A COMPARISON OF NOSQL DATABASE SYSTEMS: A STUDY ON MONGODB, APACHE HBASE, AND APACHE CASSANDRA

ALI HUSSEIN HAMMOOD

JUNE 2016

A COMPARISON OF NOSQL DATABASE SYSTEMS: A STUDY ON MONGODB, APACHE HBASE, AND APACHE CASSANDRA

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES OF
ÇANKAYA UNIVERSITY

BY
ALI HUSSEIN HAMMOOD

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF
COMPUTER ENGINEERING

JUNE 2016

Title of the Thesis: **A COMPARISON OF NOSQL DATABASE SYSTEMS: A STUDY ON MONGODB, APACHE HBASE, AND APACHE CASSANDRA.**

Submitted by **Ali Hussein HAMMOOD**

Approval of the Graduate School of Natural and Applied Sciences, Çankaya University.

Prof. Dr. Halil Tanyer EYYUBOĞLU
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Müslim BOZYİĞİT
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Murat SARAN
Supervisor

Examination Date: 08.06.2016

Examining Committee Members

Assoc. Prof. Dr. H. Hakan MARAŞ	(Çankaya Univ.)	
Assist. Prof. Dr. Murat SARAN	(Çankaya Univ.)	
Assist. Prof. Dr. Erol ÖZÇELİK	(Atılım Univ.)	

STATEMENT OF NON-PLAGIARISM PAGE

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name : Ali Hammood
Signature :
Date : 08.06.2016

ABSTRACT

A COMPARISON OF NOSQL DATABASE SYSTEMS: A STUDY ON MONGODB, APACHE HBASE, AND APACHE CASSANDRA

Ali Hussein Hammood
M.Sc., Department of Computer Engineering
Supervisor: Assist. Prof. Dr. Murat SARAN

June 2016, 51 pages

Due to their many useful features, database management systems have been used widely with relational data for over 20 years. However, such systems are not able to handle massive and complex data efficiently. New systems known as NoSQL database management systems have appeared to deal with massive and complex data that provide fast and high performance. In this thesis, we discussed and tested three kinds of NoSQL database system in order to reveal their capabilities and how they respond in different operations. For this purpose, we set up a novel testing environment for each workload and examine the responses for the three systems. The results of this study show the weaknesses and strengths of each database system used in the study. Due to the different architectures of each database that we tested, we have seen different responses for each with changed workload operations. In our work, we used the Yahoo Cloud Serving Benchmark (YCSB), which is a framework designed by Yahoo to test database performance. According to the results obtained, we can conclude that *MongoDB* performed very well with low throughput, but not as well with high throughput. *Cassandra* and *HBase* performed very well under heavy loads due to their optimized designs. In the read operation, *HBase* has poor performance as compared to other systems tested.

ÖZ

NOSQL VERİTABANI SİSTEMLERİNİN KARŞILAŞTIRILMASI: MONGODB, APACHE HBASE VE APACHE CASSANDRA ÜZERİNE BİR ÇALIŞMA

Ali Hussein HAMMOOD
Yüksek Lisans, Bilgisayar Mühendisliği Anabilim Dalı
Tez Yöneticisi: Yrd. Doç. Dr. Murat SARAN

Haziran 2016, 51 sayfa

Sahip oldukları birçok yararlı özellik nedeniyle, veritabanı yönetim sistemleri 20 yılı aşkın bir süredir ilişkisel veri ile yaygın olarak kullanılmaktadır. Ancak, bu tür sistemlerle büyük ve karmaşık verileri verimli olarak işlemek mümkün değildir. NoSQL veritabanı yönetim sistemleri olarak bilinen yeni sistemler hızlı ve yüksek performans sağlayarak büyük ve karmaşık veriler ile başa çıkmak için geliştirilmiştir. Bu tezde, NoSQL veritabanı sistemlerimden yaygın olarak kullanılan sistemlerin yeteneklerini ve farklı operasyonlarda nasıl tepki verdiklerini ortaya çıkarmak için detaylı testler yapılmıştır. Bu amaçla, birçok farklı iş yükü tanımlanmış ve bir test ortamı kurulmuştur. Bu çalışmanın sonuçları çalışmada kullanılan her bir veritabanı sisteminin zayıf ve güçlü yönlerini ortaya koymaktadır. Test edilen her bir veritabanı sisteminin sahip oldukları farklı mimarileri nedeniyle değiştirilen iş yükü operasyonları ile her biri için farklı tepkiler gözlemlenmiştir. Çalışmada, veritabanı performansını test etmek için Yahoo tarafından tasarlanan bir kıyaslama çerçeve uygulaması olan Yahoo Cloud (YCSB) kullanılmıştır. Elde edilen sonuçlara göre, MongoDB düşük yükler ile çok iyi performans göstermiştir. Ancak, Cassandra ve HBase optimize tasarımları sayesinde ağır yükler altında çok iyi performans göstermiştir. Okuma işleminde ise, HBase test edilen diğer sistemlere kıyasla düşük bir performansa sahiptir.

ACKNOWLEDGEMENTS

I would first like to thank my thesis advisor Dr. Murat Saran of the Computer Engineering Department at Cankaya University, without his helpful advice, valuable comments and guidance this thesis could not be completed. His door was always open for me whenever I need his help. I want to thank my family for their support. Finally, I would like to thanks my friends, teachers for every thing.

Table of Contents

LIST OF FIGURES

FIGURES

LIST OF TABLES

Tables

CHAPTER 1

INTRODUCTION

The rapid increase of data sources nowadays produces a massive data that needs to be managed. A few decades before, companies start to manage data by introducing relational data base management systems (RDBM). In 2009 a new era of data base management systems started when NoSQL database systems appeared. Many foundations claimed that their database managements systems better than the others. In this study, we have made a comparison between three NoSQL database management systems (*MongoDB*, *Hbase* and *Cassandra*) in terms of read, write, update, and load performances by using the latest release of the Yahoo Cloud Serving Benchmark (YCSB-0.5.0) application during our practical work. Before couple weeks a new version of Yahoo cloud serving benchmark released (YCSB-0.7.0). Yahoo Cloud Serving Benchmark supports the latest version of *MongoDB* (3.2), Apache *Cassandra* (2.0.4) and Apache *Hbase* (1.0.2). These versions have been used in this study. NoSQL database systems are designed in such a way to provide scalability in the horizontal direction. Most NoSQL databases have been designed to store data structures that are either simple or more similar to those of object-oriented programming languages compared to relational data structures [1]. NoSQL database systems have many advantages over relational database systems. Examples of these advantages include:

1. NoSQL implementation being easy;
2. Data replication to multiple nodes (therefore identical and fault-tolerant) and its ability to be partitioned;
3. Once the data is written, at least one node will hold the data which will then be replicated to others;
4. No schema being required, thereby making it easy to distribute; and
5. NoSQL database systems being open source.

1.1 Aim of the study

In recent years, we have seen the appearance of new types of database known as NoSQL databases. The term NoSQL was first coined in 1988 to name a relational database that did not have an SQL (Structured Query Language) interface [2]. In this thesis, we test the most widely used NoSQL database management systems, namely *MongoDB*, *Cassandra*, and *HBase* [32]. We examine the performance of each system from different perspectives by applying different workloads via the Yahoo Cloud Serving Benchmark (YCSB). The performance of each system differs due to the differences in their respective data storing mechanisms. The results of this study show the weaknesses and strengths of each database system used in the study. We set up a testing environment for each workload and examine the responses for the three systems.

1.2 Significance of the study

Nowadays, there are more than 150 different NoSQL databases with different characteristics [3]. Due to this variation, it is not easy for IT managers to select the perfect database that meets their requirements. The results of this study will help IT decision makers with regard to understanding the characteristics of NoSQL databases. In addition, the results will guide them while selecting a suitable NoSQL database. Although many studies have introduced comparisons between NoSQL databases, in this study, we have used the latest versions of NoSQL database systems supported by the latest release of the Yahoo Cloud Serving Benchmark (YCSB-0.5.0). In this study, we have highlighted the most widely used NoSQL databases: *MongoDB* (3.2), Apache *HBase* (1.0.2), Apache *Cassandra* (2.0.4). Although they carry out the same function (managing data), the data handling methodology for each differs.

1.3 Big Data

Big data can be defined as the capability of managing a huge volume of data within the right time and proper speed [4]. Big data may be defined as very huge and complex collections of datasets which cannot be managed using relational database management systems (RDBMs). New data are generated every day from different

sources including pictures, social media, videos, etc. Due to this rapid growth of data, processing these data using on-hand database management systems becomes very difficult. Three characteristics to describe big data are as follows:

- **Volume**: The quantity of data stored and generated.
- **Velocity**: How fast that data is generated and processed.
- **Variety**: The nature and various types of data.

One of the solutions that have been proposed to overcome the fast growth of data has been applying better hardware; however, this approach has not been sufficient as the hardware enhancement reached a point where the growth of data volume outpaces computer resources [5]. Now, big data comes in three forms:

1. Structured data: The term structured data refers to the fact that the format of the data and the length are known. Examples of structured data include emails, phone numbers, IDs, names, addresses, etc. There are two sources that provide structured data: data generated by human intervention such as gaming data and input data. The second source is the data generated by machines such as sensor data, web log data and financial data.

2. Unstructured data: The data that do not have specific formats or known lengths. These are found everywhere and are used widely. The sources of these unstructured data are human-generated data such as website content, mobile data and social media. The data generated by machines are the second source of unstructured data, which can be found as radar data, sonar data and satellite data.

3. Semi structured data: This kind of data combines structured and unstructured data. Dealing with this degree of data complexity is not so easy. Tall data and wide records lead to long running queries; therefore, new methods need to appear in order to overcome this challenge and manage huge data.

1.4 Cloud computing

Cloud computing is simply computing that involves a large number of computers connected through a communication network, such as the Internet [6]. Cloud computing has been pushed as a rentable IT infrastructure, providing the benefits of elasticity, low upfront cost, and low time to market [7]. According to these features, the deployment of applications has been enabled, which would not have been possible in enterprise infrastructure settings. The ease of use of cloud computing has made it the standard for distributed management systems. The need for database advancement has increased especially with the rapid growth of data in the web world. NoSQL database systems have fulfilled those needs with their high scalability and the simplicity of their programming models. Cloud computing comprises many technologies and fundamental concepts. In order to construct a model for on-demand access to a shared pool of computing resources which can be configured, many building blocks synchronize. We will mention some of these blocks briefly.

- Virtualization of Computing Resources

  Cloud computing differs from virtualization with many definitions suitable for this concept, which means the way to create not a physical but a virtual version of a resource or device. For example, we can create storage devices, networks, operating systems, etc. Cloud computing has some features we must mention, virtual resources are used by cloud computing, and cloud computing also determines allocation and usage [9]. Of the many examples of virtualization in cloud computing, we can mention Amazon as one notable example.

- Scalable File Storage

  The urgent need for Scalable File Storage has appeared with data growth to store large numbers of files. The Amazon file storage system S3 and Google File System (GFS) are good examples.

- Existing Web Technologies

  This is simply a distributed application that is a part of a cloud computing system. There are many web standards such as HTTP, HTML and DHTML. In addition to

these, there are some other web services also play an important role in evolving the basic building blocks of cloud-based applications such as JSON, REST, and SOAP [8].

1.5 Related work

Many papers, researches, blogs, comments proposed about the evaluation of NoSQL database systems to discuss several aspects such as its benefits, and find the suitable NoSQL database system that meet the requirement of a project. Several NoSQL databases compared in [33]. The author in this thesis compared mongo dB, Cassandra, HBase, and Riak from different perspectives. The author used Yahoo cloud serving benchmark (ycsb) to test the performances of these four systems using the same test environment and applying different workloads on these systems. The author conclude that each system has a different response when applying a workload due to the differences in designs. In [34] another study was proposed to compare four NoSQL database systems and their performances and these systems are Cassandra, Mongo dB, and HBase using Yahoo cloud serving benchmark and by applying to the types of each data set, large and small data set to see the different when the data set fit the memory (small data set), and the second case when the size of data set is very large, figure (1.1) shows system response for workload A (will be explained in chapter -4) the database systems.

