, systematic and transparent means of gathering appraising and synthesizing peer-reviewed evidence to answer the study's sub-question one: What are the different hardware requirements for NoSQL data stores to operate on, to achieve high availability and reliability?

The systematic reviews were used to reduce bias at all stages of the review process.
The stages of conducting the review:

1 Defining the systematic review questions
2 Followed by a proposed methodology for the review or review protocol
3 Then a search strategy was performed to conduct a thorough search of literature
4 Results were screened against a pre-specified selection criterion to identify the included studies
5 An appraisal of the quality of studies found
6 Synthesizing the evidence found
7 Then the results were published to disseminate the review
8 Update the review where new evidence was found

2.8 Systematic review

To begin the first step of the systematic review the study identified the goals of this review. The study wanted to find solutions done by other researchers and how they differ from each other and look at the implications they would have on the research study. A systematic literature review paper by Kitchenham et al. (2010:2) argued that all empirical software engineers should adopt evidence-based practices as this is a repeatable research method and could be replicated by other researchers in the field (Milani & Navimipour, 2017).

2.8.1 Defining research questions

RQ1: What empirical evidence exists to help developers choose the right NoSQL database for their project that would guarantee high availability and reliability?
RQ2: How does the empirical evidence found, compare with each other in terms of their approach, methods and constraints used in addressing RQ1?
RQ3: What is the strength of the empirical data found for each solution or approach used?

I. What implications would these findings have?

RQ4: What are the implications of the findings for the design of the new empirical data framework of what has been found in RQ3?

I. What is the gap that could be created or exploited to create a solution after looking at other solutions?

2.8.2 Defining the systematic literature review protocol

The study also explored that reviewing the existing literature and addressing a particular topic, the normal review process rarely shows any systematic procedures used. This does not ensure the use of all relevant material to be included in a study, mostly just material that supports the arguments in a review. Most research methods and frameworks do exist in the field of computing but software engineers are not exposed to these procedures and frameworks (Kitchenham et al., 2010). A systematic review should be the obvious research method for this research. The study tried to understand the problem and find a solution on how to solve the problem (Milani & Navimipour, 2017). The systematic review has improved the position of computer science and software engineering in many ways. Authors and students benefit having clear procedure-driven research when reviewing the background material for their thesis and identifying where the background supports or conflicts with the thesis.

The goal to define the systematic review was to specify systematically how the review process works. To see how each stage of the process unfolds, and to focus on the objective of the review process. Before moving on to the next stage, each stage of the review process was reviewed. The study specified the steps before starting any stage in order to keep the structure and integrity of the review process and the objective of the systematic review.

2.8.3 Search strategy

The research followed a two-step procedure for the search strategy:

I. Gather a list that includes all sources that would be relevant studies for the review.
II. Decide how the study would search these sources.

The study used the university library online research databases to create the list of sources. The study also added relevant journals searched online manually. As part of the review, the study made sure that all the sources are active and available by importing them into Mendeley-reference-management software for citations.

The search process was restricted to search only for published journal articles and papers presented at conferences. The query strings applied based on the titles and abstract, papers from 2014-2017 was searched using the online research database. The most used databases we searched a keyword or string was the Google scholar search engine. The online research databases used included ACM digital, Emerald insight, IEEE Explore, ScienceDirect and SpringerLink. The online search databases are illustrated in Table 2.5. Our four groups created formed the search string, which are the key terms used as shown in Table 2.6.

From the search terms, the study created the four groups and added synonyms to search similar words that have the same meaning or related semantic meanings within the study literature. Directly related to the first systematic review research question (RQ1) and based on results (RQ2) would be answered by uniquely gathering different sets of research studies: Group 1 finds all the studies that focuses on NoSQL data stores; Group 2 finds the literature related to databases; Group 3 find all the studies relating to relational databases; Group 4 finds literature about programming data structures. The study used a Venn diagram to create an intersection of the four sets as illustrated in Figure 2.16. The study used a Boolean expression for the search strategy by using the logical OR-operator within the groups to focus the studies to include any of the terms used per group. The study then put the groups together with an AND-operator. The study used at least one of each of the terms in each group as seen in Table 2.6 when performing the search string. Practically this allowed to use the one search string to

perform 48 different searches as shown below:

([*G1, T1*] *OR* [*G1, T2*]) OR ([*G1, T3*] *AND* ([*G2, T1*] *OR* [*G2, T2*] *OR* [*G2, T3*]) OR ([*G2, T4*] *AND* ([*G3, T1*] *OR* [*G3, T2*] *OR* [*G3, T3*] *OR* [*G3, T4*] *AND* ([*G4, T1*] *OR* [*G4, T2*])

Table 2.5: Online search databases adopted from *(Milani & Navimipour, 2017)*

Online Research database	Website URL
Google Scholar	https://scholar.google.co.za/
ACM Digital	http://dl.acm.org
Emerald Insight	http://www.emeraldinsight.com
IEEE Explore	http://www.ieeexplore.ieee.org
SpringerLink	http://link.springer.com
ScienceDirect	http://www.Sciencedirect.com

Table 2.6: The four groups of search terms

	Group 1	**Group 2**	**Group 3**	**Group 4**
Term 1	NoSQL	Databases	Relational	Data structures
Term 2	Not-only SQL	Data stores	RDBMS	hierarchical
Term 3	Non-relational	Databanks	Relative	
Term 4		Records		

2.8.4 Search results

This contains the summarized results of the systematic literature review. Out of 61 papers only 23 articles were selected and qualified; these are shown in Table 2.7. The articles published from 2014 to 2017 are all related to NoSQL data stores, relational databases and relational data structures. This search query used to gather the research articles from the online research databases from 2014 – 2017.

Table 2.7: Search results

Year	Online research databases / Digital libraries						Total
	IEEE	ACM	Science Direct	Emerald	Springer	Scholar	
2010							
2011	1						
2012							
2013	1			1			
2014	3			1		2	
2015		2			2	1	
2016	3				1		
2017	3		2				
Total	11						
Analyzed	11	2	2	2	3	3	23

As shown below in Figure 2.14 there was a visualization of the keywords used when using the

Boolean expression searching for articles during the last seven years and the number of publications and research articles based on the Boolean expression used. As illustrated in Figure 2.14 and Figure 2.15, there was a significant increase of research articles from 2012 to 2014.

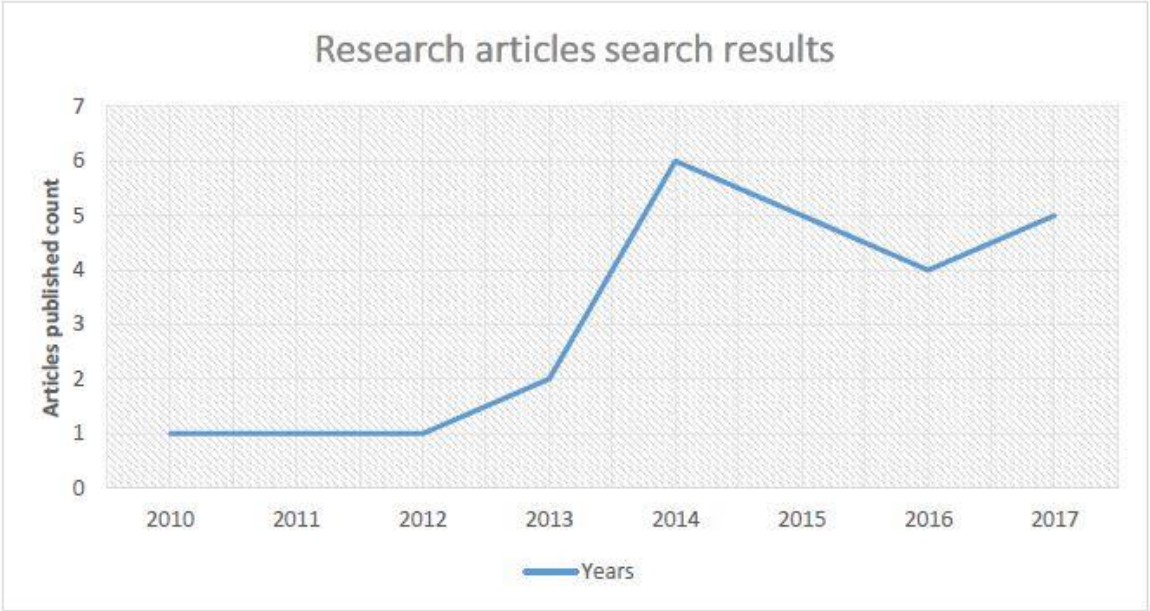


Figure 2.14: Importance of topic by years

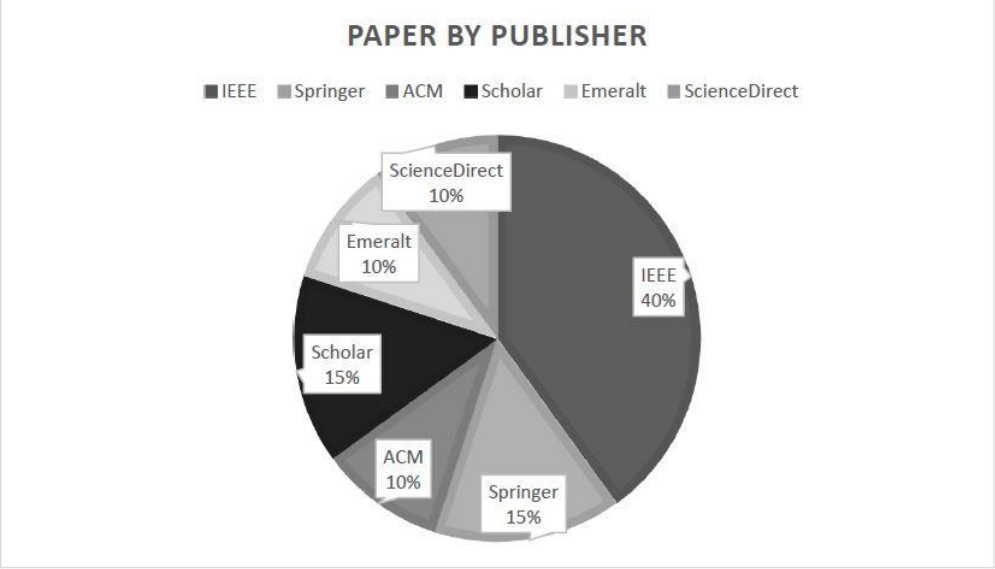


Figure 2.15: Distribution of paper by publisher adopted from *(Daiga PLASE, 2017)*

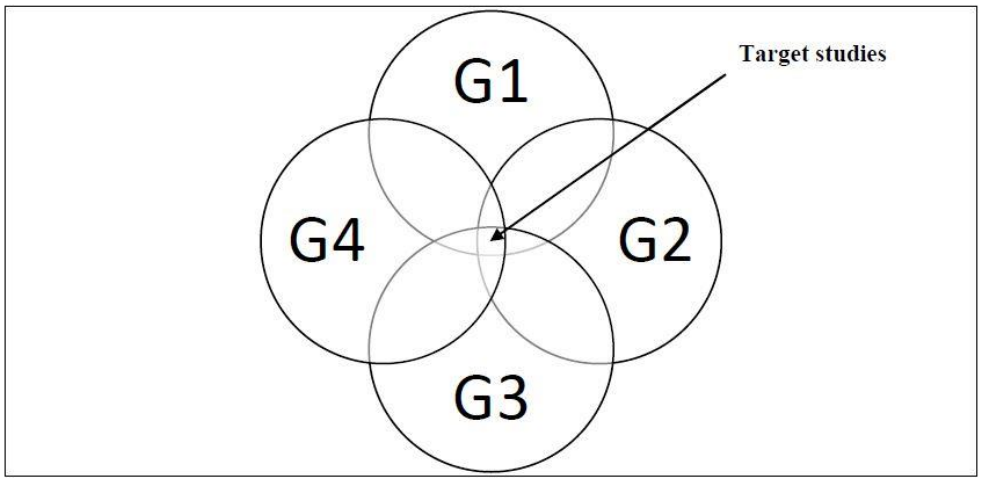


Figure 2.16: Venn diagram showing the combination of the search terms used

As mentioned, the Venn diagram was used to create an intersection of the four group sets shown in Table 2.6 above. This allowed to use the Boolean expression and search the key terms. Below in Table 2.8, the study applied an inclusion criteria to match each article against the criterion.

Table 2.8: The inclusion criteria for the study

Criterion Identifier	Criterion
IC1	Articles concerning NoSQL data store types and platforms
IC2	Articles concerning NoSQL issues and challenges
IC3	The study focuses on choosing the right NoSQL database
IC4	The study focus on comparing NoSQL and RDBMS data structures
IC5	Articles concerning NoSQL data store experiments
QC1	Is there a clear statement of the aims of the research
QC2	Is the study put into context of other studies and research

2.8.5 Study selection

The study selected research papers based on the online research database search query. Articles about NoSQL and relational databases and data structures were selected. The study checked the title that's relevant to the study and the research and filtered the search query. The study ignored books and removed articles that were not published and focused on studying the abstracts and the conclusion of these papers (Milani & Navimipour, 2017).

The study had a goal to filter down the studies found during the search stage and there were studies relevant to answer the research question. The study used a set of inclusion criteria and quality screening criteria as illustrated in Table 2.8 and Table 2.9.

The inclusion criteria used a three stage process:

1. Abstract inclusion criteria screening
2. Full-text inclusion criteria screening
3. Full-text quality screening

The study summarized the exclusion criteria using these stages:

1. The exclusion criteria were based on the study area (Software engineering and computer science).
2. Exclusion based on the title of the article.
3. Exclusion based on removing books and unpublished articles.

Exclusion based on the abstracts and conclusions.

Table 2.9: Exclusion criteria for the study

Criterion Identifier (Exclusion)	Criterion (Clarification)
EC1	Articles created or published prior to January 2012, Figure 2.14 shows the number of hits published, concerning NoSQL in our six databases used for our systematic reviews, there is a rise in the number of articles published since 2012.
EC2	Articles comparing NoSQL with SQL, it's not relevant in this research
EC3	Articles concerning the CAP theorem, it's not relevant in this research
EC4	Articles not written in English language, this is necessary for the authors and its reviewers of this paper
EC5	Articles only focused on comparing NoSQL data store types without discussing how they distribute the data across nodes
EC6	Articles that's not free to obtain through the Cape Peninsula University of Technology library, don't have a budget to purchase articles
EC7	Articles without experiments on how to setup NoSQL databases on systems.

The study found a number of articles that did not contain NoSQL or databases. The study filtered the studies based on reading the abstracts of the 61 papers found during the search phase. If the studies in the abstract included the first two inclusion criteria, they would have been accepted for the next stage of the study. Based on the inclusion criteria above only papers from 2014 to 2017 were selected containing the IC1 to IC3, due to time constraints the study had to filter the results from 2014 till present. The result of the abstract filtering rejected 29 articles and 21 articles passed to the full-text screening. The study filtered out the articles that failed to meet requirements from the inclusion criteria IC4 and IC5. The study could not get all the full details of the articles using only the abstracts, therefore needed the full-text inclusion criteria screening, and applied the same strategy used for the abstract inclusion

criteria. The results of this stage were that the study rejected eight articles leaving 15 articles for the final stage of the study selection process.

In the final stage of the study selection, the study had to filter out studies that did not meet the quality criteria QC1 and QC2. In this stage, the study assessed the remaining articles to see if these articles meet the quality criteria. All 15 articles passed the quality screening criteria and they all formed the literature basis for the systematic review.

2.8.6 Quality assessment

To focus on the RQ3 where the study needed to look at the strength of evidence presented by the studies in the review, the study accessed the quality of each study using the following criteria:

Table 2.10: Quality assessment questions and answers

ID	Question / Answer
QC1	Is there a clear statement of the aims of the research? 1.0: Yes, there is a clear statement 0.5: Partly, part of the statement 0 : No statement
QC2	Is the study put into context of other studies and research? 1.0: Yes, study is put into context with other studies 0.5: Partly put into context 0 : Not specified
QC3	Are system/algorithm design decisions justified? 1.0: Yes, the design decisions are justified 0.5: partly decisions justified 0 : No system or algorithm decision
QC4	Is the test data set reproducible? 1.0: Yes, the data set is reproducible 0.5: Partly reproducible 0 : No reproducible test data
QC5	Is the study algorithm reproducible? 1.0: Yes, the study algorithm is reproducible 0.5: Partly reproducible 0 : No study algorithm used
QC6	Is the experimental procedure thoroughly explained and reproducible? 1.0: Yes, the experimental procedure is properly explained 0.5: partly reproducible 0 : no experimental procedure
QC7	Is it clearly stated in the study which other algorithms the study algorithm(s) have been compared 1.0: Yes, algorithms have been compared 0.5: Partly compared algorithms 0 : No algorithms compared
QC8	Are the performance metrics used in the study explained and justified? 1.0: Yes, performance metrics are used and properly explained

	0.5: Partly explained and justified 0 : No performance metrics used
QC9	Are the test results thoroughly analysed? 1.0: Yes, the test results are thoroughly analysed 0.5: Partly analysed 0 : Nothing analysed
QC10	Does the test evidence support the findings presented? 1.0: Yes, evidence does support findings 0.5: Partly support findings 0 : No findings presented

2.8.7 Rationale for the criteria

The first two criteria were used in the study selection process; these two are considered a good research practice as they clearly identify the aims of the systematic review search strategy and allows the current research to be in context of the other research. The third criteria checked that the current process or algorithms used are properly analysed. This allows the study to analyse an approach, before conducting experiments on it and to answer (RQ4). The criteria QC4, QC5, QC6 and QC8 are related to reproducible research to allow other researchers to reproduce each step done of the work done by the study, to compare results to their own approaches. Criteria QC7 looked at the approaches done and how the study compared the results of other studies against the current study. Criteria QC9 and QC10 checked if the results are thoroughly analysed as a quality assessment to check if the evidence supported the findings presented in the study. There should be a clear connection between the evidence presented and the conclusions of the study. The study accessed each article with the quality questions above and exclusions as illustrated Table 2.9.

2.8.8 Results of the review

Nine articles were selected for the final selection. Upon a closer look, the final selection revealed that eight of the articles all performed experiments. Benchmark tests were performed on eight of the articles selected. None of the articles selected tested the programing language drivers used in NoSQL data stores (see Appendix A).

To answer the systematic review research questions, the study investigated selected papers in order to provide a broader understanding of the research aim and to highlight the review research questions. According to Yassien & Desouky (2016:1), researchers have a challenge to determine which data model they should use for applications. The study also explored different workloads studying the latency, throughput and runtime using a Yahoo Cloud Serving Benchmark test. Yassien & Desouky (2016:1) found that modern web applications require high

availability and low latency to handle big data sets.

The datasets were executed using an experimental research setup on a single workstation. Yassien & Desouky (2016:2) used three different databases, one SQL database and two other NoSQL databases. More studies should be conducted to cover more data models as not all of the NoSQL models were used in this research. Yassien & Desouky (2016:8) found that SQL is ideal for applications that just read mostly from databases. HBase, which is a NoSQL column-family store database, utilized more CPU resources and writes in memory of average 4GB of RAM. MongoDB, which is a document store NoSQL database, had a high allocation of memory used with an average of 8GB RAM.

According to Yassien & Desouky (2016:8), more studies would need to be done for the rest of the NoSQL data stores and choosing the right database system for an application would still be a high critical task. Some studies analyzed the role of NoSQL and the challenges it presents when having to choose the right NoSQL database. As pointed out by Bajpayee et al. (2015:1), the architecture decision should be made based on the requirements as the features of NoSQL databases are not always the same and can't make any general comparisons.

According to Bajpayee et al. (2015:2), its impractical to prototype a design for production as this would require hundreds of servers and new databases are constantly emerging. The aim of the research used two-use cases first, was to retrieve recent medical results for a particular patient; second was to read a new medical test result from all locations about a medical patient updated. Bajpayee et al. (2015:2) compared the performance and scalability using one server scaling to nine instances. The authors used three NoSQL data stores, MongoDB, which is a document store; Cassandra which is a column store and Riak, which is a key-value store database.

The nine instances replicated a distributed environment across three data centers. In addition to the studies described above, Amazon EC2 cloud instances had to be created to run the database servers and the test client was also executed using an Amazon cloud instance. Bajpayee et al. (2015) used the Yahoo Cloud Serving Benchmark for their test client to test latency and throughput. The test client used a selected workload and ran it three times and the client threats were simulated from 1 -10, 25, 50, 100- 500, 1000. Bajpayee et al. (2015) argued that using the single node made the research impractical to use for a production environment. Clearly, this showed that they wanted to illustrate which database can distribute the records efficiently and use the single node to give insight why scaling and adding more nodes would be better than using faster nodes with more storage.

Not all the 21 papers from 2014 to 2017 focus directly on the programming language drivers used by developers to store read and write to databases. The papers did not give a clear and reliable answer to the research sub-question three about the best performing language driver to use. In summary, there is little evidence NoSQL data stores have advantages over relational databases management systems as they can scale on commodity hardware (Qi et al., 2014). Using inexpensive commodity hardware, NoSQL scaling horizontally proves to be cost effective handling large volumes of data, to distribute the data for replication (Qi et al., 2014).

NoSQL data stores can also be defined as fault-tolerance to handle server failures and continue uninterrupted. One advantage over relational databases systems is that NoSQL do not have data structure requirements like relational database management systems, they can virtually support any data structure (Qi et al., 2014).

The next phase of the review provides the results of the key results obtained in the literature review articles, these were papers found by searching the sources of evidence discussed in the introduction of the literature review.

2.9 Findings of literature review

It is clear from the findings of the literature review, that while studies have been performed on performance and reliability of distributed systems and commodity hardware, commodity hardware or low-end cost-efficient server hardware as defined by Barroso & Hölzle (2013:24), are the building blocks for data centers. Most of the performance benchmarks were done on HPC (High End Performance Clusters). This is an expensive approach, as opposed to using inexpensive commodity computers.

NoSQL Key-value data stores have to handle registers and developers should know which data store to use. Developers should use of the right data type to avoid inconsistencies (Takada, 2018). To ensure an anomaly-free operation, developers should use the right data type for their application.

