AN EVALUATION OF NON-RELATIONAL DATABASE MANAGEMENT SYSTEMS AS SUITABLE STORAGE FOR USER GENERATED TEXT-BASED CONTENT IN A DISTRIBUTED ENVIRONMENT

by

PETRUS DU TOIT

submitted in accordance with the requirements for the degree of

MASTER OF SCIENCE

In the subject

COMPUTER SCIENCE

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: DR WYNAND VAN STADEN

10 July 2016

ABSTRACT

Non-relational database management systems address some of the limitations relational database management systems have when storing large volumes of unstructured, user generated text-based data in distributed environments. They follow different approaches through the data model they use, their ability to scale data storage over distributed servers and the programming interface they provide.

An experimental approach was followed to measure the capabilities these alternative database management systems present in their approach to address the limitations of relational databases in terms of their capability to store unstructured text-based data, data warehousing capabilities, ability to scale data storage across distributed servers and the level of programming abstraction they provide.

The results of the research highlighted the limitations of relational database management systems. The different database management systems do address certain limitations, but not all. Document-oriented databases provide the best results and successfully address the need to store large volumes of user generated text-based data in a distributed environment.

Key words:
Relational Databases, Database Performance Measurement, Distributed Databases, Column-Oriented Databases, Key/Value Databases, Database Benchmarking, Database Management Systems, Horizontal Scalability.

ACKNOWLEDGEMENTS

At school a teacher once told me that knowledge is power, and so a lifelong pursuit started.

I would like to thank my supervisor, Dr. Wynand van Staden, for all his support and guidance throughout this journey.

I would like to thank my parents, Willem and Hanna du Toit, for teaching me to work hard for what I want, to aim high, to persevere and to finish what I start, to always dream big and finally for providing me with what I needed in life. Thank you for always believing in me.

To my brother Johannes and my sister Johanni, thank you for the support, encouragement and being there when I needed you.

DECLARATION

Name: Petrus du Toit _____

Student number: 4912 779 9_____

Degree: M.Sc. Computer Science_____

Exact wording of the title of the dissertation or thesis as appearing on the copies submitted for examination:

AN EVALUATION OF NON-RELATIONAL DATABASE MANAGEMENT SYSTEMS AS SUITABLE STORAGE FOR USER GENERATED TEXT-BASED CONTENT IN A DISTRIBUTED ENVIRONMENT

I declare that the above dissertation/thesis is my own work and that all the sources that I have used or quoted have been indicated and acknowledged by means of complete references.

_____ _____

SIGNATURE DATE

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

ACID: Atomic, Consistent, Isolated, Durable

BASE: Basically Available, Soft state, Eventually consistent

CAP: Consistency, Availability, tolerance to network Partitions

DBMS: Database Management Systems

High availability: Availability to query and store data regardless of hardware failures, maintenance of schema updates to the database

Horizontal scalability: The capability to add additional hardware to the data storage and the capability to manage data across the additional hardware

JSON: Javascript Object Notation

OLAP: Online Analytical Processing

OLTP: Online Transaction Processing

ORM: Object Relational Mapper

Programming abstraction: The closeness of presenting data in the database as to the presentation in a programming language

RDBMS: Relational database management system

User generated data: Unstructured data consisting of text which is created by users online

CHAPTER 1

INTRODUCTION

Non-relational database systems promise improvements over relational database systems when storing large amounts of unstructured data in distributed environments. In order to understand the improvements non-relational database systems offer when compared to relational database systems, an understanding is needed of how different data models are used and how to measure their successfulness in addressing the limitations of relational database systems.

This research identifies the limitations of traditional databases when storing large amounts of unstructured data, and presents alternative non-relational database systems that have been developed to address these limitations. To determine whether these alternative databases address the limitations when storing large amounts of data, this research proposes a method to measure the successfulness in addressing the identified limitations. The results captured for this research are then interpreted and conclusions are presented.

The rest of this introductory chapter is structured as follows:

Section 1.1 provides background to the research.

Section 1.2 presents the problem statement of this research.

Section 1.3 discusses the research questions including the primary and secondary research questions.

Section 1.4 discusses the research objectives.

Section 1.5 is the thesis statement of this research.

Section 1.6 discusses the delineations and limitations of this research.

Section 1.7 explains the significance of this research. And finally,

Section 1.8 gives the chapter overview for the rest of this research.

1.1 Background

The Relational Database Model was developed in the 1970s as a way of storing information, mainly in a single location (Codd 1970:377). The need for the model was driven by the data store requirements of the time and although the relational data model is best suited for transactional-based business applications, it is not ideal for all data storage cases (Stonebraker 2008). The internet and mobile computing have made it easier for users to generate and consume large volumes of information, which is not fit for the traditional relationship model. The new data requirements that have emerged are more prone to large amounts of read access in relation to data updates, which, in most cases, do not have to occur synchronous. Data generated by users on the internet is typically unstructured and non-transactional, which has not been considered with the development of the Relational Database Management System (RDBMS). Due to the volume of data and, more importantly, the type of information being generated today, alternative data models are needed (Hewit 2011:5).

One of the problem areas that exists is the need to store data distributed over multiple database servers and hence the requirement for horizontal scaling of data storage. The large volume of data generated by internet users cannot be stored on a single server and this storage limitation has to be addressed in some way. Furthermore, the internet, Virtual Private Networks (VPNs), Wide Area Networks (WANs) and even Local Area Networks (LANs) have made it possible for organisations to have data stored at different geographic locations.

The decisions for data storage architecture of organisations are directly influenced by the way data will be stored and used, and it has become a case of the data model dictating the architecture (Lai 2009). An example of architectural decisions being dictated by data storage is having users at remote sites that require data access. In order to improve the response time and to eliminate the communication overhead of remote sites, a local copy of data is stored and data synchronisation is achieved through replication (Wiesmann, Pedone, Schiper, Kemme & Alonso 2000). Although RDBMS systems are sold as one solution for all, on a very large data scale they have scalability shortcomings when storing data over multiple database servers. According to Rys (2011:50), additional application complexities need to be

considered when implementing a distributed architecture. Since processing and data are distributed, additional transactional consistency needs to be added. Implementation of an RDBMS in a distributed architecture makes the requirement of strong consistency over multiple partitions difficult. Database functions like distributed querying, synchronous processing and transactions will run into availability and scalability issues (Rys 2011).

Another problem encountered in data warehousing environments is the management of large volumes of data, which can potentially be a rich source of knowledge. Data warehousing requires data mining and data analysis capabilities in order to extract information. The extraction of information over a large set of unstructured data becomes difficult, as it is a computational intensive operation. Data warehousing and data mining, which include the effective querying, analysis and representation of data are, therefore, less suited for the RDBMS model. An example of where RDBMSs are less suited is in Online Analytical Processing (OLAP) of data, which requires data to be accessed as a stream. Stream processing requires data only to be committed eventually, but it is essential for real-time processing, which mostly needs to take place in main memory (Seltzer 2005).

A further problem area is the mismatch of data presentation in the database and the programming language abstraction. Advances in programming language towards object-oriented languages mean that data manipulation and data representation requires some sort of abstraction. Object-oriented and object-relational databases allow the storage of objects directly, but they do not scale up to high transaction volumes and high user volumes (Leavitt 2000), and they have performance limitations (Subramanian & Krishnamurthy 1999). Data in RDBMSs is accessed through a query language known as the Structured Query Language (SQL) and is based on a subset of relational algebra (Codd 1971). Programming languages and development frameworks usually implement abstractions to the database to hide the use of SQL through Object Relational Mapping (ORM); for example, the ADO.Net Entity Framework for .Net 4.0[1] and the active record pattern for Ruby on Rails[2]. The data model is, therefore, abstracted from the programming model.

[1] Information available at http://msdn.microsoft.com/en-us/library/bb399572.aspx
[2] Available online at http://guides.rubyonrails.org/getting_started.html

Alternative Database Management Systems

To address these problem areas, various alternative Database Management Systems (DBMS) exist and promise improvements over the limitations that RDBMS face when managing large volumes of unstructured data in distributed environments. These database systems use non-relational data models and are mostly schema-less implementations that provide scalability over multiple servers. Addressing the limitations is achieved by allowing a certain degree of stale data, which can be regenerated easily. This philosophy is known as the BASE philosophy, which refers to *Basically Available* (data may be stale), *Soft* state (easily regenerated data) and *Eventually* consistent (data committed to disk at some stage) (Brewer 2012:24).

Advances in CPU, memory, network speed and secondary storage have made the cost of hardware cheap (Stonebraker 2008). Consequently, it has made horizontal scaling of data over multiple database servers hosted on commodity hardware easier and more feasible.

These alternative non-relational database systems also provide simple operations to programmers to access and manipulate data directly without adding additional abstraction layers and, therefore, reducing programming cost.

Unstructured user generated text data are generated daily on internet web pages and consists of anything from news articles to blog entries. This information is viewable to users in web browsers. Storing this type of data can provide can provide rich data for data mining applications and would be beneficial for organisations in finding trends in events or gauging popularity of items based on how often it is mentioned on web pages. The volume of this data however will make it impossible to store on single servers. The reason for this research is to find out how this data can be stored in non-traditional RDBMS.

1.2 Problem statement

The alternative non-relational database systems that exist use different data models to store data and promise improvements over relational database systems in different ways. They promise capabilities to store and manage unstructured data in distributed environments and allow a closer level of programming abstraction to fit the programming model.

To determine if non-relational database systems deliver on their promise to address the shortcomings of RDBMSs meaningfully, a mechanism is required to test the veracity of the promises. This mechanism needs to test if each of the shortcomings identified in RDBMSs has been addressed and whether it is capable of performing the measurement against different database systems.

In order to understand the improvements non-relational databases make and what their usefulness is, an understanding is required of how each system and data model can be used to:

- Store large volumes of unstructured data, specifically text
- Provide data warehousing capabilities, such as querying, filtering, searching and analysis
- Allow horizontal scalability of data over multiple database servers with high availability in distributed environments
- What level of programming abstraction is provided.

Since the different database implementations use different data models, a method is required to measure and compare the improvements of each database system in a meaningful way. This research addresses this problem by presenting a mechanism to test the different requirements against different database implementations.

The research questions that exist to address the research problem will be considered next.

1.3 Research questions

This section presents the main research question identified to address the research problem and also lists the secondary research questions that are considered to answer the main research question.

1.3.1 Main research question

The primary research question addressed by this research is:

Do the different non-relational database implementations address the limitations of relational databases when storing unstructured, user-generated, text-based content in distributed environments?

1.3.2 Secondary questions

The primary research question is answered by measuring the enhancements identified against each non-relational database implementation, and these measurements are in context of the primary research question. Secondary research questions are also considered to determine the metrics of the measurements, which are required to answer the primary research question.

The secondary research questions that are considered are:

- How do non-relational databases provide data management capabilities like querying, filtering and analysis?
- How do they provide scaling and distribution of data over multiple servers?
- What data mining capabilities do non-relational databases present?
- What level of programming abstraction and interfacing do they provide?

This research proposes a method of testing the veracity of the promises made by non-relational database management systems. In order to answer the research question, this research evaluates non-relational databases on their data ingestion and storing capabilities of user-generated, text-based content, efficiency of data warehousing capabilities that are available, horizontal scalability of data stored over commodity hardware, and programming language abstraction in contrast to the data model used.

This section presents the research question and secondary questions that have been considered to answer the primary research question. The next section describes the research objectives.

1.4 Research objectives

To address the research problem by answering the research questions that were presented in the previous section, the research objectives will be considered next.

The objective of the research is to answer the research question by evaluating current non-relational database management systems as suitable data stores for unstructured, user-generated text-based content in distributed environments. The research focus is on large scale data stores, which require implementation on distributed architecture and evaluates the current non-relational database implementations against:

1. The ability to store and manage user-generated content consisting of text
2. Searching and analytic capabilities for data warehousing purposes without any data manipulation normally required for warehousing
3. Scalability in terms of the volume of data that is stored over multiple distributed servers
4. The query processing and programming language abstraction.

According to Cooper, Silberstein and Sears (2010), most database models can be compared qualitatively, but comparing their performance is more difficult. Since the various models are developed for specific requirements, performance is normally reported for specific workloads, and comparison is difficult when different workloads are used. This research presents a mechanism to compare the different databases using the same workload on the same servers to provide a more meaningful comparison.

Therefore, the objectives of this research are:

1. To determine the suitability of current non-relational databases for addressing the need to manage unstructured user-generated text-based content
2. To identify shortcomings not currently addressed by non-relational database implementations, which can be addressed in further research
3. To determine which data management systems allow for closer programming language integration without the need for extra levels of abstraction, which could slow down processing and analysis

4. To develop a data presentation model for user-generated content that is non-relational, and to construct a test framework for testing data analysis and querying over distributed data stores.

This section has discussed the objectives of this research. In order to achieve these objectives the thesis statement of this research is presented next.

1.5 Hypothesis

Non-relational database implementation promises improvements over relational databases for storing large volumes of unstructured data in distributed environments by implementing the following features:

1. Schema-less storage of unstructured user-generated content in a distributed environment
2. Fast reads on large volumes of data in a distributed environment
3. Data analysis and data mining capabilities in a distributed environment
4. Fast data manipulation on large datasets with minimum overhead through a programming interface
5. Quick and easy scalability of data storage over commodity hardware.

Measuring how successful these features are implemented through a set of benchmarking experiments can provide information on the successfulness of each data model and database system. By implementing a benchmarking framework, insight is gained into how each data model implements these features and what shortcomings may exist. By using the same input dataset and the same tests for each database, a meaningful comparison can be made between the different data models implemented.

The hypothesis, therefore, is:
Non-relational database management systems address the limitations of relational database management systems (RDBMS) for storing large amounts of unstructured, user-generated text-based content in a distributed environment.

The research presented here is designed to test this hypothesis through a set of experiments and will be discussed in the chapter on research design and methodology (Chapter 5).

This section has presented the hypothesis of this research. The next section will discuss the delineations and limitations of this research.

1.6 Delineations and limitations

Performing this research to address the research problem had certain limitations, which are discussed in this section.

The research was conducted using only open-source implementations of non-relational database management systems, which are freely available and were installed with default settings with no additional configuration enhancements or optimisations. Numerous data models and open-source implementations exist, but this evaluation was conducted on the main database models presented in chapter 4, namely column-oriented databases, document-oriented databases and Key/Value databases. For each of the three database models identified for this research, only the two most popular implementations were evaluated, namely:

- HBase and Cassandra (column-oriented databases)
- MongoDB and CouchDB (document-oriented databases)
- Redis and Voldemort (Key/Value databases).

Although other database management systems exist, like graph databases, this research focuses on non-relational databases and graph databases are highly relationship-oriented. Furthermore, the selected database models are presented as suitable alternatives to relational models by presenting enhancements to address the limitations listed in the problem statement section of this chapter.

The approach followed by this research, as described in the research design chapter (Chapter 5), provides a standard input dataset for testing each of the database implementations, which consists of user-generated text-based data. This dataset consists of only the text representing documents retrieved from the Web through a Web crawler. The input data used for the research was retrieved from the internet from publically available websites, forums and blogs. The information is not analysed in any form, as this research is simply interested in the storage, retrieval and manipulation capabilities of each database system. This process is described in detail later in the research design and methodology chapter.

The different database system installations required for this research were done from a data warehousing point of view and evaluated in terms of horizontal scalability of data storage on commodity hardware in a distributed environment, data mining and analysis over distributed implementations, and the abstraction overhead added by data manipulation and access through programming interfaces.

Scalability of data and redundancy was tested on low-cost commodity servers, implemented on a Linux stack. Scalability of data was considered in terms of the volume and size of data, and the ability to distribute data over multiple servers, not in relation to the volume of queries that could be handled. In order to provide full control over the environment the test was conducted on a dedicated set of hardware. The use of online virtual servers at hosting companies and cloud-based services was considered, but the decision was made to use dedicated hardware, which was not shared by other unknown users who could potentially interfere with performance.

This section has discussed the delineations and limitations of this research. The significance of the research will be discussed next.

1.7 Significance of the study

The internet has evolved into a media platform, which allows users to generate content at free will. The explosion in handheld devices, such as smartphones and tablets has made the creation and consumption of data even easier. Anyone is potentially a publisher and as a result, in the explosion of data being generated and the nature thereof, the traditional RDBMS is showing its age. Although still of importance in the business domain (for which it has been created), which requires transactional data processing, it has shortcomings for a new generation of data storage requirements. Organisations and businesses are also generating more data and not all data requires relational storage. The large amounts of data being generated also exceed the storage capabilities of large servers and require data to be distributed closer to users.

New data requirements lead to the creation of new data storage models and implementations, mostly out of specific requirements or to solve specific problems. Different data storage

models exist, and various database management system implementations of these models have been developed, each presenting enhancements over RDBMSs.

This research is a first step towards determining if current non-relational models and database implementations are successfully improving on some of the limitations of relational database systems when storing large volumes of data in distributed environments. By evaluating the different non-relational implementations available and the models they use, their weaknesses and strengths in storing unstructured user-generated content in distributed environments can be identified. The knowledge gained can be applied in the development of new data management models in the future. This section has discussed the significance of the research and why it is important. The last section will provide an overview of the chapters of this research.

1.8 Chapter overview

This chapter provides an introduction to the problem and why this research exists. The rest of this research entails the following chapters:

Chapter 2 introduces the relational database model, the relational database management systems and the implications of the constraints imposed by the relational model.

Chapter 3 discusses the need for alternative database management systems.

Chapter 4 describes the non-relational database systems that exist. It provides an overview of how they differ and what data models they use. It also discusses the database management systems evaluated in this research.

Chapter 5 discusses the research design and methodology followed in this research.

Chapter 6 presents the results of the research performed.

Chapter 7 presents the research findings.

Chapter 8 provides a conclusion to this research.

CHAPTER 2

RELATIONAL DATABASE MANAGEMENT SYSTEMS

Modern relational database management systems (RDBMSs) are built on the relational data model, which is introduced in this chapter. This chapter considers the relational data model and RDBMSs in order to understand how their design has led to the constraints they present in modern applications.

This chapter describes relational database management system concepts, why relational databases exist and provides an overview of why there are limitations. The different areas where limitations exist and the reasons for the limitations are highlighted. Understanding these limitations provides clarity on the proposed improvements of non-relational database management systems over relational database management systems. Non-relational database management systems are discussed in later chapters.

This chapter is structured as follows:

Section 2.1 provides an introduction to relational databases.

Section 2.2 describes the relational data model and the reasons it exists.

Section 2.3 discusses the constraints that dictate the relational model and design of relational database systems.

Section 2.4 discusses the architectural implications of the relational databases and the issues that arise with scalability, data management and programming abstraction. Issues that have an impact on performance are also discussed.

Section 2.5 provides the conclusion to this chapter.

2.1 Introduction

In 1970, Codd (1970:377) published a paper on the relational model of data. This work became the foundation of modern relational database management systems (RDBMSs). The main commercial RDBMSs available today (Oracle, Microsoft SQL Server, MySQL) all trace their roots back so System R (Astrahan, Blasgen, Chamberlin, Eswarin, Gray, Griffiths, King, Lorie, McJones, Mehl, Putzolu, Traiger, Wade & Watson1976) from the 1970s (Stonebraker, Bear, Centitemel, Cherniak, Ge, Hachem, Harizopoulus, Lifter, Rogers & Zdonik. 2007:1150).

Codd (1970:377) presented the relational model to allow data independence from applications, which would allow changes to data without affecting the applications using it. This was especially useful in shared data banks, allowing different applications to use the same data in different ways. At that time, most formatted data systems stored data in tree structures in files (McGee 1969) or used a network model (Bachman 1965), which made data access difficult, as it was dependent on data access paths (Codd 1970:379). The relational view of data presented data in a more natural way by using the mathematical notion of sets with data viewed as tuples and thereby creating a table structure for a certain domain or relation (Codd 1970:379).

By using sets to represent relations, it is possible to do operations like permutation, projection, join, composition and restrictions on sets or relations (Codd 1970:383-385). These operations are defined in relational algebra, which is a collection of operations that allows data manipulation derived from mathematical set notation (Elmasri & Navate 1994:153). Modern database management systems implement these operations in the Structure Query Language (SQL), which through *select*, *join* and *where* statements allows for data access and manipulation.

The relational view of data also provided redundancy and consistency support on data (Codd 1970:386-387). Codd (1985) presented 12 rules for defining a fully relational database, which describes the constraints required in implementing a fully relational database system. Implementation of these rules in database management systems, however, presents limitations for modern database management requirements. As highlighted later by Seltzer (2005:52), the main reason for developing the model was to separate the data storage from

the applications, since changes to the data store required developers to change applications. The separation of application logic from the data resulted in the development of a language for managing data through queries, which became known as the Structured Query Language (SQL).

Database management systems (DBMSs) of the 1970s and 1980s mostly focused on the processing of business data, which relied on the transaction concept. Transactions on data describe the change in state from one value to another. Transactions preserve consistency in these changes, and they are required to be atomic and durable. This means that a transaction either happens or not, and when completed, the state is preserved (Gray 1981). The processing of business data in a transactional manner is referred to as Online Transaction Processing (OLTP) (Stonebraker & Catell 2011). Traditionally, OLTP databases were many times larger than main memory and data had to be stored somehow. Several features were built into DBMSs to address this issue, such as on disk data structure storage for tables, log-based recovery, record locking for concurrent querying and buffer management (Harizopoulos, Abadi, Madden & Stonebraker 2008). Although these features are required for critical data applications that need redundancy and consistency, they add to the complexity in storing data in distributed environments.

RDBMSs are commonly sold as a solution for all data storage needs. In order to differentiate markets and to address niche markets, vendors expanded the functionality of their RDBMSs. This has caused RDBMSs to have a large feature set of which only small fractions are used by different applications (Seltzer 2005:52). The complexity increased the dependency on specialists and often clients pay for features that they do not require (Seltzer 2005:52). The result is that RDBMSs overdeliver on the core requirements of most data storage needs. The relational database model was created out of the need for business data processing.

This section has provided an introduction to the relational database model and touched on some of the functionality, which is used in the arguments for the need for alternative data storage models. The next section describes the relational data model in more detail. It also provides insight into how data is stored. It is useful in understanding why the relational model may not be the best solution in all cases.

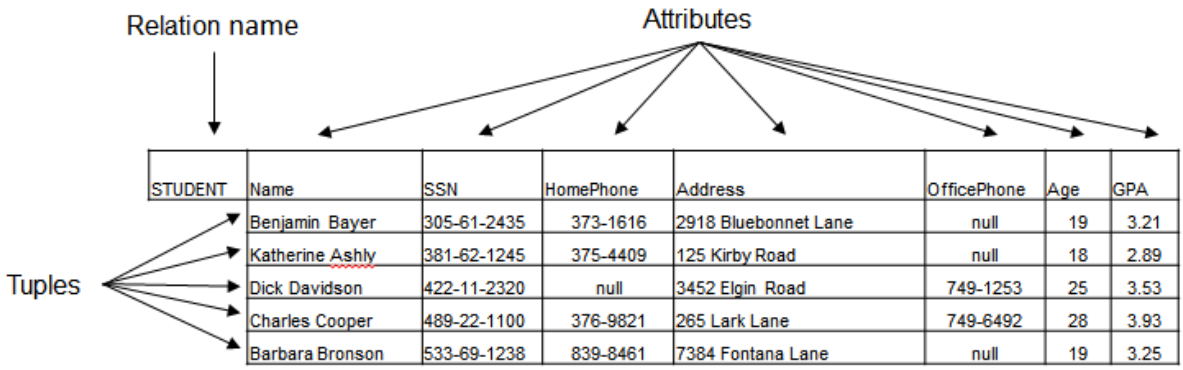
2.2 The relational data model

The previous section has provided an introduction to the relational database model. This section describes the database model in finer detail in order to understand the implementation of relational database management systems.

From the introduction of this chapter it was learned that the relational model presents data in a mathematical notion (Codd 1970:379). The relational data model represents data as a set of relations. A relation can be viewed as a table, which consists of a number of rows, each representing a collection of related data types. To assist with interpreting data in the relational model, tables and columns have names. In relational terminology, a table is called a relation, a column name is an attribute and it forms part of a specific domain. The value that can appear in each column presents the domain of an attribute and is defined by a data type or format. A row is known as a tuple. (Elmasri & Navate 1994:138).

The domain of an attribute is defined by a data type, a name and a format, which presents the smallest element of the relational model. Each attribute of a relation is described by a domain. A domain is the formal specification of atomic values, which can be assigned to an attribute, like a string of alphanumeric characters, for example. Domains need to be defined beforehand as part of the database schema. A schema defines the database in terms of what relationships may be stored as well as the definition of each relationship. The schema defines a set of relations and each relation needs to be defined through a set of attributes. The relational model requires these sets to be defined beforehand and as such dictates a fixed schema, which may not be changed easily.

When defining the schema beforehand, a situation may arise where no data is available upon storage. This can be handled by a special NULL element, which allows the storage of an attribute in a tuple if no value is available (Elmasri & Navate 1994:138-139). A tuple (or row) consists of a set of domains (or data structures) describing attributes of a relation. Thus, a relation consists of a set of tuples, as can be seen in Figure 2-1.



STUDENT	Name	SSN	HomePhone	Address	OfficePhone	Age	GPA
	Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	null	19	3.21
	Katherine Ashly	381-62-1245	375-4409	125 Kirby Road	null	18	2.89
	Dick Davidson	422-11-2320	null	3452 Elgin Road	749-1253	25	3.53
	Charles Cooper	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
	Barbara Bronson	533-69-1238	839-8461	7384 Fontana Lane	null	19	3.25

Figure 2-1: The attributes and tuples of a relation (Elmasri & Navate 1994:140)

The relational model requires each tuple to present a unique relation in the schema. In order to access a relationship, each atomic value should be accessible by a combination of the relation name (table name), the attribute (column name) and a unique identifier of the tuple (or row). The relational models have different types of unique identifiers, which are also known as keys. Candidate keys are a single field or combination of fields that can uniquely identify each tuple in the relation. Primary keys are candidate keys that are the most appropriate or main reference key of a tuple. Each tuple must have a unique or primary key to describe it (Elmasri & Navate 1994:143-145).

The combination of all the attributes in a tuple, along with the primary key, must describe a unique entity in the schema and is known as the super key of that relationship. Secondary keys are keys that can uniquely identify a tuple, but is not the selected primary key. In the student example in Figure 2-1, the social security number (SSN) can be used to uniquely identify a student and in this case, it can be the primary key. However, a combination of the name and address and GPA could theoretically also provide access to a unique student and can be used as a secondary key.

A relational schema also adheres to a set of integrity constraints. Since a schema can have multiple relations, primary keys can be used to reference other relations to provide data integrity and these are known as foreign keys (Elmasri & Navate 1994:145).

Relational algebra is a collection of operations that provides a way of manipulating relational data. There are two groups of operations. The one group is derived from mathematical set theory where each relation is defined as a set of tuples. These operations include *union*,

intersection, *difference* and *Cartesian product*. The second group is specific to relational databases and the simplest operations are *select*, *project* and *join*. *Select* is used to select a subset of tuples from a relation that satisfies a condition, and *project* defines the attributes (or columns) to return (Elmasri & Navate 1994:153-165).

This section has described the relational data model. The next section looks at constraints the relational model enforces on database management systems.

2.3 Constraints imposed by the relational model

The relational data model described in the previous section imposes certain constraints on how relational database management systems (RDBMSs) are implemented. As a result, RDBMSs are designed from a top down point of view, based on predefined semantics (Brewer 2005). This section discusses the constraints imposed by the relational model when implementing a relational database management system.

The relational model has been designed to store data for business processing and needs to support transactions. In relational database terms a transaction is defined as the changing of data from one state to another. To ensure transactional integrity in RDBMSs, transactions need to be *Atomic*, *Consistent*, *Isolated* and *Durable* (Gray 1981:1) and this has become known as the ACID properties. When updates occur, every update in the transaction must succeed (*Atomic*), all viewers must view the same resulting state (*Consistent*), transactions must occur separate of one another (*Isolated*) and once a transaction succeeds the changes cannot be lost (*Durable*) (Hewit 2011:7).

Transactions must transform the state of data in such a way that it adheres to the ACID properties (Gray 1981:1). The implementation of ACID properties in RDBMSs has made scalability of data over multiple servers difficult, as it is difficult to ensure a consistent view over distributed database servers (Hewit 2011:7). ACID properties in RDBMSs are in conflict with the requirements of consistency, availability and tolerance to network partitions in distributed environments. This is known as the CAP theorem. It states that in shared or distributed data stores, you can at most satisfy two of the requirements of *Consistency*, *Availability* and tolerance to network *Partitions* (Brewer 2000).

Atomicity and durability are beneficial to all systems and do not have any impact on the presence of network partitions. Consistency in terms of ACID properties implies that transactions preserve all the database rules such as keys, while in CAP, it refers to having a single copy of data. In the presence of network partitions, consistency cannot be maintained, as a partition recovery needs to restore ACID consistency across all partitions. Isolation is the core of the problem, as ACID isolation can only operate at most on only one side of the partition (Brewer 2012).

Modern data storage requirements make the storage of data on a single machine impossible and require a distributed architecture for storage. The ACID properties required for transactions are implemented in RDBMSs and make it less ideal for storage in distributed environments since it cannot satisfy all the CAP requirements. RDBMSs provide workarounds for addressing this limitation through federated databases, and these will be discussed later in section 2.4.1 of this chapter.

In the next section, we look at how the constraints mentioned in this section have implications for the implementation of relational database management systems when they are used to store and manage large volumes of data in distributed environments.

2.4 Implications of the constraints of the relational database model

In the previous section the constraints of the relational model have been introduced. This section provides the details on what these constraints impose on the implementation of relational database management systems. The implications in terms of scalability of data storage over multiple database servers, management of distributed data and programming abstraction are considered for large datasets. The approach followed by RDBMS implementations to address these issues is also described. Finally, the performance implications in terms of data retrieval and storage are considered.

2.4.1 Scalability of data storage

The nature and scale of data generated in modern internet based systems, like search engines and social networks, mean that data cannot be stored on a single database server. As a result, scalability of data storage becomes a requirement. For the purpose of this research, scalability is the degree to which large volumes of data can be stored and distributed over multiple

database servers. This section looks at why the volume of data generated by internet-based systems cannot be stored on single servers and what approaches to scalability exist in RDBMSs.

The following scenario provides an overview of what organisations tend to do when facing data storage scaling issues. These methods will be discussed in further detail later in this section. When an organisation encounters a scenario where data storage needs to be scaled to multiple servers due to the volume of data to store, an architectural approach to scaling data storage can be achieved through partitioning of data (Rys 2011).

Hewitt (2011:3) describes the process, which organisations may follow when scaling of data storage is required and the issues related to scaling are encountered in traditional RDBMSs. The first solution organisations try to employ is vertical scaling: adding better hardware to the solution. When the problem persists, data partitioning over multiple servers is considered, more servers are added and the solution is clustered. Due to the transactional nature of RDBMSs and its adherence to ACID properties, clustering requires data replication in order to address the data consistency required. Data replication brings additional problems to the solution, i.e. ensuring replication takes place, latency issues, and consistency over geological separated sites. Next the databases may be optimised for performance, using stored procedures or indexes for better performance. The applications are also tweaked and modified in an attempt to better performance. Finally, caching layers is employed, which leads to more consistency issues.

The scenario discussed above highlights the processes followed when the need for data scaling is encountered. Scaling data using RDBMS databases in distributed environments can be achieved through the following methods (Vaquero, Robero-Merino & Buyya 2011):

- Clustering
- Caching
- Through the use on alternative database implementations.

Caching and clustering as a means of implementing scalability in RDBMSs are discussed next, while scaling through alternative database implementations is discussed in section 3.2.

From the scenario described earlier, the first method is an architectural approach to scalability in a RDBMS through either functional or data partitioning (Rys 2011:49). Functional partitioning provides scaling by implementing different applications that function separate from one another. It entails the separation of data used by the different functional units in an organisation and storing each function's data in its own database server (Rys 2011). For example, the Finance division may have a different database than Human Resources. The possibility still exists that the volume of data generated by a functional group can still exceed the storage capability of a database server.

Data partitioning splits the processing of application across different data partitions. Data partitioning allows local copies of data at sites closest to the users. The local data partition, therefore, provides local consistency and query processing. Servers (referred to as nodes) are mirrored to provide high availability (Rys 2011). This approach, however, only addresses the requirement of availability to remote users and a master database is still required.

To ensure that the latest data is available at the remote site, replication is often used in distributed RDBMS implementations as a mechanism to achieve scalability (Kemme & Alonso 2010). Replication allows data to be accessed locally in distributed environments by allowing clients to access a local copy of data (Wiesmann *et al*. 2000). Several replication models exist. In *Active* replication, all replicas receive the same client requests in the sequence that they appear and apply this to the local data copy. *Passive* replication is also referred to as *backup* replication where each client sends a request to a primary server, which, in turn, sends update messages to all replicas. *Semi-active* is an intermediate replication model between active and passive models. It uses a leader to make a decision whenever non-deterministic decisions must be made, which push the decision down to followers (Wiesmann *et al*. 2000). This approach does not address the issue of scaling data over multiple servers.

To spread data across multiple servers, clustering is used. Clustering refers to having a cluster of database servers, each of which stores different partitions of data. Clients access the clustered servers without knowing that it is requesting information from several servers (Strauch 2011). The use of commodity hardware on high speed local computer networks allows a "Shared Nothing" architecture where each server (referred to as a node) in a cluster has its own disk and memory. This allows data to be divided into different partitions, both

logically and physically, and then stored on different servers (Pavlo, Paulson, Rasin, Abadi, DeWitt, Madden & Stonebraker 2009).

Sharding of data partitions can also be used to achieve scalability. Sharding data means that data is partitioned in such a way that data typically requested and updated together is stored on the same node in a clustered environment. Load and storage volume are then evenly distributed across the nodes. The disadvantage of sharding is that data on different nodes cannot be joined easily (Strauch 2011:36).
Partitioning of data over different nodes, however, does impact the consistency of data and the agility in making schema changes. This requires additional redundancy measures (Rys 2011:50). Using RDBMSs to scale data storage across multiple nodes complicates the ability to modify schemas, which becomes difficult or slow to implement (Rys 2011). This means that any schema changes may require some down-time to implement on all nodes.

Although RDBMSs can be implemented in distributed environments, these implementations are not without problems. Transactions make scalability more difficult, as consideration in distributed transactions must be taken. This is normally done through the use of two phase commit. Two phase commit blocks transactions competing for resources, which may lead to clients queuing for resources (Hewit 2011:8). Operations are synchronous and wait for a response to queries from other nodes, which may cause queries to fail if a response is not in time (Rys 2011).

When partitioning and clustering are not sufficient, the second approach, as highlighted by the scenario described at the beginning of this section, is the use of caching. Caching entails the storage of frequently accessed data in main memory. Memory caches are in-memory databases, which are queried first to lessen the burden on databases by having data resident in memory and eliminating disk access to retrieve data (Fitzpatrick 2004). In order to speedup frequent data queries, caching systems such as Memcached (Fitzpatrick 2004) are often used. When a user request is made, an in-memory cache is first queried to see if the data is present before accessing the database. Distributed caching provides access across multiple database nodes in a clustered environment (Vaquero *et al*. 2011). Caching adds to the complexity by adding an additional level to the architecture.

Employing clustering and partitioning of data or caching as a means of achieving scalability in distributed environments creates additional difficulties and has performance penalties. These problems often lead to the use of a third approach, which is the use of alternative database implementations designed to operate in distributed environments. The alternative database implementations that exist are discussed in chapter 3.

This section has described the problems when storing large volumes of data in distributed environments using RDBMSs. The next section presents the implications the relational database model has when managing and analysing data in distributed environments.

2.4.2 Management of distributed data

The previous section has described the implications of storing large volumes of data in distributed environments using RDBMSs. This section describes the management of large volumes of data in distributed databases and the implications of the constraints imposed by the relational model on data analysis and warehousing of large volumes of distributed user-generated data. The implications on data analysis and warehousing are described first before highlighting the limitations of RDBMSs as data warehouses. New approaches to distributed data analysis are described in section 3.3.

Since a database is defined as a collection of interrelated data of different types, which is organised to be accessed conveniently, a database management system (DBMS) is software, which allows the creation and maintenance of databases (Ralsten, Reilly & Hemmendinger 2003:517-519). In a distributed environment, this management of data becomes more difficult. A distributed database management system (DDBMS) allows the management of multiple databases distributed over a network and makes the processing of data from these databases transparent to users (Özso & Valduriez 1996). Therefore, the role of a database management system is firstly to manage and maintain data, and secondly to provide data access to applications. Data analysis and warehousing are applications to data storage. They provide insight to organisations into the data stored in their DBMS. These applications are considered next.

Data warehousing is a collection of technologies known as *decision support systems* that enable knowledge workers to make decisions. Data warehouses enable online analytical

processing (OLAP), which allows users to query data over multiple dimensions (Chaudhuri & Dyal 1997). Data accumulated over periods of time is often analysed to gain insight into the data (Jacobs 2009). User generated text-based data can potentially provide valuable data for analysis. As such there is a need to provide data analysis for data warehousing exists. The limitations presented by RDBMS will be considered next, as well as how they are dealt with in current RDBMS implementations. These limitations are relevant to the research in determining the type of data queries data warehousing applications generally face and are incorporated into the research design in chapter 5.

OLAP differs from Online Transaction Processing (OLTP) for which traditional relational databases have been developed originally. OLTP databases are focused on transaction processing on smaller datasets and consist of short atomic isolated transactions. Consistency is crucial in these databases. OLAP, in contrast, encompasses ad hoc query-intensive processes, which consist of scans, joins and aggregates. Scans refer to reading data from a certain point, like a key. The relational model can also store related data in different tables and in a warehousing environment; the two tables need to be joined to return the required data. Aggregations are calculations on a specified dimension, like counting and ranking. In a data warehouse, data is typically modelled in a multidimensional array. These dimensions are attributes of data like time of sale, sales person and district.