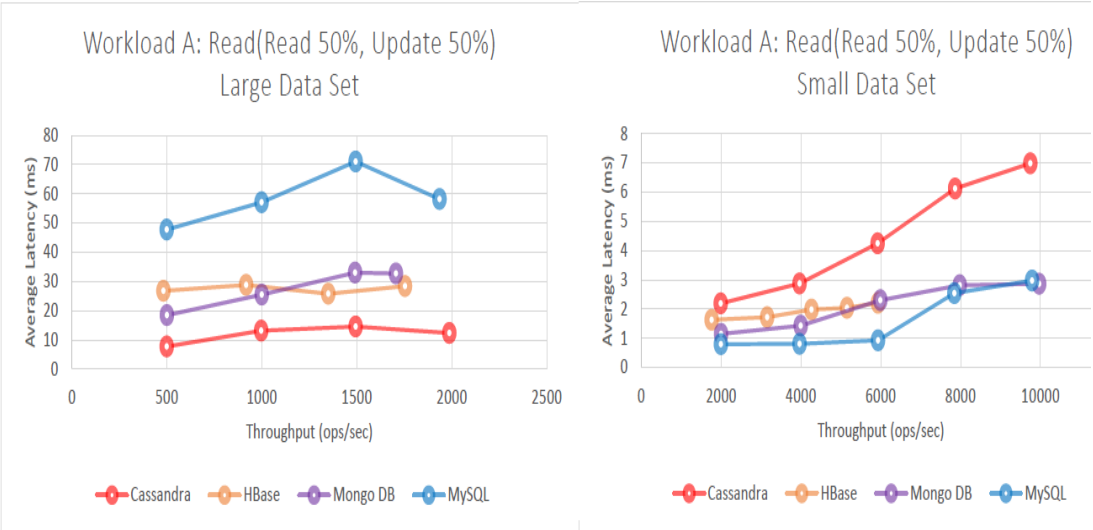

Figure (1.1) Different data set size

The author found that Cassandra performs better with large data set, the throughput is the highest with lowest average latency than the other systems. MySQL performances well with small data set. In addition, the author thought that there is no good or bad database system, it depends on your need whether this system is meet your requirement or not. HBase performed well in general but not that good in update, Cassandra has great throughput with latency cost, Mongo dB perform well with small data sets. Although HBase and Cassandra better than Mongo dB in term of throughput and input, Mongo dB has a better read and update than Cassandra and HBase. The team of end point in [35] performed a series of tests on several NoSQL database systems based on the number of nodes. they increase the number of nodes to double in each test starting with 2 nodes up to 32 nodes. they notice that the performances change with the increase of nodes number. Each system shows different response for each number of node and for each workload as we can see in the figures below [35].

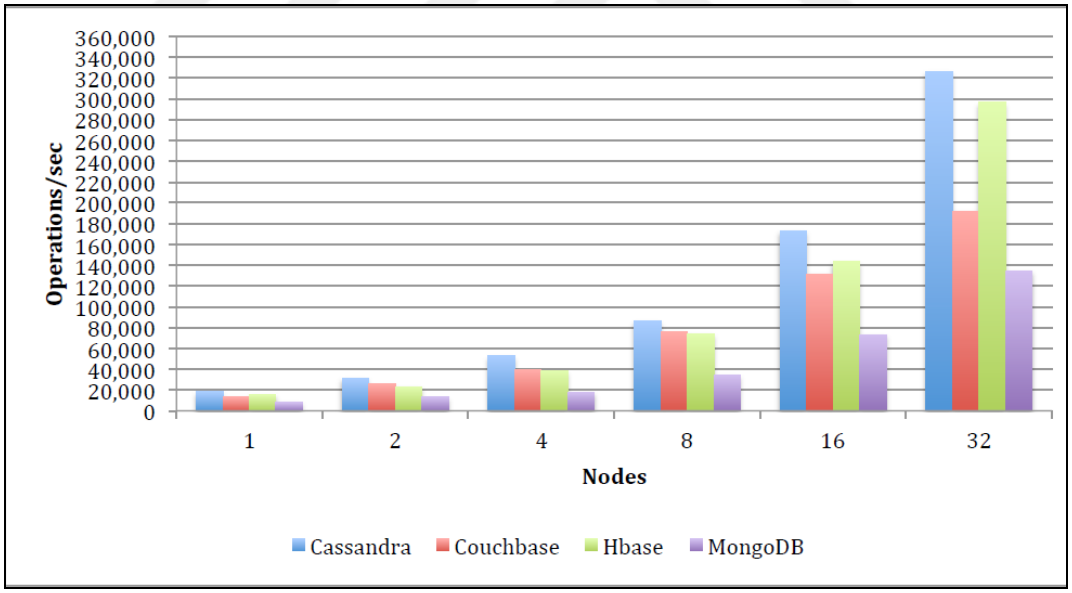

Figure (1.2) Nodes effect on performances

We can recognize that Cassandra throughput (operation /sec) increase more than the other systems when the number of nodes increase, which means the number of nodes effect on the systems performances. In [36] the authors compared the relational and non-relational data base systems of 14 different NoSQL Databases for different

perspectives such as their data models, query possibilities, concurrency control, partitioning and replication opportunities. They come up with a conclusion that NoSQL databases are better for operations that are very fast and simple for very large datasets than relational database systems. The study in [37], focused on the advantages of use of NoSQL technology by evaluating and analyzing the throughput of several database systems in terms of execution time, and show the scalability advantages of NoSQL databases.

1.6 Thesis Structure

This thesis consists five chapters. The second Chapter is talking about a background of how database systems developed recently and the techniques used to handle and manage the data. In chapter 3 we discussed in details the NoSQL data base systems that we used in our study. Chapter 4 view and discuss the results that we got while chapter 5 talks about the conclusion and the future work.

CHAPTER 2

BACKGROUND

The intention of this chapter is to explain the types of NoSQL databases and the fundamental concepts related to NoSQL databases, such as the *MapReduce* pattern, which is used in some NoSQL databases. We also focus on scaling, replication and sharding.

2.1 Types of NoSQL Data Stores

In general, NoSQL databases can be classified into three main types: *Documents* (document-oriented database), *key-value*, and *Extensible record data stores*.

1. Documents data stores: Also known as a document-oriented database, this program is used to retrieve, manage and store information. The data is semi-structured data. It is considered a subclass of key-value; however, they differ in how they process data. The name *documents data* comes from the manner of storing. Data are stored in documents as lists in JSON format. *MongoDB, Couch dB* are examples of documents data.

2. Key-value data store: This is a data store which works by matching key and value. "Its data model follows a famous memcached distributed in-memory cache "[9].

3. Extensible Record data stores: This stores tables of extensible records. It consists of rows and columns which can be shared by being divided over nodes. *HBase* is an example of this type of data store.

2.2 Scaling

Scalability can be achieved using two technologies: *sharding* and *replication*. Scalability for databases can be done in three different ways: *read operation*, *write operation* and the *volume of database*.

2.2.1 Sharding

The term *sharding* can be defined as the process of splitting data into many shards and distributing them over nodes. In other words, it is the breaking up of big data into many small databases so as to manage them more easily.

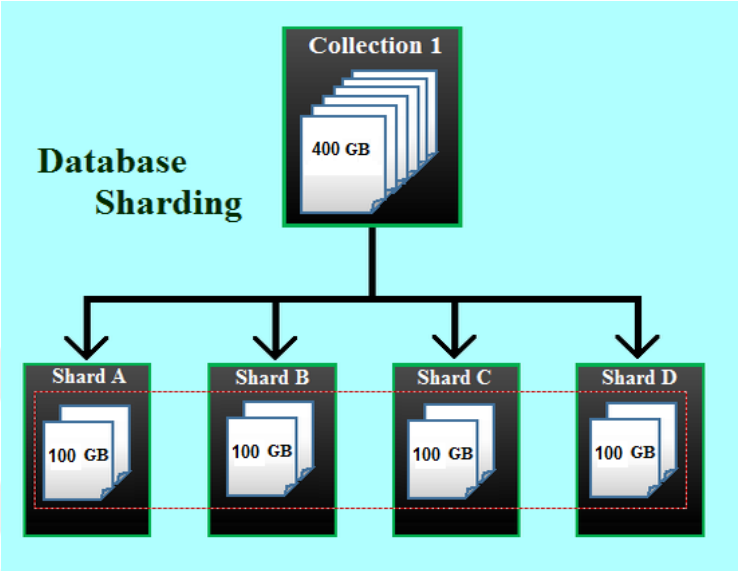

Figure (2.1) Sharding

2.2.2 Replication

Replication can be achieved by copying each piece of data across several servers. In other words, each piece of data can be found on multiple servers. There are two methods to implement replication:

- Master-slave replication: The idea of this form is to set one node as a master which handles writes, and a slave which is synchronized with the master and carries out actions such as read.

- Peer-to-peer replication: This provides the ability to write to any node and synchronize data across nodes. Master-slave replication decreases the possibility of update conflicts, while peer-to-peer replication prevents loading all writes onto a single server and creating a single point of failure. Nowadays, a system may use either technique or both techniques.

2.3 NoSQL data store foundation

NoSQL databases have three foundational principles: BASE property, Consistency Model, and CAP Theorem [10], which are discussed in this section.

2.3.1 Consistency Availability and Partition Tolerance (CAP) Theorem

Eric Brewer introduced the CAP Theorem in 2000 [11]. In 2002, the CAP Theorem was proved by Gilbert and Lynch [12]. This theorem assumes that any distributed system contains three basic properties:

- Consistency: Within a cluster, if we read or write from/to any node, the data will be the same across the cluster.
- Availability: We are able to access the cluster even if a node goes down.
- Partition Tolerance: If a partition (communications break) between two nodes, the cluster continues to function.

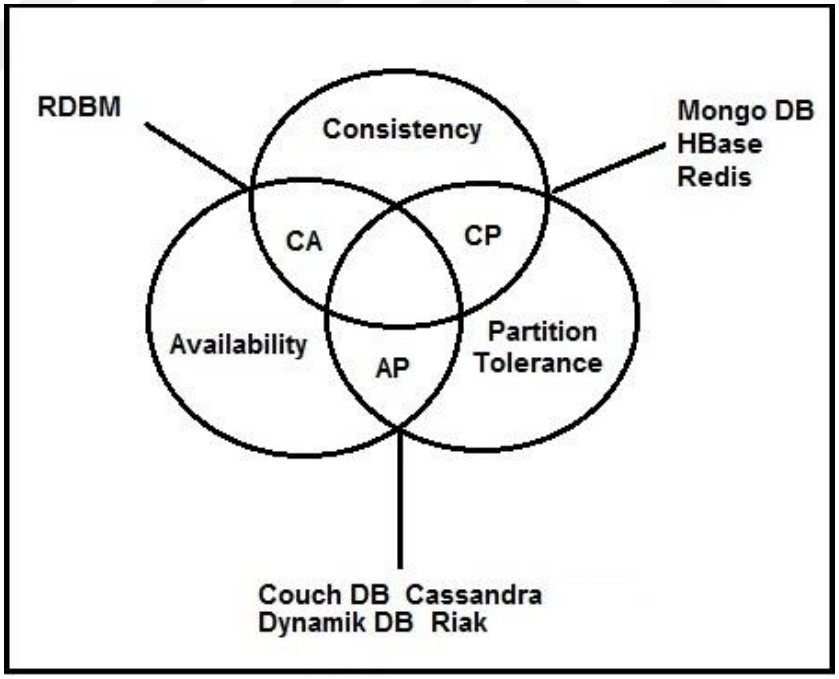

Figure (2.2) CAP Theorem [13]

For any distributed system it is impossible to satisfy all the properties simultaneously. Partition tolerance must be available in every distributed system;

otherwise, this system is not a distributed system. Therefore, any distributed system will have two properties at most, namely partition tolerance and one of availability or consistency (but not both).