There is limited research done on the programming language use on specific data types and empirical evidence for the existence of associations between code quality programming language choice, language properties, and usage domains, could help developers make more informed choices (Ray et al., 2017).

> The data collected by Ray et al. (2017:10), *"indicates functional languages are better than procedural languages; it suggests that strong typing is better than weak typing; that static typing is better than* dynamic*; and that managed memory usage is better than unmanaged. Further, that the defect proneness of languages in general is not associated with software domains. In addition, languages are more related to individual bug categories than bugs overall."*

Meyerovich & Rabkin (2013:9), studied factors that determined programming language adoption: the first would be the use of open source libraries that is available for developers to use; this is the most influential factor for choosing a programming language. The second is the availability of existing code as this outweighs intrinsic or build-in functions such as language simplicity, safety and ranking. The third factor would be the developer experience, libraries and legacy code. The fourth factor is the company size, as employees at large enterprises that value legacy code more than smaller size companies, simplicity, platform constraints and development speed matter less. Compared to developers overall, those at larger organizations weigh commercial libraries more and open source libraries less (although open source is still weighted more highly). These factors help developers choose which language to adopt.

Based on the finding of the literature review, NoSQL data stores needed to be reliable and highly available for this research study. In order for an application to be scalable and responsive it needs to use the Reactive Manifesto. High availability applications should use the Reactive Manifesto supported by the CQRS approach design. This will allow the research to focus on what NoSQL database to put on the write-side model and what database to put on the read-side of the model.

The next section will discuss the research method chosen for this study, based on the research sub-question two, research sub-question three and research sub-question four, as well as the objectives we needed to achieve. The unit of measurement and data collection methods will be discussed. The data analysis will explain how we analysed the data captured and what methods we applied.

CHAPTER THREE
RESEARCH METHODOLOGY

3.1 Introduction

A research process has to be mapped properly. This allows academics to accept the results of a study (Terblanche et al., 2013). Thus this chapter will cover a detailed description of the research process and the reason for the study. The research methodology is built around the research questions which arise from external factors as a need to solve a problem (Terblanche et al., 2013). The main research question and sub-questions are discussed on page 13 within this thesis.

> The term 'research design' and 'research methods' are many times used interchangeably but according to Jalil (2013), *"they are distinct concepts. 'Research design' refers to the logical structure of the inquiry. It articulates what data is required, from whom, and how it is going to answer the research question."*

Figure 3.1 shows the five components that forms part of the research design.

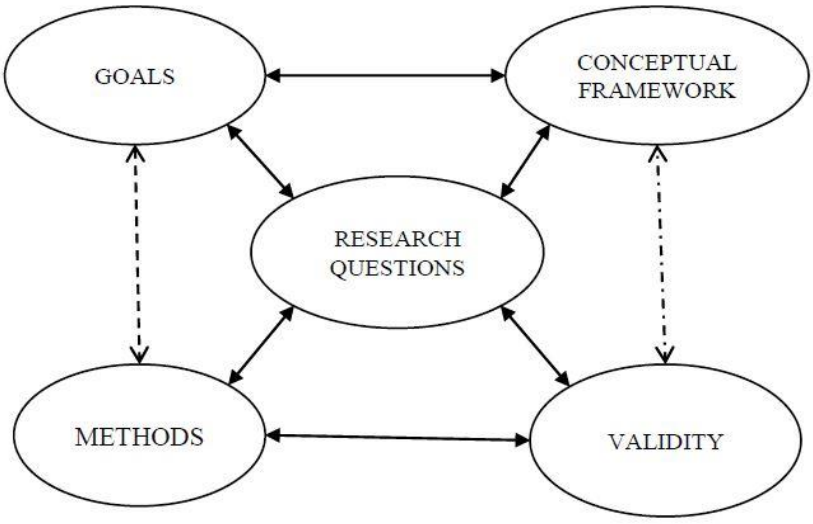


Figure 3.1: Model of the research design

(Adopted from *Maxwell, 1998)*

The top part of the model gets developed first, because the research questions should have a clear relationship with the goals of the study. The goals of the study should be based on the current knowledge and theories relevant to the study. The methods used in this study should

answer the research questions and deal with possible validity threats to the answers.

3.2 Research paradigms

To understand the philosophical underpinnings of the research design and methods, the study needed to draw a clear line between ontology and epistemology. Ontology deals with the existence of truth and facts and believes about reality. Epistemology is concerned about learning about the reality and deals with the nature of knowledge and to gain knowledge. Axiology refers to the study's aims of the research and to clarify the value of the research. Thus a positivism approach was taken as this research was independent from the data gathered and maintained an objective stance.

A positivist researcher believes that an object that's being studied has characteristics which can be measured using quantitative research methods (Terblanche et al., 2013). A useful insight into paradigms would be that ontology deals with the nature of reality, epistemology deals with the relationship between the researcher and the research object, and methodology deals with how we gather knowledge about the world (Steenhuis & de Bruijn, 2006). The ontological viewpoint of a positivist is that the researcher and the research object are considered independent of each other and also logically aligned, thus the preferred methodological choice is experimentation, manipulation and testing of the hypothesis (Steenhuis & de Bruijn, 2006).

The study followed a positivism paradigm by using a quantitative methodology approach. The use of this scientific method allowed an experiment to be setup to get measurements of what the study wanted to observe as the main goal, to discover and provide results (Mertens, 2014). Research is one of various ways of knowing or understanding, to do a systematic inquiry designed to collect, analyse, interpret and use data. Therefore research can be conducted for different reasons in order to understand, describe, predict, or to do empirical research by building existing knowledge about an educational phenomenon (Mertens, 2014). Please see Figure 3.2 below illustrating the scientific research approach.

To construct the hypothesis, we used the following variables see Table 3.1:

A. NoSQL data stores (dependent)
B. Nodes (Extraneous)
C. Programming language driver, JAVA (independent)
D. Data objects (independent)

The study could change the value of D to see the effect it had on A and C. The value of D depended on the different test cases. The study increased and decreased this value to see the effect it had on A and C. The Conceptual framework as illustrated in Figure 3.4 allowed the study to test the experiment after constructing the hypothesis.

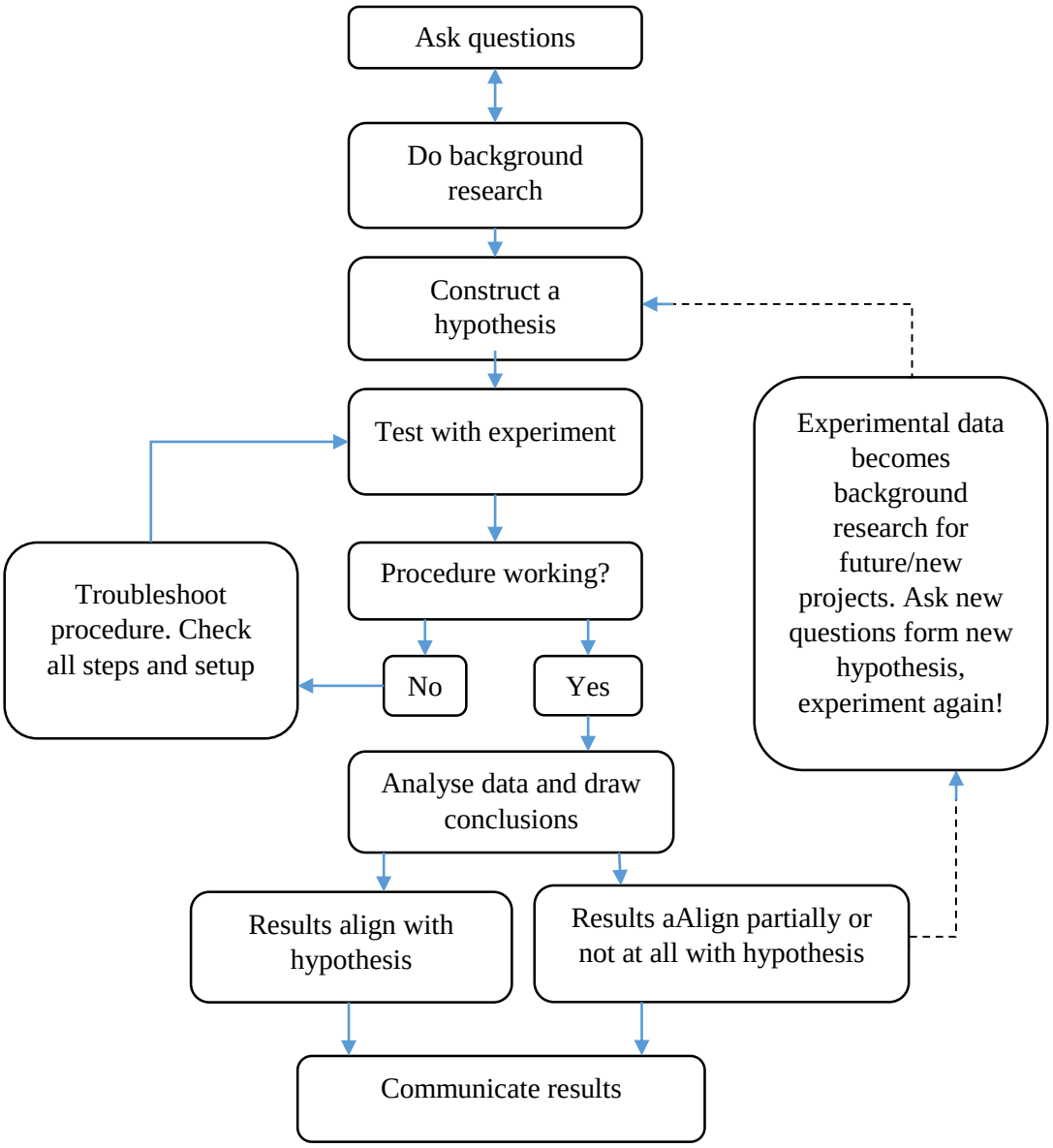


Figure 3.2: Scientific method research paradigm

As illustrated in Figure *3.3* below, the first step of the empirical cycle would be observation, by collecting the empirical facts and organizing it. The second step would be induction where the hypothesis gets formulated on the basis of the observed facts. The hypothesis needs to be defined in measurable variables in order to derive concrete predictions (Steenhuis & de Bruijn,

2006).

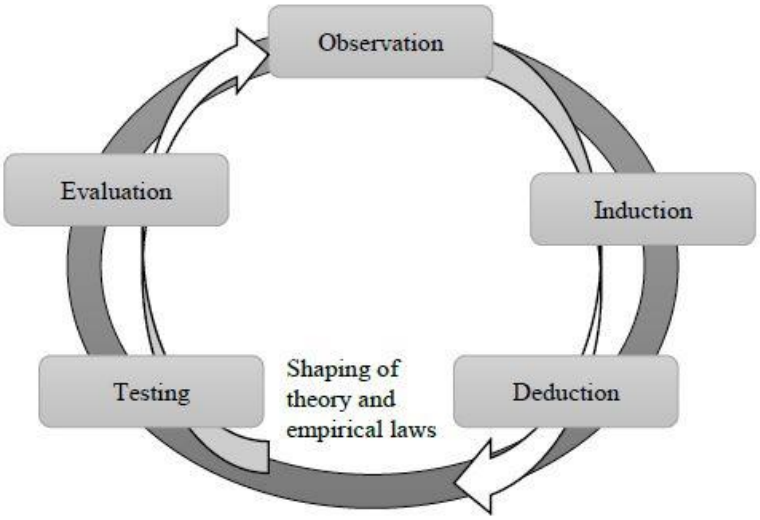


Figure 3.3: Empirical cycle

(Aadapted from *Steenhuis & de Bruijn, 2006)*

According to Steenhuis & de Bruijn (2006:3), *"the main focus is the deduction phase, then next the predictive statements are checked in the testing phase by collecting new empirical data in order to examine whether the relationships among the variables as predicted can be found in the new data obtained"*.

Figure 3.3 illustrates the process of generating new empirical material from the research questions. The last phase of the empirical cycle identified the results are interpreted. This phase generates new ideas for new hypothesis and research questions, which completes the cycle and returns to the first phase of the empirical cycle (Steenhuis & de Bruijn, 2006).

3.3 The scientific paradigm

Computer science is a branch of natural (empirical) sciences, on par with "astronomy, economics, and geology" (Eden, 2007). Many programs are unpredictable, the scientific paradigm seeks a posteriori (deductive reasoning) knowledge originating from the empirical evidence by conducting scientific experiments (Eden, 2007).

> As defined by Eden (2007), *"computer science is the study of the phenomena that surrounds computers which is an empirical discipline and experimental science"*.

The scientific paradigm tests hypothesis (claims) as experiments with programs tend to go beyond establishing just reliability. To discover empirical evidence through simulations and artificial intelligence, research method types should be controlled experiments and validity is very important (Eden, 2007). The deductive methods of theoretical computer science have been effective in theorizing, reasoning, constructing and even predicting. Computer science is indeed part of the branch of natural sciences as the methods include deductive and analytical methods of investigation (Eden, 2007).

According to Eden (2007:21), all programs are non-linear or self-modifiable, a priori knowledge (deductive reasoning) is unstainable, thus the methods of computer science must be combined with deductive reasoning and scientific experimentation.

3.3.1 Empirical research

Empirical research is used to gain knowledge by means of direct and indirect observation. Empirical evidence or observations can be analysed using quantitative research. When researchers use the quantitative research method they can answer empirical questions with the evidence collected usually called data.

3.3.2 From the view point of the causal relationship

The study used the concepts from the conceptual framework to convert the concepts into variables. The study used these variables to investigate the causal relationship between them. The study used three sets of variables for the study.

There is a classification of the causal model variables used for the study as shown in Table 3.1. The independent variable will affect or bring change in the dependent variable. The extraneous variable will not be measured in the study but will increase or decrease the strength of the relationship between the independent and dependent variable (Kumar, 2005).

Table 3.1: Casual model variables

Independent variable	Dependent variable	Extraneous variable
Data objects JAVA Programming language driver	Graph data stores, Column data stores, Key-value stores, Document stores Duration	Computer nodes Memory CPU cores Hard drive storage

3.3.4 From the viewpoint of the study design

Being a control experiment, the study manipulated the independent (cause) variable as seen in Table 3.2. The study's intervention focused on the different NoSQL data stores, using an experimental intervention, by measuring the write metrics, read metrics, update metrics and delete metrics per data store. The study focused on the four NoSQL categories namely; graph, document, key-value and column stores. The study was able to change and control the active variables during the study. The study couldn't change or control the attribute variables for the study, but could choose which graph, document, key-value and column store to choose.

Table 3.2: Active and attribute variables

Study intervention (active variables)	Study population/category (attribute variables)
Different NoSQL data stores per category, Java Programming language driver	Graph data store Document data store Key-value data store Column data store

3.3.5 From the viewpoint of the unit of measurement

The study measured the variables continuously as numeric values. The level of measurements for time was used as a quantifiable variable to measure the differences in ratios (ratio scale) to achieve objectives and accurate results (Kumar, 2005). The attributes of the time variable are differentiated by the degree of difference between them, there is absolute zero or a fixed starting point. The study could find the ratio between the attributes and measure time in nanoseconds, seconds, or minutes (Kumar, 2005).

3.4 Research design

The experimental research design is the plan and strategy to show how the study obtained answers to the research questions and problem statement. This is the study blueprint for how the research study should be done by operationalizing the variables so they can be measured. To select the sample of interest, to study and to collect the data to be used as the basis for answering the research questions and analysing the results (Kumar, 2005). The experimental research used laboratory experiments carried out in a specially created setting so the experimenter would be able to control the extraneous variables.

To determine the events during experimental design, researchers control and manipulate the conditions of an experiment. Intervention is presented to measure the difference the

experiment makes when changing the value of an independent variable. By changing the value of the independent variable, the study can determine the change and effect the independent variable has on the dependent variable.

3.5 The conceptual framework

The conceptual framework presents synthesis of the literature to explain the phenomenon. This maps the actions required during the duration of the research, given the previous knowledge of other researchers' points of view and highlights the observations on the subject of the research.

According to McGaghie et al. (2001:2) the conceptual framework sets a stage for the presentation of the research questions which then drives the investigation. Reports based on the problem statement, which explains why the problem statement of the thesis highlights the situation and concerns why the study decided to conduct the research study. The conceptual design framework in Figure 3.4, illustrates how the study generated the results based on using experiments from the experiment setup. The study looked at several data stores and installed each chosen data store on the cluster.

The third sub-objective of the study used the conceptual framework to implement and study the variables used in the study. NoSQL data stores would be used as inputs and can be changed based on the study's objective as seen illustrated by the arrows pointing inwards towards the CRUD operations. The use of API was to send and retrieve objects from the NoSQL data stores. The language drivers were also used as inputs to be studied using the CRUD operations. The nodes on the right side of the conceptual model were use to increase and decrease the resources needed when scaling the NoSQL data store across the nodes. Using the CRUD operations, the output results were generated based on the CRUD metrics to generate the performance metrics for each NoSQL data store category.

The data stores were each characterised as documents store, key-value store, column store and graph store. Each of the databases had different setup requirements based on their design principles. For the developer's approach to the post-setup of the cluster, the study installed JetBrains IntelliJ Integrated Development Environment (IDE) on a workstation to create the reactive three-tier application and used the JAVA Development Kit (JDK) and libraries for the programming language driver. The study used the IDE with many dependencies and libraries to select the best performing driver for JAVA programming language. The IDE allowed the study to convert the three-tier application into a container to be used within the kubernetes

cluster. This allowed the study to scale the application based on resources needed or to decrease the amount of instances.

The model had to be adjusted for the research aim and to stay focused on achieving just that. From the NoSQL family the study chose only one database per data store. For Document store the study chose Mongo DB. The study looked at the Key-value category and chose Redis DB for that category. For the Column store the study chose Cassandra DB and for the Graph data store the study selected Dgraph DB. The selection was based on the available drivers for the JAVA programming language. For each of the selected NoSQL categories the study could use a JAVA driver to interact with the databases.

The study created connections within the three-tier application to connect to all databases at the same time. The model used was using a domain model which would have the common data objects. In the experiment the study used an object called person, from the domain model the object was used to create several objects and increase the stages every time. The person object had to go through each layer of the application to generate the results from the domain model.

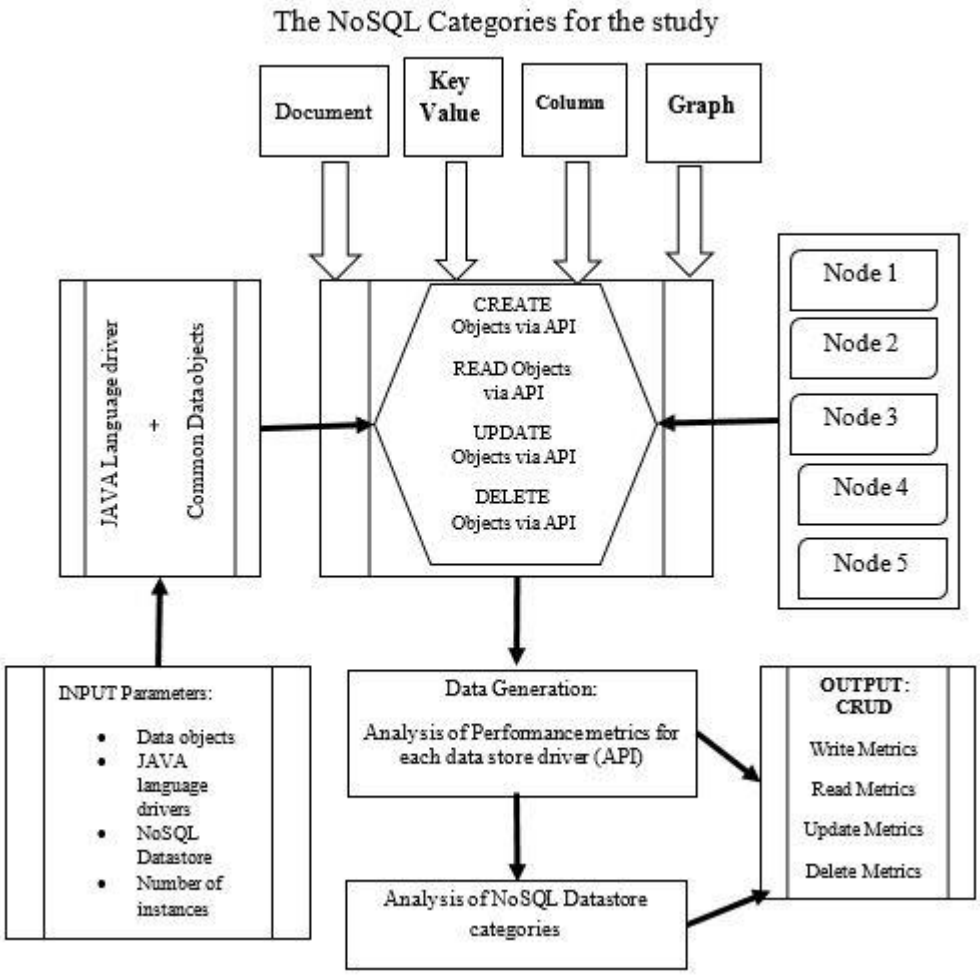


Figure 3.4: The conceptual framework

3.6 Data collection

The study used the duration of seconds as a unit of measurement to obtain the speed of write, read, update and delete of NoSQL data stores and the programming language driver. The study wanted to compare the effectiveness of the four NoSQL data stores on measuring their read, write, update and delete speed. The study used randomization to ensure compatibility by using the same JAVA language driver and the same data objects. The study recorded the observation on numeric scales in a tabular form.

3.7 Data analysis

The study classified information collected during data collection as raw data. The study needed to ensure that data was clean and free from inconsistencies and this is the first step of the analysis (Kumar, 2005). The data collected had to go through a process to transform the data into numeric values called codes, to allow the information to be easily analysed. The unit of

analysis would be the data obtained from the experiment. The population of interest was the measure of the duration (time) of the results obtained. The sampling technique applied, as mentioned above, would be random sampling, and this the sample strategy used. The rationale for using probability sampling is to generalise the results from the sample to the research population and to minimize sampling bias. By using simple random sampling (not haphazard), there would be an equal chance that each of the samples will be selected.