Some of the OLAP operations include *rollup*, *drill-down*, *slice-and-dice* and *pivots* (Chaudhuri & Dyal 1997). In its simplest form, *pivots* allow the selection of two dimensions of a multidimensional array and performing an aggregation to a measure on them (Chaudhuri & Dyal 1997). *Rollup* and *drill-down* are related to pivoting where *rollup* allows an additional group-by on a dimension and *drill-down* is the reverse of a *roll-up* (Chaudhuri & Dyal 1997). *Slice-and-dice* is used to reduce the dimensionality of data (Chaudhuri & Dyal 1997).

Typically data warehouses are constructed by having a copy of the master database against which data queries for analytical purposes can be executed (Stonebraker & Cetintemel 2005; Jacobs 2009). Since data needs to be modelled first, there is a process to get the data into a data warehouse, typically from different sources. Data needs to be extracted from the data sources whereafter it needs to be transformed into the data warehouse structure. Typically data warehouses require a flat structure on the data to eliminate the use of expensive joins

when querying data. Transformation entails data cleaning, scrubbing and auditing since it may come from different sources. The data is then loaded into the data warehouse (Chaudhuri & Dyal 1997).

RDBMSs support data warehousing, but traditional RDBMS techniques like dimensional mapping and cube-based OLAP are either too slow or too limited to provide adequate analysis of large datasets (Jacobs 2009:29). In the multidimensional array view of data, data is modelled as a set of numeric measures that form the object of analysis. Each numeric measure depends on a set of dimensions. Dimensions provide the context of the measure and describe a set of attributes of the data. This model is highly dependent on aggregations as one of the operations. Other operations implemented include ranking and selection (Chaudhuri & Dyal 1997).

Although RDBMS has data warehousing and analysis features implemented, implementing data warehousing environments on RDBMSs for large volumes of data have limitations in terms of performance and these limitations are discussed next.

RDBMSs have been designed for efficient transaction processing: it is easier to add or insert data into a relational database than to extract it because extraction is normally done on very small datasets (Jacobs 2009:29). Data warehousing applications, however, are more focused on analysing and thus reading data. The transactional implementation of RDBMS also means that data is stored in the sequence the transactions have occurred. Retrieving data for analysis, however, means that data is retrieved non-sequentially. This has an impact on performance, especially if datasets have grown larger than what can be stored in the memory of the host machine. Accessing data from a hard disk, especially randomly, has huge performance costs (Jacobs 2009).

Data warehousing and OLAP entails analysis of data by executing procedures or queries over large scale data repositories. There are two main problems in extracting data. Firstly, data is normally stored in different formats and needs to be combined into a format that can be interpreted. Secondly, the information needs to be extracted, processed and transformed into a presentable form, such as graphs, to be of value as business intelligence (Cuzzocrea, Song & Davis 2011). This entails additional processing, which comes at a cost.

Large scale data analysis in businesses is done on enterprise data warehouses by using business intelligence software to generate reports. These reports utilise aggregation functions, such as sum and average calculations over the data to extract knowledge. Data warehouses are implemented as centralised systems from which management reports can be created. The centralised approach and large volume of data makes data warehouses a very expensive resource. In large organisations, different business functions create data in their own data stores. As a result, data is imported decentralised; this is in contrast to centralised data warehousing implementations. Several differences between traditional data warehousing implementations and decentralised data sources exist (Cohen, Dolan, Dunlap, Hellerstein & Welton 2009), which are considered next.

First of all, in traditional data warehousing, new data sources need to be integrated and cleansed carefully before they are made available, a process that can be time consuming. The rate at which new data is generated requires quick integration into the environment, regardless of the quality. Secondly, traditional data warehouses require long-term planning and design. Modern data analysis in warehouses requires quick ingestion, digestion and output on a growing number of data sources. This means that data continuously changes at a rapid pace and as a result, data analysis must be agile. Thirdly, there is an increase in the sophistication of data analysis methods, which goes deeper than traditional business intelligence. Data analysis is also required over large datasets; therefore, modern data warehousing should provide deep data repositories and the capabilities to execute fairly sophisticated analysis algorithms (Cohen *et al*. 2009).

Using RDBMSs for data warehousing has performance cost due to their transactional nature. Furthermore, it is expensive to implement a data warehouse using RDBMS as it requires long-term planning. Integration of new data sources also tends to be slow, as data extraction and transformation need to be implemented. The nature of OLAP also entails sophisticated computational intensive operations, which impact responsiveness of queries on large datasets.

New approaches to data analysis in distributed environments to overcome the problems with RDBMS have been developed and are discussed in section 3.3.

In the next section, the implications of the relational database model constraints on programming language are presented.

2.4.3 Programming abstraction

In addition to managing data and data structures in a database, DBMSs also provide access to applications and clients to manipulate and request data. Accessing data for manipulation and client requests programmatically is discussed in this section.

The use of RDBMSs has become a very popular database management system and has enjoyed tremendous success in data centres around the world (Seltzer 2005). To address data access and manipulation, the Structured Query Language (SQL) has been developed to manage data in RDBMSs.

SQL is used to manipulate, manage and extract information. It has, among others, been developed to allow transactional information retrieval at a time when COBOL was used. SQL was originally designed and developed by IBM as an interface to System R and was called SEQUEL (Elmasri & Navate 1994:185). It was based on a relational sublanguage defined by Codd (1971), known as “relational algebra”.

Apart from managing relational databases, SQL is also used to manipulate and retrieve data, as stated above. Retrieving data in SQL is done through queries such as SELECT (Elmasri & Navate 1994:193-212), and manipulation is achieved through update statements like INSERT and UPDATE (Elmasri & Navate 1994:212-215).

Data access and manipulation from programs, therefore, require the construction of queries or SQL statements, which are then executed against a database; therefore, adding an additional layer of abstraction, which may differ from the programming paradigm used when constructing an application. The advances in programming language to an object-oriented approach have created a data representation mismatch between programming objects and relational data. Object-oriented programs present data as objects with different properties. In order to store an object in a relational database, each property of the object needs to be mapped to an attribute (column) in the database to be stored. Retrieving data requires a result set from the database to be traversed and each attribute (or column) needs to be assigned to the corresponding property of the object before the object can be used. This process has been simplified through the use of Object Relational Mapping (ORM), but this does require additional processing. This will be discussed in detail in section 3.2.

In addition, one of the major performance bottlenecks in OLTP applications is the added overhead in communication to the DBMS. To avoid unnecessary communication between applications and the DBMS, stored procedures are used to execute logic in the DBMS (Stonebraker 2010:10). Stored procedures are sets of SQL statements and are executed by the database server, which adds additional processing to the database server.
The RDBMS and SQL language provide access to applications to manipulate and query data. Although the RDBMS has been developed to separate the programming language from the data model, in large scale analytical applications the use of SQL has penalties on performance.

The next section explores performance-related problems that RDBMSs have when dealing with large datasets in a distributed environment.

2.4.4 Performance on data access

The relational database model also has implications for data access performance when accessing large volumes of data in distributed environments. This section explores the impact on data access performance by looking at data access overhead created by their storing policies, the impact of data fragmentation and the impact of the normalisation of data.

The first factor that influences performance is the storing policies of RDBMS, which require that data adhere to the ACID properties. Techniques applied to ensure that ACID properties are adhered to have an impact on the performance of databases. The consistency and durability constraints of RDBMSs add additional overhead to the performance of the system, which can be attributed to the following (Harizopoulos *et al*. 2008):

- Logging
- Locking
- Latching
- Buffer management.

These features exist to ensure that in RDBMSs all transactions are consistent and can be recovered after failure. Logging means that all transactions are written to disk twice, as all transactions are also written to the transaction log, which may be used to recover during a

failure. Another feature, locking, is required to ensure that the latest data is available by ensuring that when a transaction is written, the lock manager prevents any updates and blocks other requests until the transaction is completed. This may cause other transactions to wait for extended periods or eventually time out.

In multithreaded databases, latching of data structures is required before they can be accessed, which is a costly operation. A latch is a low-level, short-term lock on memory structures stored in RAM, which prevents concurrent access but does not have deadlock detection (Gray 1978; Gottemukkala & Lehman 1992). Since some data is stored on disk, access is managed through buffer pools, which affect performance by swapping out data between primary and slower secondary memory (Harizopoulos *et al*. 2008). These features, therefore, may give rise to situations in distributed environments where clients wait too long for access to data.

The second factor that has an impact on database performance is fragmentation. According to Özsu and Valduriez (1996), distribution of databases is achieved through fragmentation and replication of data. Parallel or distributed systems range from Shared Nothing (where each database server runs on dedicated hardware) to Shared Memory (where memory is shared between database servers) systems. These include distributed systems that implement shared-disk architecture. Distributed systems need to provide the same functionality transparently as centralised DBMS. The main challenges are in query processing and optimisation, which need to access fragmented data (data spread across distributed systems) and to provide concurrency control.

The locking-based algorithms used may lead to deadlocks in distributed systems. Reliability protocols like two-phase commit (2PC) are used to deal with communication failures in distributed systems by also logging transactions to the transaction log. Replication protocols are used to ensure that when an update transaction terminates, all the physical copies need to be identical (Gray 1978; Özso & Valduriez 1996). A typical protocol is Read-one/Write-all, which waits for all copies to update before the transaction is terminated. However, if a copy fails, the transaction may be blocked. Another approach is the use of quorum-based voting where operations have to collect a quorum of votes before they can complete (Özso & Valduriez 1996). Fragmentation of data in distributed systems, therefore, requires additional protocols to ensure data integrity across multiple databases.

The third factor, which can have an impact on performance, is the normalisation of data. When designing a relational database schema, a process called *normalisation* is followed to ensure that relations are broken into smaller and more desirable relationships, based on their keys (Elmasri & Navate 1994:407). Various forms of normalisation exist in relational database theory. They were designed to prevent update anomalies in data and consistency (Kent 1983). Normalisation of tables in RDBMSs has a performance impact on data analysis. Analysis may be performed by joining multiple tables into a single query.

Poor performance impact on analysis is observed when joining a table containing transactions (which are stored sequentially) with another table (which is accessed randomly). The larger the datasets being joined, the more time needed to access the joined data randomly. The performance decreases even further when additional tables are joined (Jacobs 2009). Fragmentation of data in distributed environments may further complicate joins if data is separated on different databases servers.

This section has presented the factors that influence the data access performance when RDBMSs are used for storing large volumes of data. Using RDBMSs in distributed environments add additional complexity to the implementation of the system, since additional work is required to overcome the limitations of a DBMS designed for a centralised environment, especially when dealing with fragmentation of data.

The implications of the constraints that the relational database model imposes on the implementation of RDBMSs have been discussed in terms of the scalability of data across multiple servers, the data management and data warehousing requirements and limitations when dealing with large volumes of data, the programmability abstraction and programming interaction with RDBMSs, and finally the data access performance implications. The next section provides a conclusion to this chapter.

2.5 Conclusion

This chapter has introduced the concepts of the relational data model and the implementation of this model in RDBMSs. The constraints that the relational database model imposes on the implementation of RDBMS in distributed database systems have also been discussed. The

relational database constraints that exist and the architectural implications of using RDBMSs for store large volumes of user-generated content have been discussed in terms of scalability, data management, programming abstraction and data access performance issues.

The next chapter discusses the need for alternative database management systems.

CHAPTER 3

A NEED FOR ALTERNATIVE DATABASE MANAGEMENT SYSTEMS

The constraints the relational database model imposes on the implementation of database management systems, as discussed in the previous chapter, led to the development of alternative database models. The previous chapter also introduced the relational database model and highlighted some of the limitations, which relational database management systems (RDBMSs) have in storing and managing large volumes of data in distributed environments.

Although RDBMSs provide approaches to address these problems identified in the previous chapter, the associated complexity of these approaches has raised the need to search for alternative solutions that address the limitations. This chapter describes the need for alternative database systems and data models, highlights the programming abstraction limitations of RDBMSs as well as new approaches that exist to handle large volumes of data in distributed environments.

The chapter follows the following structure:

Section 3.1 discusses the need to store alternative data and indicates how different application areas influence the need for alternative databases.

Section 3.2 describes the need for alternative programming abstraction layers and indicates how alternative databases can provide enhancements on application performance.

Section 3.3 provides detail on how alternative databases address the issues discussed in the previous chapter as well as the need for alternative databases discussed in sections 3.1 and 3.2.

Section 3.4 concludes this chapter.

3.1 The need to store and analyse different data models

This section looks at how the data used in modern systems have changed from the data requirements of traditional business applications as a driver for alternative data models. Data generation is no longer limited to desktop computers or computers locked in server rooms. Personal mobile devices, smartphones and even mobile telephones are used to generate data (Seltzer 2005:50). Text services and multimedia messaging and location-based services create new data-generation opportunities (Seltzer 2005:52). With the increase in internet usage and social media in particular, the generation and consumption of information have grown.

The storage of user generated text based data presents certain difficulties for RDBMSs. In the previous chapter the RDBMS has been discussed. This chapter will identify how the type of data, namely user generated text based documents, and the distributed storage required present difficulties for traditional RDBMS. In addition the advances in database models and the distributed nature available by alternative database management systems will also be discussed before actual implementations are presented in chapter 4.

When the relational database model was designed the management of data was done by a few experts. Today the data users are far more empowered to generate the data. The design of systems and data usage are far more dictated by the users and how they use systems. Users on social networking sites can add any type of information or content and as a result, the management and computational needs of data are shifting to a user-centred approach (Badia & Lemire 2011:63).

Social media and user-generated content have different characteristics than those required by businesses. Therefore, alternative approaches exist to find a suitable storage medium. Since users are now enabled to generate any type of data, the data stored can change its form easily as new information becomes available. Traditionally relational databases did not change frequently and were bound by the database schema. With regular changes to the data structure and the type of data stored, most organisations using a RDBMS need to make changes to the database schemas regularly (Badia & Lemire 2011:64), which are both difficult and costly.

In the traditional sense, data was stored in a centralised location and systems implemented on RDBMSs assumed a centralised architecture of the data stored. Modern data requirements assume a more distributed architecture as the volume of data generated cannot be stored on a single database server. Furthermore, in a connected world there is also the need to share data between organisations and systems. Integration and data exchange between different systems become difficult as not all systems share the same view of data (Badia & Lemire 2011:64).

RDBMSs have some limitations when facing challenges with modern data requirements. According to Stonebraker *et al.* (2007:174-176), the following application areas exist, which may not be suited for RDBMSs:

- Text processing
- Data warehouses
- Stream processing
- Scientific and intelligence databases

These application areas drive the need for alternative database models and each of these application areas are considered next. A holistic view of the limitations imposed on these application areas is provided at the end of this section by highlighting the common themes.

3.1.1 Text processing

Text processing entails the storing, indexing and searching of text. Text, especially if generated by users, can be unstructured and have variable lengths. Even though text search engines deal with large datasets, they do not use RDBMSs. Google has developed the Google File System (Chemawat, Gobioff & Leung 2003) as its storage system and Bigtable (Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes & Gruber 2008) for managing large datasets. RDBMSs have been designed on the semantics of ACID transactions, which have to be consistent, but text search requires something different: availability over consistency (Brewer 2005).

Search engines have multiple incoming data streams from Web crawlers, which need to be appended to the current index. Appending to the master index is a once-off event (Stonebraker & Cetintemel 2005). In search engines, updates from crawlers are not available immediately and a certain degree of old data is tolerated to reduce the degree of

inconsistency. The main goal of a text search is to execute queries, which do not cause any updates. Updates are also less frequent than queries and as a result, search engines seldom deal with atomicity or isolation. Durability is easier to achieve, as search engines can easily regain lost data from copies by re-crawling a site (Brewer 2005).

Search queries are normally short phrases or single words while the number of words to consider in any language is large. Searching of text implies tracking and indexing ten million distinct words in billions of documents (Brewer 2005). Search engines also consider the relation of where words occur in documents as well as their relevance to the queries. Another challenge is that a single query can produce thousands of results (Brewer 2005) and, therefore, databases must be optimised for reading.
The problem with using RDBMSs for search engines is that there is a semantic mismatch and in general search results are slow (Brewer 2005). Although current RDBMSs provide searching capabilities, they fall short on performance and availability requirements of search engines (Stonebraker & Cetintemel 2005), and they are not suitable for large scale Web search applications (Brewer 2005).

Due to the large volume of data stored by search engines, data needs to be stored over a large number of machines. The availability requirement of text search engines needs to be achieved through quick recovery and replication in the event of hardware failure. RDBMS uses a transaction log for recovery. In case of a failure, all the transactions are applied from the log in the order in which they have occurred. This can be a time consuming process, which would impact the availability of a text search engine during a data restore (Stonebraker & Cetintemel 2005). The main problem is that during a restore, the database is unavailable for querying. Although re-crawls of websites are also slow, the availability of the search engine is not affected.

This section has highlighted that relational database management systems are not suitable for text-based data and why alternative databases exist to deal with this type of data. The next section discusses how data warehousing influences the requirement of alternative databases.

3.1.2 Data warehousing

Data warehousing provides organisations, as part of their decision support system, with the ability to mine data already collected and stored. Retail organisations store user transaction data, which has the potential for data mining on buying preferences and trends, product popularity and preferences of different geographic locations (Seltzer 2005:53). Data collected as part of an organisation's daily functions is appended only periodically in batches to data warehouses. For mining purposes, information is accessed mostly as read operations and only subsets of all the data available are normally queried (Seltzer 2005:53). Traditional RDBMS implementations are optimised for writing by writing a complete record row to disk. Data warehousing needs to be optimised for reading data since once data is added, it seldom changes (Stonebraker & Cetintemel 2005).

The main difference between data warehousing applications and OLTP applications is that data warehousing access only certain columns and not complete records. This means that data warehousing will be more efficient if data is stored by column, rather than rows. By implementing column-store architecture only relevant columns are loaded when queried and not whole record sets (Stonebraker & Cetintemel 2005).

Reading large volumes of data can also influence the system. Data warehousing and business intelligence require complex queries to be executed, which can have an impact on the response time required for online transaction processing. As a result, enterprises implement data warehouses that contain snapshot data against which complex queries can be executed (Stonebraker & Cetintemel 2005) without affecting the transaction speed of their normal online transaction processing.

In this section, it was highlighted how data warehousing applications differ from traditional OTLP applications and why data warehouses could benefit from an alternative database model. The next section will discuss stream processing and why it would benefit from an alternative database model.

3.1.3 Stream processing

Stream processing entails processing of real-time information as it arrives and can be seen more as a filtering task (Seltzer 2005:54). Mobile devices like smartphones can submit

geographical location information in real time. Messaging from various devices, like smartphones, cell phones and internet connected devices can also provide continuous streams of data. Data feeds like real-time financial feeds, real-time flight tracking feeds and monitoring systems also provide data streams that need processing or filtering.

Stonebraker and Centintemel (2005) declare the main difference between traditional transactional processing (as implemented in RDBMSs) and stream processing is that transactional processing is outbound or pull processing, while stream processing is inbound or push processing. In essence pull processing of traditional RDBMSs requires that data received be committed to storage before any query or processing can occur as required by the isolation property of the ACID properties. In contrast, push processing required for stream processing requires input streams to be processed and results given first, while storage is optional. Stream processing, in most cases, requires the storage of state information, which may include reference data (for lookups), translation tables and historical data. In these instances, storage is eventual consistent.

Steam processing, therefore, has different requirements as implemented by relational databases and requires an alternative database model. The next section discusses the requirements of scientific databases and how they benefit from an alternative database model.

3.1.4 Scientific and intelligence databases

Scientific activities collect large amounts of data through sensors (such as satellites and microscopes) or can generate large amounts of data through high-resolution simulations (Stonebraker & Cetintemel 2005). It was estimated that the Large Synoptic Survey Telescope (LSST) planned for construction in Chile will record 30 terabytes of images every day while the Large Hadron Collider (LHC) will generate approximately 60 terabytes of data per day (Bryant, Katz & Lazowska 2008). From the current data available online[3], the LHC generates 1 petabyte of data per second. After filtering data and storing only data of interests, the LHC experiments produce about 15 petabytes of data per year that need to be stored.

These very large datasets need to be analysed and queried efficiently. As a result, the indexing of these databases needs to be highly efficient. Furthermore, the analysis of data

[3] Available at http://www.lhc-closer.es/1/3/12/0

requires efficient and accurate aggregation functions. The nature of the application area requires archiving, lineage and error propagation techniques, which suggest yet another database engine (Stonebraker & Cetintemel 2005).

This section has described the requirement of scientific databases in order to manage large volumes of data. So far the requirements for text processing, data warehousing, stream processing and the scientific databases have been discussed. The next section highlights the common themes identified by each of the requirements by modern applications.

3.1.5 Common themes

From the previous subsections some common themes can be extracted to validate the requirements of alternative databases. These common themes are discussed now.

Internet users typically generate loosely structured text-based data, which typically do not require strong transactional integrity. This data tends to have common problem areas related to scalability, querying and analysis due to the scale.

Each application area requires the management of large datasets that may not fit on a single server and as a result, requires distributed data storage architecture. In addition to storage, analysis of large volumes of data (like in scientific databases) also require processing that cannot be achieved on a single computer. This highlights the requirement for horizontal scalability of data storage, as data may grow over time. Data must also be highly available and a certain amount of stale data is tolerated in favour of availability, which implies that data can be committed eventually. The scalability and availability requirement also implies a certain amount of fault tolerance and data replication in case of hardware failure.

Certain application areas require database systems that are more read-oriented with only occasional updates. Once data is written it is seldom modified. The database, therefore, needs to be optimised for querying and must provide efficient data analysis capabilities. Querying implies some form of programming interface or abstraction layer, which can handle large datasets and distributed data.

The application domains and their data requirements present a need for more flexible solutions. Flexibility can be achieved through the following (Seltzer 2005:54):

- Develop the data store for each application to be specific to that application.
- Provide data management services for every application domain (which is the approach followed by commercial RDBMSs and which add extra complexity).
- Develop configurable storage systems, which can be applied to different application domains.

The requirements discussed in this section highlight some of the limitations present in RDBMSs. To address these limitations, alternative database models are required. In addition to the influence of modern data requirements on database models discussed in this section, additional factors exist that provide strong arguments for alternative databases. Apart from the type of data and the application areas discussed in this section, the interaction between programs and the database also needs to be considered and this is discussed next.

3.2 The need to access different data models programmatically

The previous section has described the influence modern data requirements have on the database model implemented. It has also discussed the different application areas, each having different data storage requirements. In section 2.4.3 of the previous chapter, the implication of the relational database model on the programming abstraction has been discussed. This section provides more details and arguments on why alternative databases are required when accessing large volumes of data from a programming abstraction point of view.

In the previous chapter, SQL has been discussed as a means for programmers to access and manipulate data. Accessing and manipulating relation data from a programming point of view require the mapping of programming functions to SQL statements, as stated in the previous chapter. Advances in programming languages towards an object-oriented approach mean that the data model and programming model differ. Therefore, data needs to be transformed from a relational model to an object-oriented model for programmatic manipulations and then back to a relational model to store in a database.

Traditional relational modelling consists of three steps, namely conceptual modelling, logical modelling and physical modelling. Conceptual modelling decides what needs to be in the database whereafter a logical model is created in the form of a database schema. Physical modelling creates the physical implementation of the database model. There is normally a mismatch of objects between the database structure and the program structures (Badia & Lemire 2011:63). Object-relational mapping (ORM) frameworks, such as Microsoft's Entity Framework and Hibernate, try to address this problem, but add additional processing overhead to the data processing.

The ORM fits between the objects used in the programming language and the relational presentation in the database. Mappings between application objects to relational data in the database are done in two steps. The ORM system automatically tracks changes made to data in the objects and performs the necessary SQL manipulation commands against the database. The benefit to the developer is that the abstraction of the data allows data to be viewed as objects (O'Neil 2008). Although ORM provides a benefit to developers, it does have an impact on application performance.

From a programming point of view the use of relational databases as data storage can, therefore, have a significant impact on application performance. The scale and volume of data stored in modern implementations will have large performance penalties when using relational database systems, since ORMs do not provide a way to limit the data returned from the program calling it and transform object calls to underlying SQL statements. In this and the previous section arguments for alternative database systems have been made. The next section discusses the way in which alternative database models address the needs.

3.3 Addressing the needs with alternative databases

The previous chapter has introduced the relational database model and has discussed relational database management systems. The constraints of the relational model have also been discussed. The previous two sections of this chapter have provided arguments on why alternative databases are required and what requirements they need to address.

This section describes how alternative databases address the main issues highlighted by describing how the scalability of data over multiple database servers differs from the

relational model. The section also describes the approaches to the management and analysis of data in distributed databases, and finally discusses the approach they implement for accessing data programmatically.

3.3.1 Scalability of data over multiple database servers

Section 3.1 has discussed the influence that the type of data being generated and the different application areas has on the need for alternative database systems. From the common themes identified in section 3.1.5, it can be learned that the volume of data that needs to be stored cannot be stored on a single server, and also that data processing required on distributed database implementations cannot be performed on a single machine. This section highlights the problem of scalability of data and how alternative database systems address this issue.
The scalability of data storage requirement exists because the volume of modern data generated is too large to store on single servers and data analysis processing is too resource-intensive to occur on a single server. According to a White Paper by the International Data Corporaion (IDC) (Gantz, Reinsel, Chute, Schlichting, McArthur, Minton, Xheneti & Toncheva, Manfrediz 2007), it was estimated that the information created, captured and replicated has been in the region of 988 exabytes in 2010.

A report published in 2012 by IDC revised this figure to 1 227 exabytes created for 2010, based on actual results, and it has also indicated that consumer-generated data alone in 2012 was 1 934 exabytes (Gantz & Reinsel 2012). The amount of data generated by users and organisations increases daily and organisations frequently run into scaling problems. For large scale internet-based data requirements, scalability is important (Rys 2011).

The main drivers for scalability are (Rys 2011:48-49):

- The scale of user load
- The scale of the amount of data being generated
- Computational scalability and agility of scale.

In the previous chapter the implications of scaling have been discussed. It has also been highlighted that scaling data by using relational databases in distributed environments can be achieved through the following (Vaquero *et al*. 2011):

- Caching

- Clustering
- The use of alternative database implementations.

In section 2.4.1 of the previous chapter, caching and clustering have been discussed as methods of achieving data storage scalability, using RDBMSs. The third option, namely using an alternative database model, has not been discussed yet. The use of alternative database implementations to address the requirement of data storage scalability is now discussed.

In the presence of network-shared data, the CAP theorem states that only two of the three desirable properties of *consistency* (C), *availability* (A) or tolerance to network *partitions* (P) can be satisfied (Brewer 2000). In wide area systems where data is stored across networks, the general belief is that tolerance to network partitions cannot be sacrificed and as a result, a decision must be made between having either data consistency or availability.

The term NoSQL is loosely used to describe a class of non-relational database systems that does not use SQL to query data (Pokorny 2013). The NoSQL movement's view is to focus on availability first and consistency second (Brewer 2012:23).

To address the implications of the ACID philosophy used by RDBMS a different design philosophy was required. The BASE philosophy was developed and referred to *Basically Available*, *Soft* state and *Eventually* consistent systems (Brewer 2012:24). The BASE philosophy was born out of the need to address three fundamental challenges in deploying network maintenance systems, namely scalability, availability and cost effectiveness. Since availability is desired more than consistency, stale data can be tolerated briefly. The soft state of data implies that data is not durable and can be regenerated at the expense of additional processing or disk input/output (I/O). By allowing for eventual consistent data, communication of changes or committing changes to secondary storage can be postponed and thus the issues with scalability are addressed (Fox *et al*. 1997:50).

Since NoSQL databases were designed to be horizontally scalable, they distribute data over multiple nodes differently than RDBMSs, which rely on replication. NoSQL implementations

use two key-based strategies to partition data across multiple nodes, namely range-based partitioning and consistent hashing partitioning.

Range-based partitioning is done by using a routing server that splits keysets into blocks allocated to each node. Each node is then responsible for storing and querying done on a specific key range. Clients request the partition table from the routing server to know which node to direct queries to. Range-based partitioning has the advantage of handling range-based queries very efficiently, since the ranges are stored in close proximity of each other. Range queries, however, have performance penalties, since neighbouring keys may be distributed over different nodes.

The consistent hashing strategy is a much higher available approach and has the advantage of no single point of failure, as is the case with the routing server required for range-based partitioning. Consistent hashing distributes keys through a hash function and each machine is responsible for a hash region. The main advantages of this approach are a simple architecture and it allows for dynamic cluster resizing (Hecht & Jablonski 2011).

As mentioned previously, using RDBMSs to store large datasets will eventually run into scaling issues. Although some techniques like caching and clustering over multiple nodes through replication exist, at some stage these solutions become inefficient. Alternative database models have been created specifically for horizontal scaling. Using different techniques to relational databases, they have been designed for data storage scaling over distributed database servers.

Scalability of data over multiple servers in a distributed environment also adds complexity to the management of data and, therefore, different approaches to RDBMS are required. The next section discusses data management approaches followed by distributed databases when managing distributed data.

3.3.2 New approaches to data management and analysis in distributed environments

The previous section has described the approach followed by alternative database systems to address the issue of data storage scaling. Distributed data also needs to be managed

efficiently. Section 3.1 has highlighted the requirement of analysing distributed data for data warehousing applications, which tend to be costly operations performed on data.

This section discusses the management and analysis of data in distributed environments, using alternative methods than those used by relational databases.

For internet-related data, analysis is performed on large amounts of unstructured data and content, and this is particularly true for search engines. Extracting information from unstructured content is a computation-intensive process involving character level operations such as tokenisation and pattern matching. This analysis is also done on a dynamic dataset, with new documents added or existing documents updated continuously. The analytics workflow, therefore, exists of an ingestion phase, storage and processing phase as well as an export phase. The volume of data being processed means that data and content need to be stored in distributed databases. A popular paradigm for analysing data in distributed environments is the Map-Reduce paradigm used by Google (Beyer, Ercegovac, Krisnamurthy, Raghaven, Rao, Reiss, Shekita, Simmen, Tata, Vaithyanathan & Zhu2009).
With the Map-Reduce programming model, the user creates a map function that takes a set of input key/value pairs, which produces an output set of intermediate key/value pairs. All the output sets with values associated with an intermediate key are then grouped together and passed to a reduce function created by a programmer or user. The reduce function accepts an intermediate key and a set of values associated with the intermediate key, and merges these values together to produce a smaller set of values. Typically a reduce function will produce either zero or one result.

The advantage of Map-Reduce is that it allows computation in distributed environments while hiding complexities of parallel computing. Typically a Map-Reduce program starts by splitting the input datasets into multiple pieces. It then executes copies of the program on nodes in a cluster. Coordination is done by a master, which is started on one of the nodes. The master keeps several data structures for the management of jobs and assigns the work to the worker processes on the other nodes. The master program assigns each worker a map or a reduce function. Typically there are more map functions than reduce functions.

Each worker assigned a map function parses the input set and produces intermediate sets in buffered memory, which are periodically stored to local disk. The worker then notifies the

master of the location of the intermediate set. The master then notifies a reduce worker of the location of an intermediate set. The reduce worker then reads the intermediate set through remote procedure calls, as it may be located on a different machine. The reduce function then iterates over the intermediate data and for each unique intermediate key, the value is passed to the user's reduce function. The result of the reduce function is then written to an output file and the worker notifies the master. Once all the reduce tasks have been completed, control is passed back to the user function (Dean & Ghemawat 2008).

Map-Reduce does not provide any speed-up of any data analysis methods but simply provides a mechanism to partition data into smaller pieces to allow parallel execution. A requirement for addressing problems through Map-Reduce is that the problem set needs to be partitioned into smaller pieces, which can be handled independently from one another. Searching through text is a problem set, which benefits from Map-Reduce (Janert 2011:356).

The Map-Reduce paradigm was developed for use in large scale data analysis and allowed execution of queries in a distributed environment. It could also be executed on the databases. In contrast, features for data analysis in data warehousing environments on RDBMSs are usually implemented by separating production databases from data warehousing databases. This is mainly done because of the data presentation requirements and computation expensiveness of online analytical processing (OLAP). The transactional nature of relational databases makes it less ideal for data warehousing, which is more query-driven and computational intensive. In distributed environments, data warehousing becomes more of a problem as join operations become difficult. Map-Reduce addresses this problem by allowing distributing computation across database nodes.

This sub-section has described the Map-Reduce paradigm as an approach to analyse data in distributed environments. The alternative database implementations presented in the next chapter, which have been used for this research, implement some form of support for Map-Reduce processing.

The next section discusses programming abstraction provided by alternative database implementations.

3.3.3 Programming abstraction

The need to access data differently has been highlighted in section 3.2 by describing the programming abstraction provided by RDBMSs through SQL and the use of ORM. This section discusses how this aspect of accessing data is addressed by alternative database systems through application programming interfaces (APIs).

Application performance can be affected by the domain object model, the mapping of data to the programming model and the database schema. The method in which persistence is provided can also have an impact on performance; for example, is each change committed to the database as it occurs or is it cached? In a lower application programming interface (API) access and the amount of data retrieved can be controlled. In higher level APIs such as ORM, the application usually does not have control over the amount of data being retrieved (Russell 2008).

Managing and manipulating data at application level using native programming language data structures have a significant performance advantage over using an RDBMS. In a study conducted by Jacobs (2009:38), a 100 GB data sample with random data was used to represent census data. The data consisted of 6.75 billion records, each representing a person, and ten columns for age, gender, income, ethnicity, language, religion, housing status and location. A simple aggregation query was used to determine what the median age by gender for each country was. A simple counting strategy was used by utilising buckets and counting the records present in each combination of country, gender and age. By trying to extract the median age per gender per country programmatically on a desktop computer with minimal RAM and CPU, the desired results were extracted in 15 minutes, mainly limited by the 90 megabyte per second read speed of the disc.

Executing the same test on a server with 128 GB of RAM and thus storing the same dataset entirely in memory, the results were extracted in less than a minute. To compare these results with relational database performance, the dataset was imported into a commonly used enterprise grade relational database, namely PostgreSQL[4]. The database was running on an eight core Mac Pro workstation with 2TB RAID 0 of hard disk for storage and 20 GB RAM. Interestingly the same amount of records could not be stored in the database, as the machine

[4] Available online at http://www.postgresql.org/

ran out of disk space. This data inflation is typical of RDBMSs. The bulk load process was stopped after six hours. As a result, only one billion records were stored with only three columns: country, age and gender.

To obtain the same result, a SQL select query was executed on the database, namely *SELECT country, age, gender, count(*) FROM people GROUP BY country, age, gender*. Although a significant smaller dataset was used, the query took more than 24 hours to complete. Investigation revealed that, although the trouble was not storing the data, as only more disk storage was required, the difficulty was in analysing the data (Jacobs 2009). Non-relational database implementations provided data access at API level. It is discussed in the next chapter.

This section has described how alternative databases address the scalability of data over distributed database servers, has discussed the new approaches that exist to manage and analyse distributed data and finally, has provided insight into how the programming abstraction differs. The next section concludes this chapter.

3.4 Conclusion

This chapter highlighted the need for alternative databases. The need to store and alternative data models were discussed in terms of the type of data generated by users in the internet environment. The requirements were discussed in terms of text processing, data warehousing, stream processing, and scientific and intelligent databases, while the common themes were also highlighted.

The chapter then discussed the need for alternative data bases from a programming point of view and why the query language used in relational databases provided data mismatches with modern object-oriented programming languages.

Finally, the chapter discussed how alternative databases addressed the issue of data scalability over multiple database servers, what modern techniques existed to manage and analyse data in distributed environments before discussing how they addressed the performance issues from a programming abstraction point of view.

The next chapter introduces the alternative database models and their implementations. It also provides an overview of different implementations of different models.

CHAPTER 4

NON-RELATIONAL DATABASES

The previous chapter discussed the need for alternative database models and highlighted the challenges of using RDBMSs to store and manage large volumes of data in distributed environments. New data models and application areas were discussed and alternative databases were introduced on a high level. The approaches followed by these alternative databases to address the challenges faced by RDBMSs were also discussed.

This chapter presents the alternative DBMSs that exist in more detail. Referred to as the NoSQL movement, the motivation is to achieve high scalability to handle the large data workloads of the internet (Rys 2011). The term "NoSQL" can be misleading. In general, it is translated to "Not Only SQL" and considered next generation databases that are non-relational, distributed, open-source and horizontally scalable (Edlich *n.d.*).

This chapter explores the different data models used by these alternative DBMSs, explains how they are implemented and what their features are. The subsequent chapters consider the databases discussed in this chapter to answer the research question of whether these database implementations address the limitations of RDBMS when storing user-generated, text-based data in distributed environments.

This chapter is structured as follows:

Section 4.1 provides an introduction into the alternative databases and the different database families.

Section 4.2 discusses column-oriented databases.

Section 4.3 discusses Key/Value databases.

Section 4.4 discusses document-oriented databases.

Section 4.5 introduces graph databases.

Section 4.6 provides a summary of the different non-relational databases.

Section 4.7 looks at the additional application areas of non-relational databases.

Section 4.8 provides the conclusion of the chapter.

4.1 Introduction

The need for alternative databases has led to the development of different database models, each having numerous implementations to which are collectively referred as NoSQL databases. Storing user generated text based data in a distributed environment requires the need for alternative data models than used in traditional RDMBS. This type of data is less reliant on a pre-defined schema and is prone to complex queries that are prevalent in data warehousing applications. In addition to a different data model, there is also a need for scalability of data storage over multiple database servers. For performance requirements in accessing data, there is also a need to investigate the level or programming abstraction provided by these database systems. The rest of this chapter discusses different databases in terms of the data model, the scalability and the level of programming abstraction to provide detail as to why they were chosen for this research.

This section provides a high-level overview of the alternative database models that exist. Subsequent sections describe each of the database models considered for this research and discuss the implementations of each model used for this research.

According to Cattell (2010b), NoSQL systems are not bound by predefined schemas, and objects can have any number of attributes of any type. These do not have a SQL processor that can cause poor performance through expensive joins required for reporting on normalised data, but instead, data access is provided through a simple query API. Cattell (2010b) classifies NoSQL databases, based on a set of desired features. These features are:

- Horizontally scalability of simple operations
- Partitioning and replication of data over multiple servers
- Simple call level interfacing to the databases
- A weaker consistency model

- Dynamic schema or data record attribute changes
- An efficient mechanism for managing distributed indexes.