Data store design comes in three varieties:

- **CA systems**: As mentioned previously, any distributed system should have a partition tolerance property; therefore, it is correct to describe it as a distributed system. Because of this consistency and availability without partition tolerance, it will not make the system distributed. Examples of this kind (CA) of system include relational database systems which have low scalability, HDFS Name Node, Vertica and Aster Data. These systems are suitable when system load is low.

- **CP systems:** Consistency and partition tolerance exist in these systems; however, with no availability. Therefore, if one of the nodes in a cluster goes down, there will be no access to the cluster. Good examples of this kind of system include *Hbase* and *Big Table*, both of which can be programmed easily. Data consistency can be achieved in a scalable way. According to many people, *MongoDB* can be considered to be a CP. A recent blog post disproved this popular thought and shows that the *MongoDB* behave does not always follow CP system [14].

- **AP systems:** Because of the availability nodes remain online even if they cannot communicate with each other and will resync data once the partition is resolved, but no guarantee that all nodes will have the same data. This type is not easy to program but provides high scalability, such as *Cassandra*, *Couch DB*, and *Simple DB*. They are good for low-latency applications.

2.3.2 BASE

BASE property stands for (Basically Available, Soft state, Eventual consistency). BASE is considered to be the first prime requirement for system reliability. 'Basically Available' means that the system cannot guarantee the availability of the data as regards the CAP Theorem. It is possible that the system status changes from

time to time, and that's could happen even if there is no input of data due to eventual consistency and that's called soft state. Finally, 'Eventual Consistency' means that if there is no given input to the system, with time the system eventually becomes consistent. Amazon and Apache *Cassandra* are examples of such systems. In these systems, the priority over consistency is given. In terms of the CAP Theorem, these systems are AP systems.

2.3.3 The Model of Consistency

Clients and servers have different aspects in terms of consistency. Werner Vogels described the three types of client-side consistency by [15]:

- Strong consistency: We can describe consistency as being strong if we perform a read operation it should returns the last written values.
- Eventual Consistency: If an update is in progress, every node will update its data but not necessarily at the same time; eventually, every node will have the same data.
- Weak consistency: Here, we cannot guarantee returning the same value during the subsequent access to the database system.

NoSQL Data stores provide some or even total consistency. *HBase*, *Big Table* is considered to have strongly consistent data stores; however, Apache *Cassandra* is considered to be consistent.

CHAPTER 3

NOSQL DATABASE MANAGEMENT SYSTEMS

This chapter introduces the NoSQL database systems that we used in this study: *MongoDB* (3.2), Apache *Cassandra* (2.0.4), Apache *HBase* (1.0.2) with detailed explanations for each system. For a good comparison, different aspects are discussed, including the architecture, which clarifies how the system is designed. The data models explain how the database stores and manages data and queries models in which the surveyed databases differ the most.

3.1 *MongoDB*

Nowadays, *MongoDB* is one of the most widely used NoSQL databases. *MongoDB* is a schema less document-oriented database developed by 10gen and an open source community [16]. It is open source, written in C++, schema less and document-oriented. *MongoDB* has a structured document query mechanism and it is used to store large files, such as images, videos, etc.

3.1.1 Architecture

The word *mongo* comes from the word "*humongous*." In terms of features, *MongoDB* considers scalable and high performance NoSQL databases. *MongoDB* is designed in a way that introduces the flexibility that satisfies the evolution of applications. It stores data in real time with high querying capabilities. *MongoDB* is used for online data and also in applicable industries. To store data, *MongoDB* uses memory-mapped files, which are files with data. The operating system places such files in memory. These kinds of files enable the operating system virtual memory manager where it stores parts of the database in memory or on the hard disk. Due to the storage mechanism, *MongoDB* loses the ability of control when data is written to the hard disk. *MongoDB* comes with a different package. *Mongod*, *Mongo* and *Mongos* represent the main processes of *MongoDB* package [17]. ***Mongod*** is the

primary daemon process for the *MongoDB* system. It is responsible for handling data requests, managing data access and furthermore it provides background data management**.** ***Mongo*** represents the client or the interactive shell. It can be used to update, query data and perform administrative tasks. ***Mongos*** is a routing service used rarely/sparingly for (*MongoDB* shard). It deals with queries and it determines the location of data within a sharded cluster. A "collection" can be defined as a group documents [18]. It resembles tables in RDBMS. Documents within the same collection may have different fields, but usually the documents within the same collection are of similar purpose.

3.1.2 Data Model

MongoDB stores data in documents as binary JSON (BSON) objects as shown in Figure (3.1). To avoid collision and to manage documents speedily, there is a unique key "_id" in a document within a collection. These keys have a number of features, such as:

- Repeated keys not being allowed in *MongoDB.*
- A null character used only to determine the end of the key, which is why it is not possible to use the null to generate keys.
- The dot (".") and dollar sign ("$") are used in definite scenarios.

Figure (3.1) *MongoDB* document as JSON

MongoDB is case and type sensitive. The example below describes this feature for two different documents.

{“Name”: “ali”}

{“name”: “ali”}

MongoDB has the ability to store different shapes of documents within the same collection.

3.1.3 Query Model

MongoDB is bundled with several types of queries. The query model of *MongoDB* allows queries over all documents inside a collection, including embedded objects and arrays [19]. Depending on the parameters used, queries return specific fields of a document or a whole document. There are a number of features of the query model to query specific results:

- **Key-value query**: The primary key is used often to query a document.
- **Comparators**: These are used to query documents in a specific range such as greater than (>), greater than or equal to (>=), less than (<), and less than or equal to (<=).
- **Conditional and logical operation**s: The *and* operation represents the logical operation that is supported for the conditional operations (*equal*, *not equal*, *exist*, *not exist*).
- **Aggregation queries**: *min*, *max*, *count*, etc.
- **Group by**: it does the same as the function Group by that exist in SQL
- **Sorting**
- **Map reduce queries**: Used to process large volumes of datasets.

3.1.4 *MongoDB* Features

MongoDB it is a high performance database system. It is a powerful and flexible document oriented database. Moreover, it is an open source database. Data is stored in documents and collections instead of tables, which allows the process of representing complex relationships more easily. *MongoDB* is a scalable database with rich secondary indexes including geospatial and TTL indexes [20]. *MongoDB* is easy to configure and can store very large data sizes without any difficulties, scheme-less. In cases of failure, it is easy to administer and overcome such problems. Version 3.2 of *MongoDB* comes with a storage engine which is memory mapped engine. There are five new capabilities available in the latest version as explained below [21]:

- **New connector for BI (Business Intelligence) and Visualization Tool**s: This enables users to visualize their *MongoDB* Enterprise data using existing relational business intelligence tools such as Tableau. The aim is to connect these tools to a datacenter and find data in tabular form. This is not an easy task when working with *MongoDB*. The purpose of creating a *MongoDB* connector for BI components is to connect the *MongoDB* server with Business Intelligent tools without storing any data. The new connector for BI works as shown in the figure below:

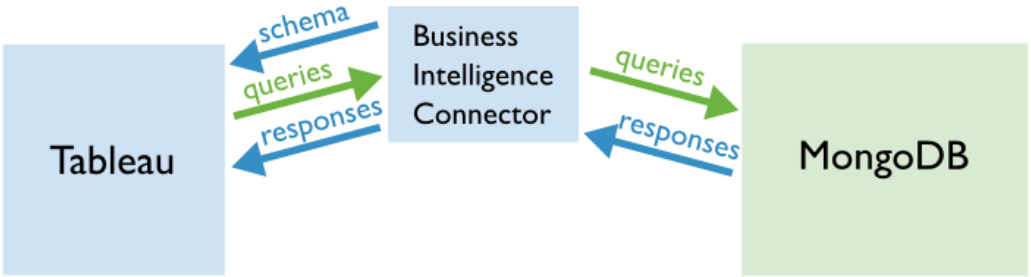

Figure (3.2) *MongoDB* connector for BI [43].

- **Encryption for data at rest**: Security is one of the great concerns for firms today. With encryption for data at rest, organizations can address their stringent security requirements.

- **Document Validation**: The latest version of *MongoDB* supports document validation, which eases the burden for companies during the process of verifying data types.

- **Dynamic Lookups**: For modeling data, lookups introduce astounding flexibility, which is part of the aggregation framework.

- **Schema Visualization**: In order to have a good understanding of data structure, *MongoDB* (3.2) supports a new graphical interface, code named *mongoScout*, which is designed to carry out several operations such as analyzing collections in order to visualize the availability of fields and the cardinality of their values [21].

3.1.5 Replication

A replica set in *MongoDB* is a group of *mongod* processes that maintain the same data set [22]. *MongoDB* supports replica set configuration, which is basically similar to Master-Slave, but not exactly identical. The different is that the replica set has an automatic failover mechanism in case the primary node becomes unavailable. If any connection problems occur between the primary and the secondary nodes, one of the secondary nodes will become the primary. Because of this mechanism, a replica set provides redundancy and availability. A replica set can be expressed as a cluster of nodes of size N. A replica set cannot have more than one primary node, which is the only node that can accept writing operations. Whenever the client start sending data to the primary replica, this data will be copied and passed to the secondary replica nodes. Figure (3.3) shows the mechanism of replication.

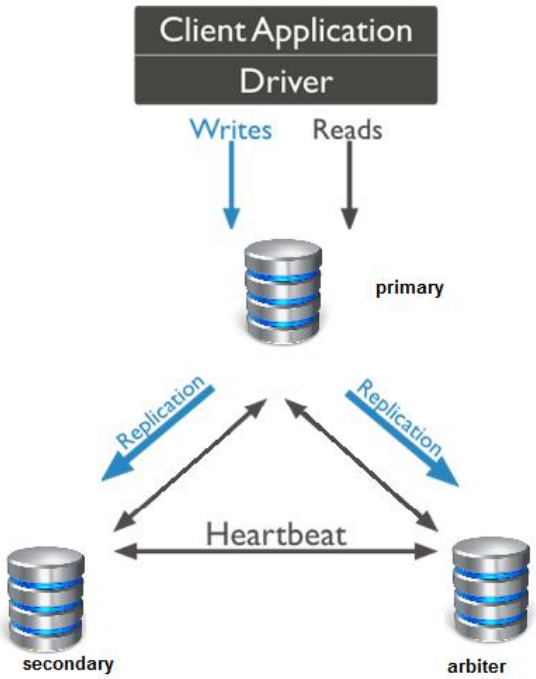

Figure (3.3) Replica set

Arbiter is a replica set member which is used to add a vote in case of choosing primary replica. Arbiter cannot hold any data and cannot be a primary replica. Heartbeat is used to check the live state of the nodes.

3.1.7 Sharding

MongoDB introduces a sharding process which can be defined as the process of storing data on several machines, which is a great advantage with the rapid increase of data. A single machine that is used to store data may not be a good choice due to the low throughput during the read/write processes. Sharding overcomes this problem by using horizontal scaling. The *MongoDB* cluster has three components: shard nodes, configuration servers and routing services (*mongos*). Each shard contains a replica set and the shards are used to store actual data. Increasing the number of nodes within each shard leads to increasing redundancy and availability. Figure (3.4) clarifies sharding architecture in *MongoDB*.