3.7.1 Coding

The data output during data collection was captured as JavaScript Object Notation (JSON) format. This was captured as JSON format with start, end, duration and objects as the variables.

The study converted the JSON format file to a comma-separated values (CSV) file which allowed data to be saved in a tabular format. The study used a free, browser-based JSON to CSV program to convert the file. The study used RStudio Integrated Development Environment (IDE) as the coding program to code the data directly from the CSV format tabular data to plot the graphs.

The next section the study used a Kubernetes cluster as part of the commodity cluster setup to achieve high availability and reliability of NoSQL data stores. Based on popularity Kubernetes became the standard way of deploying distributed applications. This platform allowed developers to scale applications to support heavier workloads and be a more robust system.

CHAPTER FOUR
RESEARCH FINDINGS AND DISCUSSIONS

In this section, the basic informational and formal assumptions made were discussed. To be clear about which assumptions may be driving the results. The results determined the changes in the parameters and by this, the study meant the following:

Not every research paper developed will contain a full theoretical model. If the researcher explains the assumptions, mechanisms and the results are clear, then a theoretical model could be redundant. The empirical model used in the research study, should be included in every paper.

It is always better to start with the simplest model for the research questions addressed. Think carefully about the continue vs the discrete time, be open to assumptions about how the results would be obtained.

To help illustrate the question(s) used during testing, the model should be simple. The form of the conceptual model should be appropriate to the research problem. During the experiment pre-phase, the study determined that the study target group needed special setup needs.

The study setup a data center, clustering the nodes for the experiment to gather the data needed. The study chose the Kubernetes (Production Ready Cluster) because it is an open-source system for automating deployment, scaling and management of containerized applications. The Kubernetes cluster clearly scaled the research instruments by adding multiple nodes to the cluster. The NoSQL databases were deployed as containers using Docker as the distributed services. During the next phase of the experiment, the study setup the client developer environment. The study used the common programming language driver JAVA to test the NoSQL data stores to measure the read metrics and write metrics.

There has been a lot of excitement around using containers and Docker as open-source systems for automating deployment. The study used kubernetes distributed system that allowed the packages and application to scale across the network. Applications such as the Kubernetes NoSQL data stores needed the setup constructed like this, so that they can handle the demands of performance, reliability and availability. This setup allowed developers to deploy framework of services needed to run the modern NoSQL data stores using Docker containers on the same set of shared computing resources.

4.1 Persistence technologies and drivers

As a datacenter operating system, Kubernetes is a distributed system, a cluster manager, a container platform, micro services platform. As a distributed system, Kubernetes includes a group of pods, services and disk volumes that are coordinated by a group of containers. This forms part of the experiment. The data stored on the disk volumes should be persistent even after the pod or service is removed and should be non-volatile storage.

4.2 High availability and reliable architectures setup

Kubernetes are defined as a set of building blocks, which collectively provides deployments and scaling applications based on their CPU, memory and disk space requirements. Thus, Kubernetes is loosely coupled in order to meet different workloads. Kubernetes makes use of a Kubernetes API which the internal components used to communicate to the compute and storage resources which can be defined as objects.

The basic scheduling unit Kubernetes uses is called a pod, a pod consists of one or more containers that share resources. Every pod within the Kubernetes cluster gets assigned a unique pod IP address which allows the applications to use ports without conflicts. Applications developers should never use the pod IP address, instead they should use the service name. A pod can also be a volume of a disk.

Kubernetes make use of services which is a set of pods that work together such as the three-tier application we had setup for this experiment. Kubernetes makes use of a Domain Name Service (DNS) to assign to the services and load balances the traffic in a round-robin manner, so if the pods experience any failures on machines or nodes, they can move to another node.

The file systems used in Kubernetes volumes provides persistent storage that stays present as the pod exists. The storage is used as shared disk storage space for the containers that live inside the pod. Namespaces are called non-overlapping sets because that provides the ability to partition resources it manages, used for environments that’s spread across multiple teams and projects. Please see below for a visual representation of how the Kubernetes cluster works in Figure 4.1 below.

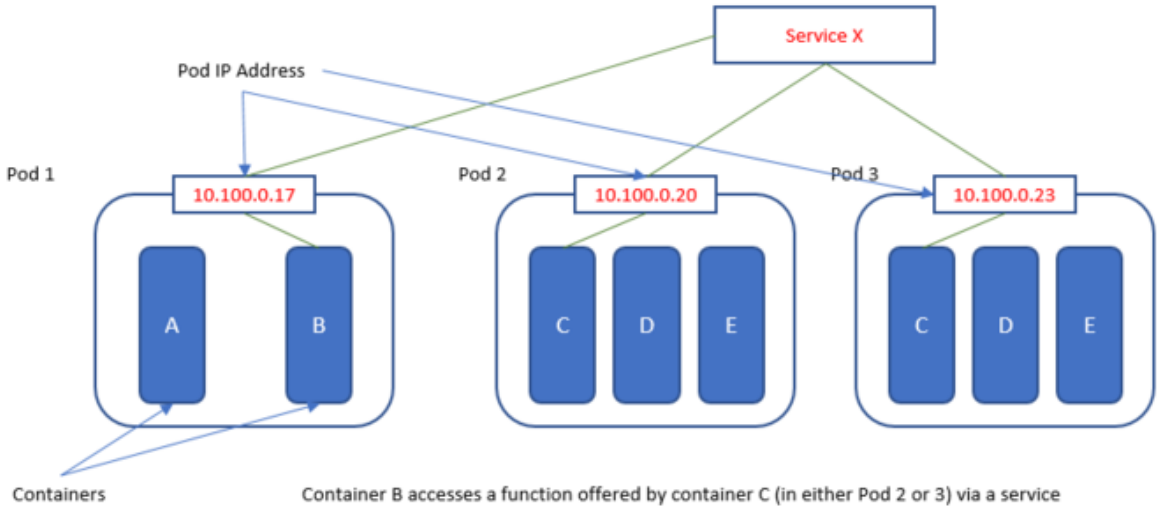


Figure 4.1 Simplified view showing how services interact with pod networking in a Kubernetes cluster

4.2.1 Requirements

For the study to achieve the experimental design, the study needed to focus on commodity clusters to present a consistent peer to peer view, this forms part of research question. Below we can see in Figure 4.2 how data gets distributed amongst the nodes, the same data stored on Node1 are stored on Node4 and the data gets evenly spread across the nodes, If Node1 crashes the data still remains on Node4.

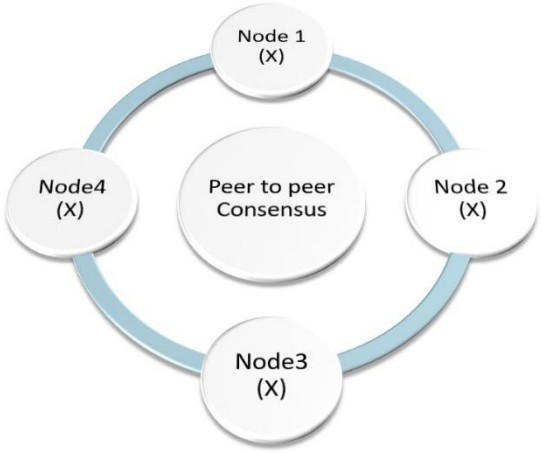


Figure 4.2: This figure illustrates how the data(X) gets distributed amongst the server nodes

Requirements for the commodity cluster:

- The target servers must have access to the Internet in order to pull Docker images, otherwise additional configuration is required
- The target servers are configured to allow IPv4 forwarding.
- Your ssh key must be copied to all the servers as part of your inventory.
- The firewalls are not managed; you'll need to implement your own rules the way you used to. In order to avoid any issue during deployment you should disable your firewall.
- If kubespray is run from a non-root user account, the correct privilege escalation method should be configured in the target servers.

Before you begin this study make sure you have the following setup:

- One or more machines running:
    - Ubuntu 16.04+
    - 2 GB or more of RAM per machine
    - 2 CPUs or more
    - Full network connectivity between all machines in the cluster (public or private network is fine)
    - Unique hostname, MAC address
    - Swap disabled. You **MUST** disable swap in order for the kubelet to work properly

It is very likely that hardware devices will have unique addresses, although some virtual machines may have identical values. Kubernetes uses these values to uniquely identify the nodes in the cluster. If these values are not unique to each node, the installation process may fail (Baier, 2017).

Table 4.1: Master node required ports

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	6443*	Kubernetes API server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10251	kube-scheduler	Self
TCP	Inbound	10252	kube-controller-manager	Self

Table 4.2: Worker nodes required ports

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	30000-32767	NodePort Services**	All

Table 4.3: Runtimes for kubernetes

Runtime	Domain Socket
Docker	/var/run/docker.sock
containerd	/run/containerd/containerd.sock
CRI-O	/var/run/crio/crio.sock

If both Docker and containerd are detected together, Docker takes precedence. This is needed, because Docker 18.09 ships with containerd and both are detectable. If any other two or more runtimes are detected, kubeadm will exit with an appropriate error message (Baier, 2017).

Refer to the CRI installation instructions for more information.
Installing kubeadm, kubelet and kubectl
You will install these packages on all of your machines:

- Kubeadm: the command to bootstrap the cluster.
- Kubelet: the component that runs on all of the machines in your cluster and does things like starting pods and containers.
- Kubectl: the command line utility to talk to your cluster.

Kubeadm will not install or manage kubelet or kubectl for you, so you will need to ensure they match the version of the Kubernetes control plane you want kubeadm to install for you. If you do not, there is a risk of a version skew occurring that can lead to unexpected, buggy behaviour (Baier, 2017).

4.2.2 Network plugins

You can choose between 6 network plugins. (Default: calico, except Vagrant uses flannel)

- flannel: gre/vxlan (layer 2) networking.
- calico: bgp (layer 3) networking.
- canal: a composition of calico and flannel plugins.
- cilium: layer 3/4 networking (as well as layer 7 to protect and secure application protocols), supports dynamic insertion of BPF bytecode into the Linux kernel to implement security services, networking and visibility logic.
- contiv: supports vlan, vxlan, bgp and Cisco SDN networking. This plugin is able to apply firewall policies, segregate containers in multiple network and bridging pods onto physical networks.
- weave: Weave is a lightweight container overlay network that doesn't require an external K/V database cluster. (Please refer to weave troubleshooting documentation).

- kube-router: Kube-router is a L3 CNI for Kubernetes networking, aiming to provide operational simplicity and high performance: it uses IPVS to provide Kube Services Proxy (if setup to replace kube-proxy), iptables for network policies, and BGP for ods L3 networking (with optionally BGP peering with out-of-cluster BGP peers). It can also optionally advertise routes to Kubernetes cluster Pods CIDRs, ClusterIPs, ExternalIPs and LoadBalancerIPs.
- multus: Multus is a meta CNI plugin that provides multiple network interface support to pods. For each interface Multus delegates CNI calls to secondary CNI plugins such as Calico, macvlan, etc.
- The choice is defined with the variable kube_network_plugin. There is also an option to leverage built-in cloud provider networking instead. See also Network checker.

4.2.3 Creating highly available clusters with kubeadm

For both methods you need this infrastructure:

- Three machines that meet kubeadm's minimum requirements for the masters
- Three machines that meet kubeadm's minimum requirements for the workers
- Full network connectivity between all machines in the cluster (public or private network)
- sudo privileges on all machines
- SSH access from one device to all nodes in the system
- Kubeadm and kubelet installed on all machines. Kubectl is optional.
- For the external etcd cluster only, you also need:
- Three additional machines for etcd members

1. Create a kube-apiserver load balancer with a name that resolves to DNS.
    - In a cloud environment you should place your control plane nodes behind a TCP forwarding load balancer. This load balancer distributes traffic to all healthy control plane nodes in its target list. The health check for an apiserver is a TCP check on the port the kube-apiserver listens on (default value **:6443**).
    - It is not recommended to use an IP address directly in a cloud environment.
    - The load balancer must be able to communicate with all control plane nodes on the apiserver port. It must also allow incoming traffic on its listening port.
    - HAProxy can be used as a load balancer.
    - Make sure the address of the load balancer always matches the address of kubeadm's **ControlPlaneEndpoint**.
    - Add the first control plane nodes to the load balancer and test the connection:

A connection refused error is expected because the apiserver is not yet running. A timeout however, means the load balancer cannot communicate with the control plane node. If a timeout occurs, reconfigure the load balancer to communicate with the control plane node. Add the remaining control plane nodes to the load balancer target group (Baier, 2017).

Steps for the first control plane node:

1. On the first control plane node, create a configuration file called **kubeadm-config.yaml**: **controlPlaneEndpoint** should match the address or DNS and port of the load balancer. It's recommended that the versions of kubeadm, kubelet, kubectl and Kubernetes match. Initialize the control plane sudo kubeadm init --config=kubeadm-config.yaml --experimental-upload-certs. The **--experimental-upload-certs** flag is used to upload the certificates that should be shared across all the control-plane instances to the cluster. If instead, you prefer to copy certs across control-plane nodes manually or using automation tools, please remove this flag and refer to the Manual certificate distribution section bellow.

Kubeadm join ip address: 6443 --token --discovery-token-ca-cert-hash.
Please note that the certificate-key gives access to cluster sensitive data, keep it secret!
As a safeguard, uploaded-certs will be deleted in two hours; if necessary, you can use kubeadm init phase upload-certs to reload certs afterward.

You can then join any number of worker nodes by running the following on each as root:
Kubeadm join ip address: 6443 --token --discovery-token-ca-cert-hash.

4.2.4 Configure the cluster

For the experiment the study setup the cluster as shown below. The study setup five nodes for the Kubernetes cluster using inexpensive commodity hardware and two nodes for the Gluster file system which is a free open source software scalable network file system. The Kubernetes cluster used the Gluster file system if the study needed to scale large files on the cluster, see Figure 4.3.

The Kubernetes cluster nodes:

- Node 1 – 8 CPUs, 16 GB memory
- Node 2 – 8 CPUs, 16 GB memory
- Node 3 – 4 CPUs, 32 GB memory
- Node 4 – 4 CPUs, 32 GB memory

- Node 5 – 4 CPUs, 32 GB memory

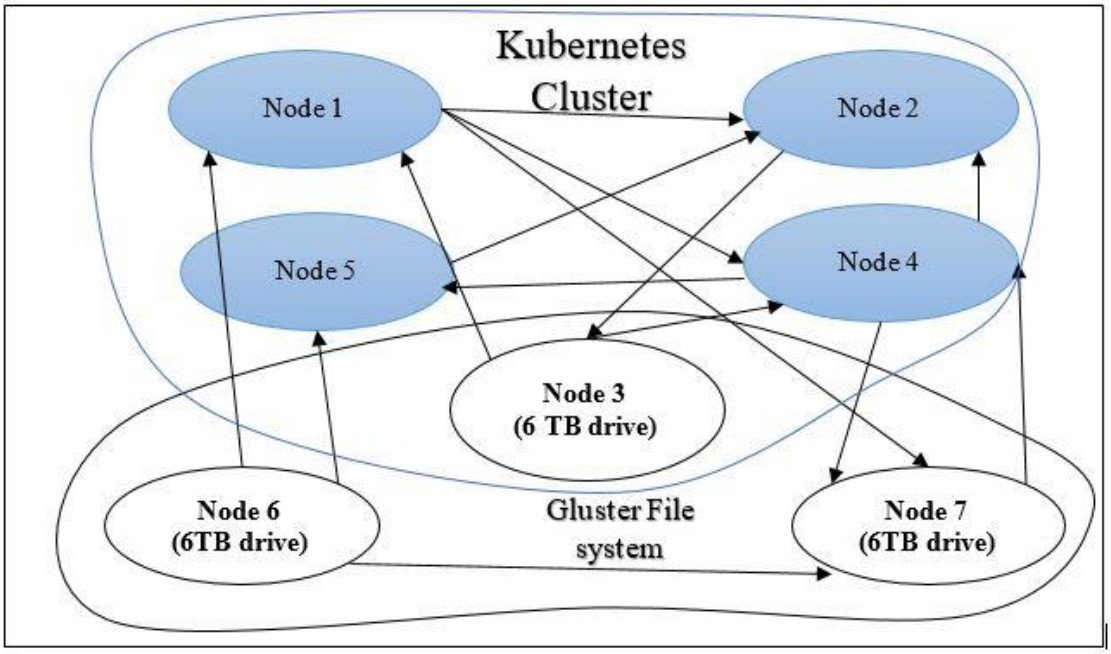


Figure 4.3: Kube Cluster and Gluster FS

4.2.5 Installing kubeadm, kubelet and kubectl

As shown below in Figure 4.4, this starts the process of installing the three packages on the nodes. For the experiment we used the Ubuntu Linux operating system.

Ubuntu, Debian or HypriotOS | CentOS, RHEL or Fedora | Container Linux

```
apt-get update && apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y kubelet kubeadm kubectl
apt-mark hold kubelet kubeadm kubectl
```

Figure 4.4: Kubeadm, kubelet and kubectl install

After kubeadm was installed, the study run **apt-get update && apt-get upgrade** or **yum update** in the terminal to get the latest version of kubeadm. The kubelet restarted every few seconds as it waited in a crash loop for kubeadm to tell it what to do. This crash loop is expected and normal. After initializing the master node, the kubelet runs normally.

4.2.6 Initializing master node

The Master Node is the machine where the control plane components run, including etcd (the cluster database) and the API server (which the kubectl CLI communicates with).

The study had to choose a pod network add-on, and verify whether it required any arguments to be passed to kubeadm for initialization, depending on which third-party provider set the --pod-network-cidr to a provider-specific value. To use different container runtime or if there is more than one installed on the provisioned node, specify the --cri-socket argument to kubeadm init. (Optional) Unless otherwise specified, kubeadm uses the network interface associated with the default gateway to advertise the master's IP. To use a different network interface, specify the --apiserver-advertise-address=<ip-address> argument to kubeadm init.

4.2.7 Accessing Kubernetes Dashboard

IMPORTANT: HTTPS endpoints are only available if the Recommended Setup, followed Getting Started guide to deploy Dashboard or manually provided --tls-key-file and --tls-cert-fileflags (Baier, 2017).

4.2.8 Kubectl proxy

The Kubectl created a proxy server between the nodes and the Kubernetes API server, this should only be accessible locally by default for security reasons.

As illustrated below in Figure 4.5, the screenshot of the nodes showing that they all have a healthy status can only be viewed with the Kubernetes-dashboard.

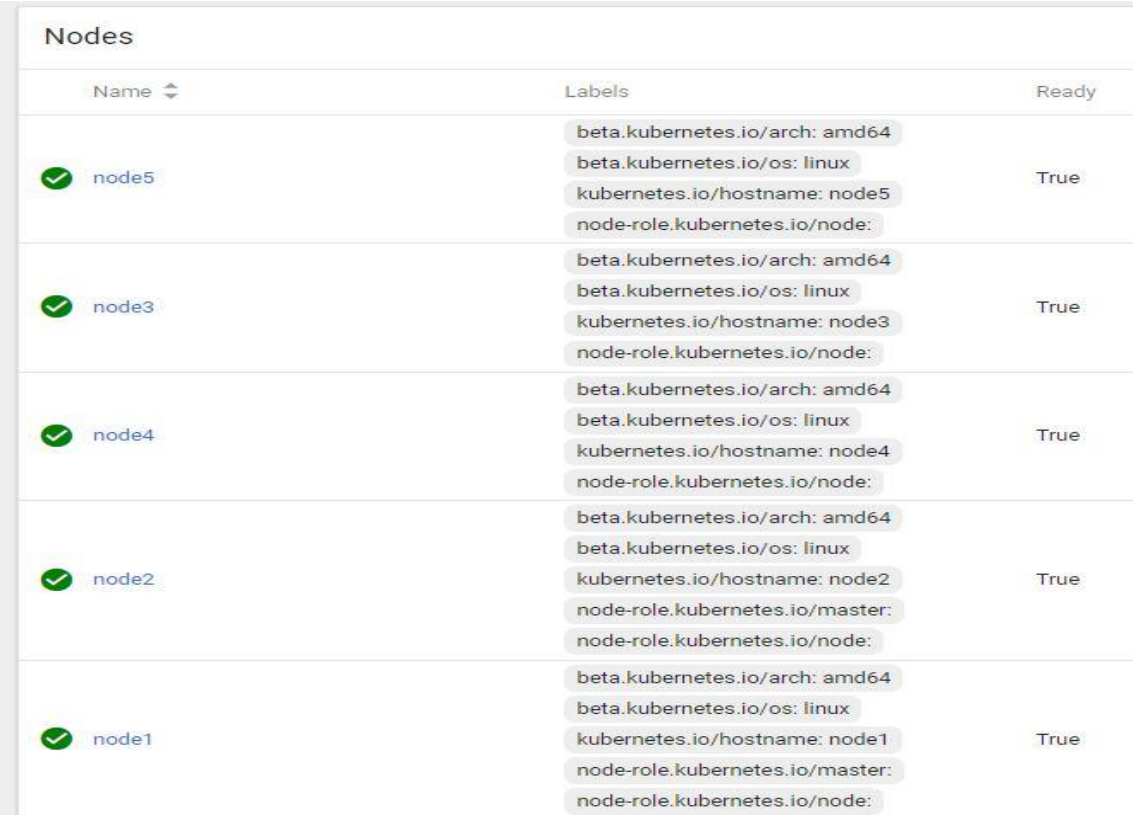


Figure 4.5: Kubernetes nodes

To build the three-tier application, the study used the reactive approach based on the Reactive Manifesto, this provided security against failures within the tree layers. This was the asynchronous message-passing style, where there would be constant communication between the three layers. This also provided less system overload and only consumed resources while active. The application adapted to the changes as we increased or decreased the input load and we didn't experience any bottlenecks. This provided a cost-effective way to run the application on commodity hardware. The study could easily change the repository packages data stores if needed to recover a service that failed in the upper layer without stopping the presentation layer. Please see Figure 4.6 for illustration of the package domain model.