These desired features in essence require that data be stored easily in distributed environments over multiple servers. Simple operations like retrieving, inserting and updating data should be executed easily across multiple servers. The need for a simple call level interface removes the requirement of a query processor (like SQL). The requirement should allow for fast data access and processing. By having a weaker consistency model, older or stale data is tolerated in favour of the availability of data. Since changes to schemas can have an influence on the availability, i.e. down-time to implement changes, the NoSQL databases allow dynamic schema changes. Scalability of data storage across multiple database servers also requires an efficient mechanism to manage distributed indexes, as data may be added and removed frequently.

To allow for operations to scale over hundreds or thousands of servers is one of the most important features of these databases and the implementation of 100% ACID properties is of lesser importance. Consistency required by ACID is sacrificed in favour of availability. In addition, high availability over multiple servers makes scalability more useful and is one of the main aims of NoSQL systems (Cattell 2010b).

Understanding the horizontal scalability that NoSQL databases achieve requires some terminology. The concept of "clusters" has already been discussed in section 2.4.1 of chapter 2. To reiterate, in distributed computing environments, collections of interconnected stand-alone computers that are working together as a single integrated computing resource is known as a cluster. Clusters consist of interconnected computers, and in computer networking terms, any system or device connected to a network is called a node. A cluster can then consist of many nodes. By using high-speed networks, multiple database servers or nodes can be connected to one another (Yeo, Buyya, Pourreza, Eskicioglu, Graham & Sommers 2006), and these can be presented as a single integrated resource.

For the purpose of this research, a node refers to a single database server instance running on a single standalone computer. There is only one instance of the database running on an actual machine. A cluster refers to a collection of database nodes (and thus database servers)

working together to present clients with a uniform database, although data may be distributed over several nodes. When accessing the database, the client is only aware that it is updating a database. He or she is unaware that the data is actually stored and distributed to any of the nodes (and therefore, database servers on separate machines) in a cluster configuration.

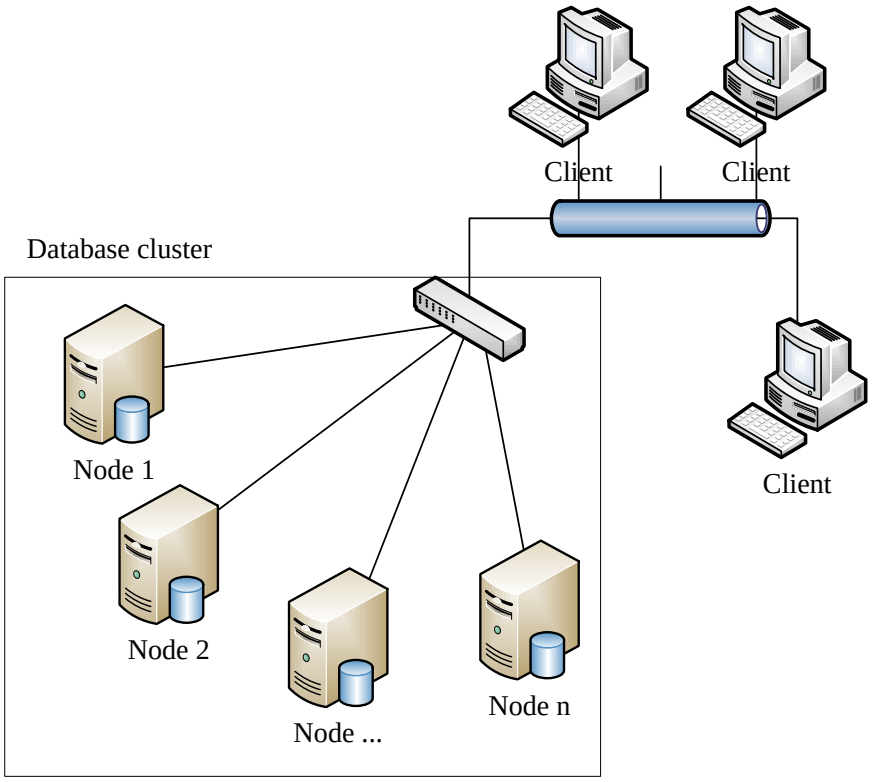


Figure 4-1: Clustered database environment with multiple nodes

According to the NoSQL website[5], the alternative database models that differ from the relational database model are column-oriented databases, document-oriented databases, key-value databases and graph databases. Each of these database models is discussed next in terms of the model implemented, scalability and programming abstraction, which have been highlighted in the previous chapter as enhancement areas for modern database requirements. For each of these alternative data models, the open source implementations used in this research are also discussed with specific reference to the data model, scalability and programming interface provided.

[5] Available at http://www.nosql-databases.org

4.2 Column-oriented databases

In the previous chapter, data warehousing has been discussed as one of the application areas that benefits from alternative database models. Column-oriented databases have their origin in data analysis and business intelligence; they address the scalability problem over multiple distributed servers (Strauch 2011:104). Data warehousing applications require fast access to data where query results are returned from different columns (Stonebraker & Cetintemel 2005). Column-oriented databases address this requirement by using a data model that is suitable for columnar queries.

This section introduces the column-oriented family of databases by describing their data model, scalability capabilities and programming abstraction before exploring two open source implementations, namely Cassandra and HBase.

Data model

Column-oriented databases provide a data model closely related to the relational model, but which allow dynamic attributes. The main difference between relational databases and column-oriented databases is that relational databases store null values for each column that does not have a value, whereas column-oriented databases only store data that is available, and thus only a column if there is data (Hecht & Jablonski 2011). Storing only values if they are available provides an implementation not bound by a schema.

The basic model of column-oriented databases is rows and columns (Cattell 2010b), based on Google's Bigtable (Chang *et al*. 2008). Column-oriented databases store data in a distributed multidimensional sorted map. A map is a data structure consisting of a key/value pair; for example *Map (key, value)*. Multidimensional maps are, therefore, maps where the value is also a map; for example *Map (key, Map (key, value))*.

In Bigtable, data is stored in a persistent multidimensional sorted map in a distributed environment, and organised in three dimensions, namely rows, columns and timestamps. In Bigtable, storage is referenced by accessing a row key, column key and timestamp, while *values* in the map are stored as a byte array. The timestamp dimension allows multiple versions of the same data to be stored at different timestamps (Chang *et al*. 2008).

Column-oriented databases store data in a row, which can store an arbitrary number of key/value pairs. These represent a column, where the *key* denotes a column identifier and the *value* is the actual value stored. A key/value pair is a simple data structure consisting of a unique identifier, the key, and the associated value it stores, i.e. *(key, value)*. The timestamp dimension used in Bigtable is not required by other column-oriented database implementations. Column-oriented databases store values as byte arrays and as a result, data is only stored and not interpreted by the database itself. Data interpretation and relationships between datasets, therefore, must be handled at the application level (Hecht & Jablonski 2011:337). The database has the only function of storing data. The application accessing the data, therefore, needs to be aware of the data type stored. This is in contrast to relational database models described in section 2.2 where the database schema defines the data type of each attribute (or column).

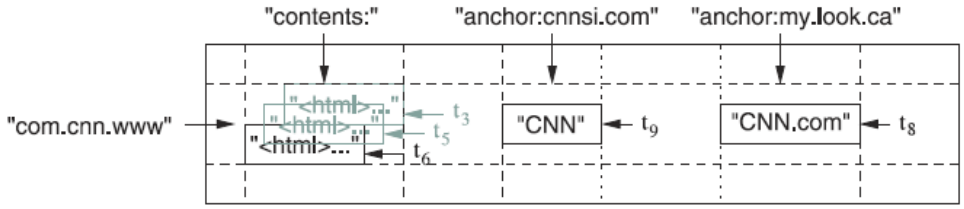


Figure 4-2: Bigtable, a table representing web pages (Chang *et al*. 2008)

Rows are used to group data elements together to form a unit of load balance, meaning that row keys are used in the partitioning strategies discussed in section 3.3.1 of the previous chapter. Row keys uniquely identify a row and are not part of the key/value pair that identifies a column, as discussed earlier in this section. Row keys can be arbitrary strings with a length of up to 64 KB and rows can consist of multiple columns with variable lengths. This means that each row can have a different number of columns, as the column is identified by a key and the value of each column can also have a different size.

Columns, in turn, are grouped into column families, which form the unit of access control. The aim of column-oriented databases is to allow data access in columnar form instead of per record, as in the relational model. Column keys identify a column and are stored as a set consisting of a family and a qualifier, and are named using the syntax *family: qualifier*.

Qualifiers can be arbitrary strings. A family qualifier combination provides access to the actual data. The column-oriented storage can be explained by looking at Figure 4-2. In this figure, there is a row, with the row key being "*com.cnn.www*". This row stores multiple columns, identified by a key and a value. In the simplest form, the first column stores the actual HTML, which is the *value* and the *key* is "*contents*". The next two columns illustrate the concept of a family and qualifier set where the family is "*anchor"* and the qualifier for the second column is "*cnnsi.com*". Actual data, *"CNN"* is, therefore, accessed through row "*com.cnn.www*" and column *"anchor:cnnsi.com"*. From this example it is also possible to see that different rows can have a different number of columns, i.e. some pages can have more anchors than others.

This section has described the data model used by column-oriented databases and illustrates how access to data in a columnar manner is achieved. The next section describes the scalability of data storages allowed by column-oriented databases.

Scalability

Column-oriented data stores provide higher scalability of data storage. Availability through partitioning and database wide ACID properties are abandoned to allow for distributed storage. Scalability of data storage is achieved by splitting rows and columns over multiple servers. The different strategies to achieve scalability have been discussed in chapter 3, section 3.3.1. As discussed in the previous section, row keys are used as the unit of load balancing.

Rows are stored across multiple servers by sharding on the row key and are typically split by a range. Sharding refers to the splitting of data over different nodes in a cluster and has been discussed in section 2.4.1. Sharding allows queries on ranges to be directed to only the server in question and not all servers. In addition to rows, column groups can also be used to split columns over multiple servers. Both row and column partitions can be used simultaneously on the same table (Cattell 2010b).

This section has described how data storage can be scaled over multiple servers by column-oriented databases. The level of programming abstraction provided by column-oriented databases is discussed next.

Programming abstraction

As mentioned earlier in section 4.1, one of the features of NoSQL databases is that they have a simple call level interface to the database. NoSQL implementations mostly do not provide a query language to access data, as data manipulation is mostly done through an API or REST interface. REST interfaces are used by document-oriented databases and will be discussed later in section 4.4. The programming abstraction presented by column-oriented databases is discussed next.

Column-oriented databases do not have a query language like SQL for relational databases, which allow client applications direct access to the data. However, column-oriented databases provide querying capabilities such as range queries and operations like "in", "and/or" and regular expressions, but only on the keys and not on values. This means that these operations are only available in telling the database server which columns to access. Since the column-oriented databases are deployed in distributed environments with huge amounts of data, queries may need to return results from different nodes. Therefore, column-oriented databases provide map-reduce capabilities to enable parallelised calculations on different servers (Hecht & Jablonski 2011:338).

This section has introduced column-oriented databases and discussed the data model used by this database family, how scalability of data storage is addressed and the level of programming abstraction they provide. The implementations of Cassandra and HBase, as column-oriented databases, are discussed next.

4.2.1 Cassandra

The previous section has introduced the concept of column oriented databases and described the three areas of interest for this research, namely data model, scalability and programming abstraction. This section describes Cassandra as a specific implementation of column-oriented databases by providing more details on the three areas highlighted.

Cassandra was developed to be used by Facebook for their inbox search. It was designed to be used in large deployments over multiple data centres with lots or writes and support for statistics and analysis (Hewit 2011:26). This makes it of interest in terms of scalability of data storage in distributed environments, as well as from the data warehousing point of view.

Data model

Cassandra has a multidimensional map of Key/Value data storage architecture where every table consists of a row key of arbitrary length containing columns, which consist of key/value pairs, as seen in figure 4–3. Operations under a row key are atomic, meaning that they need to complete and can only be saved if the transaction is processed successfully. Columns are grouped together into sets, which are called "column families" (Lakshman & Malik 2010); these families are used to access data. A column family associates similar data, as described earlier in section 4.2.

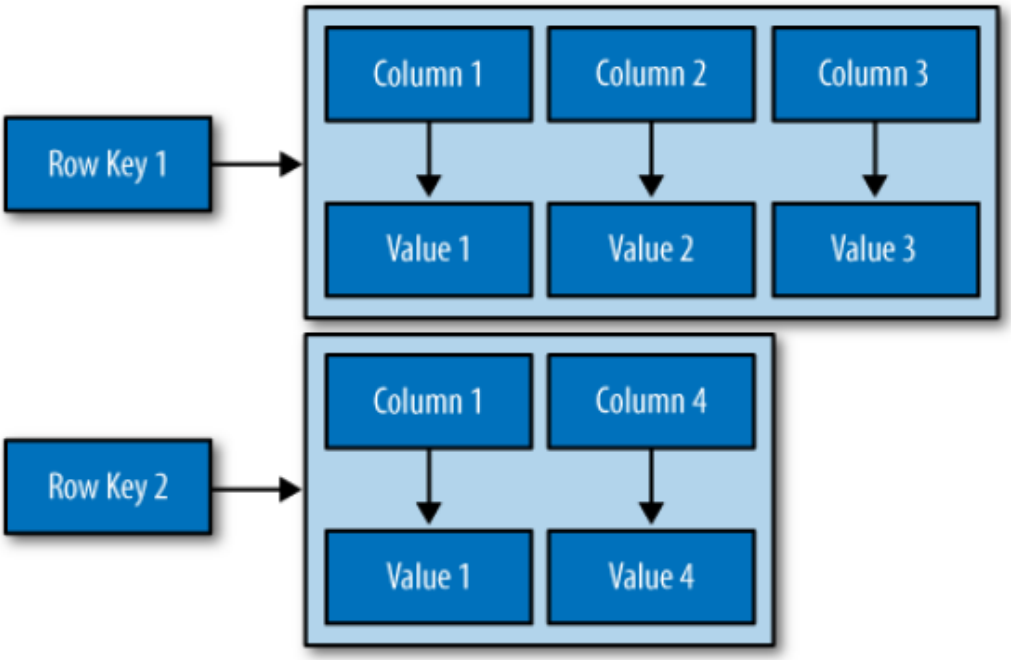


Figure 4-3: A column family in Cassandra (Hewit 2011:44)

On a more technical level, columns in Cassandra consist of keys (called the name) and the value, which are both stored as byte arrays. A column also has a timestamp, as can be seen in figure 4–4. Row keys and column names do not have to be strings like in relational databases, but can also be integers, Universal Unique Identifiers (UUIDs) or any type of byte array (Hewit 2011).

Figure 4-4: Column format (Hewit 2011:50)

Cassandra supports simple and super column families. Column families are used to store groups of columns together in a collection, as described in section 4.2 when Bigtable was discussed. For example, a simple column family can be used to define a family called "*anchor*" and group additional columns with column names like "*anchor:cnn.com*". Simple columns only store values (as in figure 4–3) while super columns are (as in figure 4–5) columns within a column family. Super families allow Cassandra to group column families together. For example, it is possible to store "*anchor:news:cnn.com*" *and* "*anchor:sport:sport.cnn.com*". Columns are accessed through the *column_family: column* or *column_family: super_column: column* convention (Lakshman & Malik 2010).

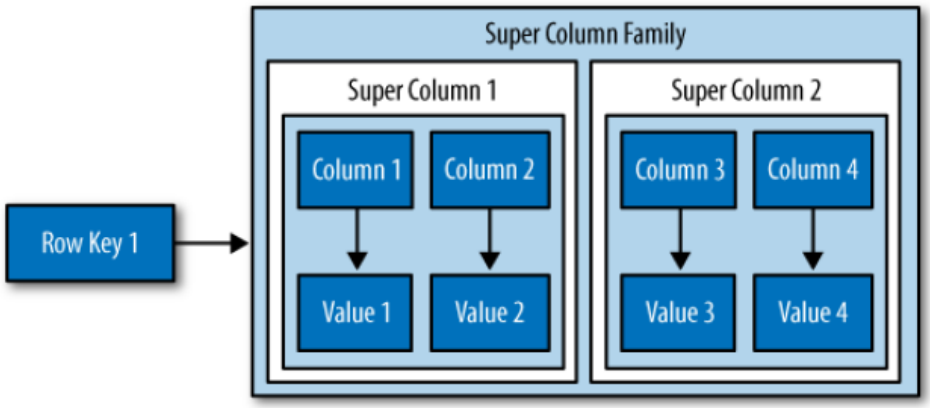


Figure 4-5: A super column family (Hewit 2011:45)

Sorting is an implementation decision and specified in the column or super column family. It is also important to know that Cassandra does not support sorting on the actual values but only on the keys (or names) of columns.

Cassandra does not deviate from the column-oriented database data model described earlier. The next section discusses the data storage scalability of Cassandra in distributed databases.

Scalability

The approach followed by column-oriented databases to achieve scalability of data storage over multiple servers has been described earlier. This section describes how Cassandra achieves data storage scalability.

Scalability in Cassandra is achieved by using multiple servers or machines (known as nodes) in a clustered configuration. Cassandra has been designed to be a multi-node database and as a result, data is assigned to nodes in a cluster, which is arranged in a ring structure (figure 4–6 below). The nodes in a database are a shared-nothing node, meaning that each database node has its own hardware.

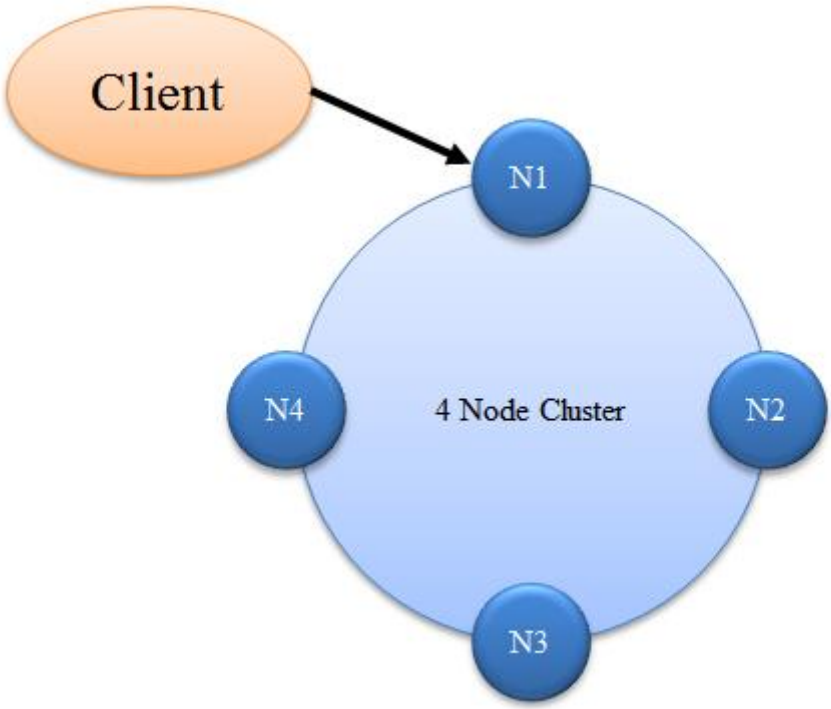


Figure 4-6: Ring structured cluster with four nodes

The *cluster* is, therefore, the outermost structure. Clusters are containers for *keyspaces* and typically a cluster contains a single *keyspace*. A *keyspace* is similar to a database in RDBMS terminology. A *keyspace* can have the following attributes: a replication factor, a replica placement strategy and column families (Hewit 2011:45-46). Replication factor sets the amount of nodes in the ring, which should be used to store replicas of each row. The replica placement strategy sets the method to use when placing replicas on nodes and can be

SimpleStrategy (rack unaware [6]), *OldNetworkTopologyStrategy* (rack aware [7]) and *NetworkTopologyStrategy* (datacentre sharding[8]). The column family attribute specifies the columns that the *keyspace* will store and is similar to the tables in a database in RDBMS terminology.

Cassandra caches update to memory and flush them to disk where they are periodically compacted. Cassandra supports both partitioning and replication on data while failure detection and recovery are fully automatic. Cassandra does not have a locking mechanism and replicas are updated asynchronously. New nodes are automatically added to a cluster and node failures are detected automatically (Cattell 2010b).

Writing data in Cassandra does not cause any disk reads or seeks, as data is written upfront to the primary memory in *memtables*; therefore, this operation is fast. Commits are only done to secondary storage later. Cassandra only supports appended writes and if the key already exists, the data will be replaced. Cassandra allows for consistency levels to be specified through client queries. This allows the programmer to specify how many nodes must respond to a write before feedback is returned to the client (Hewit 2011:130-131).

To read data from Cassandra, a client can connect to any node in the cluster. If the required data is not stored on the node, the node will act as a coordinator and read the data from a node that does have the data. Cassandra, therefore, needs to perform seeks to locate data and reads are slower than writes (Hewit 2011:132).

This sub-section has described how Cassandra implements the scalability of data in a distributed environment. The next section discusses the level of programming abstraction that is available to programmers.

[6] Nodes can be anywhere on the network.

[7] Nodes have to be on the same switch on the network or in the same server rack.

[8] Nodes are located in the same data centre.

Programming abstraction

As mentioned earlier, column oriented databases does not have a query language, but provide direct client data access. This section describes the programming abstraction that Cassandra presents.

Cassandra has been developed in Java and a client interface is provided through the Thrift[9] framework (Cattell 2010b). Cassandra has support of many high-level client libraries in different languages, including libraries for Python, Java, .NET, PHP and Perl. Although these libraries support easy development integration, they are all built on the Thrift framework, which is the driver-level interface for Cassandra (Apache *n.d.*).

Cassandra does not have an update operation. Updates can be handled by doing an insert with an existing key, in which case Cassandra will overwrite the values of any matching keys (Hewit 2011:129). Cassandra provides a way to load datasets in a batch. Since data is never updated in Cassandra but only appended to or overwritten if the key already exists, no roll-back feature exists in case a batch update fails and this needs to be handled by the client application. In addition to the delete operation, which removes a single column, batch deletes are possible by using a deletion structure with a batch mutate operation (Hewit 2011:150-152).

Data can also be accessed through a key range. Key ranges allow access to data that falls within a key range or a set of rows (Hewit 2011:133-146). The column-oriented nature of Cassandra allows data to be accessed through *ranges* and *slices*. A *range* is a set of elements between two points, i.e. a start and an end. *Ranges* refer to a range of keys or rows and can be seen as a vertical selection, i.e. get all the columns between row key X and Y. A *slice* is used to present a selection of columns; for example, get all the columns where the column family is "*anchor*". Both read and write operations use a slice predicate to specify the set of columns to be used. Slice predicates can be populated with either the list of column names or by a range of column names, and they highlight the importance of identifying columns correctly. An example of column *slices* using range can be 'get all the records where the column name is between "*anchor:cnn.com*" and "*anchor:fox.com*" ', which, in this case, would require some sort of string processor to determine the range properly. Cassandra also provides the

[9] Thrift is a software framework for scalable cross-language services development and more detail can be found at https://thrift.apache.org/

capability to retrieve multiple rows of data for a specified slice. The results are returned in a map that consists of a byte array (the keys) and a list of columns (Hewit 2011:147-148).

In line with other column-oriented databases, complex queries and data analytics are supported through the Map-Reduce functions.

This section introduced Cassandra as an open source implementation of column oriented databases and provided details on the data model, the scalability and the programming abstraction specific to Cassandra. The next section presents HBase as the other column-oriented database implementation used for this research.

4.2.2 HBase

HBase is the second implementation of a column-oriented database that is used in this research. This section discusses HBase in relation to the three areas of data model, scalability and programming abstraction. HBase is an open source implementation of Google's Bigtable data store (Cattell 2010b).

Data model

HBase implements the column-oriented data model, with data being stored in columns and rows. A row can consist of multiple columns and each row is identified by a unique row key. A table consists of multiple rows in return. HBase, however, adds a timestamp as an additional dimension on a column and as a result, each column can store multiple versions of the data. Each column value/timestamp combination is referred to as a *cell* (George 2011:17).

The columns in a row can be grouped into column families and all columns in a family are stored together in the same file, known as an *HFile*. Column families are defined at design time and must be named with printable characters. They are referenced by the *family:qualifier* notation. Qualifiers are byte arrays. There is no limit on the number of columns in a column family or the length of a value stored (George 2011:18).

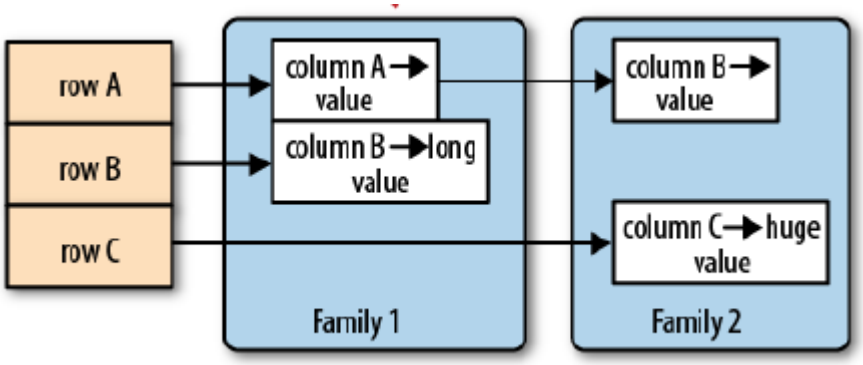


Figure 4-7: Rows and columns in HBase (George 2011:19)

The Bigtable data model on which HBase is built, stores data as a multidimensional sorted map, which is accessed through a row key, column key and a timestamp. To access data an application requires the table, row key, family, column and the timestamp (George 2011:19).

HBase is an open source implementation of Bigtable, which has been discussed at the beginning of section 4.2. This section has described the HBase model in detail and the next section discusses how HBase implements scalability of data storage.

Scalability

Continuous row ranges are stored together in what is called a *region*. *Regions* form the basic unit of scalability. Initially data is stored in one region until it becomes too large. The regions are then split dynamically into two ranges at the row key in the middle, creating two equal halves. Each region is then served by exactly one *region server*, but a region server may serve many regions (George 2011:21-22).

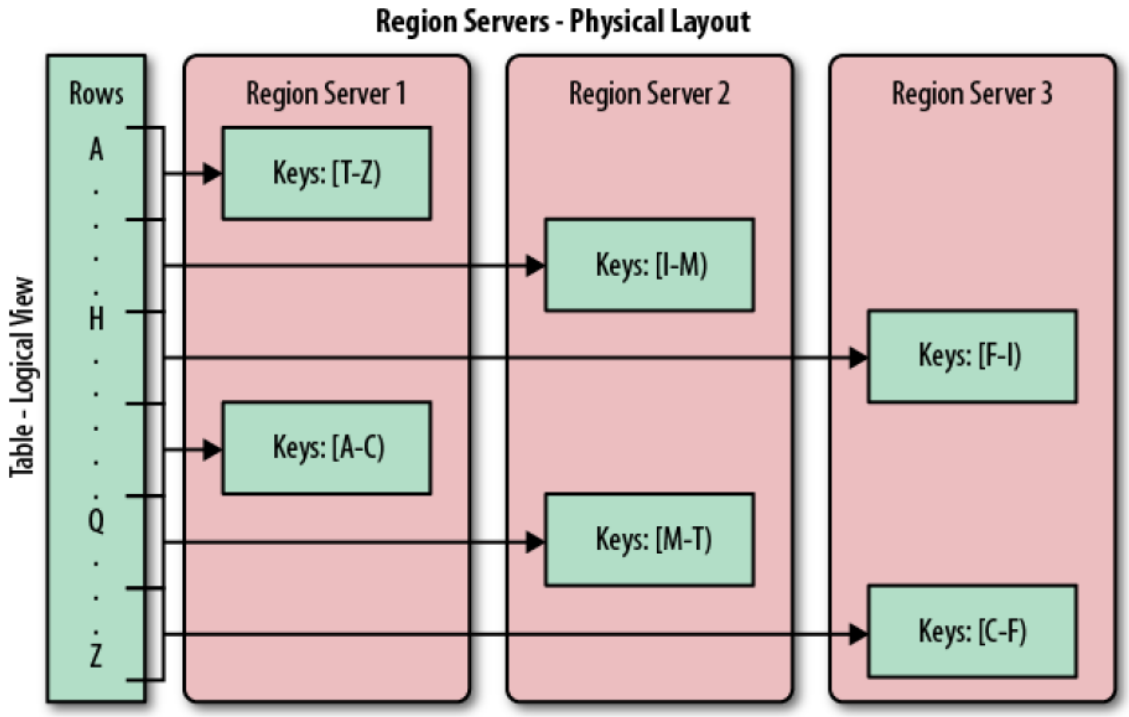


Figure 4-8: Rows are grouped into regions served by different server (George 2011:22)

HBase mutates data on an atomic per row basis and as a result, when updating a row, it is locked for that period. When multiple clients want to update the same row, contention may occur (George 2011:75).

To run HBase in a distributed environment requires a distributed file system, which is normally the Hadoop Distributed File System (HDFS). HDFS has built-in replication, fault tolerance and scalability. HDFS is not a low-level file system used by the operating system, but a storage layer file system, which eventually persists data to disk (George 2011).

Programming abstraction

The previous section has described the scalability of data storage in HBase. This section describes the programming abstraction presented by HBase. HBase does not have a query language and data access is directly achieved through an API. HBase has been developed in Java and has a native Java API, but it also has support for the Thrift framework (similar to Cassandra) and also provides a REST interface (Cattell 2010b).

HBase provides data manipulations through *put*, *get* and *delete* methods. Each of these requires an corresponding object. Each object is identified by a unique row key, which is a

byte array. HBase provides helper classes to assist with the creation of row keys; for example, a byte array can be generated from a string. Since HBase is an implementation of Bigtable, versioning of data is achieved through timestamps. If a timestamp is omitted from the object creation, a timestamp is added by HBase. HBase stores values and keys as byte arrays, and as a result, any type of data can be stored (George 2011:76-78). Results of queries are returned as a *Result* class. The *Result* class returns everything for the matching row and provides access to the column family, the qualifier, timestamp and values (George 2011:95-104).

When retrieving data, the HBase API also provides support for scans. A *Scan* object is created to start a new scan, and a start row, end row and filter can be specified. The scanner returns the result to the client as a *ResultScanner* object, which can be iterated to access results. Like *Gets*, *Scans* can also specify column families, qualifiers and timestamps. *Filters* are useful in comparisons like LESS, LESS_OR_EQUAL and EQUAL (George 2011).

Batch processing can be done on *Put*, *Get* and *Delete* objects through the *batch* method. The *batch* method requires a list of *Row* objects. *Row* objects are ancestor objects to the *Put*, *Get* and *Delete* objects and as a result, any combination of *Get*, *Put* and *Delete* can be added to list of *Row* objects and will be executed in batch (George 2011:114-117).

HBase was developed as a database to be used by the Apache Foundations Map-Reduce framework called Hadoop. HBase could be used as both the source and destination of Map-Reduce jobs (Strauch 2011). This sub-section presented HBase as an implementation of column-oriented databases and discussed it in terms of the data model, scalability and programming abstraction it implements.

HBase and Cassandra are both implementations of the column-oriented family of databases. This section has described column-oriented databases, which use a data model closely related to relational databases, but is not bound by a predefined schema. By relaxing the ACID properties and allowing eventual consistency they allow data storage to be scalable over multiple servers. Data management and access are directly using APIs and no query language processor is present.

The next section discusses the Key/Value family databases as an alternative database model.

4.3 Key/Value databases

In the previous chapter, the need for data storage capabilities over distributed servers and the need for fast access to this large volume of data have been discussed. Key/Value databases are highly scalable databases and favour scalability over consistency (Strauch 2011:53). Tey also address the requirement for fast access to distributed data. To achieve fast data access, Key/Value databases have a simple data model common to maps or dictionaries where values are stored against a key. The characteristics of Key/Value databases are discussed in this section. As in the previous section, the data model, data storage scalability and the programming abstraction presented by these databases are discussed.

Examples of Key/Value databases are Amazon Dynamo, Membase, Redis, Couchbase Server and Voldemort. Voldemort and Redis are discussed as open source implementations of Key/Value databases as, both have been used in this research.

Data model

Key/value pairs are similar to data maps or a dictionary where data or values are stored against a unique key where keys are the only way of retrieving or sorting data. Values are stored as byte arrays independent from one another. Consequently, no relationships exist and values must be handled by application logic. The simple data structure allows for schema-free storage and new values of any kind can be added during runtime without affecting availability (Hecht & Jablonski 2011). Values (or objects) are typically not interpreted, but simply handed back to the application as binary large objects. Although this eases processing load on a database, it does move the burden of interpreting the data to the client application.

Scalability

Key/Value data stores are highly scalable data stores. To achieve a high level of scalability, analytical functions like aggregation and joins are commonly not available (Strauch 2011:52). This means that from a data warehousing point of view, these databases are not useful. Where they are useful is in fast access and scalability over distributed environments. Approaches to data storage scalability will be discussed later in this section under each database implementation.

Programming abstraction

Key/Value data stores do not implement a query language and are designed to be very simple in storage capabilities in favour of fast data access. Due to the simple data model, data manipulation is limited to *put*, *get* and *delete* operations only through an API. There is no intermediate query language, such as SQL and data manipulation needs implemented in the application layer. Key/Value databases, therefore, do not support complex queries (Hecht & Jablonski 2011). The result is that all the processing and manipulation required need to be handled at application level.

This section introduced the Key/Value database model as an alternative database model. The data model, storage scalability and programming were discussed briefly. In the next section the open source implementations called Voldemort and Redis are discussed, as they have been used for the purposes of this research.

4.3.1 Voldemort

This section describes Voldemort as a distributed Key/Value database developed by LinkedIn. Voldemort provides Multi-Version Concurrency Control (MVCC) on updates while version information on records is provided through vector clocks[10]. This requires any updates and deletes to have the version information available in order to succeed. Voldemort supports asynchronous replication, but does not guarantee consistency on data (Cattell 2010b).

Data model

Voldemort uses the simple data structure of a Key/Value pair to store data. Data is stored in what is called a *Store* and can be seen as a 'table' in RDBMS terminology. Each key must be unique and can only have one associated value. A value, however, may consist of more complex objects such as lists or maps. Voldemort only allows a single value to be updated at a time and data can only be accessed through the unique key (LinkedIn *n.d.*).

To allow for fast data access, Voldemort is a memory-resident database. Data is stored in memory and only written to secondary storage eventually. As a result, Voldemort does not manage persistence itself, but uses a pluggable persistence layer. Voldemort, by default, uses

[10] Vector clocks are algorithms to generate an order of events in distributed systems. Vector clocks fall outside the scope of this research, and are described by Singhal and Kshemkalyani (1992).

BerkelyDB as persistence storage (Bunch, Chohan, Krintz, Chohan, Nomura, Kupferman, Lakhina & Yiming 2010:309), but can also use MySQL as persistent storage (Sumbaly, Kreps, Feinberg, Soman & Shah2012). It is important to note that a RDBMS is only used for persistence, and none of the other features are available in RDBMS. The data stored is serialised as byte arrays and compressed; therefore, the data cannot be interpreted by the RDBMS in a meaningful manner. Voldemort is responsible for all the desirable features, like scalability of data. It separates this data from the persistence layer.

Scalability

Voldemort achieves data storage scalability by replicating data across a cluster consisting of multiple nodes with each node having a unique identifier. Each node in the cluster must contain the same number of *Stores* (Sumbaly *et al*. 2012).

Voldemort implements automatic sharding of data on nodes using consistent hashing, as described in section 3.3.1 of chapter 3. The number of nodes used to keep a copy of data is configurable and is distributed automatically. Nodes can be added and removed from clusters, while Voldemort will automatically adjust to use the new configurations (Cattell 2010b).

Configuration information is stored by every store. These include the replication factor, required reads, required writes, key/value serialisation and compression, and storage type to use. The replication factor specifies the number of nodes to which each key/value pair must be replicated. Required reads and writes indicate the number of nodes to read from or write to in parallel for a request to be successful. Each node stores a complete cluster topology and store definitions. Therefore, each node allows clients to connect to any node and request information directly from the node storing the information (Sumbaly *et al*. 2012).

Voldemort allows client read/write operations to occur at any node. This may lead to inconsistent data for short periods. Data is requested on a key and as a result, Voldemort may return multiple versions of records. Client applications need to determine the latest version through the version information supplied with the result.

Programming abstraction

The previous sections have highlighted that Voldemort focuses on fast data access and as a result, no query processor is available. Voldemort provides a database that has fast distributed

data access. Voldemort has been developed in Java and provides client access through an API exposed via the Thrift framework similar to the one Cassandra and HBase use. This allows for data access support to multiple programming languages (Bunch *et al*. 2010:309).

Voldemort uses a plug-able architecture, which separates functions by modules, as discussed in the scalability section. Each module can be interchanged easily. The client API module is a simple interface, and allows *put* and *get* functions. In addition, the routing module handles replication and partitioning of data in distributed environments. In read/write configurations the *Conflict* resolution module handles inconsistency. The serialization module handles the serialization of data into byte arrays for storage. As mentioned in the data model section, the persistence module or storage engine handles the physical storage of data (Sumbaly *et al*. 2012).

At the lowest data access level, keys and values are stored as byte arrays. Voldemort implements serialization to provide a pluggable interface for user-required data formats. Higher level types can be supported through serialization by implementing a *Serializer* class for the type or data format. The client is thus responsible to serialize and de-serialize the data stored in the database correctly. Voldemort implements standard serializers out of the box, which provide support for JSON, Strings and Thrift. Byte arrays can also be stored natively by using *Identity*, which disables serialization and de-serialization (LinkedIn *n.d*.).

By supporting JSON, complex data structures can be supported. This allows programmers to store data as documents that may be comprised of numbers, strings, boolean values, objects and arrays. JSON is a lightweight data interchanged format that is based on a subset of the JavaScript programming language. It is a highly readable format and consists of a collection of name/value pairs contained in an ordered list (Douglas 2006). An example of a document in JSON format is:

```
Doc = {
                "Greeting" : " Hello, World!",
                "foo" : 3
}
```

Figure 4-9: Example of a JSON document

In this example the document consists of two keys, namely "Greeting" and "foo". Each key has an associated value, i.e. the value for "Greeting" is "Hello, World!" and the value for "foo" is three (Chodorow & Dirolf 2010:5). JSON is also used to store documents, as will be seen in the document-oriented databases described later in section 4.4.