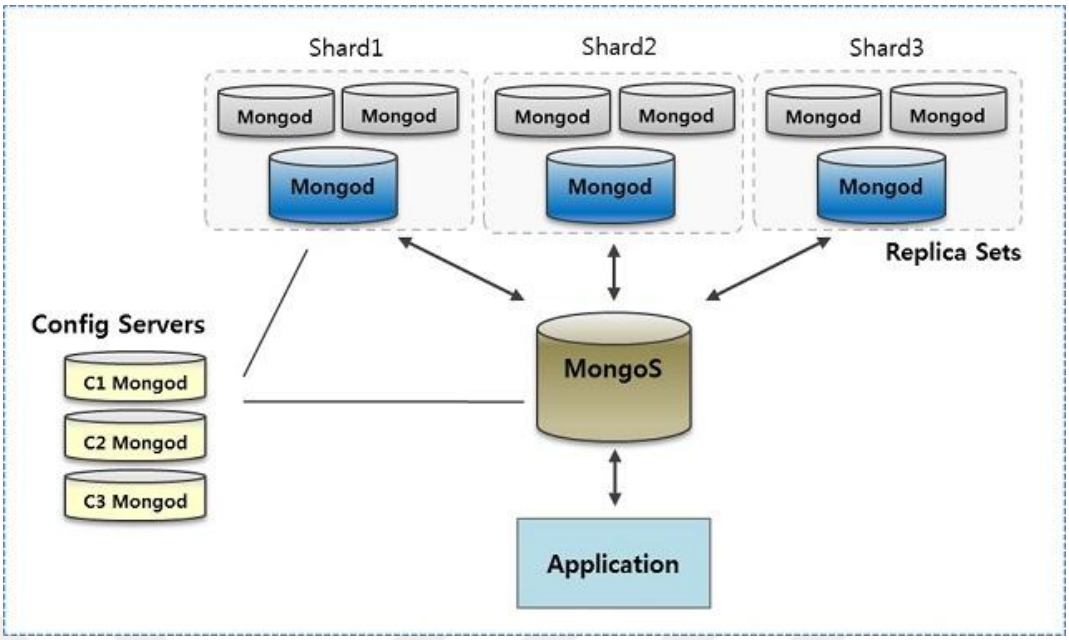

Figure (3.4) Sharding Infrastructure [44].

Configuration servers (*mongod*) are used to hold metadata which contain a mapping for the actual data in the shards. Routing servers use metadata to route operations to particular shards.

MongoDB distributes data, or shards, at the collection level. Sharding partitions a collection's data with the shard key [23]. The first step to shard a collection can be carried out with a sharding key. A shard key is similar to indexing. A shard key is divided into chunks evenly across the shard by *MongoDB*. Each chunk contains several documents in order. The main benefit of chunks is balancing the shards. If a shard size grows larger than the other shards, some contents of the chunk will be migrated to other smaller shards in order to rebalance the sizes of the shards. If a new node is added or removed from the cluster, the chunks will redistribute the data across the cluster.

3.1.7 Failure Handling

MongoDB introduces an automated failover, thereby making it more reliable. Therefore, if one *MongoDB* node crashes, the data may be lost or corrupted and even if the crashed node comes back to work correctly, the data still needs maintenance in order to find the corrupted data and repair them. A node crash may occur for

different reasons such as hardware problems, connection problems, etc. Here, the most important benefit of replication appears. If problems occur with one of the shards node and there is replica for that node, the replica will overcome the crashed node until it works again. The worst case scenario occurs when all nodes within a shard are broken. *MongoDB* then will not be able to execute any operation on the data in this shard. The same situation occurs when one of the configuration servers fails. *MongoDB* will also lose the ability to split and merge data between shards.

3.2 *Apache Cassandra*

Cassandra is one of the NoSQL database family written in Java. It is a distributed system that introduces high scalability and a fault-tolerant data store. *Cassandra* was originally developed by Facebook for the purpose of handling their inbox search feature that enables users to search through their Facebook inbox [24]. It is an open source project released in 2008. In 2010, it became a top-level Apache project. The main purpose of developing *Cassandra* was to meet storage requirements for the Index Search Problem [25].

3.2.1 Architecture

In most database systems deployed over several nodes, there exists a master-slave relationship between these nodes. The main tasks of the master are to distribute and manage data. Slaves synchronize their data to the master. If something fails with the master node, the setup of the master-slave might have a reverse influence. *Cassandra* is designed in a way to overcome these obstacles. The architecture of *Cassandra* is known as peer-to-peer, so every node in the cluster plays an identical role. There is no master in the *Cassandra* architecture which represents a point of failure. The data are split among all the nodes within the cluster. Due to peer-to-peer networking, the performance of the database improves. We can state the common features of nodes in the *Cassandra* architecture thus:

- The nodes within a cluster perform the same operations.
- Each node responds to read/write requests even if the data is not located on this particular node.

- *Cassandra* supports failover, so if any node in the cluster goes down, read/write operations can be served from other nodes in the cluster.

There are several supportive constructs for *Cassandra*. We discuss them as follows:

3.2.1.1 Key Space

Cassandra preserves an internal keyspace that is used to keep metadata about the cluster to assist in all types of operations [26]. Metadata is stored by the system of the keyspace locally for each node in addition to hinted handoff information. Metadata includes [26]:

- The node's token
- The cluster name;
- Keyspace and schema definitions;
- Information about data migration; and
- The bootstrapping knowledge related to every node.

3.2.1.2 SSTables, Memtables and Commit Logs

Cassandra's durability achieved with the assistant of commit logs, so if a writing operation is in progress, it will be immediately captured by the commit logs which are introducing a crash-recovery mechanism. A write operation will not be considered successful until it is written to the commit log. The advantage of a commit log appears when a write operation fails the in-memory store. However, it is still possible to recover that data. After the data is written to the commit log, it is written to MemTable, which is designed in such way to flush the values to disk in a file called SSTable when the number of values stored in the MemTable reaches the threshold; then a new MemTable will be created. (See Figure (3.5))

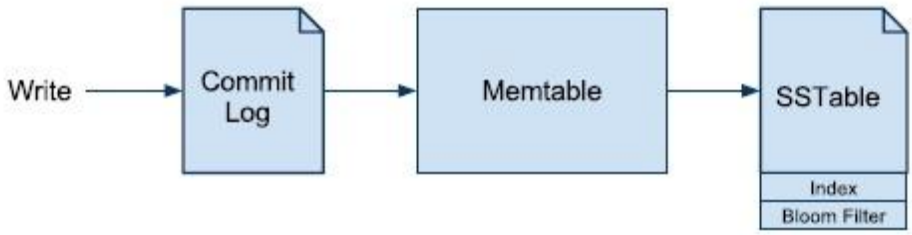

Figure (3.5) Writing in *Cassandra*

For each memTable, there is a bit flag to determine whether it needs flushing. To determine how many copies for each piece of data in the system we can simply do it by setting the replication factor to the required copies number. It is not based on the number of nodes in the cluster. The replication assists *Cassandra* to achieve high scalability and durability. Consistency in *Cassandra* comes with different levels that can be set. It is maintained by the quorum. The consistency level determines the number of replicas on which to write and which must succeed before returning an acknowledgment to the client application [27]. These levels are (0, 1, ANY, QUORUM and ALL) [26]. A quorum of replicas is a majority of replicas, or it can be represented as [(Replica Number/2) + 1]. An index file is used to reduce the number of seek instructions to find data in a specific row key, which stores no actual data.

3.2.1.3 Hinted handoff

The need to this special feature of *Cassandra* appears when one of the nodes goes down. To avoid data loss which is stored in that node, the remaining nodes in the ring will acquire that data and keep it temporarily. As soon as the node goes back to the ring, the data will be recollected and be sent back to that node. This means that we can ensure the availability of the ring.

3.2.1.4 Compaction

Compaction can be defined as the process of emptying space by merging large accumulated data files occupying that space. The purpose of performing such an operation is to merge SSTables. The keys are moved, columns are combined and tombstones are neglected [26]. If a delete operation is performed, the record will not

be deleted immediately; it will be treated as an update operation. *Cassandra* will mark this record with a tombstone instead of totally deleting this record. This procedure will reduce the size and it can be stored in the RAM instead of on disk. We can control the level of compaction by changing the value of the flag.

3.2.1.5 Bloom Filter

Bloom Filter was invented in 1970 by Burton bloom. It is a probabilistic data structure that is used to tell us whether or not data for any particular row exists in the SSTable using a fast, non-deterministic algorithm. It plays an important role especially with large volumes of data as a booster. It is considered to be a cache memory which allows quick searches.

3.2.1.6 SEDA

Staged Event-Driven Architecture (SEDA) is intended to support massive concurrency demands and simplify the construction of well-conditioned services [28]. *Cassandra* uses this model in order to divide different operations into several stages with the events and various thread pools relevant with every stage. "SEDA stages are composed of three components: *event queue*, an *event handler*, and an *associated thread pool* "[26].

3.2.2 Data model

Cassandra is a distributed system used to distribute data in multi-dimensional tables. These tables are indexed by keys and contain rows without specific sizes. The manner in which the cluster acts does not differ from a database server. Many instances are available in each database server and any instance is in charge of several of databases. The shape of the *Cassandra* cluster is a ring. Clusters are containers for keyspaces. A *Cassandra* keyspace is identical to the keyspace in a relational database, so every keyspace needs to have a name and a set of attributes which describes the behavior of the keyspace. A key space has three features:

• **Replication factor**

Replication factor means the number of nodes that will store a row of data. For example, if we set the replication factor to 3, then this copy of rows will be stored in 3 nodes. The replication factor is a very important factor to attain a high level of consistency.

• **Replica statement strategy**

Cassandra supports three different strategies which show how replicas are placed in the ring. These strategies are the *Simple Strategy* (formerly known as the *Rack Unaware Strategy*), the *Network Topology Strategy* (formerly known as the *Rack-Aware Strategy*), and the *Network Topology Strategy* (formerly known as the *Datacenter-Shard Strategy*).

• **Column family**

The column family is a container that contains rows. It is similar to a table in relational database systems. Each row contains an ordered column. The structure of the data can be represented by a column family. A key space can have one or many column families. Each

column family comprises three features: *name*, *value* and *time stamp*. Validator is the name of a data type for column values. The data type for the column name is called a comparator. As can be seen in Figure (3.6), *Cassandra* deals with vast types of data.

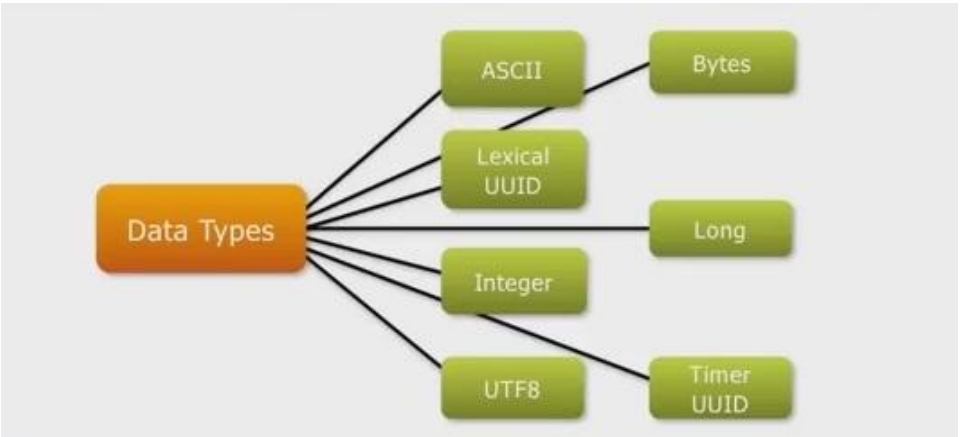

Figure **(3.6)** Types of data supported by *Cassandra* [45]

3.2.3 *Cassandra* Features

Cassandra is considered a key value store. It is open source with high availability. In comparison to other relational management database systems, *Cassandra* does not need the matching of a column within a row. Due to *Cassandra*'s decentralized architecture, any node can respond to requests, thereby enabling *Cassandra* to avoid single node failure.