The presentation layer was responsible to create the sessions between the application and the NoSQL databases, while the service layer created the amount of objects from the presentation layer. The repository layer stored the objects within the different databases as needed.

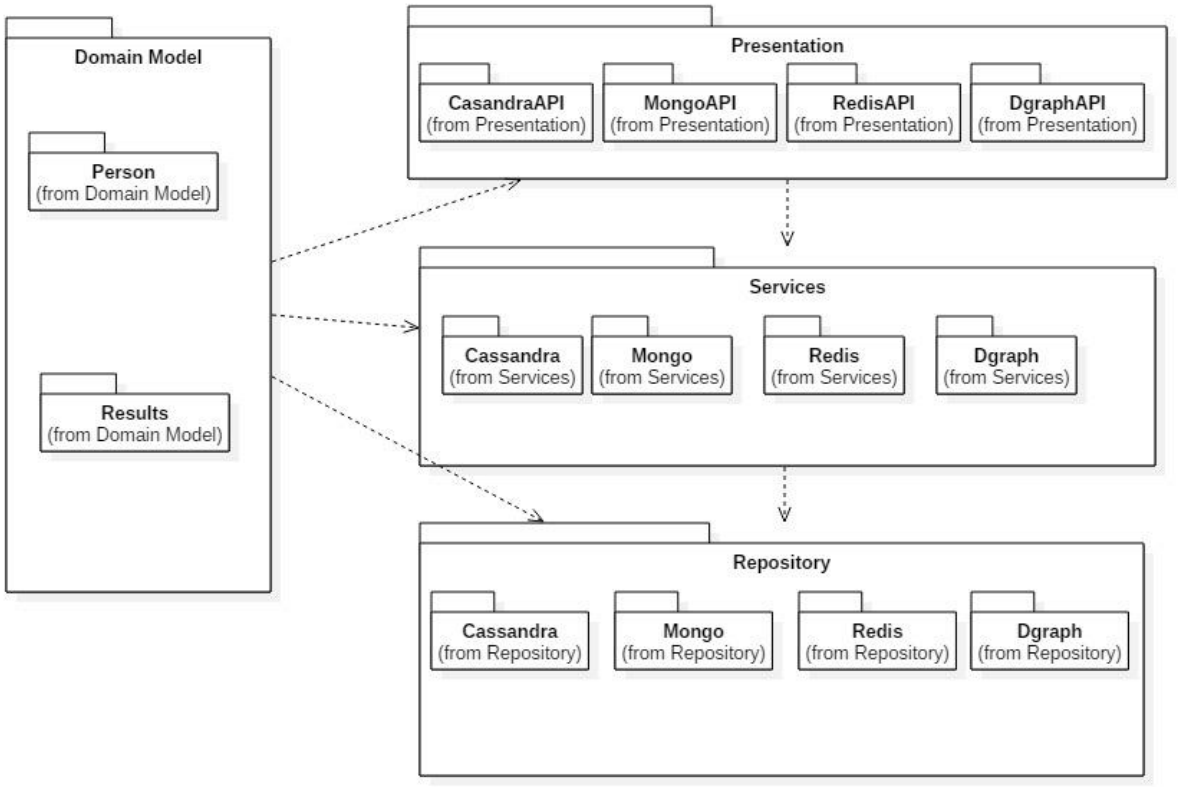


Figure 4.6: Packages domain model

As illustrated in Figure 4.6, the study developed the application in layers using Java programming language and then used Docker (docker.com), Docker file and Maven (maven.apache.org) plugins to create a JAVA Archive (JAR) file. This JAR file was used to package and compress the JAVA class files and resources into one file for distribution to build a Docker container image. The Docker image when deployed was uploaded on Docker hub (hub.docker.com). This allowed to import the compressed three-tier application to be pulled down from Docker hub into the cluster environment Kubernetes. Using the Kubernetes environment, the study used a Yaml file. YAML stands for "YAML Ain't Markup Language," this allowed the study to configure the application design needs and requirements.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: waldon-deployment
  labels:
    app: waldon
spec:
  replicas: 1
  selector:
    matchLabels:
      app: waldon
  template:
    metadata:
      labels:
        app: waldon
    spec:
      containers:
      - name: waldon
        image: waldonhendricks/research:latest
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: waldon-service
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 8080
    nodePort: 30010
  selector:
     app: waldon
```

Figure 4.7: YAML config file of JAVA app

This could be easily adjusted based on the application needs. Under “spec” the study specified the Docker hub link where the image was kept and pulled from. The Docker hub created a container “https://hub.docker.com/r/waldonhendricks/research” for the image to be pulled from Docker hub.

A similar approach was done for the NoSQL data stores, but for this the study used the YAML file to build the NoSQL data stores and pull them from Docker hub repositories into the Kubernetes cluster environment. The study achieved elasticity by scaling the NoSQL data stores across the Kubernetes nodes and could replicate the data stores across the nodes.

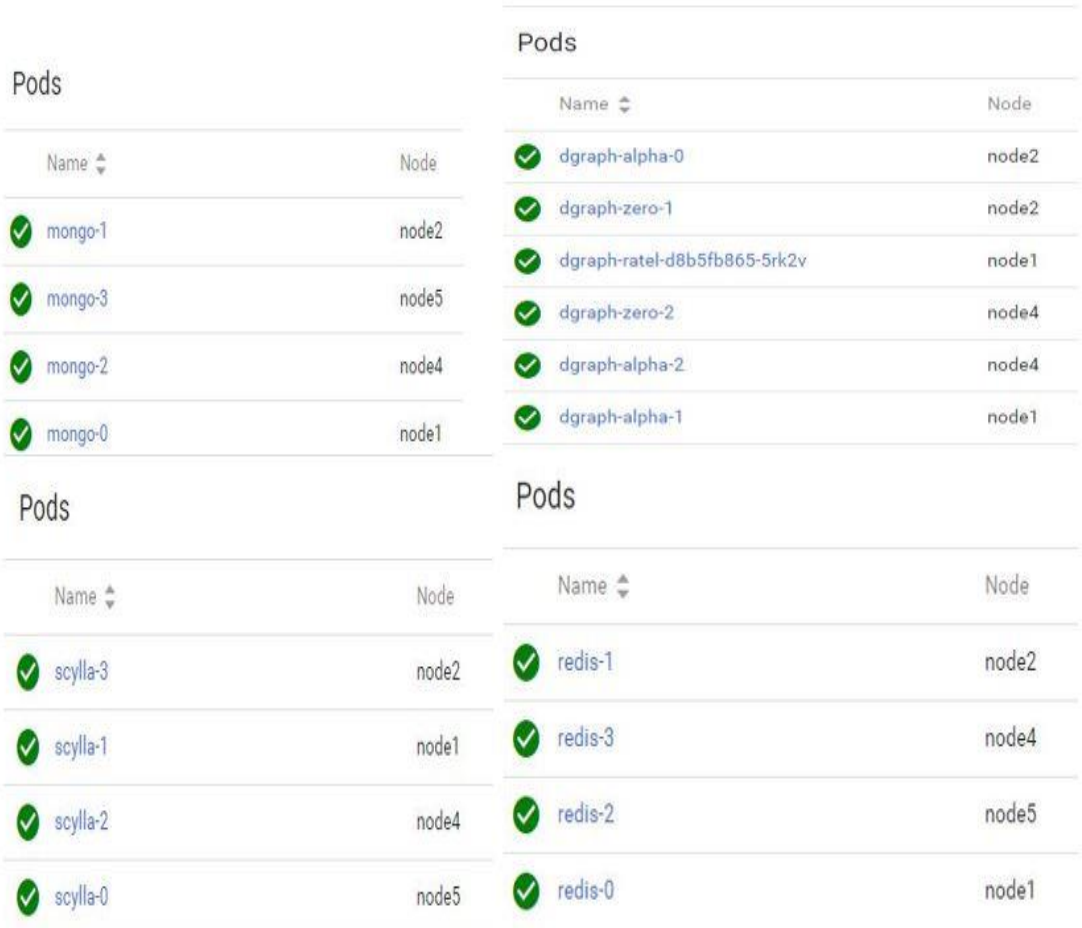


Figure 4.8: NoSQL Mongo, Cassandra, Redis, Dgraph kubernetes pods

In the next section the study discussed the results of the different test cases made and how CRUD operations performed with each NoSQL data store and the JAVA programming driver that created the objects using the package domain model discussed above.

4.3 Read, write, update and relete results

The study selected the four data stores to do the create tests using the API endpoint to connect the data stores. The tests determined which data store would be chosen as the CQRS approach design for the write-model. The results from the create phase used each data store and created a session via the JAVA API driver. The study used the API endpoint sending 50 objects to the NoSQL data stores from the IDE client workstation and incremented the objects each time by 50 objects. The goal was to send and receive 3000 objects by incrementing the previous results by 50 objects than 100 objects to see the effect on the JAVA driver used per NoSQL category.

4.3.1 Create esults

For the Dgraph test the study created 50 records, then 100, then 150 and so forth, each time

incrementing the objects by 50 objects. After creating 50 objects the study captured the duration results and then deleted the objects created each time and calculated the duration per 50 objects on every test made. The first 50 objects created were 0.221 seconds and at 1000 objects the duration to create was 2.709 seconds. The study then increased the incremented total by 100 objects each time at 1000 objects created, the total duration until 3000 objects was 7.833 seconds (see Figure 4.9).

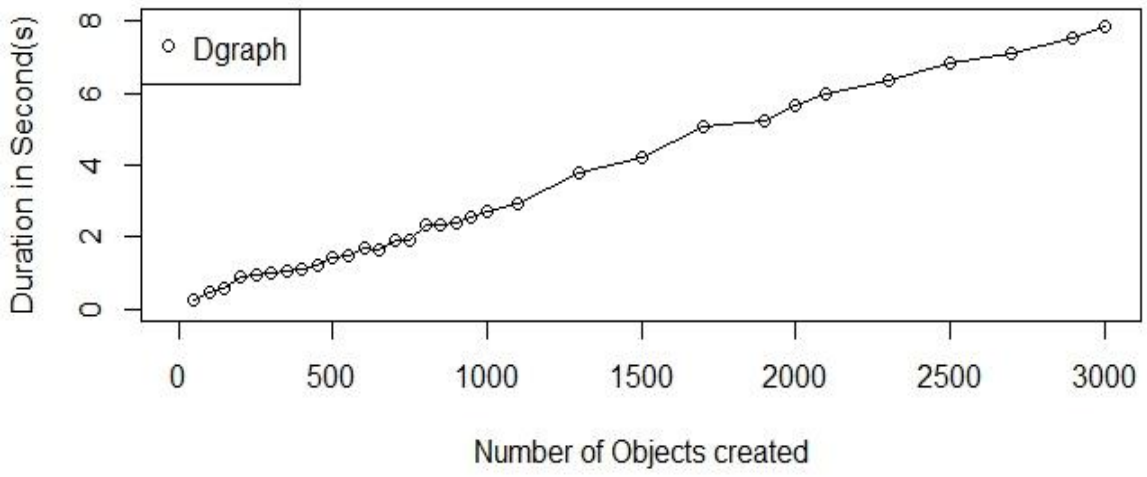


Figure 4.9: Dgraph create test 3000 objects

The Mongo API endpoint with the java driver created 50 objects. The study incremented each test by 50 objects, and experienced a slow response from the client each time increasing the objects. The Mongo API endpoint created more connections each time the study created objects. For 100 objects, 100 connections were created and for 150 objects, 150 connections. The study noticed that as the studies increment the objects, the application consumed more memory usage as Mongo was an in-memory data store. The JAVA client consumed more memory and the applications of the host operating system started to respond slower and slower. The study created 50 objects, then incremented by 50 to reach 100, then 150 and finally reached 700 objects. The workstation had no more memory left and had to be restarted to free up more memory. When the study created 50 objects the duration was 0.671 seconds, at 700 objects created the duration was 9.437 seconds (see Figure 4.10).

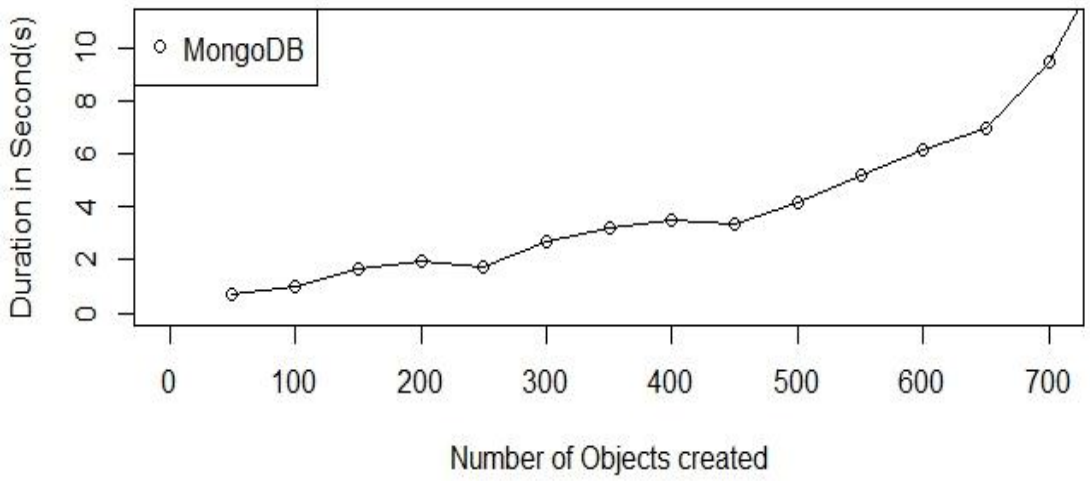


Figure 4.10: MongoDB write 700 objects

The study used the datastax Cassandra API driver and started with one keyspace creating 50 objects, then incrementing the 50 objects by 50 more with each repetition test. The Cassandra data store performed well. The study completed several tests on Cassandra using the create API endpoint and could successfully reach the goal. Of 3000 objects created the duration for 3000 objects was 9.247 seconds. After creating 50 objects the duration was 0.188 seconds and at 1000 objects by incrementing each test by 50 each time the duration was 3.122 seconds (see Figure 4.11).

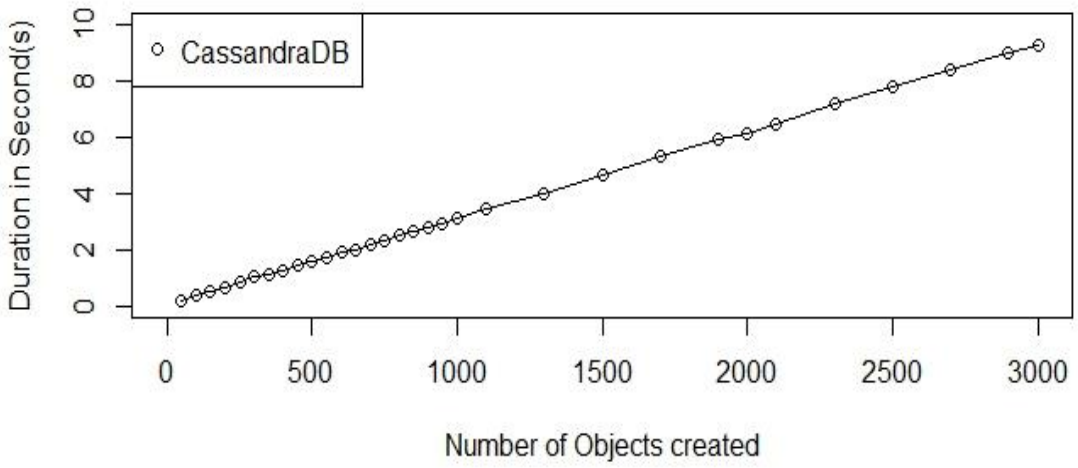


Figure 4.11: Cassandra create 3000 objects

The study used the Jedis JAVA driver to connect to the Redis DB. Created 50 keys or objects from the client to the databases. The connections showed 50 clients connected. The study increased the keys from 50 to 100 keys and noticed 151 clients were connected to the database. The study then continued incrementing the keys by 50 keys each time. And at this moment noticed the client connections multiplied by two. For 150 keys, 301 clients were connected. The study created 300 keys but had 1051 clients socket connections connected. The study flushed the DB each time the study created keys. The study incremented the list of objects, but noticed that the client closed connection at a certain socket port number value and then restarted the connections by 150 connections. The study experienced a connection reset at 600 keys and couldn't allocate more resources for the client.

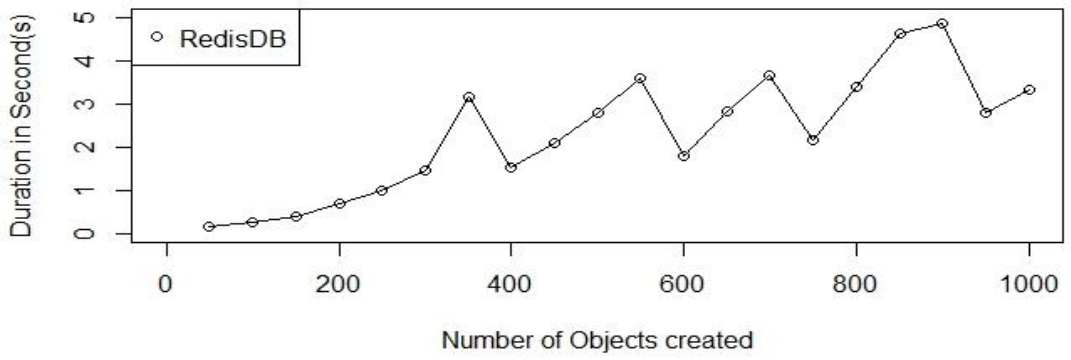


Figure 4.12: RedisDB create 950 objects

When investigating the object creation, the study noticed that if it restarts the Redis DB Server to close the client connections, the client could create 900 objects all at once. The study created 950 objects, but the database closed connections to the client, as no socket connections were available. The study also noticed that when it restarted the DB server the object creation took slightly faster to create than the previous attempt, as the DB server had 0 connections before the study created those objects. The study tried creating 1000 objects, but had an unexpected error (type=Internal Server Error, status=500). java.net. SocketTimeoutException: read timed out and only 816 objects were created. The duration was 0.16 seconds to create 50 objects, but with 950 objects the client duration time was 2.796 seconds to create (see Figure 4.12).

The Cassandra DB object creation did not put pressure on the JAVA client. Cassandra DB increased slowly as the study added more objects to the test. MongoDB took the longest duration in seconds. The Redis DB started well with fewer objects but as the study started increasing the objects, at 350 objects the Redis DB closed socket connections and the client

connections restarted and also the duration to create objects increased. Dgraph DB performed well as the study incremented the objects by 50 each time and at 200 objects Dgraph was slightly higher in duration of objects created than RedisDB and Cassandra DB. But at 300 objects Dgraph DB stayed steady as the study incremented the amount of objects by 50 each time and performed better when reaching 500 objects than Cassandra DB (see Figure 4.13).

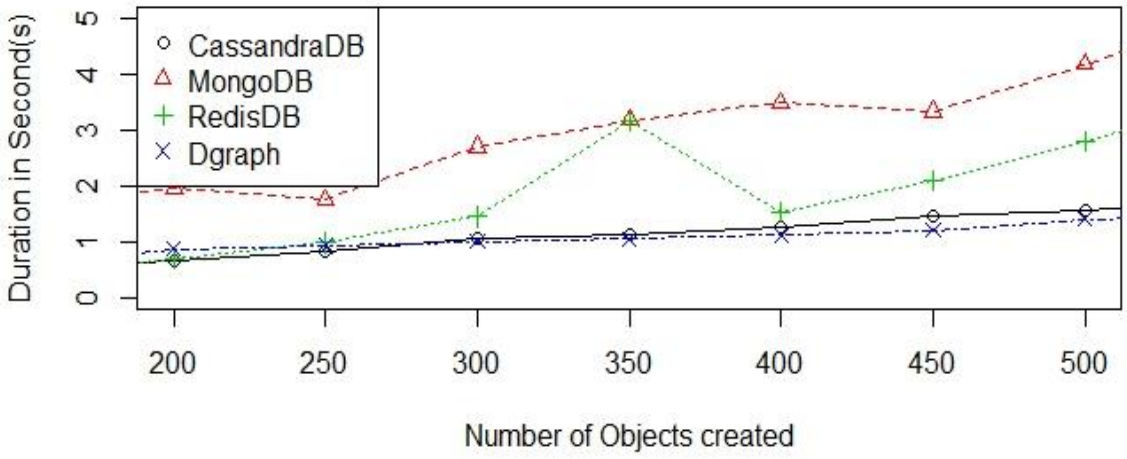


Figure 4.13: Create-results all DBs 200-500 objects created

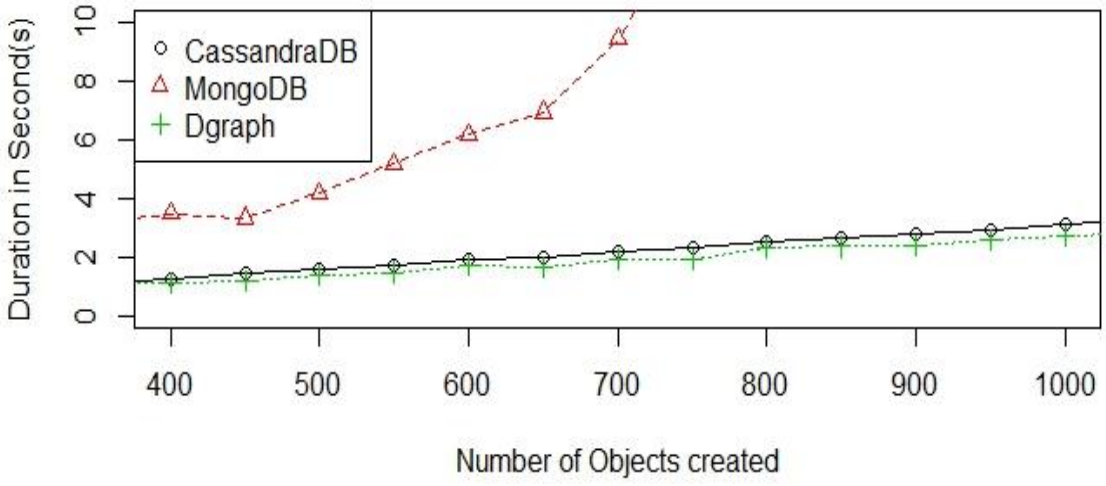


Figure 4.14: Create-results Cassandra, MongoDB and Dgraph

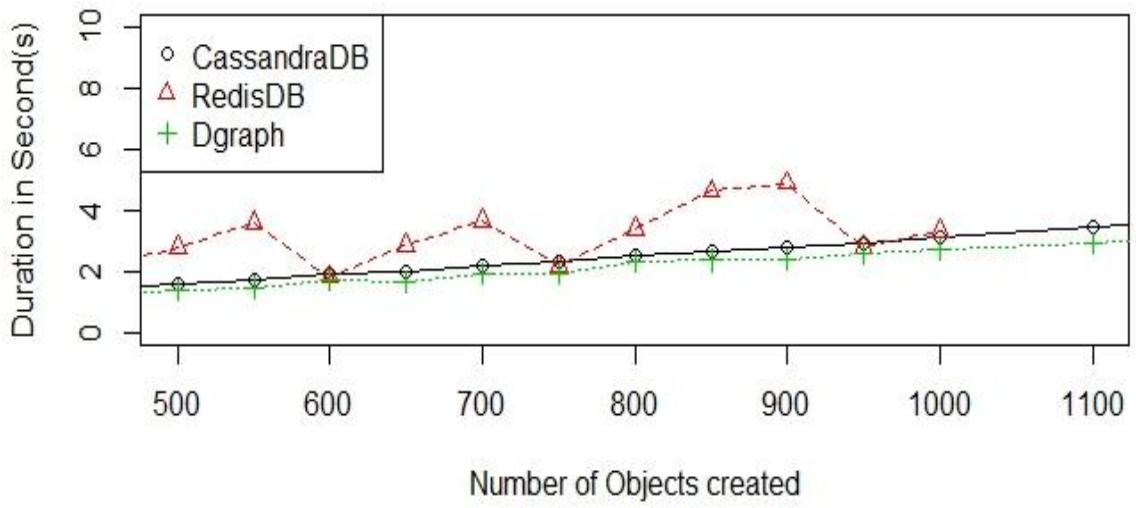


Figure 4.15: Create-results Cassandra and Dgraph and RedisDB

Cassandra and Dgraph had the lowest duration when objects were created and increased by each test run. Mongo took the longest duration and the drop in the duration by 700 objects where the client IDE had to be restarted. Redis failed at 600 at first and after a connection reset it had only reached 1000 objects (see Figure 4.14 and Figure 4.15).

4.3.2 Read results

The study discarded Dgraph DB for the read phase, as the dropout for Dgraph not to be selected for further tests. The selection was to choose the other data stores and do tests on the read API endpoint to determine the best selected data store for the read-model. For this phase the study created the objects and then read the objects created, then deleted the objects again and recreated the next set of tests and this was an iterative process.

The study used the Jedis client API driver to measure the read duration of the database. Started with 50 keys or objects and incremented the tests by 50 each time, then the JAVA driver gave a java.net. Socket Exception: Connection reset at 450 keys with 2486 client socket connections. The study read 50 objects and the duration was 0.207 seconds. Read 350 objects the duration was 3.418 seconds and after this test the socket connections closed at 350 objects (see Figure 4.16). The drop after 350 objects was the client connections that restarted due to the socket connections that was closed by the client to the Redis DB.

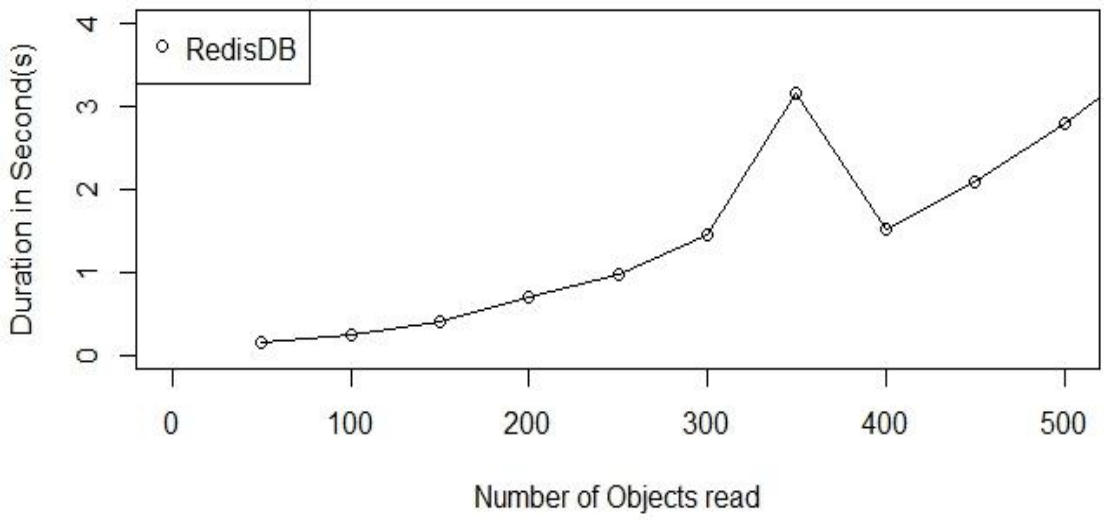


Figure 4.16: Redis read 400 objects

Cassandra DB performed with no problems. The study read 50 objects and incremented the objects by 50 with a duration of 0.08 seconds. Reached 1000 objects with duration 0.028 seconds to read 1000 objects (see Figure 4.17). Based on this tests phase the study had to create 10 objects, read the 50 objects, get the read metric then delete the 50 objects again and so that was also an iterative process. With the Mongo DB, the study started with 50 objects and incremented by 50 each time the study completed a test. The study could only create and read 800 objects as the client ran out of memory resources available. Read 50 objects which was 0.007 seconds, but as the study increased the read objects by incrementing by 50 each time per test, the study consumed more memory from the host. At 800 objects read the duration was 0.011 seconds, but couldn't continue as the study didn't have any memory left as seen in Figure 4.17 below.

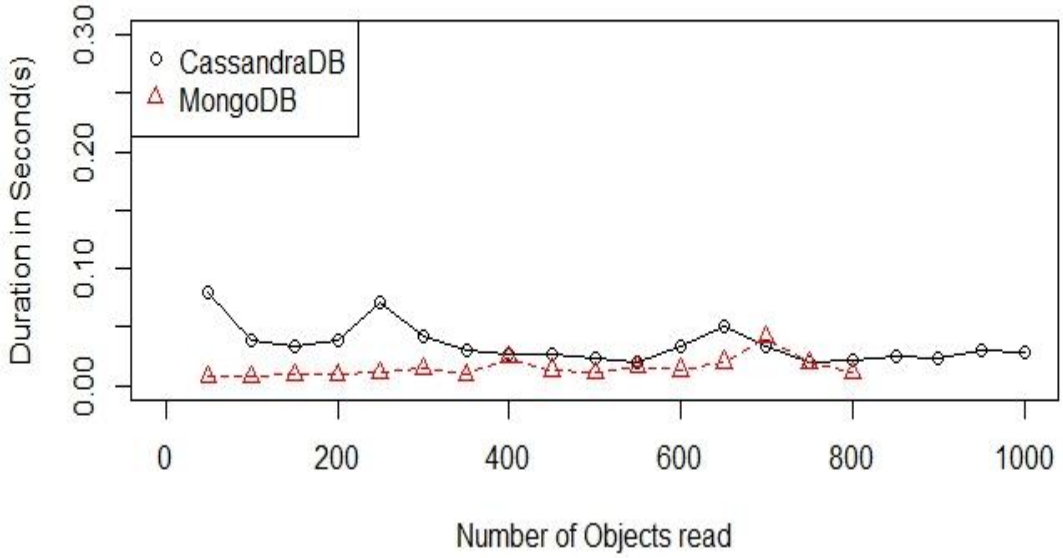


Figure 4.17: CassandraDB and MongoDB read of 1000 objects

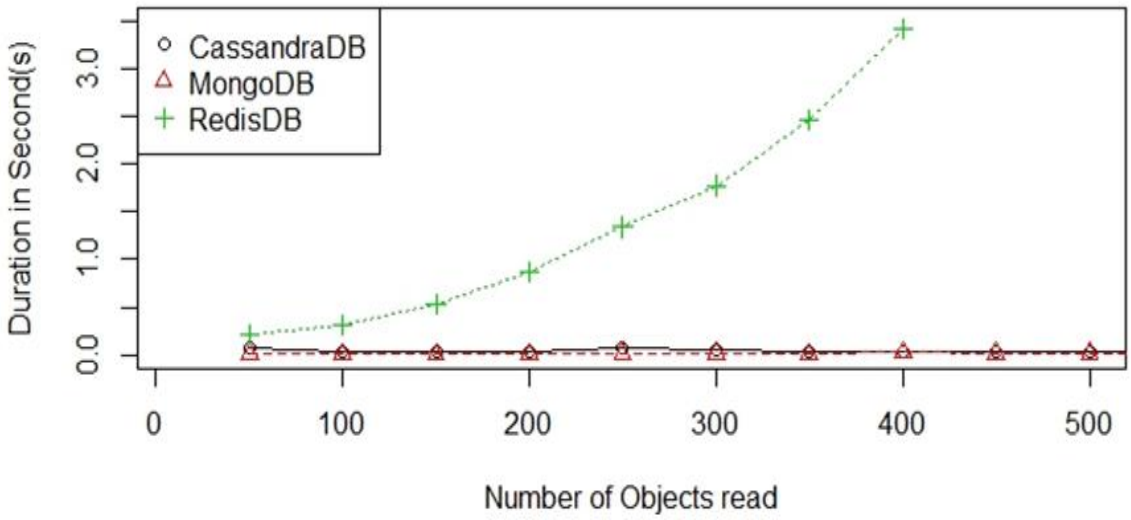


Figure 4.18: Create 400 objects results

Figure 4.18 above shows that between 50 objects and 400 objects read, Mongo DB and Cassandra DB read the fastest objects. Redis DB could only reach 400 objects, then closed socket connections as too many connections were created. At this point the study considered

RedisDB as a dropout of the experiment for the read metrics captured.

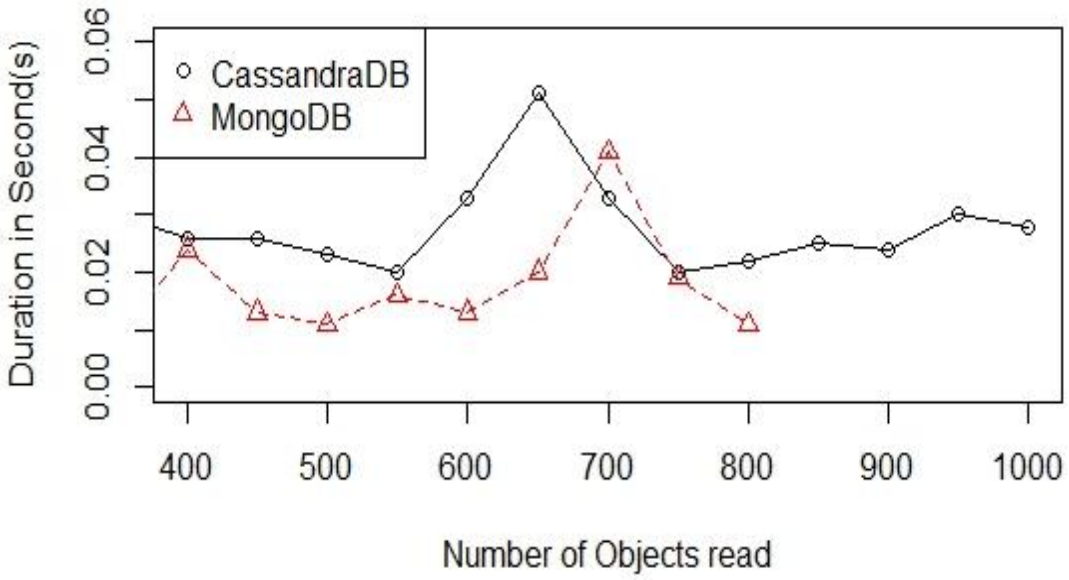


Figure 4.19: Read-results CassandraDB and MongoDB using IDE client

In Figure 4.18 above RedisDB took the longest duration to read the objects. MongoDB ran out of memory by 800 objects on the IDE client, but Cassandra was the only data store out of the three data stores to reach 1000 objects with 0.028 seconds (see Figure 4.19).

4.3.3 Update results

The study wanted to achieve the aim and complete the CRUD operations, and continued with the three data stores to see which database would update objects by adding data to the already created object. Simply the study wanted to create 50 objects, then update the objects by adding text “update” next to the object already created.

The study used the Jedis driver API and created 50 objects, the study performed an update on the objects and noticed the objects created more client connections as the study updated the objects each time by 50 objects. The study could only update 350 objects and by this time had 2494 clients connected and got a java.net. Socket Exception: Connection reset error. To update 50 objects the duration was 0.811 seconds, but could only update 300 objects with a duration of 5.931 seconds, see below in Figure 4.20.

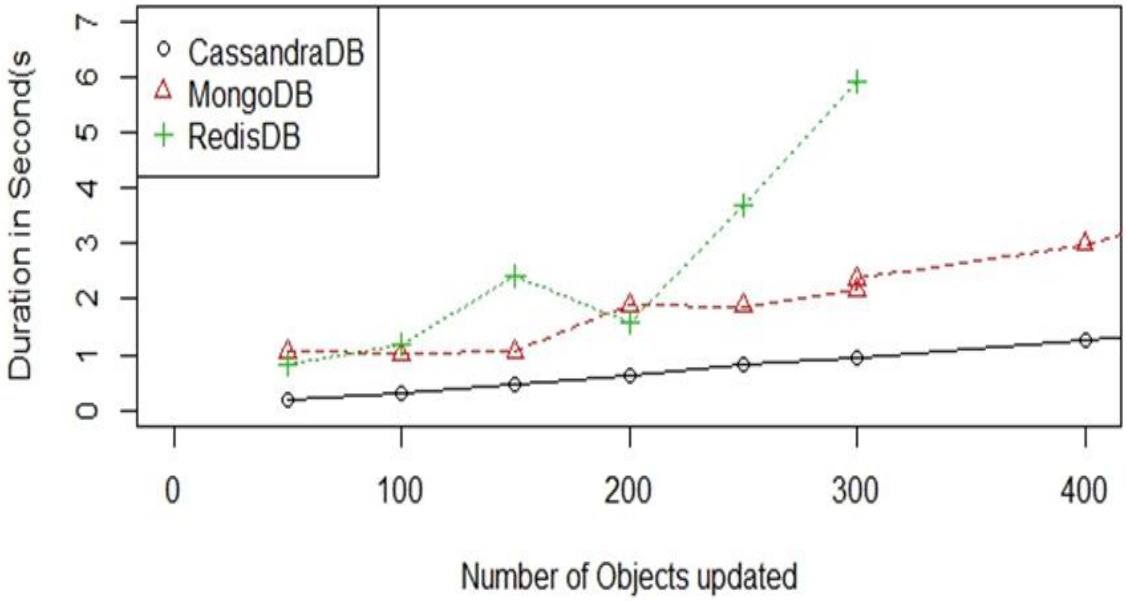


Figure 4.20: RedisDB, MongoDB and CassandraDB update results

The study had no errors with the update API using the datastax Cassandra driver and started from 50 and incremented by 50 each time and read and update objects till 3000 objects could be updated. At 50 objects updated the study had reached a duration of 0.18 seconds to update. At 300 objects updated the duration was 0.942 seconds and continued to update 800 objects with a duration of 2.478 seconds. The study reached 3000 objects updated within 9.326 seconds (see Figure 4.21).

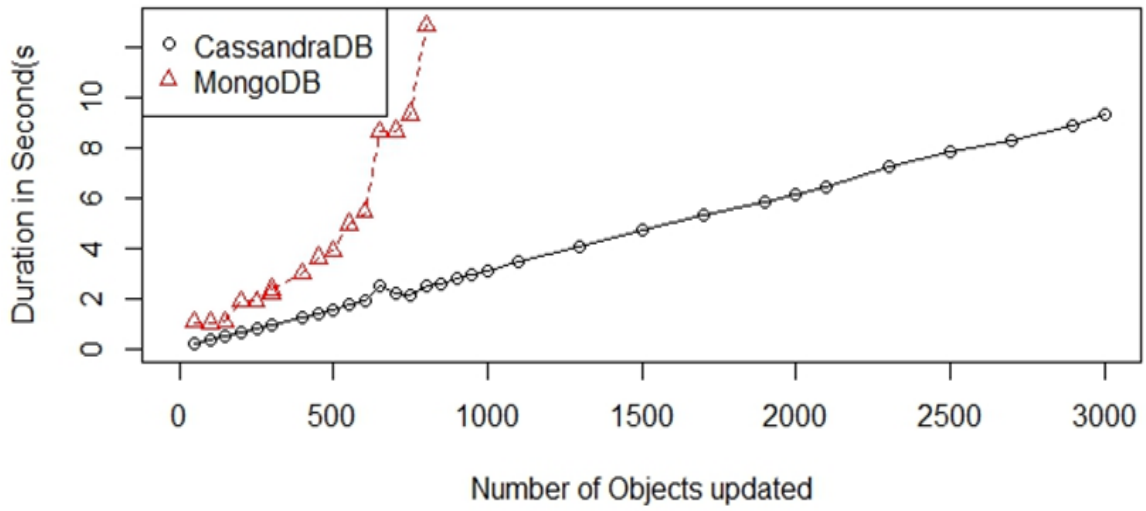


Figure 4.21: CassandraDB and MongoDB update 3000 objects

The study started using the mongo client update from 50 objects, and then incremented the tests by 50 each. The study noticed that the client consumed memory each time the study incremented the objects using a create then updating the object adding the word “updated” next to the object already created. The study could only update up till 800 objects as the client workstation had almost no memory left. The study also noticed all programs running had a slow response and had to stop the API service running on the IDE. To update 50 objects, the duration was 1.041 seconds. To update 300 objects, the duration was 2.166 seconds, but could only update till 800 objects with a duration of 12.838 seconds, see Figure 4.22.

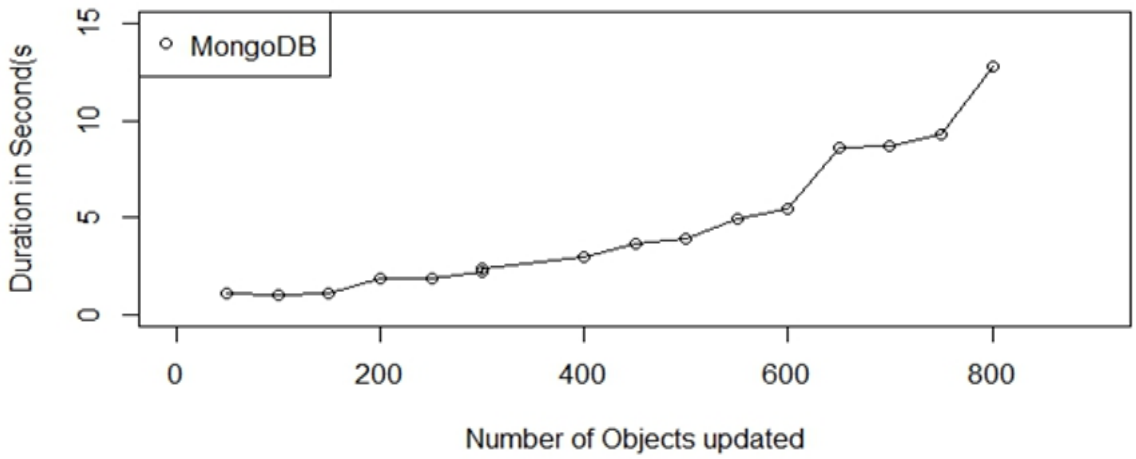


Figure 4.22: Update 300 objects results

Cassandra updated the objects created till 3000 updated objects. Mongo consumed more memory from the host. Each time an update was made Mongo incremented the objects by 50 till only 800 objects and took the longest duration. Redis created more connections to the database each time the study incremented the update operations. This caused the Redis data store to close connections as it couldn't handle anymore connections. Cassandra DB updated 3000 objects within 9.326 seconds.

4.3.4 Delete Results

During the delete phase the goal was to see which database deleted the fastest by choosing 3000 objects, then decrease the amount by 1000 objects each time till 500 objects. Since Redis DB couldn't handle the amount of objects sent, Redis DB was not part of the delete phase. RedisDB was considered a dropout for the delete phase experiment. The study focused on MongoDB and Cassandra DB. Started creating 3000 objects. This consumed 2000 MB memory from the host client. The study noticed the duration was 20.412 seconds to create the objects, which was still longer than the create phase results above. The Delete duration for 3000 objects was 21.846 seconds. The study created 2000 objects and 17.81 seconds to create. The study then deleted the objects with the delete API and the duration was 21.029 seconds. The study had to stop the Mongo DB client as it consumed almost all the memory of the workstation.

The study started with Cassandra from 3000 objects and the duration for Cassandra was 10.359 seconds to create the objects, slightly more than in its create phase, but that was done by incrementing the tests each time. The study then deleted those objects with the delete API

and the duration was 10.09 seconds to delete 3000 objects. Then decrease the objects starting from 2000 objects and those were created in 6.388 seconds. The delete API duration was 6.505 seconds to delete 2000 objects. The study noticed that the less objects created, the quicker the duration took to create and delete them. When the study created 500 objects the duration was 1.558 seconds and the delete duration was 1.549 seconds.

The study we created 3000 objects with MongoDB the API crashed with a java.net. Socket Exception: Connection reset, only 2486 objects were created and couldn't delete as the study received a java.net. Socket Exception: Software caused connection abort: receive failed. The study had to decrease the object creation by 1000 objects. But ran out of socket connections when trying to delete the records with the delete API.

In Figure 4.23 deleting on the Mongo DB 3000 objects the duration was 21.846 seconds to delete. The study tried to delete 2000 objects and could not continue with the tests as the study ran out of memory to do any further delete API requests. Cassandra DB continued deleting objects till it reached zero objects. At 500 objects Cassandra DB could delete those objects in 1.549 seconds.

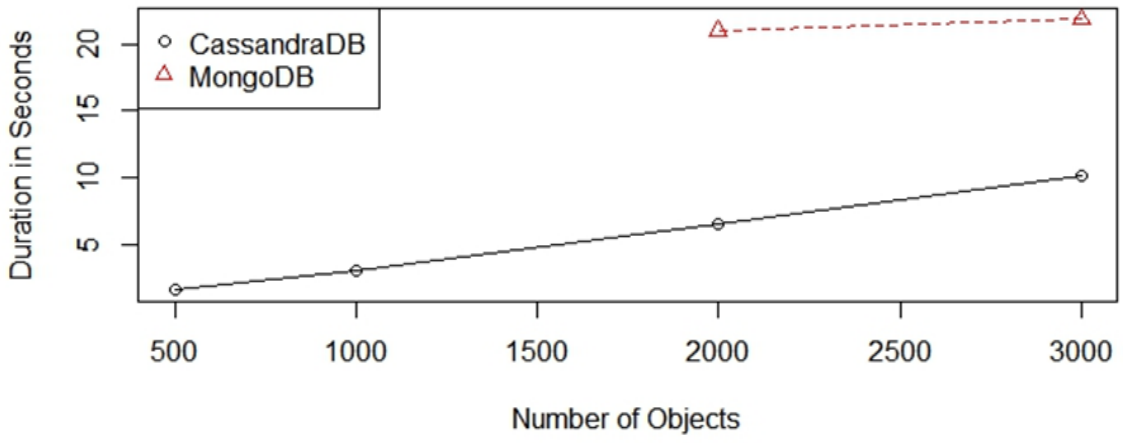


Figure 4.23: Delete-results using IDE client

In Figure 4.23 above Cassandra DB deleted 3000 objects over the fastest duration. As the study decreased the amount of objects, Cassandra DB deleted the objects faster from the data store. Mongo DB consumed almost all the host memory and also took the longest duration deleting objects and could only delete 2000 objects.

4.3.5 CQRS write-model results

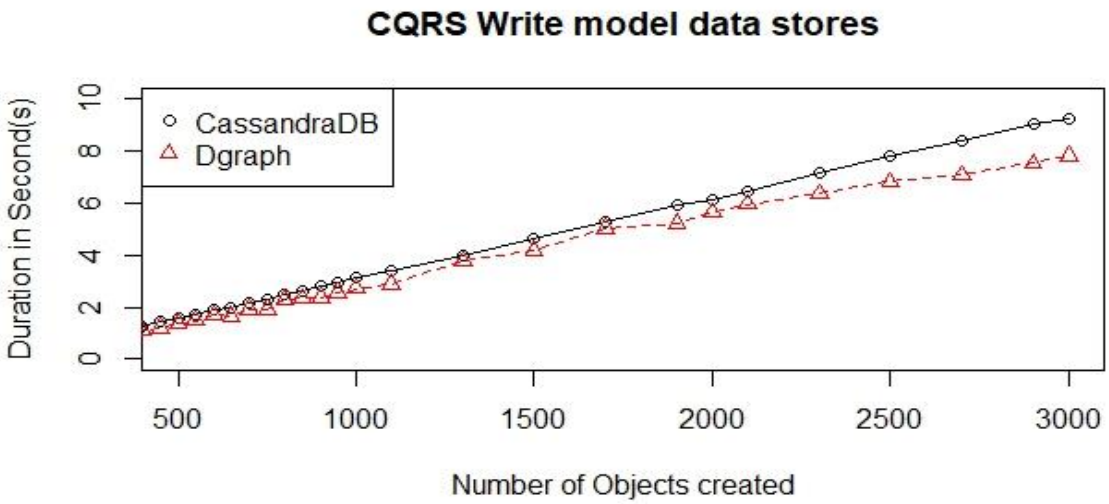


Figure 4.24: Write-model data stores

Figure 4.24 shows the two data stores that performed the best when sending objects to the database. The study experiment results showed that the wide column stores and graph data stores performed the best. The write model create workload simulated multiple users. The study added 50 users called person and incremented on each test by adding 50 more persons on each test case. Based on the results gathered the study selected graph data stores as the preferred write model when designing a CQRS pattern application.

4.3.6 CQRS read-model results

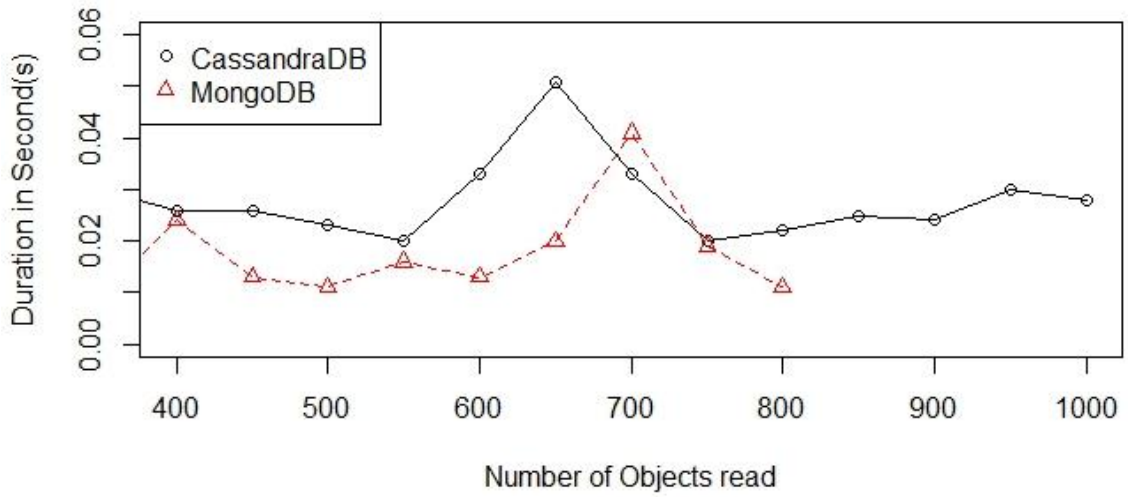


Figure 4.25: Read-model data stores

Figure 4.25 shows the two chosen data stores that performed the best when reading objects from the database. The study selected wide column data store Cassandra DB and Document Store Mongo DB as the read models. Based on the results of the read model the study found that Mongo DB as document store started reading faster than Cassandra DB. Mongo DB was able to read 800 person objects, but each increment of read tests consumed more memory from the host operating system. After 800 objects Cassandra DB continued with read tests until 1000 objects was read. The study could also see a rise in the duration with Mongo DB and Cassandra DB object read duration dropped as the study continued each test. Based on these results the study selected Cassandra DB as the preferred read model when designing a CQRS pattern application.

Table 4.4: The CQRS application architecture's objectives

Architecture element	Considered solutions	NoSQL data store
Write model	Graph-oriented database	Dgraph data store
Read model	Column-oriented database	Cassandra data store

4.4 Limitations of the research and validity

Random selection ensures external validity and the generalizability of the study results. Random allocation of the subjects ensures internal validity to control the intervention or experiment groups. The intervention should be applied multiple times to ensure validity. The sequence of treatments was set to be random.

The measurement tool used in this research study can be used by other researchers to obtain reproducible results when applied in the same setting. If the dependent variable is the same as this study the measurement instrument won't be a problem. The assumption will be that the manipulation of the independent variable caused the change in the dependent variable. The advantages of the research design would be that we eliminate or we can control the unwanted, unrelated variables of the study. The manipulation of the independent variable and the observation of the effect it has on the dependent variable makes it possible to determine the cause and effect relationships. Another advantage is the experiments are in a controlled environment, which means they repeatedly experimented and the results can be compared with one another. This is known as replication.