The standard API only allows for simple operations to *put*, *get* and *delete* data. Complex querying capabilities do not exist and these must be handled in the code. This, however, means that the DBMS only provides data storage, and that data analysis and warehousing features are not supported by Voldemort.

In this section, it has been discussed that Voldemort implements a simple Key/Value data model as well as providing scalability and a simple programming interface. Voldemort aims to provide fast data access by being a memory resident database; however, consistency is of lesser importance. Voldemort also uses a pluggable architecture and as a result, data persistence is not handled by Voldemort itself, as it can be provided by relational databases like MySQL.

The next section discusses Redis, which is an open source implementation of the Key/Value database model.

4.3.2 Redis

Redis is an implementation of Key/Value databases. It is described in this section in terms of the data model, the data storage scalability and the programming abstraction it provides. Redis stores data in primary memory to provide fast data access. Data is copied to disk for backup or server shutdown. Redis provides asynchronous replication (Cattell 2010b).

Data model

In line with the Key/Value database model described earlier, Redis stores data as key/value pairs. Data is stored against a key, which identifies the data stored uniquely and value is stored against each key. Values can be keys, integers and serialized objects in XML or JSON documents. Redis treats the value as a byte array and does not care what is stored. The client application is required to process the value (Seguin 2012:7).

Although Redis is classified as a Key/Value database, it also supports complex data types such as strings, hashes, lists and sets. Redis stores datasets in memory. Persistence is achieved through either writing whole datasets to disk periodically or appending to a log file (Redis *n.d.*). Redis supports atomic operations on updates to stored values through locking (Cattell 2010b).

Scalability

Scalability of data storage in Redis is achieved through the use of replication and clustering. Replication of data "from a master to slaves" means that the exact copy of the master data can be queried on slave nodes. Redis also allows slaves to be replicated from other slaves. Replication is non-blocking on the master side, which allows the master so serve queries while slaves are synchronising (Sanfilippo 2012).

Through Redis Clustering, data can be sharded automatically over multiple nodes in the cluster. Redis does not use consistency hashing to shard data, but uses hash slots. This allows nodes to be added and removed easily by moving slots between nodes, and no down-time is required. In multi-node implementations, the distributed hashing of keys over servers is implemented in the client. Redis implements asynchronous replication to other nodes. A master-slave replication model is used to ensure that data can be served in the event of a nodes failure (Redis *n.d.*).

Programming abstraction

In line with other Key/Value databases, the aim of Redis is to provide fast access to data. No query language is available. Redis has been developed in C and client libraries exist for a large number of languages. The Redis server provides a wire protocol interface to access data. Wire protocols refer to APIs implemented directly above the physical layer in networks and they provide data transportation between two points (PC Magazine Encyclopedia *n.d.*). Redis provides *insert*, *delete* and *lookup* operations (Cattell 2010b).

Redis stores data in a database and each database is identified by a number instead of a name. Keys are used to identify data and the values are stored as a byte array. Values can be anything including strings, integers or serialized objects, like JSON. The serialization to byte arrays is done by the driver used (Seguin 2012).

Strings are the most basic data type and can have a maximum length of 512 megabytes. Strings are saved in binary and can store any kind of data or serialized objects, which include multimedia. Strings can be used as counters and can be appended to. Strings can also be used to access ranges of values (Sanfilippo 2012). Redis has built-in string manipulation functions like *length*, *get range* (or substring) and *append* (Seguin 2012).

In addition to *Strings*, *Lists*, *Sets* and *Hashes* can also be stored. Each of these is discussed next.

Lists are lists of strings, which are sorted by the order on which they were inserted. New strings can be pushed onto the list at either side of the list. New lists are created when a string is pushed against an empty key while removing the last item of a list will remove the key (Sanfilippo 2012). Redis provides support for functions and operations on lists, like range queries (Seguin 2012).

Sets are unordered collections of strings and do not allow repeated members; consequently, adding the same element will not result in multiple copies. The advantage of this is that when a new element is added, there is no need to see if it already exists. Sets support server side commands, which allow unions, intersections and difference computations on sets. Redis supports sorted sets, which use a score to determine the order of elements. Sorted sets allow for fast add, update and remove operations (Sanfilippo 2012).

Hashes allow for an extra level of data storage and can be seen as having a field on a key/value pair (Seguin 2012). Hashes are maps between string fields and string values, and are used to store objects (Sanfilippo 2012).

Although Redis is seen as a Key/Value database, it provides additional features and data structure support that do not exist in other Key/Value databases.

This section has described Key/Value databases, which use a simple data model by storing a value against a key. These databases provide fast data access and are highly scalable. The Key/Value databases discussed in this section do not provide analytical processing of data and instead rely on the client application to manipulate data. Voldemort and Redis have been presented as implementations of Key/Value databases and have been used in this research.

The next section discusses document-oriented databases, which are built on the ideas of Key/Value databases.

4.4 Document-oriented databases

The previous section has discussed Key/Value databases and the simple data structure they use for fast data access. Document-oriented databases provide slightly more meaningful data structures than Key/Value databases by implementing a data model that recognises the structure of objects (or documents) it stores (Strauch 2011:69).

This section describes document-oriented databases in terms of the data model it uses, the approach to data storage scalability and the programming abstraction it provides. Document-oriented databases are useful in storing images, text and xml. They can scale over multiple servers (Cattell 2010a), which fits in with the requirement of storing text-based data, as discussed in section 3.1.1 of chapter 3.

MongoDB and CouchDB are presented in this section as the implementations of document-oriented databases that have been used in this research.

Data model

In document-oriented databases, data is encapsulated in a set of key/value pairs, similar to Key/Value databases. The difference is that the structure of the data is recognised as documents by document databases. The most common encapsulation of data is through the use of a JSON-like structure (Hecht & Jablonski 2011), similar to what is also used by Voldemort (section 4.3.1) and Redis (section 4.3.2).

Documents are stored in collections that are similar in structure. Each document requires a unique key, which needs to be unique in the collection (Hecht & Jablonski 2011). If a key is not provided, it will be created internally by the database server (Lerman 2011). A simple querying mechanism is implemented to retrieve documents (Cattell 2010b). Document databases can associate any number of fields or any length to documents, as documents are retrieved through dynamic and unpredictable queries.

In contrast to Key/Value databases, the values can be queried directly and this allows for storing complex nested data structures. By using JSON to store documents developers have support for different data types. Document stores do not have any schema restrictions and as a result, elements can be added or removed from documents at run-time without impacting existing documents (Hecht & Jablonski 2011).

Scalability

Document databases provide horizontal scalability of data storage and data can be partitioned over multiple servers. Document databases provide replication methods for automatic data recovery and data is persisted to secondary storage (Cattell 2010b). Scalability will be discussed in finer detail for each database implementation later in this section.

Programming abstraction

Data access in document-oriented databases is provided through an API and no query language is available. In addition to basic read/write operations, document stores provide much richer features like range queries on values, secondary indexes, querying on nested documents and operations like "*and*", "*or*" and "*between*". MongoDB, a document-oriented database, allows more advanced query extension with regular expressions, and provides operations like *count* and *distinct* (Hecht & Jablonski 2011:338). More details on the programming abstraction of each database implementation will be discussed later.

MongoDB and CouchDB are discussed next as implementations of document-oriented databases used in this research.

4.4.1 MongoDB

The previous section has introduced document-oriented databases as alternatives to relational databases. This section discusses MongoDB as an implementation of document-oriented databases. MongoDB is a document-oriented database, which provides a document querying mechanism. MongoDB provides indexes on document collections and is a lockless implementation (Cattell 2010b). MongoDB will now be discussed on the data model, the scalability of data storage and the programming abstraction it implements.

Data model

MongoDB stores documents and recognises the structure of documents. As discussed earlier, document-oriented databases use JSON-like structures to present documents. Documents are an ordered set of key/value pairs. Keys must be unique and are made up of UTF-8 characters while the value stores the actual document. MongoDB is case-sensitive and type-sensitive, which means that if the type of the value of a key/value pair is different than the type of a key/value pair with the same key, the documents are considered different. Documents can be viewed as a row in relational databases (Chodorow & Dirolf 2010:5-6). Figure 4–9 of section 4.3.1 provides a simple view of a document in JSON format.

MongoDB stores information in *databases*, which are high-level containers. An instance on a server can have zero or more *databases*. Each *database* contains zero or more *collections*, and each collection contains zero or more *documents*. Documents contain one or more *fields* (Seguin 2011). *Collections* are schema-free and *documents* can have different presentations in the same collection. *Collections* are identified by their names and a name is a UTF-8 character string. *Sub-collections* can be implemented through a naming convention by using "." to define sub-collections, but there are no relationships in MongoDB; this is purely for logical separation (Chodorow & Dirolf 2010:8).

MongoDB supports dynamic queries as well as Map-Reduce functions to perform complex aggregation queries across documents. MongoDB provides atomic operations on fields by allowing fields to be located and updated (Cattell 2010b).

Scalability

MongoDB allows data storage to be scaled over multiple servers in a distributed environment. It should be noted that multiple databases can also be hosted on a MongoDB instance, and each database is completely separate from the other with each database having its own set of permissions (Chodorow & Dirolf 2010). This setup, however, is only useful when dealing with small databases that can have shared hardware; however, for the purpose of this research this configuration is not useful.

MongoDB can distribute documents over servers in a distributed environment through automatic sharding. Replication is used for redundancy and is not used for scalability; rather a master-slave replication model is implemented.

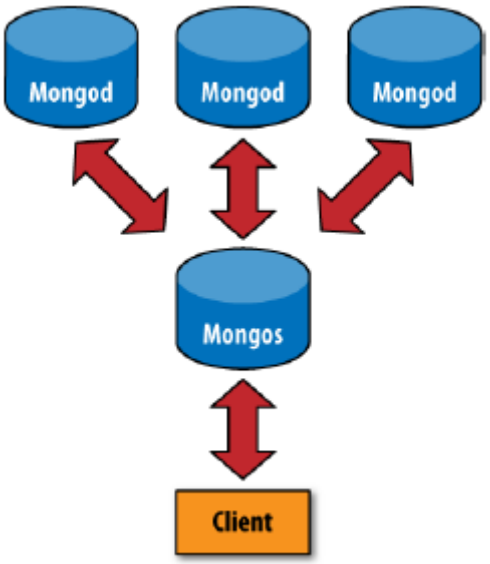


Figure 4-10: Sharded client connection (Chodorow & Dirolf 2010:144)

In order to shard data over multiple database nodes in a cluster, a *mongod* (Mongo Daemon) instance is required to store the subset of the collection's data, as indicated in figure 4–10. In addition, *config servers* are required for storing the metadata of the cluster and this is also hosted in a *mongod* instance. Queries from a client are then directed to the appropriate shard on a *mongod* instance through a routing service, called *mongs* (MongoDB.org *n.d.*).

Programming abstraction

MongoDB is a document-oriented data store, which adopts a JSON-style document model. It has been written in C++ (MongoDB.org *n.d.*). Client access is provided through language bindings for popular languages like Java, and an interactive shell is provided to access data using MongoDB's own query language (Bunch *et al*. 2010:309).

Data is stored, based on the JSON representation and supports all the basic data types supported by JSON. Additional data type support for numbers, dates, arrays and embedded documents relevant for database purposes are also supported in MongoDB (Chodorow & Dirolf 2010:16-19). Although MongoDB stores JSON documents, it serializes them to a binary presentation. Clients encode the local document structure into byte notation and send it to the MongoDB server over a TCP/IP socket connection. Binary Large Objects (BLOBS) are also supported in MongoDB, which allows the server to store images and videos (Cattell 2010b).

Data insertion is done through the *insert* method of a collection. Keys are automatically added if this does not exist. MongoDB supports batch inserters by accepting an array of documents in the insert method. This creates only one single TCP request to the server and reduces any network overhead. To remove data, the collection's *remove* method is used. If no parameters are specified, all the documents in the collection will be removed. A query document can be provided as a parameter on removes, which will remove all the documents in the collection that satisfy the query document. Document removal is permanent and there is no rollback feature in MongoDB (Chodorow & Dirolf 2010:23-25).

MongoDB allows documents to be updated through the *update* method on a collection. Updates are atomic and, therefore, the update that reaches the server first will occur first. To update only certain parts of a document without updating the whole document, MongDB has modifiers to do this. Using modifiers like *$inc* to increment a value or *$set* to change a value allows a client to update only part of a document. Modifiers for arrays are also available through the API. In addition to updates, MongoDB has a method called *upsert*, which inserts a record if it does not exist; otherwise the existing record is updated (Chodorow & Dirolf 2010:27-36).

To retrieve records from a database, the collection's *find* method can be used. If no query document is provided, all the documents in a collection are returned. To return specified documents, a query document consisting of a set of key/value pairs is passed as a parameter. By specifying multiple key/value pairs as an array, *AND* conditions can be created. MongoDB also allows a query to specify which key/value pair of a document to return. Queries have query conditionals, which allows for comparisons like "greater than", "less than", "greater than equal" and "less than equal".

Queries support the retrieval of documents that are in a range through the *$in* operator and there is support for *OR* queries through the *$or* operator. The *$not* operator can be applied on top of any other criteria to find documents that do not match the criteria. The $*slice* operator is used to return a subset of the results that is returned and can be used to return, for example, only the top ten documents. Query objects can have regular expressions to allow for matching of fields in a document, based on a regular expression (Chodorow & Dirolf 2010:45-52).

The *find* method returns the result as a cursor that allows the client control over the output. Results can, therefore, be sorted, limited or skipped to start at a certain position (Chodorow & Dirolf 2010:56-58).

MongoDB supports data aggregation through built-in query functionality that allows for counts, distinct selection and grouping of documents. For more advanced aggregation, the Map-Reduce framework is supported through built-in functions (Chodorow & Dirolf 2010:81-92).

MongoDB is a document-oriented database implementation that provides horizontal scalability capabilities. MongoDB allows queries and functions on the values stored, and provides a programming interface with a comprehensive set of functions and features to access and manipulate data.

The next section presents CouchDB as the other document-oriented database used in this research.

4.4.2 CouchDB

In the previous section, MongoDB has been discussed as an implementation of document-oriented databases. This section discusses CouchDB as the second implementation of document-oriented databases used in this research.

CouchDB is an eventual consisted document-oriented database, which stores documents in collections in a schema-free database (Cattell 2010b). Scalability is achieved through asynchronous replication. Clients can access the data from any node, as each node contains its own version of the database. CouchDB uses B-Trees for indexing and results of queries can be sorted or can be from a certain range (Cattell 2010b). CouchDB is be discussed next in terms of the data model, scalability and programming abstraction it provides.

Data model

CouchDB stores documents that are represented as JSON documents. Each document in a database has a unique key and can be any string, normally a universal or globally unique identifier (UUID or GUID). Documents consist of fields, which can either be text, numbers

or boolean values, or it can consist of other documents or lists (Cattell 2010b). To modify or update a document, the document is first loaded by specifying the key; the changes are then made to the JSON structure and the document is saved as a new version. For updates and deletes, the revision number (denoted as *_rev*) must also be supplied (Anderson, Lehnardt & Slater 2010).

CouchDB implements a multi-versioned concurrency control (MVCC) mechanism to manage concurrent access to the data. When a document is updated, an entirely new version of the document is created and it is saved over the old document. Requesting a document will always return the latest version of a document. A CouchDB server can host multiple databases, which, in turn, store many documents.

CouchDB allows each document stored to define its own structure and there is no schema. In order to make data more useful to users and applications, CouchDB introduces the view model. Views allow data to be presented more structured to allow reporting, aggregation, filtering and joining. This is done through *design documents*, which do not alter the underlying structure of documents, but can still be replicated across servers. Views are the *map* function of the Map-Reduce framework and are implemented as Javascript functions (Apache 2012).

CouchDB is an eventual consistent database and data is committed to disk at some point (Anderson *et al*. 2010:6-11).

Scalability

CouchDB implements an incremental replication model to synchronise data over multiple servers, either for redundancy or for load balancing.

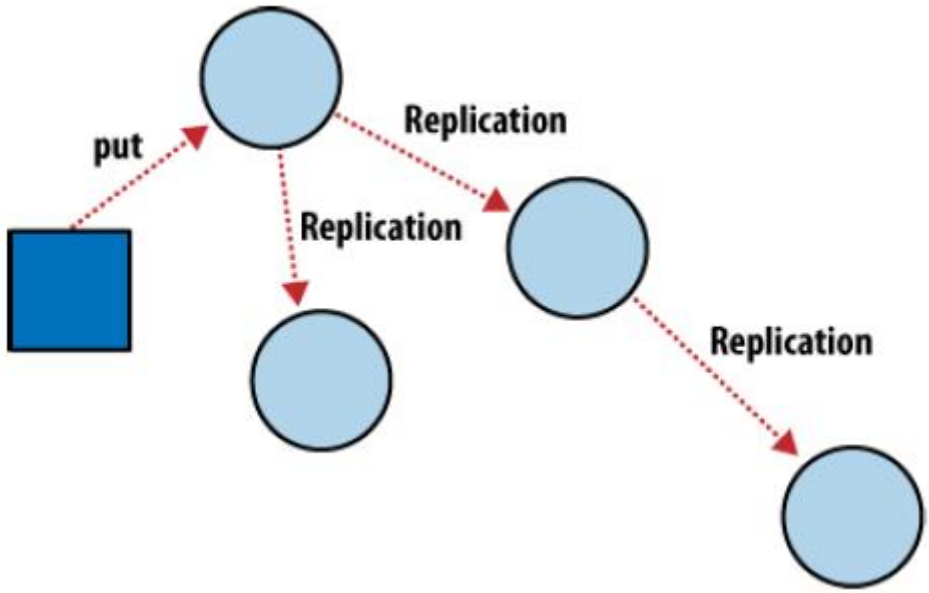


Figure 4-11: Incremental replication between nodes (Anderson *et al*. 2010:17)

In a distributed configuration data is replicated to other nodes at regular intervals. CouchDB is an eventual consistent database; this means that at some point the data will be available on the other nodes. One feature of the incremental replication approach is that databases are self-contained on a node and after replication it can function isolated if there is a network failure. Replication contention on a document is handled by using the latest version of the document as the update to save (Anderson *et al*. 2010:15-17).

In order to scale data over multiple database instances in a clustered environment, CouchDB requires the use of a partitioning and clustering application called CouchDB Lounge. Data can be sharded across multiple database instances using a consistent hashing function to map data partitions to different database instances. In order to provide redundancy replication can be used to store multiple copies over different database instances or cluster nodes (Anderson *et al*. 2010).

Programming abstraction

CouchDB has been developed in Erlang and provides a REST interface to clients (Cattell 2010b). The REST API provides access over HTTP, utilising the JSON format and is supported by libraries for many programming languages. It has native support for Binary Large Objects (BLOBS). CouchDB implements an optimistic lockless update model, which implies that client updates to a document that has been modified by another client will cause

conflict errors and needs to be resolved by the client application by reloading the document (Apache 2012).

The REST interface allows communication to the server using HTTP requests, which support different methods to pass instructions to the server through the Request-URI. Documents can be retrieved, inserted and updated through the API.

Document identifiers and revision information are important for implementing a multi-versioned concurrency controlled system (MVCC), as the version numbers tell client applications what the latest version of the document is. When a document is updated, the whole document is loaded from the database and the changes are made to the JSON structure. The whole document is then stored by the database with a new version number. Updates require both the document identifier and the revision number of the document to update. Update requests send the document identifier as part of the URL, stating the database name and the document identifier. The revision number is sent as part of the JSON document in the body of the request (Anderson *et al*. 2010:39).

CouchDB supports the storage of binary objects like multimedia as attachments to documents. The binary object is specified as binary data that is read into the body of the HTTP request and the content type is identified by its MIME type. Adding attachments also requires a revision number for the document to which the data is attached (Anderson *et al*. 2010:41-42).

This section has described document-oriented databases with relation to their data model, scalability capabilities and programming abstraction. Document-oriented databases retain the structure of the documents stored and provide searching capabilities on the contents of documents. Document-oriented databases also provide horizontal scalability over multiple nodes. MongoDB and CouchDB are implementations of the document-oriented data model that forms part of this research.

The next section introduces graph databases as an alternative to relational database models.

4.5 Graph databases

Graph databases are specialised databases for efficiently managing data, which is heavily linked. They are useful where many relationships between data exist. An example of simple implementations is FlockDB[11], which is Twitter's implementation of storing relationships between users. Graph databases are useful in location-based implementations, navigation systems and recommendation systems (Hecht & Jablonski 2011).

This research focuses on the storage of user-generated text-based documents in non-relational databases where graph databases are useful in storing relationships between objects. For this reason, they have been excluded from this research.

4.6 A summary of the non-relational databases

The previous sections have discussed the different types of non-relational database models. Each of these database models aims to address certain limitations found in relational databases.

Table 4-1: Comparison of database models

Database	**Data model**	**Scalability**	**Programming Abstraction**
Cassandra	Wide Column	Clustered Ring	Java through Thrift
HBase	Wide Column	Regions, requires HDFS	Java API, also support Thrift
MongoDB	Document	Daemon servers managed by a configuration server	Java bindings
CouchDB	Document	Requires partition and clustering application	REST interface
Voldemort	Key/Value	Clustered	Java through Thrift
Redis	Key/Value	Replication and Clustering	Wire protocol

[11] Available at https://github.com/twitter/flockdb

Table 4-1 provides a comparison of the different database models in relation to the data model they use, how scalability is achieved and how programming access is provided.

This section provides an overview of the different models that have been discussed in this chapter. Column-oriented databases provide a model similar to the relational model where rows and columns are used to store data, but which allow for additional properties on columns. Data retrieval is in a columnar fashion; this makes it appealing as model for data warehousing applications. Column-oriented databases allow data storage to be scaled over multiple servers in distributed environments.

Key/Value databases provide a simple storage model where data is stored against a key. These data stores are highly scalable but have limited application for data warehousing, as queries and retrieval of data are achieved through the keys. The value is normally stored as byte arrays and need to be interpreted by the client application, as the database server is only responsible for storage of data.

Document-oriented databases provide enhancements over Key/Value databases by emphasizing the structure of the documents. Document-oriented databases are schema-less and documents are stored against a key. The advantage over Key/Value databases is that the values or documents are searchable and not only the keys. Document-oriented databases also provide horizontal scalability of data storage in distributed environments.

Graph databases have also been introduced, but the storage model falls outside the scope of this research.

The development of alternative database management systems is the result of limitations identified in the relational model when storing distributed data. The scalability and alternative data models implemented create opportunities for application in different areas, which is discussed in the next section.

4.7 Other application areas

The previous section has highlighted that different types of non-relational databases aim to address different limitations presented by RDBMS and these different application areas are presented here.

NoSQL database systems are normally used in cloud-based systems where traditional database systems with strong transaction requirements make implementation difficult. Cloud-serving systems have the following characteristics: scale-out, elasticity and high availability (Cooper, Silberstein & Sears 2010). For the large datasets scale-out is achieved by having database systems running on different commodity servers, and to be effective loads must be balanced evenly across the servers. Elasticity allows servers to be added and removed as required. Cloud systems need to provide a high level of availability, as commodity hardware can easily fail. Traditional RDBMS implements strong ACID properties, which makes implementation in cloud environments difficult, especially for scaling and elasticity.

The scalability capabilities of NoSQL databases, therefore, make them much more suitable for these types of implementations.

This section has described additional application areas of NoSQL databases. The next section provides the conclusion of this chapter.

4.8 Conclusion

This chapter has presented the alternative non-relational database systems that exist. These database systems exist because of the limitations RDBMSs have in certain scenarios, as described in chapter 3. Although the relational database model has been implemented successfully for different applications, it might not be the best implementation for modern internet scale data implementations, which require high levels of scalability for large volumes of data. Alternative data models and data stores have been implemented in order to address scalability issues, but the question arises whether they are effectively addressing the needs of user-generated content processing, data warehousing needs and programming abstraction to make them useful.

Chapter 2 discussed the relational database management systems and chapter 3 described the need for alternative database management systems. This chapter discussed the alternative database systems and how they provided improvements over relational databases in terms of their data model, scalability and programming abstraction available.

Specific database implementations of each model were then discussed and their ability to address the limitations of RDBMS was highlighted. The specific implementations discussed formed part of this research.

In chapter 1 the question of whether these alternative non-relational database systems delivered on the promise was presented. The next chapter describes the research methodology followed to address this question.

CHAPTER 5

RESEARCH DESIGN AND METHODOLOGY

The reasons behind the problem statement, research questions and research objectives presented in Chapter 1 were discussed in detail up to now. The previous chapters described relational database management systems, the limitations with RDBMSs in certain scenarios and why there was a need for alternative database management systems. The alternative database management systems were also presented and discussed to provide insight into how they attempted to address the limitations of RDBMSs.

This chapter describes the research design and methodology followed to test the hypothesis of this research by evaluating non-relational databases. It also provides insight into how successful they are at addressing the limitations of relational database management systems. The chapter starts with an introduction to the research that will be followed and provides information on why certain decisions have been made. The research process and research design are then discussed and the methodology followed is presented after that. The chapter also presents the limitations of the research, other research in this field. Finally, concluding remarks are provided.

This chapter has the following structure:

Section 5.1 provides an introduction to this chapter.

Section 5.2 presents an overview of the research process.

Section 5.3 describes the research design followed and argues why this research method has been chosen.

Section 5.4 describes the research method used in detail. It provides the details of the research environment, what data has been used and how it has been collected, the analysis of the research data, the measurements used and finally, the validity of the research.

Section 5.5 highlights the limitations of the research.

Section 5.6 discusses related research.

Section 5.7 provides concluding remarks on the research method.

5.1 Introduction

A systematic, thorough process is required to test the research hypothesis to determine if alternative non-relational database management systems successfully address the limitations of relational database management systems when storing large volumes of unstructured, user-generated text-based data in distributed environments. To answer the research questions, the research method followed needs to describe the process followed to measure the effectiveness of different non-relational database systems in storing user-generated text-based data in a distributed environment.

In order to determine which research method to use, the type of data that these databases will store needs to be considered. Non-relational databases are presented as alternatives for storing text-based data. The research, therefore, uses text-based data as input dataset. The internet provides a platform for anyone to generate unstructured content. The large quantity of unstructured text documents requires a distributed data storage model, which allows scalability over large numbers of commodity servers. For the purpose of this research, scalability will be considered as the ability to store large volumes of data across multiple servers instead of the scaling of the number of operations that can be performed.

The driver for this research is that the Relational Database Management Model implements the Entity Relational Model, which has been designed for business-related transactional processing, implements strong ACID properties (Codd 1970), but which is not the ideal storage model for all cases (Stonebraker 2008). The data requirements and volume of data generated on the internet requires alternative data models (Hewit 2011:5) that are designed to cater for internet-scaled databases, which span multiple commodity servers. Instead of implementing strict ACID properties, these implementations follow a BASE philosophy where data is eventually committed but always available (Brewer 2012:23). These alternative data models have been developed as open source projects and are collectively known as NoSQL databases.

The purpose of this research is to evaluate the alternative, or NoSQL, databases as suitable implementations for storing and managing unstructured user-generated text-based content in a distributed environment.

Different implementations of each data model are evaluated to determine:

- Their capabilities in storing and managing user-generated text-based content
- Their efficiency as data warehousing solutions
- How efficient data retrieval and data analysis can be done on data
- The scalability capabilities
- The programming language abstraction that exists for each implementation.

The rest of this chapter describes the research design chosen for this research and the methodology followed to perform the evaluation of the database models.

5.2 Research overview

In order to provide an understanding of why the chosen research method has been selected, an overview of the research methods available is presented in this section. Different forms and methods of research exist, and each one is applicable to a specific situation. In order to choose the correct method, they need to be considered and evaluated on their merits applicable to this research.

Research can be either empirical or non-empirical (Mouton 2004:53). Empirical research is done by collecting data through observation and reporting on the data collected. Empirical research can be done by either using new or primary data, or by analysing existing data (Mouton 2004:57). The quality of empirical research depends on the quality of the observation (Olivier 2009:12).

There are different methods that can be followed in obtaining the data required. These include:

- Ethnographic research that aims to provide an understanding of a community or group, and normally takes place in the field (Mouton 2004; Oates 2010)

- Case studies that provide a description or understanding of a certain case or small number of cases in real-life situations (Mouton 2004; Oates 2010; Olivier 2009)
- Surveys that use questionnaires to obtain data from a group or sample of the population; the findings can be generalised for the larger population of the targeted group (Mouton 2004; Oates 2010; Olivier 2009;)
- Experiments that are used to make observations under high control environments where the cause and effect on relationships are tested to prove or disprove a hypothesis (Mouton 2004; Oates 2010; Olivier 2009)
- Action research that focuses on research in action by implementing something in a real-world situation and learning from what has happened (Oates 2010)
- Designing and creating, or prototyping can be used to prove that a new model can be implemented (Oates 2010; Olivier 2009)
- Evaluation research that can consist of either experimental, quasi-experimental or qualitative studies to determine the successfulness or effectiveness of certain interventions that have been implemented (Mouton 2004)
- Models and simulations are used to describe something that is more complex in order to understand and manipulate it (Olivier 2009).

Non-empirical research aims to answer questions that are not measurable in the physical world, and it focuses on scientific concepts and theories. This type of research focuses on philosophical analysis, conceptual analysis, theory building and literature reviews (Mouton 2004).

In addition to the different methods that can be followed, there are two approaches to research that can be followed: quantitative or qualitative. Quantitative or positivist research implies that observations and measurements are done in an objective manner, whereas qualitative or anti-positivist research follows the approach of experiencing behaviour (Welman, Kruger & Mitchel 2005:6-7). The positivism view is that results must be verifiable by other scientists and as a result, the observations must be measured objectively (Olivier 2009:109-111). The anti-positivist view is to focus research on experiencing human behaviour and is in essence describing the behaviour (Welman *et al*. 2005).

Mouton (2004:146) provides a classification framework for different research design types by using four dimensions. The first dimension determines if research is empirical or non-empirical. Secondly, it considers whether new data is collected or existing data is analysed. Thirdly, the type of data is considered, i.e. numerical or textual data, and lastly the degree of control of the research environment is considered. This framework is useful in determining the type of research design to follow.

Dodig-Crnkovic (2002) states that characteristics of the classical scientific methods can be found in computer science and suggests that research in this field can by definition follow an empirical tradition. According to Dodig-Crnkovic (2002), research in computer science can be divided into the following methodological areas: the theoretical method, the experimental method and the simulation method. Dodig-Crnkovic (2002) points out that all three methods have modelling as a common method. She suggests that the starting point of research should be the abstraction (or modelling) of the problem.

In addition, scientific knowledge is gained through systematic observation in a controlled manner and the results must be replicable (Welman *et al*. 2005:5). Quantitative or positivist research implies that observations and measurements are done in an objective manner (Welman *et al*. 2005:6-7). However, Oates (2010:287) points out that quantitative data is numerical and qualitative data is non-numerical; both types can be used by the positivist, interpretative and critical paradigms.

Positivism underlies the scientific method. This method assumes that the world is ordered and not random, and that it can be investigated objectively (Oates 2010:293). Positivism and the scientific method aim to find universal laws, patterns and regularities, which are mainly achieved through experimentation (Oates 2010:284). Interpretive research in information systems and computing tries to understand the social context of the system (Oates 2010:292), while critical research focuses on the identification of relations, conflicts and contradiction of computer systems in the social conflict (Oates 2010:296). The three basic techniques of the scientific method are reductionism, repeatability and refutation (Oates 2010:283).

Experiments are commonly used in the scientific method. One of the three main goals of experiments is to prove a theory (Olivier 2009:68). Experiments are in many cases used to compare two or more cases by trying to keep all the variables constant, except for the

variable being tested (or the experimental variable) (Olivier 2009:69). Experiments are designed to test a hypothesis (Oates 2010:128), which correlates with Olivier's goal of experiments (Olivier 2009:68).

Experiments are conducted by doing a pre-measurement of a variable, applying an intervention through a variable and doing a post-measurement to observe a change (Welman *et al.* 2005:78). Two types of variables are generally considered, namely dependent and independent variables. Independent variables can be seen as the "causes", and applying intervention to the independent variables affects the dependent variable (Oates 2010:129). Measurements are done on the dependent variables (Welman *et al*. 2005:78), which can also be seen as the "effects" (Oates 2010:129).

Since different research methods exist, as described in this section, the next session provides the arguments of why the research design for the research presented here has been selected.

5.3 Research design

This section describes the research approach followed in this research. The research presented here has followed a multi-phased research process that is highlighted in figure 5.1. The first phase consisted of a literature review to determine why the problem presented in this research exists by looking at relational database management systems. The literature review has also provided information on the need for alternative database management systems. Finally, it has presented the different non-relational database management systems and has explained what their capabilities are, as presented in chapters 2, 3 and 4.

The next phase of the research was the empirical research phase. Based on the different methods described in the previous section, the method chosen for the research presented here on the evaluation of different database implementations was an experimental method.

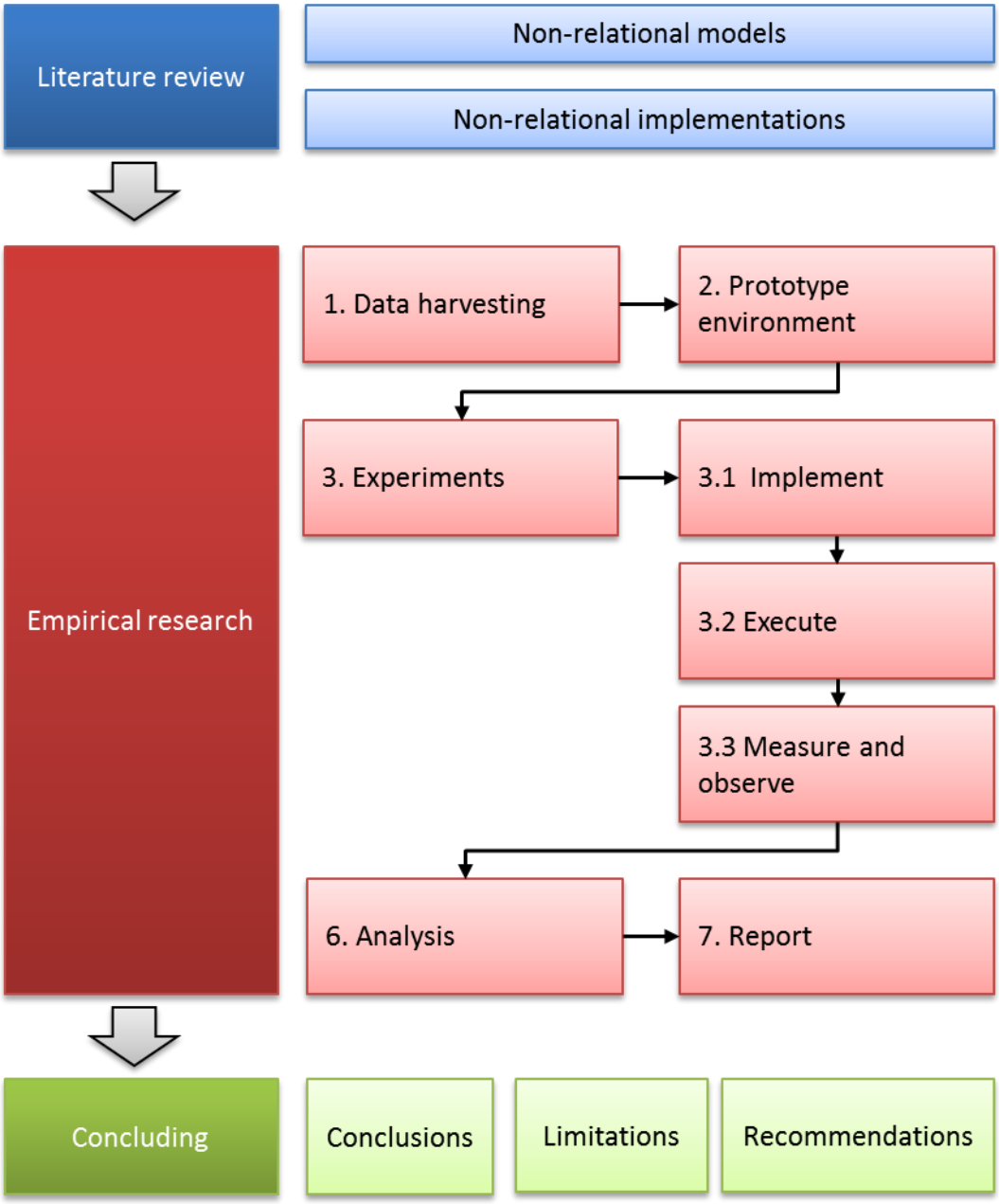


Figure 5-1: High-level overview of research approach

The starting point of research can be the creation of a model, as mentioned in the previous section. Therefore, modelling has been used as the starting point for this research. Since a model can be constructed in terms of the data to be stored (user-generated text-based content) and the environment (a distributed environment on multiple servers), the experimental method has been used. Using an input dataset, based on our data model of user-generated text-based content in a controlled environment, i.e. the same commodity servers with different database implementations, we can repeatedly observe and measure the performance

of each database implementation. Since we are observing and measuring benchmarking results of how well a task is performed, numerical data is collected. The creation of numerical data, therefore, fits into the experimental method, according to Mouton (2004:155).

From the research overview section it has been learned that experiments have two sets of variables, namely dependent and independent variables. Dependent variables are those variables of an experiment that can be measured, whereas independent variables can be seen as those variables that can be changed to cause an effect on the result. By using the same dataset on the same distributed environment, a set of dependent variables can be defined to measure the performance in terms of database operations. The dependent variables for this research can be defined as the different database requirements that are addressed by non-relational databases and measured in terms of the duration of each operation. By changing the database implementation, the number of database servers available for scalability and the size of the input datasets, the independent variables for this research can be defined to have an effect on the outcome of the measurements.

In order to assist with the execution of the experiments, prototyping has been required. Prototypes are simplified systems that are created to present a final or complete system (Olivier 2009:51). A prototype normally does not constitute research but it can be valuable in conducting experiments (Olivier 2009:52). Since the horizontal scalability of non-relational databases is one of the requirements to prove the hypothesis, a prototype distributed or clustered environment can be built to use as the test platform for the experiments. In order to get a sample input dataset consisting of text-based data relevant to the experiments required for this research, a simple Web crawler has been used to retrieve user-generated text-based content from publicly available blogs, websites and news sites on the internet.