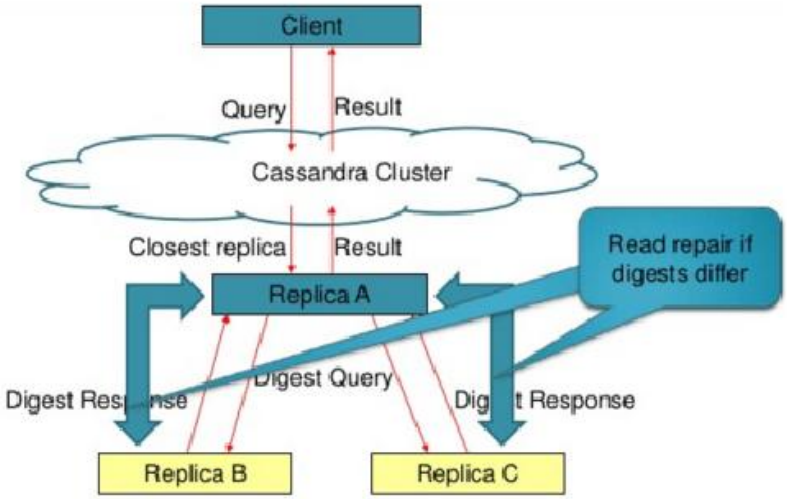

Figure (3.7) Read repair in Cassandra

The gossip protocol makes the nodes able to communicate with each other [29], gossip protocol used to send and receive information every second across the cluster. The consistency of *Cassandra* can be set in order to strike a balance between availability versus data accuracy. *Cassandra* is considered to be an AP according to the CAP Theorem. *"Cassandra* has integration point support for other software such as *Apache Hive* [30], *Apache Pig* [31], and *MapReduce*.

3.2.4 Fault Tolerance

As mentioned previously, the architecture of *Cassandra* is a peer-to-peer ring shape. Every node in the ring is identical. The data between the nodes are distributed evenly. If the replication factor is N, the data will be distributed among N nodes where N represents the replication factor. The system responsible for electing a leader of the cluster is called the zookeeper. After a leader is selected, N – 1 node

will get keys. If one of the nodes goes down, other nodes within the cluster will receive the data belonging to the failed node. When the failed node comes back to the ring again, the data belonging to it will be moved back to it from the other nodes. If the leader node goes down, then another node will be elected as leader. Reasons for failure may include hardware failure, bugs, power cuts or natural disasters.

3.3 *Apache HBase*

HBase is one of the NoSQL database systems. It is an open source distributed system which provides scalability and is fault tolerant, which is a way of storing a large volume of spares of data that are run on top of HDFS (Hadoop distributed file system). It is well for sparing data sets which are popular in many cases of big data use. Unlike relational database systems, *HBase* provides fault tolerance. *HBase* does not support a structured query language like SQL. *HBase* is written in Java. *HBase* comprises a set of tables which are stored in HDFS. Similarly, to a traditional database, each table consists of rows and columns. Each row has a row key which identifies rows uniquely. These keys do not have a specific data type.

3.3.1 *HBase* Architecture

HBase architecture is composed of three main layers (See Figure 3.8).

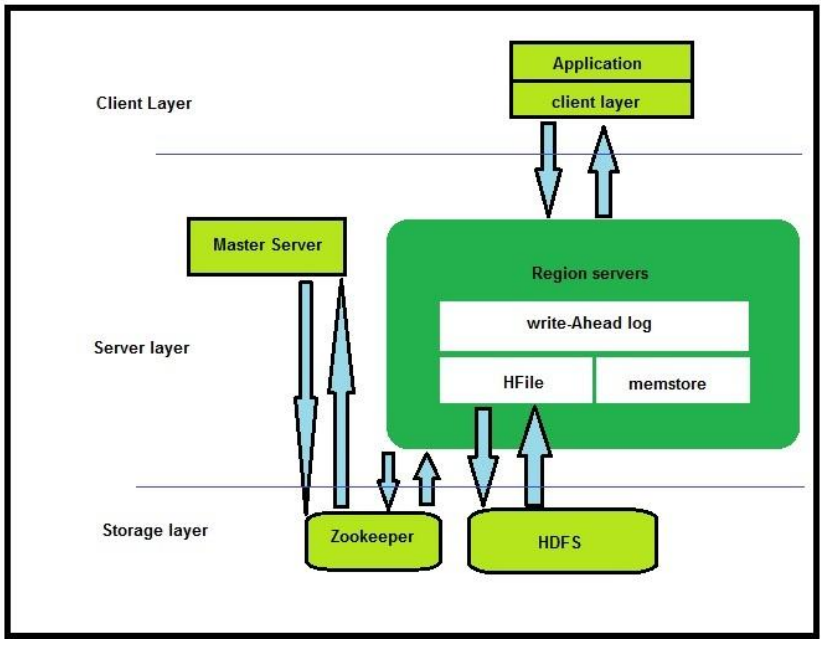

Figure (3.8) *HBase* architecture

1. **Server layer**

   This layer comprises two components:

- Master server

  In order to run *HBase* on top of HDFS, *HBase* must utilize the master's architecture which contains a master node and slave nodes. The master node is responsible for making decisions and managing one or many slaves; but it is not responsible for sorting and retrieving data, so it is lightly loaded. Master server is performed just like HMaster [38]. The master server has an external interface used to interact with region servers, clients and other utilities. In addition to the previous jobs that the master server does, it is used to manage and control data balancing across region servers. *Zookeeper* is used to communicate between the master and clients. If the master server goes down, the cluster will still be working; however, it is better to restart the master because it is also responsible for maintaining the current state of the distributed systems.

- Region server

  A region can be considered a primary unit of scalability and load balancing. When the client creates a table, each region will be assigned to each table and the created table will be used to store and retrieve data. In some cases, when the data are very big, the table grows and passes the configurable limits. In contrast to other database management systems, *HBase* supports auto-sharding, which means automatic scalability will take place for the table that passes the configurable limits. The split region is also known as the daughter region [38]. Each region is served by one region server. Figure (3.9) shows the splitting.

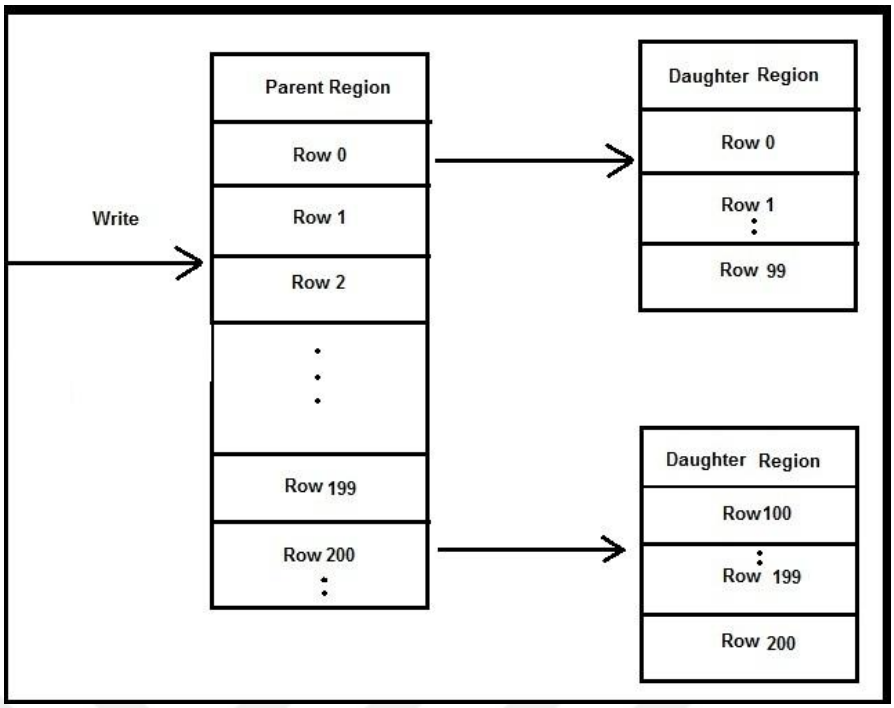

Figure (3.9) *HBase* region split

2. **Storage layer**

This layer has two component file systems and a coordination service.

- File system

The file system used by *HBase* is usually HDFS because it is designed to deal with huge datasets and provide access to data. In addition, HDFS can store and retrieve files safely due to the robust and scalable mechanism that it provides. By using HDFS, data availability will not be affected even if the storage server is offline because HDFS performs well with data replication between nodes.

- Coordination service

The foundation of Apache *HBase* has released a service for distributed applications. *Zookeeper* is an open source coordination service that provides an interface-like file system. It presents name server, synchronization, configuration management and

more. "The coordination and communication of states between the master server and the region server is handled by *Zookeeper*". *Zookeeper* guarantee that no more than one master runs at a time in *HBase* and it is used for saving the bootstrap locations for region discovery [38].

3. **Client layer**

   The client layer represents the user interface to communicate with the *HBase* installation using its client library. The client talks to the *Zookeeper* to obtain some details about the "- ROOT-" table, which gives references to ". META." tables that provide all the regions to serve rows. Whenever the client finds a matching for the ". META." region in the "- ROOT-" table, the client check the ". META." table to fetch the right user table region. If the region server goes down, then the client repeats the whole procedure.

3.3.2 *HBase* data model

HBase is designed to deal with semi-structured data which may vary in data type, field size and columns. The flexibility of the layout in *HBase* makes data partitioning easier and data distribution across the cluster more flexible. Key-values are used to store data items. Because keys and values together are formatted as byte array; the user can store different data types. For instance, *HBase* used in Facebook in order to store chatting and texting [39]. Different logical components have been used to build the data model in *HBase,* such as the following:

1. Tables

   The data in *HBase* is stored and organized in tables. These tables are comprised of rows that are stored in different parts, called regions, which are served by one region server. Each table has a name comprising a set of characters.

2. Rows

   A row comprises one or more column. Each row is identified by a key called a row-key. A row-key within a table is unique.

3. Column family

   A column family is a group of rows. When a table is created, the user should define a column family while creating the table.

Logical View of Customer Contact Information in HBase

Row Key	Column Family: {Column Qualifier:Version:Value}
00001	CustomerName: {'FN': 1383859182496:'John', 'LN': 1383859182858:'Smith', 'MN': 1383859183001:'Timothy', 'MN': 1383859182915:'T'} ContactInfo: {'EA': 1383859183030:'John.Smith@xyz.com', 'SA': 1383859183073:'1 Hadoop Lane, NY 11111'}
00002	CustomerName: {'FN': 1383859183103:'Jane', 'LN': 1383859183163:'Doe', ContactInfo: { 'SA': 1383859185577:'7 HBase Ave, CA 22222'}

Table (3.1) A column family [4]

We can see two column families in this table. The first is the customer name and the second column family is the contact info.

1. Column qualifier

   It is for indicating columns. The qualifier might be any random byte array. Unlike row keys, a column family does not have any data type.

2. Cell

   Each cell comprises three components together. These components include a row key, a column family, and a column qualifier.

3. Timestamp

   The timestamp is responsible for versioning data. A cell may have several versions. The last version will be seen first because the versions are in descending order. The default number of the cell versions is set to 3.

3.3.3 *HBase* storage mechanism

There are many ambiguous aspects in the mechanism of data storage in *HBase*, which we need to clarify." *HBase* storage design is based on Log-Structured-Merge Trees (LSM Trees) "[40]. The log file is used to store incoming data; then, in an in memory update, it is stored with the last modification of the log-files. The in memory store is very useful for fast searches. The in-memory store has a fixed size, so whenever it is full, another store file is created in the disc and all the new updates that are stored in the in memory will be flushed to the new store file. The purpose of using the HFile class is to implement the actual storage of files. These files are also called store files, which are used mainly to store *HBase* data efficiently.