4.4.1 Testing platforms limitations

The study aimed to scale the application on distributed clusters and achieve that, but couldn't do the tests on the clusters due to time constraints. For the application to be elastic the study could have explored the commodity hardware more and scaled more applications on these nodes, but the study could not achieve that as that would take the focus away from the main objective. More studies would need to be done regarding distributed database systems running on inexpensive commodity hardware.

4.4.2 Language limitations

The study determined the common data objects used in programming languages using the reactive manifesto three-tier application design, the presentation layer package, the service layer package and the repository package classes. The data objects such as key spaces, column families, and indexes all make up the NoSQL data store different categories. The study used the person object and generated several users in keyspaces, column families, and indexes stored for each data category. The data objects for the research was stored and read as lists. The best performing drivers for the Java programming language was the Datastax Java Driver for Apache Cassandra used with Maven project object model (POM). This driver allowed the study to do all create, read, update and delete operation tests and performed the best based on the results achieved.

It’s clear from the above discussion that the study had limitations towards the end of the study. The study couldn’t study other programming languages and could have explored more drivers based on the Java programming language. Considering that research has shown that Cassandra Java programming language driver performed the best.

4.5 Generalizability limitations

The application used the three-tier architecture to be more responsive, resilient, elastic and message driven. This followed the Reactive Manifesto approach, but for the research the study could only test the application to be responsive and message driven. Further investigation should be conducted in order to test the application needs, to be resilient and elastic. This would complete the Reactive Manifesto design approach. The experiment results indicated that the application had the basic traits of the reactive manifesto application.

4.6 Findings of the research study

The first research sub-question was, what are the different hardware requirements for NoSQL data stores to operate on, to achieve high availability and reliability? The reason for this question was to determine the different hardware devices NoSQL data stores can operate on. These hardware devices should allow the data stores to be highly available and reliable.

Chapter 2 Literature review:

Studied the commodity hardware and how they can be used as highly available and reliable hardware. This chapter showed that studies were performed on commodity hardware, benchmarking these inexpensive commodity computers. Most benchmarks done were on cloud servers and the experiments done were to test the performance of these cloud environments. The study needed to look at the different hardware requirements to achieve scalability, reliability, replication and availability of NoSQL data stores. To answer this, the study setup a data center with inexpensive commodity hardware to run the NoSQL databases on. The study made use of low end computers that’s cheap and easy to buy. The study installed Kubernetes on the commodity computers to coordinate the services to run NoSQL data stores on.

The second research sub question asked what is the best architecture to achieve high availability and reliability? The reason for this question was to find out what types of application architecture can be classified as highly available and reliable. The study found evidence that discussed how to help developers should they want to design their own architecture for their

applications. These principles when adopted would allow developers to create highly available applications. Evidence discussed in Chapter 1.

The study needed the Reactive manifesto and the framework that supports this is the CQRS. The CQRS allowed the study to identify what NoSQL database to put on the read-side and what to put on the write-side.

The study used a Conceptual framework as a tool to achieve the results of the study. The limitation of the drivers was to only look at the Java driver. Future work can look at other languages like Go Lang, Scala and PHP for the NoSQL databases and have empirical evidence to find out which one is the best driver. The study chose this driver based on popularity, through the checking of the usage of the driver used in reactive systems.

Chapter 3 Research Methodology:

The methods were formulated from the findings of the literature review. Since the study wanted to answer the research question the study used the concepts gathered from the literature review and created a conceptual framework to answer the research question using the experimental research method applied. The study followed the Reactive Manifesto guidelines to make the application highly available against failures and reliable. The study answered this question by setting up a lab experiment using a three-tier programming language design where we separated the classes of the code into three different layers. The study separated the database and API code from the services into three layers. If the repository or database had to change, the study could change the code segment without affecting the rest of the application classes like the services and endpoint API.

The third research sub question was about finding the best Java Drivers used to persist data in the four types of NoSQL DBs. The reason for this question was to find programming language drivers most commonly used with NoSQL data stores. To find the answer to the question the study selected only four NoSQL categories for each category. The study selected only one data store per category for the research. This selection was possible by searching the best programming driver package used by developers through the literature review. Most research papers used benchmark tools and didn't focus on the developer environment. The study chose the document store driver based on popularity by checking the usage of the driver on document store NoSQL data stores. The study followed the same process for column store, key value store and graph data store categories. Based on popularity in the four categories the study checked the usage of the driver and compared the drivers. The study chose Java

programming language based on usage in all the four NoSQL data store categories.

The experimental setup used Kubernetes to help build the CQRS infrastructure application architecture. The study was able to have the database somewhere else as a stateless container. This helped to scale the nodes if more resources were needed. The study could have chosen the DCOS (Datacenter operating system) architecture, but for the study chose Kubernetes by increasing the nodes, changing databases and keeping the application in the same state.

Chapter 4 Findings and Discussions:

The study selected the best Java programming language for this research based on popularity and usage in the programming language community. The study aimed to answer the research sub question about finding the most common pattern for persisting data objects used by Java Developers. The reason for this question was to determine the most common patterns used for persisting data objects that's used by developers. The study answered this question by running the lab experiment and sending objects to the four different data store categories. The study used the CRUD operations for each data store per category. To answer the research, question the study selected the most reliable and available data store based on sending objects and retrieving objects. As the maintained using the same objects testing the data stores, the results found answered the research question to add to the body of knowledge. The study found that two NoSQL categories could be used as either read or write data store when setting up a CQRS model according to the Reactive Manifesto approach.

The next section will cover the conclusions for the study, what work the study covered and what the study could achieve during the research experiments. The study will look at the recommendations for further studies for this project and highlight limitations and what future work can be done following this research project.

CHAPTER FIVE
CONCLUSION AND RECOMMENDATIONS

5.1 What has been done so far?

This study was done to select and identify the highly available and reliable unstructured NoSQL data store in each of the four categories based on empirical evidence. The study achieved this by the systematic review. Searching for the relevant papers regarding NoSQL data stores on experimental research done and looked at benchmarks performed on the data stores. As part of the search using the systematic review reviewed for evidence, where client tests were done and if articles mentioned programming language, drivers are used in the study. The last part of the systematic review was to look at the platforms where the studies were done and how the data stores performed and explored the settings based on the aim to use inexpensive commodity hardware in the research study.

The objectives for the research study followed an experimental design approach. The study evaluated the research questions based on the dependent variables and the independent variables to determine the relationships between them. The research instrument at first was just the integrated development environment but when scaling the application, the research instrument changed to a cluster environment. The study could collect the data using the integrated development environment and apply for testing and data collection. This allowed the research approach to replicate variables and to allow the study in another setting or cluster environment.

The study recommends what NoSQL database to use for a CQRS application design approach. This would help developers decide which data store performs the best when doing CRUD operations. The focus of the study was to find the read-model data store and the write-model data store and use these data stores for a CQRS pattern approach to design applications and most importantly, stay focused on the Reactive Manifesto application design needs. The study recommends using a Graph-reoriented data store for the write model and a column-oriented data store for the read-model of a CQRS application design.

Based on the CQRS core principle there should be a strict separation of commands and queries, a pattern is where each method used is either a command sent to the data store from the client or query that needs to return data back to the client. Both command and query should be performed independently. In simple terms when a user asks a question this should not change the answer.

The study's research design allowed the study to view the data by measuring the interventions and collect all the data points where interventions were made. The study could share the same amount of objects every time, per data store when an intervention was made, as this was repeatedly observed over time. The study used the same amount of objects persons for each intervention per data store and the timing of each measurement was captured.

5.2 Recommendations

Based on the results achieved, the study would recommend more studies to be done with more than one programming language driver. The study would recommend to explore Go, Scala and Kotlin programming languages using the Reactive Manifesto approach to focus on the three-tier application design. The study would recommend looking at more data structures and objects to send to a data store. The study only focused on looking at four NoSQL data store categories and could only choose one data store per category. The recommendation for further studies would be to look at more than just one data store category and add more data stores to study further.

The study would recommend increasing the memory and central processing units when setting up the research instrument on a single node while running an integrated development environment. The study would recommend using a solid state drive (SSD) instead of a normal hard drive to see the change of speed in storage for input and output. The research instrument can be setup in a cluster environment. The study recommends scaling the data stores to test the down time and recovery of a node that failed and measure the downtime duration of the node when it failed and how long it takes to be active again.

For the research study the study managed to scale the application from a single development environment to a Kubernetes cluster as a Docker container. However, the study would have wanted to interact with the NoSQL data store environment on the kubernetes cluster. The recommendation would be that after scaling the application, to follow the research approach and measure the read and write performance of the data stores. This would allow the study to be valid and reliable to be setup in another setting and answer the research questions.

5.3 Future work

For further studies to follow, studying more than just one programming language driver would help developers choose which one to adapt. This should be tested on the Reactive Manifesto three tier application to separate the packages application layer or presentation layer, service layer or coordinator layer and the repository or data store layer. The study recommends looking at more data stores and NoSQL categories that developers can make use of when sending data to the databases for storage, and when reading data storage from the databases. The application stored as containerized, could be tested on different Kubernetes platforms from bare metal to virtual machines hosted on cloud servers. Create, read, update and delete (CRUD) operations tests could adapt streaming data for input and output metrics as part of further studies on NoSQL data stores.

REFERENCES

Abadi, D., Franklin, M.J., Gehrke, J., Haas, L.M., Halevy, A.Y., Hellerstein, J.M., Ioannidis, Y.E., Jagadish, H. V., Kossmann, D., Madden, S., Mehrotra, S., Agrawal, R., Milo, T., Naughton, J.F., Ramakrishnan, R., Markl, V., Olston, C., Ooi, B.C., Ré, C., Suciu, D., Stonebraker, M., Walter, T., Ailamaki, A., Widom, J., Balazinska, M., Bernstein, P.A., Carey, M.J., Chaudhuri, S., Dean, J. & Doan, A. 2016. The Beckman report on database research. *Communications of the ACM*, 59(2): pp.92-99.

Abramova, V., Bernardino, J. & Furtado, P. 2014. Which NoSQL Database? A Performance Overview. *Open Journal of Databases (OJDB)*, 1(2): pp.17-24.

Adya, A., Wattenhofer, R.P., Bolosky, W.J., Castro, M., Cermak, G., Chaiken, R., Douceur, J.R., Howell, J., Lorch, J.R. & Theimer, M. 2002. Farsite. *ACM SIGOPS Operating Systems Review*, 36(SI): pp.1-14. http://doi.acm.org/10.1145/1060289.1060291.

Baier, J. 2017. *Getting Started with Kubernetes*. https://books.google.co.za/books?hl=en&lr=&id=fnc5DwAAQBAJ&oi=fnd&pg=PP1&dq=kubernetes+setup&ots=ZtM0AXNxzP&sig=_V-H65EVFJGgnOFt0U4N4ghAJpo 21 August 2019.

Bailis, P. & Ghodsi, A. 2013. Eventual consistency today: Limitations, extensions, and beyond. *Communications of the ACM*, 11(3): p.20.

Baker, M., Buyya, R. & Kaushik, K. 2018. Learning & Experience. : pp.1-3. http://learning.maxtech4u.com/what-is-cluster-computing/.

Barroso, L.A., Clidaras, J. & Hölzle, U. 2013. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition. *Synthesis Lectures on Computer Architecture*, 8(3): pp.1-154.

Birman, K.P. 2012. *Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services*. Springer Science & Business Media.

Bonér, J., Farley, D., Kuhn, R. & Thompson, M. 2014. The Reactive Manifesto (Version 2.0). *Reactivemanifesto.Org*, 2(16 September 2014): pp.1-2.

Bouchrika, I. 2018. Introduction-to-distributed-systems. *distributed-systems*. http://www.ejbtutorial.com/distributed-systems/introduction-to-distributed-systems.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. & Gruber, R.E. 2008. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2): p.4.

Colyer, A. 2017. Gray failure: the Achilles' heel of cloud-scale systems. *the morning paper*. https://blog.acolyer.org/2017/06/15/gray-failure-the-achilles-heel-of-cloud-scale-systems/ [29 April 2018].

Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R. & Sears, R. 2010. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, 10: pp.143-154.

Daiga PLASE. 2017. A Systematic Review of SQL-on-Hadoop by Using Compact Data Formats A Systematic Review of SQL-on-Hadoop by Using Compact Data Formats. , 5(2): pp.233-250.

Debski, A., Szczepanik, B., Malawski, M., Spahr, S. & Muthig, D. 2018. A scalable, reactive architecture for cloud applications. *IEEE Software*, 35(2): pp.62-71.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. & Vogels, W. 2007. Dynamo. In *ACM SIGOPS Operating Systems Review*. pp.205-220.

Dorband, J., Raytheon, J. & Ranawake, U. 2013. Commodity computing clusters at goddard space flight center. *Journal of Chemical Information and Modeling*, 53: pp.1689-1699.

Dwarampudi, V., Dhillon, S.S., Shah, J. & Sebastian, N.J. 2010. Comparative study of the Pros and Cons of Programming languages Java , Scala , C ++ , Haskell , VB . NET , AspectJ , Perl , Ruby & Scheme. *arXiv preprint arXiv:1008.3431*.

Eden, A.H. 2007. Three paradigms of computer science. *Minds and Machines*, 17(2): pp.135-167.

Esposito, C., Castiglione, A. & Choo, K.K.R. 2016. Challenges in Delivering Software in the Cloud as Microservices. *IEEE Cloud Computing*, 3(5): pp.10-14. https://ieeexplore.ieee.org/abstract/document/7742281/ [9 October 2019].

Feuerlicht, G. 2010. Database trends and directions: Current challenges and opportunities. In *CEUR Workshop Proceedings*. pp.163-174.

Gessert, F. & Ritter, N. 2016. Scalable data management: NoSQL data stores in research and practice. *2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016*: pp.1420-1423.

Ghemawat, S., Gobioff, H. & Leung, S.-T. 2003. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5): p.29.

Gierke, O. 2013. whoops-where-did-my-architecture-go. *http://olivergierke.de*. http://olivergierke.de/2013/01/whoops-where-did-my-architecture-go/.

Grozev, N. & Buyya, R. 2014. Multi-Cloud Provisioning and Load Distribution for Three-Tier Applications. *ACM Transactions on Autonomous and Adaptive Systems*, 9(3): pp.13-21. http://dl.acm.org/citation.cfm?doid=2676689.2662112 [9 October 2019].

Haseeb, A. & Pattun, G. 2017. A review on NoSQL: Applications and challenges. *International Journal of Advanced Research in Computer Science*, 8(1): pp.27-30.

Helland, P. & South, F.A. 2007. Life beyond Distributed Transactions : an Apostate ' s Opinion Position Paper [2007]. *Cidr*, 14(5): pp.132-141.

Hoda, R. & Azad Kamali, R. 2014. Calculating Total System Availability. I*nformation Services Organization, Amsterdam.*

Hu, W.-C., Kaabouch, N., Guo, H. & Yang, H.-J. 2016. An Empirical Study of NoSQL Databases for Big Data. : pp.60-76.

Huang, P., Guo, C., Zhou, L., Lorch, J.R., Dang, Y., Chintalapati, M. & Yao, R. 2017. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. *USENIX Conference on Hot Topics in Operating Systems*, (17): pp.150-155.

Kabbedijk, J., Jansen, S. & Brinkkemper, S. 2014. A case study of the variability consequences of the CQRS pattern in online business software. *Proceedings of the 17th European Conference on Pattern Languages of Programs*: p.2.

Kainulainen, P. 2014. understanding-spring-web-application-architecture-the-classic-way. *www.petrikainulainen.net*. https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/]17 October 2018].

Kitchenham, B., Pretorius, R., Budgen, D., Brereton, O.P., Turner, M., Niazi, M. & Linkman, S. 2010. Systematic literature reviews in software engineering-A tertiary study. *Information and Software Technology*, 52(8): pp.792-805.

Kumar, R. 2005. *RESEARCH METHODOLOGY a step-by-step guide for beginners*. 3rd ed. SAGE Publications.

Lourenço, J.R., Abramova, V., Vieira, M., Cabral, B. & Bernardino Jorge. 2015. NoSQL Databases: A Software Engineering Perspective. In *Advances in Intelligent Systems and Computing*. pp.741-750.

Lourenço, J.R., Cabral, B., Carreiro, P., Vieira, M. & Bernardino, J. 2015. Choosing the right NoSQL database for the job: a quality attribute evaluation. *Journal of Big Data*, 2(1): p.18.

Mackin, H., Tappert, C. & Perez, G. 2016. Adopting NoSQL Databases Using a Quality Attribute Framework and Risks Analysis. *Proceedings of Student-Faculty Research Day, CSIS*, A(9): pp.97-104.

