QUANTITATIVE ANALYSIS OF SCALABLE NOSQL DATABASES

by

SURYA NARAYANAN SWAMINATHAN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2015

To my professor Dr. Ramez Elmasri

ACKNOWLEDGEMENTS

I express my sincere gratitude to my advisor Dr. Ramez Elmasri. I would also like to thank Dr. Leonidas Fegaras for his support. Special thanks to Professor David Levine for taking a special interest in my research beyond the call of duty.

I am grateful to Professor Ronald Cross for his support, motivation and guidance. I would also like to thank my research mates Mr. Mohammadhani Fouladgar, Mr. Upa Gupta, Mr. Neelabh Pant and Mr. Vivek Sharma for their support and camaraderie.

Finally, I extend my gratitude towards everyone in Computer Science department for the memorable experiences I had during my masters

November 20, 2015

ABSTRACT

QUANTITATIVE ANALYSIS OF SCALABLE NOSQL DATABASES

SURYA NARAYANAN SWAMINATHAN, M.S.

The University of Texas at Arlington, 2015

Supervising Professor: Ramez Elmasri

NoSQL databases are rapidly becoming the customary data platform for big data applications. These databases are emerging as a gateway for more alternative approaches outside traditional relational databases and are characterized by efficient horizontal scalability, schema-less approach to data modeling, high performance data access, and limited querying capabilities. The lack of transactional semantics among NoSQL databases has made the application determine the choice of a particular consistency model. Therefore, it is essential to examine methodically, and in detail, the performance of different databases under different workload conditions.

In this work, three of the most commonly used NoSQL databases: MongoDB, Cassandra and Hbase are evaluated. Yahoo Cloud Service Benchmark, a popular benchmark tool, was used for performance comparison of different NoSQL databases. The databases are deployed on a cluster and experiments are performed with different numbers of nodes to assess the impact of the cluster size. A benchmark suite is presented on the performance of the databases on its capacity to scale horizontally and

on the performance of each database based on various types of workload operations (create, read, write, scan) on varying dataset sizes.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The prominence of cloud computing and advancement of Internet has necessitated for a database to scale horizontally, handle high volumes of concurrent read and write requests, and to provide an efficient storage platform. The solution to the aforementioned requirements has emerged in the form of a variety of databases referred to as NoSQL databases. There has arisen a challenge to choose the database most suitable for a particular application with the emergence of NoSQL databases. There are several benchmark reports and research papers available on the internet [1][2][3]. Most Benchmarking and comparative study reports focuses on the overall performance of a NoSQL Database on a single system or small cluster. In these benchmarking reports, the scope of the performance testing parameters was limited. In this thesis, we perform a comprehensive performance analysis on a comparatively larger cluster with additional focus on the scalability of different NoSQL databases. Some unique scenarios, which also need to be considered during NoSQL performance analysis, are defined.

Relational Databases have been dominant and the most used databases for many technologies over the past four decades. Several Online Transaction Processing Systems (OLTP) use Relational Database Management Systems (RDBMS) as a common choice, due to their ACID guarantees. RDBMS is the most efficient system for storing and operating predefined organized data. Currently, the increase in the number of internet users and the consequent increase in velocity of the data generated, disparate types of data that are not congruent to a structure have showcased the

shortcomings of relational databases. The difficulty in managing a very large amount of data in a distributed database, a rigid relational schema, the lack of conformance of data to a structured format and exorbitant licensing fees has propelled the developer community to create a new kind of data stores called NoSQL databases.

NoSQL databases came into existence due to the need for a system capable of handling massive amounts of data that offers flexibility without compromising the performance of the system. NoSQL system serves as a gateway for more alternative approaches outside the habitual relational approach for data persistence. The multi-table correlation mechanism (known as the join operation), which allows data from multiple tables to be combined, affected the database scalability in relational databases. This has been addressed in NoSQL databases because they do not have such operations, which has made NoSQL databases highly scalable at the cost of not allowing complex queries using a high-level query language. The NoSQL databases can be distributed across a high distributed and a diverse environment globally and achieve good performance.

Currently, there are several NoSQL databases in existence and are prominent for handling large amounts of data. While this is oftentimes the case, it should be understood that each database takes different approaches and has been designed to cater to specific needs. Except for being non-relational in nature, the databases do not share any commonality, and may not have the best performance results in a given circumstance. Therefore, it becomes essential to examine methodically, and in detail, the behavior of different databases under different workload conditions.

In this thesis, three NoSQL databases MongoDB, Cassandra and Hbase were evaluated. Hbase and Cassandra are column-oriented databases. MongoDB is a document-oriented database. The performance of each database based on various types of operations on a certain set of data, along with the capacity of the database

to scale horizontally, was captured. Yahoo Cloud Service Benchmark [3], a NoSQL benchmarking tool, was used for generating the data and workloads used in our experiments. The benchmark tool provided different workloads such as read only, write heavy, read-modify-write, scan etc. Additional custom workloads were created for a comprehensive study.