The main advantage of experimental research is that there is full control over the environment and this allows the evaluation of different database implementations to reflect real performance on the same hardware. Factors like processor speed and RAM do, therefore, not affect the outcome of any of the experiments.

By using the prototype environment to conduct the research, a controlled environment was available. After the executing the experiment and capturing the observations and measurements, the results were analysed and reported in chapters 6 and 7.

The final phase of this research consisted of the conclusion, and the presentation of the limitations and recommendations from the research, as presented in chapter 8.

This section discussed research approached that was followed. It highlighted the main characteristics of the experimental method of research and showed how it was relevant to the research presented here. The next section describes the research methodology followed in detail.

5.4 Research methodology

The previous section described the research approach that was followed and introduced the experimental method that was followed in this research. This section describes the research method in detail.

The data required for this research was created through a series of tests executed as experiments, which were implemented as a benchmarking application. The experiments were executed in the same environment, using the same hardware and operating system in order to have a controlled experiment environment. User-generated text-based data was retrieved from publicly available websites on the internet by a Web crawler and was used as an input dataset for each experiment. Each experiment was designed to measure database features presented as dependent variables against each independent variable, as described later in this section. The measurements were recorded and used to analyse the performance of each database implementation, and the data gathered was used to compare the different databases and to draw conclusions.

The rest of this section describes the research environment in terms of the input dataset model and the environment in which the experiments were conducted. The data collection techniques will be discussed thereafter in terms of the dependent and independent variables that were used in the experiments.

Next the analysis of the data collected during experimentation is discussed. The controls, measurements, observations and validity of the research are discussed in the last subsections of this section.

5.4.1 Research environment

This section describes the environment in which the experiments have been executed.

The research required an input dataset, which consisted of user-generated text. The experiments were all executed on a standard set of hardware and software to ensure all database servers had the same specification.

Input dataset

The purpose of the input dataset is to provide the same data input for each workload of the different experiments. The input dataset is designed to fit the model of user-generated text-based content, and all experiments used the same input dataset.

The dataset consisted of unstructured and semi-structured public text-based data retrieved from various blogs, websites and notice boards available on the internet. The researcher's own blog was used as a starting point for the Web crawl and all links were followed from this blog to gather an input dataset. The dataset was retrieved as raw text and the content stored directly onto a hard disk. By saving the rendered HTML output in raw format (i.e. with mark-up included) the data could be used to test performance and capabilities of data stores to handle text.

An RDBMS database was used to store additional information about the HTML pages. It consisted of the URLs visited and the location where the data was stored on disk. For crawling purposes the content was not stored in the database, but directly on secondary storage as HTML files. The reason for this was to have full text available, since one of the limitations of RDBMS is that it cannot store large text documents, and also to eliminate additional network latency. The crawled URLs were flagged in the database to ensure that the simple Web crawler[12] did not visit the same URLs after crawling started. The structure of this data store is shown in table 1.

[12] Web crawlers are applications designed to crawl websites and retrieve web pages (http:// http://www.sciencedaily.com/terms/web_crawler.htm).

Table 5-1: Table used by the simple crawler

ID	Base URL	Page URL	Link	Flag	Date Retrieved
1	http://www.digitaltoast.co.za	http://www.digitaltoast.co.za	ROOT	1	2013-01-04 16:34:37.293

The "*Base URL*" field stores the URL of the website from which the data was retrieved. The "*Page URL*" field stored the actual page location, i.e. http://www.digitaltoast.co.za/index.php. For the first page visited, the "*Link*" field stored "ROOT" to indicate this was where the crawler started. The first URL chosen was from the researcher's personal blog. The link elements or anchor tags in the HTML (i.e. *<a href="" />*), were also stored in the "*Link Mark-up*" field. The actual content of the page was written to disk in the same location, with the "*ID*" field used as the filename to allow the content to be linked to the record in the database. The "*Flag*" field was set for each record once the content of a page had been written to disk and the "*Date Retrieved*" field simply recorded the timestamp of the processed event.

The simple crawler used a two-stage crawling process. It started off with a URL, extracted the base URL and then visited the URL to get the content of the page at the URL. The content was then stored to disk and the location stored in a local variable. The content was then scanned to extract all the link or anchor elements. The database was updated with the base URL, page URL, the file location and the record was marked as processed.

The next stage extracted the base URL and page URL or each of the link elements, which were inserted into the database. At this stage, the content was not extracted and, therefore, each URL was marked as not processed in the database by setting the "*Flag*" to 0.

The crawler then read the database to obtain a list of all the unprocessed URLs. For each one it followed the process again by starting with the page URL. This process allowed the application to stop at any time and to continue where it left off upon resuming.

The dataset collected by the crawler provided the base from which the experiments were executed. Since each of the data stores had a different data model for storage, a uniform data structure was required to represent the data that should be stored each of the data stores. From the retrieved data, the following data elements were identified for the experiments:

- *Base URL*: This element identifies the data from which the data was retrieved.
- *Page URL*: This element stores the URL from which the data was collected.
- *Retrieval Date*: This was the date the content was retrieved.
- *Content*: This is the actual unmodified content in text format as it was retrieved.

The data retrieved from the internet after the crawl stage was written to a comma delimited text file to be read as input file for the benchmarking application. This was to remove any dependency on a database server to serve the dataset. Consequently, the benchmark application did not read any data from the crawl database, but instead read the raw data directly from disk.

Loading the dataset as input for the experiments required the benchmark application to read the text file and load the HTML pages corresponding to the "*ID"* field into the memory of the client machine. Once the dataset was loaded in the primary memory of the client machine, the experiments were executed to prevent any disk access latency to influence results.

Ethical considerations on the input dataset

Some ethical considerations had to be taken into account. First of all, the crawler had to honour the robots.txt of the sites visited to ensure that it did not flood websites with requests. The robots.txt file provided information about the Web server to Web crawlers or bots based on the standard of robot exclusion, which provided information on what the crawler was allowed to do[13]. Secondly, the data crawled and stored was from publicly available websites. The data retrieved was only used to serve as an input dataset and was not used to gather information on any person or organisation that may be used in a harmful way. No analysis of any kind was done on the data and the content was not mined for any information.

Experimental setup

The experiments were executed on a cluster of commodity machines with an additional machine acting as the client interfacing machine, which was used to run the experiments. The cluster was configured in an isolated 1 Gbps network. Table 5–2 provides the hardware of each of the cluster nodes.

[13] More information available at http://www.robotstxt.org/orig.html

Table 5-2: Hardware configuration

Component	Detail
Motherboard	ASUS P8 H61-M LX R2.0 (LGA 1155 socket)
CPU	Intel Pentium G645 2.90 GHz Dual Core
RAM	Kingston 4.0 GB DDR 1333MHz
Hard disk	500 Gb SATA HDD
Network interface	1 Gb Ethernet

All nodes were configured to run Ubuntu Server 12.04.1 LTS as operating system. To allow easy switching between different experimental setups, all the required database servers were pre-installed. This meant that after each experiment only datasets were cleared if required. They did not require a complete rebuild of the whole operating system and database server. This also allowed each experiment to run on exactly the same operating system with the same updates.

The experiments were performed against the database servers listed in table 5-3 below.

Table 5-3: Data store servers used in the experiments

Family	Data store	Version
Wide Column	Cassandra	1.1.12
Wide Column	HBase	0.94.8
Document	MongoDb	2.4.1
Document	CouchDB	1.0.1
Key/Value	Voldemort	1.3.0
Key/Value	Redis	3.0.0-beta2
RDBMs	MySQL	5.5.37

All data stores listed in table 5-3 had Java APIs and this was used for all programming interaction required. The research benchmark was implemented as a Java application.

The client machine used for the experiments was an HP workstation with an 8 Core Intel Core i7-3720QM CPU @ 2.6 GHz and 16 GB of RAM on a 64-bit operating system. The

high RAM specification allowed the benchmark application to load the complete dataset into memory before executing experiments.

This section described the environment in which the benchmarks were executed in terms of the input dataset used, the clustered database server specifications and configuration, as well as the client interfacing machine specifications.

The next section discusses the data collection process in detail.

5.4.2 Research data collection

The previous section described how the input dataset was acquired through the use of a Web crawling application. This section will describe how the data for the results of this research was collected.

A benchmarking application was used for the data collection for this research. Benchmarking was used in other research by Cooper *et al*. (2010), Shi, Meng, Zhao, Hu, Liu & Wang (2010) and Pavlo *et al*. (2009) to evaluate performance of non-relational databases. Performance benchmarking can be used to identify bottlenecks in performance. A common misconception of performance benchmarking is that it only measures speed. Benchmarking of performance can be done on functional aspects such as ease of use, ease of development or even operational aspects (Sawyer 1992).

The benchmarking application was used to answer the main research question, which was to determine if non-relational database managements systems successfully addressed the limitations of RDBMS when storing user-generated text-based content in distributed environments.

In order to answer this question, secondary research questions were considered, based on the limitations of RDBMSs, as identified in chapter 3. These questions were:

- How do non-relational databases provide data management capabilities like querying, filtering and analysis?
- How do they provide scaling and distribution of data over multiple servers?

- What data mining capabilities do non-relational databases present?
- What level of programming abstraction and interfacing do they provide?

As a result, the experiments implemented in the benchmarking application were designed to test the performance and capabilities of the different databases against:

- Data ingestion and storing capabilities
- Data warehousing capabilities available by default
- Scalability over commodity hardware when storing large volumes of data
- Programming language abstraction.

These capabilities formed the basis of the dependent variables used in the experiments. Data ingestion and storing capabilities tested the capability of a data store to store and manage user-generated text-based content successfully. These experiments measured latency in doing bulk inserts into the database during the data ingestion phase. Once the bulk data load was completed, latency of ad hoc inserts and remove operations could also be measured. Simple retrieval benchmarks were also executed.

Bulk data loading or ingestion formed part of the benchmarking performed by Pavlo *et al*. (2009) and Shi *et al*. (2010). DeWitt (1992:199-166) described using retrieval and update queries to benchmark database performance when a database was under load. Although the benchmarking was developed for relational databases, the size-up, scale-up and speed-up metrics could still be used as base for testing non-relational data stores, especially when measuring the scalability.

Data warehousing experiments were performed after the data ingestion experiments had been done. The experiments were designed to test the analytical capabilities of the databases. These experiments consisted of scan and sort queries, data retrieval requests, regular expression filtering type queries, range queries and aggregate queries. For the purpose of this research, the data warehousing capabilities of each database was tested as they were implemented in the DBMS. No additional data processing like scrubbing and transformation was done as is normally required with data warehousing implemented on RDBMSs.

Scalability experiments were designed to test the capability of database implementations to store large-scale data across multiple servers and the ease of achieving horizontal scalability. Scalability wais benchmarked against scale-up and elastic speed-up. Scale-up was benchmarked by adding more machines offline and measuring performance gains afterwards by executing the same experiments for data ingestions and data warehousing. Elastic speed-up was benchmarked by adding more machines to the experiments while they were running to see if performance had been affected in any way.

Programming abstraction and overhead were measured against ease of use of application programming interfaces (API) provided by each database. Functions available to programmers to manipulate data directly were evaluated and compared to what other databases provided. A simple client application to perform operations against the data store was implemented for each data store. The benchmark application was developed in such a way that the test sets for each database exposed the same method of functions, for example *BulkInsertTest()*. Therefore, complexity was gauged by using the simple Source Lines of Code (SLOC) metric to determine ease of implementation (Bhatt, Tarey & Patel 2012).

According to Welman *et al*. (2005:78) and Oates (2010:129), experimental research consists of two types of variables: dependent and independent variables. Applying a change on the independent variables will cause a change on the dependent variables and as a result, measurement of the dependent variables will provide the results of the experiments. The variables and how they are translated into the bechmarking application used for the research will be discussed next.

Dependent variables

Dependent variables are measured to provide results for the experiments. Each of the dependent variables represents a required feature that addresses the limitations of RDBMSs. By being able to have an implementation that could measure a dependent variable provided proof that the limitation had been addressed.

To test the successfulness, the duration of the execution of a feature is recorded and can be compared to the effectiveness of the implementation in other database management systems. Except for programming abstraction, all the dependent variables are measured multiple times

for the different experiments. This section describes the variables measured by the experiments.

The dependent variables are all measured as time spans of milliseconds. When a test is performed, the start time is recorded as a timestamp. Once the operation finishes, the end time is recorded as a timestamp. The duration is then calculated as a time span by subtracting the start time from the end time. The start time, end time and time span are all recorded for each experiment. The dependent variables are listed in table 5–4. All these variables measure the latency of performing the operation and are measured in milliseconds.

Table 5-4: Dependent variables

Variable	**Description**
Bulk Data Ingestion Latency	The initial data load execution time is measured for each of the databases.
Ad Hoc Data Insertion Latency	This performance benchmark is of interest in distributed data nodes and provides an indication of performance of distributed key generation.
Data Retrieval Latency of a Single Record	Latency in retrieving a specific record by providing a key is measured.
Data Removal Latency	Specific records are deleted or removed from the data store and latency is measured to determine the performance of these tasks.
Data Retrieval Request Latency	A dataset of a certain amount of records (X) is retrieved from the database and the response time in requesting the dataset is measured. This request only retrieves the first X number of records and no filtering or sorting is performed. Since the NoSQL databases do not have SQL processors, the queries presented here are translated into equivalent queries that may be

	performed on the data store. The query performed for this measurement is equivalent to the SQL query: *select top 1000 * from table.*
Data Scan Latency	Scan queries are queries that return a specified amount of records, but start at a defined record (Cooper *et al.*, 2010:147). In some cases, this starting point can be selected randomly. This performance measurement is the latency in returning a result to the query. This type of query can be described as: *Select next 100 records from table starting at key='xxx'.*
Data Sort Latency	Data requests are often sorted by some field. This variable measures the latency in returning a dataset, which is sorted. The query executed here can be described as: *select all records from table order by field*.
Regular Expression Filter Query Latency	Regular Expression Filter (or Grep) queries have been used by Pavlo *et al*. (2009) and Shi *et al*. (2010) in their benchmarking. Data is requested that should match an expression in the specified field. The query executed for this result can be described as: *select * from table where field like '%ABC%'.*
Range Query Latency	Data is requested from the database that falls within a range. The query executed can be described as: *select * from table where field > x and field < y*.
Aggregation Latency	Aggregation type queries are useful in data analysis and perform calculations like *sum()* on data. This variable measures the performance in returning results where

	aggregation is performed on datasets. The query executed here can be described as: *select field1, sum(field2) from table group by field1*.

In addition to the dependent variable listed in table 5–4, programmatic overhead is measured by looking at the complexity required to implement a client application to interact with each data store. Each of the databases is compared by the API functions available to the programmer and the relative ease to access the data. Unlike the other dependent variables, programming abstraction will be measured only once while implementing a client.

The programming overhead is measured by the capabilities of each database management system's API to implement a client that can:

- Insert new data
- Retrieve data
- Delete data
- Manipulate data.

The benchmark application was implemented in such a way that the same functions were exposed by the tests of each database. This allowed all benchmarks to have the same function or method, and only the length of each method could be measured. Although there were different object-oriented metrics to determine software complexity (Kumar & Kaur 2011), the Source Lines of Code (SLOC) (Bhatt *et al*. 2012) metric was used to determine the effort required to access the data.

Since all the implementations tested here had a Java API, Java was used as the development language for the client application.

The dependent variables were measured and for each instance of the database the results are stored for analysis. The dependent variables were designed to test the data management and data warehousing capabilities of different database implementations. Apart from database functions, the programming language abstraction was also considered a dependent variable.

The independent variables will be considered next.

Independent variables

The independent variables are the variables that are modified and should have an influence on the observations made in an experiment (Welman *et al*. 2005:78; Oates 2010:129). To measure the performance of each of the databases, the following variables can be modified to affect the results: Dataset input size, number of nodes in the clustered environment and the database used.

Input dataset size

The input dataset is raw text data consisting of 58 294 crawled web pages with a total disk size of 5.4 GB. The same input dataset is used for each experiment to test the same dependent variables for each data store.

Number of cluster nodes

Scalability capabilities have a performance impact on the different data stores and to test the impact the experiments are executed against different cluster sizes. The baseline for each experiment is done by using a single node for each data store and thereafter increasing the number of nodes to test the performance impact of hardware scale-up.

Database implementation

The experiments are executed against each of the database implementations, and changing the database should provide observable changes in the data gathered by the experiments.

The independent variables are the variables that are changed to allow the impact to be measured. The benchmark results are measured by executing the experiments, using different size datasets, different numbers of database nodes and different database implementations.

This section described the data collection and the types of variables experimental research required. The variables, the reasons for them and the measurement of these variables were described. The next section describes the data analysis approach followed by the research.

5.4.3 Analysis

The data created and captured for analysis in this research was done through a benchmarking application, which performed all the experiments. Each of the dependent variables measured

a feature of the database and was implemented as a test in the benchmarking application. The benchmark set was implemented as a benchmark factory, with each data store's implementation implementing the interface defined in the factory class. The factory design pattern was used to define a class or interface, which did not have any implementations of its methods and implementations were provided by subclasses (Gamma, Vlissides, Johnson & Helm 1994:121). The benchmarking client recorded the detail and results of each experiment in a database for analysis. Data collected through the experiments were used to analyse each data store's capabilities in terms of data management and warehousing, scalability and level of programming abstraction.

An analysis of the results of each of these capabilities is discussed next.

Programming abstraction

Programming overhead is determined by the development of a benchmarking application, which executes the experiments and records the results in a database. Ease of implementing the following features is measured:

- Bulk and ad hoc data inserts
- Single record retrieval
- Bulk dataset retrieval
- Record deletion
- Data scanning
- Data sorting
- Regular expression filtering (Grep) type queries
- Range queries
- Aggregation.

Figure 5.2 below provides a graphical overview of the high-level architecture implemented in the benchmarking application. The benchmark application uses a common result recorder, which records the detail and result of each of the experiments. A general purpose logger and program log are available to provide details on the execution of the application. To prevent logging from having any effect on latency, no information is logged during the execution of a process, which measures database performance.

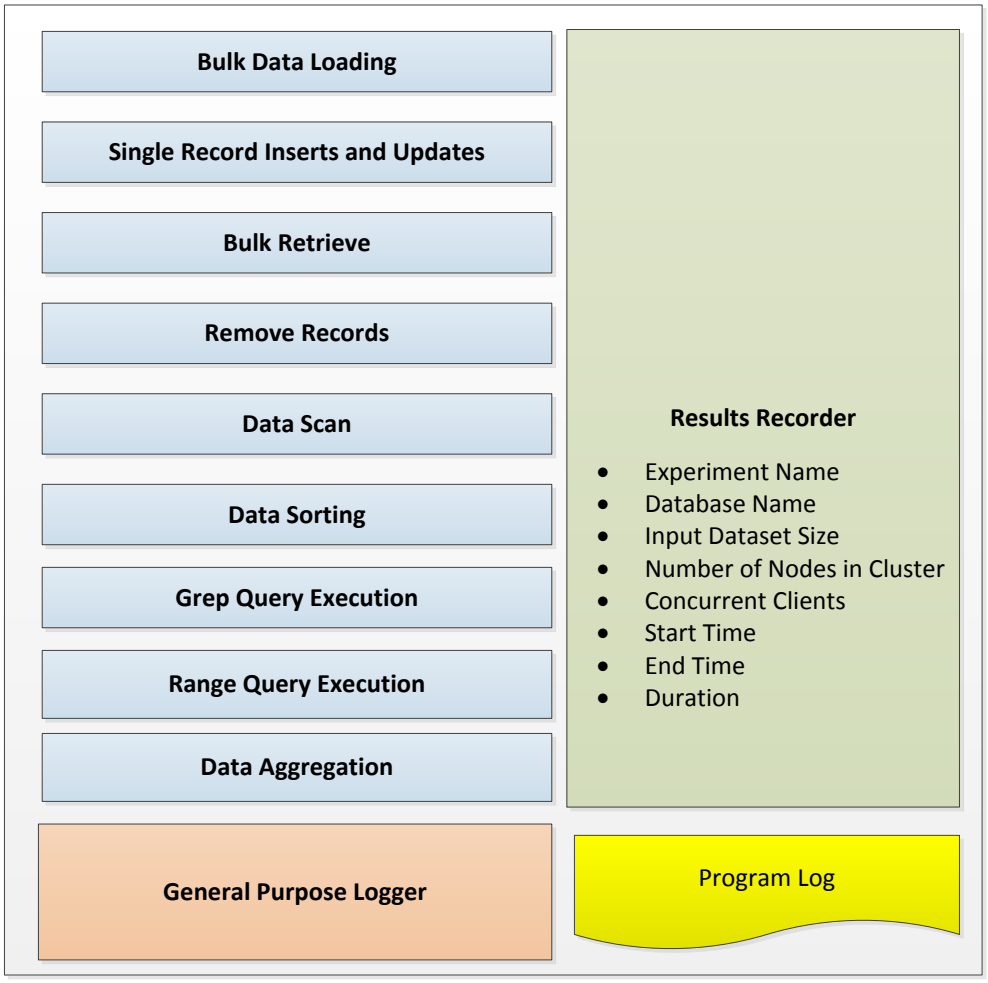


Figure 5-2: Benchmark program structure

Ease of implementation is measured in the ability to implement each of the features with out-of-the-box functions instead of writing additional code to support the function. Features in support of the benchmarking functions are compared for each data store implementation. The use of a factory design pattern in implementing the client allows all database implementations to expose the same set of functions. Actual execution code is then implemented in a child class for each database. The Source Lines of Code (SLOC) metric is then used to provide insight into how difficult it will be to implement a feature. The assumption is that fewer source lines of code should be easier.

Features can be tested in any order and do not execute sequentially. This is useful when only certain features are tested.

Data management and warehousing

The benchmarking application recorded the performance of data ingestion, data retrieval and management as well as data warehousing capabilities for each of the databases. The results of each experiment were then tabled for each database and compared to the results of the other databases.

For each of the experiments, the following were recorded:

1. Experiment executed
2. Name of data store used
3. Size of dataset to load in number of records
4. Number of nodes currently used
5. Start time of experiment
6. End time of experiment
7. Duration of execution in milliseconds.

The results were recorded in table format and the analysis was done by comparing the results of each data store on performance on the following metrics:

- Size of dataset used
- Number of nodes used.

All experiments used the same dataset for each test to ensure consistency in the performance measurement.

Scalability

The scalability metric was used to measure the ability to scale large volumes of data over multiple database servers. Scalability for the purpose of this research was not measured in the ability to scale in the number of operations performed in a certain timeframe. Scalability was measured by the benchmark application against the number of nodes and performance fluctuations when accessing the data. Scalability was measured by running the tests of different data management and data warehousing features (refer to the dependent variables) against a cluster consisting of different node configurations. Tests were performed against one node, then two nodes, three nodes and finally four nodes.

The same results were recorded as for the data management and data warehousing results, only taking into account the different number of nodes in the cluster. The results were recorded in table format in a database for each of the databases and compared to the results of the other databases.

This section provided the details of the data analysis done and the specific results that had been recorded for analysis purposes. The results were used in table format to generate graphs for interpretation. Measurement of data management, data warehousing capabilities and scalability were measured continuously when experiments were being performed. Programming abstraction was measured once while the benchmarking tool was implemented.

The next section discusses the need for controls in experimental research and provides the details of how control is achieved.

5.4.4 Controls

Experimental research aims to prove that only one factor causes observable change. Therefore, the researcher has attempted to control all variables in some manner (Oates 2010:129). One of the characteristics of experimental research is the control over the independent variable (Welman *et al*. 2005:79).

In order to provide control, the experiments used the same input dataset on the same cluster configuration while only changing the amount of data, the number of nodes and the database server being tested. Full control was maintained over the size of the dataset and number of nodes or for each experiment. The experiments were executed, using the same benchmarking application on the same client machine. The same type of results was recorded for each experiment through a uniform results recorder.

Since the research was to determine if the alternative databases were delivering on the promise of addressing RDBMS shortcomings, the same experiments were performed against a RDBMS. Since the research focused on open source implementations of non-relational data stores, an open source RDBMS, namely MySQL was used. The reason for using MySQL is that it is a non-commercial implementation like the rest of the database servers tested.

Although there are various other open source relational databases available (PostgreSQL, Firebird, etc.), MySQL has been chosen because of its popularity. According to the DB-Engines Ranking website[14], MySQL has been ranked second after Oracle and above Microsoft SQL server, both being proprietary database systems.

The research group or participants consisted of open source implementations of each of the main classifications of distributed non-relational data stores, namely column-oriented databases, document-oriented databases, and Key/Value databases. For the purpose of the research, no tweaking was performed on any database. Features were tested as they were available at install time.

Experimental research can be conducted in either a laboratory environment or in the field. Laboratory studies are conducted in a well-constructed environment where maximum control exists. Field studies are conducted in the natural environment and have many external factors or nuisance variables that can influence the research (Welman *et al*. 2005:85-86).

This research was conducted in a controlled environment, using the same platform each data store. The same hardware and operating system were used with the same software patches loaded.

The next section discusses the measurement and observation requirements of experimental research and how these relate to this research.

5.4.5 Measurements and observation

Measurement and observation form the basis of experimental research and this section relates these factors to the research presented here. Experimental research constitutes quantitative research (Welman *et al*. 2005:78). Quantitative data is normally used to measure the observable change (Oates 2010:131). The goal of experiments is not to observe the effects, but to prove a theory (Olivier 2009:75). The goal of the research presented here is to prove whether non-relational databases improve on RDBMS as claimed. This means that the aim is first to prove that a data store implements a certain feature as a way of addressing a specific

[14] Available at http://db-engines.com/en/ranking

limitation in RDBMS, and then to measure it to determine how well that feature is implemented.

When the feature tested (the dependent variable) does exist for a certain configuration and database implementation (independent variables), the result is measured in duration of executing the task or experiment. According to Oates (2010:130), time can be measured in experiments. These results are used for comparison of the effectiveness of each database model in providing a feature like data management, warehousing or scalability.

Measurements are taken through the use of a benchmarking application, which tests each database feature through an experiment. For each experiment, the following measurements are recorded:

1. Experiment name, for example "Bulk Data Load"
2. Data store name, for example "Cassandra"
3. Input dataset size, for example "20000 records"
4. Number of nodes in the cluster, for example "4 nodes"
5. Start time of the experiment
6. End time of the experiment
7. Duration of the experiment.

Observations are made on the impact of changes in the number of records of the input dataset, number of nodes in the cluster and number of concurrent clients for each of the data store implementations.

The programming abstraction measured the ease at which a developer might implement a feature or function. Although different complexity metrics exist, the simplest metric of Source Lines of Code (SLOC) was used. The reason for using this metric was because of the use of a factory design pattern that had been used when implementing the benchmark application. In essence, the same set of methods was available for each database implementation; for example, for each database the benchmark application implemented a *BulkInsert()* method. Since each benchmark child class had the same method, counting the source lines of code to implement the method could give a comparable result on the effort required to use a feature from a programmer’s point of view.

This section has given an overview of the controls used to test the features of each database through the benchmark application. The next section discusses the validity of the research.

5.4.6 Validity

This research provides an answer to the question of whether the alternative databases address the limitations of RDBMSs in storing user-generated text documents in distributed environments. Consideration must, therefore, be given to whether the use of an experimental approach yields the desired results in answering the research question.

The experiments were designed to test the different features of each database that was highlighted as a limitation of RDBMSs, namely the capability to store user-generated text, provide data analysis and data warehousing capabilities as well as the ability to store data over multiple servers in a distributed environment.

In the simplest form, a database's ability to perform an experiment is an indication of whether the limitation has been addressed or not. The more limitations addressed, the better the database implementation at addressing the limitations. By measuring the performance of the database's ability to perform an experiment insight is gained into whether the limitation is addressed more efficiently compared to other databases.

Experimental research requires internal and external validity to be considered successful. Internal validity means that the observations are indeed caused by the interventions of the researcher (Oates 2010:131; Olivier 2009:75). According to Wellman *et al*. (2005:107), internal validity is the degree to which changes in the independent variable cause the observed result. The minimum requirement for internal validity is through comparison (Welman *et al*. 2005:107).

This research was conducted by executing experiments against different database implementations on the same hardware, each representing a different participant of the group, which was used for a comparison of the databases. The recorded results were then compared to the results of the other participants.

External validity refers to the generalisation of research results observed in other situations different from the research environment (Olivier 2009:75). The aim of experiments is to produce results with a high level of external validity (Oates 2010:133). Overreliance on certain participants in the research, too few participants and non-representative participants can cause low-level external validity (Oates 2010:133). According to Olivier (2009:75), participants of a research group may include hardware platforms and operating systems. For the purpose of this research, the participants are the various non-relational database implementations. Since a model and implementation from each of the categories (column-oriented databases, document-oriented databases and Key/Value databases) have been used for the experiments, the requirements for a high level of external validity are implied.

The measurement and analysis of data recorded during the experiments were, therefore, used to determine whether:

1. The database implemented the features required and if it addressed the limitation presented in RDBMSs
2. This feature had been implemented successfully when compared to other participants in this research.

Finding the answers to these two questions in essence answers the research question. Conclusions can then be made on whether non-relational databases do indeed address the limitations of RDBMSs when storing user generated content in distributed environments.

This section described the validity of the experimental research in answering the research question. Furthermore, the internal and external validity required for research were discussed and highlighted how these kinds of validity had been applied to the research presented here.

The next section discusses the limitations of this research.

5.5 Limitations

The experimental method followed for this research had some limitations, which was discussed in this section. According to the NoSQL database[15] website, there were 150 different implementations on non-relational databases, which would make an exhaustive

[15] Available at http://nosql-database.org/

evaluation difficult to perform within a reasonable timeframe. As a result, only the most well-known implementations that supported Java as a common programming interface were selected for this research.

The research also focused on the limitation of RDBMSs of storing user-generated text-based content. The input dataset used for experimentation represented data that was not relational. Since only non-relational databases were considered, graph databases, which were classified as NoSQL databases, were excluded from the experiments, since these databases were highly focused on relationships.

Another limitation was the size of the input dataset used for the evaluation. Since it was retrieved from the Web by the researcher, only a small dataset was used, which might have an influence on the scalability tests of the databases. The dataset size used for the bulk insert experiments was also required to be loaded into the client machine's memory before tests were executed to eliminate secondary storage read performance at client level, while the actual database functions were tested and, therefore, only a limited dataset size was used. It needs to be noted that only the client machine loaded the dataset in-memory. The actual database performance measured was on normal database performance which included normal disk I/O operations of each database server.

A small distributed environment of four servers was used, which proved sufficient for generating data for this research. This might not give a true reflection of the scalability and processing capabilities that might become evident in much larger data farms, which are truly at internet scale. The results of the scalability experiments for this research could yield more detailed results if it could be executed in large-scale data centres consisting of hundreds of servers and using very large datasets as input. However, this research focuses on the concept of scalability, and larger environments could distract from the aim of this research, which is to prove that scalability is supported.

The database implementations tested were also default installation options with no performance tweaking or any additional modules installed other than those that form part of the standard installation package. Performance tweaking of databases could have an impact on the results measured while performing experiments.

For the data warehousing-related experiments, the features available at installation time were tested. Therefore, no data warehousing-related tasks like data scrubbing and cleaning and transformation were done for any of the implementations. Data warehousing in itself is a large field, and for the purpose of this research only the essential requirements were tested.

Another consideration for organisations to select a database implementation is the security measures databases provide in order to store data securely. For the purpose of this research the security aspects were not considered, as they do not provide any value in determining whether non-relational databases provide enhancements in storing user-generated text documents in distributed environments. This metric might, however, be considered in future research when other applications areas were tested.

This section discussed the limitations of the research in terms of the environment used. It also provided arguments on why certain decisions were made and what influence the decisions might have. The next section presents related research.

5.6 Related research

This section highlights research that is similar to the research presented here, and which also evaluates non-relational databases.

Hecht and Jablonski (2011) evaluated NoSQL implementations against the requirements that arose from Web 2.0 applications through a use-case survey. In their evaluation, they compared Key/Value stores, Document stores, Column family stores and Graph databases against Query possibilities, Concurrency control, Partitioning and Replication and consistency.

Moussa (2012) evaluated OLAP performance of NoSQL implementations in a cloud environment though Hadoop[16] and Pig[17] using TPC-H benchmarks. The approach was to use the standard TPC-H benchmark schema and to convert all the SQL queries of the benchmark into Pig Latin scripts. This scenario, therefore, relied on relational data for doing the benchmarking, which was in contrast with the NoSQL view of non-relational data.

[16] Available at http://hadoop.apache.org/
[17] Available at http://pig.apache.org/

The research presented here is to evaluate non-relational databases in terms of the capabilities to store unstructured user-generated text content, which does not imply relationships.

Cooper *et al*. (2010) proposed a benchmarking framework for cloud-based serving systems, which focus on performance and elasticity of cloud-based data stores. The focus was on read/write access to data stores for Web serving systems, based on a Web user requesting information from a website. The framework consisted of a client generating various workloads such as read-heavy workloads, write-heavy workloads and scan workloads. Analytical and batch processes used for OLAP were not part of the performance benchmarking, which forms part of this research.

Shi *et al*. (2010) performed benchmarking on cloud-based systems by evaluating performance of data read and write operations as well as data load and structured querying. The benchmarking was limited to HBase and Cassandra. For structured querying, they translated the queries programmatically into API calls for each implementation. They found that in some instances, for example, HBase using the Map-Reduce framework, performances were better than with RDBMSs. Fault tolerance on query processing was also simulated by removing nodes while executing queries.

The research presented here tries to extend the benchmarking to other non-relational data stores. This section has presented research that is similar in that non-relational databases have been evaluated. The conclusion to this chapter will be presented next.

5.7 Conclusion

This chapter described the research method followed. The chapter started with an overview of the research. An overview of the research process was presented to provide a background of the different types of research that exist. It then presented a discussion of why the experimental research method had been considered, and what the aim and objectives or the method should be.

The research design was then discussed and arguments were made as to why the experimental method had been chosen. An overview was also given on the steps followed to perform the research.

The research method was then explained in detail and information provided on how the experimental method was applied. The research environment was presented and the data collection process was described. The research data analysis approach was discussed, and the controls and measurements used were presented. Finally, the validity of the research and the requirements for research validity were presented. This was followed by the limitations of this research as well as a discussion of the difference between this research and similar research that exists.

The next chapter presents the results of the experiments.

CHAPTER 6

RESULTS OF THE EXPERIMENTS

The research design and experiments conducted for this research was described in the previous chapter. The performance of each database was measured by a benchmarking application, as explained in the methodology chapter.

The results of the experiments are presented in this chapter. The next chapter presents the research finding based on the results presented here.

This chapter has the following structure:

Section 6.1 presents and discusses the results of the benchmarking application.

Section 6.2 discusses the analysis of complexity.

Section 6.3 provides the conclusion of this chapter.

6.1 Benchmark results

The previous chapter described the experimental approach followed and the dependent variables that were measured to determine whether certain capabilities had been implemented in the different databases. Furthermore, the performance was measured to draw a comparison between the results of different database implementations to determine how successful these features had been implemented in relation to the other databases.

This section is divided into different subsections, each of which will focus on a different data management feature. The data ingestion capabilities are discussed first, followed by the data retrieval and data warehousing features.

6.1.1 Data ingestion

This section presents the results and analysis of the data insertion capabilities of the different database systems. Data insertion latency was recorded for different dataset sizes and for different database cluster sizes. The data recorded for each database will be presented and then compared at the end of this section. The average duration of data ingestion is displayed on the graphs as the time in milliseconds required for each dataset size over different number of nodes used to achieve the result. Each database system will now be discussed.

Cassandra

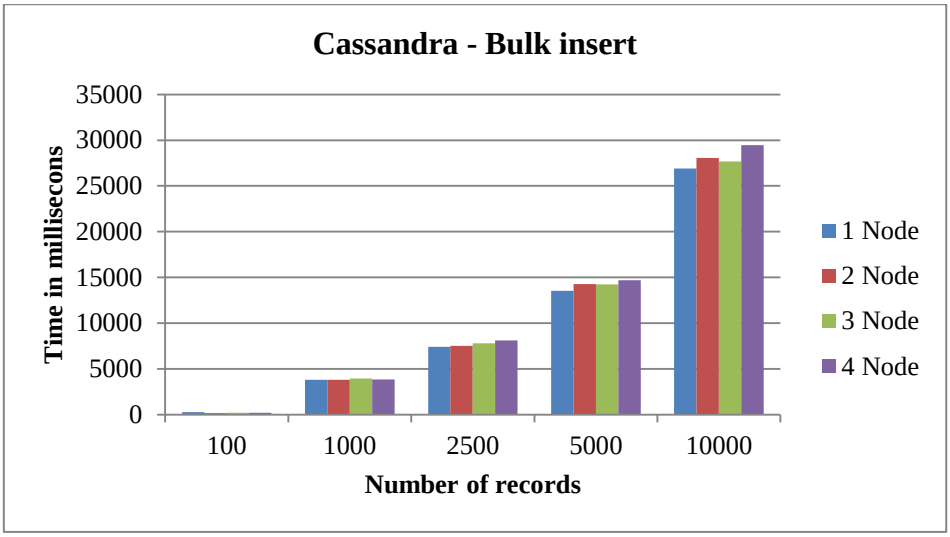


Figure 6-1: Cassandra bulk load

Figure 6-1 shows the results for bulk data inserts into Cassandra for different record set sizes. Cassandra on average inserted a record between two and four milliseconds across all dataset sizes. The longest average time recorded to insert a record was within the 1 000 record dataset size, reaching 3.8 millisecond averages. The average dropped to just below three milliseconds per record for dataset sizes of 5 000 and 10 000 records. Additional nodes provided additional capacity, but in terms of performance gains the additional latency in communication took up additional time, adding two seconds when inserting 10 000 records on four nodes versus inserting them on one node. Cassandra did not allow bulk insertion of a dataset larger than 10 000 records; therefore, results were only captured up to 10 000 records.