Martin, R.C. 2012. The Clean Architecture. *https://blog.cleancoder.com*. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html [29 April 2018].

Maxwell, J.A. 1998. Designing a qualitative study: Handbook of applied social research methods. : pp.1-22.

McGaghie William C, Bordage Georges Shea, J.A. 2001. Problem Statement, Conceptual Framework, and Research Question. *Academic Medicine*, 76(9): pp.923-924.

Mertens, D. 2014. *Research and Evaluation in Education and Psychology: Integrating Diversity With Quantitative, Qualitative, and Mixed Methods: Integrating Diversity With*.

Meyerovich, L.A. & Rabkin, A.S. 2013. Empirical analysis of programming language adoption. *ACM SIGPLAN Notices*, 48(10): pp.1-18.

Middleton, A.A.M. & Risk, P.D.L. 2015. *White Paper Introduction to HPCC (High-Performance Computing Cluster)*.

Milani, B.A. & Navimipour, N.J. 2017. A Systematic Literature Review of the Data Replication Techniques in the Cloud Environments. *Big Data Research*, 1: pp.1-7.

Mohammad, J.M. & M. Jalil. 2013. Practical Guidelines for conducting research. *Summarising good research practice in line with the DCED Standard*, 2013.

Neuman, C. 1994. Scale in Distributed Systems. *Readings in Distributed Computing Systems.*: pp.1-28.

Ngxande, M. 2015. Development of Beowulf Cluster to Perform Large Datasets Simulations in Educational Institutions. *International Journal of Computer Applications*, 99(15): pp.29-35.

Ngxande, M. & Moorosi, N. 2014. Development of Beowulf Cluster to Perform Large Datasets Simulations in Educational Institutions. *International Journal of Computer Applications*, 99(15): pp.29-35.

Ongaro, D. & Ousterhout, J. 2014. In Search of an Understandable Consensus Algorithm. *Proceedings of the 2014 USENIX Annual Technical Conference*: pp.305-320.

Peng Xiang, Ruichun Hou & Zhiming Zhou. 2010. Cache and consistency in NOSQL. In *2010 3rd International Conference on Computer Science and Information Technology*. IEEE: pp.117-120. http://ieeexplore.ieee.org/document/5563525/ [9 October 2019].

Petreley, N. 2006. The Ultimate Linux Server. *Linux Journal, 2006*, 184(3): pp.5-8.

Pruijt, L., Wiersema, W. & Brinkkemper, S. 2013. A typology based approach to assign responsibilities to software layers. *Proceedings of the 20th Conference on Pattern Languages of Programs*, (2): pp.1-14. https://dl.acm.org/citation.cfm?id=2725672 [9 October 2019].

Qi, M., Liu, Y., Lu, L., Liu, J. & Li, M. 2014. Big Data Management in Digital Forensics. *2014 IEEE 17th International Conference on Computational Science and Engineering*: pp.238-243.

Ray, B., Posnett, D., Devanbu, P. & Filkov, V. 2017. A large-scale study of programming languages and code quality in GitHub. *Communications of the ACM*, 60(10): pp.91-100.

Sachdeva, G. 2019. The Road to Reactive Programming in Java. : p.1. https://www.thistechnologylife.com/the-road-to-reactive-programming-in-java/ [18 August 2018].

Sareen, P., Professor, A. & Kumar, P. 2017. Nosql Database and Its Comparison With Sql Database. *International Journal of Computational Intelligence Research*, 13(7): pp.1645-1651.

Shooman, M.L. 2002. Reliability of computer systems and networks. *Fault tolerance, analysis and design. NY: John Wiley & Sons*.

Steenhuis, H.-J. & de Bruijn, E.J. 2006. Empirical research in OM: three paradigms. *OM in the New World Uncertainties. Proceedings of the 17th Annual Conference of POMS*: pp.1-10.

Stetson, C. 2018. Principles of Modern Application Development. https://www.nginx.com/blog/principles-of-modern-application-development/ [25 August 2018].

Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N. & Helland, P. 2018. The end of an architectural era: it's time for a complete rewrite. *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*: pp.463-489.

Strauch, C. 2014. NOSQL Databases. *Lecture Notes Stuttgart Media*: p.20. http://nosql-database.org/.

Sukuba, T. 2015. Google Code University – Introduction to Distributed Systems Design. : 1–10. *http://www.hpcs.cs.tsukuba.ac.jp/~tatebe/lecture/h23/dsys/dsd-tutorial.html* [29 April 2018].

Takada, M. 2018. *Distributed systems for fun and profit*. http://book.mixu.net/distsys.

Terblanche, J.T., Kroeze, J.H. & Gilliland, S. 2013. Why using a design and creation strategy to translate a paperbased form into an E-registration web form using HCI principles falls within the context of design science. *22nd International Business Information Management Association Conference, IBIMA 2013*, 3(July 2018): pp.1491-1499.

Tiwari, S. 2011. *PROFESSIONAL NoSQL INTRODUCTION*. John Wiley & Sons, Inc.

Vaish, G. 2013. *Getting Started with NoSQL*. Packt Publishing ©2013.

Wiesmann, M., Pedone, F., Schiper, A., Kemme, B. & Alonso, G. 2002. Understanding replication in databases and distributed systems. *Proceedings 20th IEEE International Conference on Distributed Computing Systems*: pp.464-474.

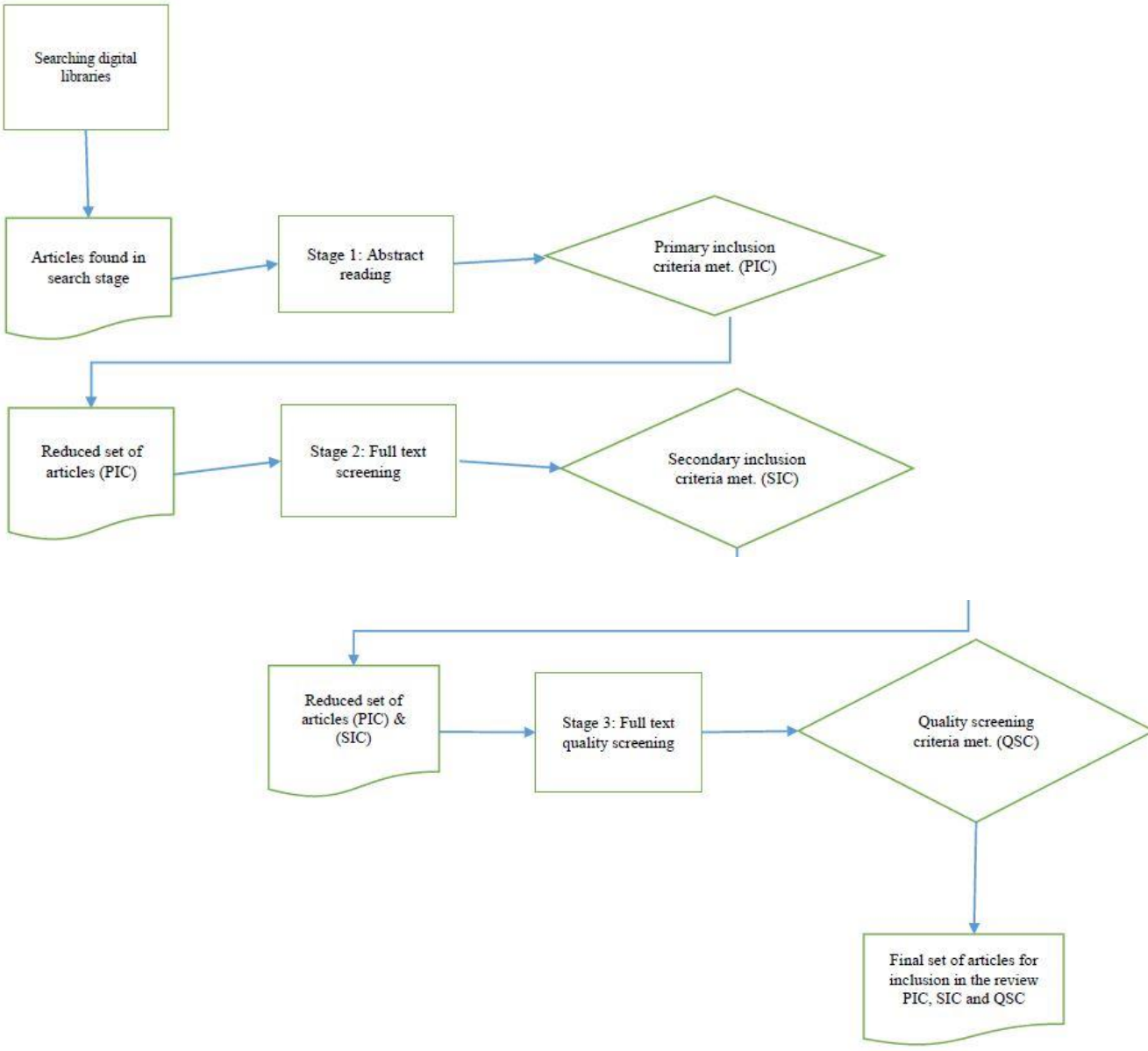
Yassien, A.W. & Desouky, A.F. 2016. RDBMS , NoSQL , Hadoop : A Performance-Based Empirical Analysis. *AMECSE '16 Proceedings of the 2nd Africa and Middle East Conference on Software Engineering*, (2): pp.52-59.

Zafar, R., Yafi, E., Zuhairi, M.F. & Dao, H. 2017. Big Data: The NoSQL and RDBMS review. *ICICTM 2016 - Proceedings of the 1st International Conference on Information and Communication Technology*, (May): pp.120-126.

Appendix A
Systematic review, quality assessment

Articles	Experiment performed	Benchmark performed	Client tests	language drivers	Platform used
[1] RDBMS, NoSQL, HadoopA Performance-Based Empirical Analysis	YES	Yahoo! Cloud Serving Benchmark (YCSB)	NO	NO	NO CLUSTER
[2] Performance Evaluation of NoSQL Databases A Case Study	YES	Yahoo! Cloud Serving Benchmark (YCSB)	AMAZON EC2 INSTANCE	NO	AMAZON EC2 CLOUD
[3] Experimental Eevaluation of NOSQL Databases	YES	Yahoo! Cloud Serving Benchmark (YCSB)	NO	NO	NO CLUSTER
[4] Quality Attribute-Guided Evaluation of NoSQL Databases A Case Study	YES	Yahoo! Cloud Serving Benchmark (YCSB)	AMAZON EC2 INSTANCE	NO	AMAZON EC2 CLOUD
[5] Quantitative-Analysis-of-Consistency-in-NoSQL-Key-values-Stores	YES	Yahoo! Cloud Serving Benchmark (YCSB)	YES	NO	NO CLUSTER
[6] Consistent and Durable Data Structures for Non-volatile	YES	Yahoo! Cloud Serving Benchmark (YCSB)	NO	NO	CLUSTER
[7] A review on NoSQL Applications and challenges	NO	NONE	NO	NO	REVIEW
[8] Which NoSQL Database	YES	Yahoo! Cloud Serving Benchmark (YCSB)	NO	NO	NO CLUSTER
[9] Big Data Management in Digital Forensics	YES	Yahoo! Cloud Serving Benchmark (YCSB)	NO	NO	AMAZON EC2 CLOUD

Appendix B
Systematic review, Search strategy



Appendix C Quality Assessment **QCQA**

#	Article	1	2	3	4	5	6	7	8	9	10	Total
1	Yassien, A.W. and Desouky, A.F., 2016, May. RDBMS, NoSQL, Hadoop: A Performance-Based empirical analysis. In proceedings of the 2nd Africa and Middle East Conference on Software Engineering (pp. 52-59). ACM.	1	1	1	1	1	1	1	1	1	1	10
2		1	1	1	1	1	1	1	1	1	1	10
3	Abramova, V., Bernardino, J. and Furtado, P., 2014. Experimental evaluation of NoSQL databases. International Journal of Database Management Systems, 6(3), p.1.	1	1	1	1	1	1	1	1	1	1	10
4	Klein, J., Gorton, I., Ernst, N., Donohoe, P., Pham, K. and Matser, C., 2015. Quality Attribute-Guided Evaluation of NoSQL Databases: A Case Study. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.	1	1	1	1	1	1	1	1	1	1	10
5	Liu, S., Nguyen, S., Ganhotra, J., Rahman, M.R., Gupta, I. and Meseguer, J., 2015, September. Quantitative analysis of consistency in NoSQL key-value stores. In International Conference on Quantitative Evaluation of Systems (pp. 228-243). Springer, Cham.	1	1	1	1	1	1	1	1	1	1	10
6	Venkataraman, S., Tolia, N., Ranganathan, P. and Campbell, R.H., 2011, February. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In FAST (Vol. 11, pp. 61-75).	1	1	1	1	1	1	1	1	1	1	10
7	Haseeb, A. and Pattun, G., 2017. A review on NoSQL: Applications and challenges. International Journal of Advanced Research in Computer Science, 8(1).	1	1	1	1	1	1	1	1	1	1	10
8	Abramova, V., Bernardino, J. and Furtado, P., 2014. Which nosql database? a performance overview. Open Journal of Databases (OJDB), 1(2), pp.17-24.	1	1	1	1	1	1	1	1	1	1	10
9	Qi, M., Liu, Y., Lu, L., Liu, J. and Li, M., 2014, December. Big data management in digital forensics. In Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on (pp. 238-243). IEEE.	1	1	1	1	1	1	1	1	1	1	10
10	Alomari, E., Barnawi, A. and Sakr, S., 2015. Cdport: A portability framework for nosql datastores. Arabian Journal for Science and Engineering, 40(9), pp.2531-2553.											
11	Anjard, R.P., 1994. The basics of database management systems (DBMS). Industrial Management & Data Systems, 94(5), pp.11-15.											
12	Lourenço, J.R., Cabral, B., Carreiro, P., Vieira, M. and Bernardino, J., 2015. Choosing the right NoSQL database for the job: a quality attribute evaluation. Journal of Big Data, 2(1), p.18.											

13	Gessert, F. and Ritter, N., 2016, May. Scalable data management: NoSQL data stores in research and practice. In Data Engineering (ICDE), 2016 IEEE 32nd International Conference on (pp. 1420-1423). IEEE.											
14	Sareen, P. and Kumar, P., 2015. NoSQL Database and its Comparison with SQL Database. Int J Comput Sci Commun Networks, 5, pp.293-298.											
15	Haseeb, A. and Pattun, G., 2017. A review on NoSQL: Applications and challenges. International Journal of Advanced Research in Computer Science, 8(1).											
16	Srivastava, P.P., Goyal, S. and Kumar, A., 2015, October. Analysis of various NoSql database. In Green Computing and Internet of Things (ICGCIoT), 2015 International Conference on (pp. 539-544). IEEE.											
17	Poljak, R., Poščić, P. and Jakšić, D., 2017, May. Comparative analysis of the selected relational database management systems. In Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2017 40th International Convention on (pp. 1496-1500). IEEE.											
18	Lourenço, J.R., Abramova, V., Vieira, M., Cabral, B. and Bernardino, J., 2015. Nosql databases: A software engineering perspective. In New Contributions in Information Systems and Technologies (pp. 741-750). Springer, Cham.											
19	Bajpeyee, R., Sinha, S.P. and Kumar, V., 2015. Big Data: A Brief Investigation on NoSQL Databases,“. International Journal of Innovations & Advancement in Computer Science, 4(1), pp.28-35.											
20	Zafar, R., Yafi, E., Zuhairi, M.F. and Dao, H., 2016, May. Big Data: The NoSQL and RDBMS review. In Information and Communication Technology (ICICTM), International Conference on (pp. 120-126). IEEE.											
21	Domaschka, J., Hauser, C.B. and Erb, B., 2014, September. Reliability and availability properties of distributed database systems. In Enterprise Distributed Object Computing Conference (EDOC), 2014 IEEE 18th International (pp. 226-233). IEEE.											

Appendix D
Data extraction form.

Author	title	Abstract	Keywords	Publisher	Conclusions
A, Veronika Abramova A, Jorge Bernardino B, Pedro Furtado	Which NoSQL Database? A Performance Overview	NoSQL data stores are widely used to store and retrieve possibly large amounts of data, typically in a key-value format. There are many NoSQL types with different performances, and thus it is important to compare them in terms of performance and verify how the performance is related to the database type. In this paper, we evaluate five most popular NoSQL databases: Cassandra, HBase, MongoDB, OrientDB and Redis. We compare those databases in terms of query performance, based on reads and updates, taking into consideration the typical workloads, as represented by the Yahoo! Cloud Serving Benchmark. This comparison allows users to choose the most appropriate database according to the specific mechanisms and application needs.	NoSQL databases, performance evaluation, execution time, benchmark, YCSB	Open Journal of Databases (OJDB) 2014 Journal article	As an overall analysis, in terms of optimization, NoSQL databases can be divided into two categories, the databases optimized for reads and the databases optimized for updates. As future work, we will compare and analyze the performance of NoSQL databases further: we will increase the number of operations performed and run NoSQL databases over multiple servers. This evaluation will allow us to better understand how NoSQL behaves while running in distributed and parallel environments. We also plan to evaluate the performance of Graph databases.
Alomari, Ebtesam Barnawi, Ahmed Sakr, Sherif	CDPort: A Portability Framework for NoSQL Datastores	Cloud computing technology has been growing over the past few years. Currently, cloud providers provide their consumers with several cloud	Cloud computing · Database-as-a-Service · NoSQL · Portability	Arabian Journal for Science and Engineering (2015) Journal article	The portability of data between different databases in the cloud platform becomes one of the main obstacles

		services. However, developers face many difficulties when they have to move their data or software from one cloud platform to another due to the lack of standards. This challenge is considered as one of the key obstacles that prevent many applications from moving to the cloud environment. In this paper, we focus on the challenge of data portability. We propose a common data model and a standardized API for SQL and NoSQL cloud databases. In particular, our approach hides the possible variations of the backend data storage models from the application layer. In addition, our framework is equipped with tools that support the conversion, transformation and data exchange between the different data storage models. The current implementation of our framework supports four different data storage systems: Amazon RDS, Google Datastore, Amazon SimpleDB and MongoDB. However, our framework is designed in a flexible way such that it can be easily			toward pervasive cloud adoption. Users may be locked-in to one platform provider because the other platforms offer different APIs to manage or access the data. Further, the databases have different data models, and in NoSQL there is no unified way to access the data. Therefore, the movement of data between them becomes a more difficult and time-consuming process.

		extended to support other data storage systems. Moreover, we offer a standard query abstraction to enable automatic translation between NoSQL query patterns and their associated SQL queries (in both directions). The experimental evaluation of our framework shows that using our framework eliminates or minimizes the effort of rewriting the application code when the backend data storage system is changed. Further, the proposed transformation tool reduces the effort of maintaining data portability between the different data models that we have considered.			
Anjard Sr, Ronald P	The basics of database management systems (DBMS)	During the past 30 years, data processing has undergone evolutionary changes. Processing with a database management system (DBMS) provides a number of advantages. For example, the location of the DBMS within the software chain provides data interdependence. A software mask within the DBMS provides data integrity. Structured query language	Software & systems; Business And Economics--Computer Applications; Data base management systems; Data processing; Implementations; Systems development	Industrial Management & Data Systems (1994)	This article has presented the very basics of today's dynamic DBMS. As evidenced from the professional magazines, there is dynamic growth and development. New, more user-friendly systems are being developed the better to meet customers' increasing and complex requirements.

		(SQL) is a standard database language used to query and update the vast majority of client-server DBMSs. The demand for better PC-based DBMSs has driven the development of client-server technology. In early implementations of DBMSs, data processing departments continued designing database applications using methods they had used with conventional files. However, the design methodology improved over the years. The online environment for a DBMS permits users outside the data center to access databases. The number of fields, segment types, record types, and tables in mainframe DBMSs has no practical limits.			
Lourenço, João Ricardo Cabral, Bruno Carreiro, Paulo Vieira, Marco Bernardino, Jorge	Choosing the right NoSQL database for the job: a quality attribute evaluation	For over forty years, relational databases have been the leading model for data storage, retrieval and management. However, due to increasing needs for scalability and performance, alternative systems have emerged, namely NoSQL technology. The rising interest in NoSQL technology, as well as the	NoSQL databases;Key-value;Document store;Columnar;; columnar; document store; engineering; graph; key-value; nosql databases; quality attributes; software; software	Journal of Big Data (2015)	There is still not enough information to verify how suited each non-relational database is in a specific scenario or system. Moreover, each working system differs from another and all the necessary functionality and mechanisms highly affect the database choice. Furthermore, we tried to

		growth in the number of use case scenarios, over the last few years resulted in an increasing number of evaluations and comparisons among competing NoSQL technologies. While most research work mostly focuses on performance evaluation using standard benchmarks, it is important to notice that the architecture of real world systems is not only driven by performance requirements, but has to comprehensively include many other quality attribute requirements. Software quality attributes form the basis from which software engineers and architects develop software and make design decisions. Yet, there has been no quality attribute focused survey or classification of NoSQL databases where databases are compared with regards to their suitability for quality attributes common on the design of enterprise systems. To fill this gap, and aid software engineers and architects, in this article, we survey and create a concise and up-to-date comparison of NoSQL engines, identifying	architecture		find the best databases on a quality attribute perspec-tive, an approach still not found in current literature. The summary table we presented makes it clear that there is a current need for a broad study of quality attributes in order to better understand the NoSQL ecosys-tem, and it would be interesting to conduct research in this domain. In particular, research is currently lack-ing in terms of Reliability, Robustness, Durability and Maintainability, with most work in literature focusing on raw performance.

		their most beneficial use case scenarios from the software engineer point of view and the quality attributes that each of them is most suited to.			
Gessert, Felix Ritter, Norbert	Scalable data management: NoSQL data stores in research and practice	The unprecedented scale at which data is consumed and generated today has shown a large demand for scalable data management and given rise to non-relational, distributed NoSQL database systems. Two central problems triggered this process: 1) vast amounts of user-generated content in modern applications and the resulting requests loads and data volumes 2) the desire of the developer community to employ problem-specific data models for storage and querying. To address these needs, various data stores have been developed by both industry and research, arguing that the era of one-size-fits-all database systems is over. The heterogeneity and sheer amount of these systems - now commonly referred to as NoSQL data stores - make it increasingly difficult to select the most appropriate system for a given application. Therefore,		2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016	There are many open challenges for NoSQL data management. NoSQL systems need to support novel application architectures (e.g., single-page or real-time apps) and deliver low latency in face of distributed storage and application tiers. There are currently no means to turn the functionality-performance trade-off into a tunable runtime configuration. Polyglot database services lack the capability to automate, optimize and learn the best choice of given database systems. They can neither route queries and data to minimize SLA violations nor preserve consistency and transaction guarantees.