3.3.4 *HBase* main operations

HBase, like any other data base management system, manages data with several operations. In this section, we briefly discuss the four main operations performed by *HBase*, namely *read*, *write*, *delete* and *search*.

3.3.4.1 Read

In *HBase*, the read operation is performed in two stages. When the client requests a specific row, the first stage of searching to obtain the query row is performed in the in-memory in order to exclude unwanted rows. If the row does not exist in the in-memory, a second stage of searching will be performed on the HFiles starting from the newest to the oldest stored data. These two stages of searching can be done using a timestamp and bloom filter in order to eliminate all undesired rows.

3.3.4.2 Write

The write operation begins when the client performs a *put* command. The data will be written to the Write Ahead Log (WAL), after which the data will be written to MemStore, which is a buffer used previously for writing with 64 MB. when the MemStore is saturated, the information will be flushed to a new HFile on HDFS continuously. Each region may contain many HFiles. The huge number of HFiles will slow down the system during the searching process. In other words, the latency of the read process will increase, which is why *HBase* combines several HFiles in order to reduce the latency through a process called compaction. The compaction comes in two versions: *major compaction* and *minor compaction*. Major compaction merges all HFiles into one single file while minor compaction deals with a small number of HFiles that are merged to one file.

3.3.4.3 Delete

When the client sends a request to delete specific data, that data actually will not be deleted directly. *Tombstone marker* is used to point a delete marker to that data. If a fetch request sends to the same data, it cannot be retrieved. A complete deletion of the marker data will occur in the next schedule of the major compaction.

3.3.5 *HBase* Replication

HBase, like many NoSQL database systems, supports a replication feature which can be considered a method of data recovery in order to avoid disasters of losing data. Replication means copying the data between *HBase* deployments. We can state several advantages of replication. For example, as mentioned earlier, replication is considered to be a way to recover the data when something goes wrong. Replication occurs to provide higher data availability. In addition, it easily copies and edits from a web-facing cluster to a MapReduce cluster that will process old and new data and ship back the results automatically [38].

3.3.6 Fault Tolerance

The main aspect of *HBase's* design is fault tolerance. *HBase* can easily deal with network connection problems or server failures to survive the message loss. One of

the jobs that master server handles, when a region server goes down, is the assignment of new server regions instead of the collapsed region server being served. There is also a master backup if the primary master dies. Since *Zookeeper* itself is also considered to be fault tolerant, it is used to declare the failed regions.

CHAPTER 4

TEST ENVIRONMENT

In this section, we discussed how we set test environment and the configuration for each database. We also highlighted the last release (now newer version is available YCSB-0.7.0) of the Yahoo Cloud Serving Benchmark (YCSB-0.5.0) with its workloads used in this study. The clusters for three NoSQL database systems are also discussed.

4.1 Cluster features

In this study, we used 5 computers with 4 GB RAM, Intel Core i3 processor with 3.33*4 GHz processor speed and 100 GB of ephemeral storage in each unit. Ubuntu 14.0.4 LTS (64-bit) was installed on each unit.

4.2 Yahoo Cloud Serving Benchmark (YCSB)

The YCSB is an open source software package used to measure performance of, and make a comparison of, several types of NoSQL database systems. The YCSB is a framework designed by yahoo, it is an open source benchmarking used to compare the performance of distributed NoSQL data stores such as *Cassandra*, *HBase*, and *PNUTS* [41]. The first benchmark was designed in 2010 to simplify the performance comparisons of cloud data stores. The YCSB performs several types of operations, such as creation, deletion, updating and reading. The YCSB comes with six standard workloads which mix different scenarios, such as read, write, update, and search. These workloads give us a good rounded picture about the performance of the database systems. There is also a second workload which can be generated by the client to highlight another aspect not covered by the core workloads. To run a workload, there are several main steps that every client should follow:

1. Install the database system

   To install a NoSQL database system, the client should realize that every system has a different way to be installed. There are some common steps between these systems, such as creating the key spaces, tables, and buckets to store the data.

2. Select suitable workloads

   Based on user requirements and to test a specific aspect of a database system, an appropriate workload must be selected.

3. Select the appropriate DB interface layer. For example, ….

4. Set the appropriate runtime parameters

   To run a workload correctly, several parameters should be set to obtain reasonable results, such as the number of threads to handle the amount of load offered against the database, setting the throughput (operations per second), and the status to control the time of running.

5. Load the data

   The user can load the data to the database by executing the command:

   ./bin/ycsb load basic -P workloads/workloada

   This command loads the core workload A.

6. Finally, run the workload

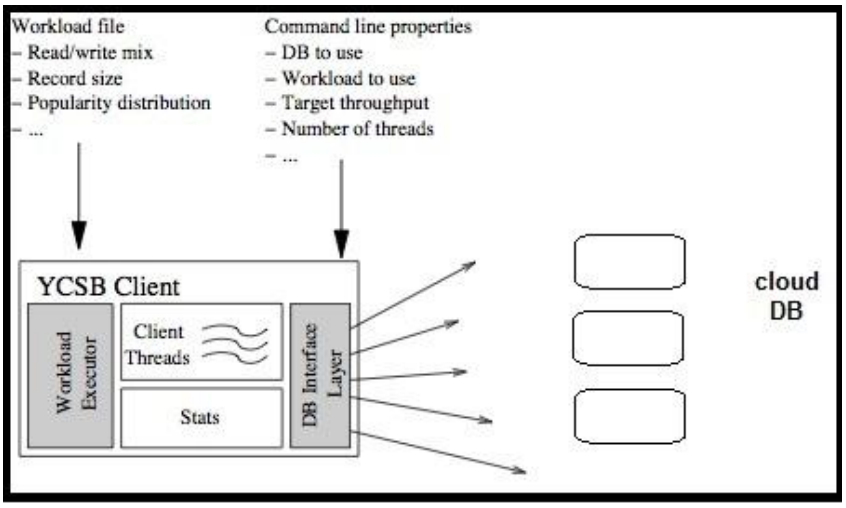

Figure (4.1) YCSB architecture

4.3 *MongoDB* Configuration

The last version of *MongoDB*, 3.2, was obtained and installed after installing the last release of Java 8. We set the settings and the configurations according the manual on the *MongoDB* website. We set up the replica set across 5 nodes and used the replica set to provide a good availability. That's could be done by replicating data in the nodes. A replica set comprises one primary server and several secondary servers. It may contain 12 secondary servers. The steps to setup the replica sets are as follows:

1. Create a directory file in each node to store the data:

   mkdir <PATH>/node1

   Repeat it for the other nodes.

2. Start mongod by:

   . /mongod --replSet mo --port 27017 --dbpath <PATH>/node1

   Repeat for the other nodes.

3. Configure the replica set:

   ```
   config = {_id: 'mo', members: [
   {_id: 0, host: 'localhost:27017'}
   ```

where mo: is the name of the replica set.

Localhost: the IP address for all nodes within the replica set.

2017: represents the port for each node.

4. Initiate the replica set:

```
rs.initiate(config)
```

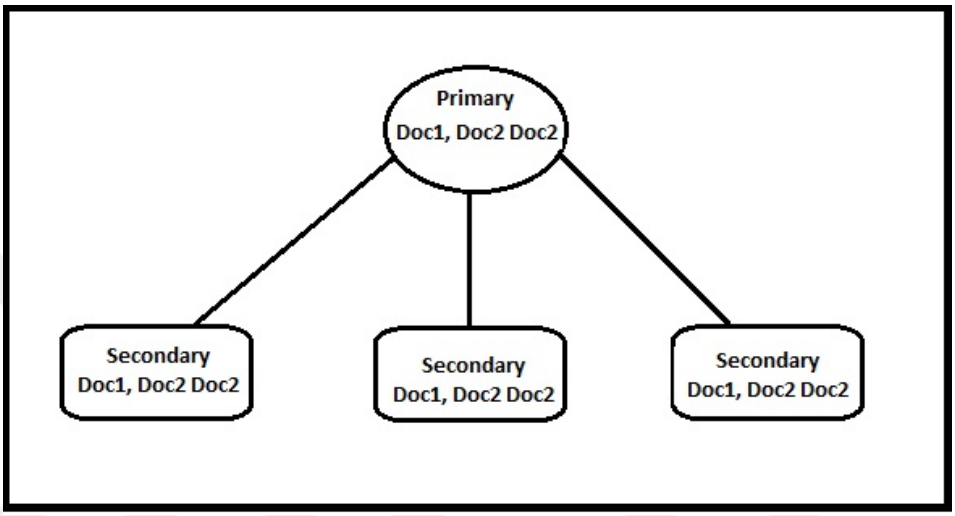

Figure (4.2) Replica set

4.4 Apache *Cassandra*

Apache *Cassandra* is an open source project designed by the Apache Foundation. Although there are several versions of *Cassandra* that have been released, the version that YCSB-0.5.0 supports is *Cassandra* 2.0.17. We set the configuration of *Cassandra* according to our requirements, by setting the following parameters in the *Cassadra.yaml* file:

1. Create three folders (data, commitlog, saved cashes) and set their locations in Cassandra.yaml

2. Set initial token = 0

3. According to the structure of *Cassandra*, we selected two seeds for each local node as its neighbors (See Figure (4.3)).

4. Set the listening address according to the local IP address of the nodes.

5. Set the rpc-address to the value: 0.0.0.0.

6. Change the simple snitch to Rack inferring snitch.

7. Remove the # from the broadcast-rpc-address.

This was repeated for the remaining nodes.

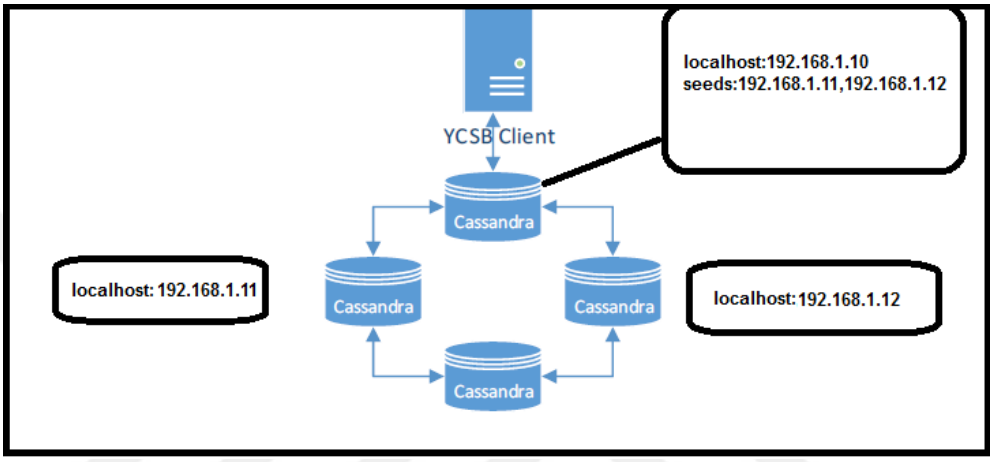

Figure (4.3) *Cassandra* configuration

4.5 Testing on Virtual Machines

Due to the rapid evaluation of *Cassandra*, many users attempt to benchmark *Cassandra*. However, they need to pay attention to several issues in order to obtain the correct results, which show the power of this system. We have tested *Cassandra* by using a powerful server with (RAM 88 GB) to see whether or not we can obtain reasonable results. We deployed five nodes, and unlike *MongoDB* and *HBase*, unfortunately, *Cassandra* results for several tests with different features for the virtual machines were unsatisfying, the throughput is very low and the average latency is very high which is against Casandra features. In real cluster as we will see in chapter 5, the throughput must be very high comparing with the throughput in virtual machines, and the average latency must be low.