HBase

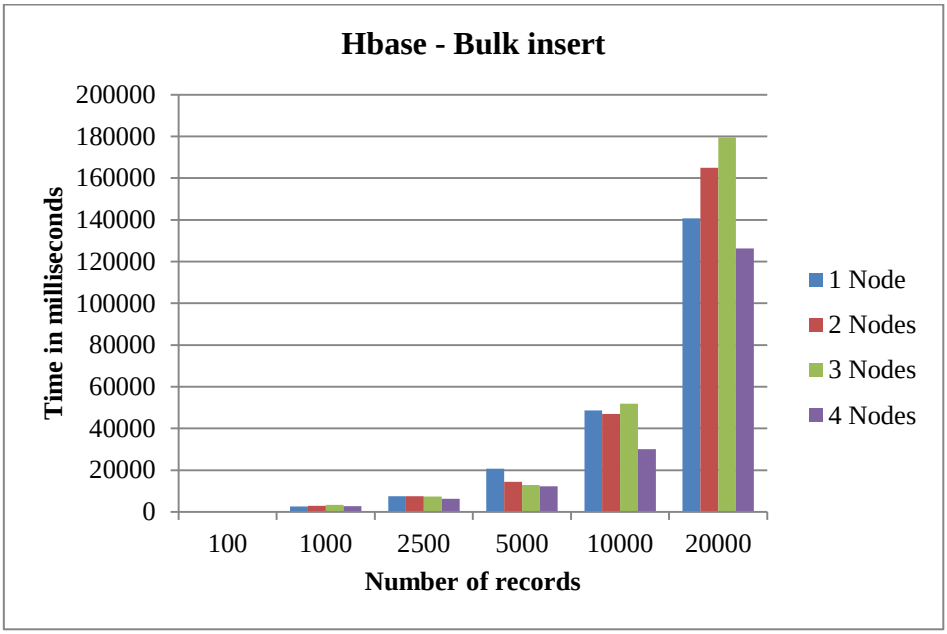


Figure 6-2: HBase bulk inserts

HBase recorded an increase in the average time to insert a record as the dataset sizes increased. Figure 6-2 shows the bulk insert performance for HBase. At 100 records the average time per record was at 2.5 milliseconds, whereas at 20 000 records the average time was at 7.6 milliseconds. HBase recorded improvements on data insertion with the addition of data nodes. At 10 000 and 20 000 records the addition of a fourth node recorded a significant increase in performance over one, two and three nodes.

MongoDB

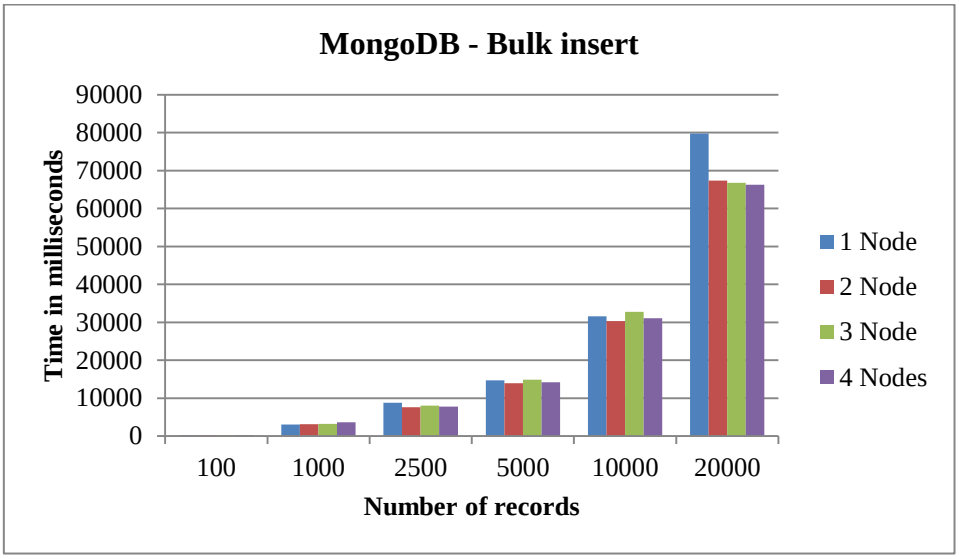


Figure 6-3: MongoDB bulk insert

MongoDB recorded constant average times per record inserted across the different dataset sizes and the number of nodes used. Figure 6-3 shows the bulk insert performance of MongoDB. On average, insertion took between 2.5 and 3.5 milliseconds per insert. Adding additional nodes did not improve on insertion, and variances in time can be contributed to inter-database communication. At 20 000 records there was a spike in the insertion time to a single node, which was about ten seconds more than the average of two, three and four nodes. This could be attributed to the fact that MongoDB optimised itself based on what the client application was doing.

CouchDB

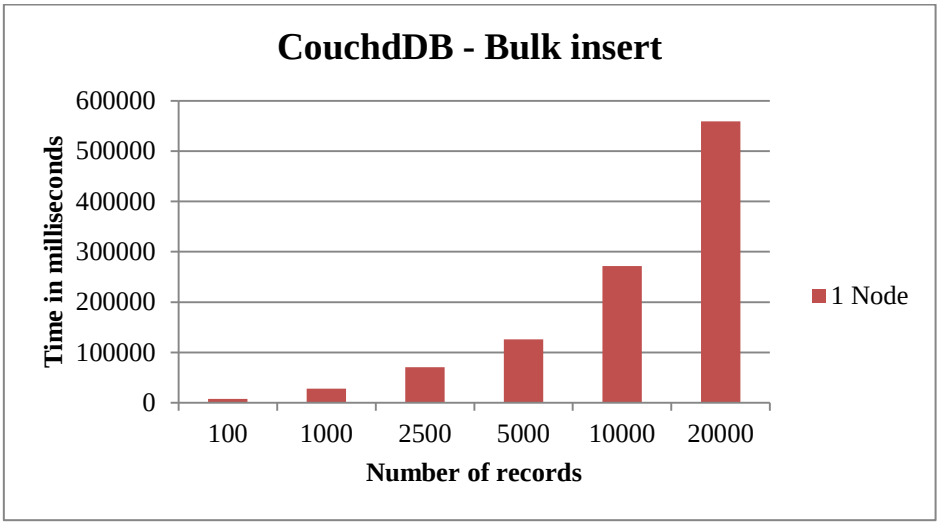


Figure 6-4: CouchDB bulk insert on one node

Data insertion in CouchDB was tested using only one node. Figure 6-4 illustrates the performance of CouchDB with bulk inserts. The reason being that CouchDB only provided replication, which allowed data to be replicated to different servers; in essence, the same data was on each server. In order to allow clustering and data sharding over different partitions, the use of a proxy service was required. This was not a feature built into CouchDB, and as a result, bulk loading was limited to only one node. Data insertion to CouchDB was slower than the other data stores, averaging at 27 milliseconds per record insert, and this was mainly attributed to the poor performance of the REST interface, which required the use of an HTTP server to communicate with the database. The HTTP server was not optimised for performance.

Redis

Redis does not allow batch loading from the client. Bulk loading is done through a batch load program, which uses an input text file that adheres to the REDIS protocol. As a result, bulk loading could not be tested via the benchmark tool. To complete the experiments, the full dataset was added as single records through the benchmark tool. Although not ideal, an average of 51 records per second could be added, taking on average 19.56ms per record. This was even faster than the bulk insert allowed by CouchDB. Since the bulk insert is not supported from a client side, this data was not recorded for the Redis database server.

Voldemort

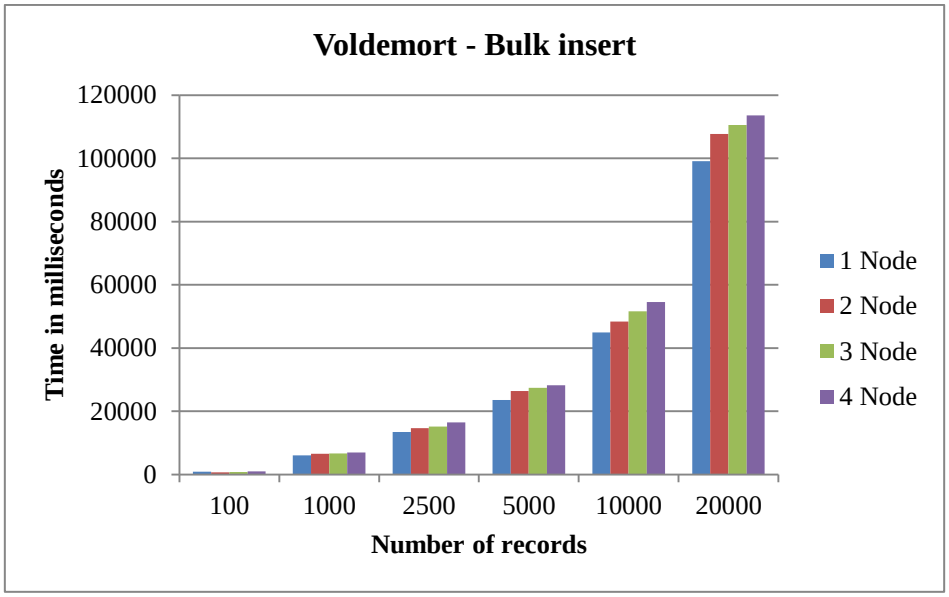


Figure 6-5: Voldemort bulk insert

The average time to insert a record decreased in Voldemort as the dataset size increased. Figure 6-5 shows the bulk insert performance of Voldemort. For 100 records the average insertion time across multiple nodes was 8.3 milliseconds and this decreased to 4.9 for 10 000 records and slightly increased to 5.3 seconds for 20 000 records. By adding nodes, the time to insert records also increased, which can be attributed to additional communication between nodes.

MySQL

For MySQL, only a single node was used for benchmarking. The reason for this was that MySQL only supported scaling through the use of a proxy, which directed data to the correct node. Although data replication was available, the same data wass in essence replicated to the other servers and would not provide any benefits from a scalability point of view.

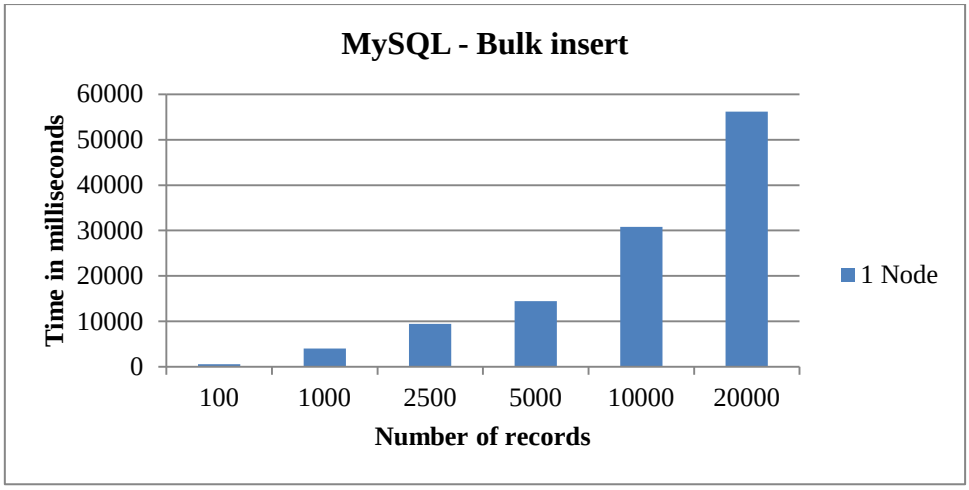


Figure 6-6: MySQL bulk insert

MySQL on average added a record in three milliseconds and remained fairly consistent through different dataset sizes. Figure 6-6 shows the bulk insert performance for MySQL for different record set sizes. Although the insertion times were faster, it should be noted that the schema constraint of fixed row sizes required data to be truncated before insertion. As a result, the content of the documents could not be stored completely, as some of the text documents had more characters than the columns would allow.

The individual results for each data store were presented up until now. The recorded results will now be compared to see how each of the databases performed when compared to the other databases.

Comparison

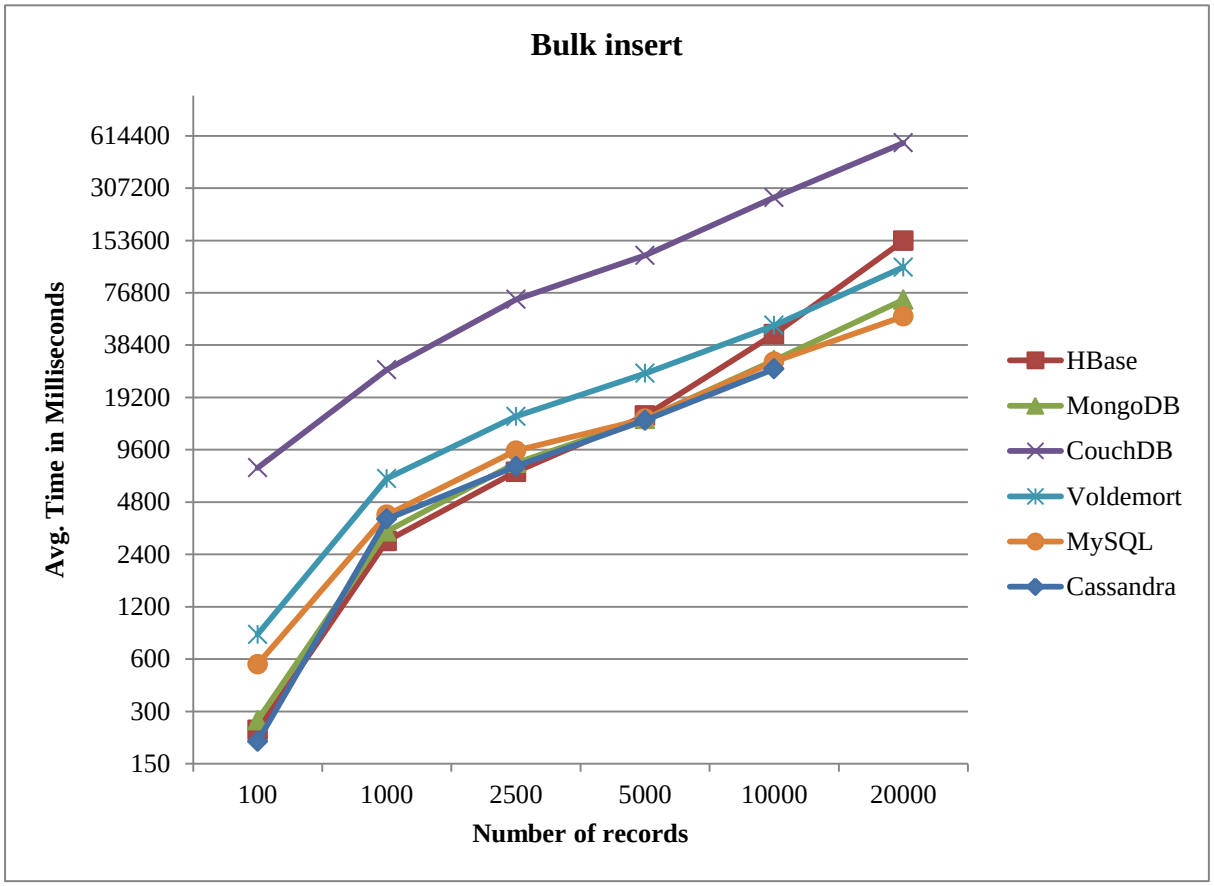


Figure 6-7: Comparison of bulk performance

Figure 6-7 compares the bulk insert performance of the difference databases. Since Redis did not allow bulk inserts from the benchmarking application, its performance was not plotted. Cassandra only allowed bulk insert for datasets up to 10 000 records. Although MySQL outperformed the other databases, it should be noted that this was for records with a size of less than 65 KB while no multi-node clustering, which would require replication and sharding, was used. CouchDB was the worst performer, and this could be attributed to the use of a REST interface requiring updates to be handled through an HTTP server. Even though inserting the dataset in Redis one record at a time, it still outperformed CouchDB.

Table 6-1: Data insertion latency showing records per second

Data store	Records per second for 58 294 records
MySQL	428.03
Cassandra	290.57
MongoDB	198.62
HBase	194.26
Voldemort	165.57
Redis	51.05
CouchDB	30.2

Table 6–1 provides the results on the average records inserted per second when inserting the whole input dataset. The dataset consisted of 58 294 webpages crawled by the web crawler as described in chapter 5 section 5.4.1. Cassandra recorded the best results for data insertion, with HBase and MongoDB performing similar. Redis performance was based on the insertion of one record at a time, since the bulk load protocol was not used.

This section presented the results of performing bulk data inserts. The next section will provide the data and analysis of data retrieval capabilities of the different databases.

6.1.2 Data retrieval

In this section, the data retrieval results are discussed. Data retrieval latency is important in data warehousing applications where large datasets are queried. The performance of each database is discussed before a comparison is made at the end of this section.

Cassandra

Retrieving single records containing all columns on average took 14 milliseconds per retrieve, longer than retrieving bulk data**,** which averaged between 2.8 and three milliseconds per retrieve. Retrieving bulk datasets could be done up to 5 000 records before the Thrift protocol's (which is used by Cassandra to communicate with the database server, as described in section 4.2.1) transport size limit of 15 MB was reached. To retrieve larger datasets, batches were used, which retrieved records at an average retrieval rate of 2.8 records per millisecond.

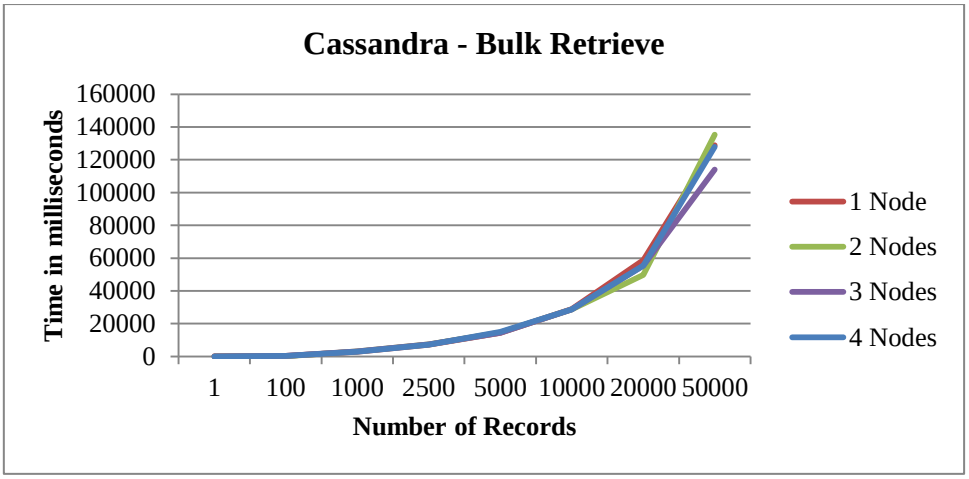


Figure 6-8: Cassandra bulk retrieve latency for different record set sizes

Figure 6-8 shows the bulk data retrieval latency for different record set sizes. Since Cassandra is a column-oriented data store, it allows specifying slices. This allowed the client to retrieve only the columns information it wanted. Retrieving only certain columns would have an impact on performance, but for the experiment only full records were retrieved since this full set of columns was requested from all the database servers, and would provide response time as compared to the responses by all the database servers tested.

Table 6-2: Time in milliseconds to retrieve datasets of different sizes from Cassandra

Result size	1	100	1000	2 500	5 000	10 000	20 000	50 000
One node	47	316	2 955	7 266	14391	28870	58 800	128 955
Two nodes	25	235	3 016	7 313	14497	28713	49 840	135 252
Three nodes	26	328	2 990	7 345	14371	28636	55 485	114 081
Four nodes	18	245	2383	6014	12209	23796	45893	92698

The addition of nodes to the cluster did not have any significant impact on smaller datasets, with the exception of retrieving one record, which was slowest on one node. Table 6-2 presents the data retrieval time over different node sizes for different dataset sizes. For result sets of 20 000 and 50 000 records there was some variation in response times with the addition of nodes speeding up response times. At 20 000 records a 1-node cluster yielded slower response rates than a 4-node cluster. At 50 000 records the 4-node cluster yielded the fastest response times. For 50000 records, the addition of a second node decreased the response time with about 7 seconds. However the addition of nodes subsequently increased query performance. In all tests the addition of the fourth node did show an increase in response speed.

Table 6-3: Cassandra data read average over four nodes

Records	1	100	1 000	2 500	5 000	10 000	20 000	50 000
Duration in ms (average)	14.4	298	2 815.4	7 227.9	15 013.9	28 607.2	55 445.5	127 901
Latency per record in ms	14.4	2.98	2.8154	2.89116	3.00278	2.86072	2.77228	2.55802
Records per second	69.444	335.57	355.189	345.882	333.025	349.562	360.715	390.927

With the exception of the retrieval of one record, increasing the size of the result set caused an increase in the number of records retrieved per second between 100 and 1 000 records, which decreased up to 5 000 records. Since records were retrieved in batches of 5 000, increasing the result set size showed performance gains in the number of records returned.

HBase

HBase returned single records on average in 3.7 ms. Returning records in a batch yielded faster retrieval times, ranging from 1.8 ms in sets of 100 records to 2.2 ms in sets of 2 500 records. Retrieving result sets larger than 4 000 records caused communication timeouts between nodes. In order to do a bulk retrieve, the client had to create a list of *Get* objects, specifying the keys of the records to retrieve. The creation of the list of *Get* objects added additional processing time to the client. The experiment could, therefore, only produce results for queries up to 2 500 records for comparison purposes.

Table 6-4: HBase data read average over four nodes

Records	1	100	1000	2500
Duration in ms (average)	3.7	189.8	1614.8	5574.1
Latency per record in ms	3.7	1.898	1.6148	2.22964
Records per second	270.27	526.87	619.272	448.503

Requesting single records also yielded slower response times per record than batches. HBase recorded the fastest response times when reading datasets of 1 000 records, achieving an average throughput of 619 records per second. Returning result sets of 2 500 records on average was slower. Running a retrieval experiment for the first time recorded slower response times than subsequent tests requesting the same results. Retrieving 2 500 records for the first time took 16.3 seconds, while the average for subsequent queries of the same data took 4.4 seconds.

MongoDB

On average, MongoDB returned single records in 5.6 ms. Retrieving batch result sets took on average between 2.1 ms for 100 records and 2.35 ms for 10 000 records. MongoDB by default returned larger result sizes to the client, allowing result sets of 20 000 records, which on average took 3.8 seconds per record. MongoDB also returned a document set of 50 000 records in 179 seconds, which took 251 seconds to insert, indicating that it is optimised for reading.

Table 6-5 MongoDB: data read average over four nodes

Records	**1**	**100**	**1 000**	**2 500**	**5 000**	**10 000**	**20 000**	**50 000**
Duration in ms (average)	17.6	314.8	2 414	6 089.2	12 610.9	25 378	56 114.8	176 203
Latency per record in ms	17.6	3.148	2.414	2.43568	2.52218	2.5378	2.80574	3.52406
Records per second	56.8182	317.662	414.25	410.563	396.482	394.042	356.412	283.764

Retrieving one record took on average 17.6ms, which is slower than returning records in batches. Increasing the result set size had an impact on performance. The best performance was recorded when retrieving 1000 records. Increasing the number of records in the result set had an impact on performance. MongoDB returned the larger result sets without requiring the use of batches.

CouchDB

The REST API used by CouchDB made data retrieval slow when doing bulk retrieves. Returning 100 records on average took 27.9 seconds, which was comparable to the time that Cassandra, HBase and MongoDB returned 10 000 records. Result sets of 1 000 records were retrieved in 7.1 minutes, while 2 500 records took 26.2 minutes to retrieve. Queries larger than 5 000 records timed out and could not be retrieved.

Table 6-6: CouchDB data read average over one node

Records	**1**	**100**	**1 000**	**2 500**
Duration in ms (average)	21.6	27 982.9	429 937	1 570 161
Latency per record in ms	21.6	279.829	429.937	628.064
Records per second	46.2963	3.57361	2.32592	1.59219

Data reads on CouchDB were slow in comparison with the other databases, achieving only 1.5 records per second for result sets of 2 500 records. Only one node was used due to the scalability limitations of CouchDB, as described in section 7.3 in which the scalability findings were presented.

Redis

Redis was able to return result sets with up to 20 000 records. Reading a single record on average took 9.3 ms, which was slower than bulk retrieves. As the result set increased, the duration also increased with the best performance measured queries requesting 1 000 records. The duration of queries per record ranged between the fastest of 3.26 ms per record for a 1 000 records to the slowest of 4.4 ms per record for 20 000 records.

Table 6-7: Redis data read average over four nodes

Records	**1**	**100**	**1 000**	**2 500**	**5 000**	**10 000**	**20 000**
Duration in ms (average)	9.3	382.9	3 258.9	8 188.5	16 644.9	34 176.4	8 8079.1
Latency per record in ms	9.3	3.829	3.2589	3.2754	3.32898	3.41764	4.40396
Records per second	107.527	261.165	306.852	305.306	300.392	292.6	227.069

Redis was able to serve around 300 records per second when result sets between 1 000 and 5 000 records were requested. Requests of 100 records returned around 261 records per second, which were slower than requesting 1 000 records, which averaged at 292.6 records per second. Requesting 20 000 records recorded the slowest performance of only 227 records per second, but requests were more than double as fast as requesting single records.

Voldemort

Voldemort could also serve queries of up to 20 000 records. One of the key features of Voldemort was fast reads and it managed to return batch data in under 2 ms per record for batches up to 5 000 records. Requesting a single record was slower at 9.5 ms per record. Increasing the number of records requested slowed down performance, with 10 000 records returned at an average of 2.8 ms per record and 20 000 records averaging at 3.16 ms per record.

Table 6-8: Voldemort data read average over four nodes

Records	1	100	1 000	2 500	5 000	10 000	20 000
Duration in ms (average)	9.5	186.7	1 740.5	4 727.3	9 635.1	28 513.8	63 351.3
Latency per record in ms	9.5	1.867	1.7405	1.89092	1.92702	2.85138	3.16757
Records per second	105.263	535.619	574.548	528.843	518.936	350.707	315.7

Voldemort also reported the best performance when requesting batches of 1 000 records and managed to serve 574.6 records per second. Requesting records in batches up to 5 000 records all recorded times above 500 records per second; however, increasing the batch size to 10 000 records caused a sharp decline to only 350 records per second, and 315 records per second for batches of 20 000 records.

MySQL

MySQL performed the best of all the databases, but the limitation on row sizes meant that the content was not complete. This meant that record sizes were smaller, since the whole record had to be less than 65 KB. The average file size and, therefore, the content was 161.99 KB and some files up to 990 KB.

Table 6-9: MySQL data read average over one node

Records	1	100	1 000	2 500	5 000	10 000	20 000	50 000
Duration in ms (average)	6.2	77.9	691	1 789.4	3 630	7 210.3	15 041.5	36 200
Latency per record in ms	6.2	0.779	0.691	0.71576	0.726	0.72103	0.75208	0.724
Records per second	161.29	1 283.7	1 447.18	1 397.12	1 377.41	1 386.9	1 329.65	1 381.22

The results for data reads of each database were individually presented up until now. The read performance of the databases will now be compared with each other.

Comparison

All databases recorded slower response times when requesting single records than when multiple records were recorded. For single records, HBase recorded the fastest response times of 3.7 ms followed by MongoDB at 5.6 ms and then Voldemort at 9.5 ms. Cassandra and Redis returned a single record in 14 ms, and CouchDB performed the poorest with 21 ms.

Since HBase experienced timeouts between nodes at around 4 000 records, only queries up to 2 500 records were recorded. Voldemort and HBase recorded the highest throughput rates for smaller queries up to 2 500 records. At 1 000 records, where the databases produced the best

results, HBase performed better than Voldemort. However, at 2 500 records Voldemort served results the fastest at an average of 1,89 ms per record, followed by HBase at 2.23 ms per record. MongoDB was also performing well averaging at 2.44 ms per record. CouchDB was by far the worst performer, recording a read time of 628 ms per record for 2 500 records. For queries larger than 2 500 records, CouchDB did not respond at all.

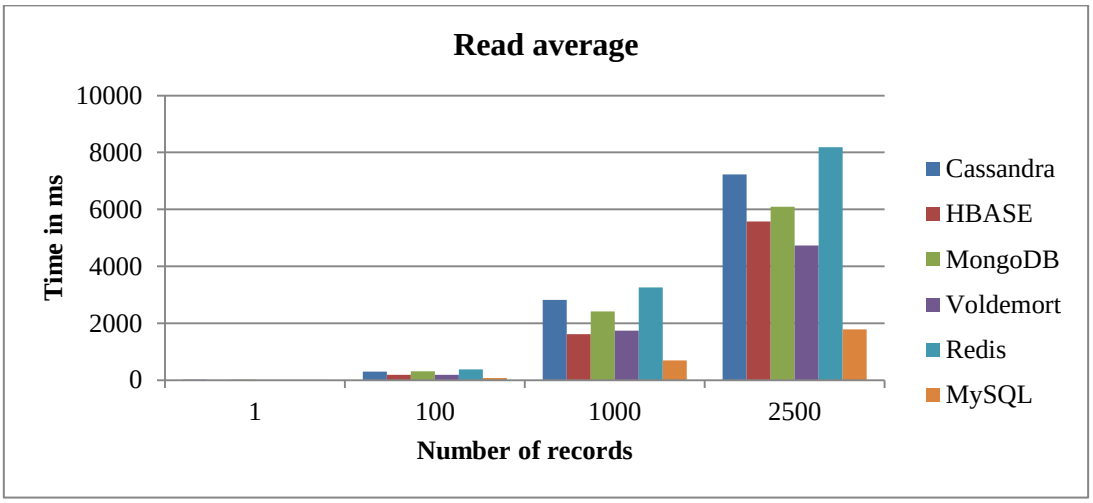


Figure 6-9: Comparison of read averages of small data sets

For results up to 2 500 records, each of the different database models featured in the top three. The Key/Value database, Voldemort, recorded the fastest responses, followed by the column-oriented database (HBase) with the document-oriented database (MongoDB) in third place.

Although MySQL performed significantly faster, it stored only truncated content due to limitations on the row size. At queries of 10 000 and 20 000 records, MongoDB (document-oriented) performed the best, followed by the Cassandra (column-oriented) with the Key/Value stores Voldemort and Redis performing worst.

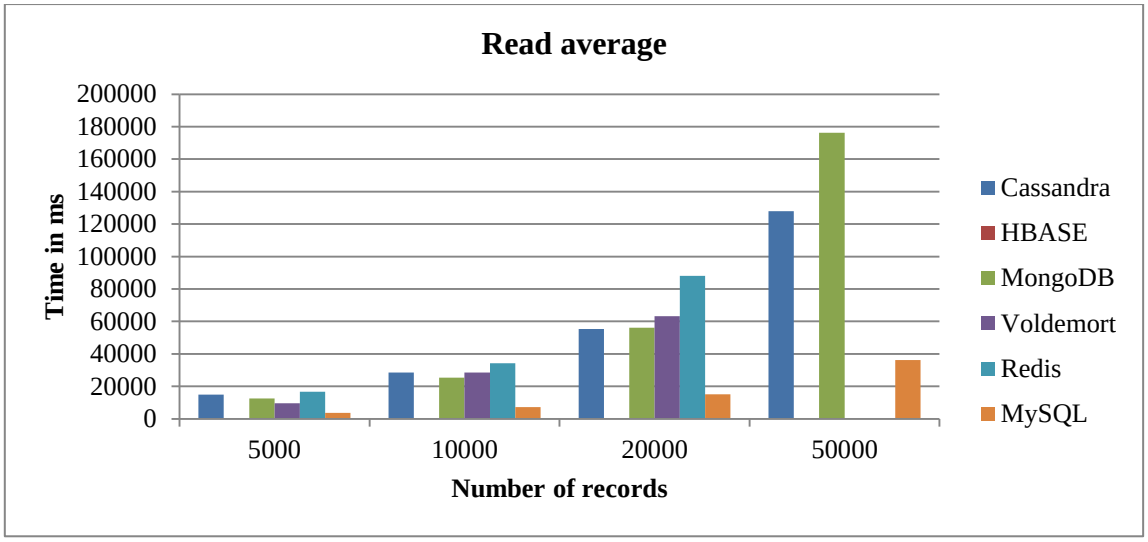


Figure 6-10: Read average on results larger than 2 500 records

At queries of 50 000 records, only Cassandra and MongoDB were able to return results, with Cassandra outperforming MongoDB.

Data retrieval performance was of importance in data warehousing applications. The read performance of each database was presented and analysed before it was compared to the results each other. The next section presents the data from the data scan benchmark tests.

6.1.3 Data scan

Data scan queries retrieved data in bulk with a specified size, starting at a specified key. The results from data scan queries against each database were recorded and presented next.

As with the other benchmark results, the findings of each database are presented before they are compared at the end of this section.

Cassandra

Although Cassandra was able to return up to 50 000 rows with normal reads, with scans the nodes ran out of memory for requests larger than 5 000 records. Scans could only be measured for queries requesting up to 5 000 records. Scans performed better than normal reads.

Table 6-10: Cassandra data scan results

Records	1	100	1 000	2 500	5 000
Duration in ms (average)	56	308.5	2 738.25	5 871	11 137.5
Latency per record in ms	56	3.085	2.73825	2.3484	2.2275
Records per second	17.8571	324.149	365.197	425.822	448.934

By increasing the number of requested records, the latency per record retrieved decreased. Requesting one record took 56 ms, while requesting 5 000 records reduced the time per record retrieved to 2.2 ms. Scans could, therefore, return on average 448.9 records per second for result sets of 5 000 records, while normal reads could only manage an average of 333 records per second for result sets of 5 000 records.

Hbase

HBase allows scans on keys. While normal reads caused communication timeouts for queries requesting more than 4000 records, scans allowed result sets up to 20000 records. By using numerical keys, scans can be used to do bulk retrieves of larger record sets. Although the scan could return larger datasets, it was slower in returning records than the bulk retrieve tests. On bulk retrieve queries up to 2500 records, the average duration per record was between 1.6ms and 2.2ms, whereas with the data scans up to 2500 records the duration per record was between 3.2 ms and 5.4 ms. Scans up to 2 500 records also recorded a higher duration on the first execution of a scan, with subsequent scans taking less time. For 2 500 records the initial scan took 14 739 ms with subsequent scans on the same amount of records on average took 3 235 ms.

Table 6-11: HBase data scan results on four nodes

Records	1	100	1 000	2 500	5 000	10 000	20 000
Duration in ms (average)	—	540.1	3 537.1	8 149.3	20 359.9	49 878.7	121 862
Latency per record in ms	—	5.401	3.5371	3.25972	4.07198	4.98787	6.0931
Records per second	1	185.151	282.717	306.775	245.581	200.486	164.12

Although HBase could serve results for scans for requests larger than 5 000 records, scans performed worse than reads. For requests of 2 500 records, which were the best performance recorded for scans, HBase was able to serve 306.78 records per second on average. Normal reads (where the keys requested were provided) could be served at a faster average of 448.5

records per second. Increasing the request size on scans beyond 2 500 records caused the throughput to slow down, reaching throughput of only 164 records per second for requests of 20 000 records.

MongoDB

MongoDB allows for data scans on fields, and not only on keys. When defining search objects, sorting is allowed on cursors and allows queries where a value is greater than a specified value. If no sort is specified, the first record to match the query will be returned. The sorting, therefore, allows data scans on any of the fields, but for larger results sorting requires an index on the field.

When performing data scans, the first scan query on average took twice as long as subsequent queries to perform the same scan, as MongoDB performed indexing on the data retrieved. Scanning for one record on average took 12.9 ms. Increasing the scan range took between 2.3 ms and 2.5 ms when scanning between 1 000 and 10 000 records. Scanning 20 000 records showed a gradual decrease in response time.

Table 6-12: MongoDB data scan results on four nodes

Records	**1**	**100**	**1 000**	**2 500**	**5 000**	**10 000**	**20 000**	**50 000**
Duration in ms (average)	12.9	409.1	2 723.9	7 082	12 030.6	26 009.6	54 455.2	132 988
Latency per record in ms	12.9	4.091	2.7239	2.8328	2.40612	2.60096	2.72276	2.65975
Records per second	77.5194	244.439	367.121	353.008	415.607	384.473	367.274	375.975

MongoDB allowed scans of up to 50 000 records. Increasing the scan result size increased the average serving speeds with the highest throughput recorded at 5 000 records, which was served at an average of 415.6 records per second. Increasing the requested result size beyond that caused a decrease in speed.

CouchDB

For CouchDB to achieve data scans, it created a view to emit documents based on the key selected and the number of records requested. The view was created the first time the query was executed and the initial duration was 933 495 ms or 15.5 minutes. Once the view was created, response time to scan for one record was 19.3 ms on average. Scan queries on larger

datasets performed even worse than bulk retrieve queries, returning 100 records in 39.8 seconds on average. Because of the poor performance, queries of requests larger than 5000 records could not be completed.

Table 6-13: CouchDB data scan results on one node

Records	**1**	**100**	**1000**	**2500**
Duration in ms (average)	19.3	39615.5	572841	2580357
Latency per record in ms	19.3	396.155	572.841	1032.14
Records per second	51.8135	2.52426	1.74569	0.96886

CouchDB could serve data scans for requests for 1 record at an average of 50 records per second. By increasing the request size to 100 records, the average serving speed was reduced down to 2.5 records per second. At requests for 2500 records, CouchDB could only serve results at a slow 1 record per second.

Redis

Redis also allows data scans on keys. Redis allowed scans of up to 20 000 records. Retrieving one record in a scan took 14.1 ms on average. At requests for 2 500 records, the best performance was recorded, averaging at 3.2 ms per record. Increasing the requested result size beyond 2 500 records revealed a decrease in performance, with average retrieval latency per record of 3.54 ms.

Table 6-14: Redis data scan results on four nodes

Records	**1**	**100**	**1 000**	**2 500**	**5 000**	**10 000**	**20 000**
Duration in ms (average)	14.1	363.7	3 339.2	8 170.5	16 710.5	34 167.8	70 945.3
Latency per record in ms	14.1	3.637	3.3392	3.2682	3.3421	3.41678	3.54727
Records per second	70.922	274.952	299.473	305.979	299.213	292.673	281.907

At 2 500 records, Redis could serve an average of 305 records per second. Similar performances were recorded at result sets of 1 000 and 5 000 records, which managed an average of just below 300 records per second. Results for queries of 10 000 and 20 000 records showed a better performance than requests for 100 records.