		these systems are frequently combined in polyglot persistence architectures to leverage each system in its respective sweet spot. This tutorial gives an in-depth survey of the most relevant NoSQL databases to provide comparative classification and highlight open challenges. To this end, we analyze the approach of each system to derive its scalability, availability, consistency, data modeling and querying characteristics. We present how each system's design is governed by a central set of trade-offs over irreconcilable system properties. We then cover recent research results in distributed data management to illustrate that some shortcomings of NoSQL systems could already be solved in practice, whereas other NoSQL data management problems pose interesting and unsolved research challenges.			
Berrington, James	Databases	A database is a structured collection of records or data that is stored in a computer so that it can be consulted by a program to answer queries.	Data; information; normalization; relational database	Anaesthesia & Intensive Care Medicine (2017)	

		Records retrieved through queries become information that can be used to make decisions. A database consists of one or more tables containing records of values for fields that pertain to the attri- butes of the object being represented by the table. Relational data- bases contain multiple tables that are linked by means of key fields. A database management system is the computer program that man- ages the database and queries the data to produce reports of informa- tion. Examples of simple databases and how they are produced are described in this article.			
Sareen, Pankaj Professor, Assistant Kumar, Parveen	Nosql Database and Its Comparison With Sql Database	-NOSQL databases is an emerging alternative to the most widely used relational databases. As the name suggests, it does not completely replace SQL but compliments it in such a way that they can co-exist. In this paper we will be discussing the NOSQL database, types of NOSQL database type, advantages and disadvantages of NOSQL.	-NOSQL; Data Stores; Relational Databases	Int J Comput Sci Commun Networks (2015)	There are few limitations in SQL database: Scalability: Users have to scale relational database on powerful servers that are expensive and difficult to handle. To scale relational database, it has to be distributed on to multiple servers. Handling tables across different servers is a chaos. Complexity: In SQL server's data has to fit into

					tables anyhow. If your data doesn't fit into tables, then you need to design your database structure that will be complex and again difficult to handle. RDBMS is a great tool for solving ACID problems when data validity is crucial, when you need to support dynamic queries. NoSQL is a great tool for solving data availability problems, when it's more important to have fast data than up-to-the-minute just updated data, when you need to scale based on changing requirements. Pick the right tool for the job.
No, Issn Sigar, Kenneth Otula	A review on NoSQL: Applications and challenges	Now a day the technology is growing rapidly stimulating and generating whopping amount of data. Every day people and companies generate huge amounts of data and this data may be unstructured, semi-structured and structured. That's why we need to design databases which can store this type of data in huge volumes. The name of this database is NoSQL databases. NoSQL database solves this type of	NoSQL, Graph DB, Key value DB, Column DB, Document DB	2015 International Journal of Advanced Research in Computer Science	Now in this modern era people are moving on SQL to NoSQL. In NoSQL have lots of features in the perspective of huge amount of storage management and their utilization. We will plan for enhancement the security issues for better use of recourses in future.

		problems. NoSQL database is being used widely and it is a commonly known as engines well scale. Therefore, it is useful to investigate how different factors, such as workload, data size and number of simultaneous sessions influence scaling capabilities. In this paper we describe the brief introduction of NoSQL and its categories and also what the benefits of NoSQL are and why we are using now.			
Klein, John Gorton, Ian Ernst, Neil Donohoe, Patrick Pham, Kim Matser, Chrisjan	Performance Evaluation of NoSQL Databases: A Case Study	For software developers, the selection of a particular NoSQL technology imposes a specific distributed software architecture and data model, making the technology selection difficult to defer. NoSQL database technologies provide high levels of performance, scalability, and availability by simplifying data models and supporting horizontal scaling and data replication. Each NoSQL product embodies a particular set of consistency, availability, and partition tolerance (CAP) tradeoffs, along with a data model that reduces the conceptual	big data; nosql; performance	Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems (2015)	NoSQL database technology offers benefits of scalability and availability through horizontal scaling, replication, a nd simplified data models, but the specific implementation must be chosen early. There were a number of challenges in carrying out such a performance analysis on big data systems. These included: Creating the test environment – performance analysis at this scale requires very large data sets that mirror real application data. This

		mismatch between data access and data storage models. This means technology selection must be done early, often with limited information about specific application requirements, and the decision must balance speed with precision, as the NoSQL solution space is large and evolving rapidly. In this paper we present the method and results of a study to compare the architecturally-relevant characteristics of three NoSQL databases for use in a large, distributed healthcare organization. We reflect on some of the fundamental difficulties of performing detailed technical evaluations of NoSQL databases specifically, and big data systems in general, that have become apparent during our study			raw data must then be loaded into the different data models that we defined for each different NoSQL database. A minor change to the data model in order to explore performance implications required a full data set reload, which is time-consuming. Validating quantitative criteria - Quantitative criteria, with hard "go/no-go" thresholds, were problematic to validate through prototyping, due to the large number of tunable parameters in each database, operating system, and cloud infrastructure. Minor configuration parameter changes can cause unexpected interactions performance effects, often due to non-obvious between the different parameters. In order to avoid entering an endless test and analyze cycle, we framed the performance criteria in terms of the shape of the performance

					curve, and focused more on sensitivities and inflection points.
Srivastava, Pragati Prakash Goyal, Saumya Kumar, Anil	Analysis of various NoSql database	In the age of internet, when data production has gone off-bounds, organizations are facing a tough challenge in terms of processing, analyzing and storing big data. The major drawback with this data is that it is not only being created at a lightning fast pace but it is also unstructured i.e. does not have a fixed schema. Moreover, it is arising from disparate and discrete sources such as the social media. NoSql or Not Only Sql databases offer a highly flexible and horizontally scalable solution to store structured, semi-structured and unstructured data. These databases store data in the form of key-value pairs which offers better availability and high throughput performance in terms of processing queries. They are designed to be highly customizable according to the user's requirements, and well suited for the needs of the overlying application as well as the underlying data being stored.	Availability; Big Data; Consistency; NoSql; Scalability	roceedings of the 2015 International Conference on Green Computing and Internet of Things, ICGCIoT 2015	Currently the NoSql technology is emerging and a lot needs to be discovered. According to a study conducted by the Information Week magazine, 44% of organizations do not know what NoSql databases are [13]. It is important to realize the potential that this storage technology carries which can cause a massive paradigm shift in an organization's method of storing and processing data. In this paper we have evaluated the most popular NoSql solutions and also discussed their architectural working and best use cases. In future an objective comparison of these databases using fixed workloads of reads and writes can be carried out to compare their relative performance.

		This paper firstly provides a general overview of the NoSQL storage technology. Later a thorough analysis will highlight the features, strengths and limitations of six most popular NoSQL databases and thus would help the organizations to choose a NoSql database which is well suited to their needs.			
Poljak, R. Poscic, P. Jaksic, D.	Comparative analysis of the selected relational database management systems	The database management system is a software that enables easier work with databases i.e. to define database structure, retrieve stored data, enter data into the database and process the previously stored data in the database. In this article we have compared 3 relational database management systems (RDBMS) - Oracle 11g, MySQL and PostgreSQL. They are compared according to the simple criteria that we defined, such as the comparison of basic data, syntax, data types and speed performance. The main contribution of the article is a comparison of 3 different RDBMSs by our own score criteria		2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)	However, we must State conclusions are based on a very simple database and benchmark - more comprehensive syntax and data type comparison, as well as speed performance will be done in the future (with larger and more complex database, as well as more complex queries).

Lourenço, João Ricardo Abramova, Veronika Vieira, Marco Cabral, Bruno Bernardino Jorge	NoSQL Databases: A Software Engineering Perspective	For over forty years, relational databases have been the leading model for data storage, retrieval and management. However, due to increasing needs for scalability and performance, alternative systems have started being developed, namely NoSQL technology. With increased interest in NoSQL technology, as well as more use case scenarios, over the last few years these databases have been more frequently evaluated and compared. It is necessary to find if all the possibilities and characteristics of non-relational technology have been disclosed. While most papers perform mostly performance evaluation using standard benchmarks, it is nevertheless important to notice that real world scenarios, with real enterprise data, do not function solely based on performance. In this paper, we have gathered a concise and up-to-date comparison of NoSQL engines, their most beneficial	NoSQL databases; Key-Value; Document Store; Columnar; Graph; Cassandra; MongoDB; Couchbase; Software engineering; Quality attributes	Springer International Publishing Switzerland 2015 Á. Rocha et al. (eds.),	there is a current need for a broad study of quality attributes in order to better understand the NoSQL ecosystem, and it would be interesting to conduct research in this domain. NoSQL is still an in-development field, with many questions and a shortage of definite answers. Its technology is ever-increasing and ever-changing, rendering even recent benchmarks and performance evaluations obsolete. There is also a lack of studies which focus on use-case oriented scenarios or software engineering quality attributes. All of these reasons make it difficult to find the best pick for each of the quality attributes we chose in this work, as well as others. We concluded that although there have been a variety of studies and evaluations of NoSQL technology, there is still not enough information to verify how

		use case scenarios from the software engineer viewpoint, their advantages and drawbacks by surveying the currently available literature.			suited each non-relational database is in a specific scenario or system.
Bajpayee, R Sinha, SP Kumar, V	Big Data: A Brief investigation on NoSQL Databases	As the usage of information technology has increased in the world, the Data generation from various resources has unexpectedly increased. The technology for handling the vast amount of data has not developed as compared to the data generation. Traditional database systems are unable to handle the increased volume of data due to its volume, Variety, Complexity, variability. To deal with this problem, Hadoop Distributed File System (HDFS) like technology is developed. The data to be processed exists in different format that is why the traditional relational database management System is suitable for the big data. To deal with the unstructured data various database tools have been developed. This paper mainly focuses on the various NoSQL Database tools that are available to deal with different types of data. It also	big data; big data tools; hdfs; nosql database; ntfs	International Journal of Innovations & Advancement in Computer Science IJIACS ISSN 2347 – 8616 , January 2015	In the age of information technology, data is a very important to extract the useful information. It is obvious that data exists in different format. The processing of big data is still a challenging task. There is no universal tool which can handle enormous and data of various formats. Document oriented, Key-Value pair, Column and graph type of NoSQL databases are developed to handle this variety of data. The summarized discussion about different NoSQL databases is helpful in selection of suitable NoSQL database.

		includes a brief comparison between (NTFS and HDFS) and (NoSQL and Traditional Relational Database).			
Zafar, Rashid Yafi, Eiad Zuhairi, Megat F. Dao, Hassan	Big Data: The NoSQL and RDBMS review	-The quantity of data transmitted in the network intensified rapidly with the increased dependency on social media applications, sensors for data acquisitions and smartphones utilizations. Typically, such data is unstructured and originates from multiple sources in different format. Consequently, the abstraction of data for rendering is difficult, that lead to the development of a computing system that is able to store data in unstructured format and support distributed parallel computing. To data, there exist approaches to handle big data using NoSQL. This paper provides a review and the comparison between NoSQL and Relational Database Management System (RDBMS). By reviewing each approach, the mechanics of NoSQL systems can be clearly distinguished from the RDBMS. Basically, such	NoSQL, databases, structured data, unstructured data, /Jig Data, Management	CICTM 2016 - Proceedings of the 1st International Conference on Information and Communication Technology	In conclusion, huge data volumes and complex associated data present great challenge that RDBMS is the only way to handle big data. Reliability, readiness and fault tolerance are the determining factor to choose the data organizing tool. Nettlix converted its data management system from Oracle to Cassandra. After conversion, the company achieves over 10,000 writes per second per node and the average latency rate is less than 0.015. The total cost of Cassandra set for running on the AmazonEC2 is $60 per hour for a cluster of 48 nodes. Such cases have shown that the NoSQL data models are built to support the insertion and reading operations effectively, keeping extra data and column-major data handling, error tolerance to short-time conflict and its effects. The

		systems rely on multiple factors, that include the query language, architecture, data model and consumer API. This paper also defines the application that matches the system and subsequently able to accurately correlates to a specific NoSQL system.			NoSQL system is being accepted globally and the key benefit is that the web based servers can perform the verification and user rights on a centralized server. The transfer of data from RDBMS to NoSQL systems is easy because both systems employ the same retrieval value i.e. in JSON form. The NoSQL systems are basically used for the applications that need high performance, reliability of the data, and data that run on multiple nodes connected to one cluster. The applications running currently can be converted on the NoSQL systems by using process of refactoring. The issues that are currently in the data management systems will help the people to use applications that meet most of their needs.
Domaschka, Jorg Hauser, Christopher B. Erb, Benjamin	Reliability and Availability Properties of Distributed Database Systems	Distributed database systems represent an essential component of modern enterprise application architectures. If the overall application needs to provide reliability and availability, the	Reliability, availability, database systems, replica-tion, partitioning, consistency, scalability	2014	

		database has to guarantee these properties as well. Entailing non-functional database features such as replication, consistency, conflict management, and partitioning represent subsequent challenges for successfully designing and operating an available and reliable database system. In this document, we identify why these concepts are important for databases and classify their design options. Moreover, we survey how eleven modern database systems implement these reliability and availability properties.			

Appendix E
Comparing studies by the variables

Comparing studies by variables					
Author (s), title	Year	Relational databases	NoSQL	database	data structures
James Berrington ,Databases	2017	X			
Comparative Analysis of the Selected Relational Database Management Systems R. Poljak, P. Poši and D. Jakši	2017	X		X	
NOSQL Database and Its Comparison with RDBMS Dr. A. B. Raut	2017	X	X	X	
Big Data: The NoSQL and RDBMS review Rashid Zafar, Eiad Yafi, Megat F. Zuhairi, Hassan Dao	2017	X	X		
Scalable Data Management: NoSQL Data Stores in Research and Practice Felix Gessert, Norbert Ritter	2016		X		
Analysis of Various NoSql Database Pragati Prakash Srivastava; Saumya Goyal; Anil Kumar	2016		X	X	
Performance Evaluation of NoSQL Databases: A Case Study John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe	2015		X	X	
NoSQL Databases: A Software Engineering Perspective João Ricardo Lourenço, Veronika Abramova, Marco Vieira, Bruno Cabral, and Jorge Bernardino	2015		X	X	
Choosing the right NoSQL database for the job: a quality attribute evaluation João Ricardo Lourenço, Bruno Cabral, Paulo Carreiro, Marco Vieira and Jorge Bernardino	2015		X	X	
CDPort: A Portability Framework for NoSQL Datastores Ebtesam Alomari · Ahmed	2015		X		

Barnawi · Sherif Sakr					
Big Data: A Brief investigation on NoSQL Databases Roshni Bajpayee Raipur (C.G) Sonali Priya Sinha Raipur (C.G) Vinod Kumar	2015		X	X	
Which NoSQL Database? A Performance Overview Veronika Abramova A, Jorge Bernardino A, B, Pedro Furtado	2014		X	X	
A review on NoSQL: Applications and challenges Abdul Haseeb Geeta Pattun	2014		X		
Reliability and Availability Properties of Distributed Database Systems Jorg Domaschka Christopher B. Benjamin Erb	2014	X	X	X	

Create API objects in seconds

Create API results

Objects	CassandraDB	MongoDB	RedisDB	Dgraph
50	0.188	0.671	0.16	0.221
100	0.343	0.973	0.239	0.463
150	0.509	1.651	0.397	0.574
200	0.654	1.952	0.693	0.873
250	0.812	1.745	0.974	0.924
300	1.065	2.705	1.443	1.004
350	1.132	3.179	3.169	1.051
400	1.275	3.486	1.52	1.119
450	1.472	3.334	2.097	1.203
500	1.554	4.183	2.794	1.401
550	1.731	5.176	3.595	1.477
600	1.916	6.18	1.804	1.705
650	1.993	6.95	2.847	1.66
700	2.167	9.437	3.654	1.905
750	2.294	13.729	2.155	1.915
800	2.505	19.942	3.398	2.311
850	2.63	19.259	4.647	2.35
900	2.788	20.281	4.886	2.376
950	2.943	23.438	2.796	2.57
1000	3.122	44.999	3.319	2.709
1100	3.422	66.082	N/A	2.9
1300	4.011	69.369	N/A	3.777
1500	4.65	95.238	N/A	4.19
1700	5.305	188.308	N/A	5.039
1900	5.913	15.829	N/A	5.234
2000	6.148	20.406	N/A	5.644
2100	6.486	24.432	N/A	5.956
2300	7.192	23.772	N/A	6.366
2500	7.821	47.034	N/A	6.827
2700	8.378	81.553	N/A	7.089
2900	9.027	N/A	N/A	7.529
3000	9.247	N/A	N/A	7.833

Read API objects in seconds

Objects	CassandraDB	MongoDB	RedisDB
50	0.08	0.007	0.207
100	0.039	0.007	0.317
150	0.034	0.009	0.537
200	0.039	0.009	0.876
250	0.071	0.011	1.34

300	0.042	0.014	1.777
350	0.03	0.009	2.47
400	0.026	0.024	3.418
450	0.026	0.013	N/A
500	0.023	0.011	N/A
550	0.02	0.016	N/A
600	0.033	0.013	N/A
650	0.051	0.02	N/A
700	0.033	0.041	N/A
750	0.02	0.019	N/A
800	0.022	0.011	N/A
850	0.025	N/A	N/A
900	0.024	N/A	N/A
950	0.03	N/A	N/A
1000	0.028	N/A	N/A

RStudio import data for plot