4.6 *Apache HBase*

According to the ability of YCSB, we installed *HBase-1.0.2* which is the latest release of *HBase* supported by YCSB. *HBase* is open source and runs on top of *Hadoop 2.6.3*. Our cluster contains five nodes with *HBase* installed on each, as shown in Figure (4.4). Furthermore, there are several parameters that we had set. The *HBase* regions were pre-split following the recommended split strategy provided on the *HBase* site [42] and [33].

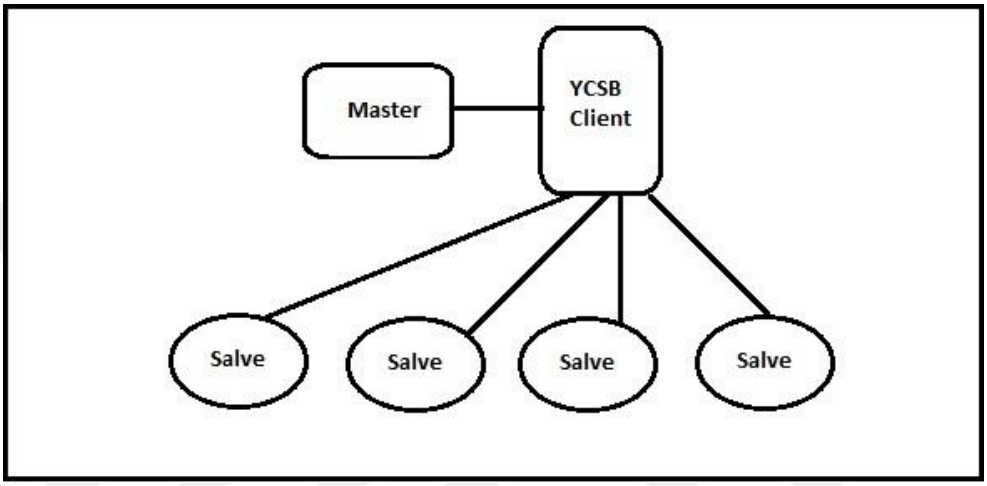

Figure (4.4) HBase configuration

CHAPTER 5

RESULTS

In this chapter, we present the benchmark results of three NoSQL database systems (*MongoDB*, *Cassandra* and *HBase*) against each other using a test framework created by Yahoo called the Yahoo Cloud Serving Benchmark (YCSB). Due to the many different factors between the three systems, such as the design, data model, data distribution across the cluster, etc., the performance of these systems will not be identical. YCSB shows the differences between these systems. The YCSB has been created to evaluate database systems in several scenarios. It has six inbuilt core workloads to simulate different operations related to database systems, such as read, write and update. These workloads are as follows:

- Workload-A (Update heavy workload): This workload is 50 percent of read and another 50 percent of write.
- Workload-B (Read mostly workload): This workload mixes 95 percent reads and 5 percent write.
- Workload-C (Read only): In this workload, it is a 100 percent read.
- Workload-D (Read latest workload): In this workload, the last inserted records are more famous.
- Workload-E (Short range): In this workload the query is for a short range of values instead of particular record.
- Workload-F (Read-Modify-Write): The client in this workload reads a record, modifies it and finally writes it.

The data used in this experiment was created by the YCSB. These data are illustrated as records stored in a table. The YCSB provides flexibility to control the record size, number of records, and even the number of fields. In our experiments, we used 1 million records each of size 1 kb. Each record consisted of 10 fields: the size of each record was 100 bytes. Due to the small size of the RAM in each node of the clusters

that we used, we were unable to increase the number of records to more than 10 million. Another important point to be mentioned is with regard to the influence of the processing speed of the nodes, which also has a direct effect on the performance of the database systems. There are several parameters that need to be set with appropriate values.

5.1 Load Phase

In each database system, we loaded a million records. From Figure (5.1), which shows the throughput versus average latency, it is easy to observe that *Cassandra* is the winner among the three systems in terms of throughput. *Cassandra* passed 19000 operations per second (as high throughput is desired). This high performance is due to *Cassandra's* data update occurring in memory, while these data are simultaneously written to disk. In terms of average latency, *HBase* has the lowest average latency (as low average latency is desired). This means that *HBase* performs well with heavy loads. Although *MongoDB* comes in the last place in this comparison, but comparing with the results in [33], the *MongoDB* foundation has improved it is product. The average latency is better than other releases of *MongoDB*.

Figure (5.1) Load Phase (1 million records)

5.2 Read and Update

As mentioned previously, workload A is a heavy update scenario. In this part, we highlight the update operation. From Figure (5.2), which shows throughput versus average latency, it is clear that *HBase* performs best based on throughput and average latency due to its write optimized design, in which its latency is lower than 0.5 ms. *Cassandra* also performed very well in comparison to *MongoDB. MongoDB* showed poor performance and we were unable to obtain higher throughput or a latency increase while we increased the target of the throughput (–t).

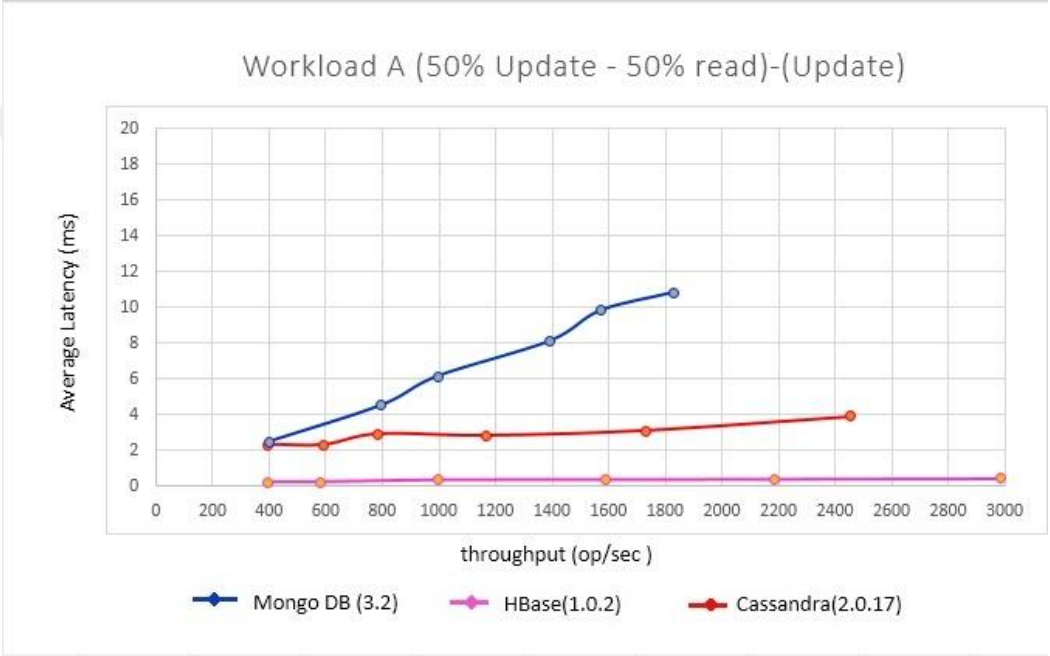

Figure (5.2) Read and Update

Tables (1,2,3) show the values of each database curve in figure (5.2)

Cassandra Read-update	
Average Latency	Throughput
2.330732	396.4024
2.31764	591.9594
2.934253	785.7589
2.8396	1168.162
3.11852	1729.242
3.904011	2455.283

Table (5.1)

MongoDB Read-update	
Average Latency	Throughput
2.48355	399.392
4.52793	797.0096
6.142658	995.5102
8.1165	1392.176
9.81749	1571.265
10.824936	1827.518

Table (5.2)

HBase- Read-update	
Average Latency	Throughput
0.23526	396.362
0.23673	578.436
0.35986	994.563
0.37864	1587.542
0.39686	2185.745
0.43425	2984.248

Table (5.3)

5.3 Read mostly

Workload B has 95% read and the reminder for the update. *MongoDB* performs best with lowest latency which increase with the increase of the target throughput, due to its support of memory mapped caching. *Cassandra* also performed well because of its key-row caching. Although the throughput of *HBase* is still high, the latency of *HBase* is the highest, as shown in Figure (5.3).

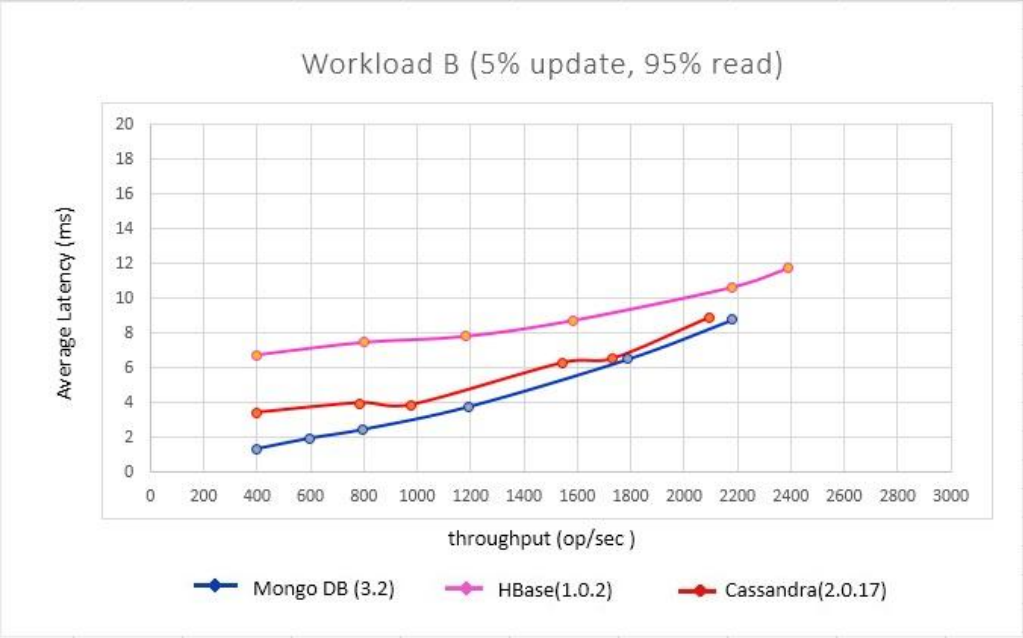

Figure (5.3) read mostly

Tables (4,5,6) show the values used in figure (5.3)

Cassandra (Read 95%-insert 5%)	
Average Latency	Throughput
3.426968	396.3349
3.9829282	785.6206
3.88002	977.6423
6.29791	1543.517
6.5293314	1729.161
8.918973	2094.372

Table (5.4)

MongoDB(Read 95%-insert 5%)	
Average Latency	Throughput
1.353671	399.3227
1.96302	598.5479
2.467718	797.499
3.786381	1194.244
6.514965	1787.469
8.795143	2180.216

Table (5.5)

HBase(Read 95%-insert 5%)	
Average Latency	Throughput
6.723	399.751
7.315	798.458
7.82	1178.254
8.712	1584.547
10.687	2178.214
11.701	2387.985

Table (5.6)

5.4 Read Only Operation

This workload has a 100% read operation. As we can see in Figure (5.4), *Cassandra* and *MongoDB* performed better than *HBase* in terms of latency. At the beginning, *MongoDB* performed even better than *Cassandra*, but with the increase of the throughput, we see that the average latency also increased. At some point, the latency of *MongoDB* became higher than the latency of *Cassandra*. *Cassandra* shows that it works better with high throughput than the other databases in this workload.