Voldemort

Voldemort only allowed data scans on keys and also allowed data scan queries up to 20 000 records. Requesting a single record returned the result in 4.4 ms. Increasing the number of

records requested decreased the performance, with the best results recorded at requests of 100 records.

Table 6-15: Voldemort data scan results on four nodes

Records	1	100	1 000	2 500	5 000	10 000	20 000
Duration in ms (average)	4.4	158.1	1 778.3	4 480.9	9 432.4	27 884.6	57 886.5
Latency per record in ms	4.4	1.581	1.7783	1.79236	1.88648	2.78846	2.89433
Records per second	227.273	632.511	562.335	557.924	530.088	358.621	345.504

At requests of 100 records, a result was served at an average of 643.5 records per second. For requests for 20 000 records, Voldemort was able to serve an average of 345.5 records per second.

MySQL

MySQL achieved high response rates of queries, but it must be taken into account that the content of the pages stored was not complete data. Increasing the request sizes did not yield much change in performance, as data was returned more or less at a constant rate. The fixed record size could have an impact on this observation.

Table 6-16: MySQL data scan results on one node

Records	1	100	1 000	2500	5000	10000	20000
Duration in ms (average)	1.5	84.2	689.6	1 861.1	3 564.5	7 059	14 545.4
Latency per record in ms	1.5	0.842	0.6896	0.74444	0.7129	0.7059	0.72727
Records per second	666.667	1 187.65	1 450.12	1 343.29	1 402.72	1 416.63	1 375.01

MySQL could serve results in excess of 1 000 records per second, with throughput peaking at 1 450 records per second for requests of 1 000 records. At requests for 20 000 records, MySQL was still able to serve records at 1 375 records per second on average.

The data scan results for each database was presented individually. The results of each database will now be compared.

Comparison

MongoDB was the only database that was able to serve result sets up to 50 000 records. Cassandra could only serve data scan requests up to 5 000 records. CouchDB was the poorest

performer, which could only manage data scan requests up to 2 500 records, serving records at just below one record per second. MySQL outperformed all the databases, but unlike the other databases, it was not able to store the complete content of the input dataset.

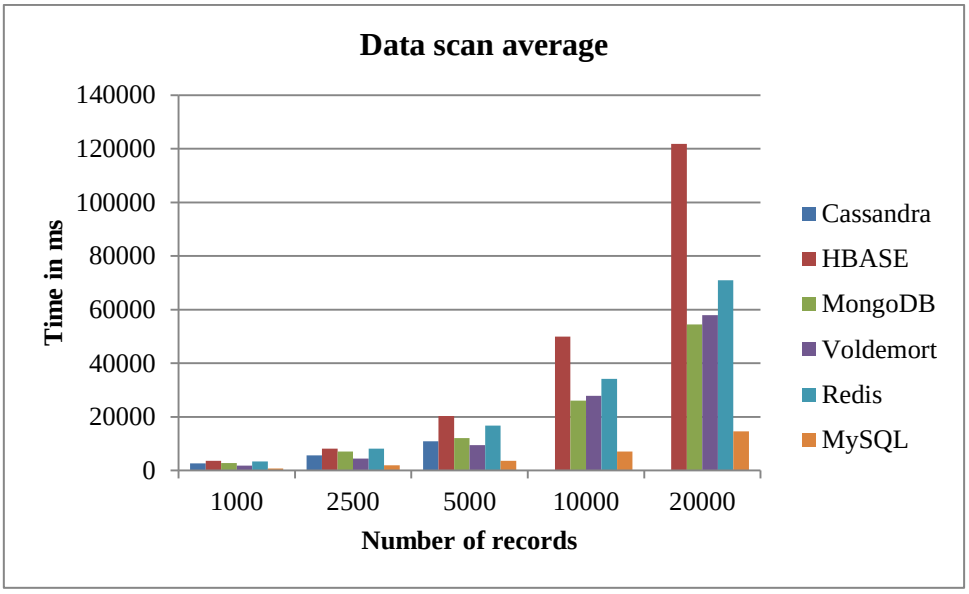


Figure 6-11: Comparison of latency in performing a data scan

HBase was the second worst performer in performing data scans. Although Voldemort allows for fast data access, MongoDB outperformed it on result sets larger than 1 000 records.

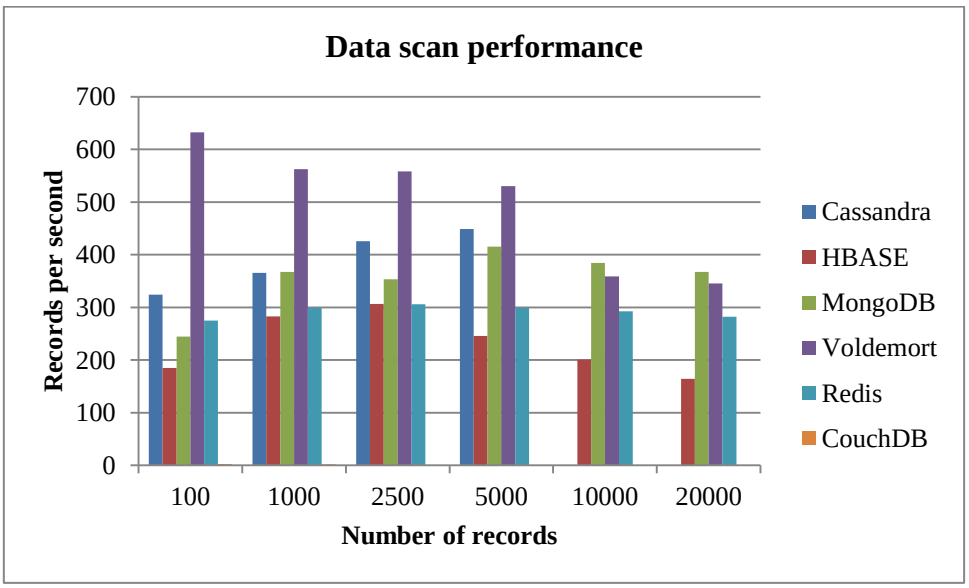


Figure 6-12: Comparison of number of records served per second

For requests up to 5 000 records, Voldemort was able to serve the most records per second. Cassandra also performed well, but was only able to serve results on scans up to 5000 records. At scans of 10 000 and 20 000 records, MongoDB performed the best. Redis showed the most constant performance through the tests, serving close to 300 records per second on all data scan tests.

The scan results of each database were presented for different query sizes and were finally compared with one another. The next section will present the results of the Grep or regular expression type queries.

6.1.4 Regular expression- (Grep) type queries

This section presents the results of Grep or regular expression type queries. The regular expression-type queries were designed to retrieve data that matches a filter. This is similar to SQL-type queries where the records that match a string are retrieved; for example, "*SELECT * FROM PAGES WHERE CONTENT LIKE '%android%'*". The five queries tested were:

- Query 1: where the base URL contained "wp-themes"; i.e. "http://www.wp-themes.com"
- Query 2: where the content or HTML contained "android"
- Query 3: where the content contained "mobile"
- Query 4: where the Page URL contained "about.html"
- Query 5: where the link information contained "itunes".

Cassandra, Voldemort and Redis do not allow regular expression filtering on the data or values, and as a result, these types of queries could not be performed for these databases.

The results of each of the databases that allowed Grep-type queries on the data fields will be presented first. Thereafter the results will be compared among the databases.

HBase

HBase supports Grep-type queries on different column families. Queries on the columns that store larger data took longer than queries on the columns with smaller data lengths. Query 1, which queried the "*baseurl*" column for the "*wp-themes*", returned eight results in an average time of 54 seconds or 6.76 seconds per result found. Queries 2 and 3 on the "*content*" column, which stores the actual HTML, returned 2 058 records containing the word

"*android*" and 45 030 records containing the word "*mobile*". Query 2 took on average 82 seconds to return the results at an average of 40 ms per matching result. Query 3 took 342 seconds on average to return the results at an average of 7.6 ms per result. Query 4, which queried the "*pageurl*" for "*about.htm*", returned one record, which took 55.9 seconds to find. Query 5, which queried the "*linkinfo*" for "*itunes*" returned 23 records in 56.6 seconds.

MongoDB

MongoDB allows regular expression searches on different fields. All the Grep-type queries, therefore, returned results. Initial response times on Query 1 was on average 64.7 seconds to return eight records, which was really slow. To speed up the performance, the database was indexed and the "*baseurl*" and "*key*" fields were used to create and index. The creation of the index improved the performance significantly and Query 1 returned the eight records on average in 57.7 ms. Query 2 took on average 66.8 seconds to return the results before the index and clustering, and afterwards it took 27.9 seconds. Query 3, which returned a larger result set, only recorded a speed-up of 20 seconds after indexing and clustering, from an average of 167.6 seconds to an average of 146.8 seconds. Query 4 returned results on average in 34.6 ms and Query 5 on average returned queries in 106.8 ms.

CouchDB

As with scan queries, Grep queries require the creation of the data view the first time it is executed, which is a costly operation. The duration of view creation for each query varied from just below 15 minutes for Query 1 to 54 minutes for Query 2.

Table 6-17: CouchDB view creation times for regular expression queries

Query	**Time**
Query 1	880 seconds (14.65 minutes)
Query 2	3246 seconds (54 minutes)
Query 3	Timed out
Query 4	2119 seconds (35.3 minutes)
Query 5	35.57 minutes

Table 6-17 presents the duration of view creation in CouchDB. The first time Query 1 executed, it took 880s (14.65 minutes) whereafter the same query returned the results in 22s on average. Query 2 took 54 minutes to generate the view whereafter the query on average

took 17 minutes to return the result set. Queries 4 and 5 took 35 minutes each to generate the view, whereafter Query 4 returned the result in 39 ms and Query 5 returned the result in 7.1 s on average. Query 3 was not able to return any results, as the view creation process timed out after one hour.

MySQL

The results for queries on the "*content*" for MySQL were discarded since the data was truncated during store. MySQL could return the results for Query 1 in 12.3 ms. Query 4 returned the results at an average of 3.56 ms and Query 5 returned the results at an average of 731.4 ms.

Comparison

A comparison between the results from the different databases will now be discussed. Table 6-18 presents the comparison of the regular expression performance per database. The number of records that matched the queries is presented in the first row. Query 1 had 8 records in the dataset that matched the regular expression searched for. Query 2 yielded 2058 results, query 3 had 45030 matches, query 4 had 8 matches and query 5 returned 23 results.

Table 6-18: Comparison of Grep query performance

	Query 1	**Query 2**	**Query 3**	**Query 4**	**Query 5**
Records returned	**8**	**2058**	**45030**	**8**	**23**
HBASE	54 059.9	82 399.2	342 856.9	55 987.2	56 552.6
MongoDB	57.7	27 985.3	146 519.3	34.6	106.8
CouchDB	2 272.667	103 2038	NA	39.88889	7 167.222
MySQL	12.33333	NA	NA	3.555556	731.4444

MongoDB recorded the best results for all the queries. Both MongoDB and HBase were able to produce results on all the queries, but HBase was performing far worse. On queries 1, 4 and 5 CouchDB performed better than HBase. On Query 5 MongoDB performed better than MySQL. The document-oriented databases performed the best on the Grep-type query tests.

Not all the databases allowed regular expression filtering on data. The results of the databases that supported Grep-type queries on data were presented first in this section before the results were compared between the different databases.

The next section presents the results of the range queries.

6.1.5 Range queries

This section discusses the results of range queries. Range queries were performed to select records that fall between a start and end value, the SQL equivalent being: "*SELECT * FROM PAGES WHERE BASEURL >='X' and BASEURL<='Y'*".

Cassandra, Voldemort and Redis do not allow this type of query on values, but only on keys. Since range queries on keys are similar to data scan queries, these tests were not performed on Cassandra, Voldemort and Redis.

On CouchDB the range queries timed out after one hour and no results could be recorded.

The range queries executed returned records where:

- Query 1: the base URL between "*http://wordpress.org*" and "*http://www.tumblr.com*"
  Query 2: the base URL was "*http://www.lifehacker.com*"
- Query*3: the base URL betwee*n "*http://jalopnik.com"* and "*http://lifehacker.com"*

The results of range queries of each of the databases that allow range queries will be discussed next.

Table 6-19: Range queries results showing average time in milliseconds and the number of records

	Query 1	**Query 2**	**Query 3**
Records returned	**15 741**	**2 189**	**3 973**
HBase	76 001.5	653 77.83	—
MongoDB	39 075	5 584.1	10 073.4
MySQL	11 611.3	2 263.6	3 503.7

HBase

Table 6-19 presents the duration of the range queries and shows the number of records returned by each query. HBase supports range queries on data by using string comparators. Range queries can also be specified using regular expression comparators. With string comparators the whole string must match; for example, the "*baseurl*" field would only match on "*http://www.wordpress.org*" and not on only "*wordpress*". Using a regular expression comparator, "*wordpress*" returned the records with the word in the "*baseurl*".

Doing range queries where the "*baseurl*" was between two strings only returned records, which matched the first string in the range and as a result, did not return any of the second strings records. Queries to return results where the "*baseurl*" was between "*http://www.tumbler.com*" and "*http://wordpress.org*" only returned results matching on "*http://www.tumbler.com*". Swapping the two values around only returned the result matching on "*http://wordpress.org/*". Range queries on string values, therefore, did not yield the expected results. Only Query 2 returned the correct results.

MongoDB

MongoDB allowed regular expression based range queries on any of the fields. Results for all queries could be retrieved. Results were retrieved at an average of 2.34 ms per record. Query 1 returned the requested records in 39 s, while Query 2 returned the results in 5.5 s and Query 3 returned the results in 9.5 s.

MySQL

MySQL was able to return the results very fast, although the result sets only contained partial information, since the content of the HTML files was truncated. Response times for Query 1 averaged at 11 s, while response times for Query 2 averaged at 2.3 s and Query 3 averaged at 3.5 s.

Of all the databases, MongoDB was the only database to return reliable results.

6.1.6 Data sorting

Data sorting for all the databases, with the exception of MongoDB and MySQL, is a design decision. Of the non-relational databases, only MongoDB allows data to be sorted when queried. No sorting results have been recorded.

6.1.7 Data removal

Data removal is supported by all the databases. Single record removal latency was low for all the databases and not considered for this research, as it does not have an impact on data warehousing and scalability capabilities.

6.1.8 Data aggregation

MongoDB supports data aggregation queries such as *count*. None of the other data store implementations provide data aggregation as part of the implementation. Cassandra, HBase and CouchDB can be used to achieve data aggregation through the use of Map/Reduce functions on the data. Voldemort and Redis do not support Map/Reduce. Since data aggregation functions are not implemented by all the database servers and external processing is required through Map/Reduce functions, no aggregation tests have been performed.

6.1.9 Conclusions on the benchmarking results

This section provides a summary of the results of data management and data warehousing experiments.

The input dataset had the best fit for the data model used by document-oriented databases. MongoDB had all the required features for data warehousing implemented and included built-in data aggregation functions as well as sorting. For data insertion it performed reasonably well, only being outperformed by Cassandra. MongoDB was also able to perform data retrieval of record sets up to 50 000 records through all tests.

For data scans, MongoDB performed the best on queries for larger result sets. MongoDB was also able to produce the fastest results in Grep-type queries on data and the only database to produce reliable results on range queries.

Although MySQL performed better in terms of speed, the big limitation was that it could not store and return complete data. The content of HTML pages had to be concatenated to fit into the record size limit of 65 KB. This meant that MongoDB was the only database to provide the capability of storing and managing large amounts of text-based data.

Column-oriented and Key/Value databases exist for specific needs. Column-oriented databases provided decent performance on reads and data scans, but had issues with larger

datasets. HBase was able to return results for Grep-type queries, but the response times were much slower than for document-oriented databases. The input dataset could also be modelled to fit into the column-oriented data model, although Cassandra only allowed range and Grep-type queries on keys.
Key/Value databases are poor choices for storing large text-based documents, as data is only accessible after requesting it by key. From a data management and warehousing point of view the document-oriented databases are the only choices for storing text-based documents with large amounts of text.

This section discussed the results of the experiments recorded by the benchmarking application in measuring the database management and data warehousing capabilities of the different databases.

The next section discusses the measurement of programming complexity and how the data retrieved was used for analysis.

6.2 Analysis of complexity

For the purposes of this research, the source lines of code for each performance test were used. The benchmark tool had the same set of features implemented for each database implementation through the use of a factory design pattern. Consequently, the same number of methods with the same input parameters and returned results were used. In this section the data for the software complexity is presented and discussed.

Not all the databases implemented all the features. Cassandra did not have a Grep-query implementation, since only the keys could be queried. Redis did not implement a bulk insert test, since bulk loads were only allowed through the Redis protocol. Voldemort did not implement Grep-type or range queries. Figure 6-13 compares the complexity in the number of source lines of code required to implement each feature between the different databases.

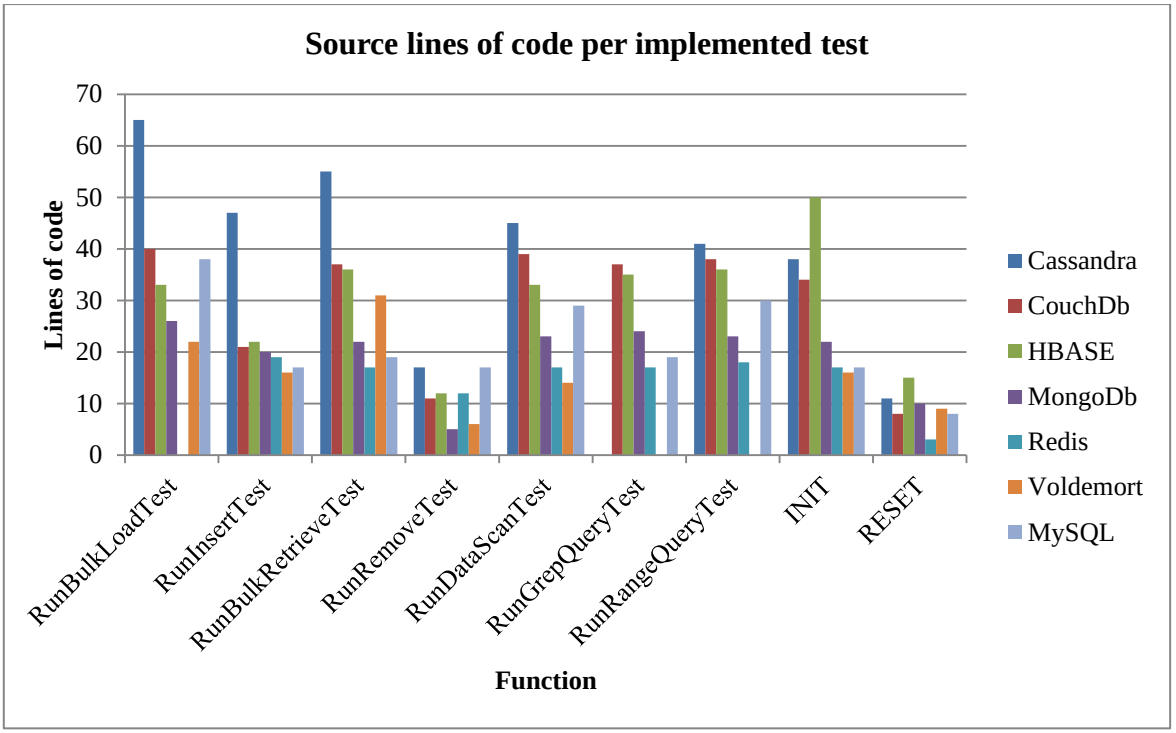


Figure 6-13: Source lines of code per implemented test

For all the functions, except the initialisation and reset functions, Cassandra required the most source lines of code. HBase required the most source lines of code for the initialisation and reset functions. Between Redis and Voldemort the least amount of code was required. Voldemort required the least amount of code to implement the bulk load and normal insert functions, and it also required the least amount of lines to initialise a connection to the database. Redis required the least amount of code to retrieve data, as well as the data scan, Grep-type and range queries.

Table 6-20: Number of source lines of code per database per function

	Bulk Load	Insert	Bulk Retrieve	Delete	Scan	Grep	Range	Init	Reset
Cassandra	65	47	55	17	45	0	41	38	11
CouchDb	40	21	37	11	39	37	38	34	8
HBASE	33	22	36	12	33	35	36	50	15
MongoDb	26	20	22	5	23	24	23	22	10
Redis	—	19	17	12	17	17	18	17	3
Voldemort	22	16	31	6	14	—	—	16	9
MySQL	38	17	19	17	29	19	30	17	8
Average	**32**	**23.1429**	**31**	**11.43**	**28.571**	**18.857**	**26.5714**	**27.714**	**9.143**

Table 6-20 shows the number of source lines of code per feature per database required. MongoDB was the only database that successfully implemented all the features and could produce the correct results. The source lines of code per function implementation in MongoDB were also below the average number of lines used for each function, which from a development complexity point of view, made it the best performer.

This section discussed findings of the programming abstraction of the implementation of the benchmark tool by measuring the complexity by using the Source Lines of Code metric.

6.3 Conclusion

The results of the experiments were presented in this chapter. The first section presented the results of experiments in terms of the dependent variables that were measured, as discussed in the methodology chapter. The results of the database management and data warehousing features that had been recorded, were presented and compared to the results of different database implementations. The programming abstraction measurements were also presented and discussed.

The next section discusses the research findings.

CHAPTER 7

RESEARCH FINDINGS

This chapter presents the results and findings of the research, based on the research design and methodology as discussed in the research methodology chapter. The results recorded during the experiments will be discussed and presented, while findings are made based on the recorded experiment results.

This chapter has the following structure:

Section 7.1 provides an introduction of the research findings and what has been measured.

Section 7.2 of this chapter will focus on the findings on data management and warehousing capabilities of non-relational databases compared to RDBMSs.

Section 7.3 discusses the scalability capabilities of the non-relational databases tested and will provide insight into how scalability has been implemented.

Section 7.4 reports on the programming abstraction portion of the research and how each database provides a programming interface.

Section 7.5 provides a summary of the research findings.

Section 7.6 provides the conclusion of this chapter.

7.1 Introduction

An experimental method was followed to test the features and limitations of different non-relational databases. Experiments consisted of a set of tests to measure the features of each database. The tests were implemented as a benchmarking application and experiments were executed by testing the different features. The goal of implementing the benchmarking tool was to determine how successful non-relational databases were in implementing features,

which are seen as limitations of RDBMSs. Another goal was to provide a mechanism for repeatability when performing the experiments.

The benchmark tests were developed to measure the effectiveness of managing and providing data warehousing capabilities of text-based data in a distributed environment. Through the implementation of the benchmarking tool, insight was gained into:

- How non-relational databases stored data
- What capabilities they had in providing for data management and data warehousing
- How they achieved scalability and how it affected performance
- What level of programming abstraction they provided.

Based on the findings of this research, it can be determined which limitations have been addressed successfully and which database implementations have been the most successful in storing text-based user generated content.

The next section provides the findings and analysis on data management and data warehousing capabilities.

7.2 Data management and data warehousing findings

The data management and data warehousing tests were designed to measure the capabilities of storing, querying and analysing text-based data stored in a distributed environment.

This section presents findings and analysis of the experiments executed to determine the storing and data warehousing capabilities of different database management systems. Each subsection presents the findings. It discusses the findings on the data management and warehousing capabilities of each type of database. The final subsection will present the conclusions on the data warehousing and data management capabilities of the different databases.

Implementation of the benchmark tests revealed that not all required features had been implemented by all data stores. In addition, limitations of the features were also discovered during implementation. This section provides the findings on these features.

All the databases implemented the basic data management features of data insertion, bulk data insertion, querying and removal. There was an exception with Redis that did not allow bulk insertion from the client, but rather through the use of command line tools and an implementation-specific protocol.

Data warehousing features were not fully implemented by all the databases. Data sorting on result sets was only provided by MongoDB as part of the default features. Sorting, for the other implementations, was mostly supported through design decisions, and depended on the selection and construction of keys. Data aggregation queries were also not supported by default, but required the use of Map-Reduce functions. They are normally implemented in frameworks outside the database. The document-oriented databases, namely MongoDB and CouchDB, provided Map-Reduce functions as part of the standard database implementation. Aggregation queries were, therefore, not tested in the experiments, as they required the use of external tools or implementation.

All of the features tested were part of the standard RDBMS features through SQL. They presented some limitations, like data sorting and aggregations, which, in most cases, were still not fully implemented or supported.

Each of the database implementations will now be considered to provide more details on the findings.

7.2.1 Column-oriented databases

Column-oriented databases tried to address the data warehousing limitations of RDBMS by providing fast access of data from different columns. Cassandra and HBase were tested as column-oriented databases and each implementation will be considered next.

Cassandra supported insertion, retrieval and removal functions of data from the client application. Cassandra used the Thrift protocol for data access and as a result, bulk insert was dependent on the transport size specified in the Cassandra configuration. By default the transport size was set to 15 MB, which required data to be uploaded in batches. The available dataset had a size of 8.96 GB for 58 000 files with an average file size of 161.99 KB. Without increasing the transport size, the bulk upload could, therefore, only be done in batches of 158.

To allow for variance in file size, the batch size was set to 120 records. Cassandra did not implement server side data sorting, Grep- (regular expression)type queries or aggregation queries. Data warehousing features should be addressed externally through the use of frameworks such as Map-Reduce. The reason for this was that data had been stored as byte arrays and returned to the client application by key. It had to be converted to the correct data type by the client application.

HBase also supported insertion, retrieval and removal from the client side. With bulk data insertions, the socket communication caused exceptions on batches larger than 1 000 records. The records were added in batches of 500 records at a time, which was more stable. HBase did not support data sorting by the database; this is a design decision, as sorting was done on keys only. Data warehousing functions like data scans, range queries and Grep- (regular expression)-type queries were supported. However, data aggregation type queries needed to be implemented through the Map-Reduce framework. Although connection time was not measured and recorded, HBase had an observable connection time much slower than any of the other databases.

Although column-oriented databases had their origin in data warehousing, they did not provide standard data aggregation functions, which needed to be implemented outside of the database itself. Column-oriented databases provided data access and presentation in a columnar manner, which was suitable for data warehousing where reports were requested on data in a columnar manner.

The next section will present the findings on document-oriented databases.

7.2.2 Document-oriented databases

Document-oriented databases recognise the structure of documents and are useful in storing text-based documents. Documents are stored, using JSON (JavaScript Object Notation), as defined in “ECMA-404 The JSON Data Interchange Standard” (ECMA International 2013). Since the input dataset used for the benchmarks are text-based documents, the document-oriented data model has the best fit for storing the data. MongoDB and CouchDB are the

implementations used for testing document-oriented databases and each will be discussed next.

MongoDB supported all of the features tested, namely insertion, retrieval, removal, data scans, Grep-type (regular expression) queries and range queries. Regular expression-type queries were allowed on the data itself and were not limited to keys only, in contrast to the other databases. MongoDB was also the only database that allowed the data sorting by the database server when queried and data returned did not need to be sorted by the client. Data aggregation was also supported through the creation of Map-Reduce functions in JavaScript. Since this required additional programming not part of the client, aggregation queries were not tested. MongoDB was also able to accept records until the client machine ran out of memory. It had no limitation on the transport size of data.

CouchDB provided support for all the features tested as well, with the exception of data sorting. The biggest limitation of CouchDB was the use of a REST API, which proved slow for large datasets. Although CouchDB supported range queries, the size of the dataset caused timeouts when trying to perform range queries; so, no results could be obtained. CouchDB supported the creation of Map-Reduce scripts in JavaScript. These sould be saved as views on the database and allowed for data aggregation queries. The CouchDB bulk insertion method required an array of documents. This limited the amount of data to be sent to the server at once, and as a result, bulk loads had to be done in batches of 1 000 records.

Document-oriented databases provided the best data model fit for the input dataset used in the experiments. MongoDB, in particular, performed well. The findings on Key/Value databases will be discussed next.

7.2.3 Key/Value databases

Key/Value databases used a simple data model similar to maps or data dictionaries. Data was stored as a value against a key. Key/Value databases addressed the need for speed and, therefore, provided fast data access. To allow the input dataset used for the experiments, the data was serialised as JSON documents similar to what document-oriented databases did. The difference, however, was that data could only be accessed through the keys, and range and Grep (regular expression)-type queries could only be done on the keys. Voldemort and Redis were tested as key/value stores and are discussed next.

Redis does not allow bulk loading from a client. Instead command line tools are used to import data from text files that adhere to the Redis protocol. Since bulk inserts cannot be done from a client, there are no bulk load results recorded for Redis. Data can only be accessed through keys and as a result, sorting, range queries and Grep-type queries on data are not allowed. Limited capabilities can, however, be implemented through carefully selected keys, for example: a Grep-type query to return data from a specific website could be created by making the website URL part of the key.

Voldemort provided the basic functionality of bulk insertion, bulk retrieval and data removal. Queries could only be done on keys. Range queries, Grep-type queries and scans were not allowed on the data itself, but only on keys.

The Key/Value database model did not really suit the input dataset and as a result, not all features required could be tested, since data had only been accessible through the keys. Key/Value databases were not really suited for storing text-based data, although they did provide fast data access, as promised.

The next section discusses the findings on the RDBMS implementation.

7.2.4 Relational database

MySQL is used as a RDBMS control database against which the other database models can be tested. MySQL is a much more mature database management system and provides really fast data access compared to some of the other database models. One of the biggest criticisms of RDBMS is that it requires a strong schema definition. Data structures need to be defined in advance and they are not flexible after implementation.

This limitation became apparent in that the row size of the tables was limited. Consequently, some of the documents retrieved could not be stored in the allowed space. Therefore, the content of the text documents had to be truncated to fit into the table space. The impact of this was that when Grep-type queries and range queries were executed, the result size differed from the actual numbers, as the stored data was incomplete.

The next section provides a summary of the findings.

7.2.5 Summary of data management and warehousing findings

All the databases tested supported the basic data management functions like inserting and reading data. Data aggregations were only supported by the document-oriented databases as built-in Map-Reduce functions, while column-oriented databases supported them through Map-Reduce frameworks outside the databases.

Table 7-1: Comparison of data store capabilities with relation to dependent variables

Feature	Cassandra	HBase	MongoDB	CouchDB	Voldemort	Redis
Insert	Yes	Yes	Yes	Yes	Yes	Yes
Bulk Load	Yes	Yes	Yes	Yes	Requires a plugin	Through protocol, not from client
Bulk Retrieve	Yes	Yes	Yes	Yes	Yes	Yes
Delete	Yes	Yes	Yes	Yes	Yes	Yes
Data Scan	Yes	Yes	Yes	Yes	Yes	Yes
Data Sorting	Design decision	On keys	Yes	No	No	Yes, only if sorted set used
Grep-type Query	Only on keys	Yes	Yes	Yes	No	Yes
Range Query	Only on keys	Yes	Yes	Yes	No	Yes
Data Aggregation	External Map-Reduce	External Map-Reduce	Yes, through built-in Map-Reduce	Yes, through built-in Map-Reduce	No	No

The input dataset used by the experiments consisted of text-based data and this model fitted the document-oriented databases best. MongoDB was the only database that could perform all the experiments with reliable results. CouchDB was highly inefficient and yielded very slow results compared to other databases, mostly because of the slow REST interface used.

The column-oriented databases could store the data successfully, but Cassandra only allowed queries and filtering on keys. HBase performed fairly well, but did not yield reliable results on range queries.

Key/Value stores performed well and although the data could be stored, it could only be accessed via keys, which made it undesirable for data warehousing applications.

From the data management and data warehousing perspective, MongoDB was the only viable database for storing the dataset used in this research.

In this section the overall findings related to data management and data warehousing capacities were discussed. Each database model's findings were discussed and more details were provided on implementations of each database model.

The next section discusses the findings on scalability.

7.3 Scalability findings

The scalability benchmarking tests provided an understanding of the capabilities of each database to automatically distribute data across multiple servers or nodes in a cluster configuration.

This section presents the research findings on the scalability capabilities of the different database families and each database implementation tested with the benchmark tool. The findings are presented per database family followed by a summary of this section.

7.3.1 Column-oriented databases

Column-oriented databases provide higher scalability and availability by partitioning data, and distributing it over multiple servers. Rows and columns are split over multiple servers by sharding on the primary keys and then splitting data on ranges, with each database node in the cluster storing a shard. Column groups can be used to split columns over multiple servers, implying that data can be partitioned by columns as well. Both row and column partitions can be used simultaneously on the same table (Cattell 2010b). Cassandra and HBase databases have been used for this research and are discussed next.

Cassandra achieves scalability through a cluster configuration and provides tuneable consistency when running servers in clusters. Consistency levels can be configured when writing and reading data. The consistency level determines the number of database replicas in the cluster configuration that must write data successfully before an acknowledgement is sent to the client. With a consistency level of ONE, a commit must be written to the commit log of at least one replica.
Although different levels existed, the experiments were executed with a consistency level of ONE, which did not require stringent consistency. During the benchmarking nodes could be added or decommissioned easily from the cluster through tools that formed part of the Cassandra implementation package. Queries could also be directed to any of the nodes. Adding and removing nodes for tests did not impact response times and stayed more or less constant. This could be attributed to data being stored in memory and only being flushed periodically to disk in the background.

HBase stores continuous row ranges together in a region and forms the basic unit of scalability. Ranges are automatically split at the row key level in equal parts and then moved to another region server. For HBase to run in a distributed environment, it requires a distributed file system.

The Hadoop Distributed File System (HDFS) was used during the benchmarking and had built-in replication, fault tolerance and scalability. Adding nodes did have a negative performance impact, which could be attributed to communication between nodes.

This section described the scalability findings of column oriented databases and document-oriented databases will be discussed next.

7.3.2 Document-oriented databases

Document-oriented databases provide scalability by partitioning data over multiple servers as well as replication methods for automatic data recovery, causing data to be persistent to secondary storage (Cattell 2010a). MongoDB and CouchDB are discussed next.

MongoDB distributes documents over servers through automatic sharding. Replication is used for redundancy and is not used for scalability like in other databases. Clustering requires *config* servers, which store the cluster metadata. Clustering requires at least three *config* server instances, and each copy stores a complete copy of the cluster metadata. A routing service, called *mongos*, is required to route queries from the application layer to the sharded cluster. Once the *config* servers and routing server are available, the MongoDB instances on each node are started and then added to the cluster through the routing service. By default, the data is not sharded across the nodes automatically. An index is required on which the sharding will be done and once an index is defined, the data is sharded. Adding nodes during the benchmark test has shown slight performance increases.

CouchDB requires the use of a proxy server to achieve scalability and does not provide scalability by default. CouchDB only provides data storage, and scalability must, therefore, be implemented in the client. CouchDB provides replication, but this only allows the exact copy of data to be stored on a different server. Replication also times out due to the large dataset sizes. For the purpose of the benchmark, and due to the poor performance in the other tests, this has not been implemented.

This section has described the findings on scalability or document-oriented databases. The next section describes the findings on Key/Value databases.

7.3.3 Key/Value databases

Key/Value databases are designed to be scalable and fast; therefore, scalability should allow queries to perform fast in distributed environments. The scalability findings on Voldemort and Redis will be discussed next.

Voldemort implements automatic sharding of data on nodes using consistent hashing. The number of nodes used to keep a copy of data is configurable and distributed automatically. Nodes can be added and removed from clusters and Voldemort will automatically adjust to use the new configuration (Cattell 2010b). Each node stores a complete cluster topology and store definitions. Therefore, it allows clients to connect to any node and request information directly from the node storing the information (Sumbaly *et al*. 2012). Adding additional nodes to the cluster slowed the performance.

Redis clustering automatically shards data over multiple nodes in the cluster. Redis does not use consistency hashing to shard data, but uses hash slots. This allows nodes to be added and removed easily by moving slots between nodes, and no down-time is required. A master-slave model is used to ensure that data can be served in the event of a node failing. Redis allows dynamic node adding and sharding through command line tools. Initially, the new node will not store any data until the cluster is re-sharded to add data blocks to this node.

This section has provided the findings on Key/Value databases. The next section provides a summary of the scalability findings across all the databases tested in this research.

7.3.4 Summary of scalability findings

With the exception of CouchDB, which required a proxy server like RDBMS to achieve scalability, all the databases were able to provide scalability over multiple nodes. Scalability was thus delivered on.

This section presented the scalability features of each database and highlighted how this had been achieved. The next section will discuss the programming abstraction provided by each database server.

7.4 Programming abstraction findings

Accessing data on the database was achieved programmatically and the ease of database interaction to developers needs was also considered for this research. Implementing the benchmarking application provided insight into the level of programming abstraction provided by each database. All the databases provided an interface to the Java programming language, either directly or through a third party API.

This section presents the findings from implementing the benchmarking tool. Each database family is discussed separately, while the database access implications to the developer will be presented for each database. Once all findings have been discussed, an analysis on the complexity of the interface is discussed.

7.4.1 Column-oriented databases

Column-oriented databases store data in a columnar manner, which resembles the table structure of RDBMSs. Cassandra and HBase will be discussed as column-oriented databases.

Cassandra supports access through the Thrift API, which allows Java clients access to the data. Access is provided as a set of libraries, which is included in the project. Data can be added by creating a *Column* object, consisting of a key and value pair. The column object is then added to the data store by specifying a unique identifier or *Row Key*, the *Column family* and the *Column*. The consistency level can also be specified.

Bulk inserts are supported by using a hash map, called a mutation map. This map contains all the updates to be applied. The mutation map consists of a key and a list of map structures as value. The key is the unique row identifier and the map list consists of key/value pairs with the key identifying the column family, with the value consisting of a list of mutations. A mutation is the column that needs to be changed or added and also consists of a key/value pair. The key is the column name and the value is the actual column data to change. A mutation map, therefore, contains all the updates that need to be executed in batch. Afterwards it is offloaded to the server to apply the updates.

The use of mutation maps adds additional complexity to the developer, since a data transformation is required before data can be added to the database. Figure 7–1 is a graphical presentation of the mutation map that is required.