```
#Waldon Hendricks
#This is the data tables captured and imported to rstudio as txt.
create.api <- read.table("NoSQLDBs_createApi_seconds.txt", header = TRUE, sep = "\t")
read.api <- read.table("NoSQLDBs_readApi_seconds.txt", header = TRUE, sep = "\t")
update.api <- read.table("NoSQLDBs_UpdateApi_seconds.txt", header = TRUE, sep = "\t")
delete.api <- read.table("NoSQLDBs_DeleteApi_seconds.txt", header = TRUE, sep = "\t")

#This is the variables per table shown as the values for interpretation.
create.cassandra <- (create.api$CassandraDB_sec)
create.dgraph <- (create.api$Dgraph_sec)
create.mongo <- (create.api$MongoDB_sec)
create.redis <- (create.api$RedisDB_sec)
read.cassandra <- (read.api$CassandraDB_sec)
read.mongo <- (read.api$MongoDB_sec)
read.redis <- (read.api$RedisDB_sec)
objects <- (create.api$Objects)
update.cassandra <- (update.api$CassandraDB)
update.mongo <- (update.api$MongoDB)
update.redis <- (update.api$RedisDB)
delete.cassandra <- (delete.api$CassandraDB)
delete.mongo <- (delete.api$MongoDB)
create.nosqldbs <- table(create.cassandra,create.mongo,create.dgraph,create.dgraph,create.redis)
read.nosqldbs <- table(read.cassandra,read.mongo,read.redis)
update.nosqldbs <- table(update.cassandra,update.mongo,update.redis)
delete.nosqldbs <- table(delete.cassandra,delete.mongo)

#creating plots

#create  plot
createX <- subset(create.api,select = c(Objects))
createY <- subset(create.api,select = c(CassandraDB_sec,MongoDB_sec,RedisDB_sec,Dgraph_sec))
myPch= (1:5)
matplot(createX,createY,type = c("o","o","o","o"),pch = myPch, col = myPch,xlab = c("Number of Objects"),ylab = c("Duration in Seconds"))
legend("topleft", legend = c("CassandraDB","MongoDB","RedisDB","Dgraph"), pch=myPch, col = myPch)
```