Figure (5.4) 100% Read

Tables (5.7,5.8,5.9) show the values used in figure (5.4)

Cassandra Read only	
Average Latency	Throughput
3.268094	297.9297
3.472531	494.1043
3.65133	881.859
4.03937	1262.556
4.5293314	1475.326
5.117724	1729.047

Table (5.7)

MongoDB Read only	
Average Latency	Throughput
1.409158	299.644
1.43743	498.982
2.3296	896.5152
3.76742	1193.845
5.72205	1588.915
6.544128	1785.905

Table (5.8)

HBase Read only	
Average Latency	Throughput
5.376	299.751
6.418	597.264
7.921	998.51
9.01	1298.289
9.254	1492.43
10.512	1794.23

Table (5.9)

5.5 Insert mostly Operation

We have set two parameters for this workload. We set the insert ratio to 90% and 10% read, and we focused on insert operations. Figure (5.5) shows how the database systems respond to this workload. Obviously *HBase* and *Cassandra* perform the best in comparison to *MongoDB* in terms of latency. The state latency is less than 1 ms. in contrast to *MongoDB*, the latency of which increases while increasing the throughput target. *Cassandra* and *HBase* also have the highest throughput.

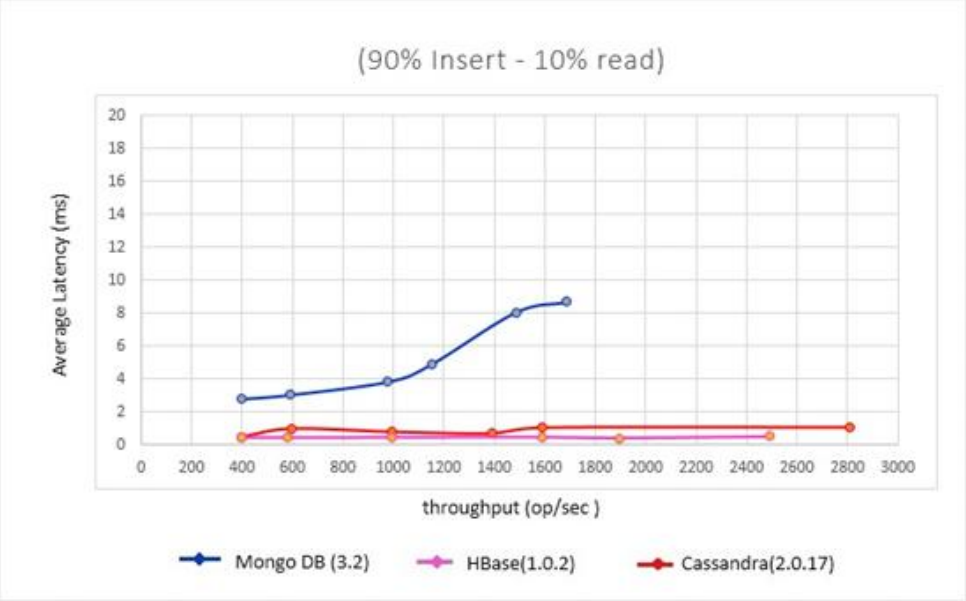

Figure (5.5) Insert mostly

Tables (5.10,5.11,5.12) below show the values which represent each curve in figure (5.5).

Cassandra-Insert mostly	
Average Latency	Throughput
0.50127	399.397
1.0320527	598.6124
0.82245	995.98
0.717604	1392.02
1.095473	1588.38
1.120561	2807.356

Table (5.10)

MongoDB- Insert mostly	
Average Latency	Throughput
2.80764	396.366
3.06553	591.84
3.85364	977.568
4.90574	1150.2
8.06184	1485.54
8.68458	1687.574

Table (5.11)

HBase - Insert mostly	
Average Latency	Throughput
0.43526	396.362
0.43673	578.436
0.45986	994.563
0.47864	1587.542
0.39686	1895.324
0.53425	2489.985

Table (5.12)

CHAPTER 6

CONCLUSIONS

In the last two-decades, relational database management systems (RDBMs) were the choice for many companies. Nowadays, data increase rapidly from many sources and in different forms, such as structured, semi-structured and unstructured. Because of the scaling obstacle that RDBM faces, the use of RDBMs is not sufficient with huge and complex data. Therefore, a need for new technology has appeared. NoSQL has overcome the scaling problem and it can deal with large volumes of data easily. In the world of big data, there are no good or bad database systems. Every database system has features which might meet a user's needs. In this thesis, we discussed and tested three kinds of NoSQL database system in order to reveal their capabilities and how they respond in different operations. Due to the architecture and design of each database that we tested, we have seen a different response for each with each operation. In our work, we used the Yahoo Cloud Serving Benchmark (YCSB), which is a framework designed by Yahoo to test database performance. According to the results obtained, we can conclude that *MongoDB* performed very well with low throughput, but not as well with high throughput. *Cassandra* and *HBase* performed very well under heavy loads due to their optimized designs. In the read operation, *HBase* has poor performance as we saw. If we compare the results that we obtained with the results in [33], we can conclude that the foundations that created these three database systems had improved their products. The latency for them is lower than before for all operations, especially in *MongoDB*.

Future work

Due the strong competition between the foundations, new improvements always present by these foundations to their products. We are plaining to use the next release of yahoo cloud serving benchmark and compare the latest versions of different NoSQL database systems. We will also test the performance of relational database

management system (RDBM) to present a comparative study with the NoSQL database management systems.

REFERENCES

1. Christof Strauch; Prof. Walter Kriha (2009): NoSQL database, university of Hochschule der Medien, Stuttgart

2. Lith A, Mattson J (2013), Investigating storage solutions for large data: A comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data. Dissertation, Chalmers University of Technology.

3. “NoSQL databases,” [Online]. Available: nosql-database.org. [Accessed 10 2 2016].

4. "Big data for dummies", Dr. Fern Halper, Marcia Kaufman, Judith Hurwitz, Alan Nugent 2013.

5. “Challenges and Opportunities with Big Data”. CRA.org. Retrieved Jan 2016.

6. Mariana Carroll, Paula Kotzé, Alta van der Merwe. 2012. Securing Virtual and Cloud.

7. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Lee, G., Patterson, D. A., Rabkin, A.,Stoica, I., and Zaharia, M. A View of Cloud Computing. Commun. ACM 53, 4 (2010), 50-58.

8. Velte, T., Velte, A., and Elsenpeter, R. Cloud Computing, A Practical Approach, 1 ed. McGraw-Hill, Inc., New York, NY, USA, 2010.

9. Burrows, M. The Chubby Lock Service for Loosely-Coupled Distributed Systems.

10. Wang, G., and Tang, J. The NoSQL Principles and Basic Application of Cassandra Model. In Proceedings of the 2012 International Conference on Computer Science and Service System (Washington, DC, USA, 2012), CSSS '12, IEEE Computer Society, pp. 1332-1335.

11. Brewer, E. A. Towards Robust Distributed Systems (abstract). In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA. (2000), p. 7.

12. Gilbert, S., and Lynch, N. A. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. SIGACT News 33, 2 (2002), 51-59.

13. CAP Theorem. http://www.abramsimon.com/cap-theorem/ (Accessed:05/05/2015).

14. Call me maybe: MongoDB. https://aphyr.com/posts/284-call-me-maybe-mongodb (Accessed: 05/05/2015).

15. Vogels, W. Eventually Consistent. Commun. ACM 52, 1 (2009),40-44.

16. 10gen.com: Home - MongoDB. http://mongodb.org/, 2009. [Online; accessed on 17-November-2015].

17. Chodorow, K., and Dirolf, M. MongoDB - The Definitive Guide: Powerful and Scalable Data Storage. O'Reilly, 2010.

18. Mongo DB Documentation Project. http://docs.mongodb.org/v2.6/ MongoDB-manual.pdf (Accessed: 12/05/2015).

19. 10gen.com: Querying - MongoDB. http://www.mongodb.org/display/ DOCS/Querying, 2009. [Online; accessed 17-November-2015].

20. Mongo DB Inc. http://www.mongodb.com/ (Accessed: 02/03/2015).

21. MongoDB Previews New Features at Global User Conference, MongoDB World, retirived from https://www.mongodb.com/press/new-features-at-global-user-conference on Januaray 2016.

22. Mongo DB Inc. retrieved from https://docs.mongodb.org/manual/replication/ on / (Accessed: September, 27,2015).

23. MongoDB Inc. retrieved from https://docs.mongodb.org/manual/ core/sharding-introduction/ on / (Accessed: September, 24,2015).

24. “Facebook’s Cassandra paper, annotated and compared to Apache Cassandra 2.0”. DataStax.com. Retrieved April 2014.

25. Lakshman, A., and Malik, P. Cassandra: A Decentralized Structured Storage System. Operating Systems Review 44, 2 (2010), 35-40.

26. Hewitt, E. Cassandra - The Definitive Guide: Distributed Data at Web Scale. Springer, 2011.

27. Teddyma, Learn Cassandra retrieved from https://teddyma.gitbooks.io/learncassandra/content/replication/turnable_consistency.html on November, 14, 2015

28. Welsh, M., Culler, D. E., and Brewer, E. A. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In SOSP (2001), pp. 230-243.

29. Ganesh, A. J., Kermarrec, A., and Massoulie, L. Peer-to-Peer Membership Management for Gossip-Based Protocols. IEEE Trans. Computers 52, 2 (2003), 139-149.

30. Apache Hive TM. https://hive.apache.org/ (Accessed: 02/03/2015)

31. Welcome to Apache Pig! https://pig.apache.org/ (Accessed: 02/03/2015).

32. Wei Tan (2014). NoSQL Data Stores in Internet-scale Computing. Tutorial at IEEE ICWS, June 27, 2014, Alaska, USA. Retrieved from http://researcher.watson.ibm.com/researcher/files/us-wtan/NoSQL-Tutorial-ICWS-2014-public-WeiTan.pdf on February 26, 2016.

33. Kuldeep Singh. 2015. Survey of NoSQL Database Engines for Big Data, Aalto University.

34. Christopher Jay Choi. 2014:A STUDY AND COMPARISON OF NOSQL DATABASES, CALIFORNIA STATE UNIVERSITY, NORTHRIDGE.

35. Benchmarking Top NoSQL Databases, Apache Cassandra, Couchbase, HBase, and MongoDB, retrieved from http://www.endpoint.com/ on January, 16, 2016.

36. Hecht, R. and Jablinski, S. 2011. NoSQL Evaluation A Use Case Oriented Survey. Proceedings International Conference on Cloud and Service Computing, pp. 12-14.

37. Cooper, B., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R.: Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10). ACM, New York, NY, USA, 143-154, 2010.

38. George, L. HBase - The Definitive Guide: Random Access to Your Planet-Size Data. O'Reilly, 2011.

39. Aiyer, A. S., Bautin, M., Chen, G. J., Damania, P., Khemani, P., Muthukkaruppan, K., Ranganathan, K., Spiegelberg, N.,Tang, L., and Vaidya, M. Storage Infrastructure Behind Facebook Messages: Using HBase at Scale. IEEE Data Eng. Bull. 35, 2 (2012), 4-13.

40. O'Neil, P. E., Cheng, E., Gawlick, D., and O'Neil, E. J. The Log{Structured Merge-Tree (LSM-Tree). Acta Inf. 33, 4 (1996), 351-385.

41. Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., Weaver, D., and Yerneni, R. PNUTS.

42. "Create Split Strategy for YCSB Benchmark". Apache.org Retrieved March 2013.

43. MongoDB Connector for Business Intelligence Retrieved from https://docs.mongodb.org/bi-connector on February, 3,2016

44. Setup Mongo DB with sharding infrastructure retrieved from http://wayneye.com/Blog/Setup_MongoDB_With_Sharding_Infrastructure on January, 4, 2016.

45. Introduction to Column Family with Cassandra retrieved from http://www.edureka.co/blog/introduction-to-cassandra-column-family/on September, 23,2015.