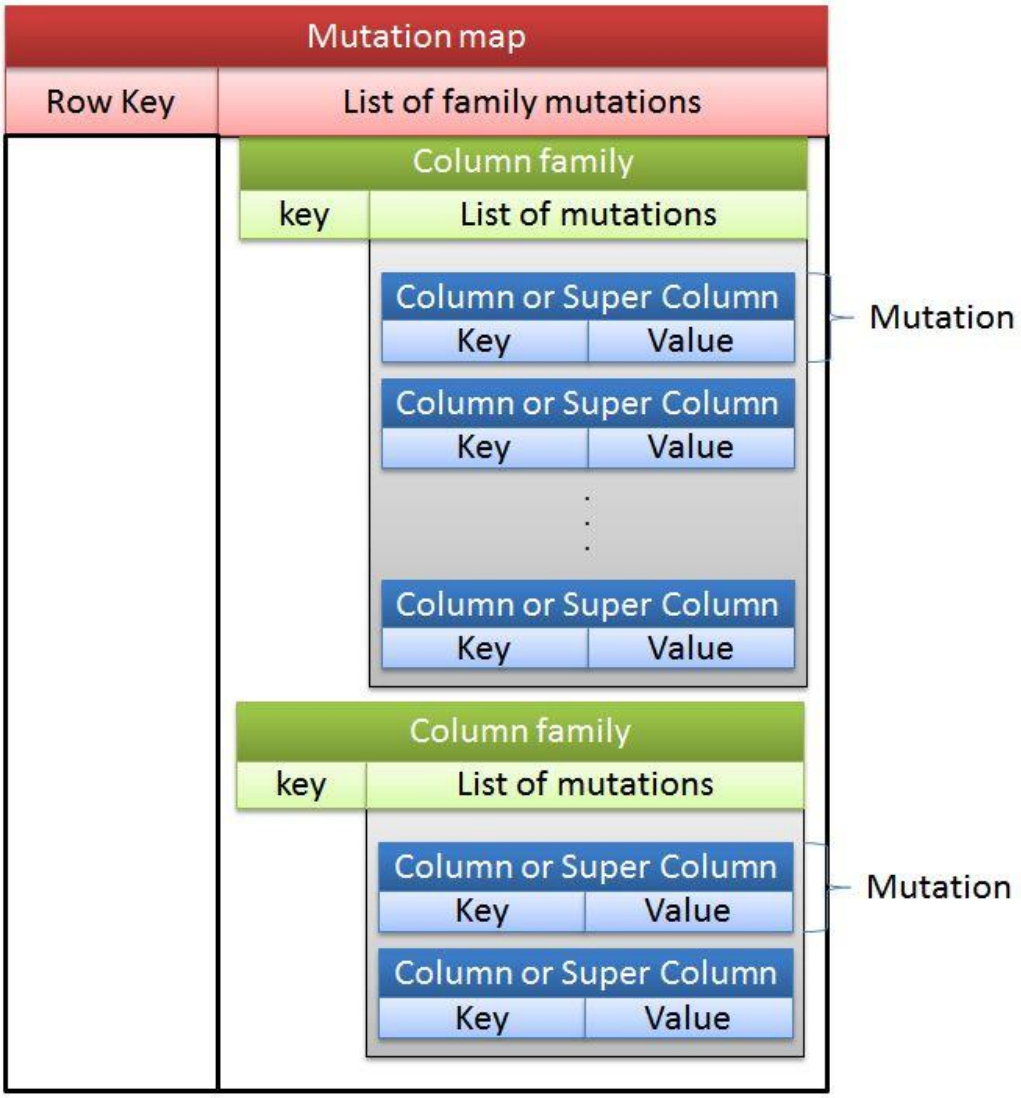


Figure 7-1: Mutation map for bulk uploads

Removing data from the database requires both the row keys as well as the column keys to be specified. This allows clients to delete only the desired column. This scenario highlights that column-oriented databases are schema-less.

Data retrieval in Cassandra is based on keys, both on row keys as well as column keys. This means that scanning, range queries and Grep-type queries are only allowed on the keys, and not on the data itself. Through the careful selection of keys, these types of queries are achievable.

To retrieve a complete row requires the specification of the columns to retrieve as well, since Cassandra allows the retrieval of single columns. Columns can be specified with the use of

slices, allowing the retrieval of multiple columns for a specific row. Data scanning is achieved through the use of a key range. Key ranges allow the start and end key to be specified. Omitting the end key will return all the results; however, a limit can be set on the number of results to return. This aspect is useful in scanning queries. To implement range queries the start and end keys can be defined to only return rows where the row key falls within a certain range.

In Cassandra, sorting is a design decision and results are not returned sorted. It means that the way in which data will be sorted must be taken into consideration when designing the row and column keys.

The programming abstraction provided by Cassandra has been discussed up to now and HBase will be discussed next.

HBase has been developed in Java and provides a native client API. The primary interface is through the *HTable* class, and it provides all the functionality to store and retrieve data (George 2011:75). To store data, a *Put* object is required, which is initiated with the row identifier. HBase stores data as byte arrays, and as a result, all data needs to be converted to bytes. The HBase API provides utilities to do this. *Put* objects require the column family, qualifier and value to be stored to be specified.

In the context of this research, this can be expressed as *"Content: baseUrl: 'http://www.digitaltoast.co.za'"*. *Put* objects can consist of a collection of data values identified by the column family, qualifier and value. A single *Put* object can, therefore, contain all the columns for a single row. The *Put* object is then updated to the data store. Batch loads are allowed through the client and there is a *batch*() method in the *HTable* class. To use this, *List of Row* objects are created and the different *Put* objects for each row are added. The *Row* object is part of the HBase API.

HBase removes the whole row through a *Delete* object, which requires the row identifier. In addition, HBase also supports the deletion of a family. Single columns can also be removed by specifying the family and the qualifier to be removed.

To retrieve data, the *Get* object is required. A row key is required for this operation. The result is returned as a *Result* object and contains the whole row. The returned results can be narrowed to return only a column family or a column, if required. Bulk retrieves of data can be done by providing a list of *Get* objects to the table's *get*() method. Note that this entails the client understanding the row identifiers to know what to retrieve. The results in bulk retrieve are returned as a *Result* array, and each value can be retrieved through the family and qualifier specified.

In the context of this research, the bulk retrieve queries required the list of row identifiers to be returned; for example, to retrieve the top 1 000 records, their row identifiers had to be added to a list. HBase provided scanning of data through a *Scan* object, which allowed the start and end rows to be specified. HBase also provided filters on *Scan* objects, which allowed for powerful data retrieval features.

In order to implement Grep-type queries, a single column value filter was used to match against a specific column value. HBase had a regular expression comparator, which was used to match single columns for certain strings. *Scan* objects also had a list of filters assigned to it, of which either all or at least one had to match. Therefore, using multiple filters on single column values allowed for range queries.

This section has presented the research findings on the programming abstraction provided by Cassandra and HBase as test databases for column-oriented databases. The next section describes the research findings on the programming abstraction provided by document-oriented databases.

7.4.2 Document-oriented databases

Document-oriented databases provide native support for documents through JSON and are designed to store user-generated content with a document structure. This section describes the programming abstraction provided by MongoDB and CouchDB as document-oriented databases.

MongoDB has a Java driver[18] for developing client applications. Data can be added as documents by creating a *BasicDBObject* object from the MongoDB library. The data is attached to this object as key/value pairs. Each object or document can have multiple key/value pairs representing the document. The object is then handed to the server through an insert command to store the data in the *collection*. A *collection* holds a collection of documents and is similar to a table in relational databases. Bulk inserts are allowed by creating a list of *DBObjects* and the list of objects is then handed to the server to insert the data. The *DBObject* object serializes documents to JSON format for storing and can also de-serialise JSON strings to objects when data is retrieved.

Document queries return subsets of the *collection* as a cursor. Bulk retrieve queries can be specified through the *find*() method without specifying any parameters. This will return all the data in the *collection*, but *limit* can be specified to limit the result size if required. MongoDB allows the use of comparators, which can be used to specify scan and range queries, where the “greater than” or “less than” conditionals can be specified to query data. MongoDB also supports OR queries by using "*$in*" or "*$or*" conditionals in queries as well as negation tests through "*$not*" conditionals. Grep-type queries can be performed through regular expression support in MongoDB.

MongoDB supports the sorting of results in queries. Sorting is done by the database server before the results are handed back to the client. MongoDB also supports *Count* and *Distinct* tools for data aggregation. It also allows grouping of results for more complex data aggregation. Map-Reduce functions are also supported by allowing JavaScript Map-Reduce functions to be specified in queries (Chodorow & Dirolf 2010:86).

CouchDB provides a REST interface, which uses the HTTP protocol to retrieve data. It is dependent on a web server to handle queries. In order to access the data through Java, the third party library CouchDB4J (Breese 2012) has been used. This API is reliant on HTTP libraries and JSON libraries to manage the communication and data extraction from results.

To insert a document requires the creation of a *Document* object, which is provided by the library. A *Document* consists of key/value pairs, which are then serialised to a JSON string

[18] Available at http://docs.mongodb.org/ecosystem/drivers/java

before they are submitted to the database. Bulk insert support is provided by serialising a collection of documents to a JSON string. Documents can also be deleted through *Document* objects.

To access data in the database, CouchDB uses the concept of a *View*. Views are map functions, which are defined in JavaScript notation and executed on the server to return a result as a set of key/value pairs (Anderson *et al*. 2010:54). To query data, a view is defined to select the relevant data and the key/value pair is returned as the result. Views can be saved in the database and used later, or they can be created on an ad hoc basis. In addition, views provide the capability to define start and end keys, and they allow the limiting of result sizes. By using views it was possible to define queries to do data scans, Grep-type queries and range queries.

Data aggregation in CouchDB is provided through built-in Map-Reduce support. Map-Reduce functions can, therefore, be defined in JavaScript functions to be executed on the CouchDB server.

This section has described the programming abstraction findings, as provided by the document-oriented databases of MongoDB and CouchDB. The next section presents the findings on Key/Value databases.

7.4.3 Key/Value databases

Key/Value databases use simple data models by only storing a key and a value pair. Queries are done on the keys only and as a result, designing the keys is important when storing data. Although user-generated documents can be stored as JSON objects, the information can only be accessed once the result set has been returned from the database. The programming abstraction provided by Voldemort and Redis as Key/Value databases will be discussed next.

Voldemort is a database that does not provide any data persistence and it requires an additional data store for persistence. By default, persistence is provided by BerkeleyDB, but this can be replaced by any data store through a pluggable interface, which allows, for example, the use of MySQL as persistent data storage. This means that, although Voldemort is a Key/Value database, it can use a relational database for data persistence. Voldemort is a

memory resident database to allow fast data access and, therefore, all queries are executed in memory.

Voldemort can be used as either a stand-alone server or it can be used as an embedded server in applications. Voldemort has been developed in Java; consequently, all libraries that are required to access or to create an embedded server are available as part of the default package and no third party interface is required.

Inserting data in Voldemort is achieved through simple *Put* statements with a key and a value. Key design is important, as queries can only be done on keys and not on the value. Voldemort does not support bulk insert as standard from the client side, but requires third party applications to do this. Deletes are done on the keys, and to delete records, the keys must be specified.

Retrieving results in Voldemort requires that all the keys that have to be retrieved must be specified. This simple query model, therefore, requires the application to know exactly which keys to request. For bulk retrieves, a list of keys must be provided to the server and the result set is returned as a map. This limitation means that Voldemort does not support any range queries, Grep-type queries, data scans or data aggregations, as these need to be implemented by the client application. Strong consideration on how keys are constructed is, therefore, required beforehand.

Redis does not provide a direct Java API and a lightweight Java client named Jedis[19] was used for this research. Although Redis is categorised as a Key/Value database, it provides additional storing mechanisms, which are not true to Key/Value databases. Values can be stored as strings, but in addition, also support storing of hashes, lists and sets. In order to use the dataset meaningfully during the experiments, a combination of data structures was used to represent the data in Redis. The *String* type was used to store a JSON representation of the complete page record. *Lists* were created to store each type of data; for example, there was a list to store the content, a list to store the base URL, a list to store the link information, etc. This was required to perform range queries. *Sets* were used to store information per "*base*

[19] Available at https://github.com/xetorthio/jedis

URL" to allow set-based operations, like determining content was found at a specified website.

Most Redis clients do not support bulk data insertion; so, the preferred way to do bulk insertion it to generate a text file using the Redis protocol. This text file is then piped on the server through the Redis command line interface (Redis 2013).

Sorting, range queries and Grep-type queries are not allowed on values, but only on keys. Redis does not allow any processing on values; however, by carefully constructing keys, certain types of queries can be constructed. Grep-type queries can be implemented through the design of the keys to return all the pages from a specific website or "*base URL*". Range queries can be constructed to return pages where the keys are in a specific range. Aggregation queries can be done to determine the number of pages per "*base URL*". Although hashes provide the additional benefit of storing both fields and values, the use of hashes removes range and sorting capabilities.

This section described the findings on the programming abstraction provided by the Key/Value databases used in this research. The next section will describe the findings on the RDBMS that was used as control for the experiments.

7.4.4 RDBMS

The benchmark tool also implemented the tests against a relational database to serve as a control. For this purpose, MySQL, which is an open source implementation of RDBMS, was used and will be discussed next.

Accessing MySQL databases from Java can be done through the *java.sql* package. MySQL has a limit in row size of 65,535 bytes (MySQL 5.5 Reference Manual). This limitation, therefore, prevented the store of documents larger than 64 KB; these documents had to be truncated to fit into a single row. All the database functions required were supported through the use of prepared statements from Java. Bulk load was achieved through the creation of batch structures on the prepared statement and could then be executed in batch. Batch sizes were set to 1 000 records at a time. Results were retrieved by executing queries from the benchmark application.

The findings on implementing the benchmark application for each database have been discussed in this section and next a summary of the findings on programming abstraction is presented.

7.4.5 Summary of programming abstraction findings

For the purpose of this research, programming abstraction measured the ease of programmatically interfacing with the databases. Since a factory design pattern was used, the same functions were implemented for all the databases and the Source Lines of Code metric was used.

Not all databases implemented all the features. On average, Cassandra required the most source lines of code to access the data. MongoDB was the only database to implement all features, and the results of the LOC metric were below the average for all the functions.

None of the databases relied on a query processor or query language. They allowed direct manipulation of data, which made it possible for data presentation between client application and data storage to be similar. This meant that data did not have to go through transformations or mappings between objects and the storage model, as had been required with RDBMS.

7.5 Summary of findings

The limitations of RDBMS in storing unstructured user generated content in a distributed environment, and hence the reason of this research, was highlighted during the execution of this research. The RDBMS used as a control could not store full data due to the limitation of row sizes. Scalability could only be achieved through the use of a proxy server, which highlighted the limitations.

The objectives of this research were firstly to determine the suitability of current non-relational databases in storing and managing unstructured content and secondly to determine the shortcomings these database implementations have. From the research it was learned that, in terms of scalability, non-relational databases tested in this research did deliver on their promise by providing the distribution of data over multiple servers by design. All databases could also store the user-generated text-based documents successfully, although some

limitations existed when data warehousing functions were considered, with the exception of the document-oriented database MongoDB, which could achieve results successfully on all tests. With the exception of document-oriented databases, none of the databases had built-in features for data aggregation functions and required external implementation of Map-Reduce functions.

The third objective was to determine the level of programming language abstraction. All database implementations provide direct access to the data and allowed data manipulation without the need for a query processor.

The final objective was to develop a data representations model and test framework. This was achieved by presenting data retrieved from the web as described in section 5.4.1. The test framework consisted of tests that measures the performance of task required for data manipulation as described in section 5.4.2. This results of this research was recorded by a benchmarking application which implemented this test framework.

Although the different database systems did address some of the limitations in RDBMSs, only the document oriented database MongoDB satisfied all the requirements for successfully storing user generated text based documents in a distributed environment.

7.6 Conclusion

This chapter presented the research findings. It started by providing an overview of the features measured for this research and giving an overview of what had been done to generate the research results.

The findings on the data management and data warehousing capabilities were then presented for each database implementation. The scalability findings and methods employed by different databases were presented next. Finally, the programming abstraction findings were discussed for each of the databases tested in this research before a summary was given on the findings.

The next chapter provides a conclusion of the research.

CHAPTER 8

CONCLUSION

This chapter presents the conclusion to this research. The research is an evaluation of non-relational databases to determine if they are delivering on their promise to overcome the limitations of non-relational databases when storing unstructured user-generated text-based content in distributed environments.

Chapter 1 provided an introduction to the problem this research set out to answer. Chapter 2 discussed the relational database model and the limitations that existed. Chapter 3 discussed the modern data storage requirements, which had presented problems for the relational database model. Chapter 4 introduced the alternative database models and management systems that existed, which aimed to solve the limitations of relational database management systems, as introduced in chapter 3. Chapter 5 presented the methodology followed by this research and described the experiments that had been performed for this research. Chapter 6 presented and discussed the results recorded for the research, while chapter 7 presented the research findings.

This chapter presents the conclusion to this research and has the following structure:

Section 8.1 of this chapter provides a brief overview of what this research has aimed to achieve.

Section 8.2 presents a summary of the research.

Section 8.3 discusses the contribution of this research.

Section 8.4 suggests future research that may follow.

Section 8.5 provides the conclusion to this chapter.

8.1 Overview of the research

This research investigated the success of non-relational database systems in addressing the limitations of relational database management systems when storing large volumes of unstructured user-generated text-based data retrieved from various blogs and websites on the internet, in distributed environments. The research also focused on the horizontal scalability of non-relational database systems when storing data over multiple servers.

Chapter 2 has introduced the relational database model, as introduced by Codd (1970:377) on which modern RDBMSs are built. The main constraint faced by relational database systems in distributed environments is due to the ACID properties of databases, which require data to be *Atomic*, *Consistent*, *Isolated* and *Durable* (Gray 1981:1). Distributed databases require data to be *Available*, *Consistent* and tolerant to network *Partitions*, known as the CAP theorem (Brewer 2000). These design constraints on RDBMSs make the storing and management of data stored in a distributed environment difficult.

Chapter 3 has described the modern data storage requirements. When storing large volumes of data in distributed environments, RDBMSs have some limitations and horizontal scalability of storage becomes a problem. In addition to horizontal scalability, RDBMSs are also not suitable for text processing, data warehousing, stream processing, as well as scientific and intelligent databases (Stonebraker *et al*. 2007:174-176). New approaches to data management and the problems of programming interfacing have been discussed.

Alternative database models and implementations exist, and have been introduced in chapter 4. The stated aim of these database management systems and storage models is to address the limitations of non-relational databases.

To test how successful these alternative database management systems are, an empirical research method was followed. To answer the research question, additional questions were asked, namely:

- How do non-relational databases provide data management capabilities like querying, filtering and analysis?
- How do they provide scaling and distribution of data over multiple servers?

- What data mining capabilities do non-relational databases present?
- What level of programming abstraction and interfacing do they provide?

The experiments presented in chapter 5 were designed to test these capabilities of non-relational databases, which were implemented in a benchmarking application. As a control, a relational database was used to perform the experiments against. The database families tested were column-oriented databases, document-oriented databases, Key/Value databases and a relational database.

The results recorded by the experiments designed in chapter 5 were presented in chapter 6. In addition, the measurement of complexity, which was useful in understanding the level of programming abstraction, was also presented.

Chapter 7 presented the research findings in terms of data management and warehousing for each database family. The findings on database scalability and programming abstraction were discussed in detail before a summary of the findings was presented.

The summary of the research will be discussed in the next section.

8.2 Summary of the research

To test the hypothesis of this research an experimental approach was followed, as presented in the research design and methodology chapter. The designed experiments tested the capabilities of non-relational databases to store user-generated text-based content by measuring:

- Data management and warehousing capabilities
- Horizontal scalability of data storage
- The level of programming abstraction available.

Data management and warehousing capabilities were tested with the use of experiments by executing data management functions like inserts and reads, and queries that might be useful in data warehousing (which included data scans, range queries, regular expression filtering and aggregation functions). Horizontal scalability was tested by measuring the impact of

adding and removing database server nodes to the cluster. Programming abstraction was tested to determine the ease of interfacing to the databases, and the Source Lines of Code (SLOC) metric was used.

The RDBMS, which was used as control had the following limitations, highlighting the validity of the limitations:

- Horizontal scalability of data storage is not supported automatically and needs to be implemented by using a proxy that distributes the queries to different database servers.
- The row size limitation imposed by the schema prevents the full text documents to be stored and as a result, data should be truncated.

These limitations demonstrated the need for alternative solutions to store large volumes of unstructured user-generated text-based documents in distributed environments.

From the research results and research findings chapters the conclusion can be made that, although different database implementations address different limitations, the data to be stored and for what it will be used plays an important role in selecting the correct database.

The data used as input dataset for this research was typical of modern data generated on the internet by users, which included text-based documents from websites and blogs. The unstructured nature of these documents is one of the design reasons behind document-oriented databases. Consequently, document-oriented databases provided the best results and addressed the limitations of RDBMSs successfully from the point of view of this research.

The contribution of this research will be discussed in the next section.

8.3 Contribution

The previous section has presented the summary of this research and this section discusses the contribution that this research has.

Understanding the current state, the actual capabilities and the limitations of non-relational databases provide insight into when and how to use each implementation. Understanding how

these databases can be used and what the limitations are can enhance the development of each database in future.

This research provides insight into how the database model used by each of the non-relational database implementations provides data management functions on large volumes of data. The research identifies the limitation of each database model when used in data warehousing applications by investigating the capabilities as related to querying, filtering and the analysis of data, which is useful for data mining. As a result, the suitability of a database system is identified for data mining purposes. The performance results from the experiments are useful for making architectural decisions when implementing new data warehousing systems.

The mechanisms to achieve horizontal scalability of data storage in distributed environments are also presented by this research. They provide insight into when and how each of the alternative database models can be used. This information can be useful when deciding which database systems to use when designing distributed data storage systems.

The results of the programming abstraction are useful when making design decisions when implementing software systems that will rely on non-relational database systems for data storage.

This research provides an answer as to how successful non-relational databases are in addressing the limitations of RDBMSs when storing unstructured user-generated text-based data in distributed environments. The results and findings of this research identify document-oriented databases as successfully addressing the limitations of RDBMSs when storing large volumes of unstructured, user-generated text-based documents in distributed environments.

This section has described the contribution of this research in relation to the research questions that have been presented. The next section discusses further research.

8.4 Further research

The contribution of this research highlights the knowledge gained from doing this research.

Future work on data can lead to a better understanding of how to manage and extract knowledge from databases. User-generated content can be valuable in extracting knowledge, while the use of databases to store this type of data in support of knowledge extraction can be researched further.

One of the limitations of this research is the scale on which the research has been performed. Future research can be performed, using much larger datasets on much larger database clusters. This can provide insight into the effect that inter-node communication latency may have on distribution of data over multiple servers.

From a data structure point of view, the type of data used in this research was suitable for document-oriented data stores. Future research could be done on document-oriented databases to focus on the limitations that currently exist and to get performance more in line of what is available in RDBMSs.

Data warehousing capabilities of document-oriented databases can also be researched further, in particular in the field of data mining of text-based documents stored in distributed document-oriented databases.

8.5 Conclusion

This chapter provided the conclusion to this research presented here. The research investigated the successfulness of non-relational database management systems in addressing the limitations of RDBMSs when storing large volumes of unstructured, user-generated text-based content in distributed environments.

The overview of the research highlighted the discussion on the relational database model and the requirements of modern data storages. It provided a brief overview of the alternative non-relational databases and how they addressed the limitations of RDBMS. A brief overview of

the research methodology that was followed, the research results and the findings were also presented.

A summary of the research was presented to highlight the discussions of previous chapters and how the research had been conducted. In conclusion to the research presented here, the contribution of the research was presented, followed by further research that could be conducted.

From the results presented in this research, the conclusion of this research is that document-oriented databases address the limitations of RDBMSs when storing large volumes of unstructured user-generated content in distributed environments. The other non-relational database families tested, address certain limitations, but focus on different data models required for the data type used in this research.

REFERENCES

Anderson, JC, Lehnardt, J & Slater, N. 2010. *CouchDB: the definitive guide*. Sebastopol: O'Reilly Media Inc.

Apache Foundation. 2012. Couchdb Wiki. Available at: http://wiki.apache.org/couchdb/ (accessed 20/02/2013).

Apache Foundation. [Sa]. High level clients. Available at: http://wiki.apache.org/cassandra/ClientOptions (accessed 20/02/2013).

Astrahan, MM, Blasgen, MW, Chamberlin, DD, Eswaren, KP, Gray, JN, Griffiths, PP, King, WF, Lorie, RA, McJones, PR, Mehl, JW, Putzolu, GR, Traiger, IL, Wade, BW & Watson, V. 1976. System R: relational approach to database management. *ACM Transactions on Database Systems (TODS)* 1(2):97–137.

Bachman, CW. 1965. Software for random access processing. *Datamation* 11(4):36–41.

Badia, A & Lemire, D. 2011. A call to arms: revisiting database design. *SIGMOD Record* 40(3):61–69.

Beyer, K, Ercegovac, V, Krisnamurthy, R, Raghaven, S, Rao, J, Reiss, F, Shekita, EJ, Simmen, D, Tata, S, Vaithyanathan, S & Zhu, H. 2009. Towards a scalable enterprise content analytics platform. *Bulletin of the Technical Committee on Data Engineering*, 32(1): 28-35.

Bhatt, K, Tarey, V & Patel, P. 2012. Analysis of source lines of code (SLOC) metric. *International Journal of Emerging Technology and Advanced Engineering*, 2(5):150-154.

Breese, MR. 2012. CouchDB library for Java using JSON. Available at: https://github.com/mbreese/couchdb4j (accessed on 14/11/2013) .

Brewer, EA. 2000. Towards robust distributed systems. Available at: http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf (accessed on 30/10/2015).

Brewer, EA. 2005. Combining systems and databases: a search engine retrospective. *Readings in Database Systems*, 4.

Brewer, EA. 2012. CAP twelve years later: how the "rules" have changed. *Computer*, 45(2):23-29.

Bryant, RE, Katz, RH & Lazowska, ED. 2008. Big-data computing: creating revolutionary breakthroghs in commerce, science, and society. Available at: http://www.cra.org/ccc/docs/init/Big_Data.pdf (accessed 20/05/2012).

Bunch, C, Chohan, N, Krintz, C, Chohan, J, Nomura, Y, Kupferman, J, Lakhina, P & Yiming, L. 2010. An evaluation of distributed datastores using the AppScale cloud platform. *IEEE 3rd International Conference on Cloud Computing (CLOUD)*, IEEE Computer Society, Jul: 305-312.

Cattell, R. 2010a. Relational databases, object databases, key-value stores, document stores, and extensible record stores: A comparison. Available at: http://www.odbms.org/download/Cattell.Dec10.pdf (accessed 30/10/2015).

Cattell, R. 2010b. Scalable SQL and NoSQL data stores. *Newsletter ACM SIGMOD Record*, 39(4):12-27.

Chang, F, Dean, J, Ghemawat, S, Hsieh, WC, Wallach, DA, Burrows, M, Chandra, T, Fikes , A & Gruber, RE. 2008. Bigtable: A distrubuted storage system for structured data. *ACM Transactions on Computer Systems*, 26(2).

Chaudhuri, S. & Dyal, U. 1997. An overview of data warehousing and OLAP technology. *ACM SIGMOD Record*, 26(1):65-74.

Chemawat, S, Gobioff, H & Leung, S. 2003. The Google File System. *ACM SIGOPS operating systems review*, 37(5), October:29-43.

Chodorow, K & Dirolf, M. 2010. *MongoDB: The definitive guide*. Sebastopol: O'Reilly Media, Inc.

Codd, EF. 1970. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377-386.

Codd, EF. 1971. A data base sublanguage founded on relational calculus. *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description*, ACM, November:35-68.

Codd, EF. 1985. Does your DBMS run on these rules? *ComputerWorld*, 14 October.

Cohen, J, Dolan, B, Dunlap, M, Hellerstein, JM & Welton, C. 2009. MAD skills: new analysis practices for big data. *VLDB '09: Proceedings of the VLDB Endowment*, 2(2):1481-1492.

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

Cuzzocrea, A, Song, I & Davis, KC. 2011. Analytics over large-scale multidimentional data: the big data revolution! *DOLAP '11 Proceedings of the ACM 14th international workshop on Data Warehouding and OLAP*, ACM, October:101-104.

Dean, J & Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107-113.

DeWitt, DJ. 1992. The Wisconsin Benchmark: Past, Present and Future, in *Benchmark Handbook: For Database and Transaction Processing Systems*, edited by J Gray, San Fransisco: Morgan Kaufmann Publishers Inc: 269-316.

Dodig-Crnkovic, G. 2002. Scientific Methods in Computer Science. *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden*, Skövde:Suecia:126-130

Douglas, C. 2006. Introduction to JSON. Available at: http://www.json.org (accessed 29/07/2012).

ECMA International. 2013. The JSON data interchange format. Available at: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (accessed 23/08/2014).

Edlich, S. 2015. NoSQL Your ultimate guide to the non-relational universe. Available at: http://nosql-database.org (accessed 3/05/2015).

Elmasri, R & Navate, SB. 1994. *Fundamentals of Database Systems*. 2nd ed., Redwood City: The Benjamin/Cummings Publishing Company, Inc.

Fitzpatrick, B. 2004. Memcached. Available at: http://www.memcached.org (accessed 27/07/2013).

Fox, A, Gribble, SD, Chawathe, Y, Brewer, EA & Gauthier, P. 1997. Cluster-based scalable network services. *Proceddings of the sixteenth ACM symposium on Operating systems principles*, 31(5):78-91.

Gamma, E, Vlissides, J, Johnson, R & Helm, R. 1994. *Design Patterns: Elements of Reusable Object Oriented Software*. Indeanapolis: Pearson Education.

Gantz, J & Reinsel, D. 2012. The digital universe in 2020: Big data,bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future, 2007*, December:1-16.

Gantz, JF, Reinsel, D, Chute, C, Schlichting, W, McArthur, J, Minton, S, Xheneti, I, Toncheva, A & Manfrediz, A. 2007. The expanding digital universe: A forecast of worldwide information growth through 2010, IDC Whitepaper, Available at: http://www.tobb.org.tr/BilgiHizmetleri/Documents/Raporlar/Expanding_Digital_Universe_IDC_WhitePaper_022507.pdf (accessed 12/12/2015).

George, L. 2011. *HBase: The definitive Guide*. Sebastopol: O'Reilly Media.

Gottemukkala, V & Lehman, TJ. 1992. Locking and latching in a memory-resident database system. *Proceedings of the 18th VLDB Conference*, British Columbia, Canada: Very Large Data Base Endowment Inc (VLDB): 533-544.

Gray, JN. 1978. *Notes on data base operating systems*. Berlin Heidelberg: Springer.

Gray, J. 1981. The transaction concept: Virtues and limitations. *Proceedings of the Seventh International Converence on Very Large Databases*, VLDB: 81 :145-154.

Harizopoulos, S, Abadi, DJ, Madden, S & Stonebraker, M. 2008. OLTP through the looking glass, and what we found there. *SIGMOD '08 Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, New York: ACM: 981-992.

Hecht, R & Jablonski, S. 2011. NoSQL evaluation: A use case oriented survey. *2011 International Conference on Cloud and Service Computing*, IEEE, 336-341.

Hewitt, E. 2011. *Cassandra - The definitive guide*. Sebastopol: O'Reilly Media.

Jacobs, A. 2009. The pathologies of big data. *Communications of the ACM*, 52(8):36-44.

Janert, PK. 2011. *DataaAnalysis with Open Source Tools*. Sebastopol: O'Reilly Media.

Kemme, B & Alonso, G. 2010. Database replication: a tale of research across communities. *Proceedings of the VLDB Endowment*, 3(1-2):5-12.

Kent, W. 1983. A simple guide to five normal forms in relational database theory. *Communications of the ACM*, 26(2):20-125.

Kumar, R. & Kaur, G. 2011. Comparing complexity in accordance with object oriented metrics. *International Journal of Computer Applications*, 18(8):42-45.

Lai, E. 2009. No to SQL? Anti-database movement gains steam. Available at: http://www.computerworld.com/s/article/print/9135086/No_to_SQL_Anti_database_movement_gains_steam_?taxonomyName=Databases&taxonomyId=173 (accessed 14/08/2011).

Lakshman, A & Malik, P. 2010. Cassandra—A decentralized structured storage system. *Operating Systems Review*, 44(2):35.

Leavitt, N. 2000. Whatever happened to object-oriented databases? *Computer* (8):16-19.

Lerman, J. 2011. What the heck are document databases. *MSDN Magazine*, November:16-19.

LinkedIn [Sa]. Project Voldemort. Available at: http://www.project-voldemort.com/voldemort/ (accessed on 5/05/2013).

McGee, WC. 1969. Generalized file processing. *Annual Review in Automatic Programming*, 5(2):77-94.

Moussa, R. 2012. TPC-H benchmark analytics scenarios and performances on Hadoop data clouds. *Communications in Computer and Information Science*, 293:220-234.

Mouton, J. 2004. *How to suceed in your master's and doctoral studies*. Pretoria: Van Schaik Publishers.

Oates, BJ. 2010. *Researching information systems and computing*. Londen: SAGE Publications Ltd.

Olivier, MS. 2009. *Information technology research*. Pretoria: Van Schaik Publishers.

O'Neil, EJ. 2008. Object/Relational mapping 2008: hibernate and the entity data model (edm). *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, New York: ACM: 1351-1356.

Özso, MT & Valduriez, P. 1996. Distributed and parallel database systems. *ACM Computer Surveys*, 28(1).

Pavlo, A, Paulson, E, Rasin, A, Abadi, DJ, DeWitt, DJ, Madden, S & Stonebraker, M. 2009. A comparison of approaches to large-scale data analysis. *SIGMOD '09 Proceedings of the 35th SIGMOD international conference on Management of data*, New York: ACM:165-178.

PC Magazone Encyclopedia. Available at: http://www.pcmag.com/encyclopedia/term/54750/wire-protocol (accessed 05/05/2015).

Porkony, J. 2013. NoSQL databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, 9,(1):69-82.

Ralsten, A, Reilly, ED & Hemmendinger, D. 2003. *Encyclopedia of Computer Science*. 4th ed., Chichester, UK: John Wiley and Sons Ltd.

REDIS . [Sa]. Introduction to Redis. Available at: http://redis.io/topics/introduction (accessed 19/05/2013).

Russell, C. 2008. Bridging the object-relational divide. *Queue*, 6(3):18-28.

Rys, M. 2011. Scalable SQL. *Communications of the ACM*, 54(6):48-53.

Sansfilippo, S. 2012. Redis. Available at: http://redis.io/documentation (accessed 14/05/2013).

Sawyer, T. 1992. Doing your own benchmark, in *Benchmark Handbook: For Database and Transaction Processing Systems*, edited by J Gray, San Francisco: Morgan Kaufmann Publishers Inc.

Segion, K. 2011. *The Little MongoDB Book*. Available at: http://openmymind.net/mongodb.pdf (accessed 17/02/2013).

Seguin, K. 2012. *The Little Redis Book*. Available at: http://openmymind.net/redis.pdf (accessed 14/05/2013).

Seltzer, E. 2005. Beyond relational databases. *Queue*,3(3):50-58.

Shi, Y, Meng, X, Zhao, J, Hu, X, Liu, B & Wang, H. 2010. Benchmarking cloud-based data management systems. *CloudDB '10 Proceedings of the second international workshop on Cloud data management*, New York: ACM: 47-54.

Singhal, M & Kshemkalyani, A. 1992. An efficient implementation of vector clocks. *Information PRocessing Letters*, 43(1):47-52.

Stonebraker, M. 2008. One size fits all: An idea whose time has come and gone. *Communications of the ACM*, 51(12):76.

Stonebraker, M. 2010. SQL databases v. NoSQL databases. *Communications of the ACM*, 53(4):10-11.

Stonebraker, M, Bear, C, Centitemel, U, Cherniak, M, Ge, T, Hachem, N, Harizopoulus, S, Lifter, J, Rogers, J & Zdonik, S. 2007. One size fits all? - Part 2: benchmarking results. 3rd Biennial Conference on Innovative Data Systems Research (CIDR), 173-183. Avalable at: http://www.cs.uml.edu/~ge/pdf/cidr07_ge.pdf (accessed 12/12/2015).

Stonebraker, M & Cattell, R. 2011. Ten rules for scalable performance in "simple operations" datastores. *Communications for the ACM*, 54(6):72-80.

Stonebraker, M & Centitemel, U. 2005. "One size fite all": An idea whose time has come and gone. *21st International Conference on Data Engineering*, Los Alamitos, CA:IEEE:2-11.

Stonebraker, M, Madden, S, Abadi, DJ, Harizopoulos, S, Hachem, N & Hellan, P. 2007. The end of an architectural era (it's time for a complete tewrite). *VLDB '07 Proceedings of the 33rd international conference on Very large data bases*, VLDB Endowment:23-28.

Strauch, C. 2011. NoSQL Databases. Available at: https://oak.cs.ucla.edu/cs144/handouts/nosqldbs.pdf (accessed 16/02/2013).

Subramanian, M & Krishnamurthy, V. 1999. Performance challenges in object-relational DBMSs. *Bulletin of the IEEE Computer Society Technical Comittee on Data Engineering*, 22(2):27-31.

Sumbaly, R, Kreps, J, Gao, L, Feinberg, A, Soman, C & Shah, S. 2012. Serving large-scale batch computed data with Project Voldemort. In: *FAST'12 Proceedings of the 10th USENIX conference on File and Storage Technologies*, Berkeley, CA: USENIX Association:18.

Vaquero, LM, Robero-Merino, L & Buyya, R. 2011. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):42-52.

Welman, C, Kruger, F & Mitchel, B. 2005. *Research methodology*. Cape Town: Oxford University Press Southern Africa.

Wiemann, M, Pedone, F, Schiper, A., Kemme, B & Alonso, G. 2000. Understanding replication in databases and distributed systems. *International Conference on Distributed Computing Systems, Taipei*, Los Alamitos,CA: IEEE: 464-474.

Yeo, CS, Buyya, R, Pourreza, H, Eskicioglu, R, Graham, P & Sommers, F. 2006. Cluster computing: High-performance, high-availability, and high-throughput processing on a network of computers. *Handbook of Nature-Inspired and Innovative Computing*. 521-551, Springer, US.