The rest of this thesis is organized in the following fashion. Chapter 2 gives a background into the emergence of NoSQL systems. In Chapter 3, the different characteristics of SQL and NoSQL systems are discussed. Chapter 4 classifies different NoSQL systems. Chapter 5 explains in some detail about the benchmarking tool, the configurations and the implementation procedure. The chapter also defines the scope of the experiments. The results of the experiments are analyzed in Chapter 6. Finally, Chapter 7 provides the conclusion and future work.

CHAPTER 2

BACKGROUND

The percentage of people using the Internet has been rising exponentially. Currently, two-fifths of the world's population is connected to the internet and as a result 2.5 quintillion (10^{18}) bytes of data [4] is generated everyday. Though relational databases have monopolized data storage in the past, organizations have begun to consider alternatives to legacy relational infrastructure to augment their data management needs. This chapter discusses the emergence of a new brand of data stores called NoSQL databases that provides a set of new management features. Section 2.1 discusses the emergence of distributed databases. Section 2.2 provides a brief history on the origin of the term NoSQL. Section 2.3 discusses the evolution of NoSQL systems. Finally, section 2.4 gives an insight into Eric Brewer's conjecture and relevance of CAP theorem with respect to NoSQL databases.

2.1 Distributed databases

The demand for decentralized databases grew in the 1980's due to developments in social and technological fields. The need for expedited response time and faster access to information exposed the drawbacks of a centralized database [5]. The advent of globalization, the inflation in demand for information, and global market presence entailed a distributed database management approach. Distributed databases emerged by combining the concepts of database systems and distributed computing. The issues relating to fragmentation of data across multiple nodes, replication of

data, and distributed query and transaction processing were addressed in distributed database management technologies.

The working definition for a distributed database is that it is a collection of autonomous functioning interrelated databases interconnected by a computer network, with a common objective to coordinate in performing a designated task, while hiding the fact that the resources and data are distributed across a network. The term "autonomous functioning" in the definition can refer to autonomy in database design, communication, and execution [6]. The degree of freedom in management of data, sharing of information, and implementation of tasks, across multiple nodes is proportional to the level of flexibility, availability, scalability and performance in the distributed database.

In the past decade, in order to deal with the substantial amount of data being collected, many new technologies, commonly referred as big data technologies, have emerged. The big data technologies are used for storage, analytics, and data mining, and their inception can be traced back to distributed computing and distributed database systems. NoSQL systems are a part of the big data technology that leverages the concepts of distributed databases to provide viable systems for storing large amounts of data.

2.2 Neologism

Carlo Strozzi first used the term "NoSQL" in 1998 for a shell based relational system [7]. The small-scale relational system was designed consciously not to use SQL as a query language, hence the term NoSQL. Strozzi prefers "NoREL" to refer to the current NoSQL systems since they are non-relational.

The term NoSQL started gaining prominence in 2009, at a meeting organized by Johan Oskarsson in San Francisco for people developing open source, distributed,

non-relational databases [8]. Eric Evans gave the name of the meeting, "NoSQL meetup". After the meeting, the term NoSQL became popular. Nowadays, the common interpretation for the term NoSQL is Not Only SQL.

2.3 Evolution

The amount of unstructured and semi-structured data generated skyrocketed in the mid 2000's. This increase was attributed to the rise in prominence of several interactive applications on the web and consequent increase in the number of users. The popularity and demand of the online applications was based on the extent of use of unstructured and semi-structured data in the application. The rigid schema-based approach of relational database technology made it increasingly difficult to handle unstructured/semi-structured data. Companies like Google, Amazon, Facebook, etc. first discovered these limitations. The lack of alternate systems to support their application requirements, and the need to address the growing demand led to the companies inventing a new data management approaches for their requirements. The new databases were tailored to satisfy the requirements of their in-house applications.

The first significant contribution in the field of NoSQL systems can be related to Google's Bigtable. In 2006, Google presented the paper on Bigtable at the Operating Systems Design and Implementation conference [9]. Bigtable was a flexible high performance system designed to manage structured data on a distributed environment. It was designed to scale to petabytes of data and over thousands of machines. The goal of Bigtable was to ensure scalability, high availability and performance, with an extensive applicability. Bigtable is used by more than sixty Google products including Google search engine, Google Earth and Google Finance.

In 2007, Amazon presented a paper on Dynamo at the 21stACM Symposium on Operating Systems Principles [10]. Dynamo is a highly available key-value store

designed to cater to the needs of Amazon's e-commerce applications. The goal of this database was to provide an always-available experience to the users of Amazon web applications.

The pioneering work by Google and Amazon attracted interest across several companies facing similar problems. This resulted in several open source NoSQL projects. The projects focused on developing different categories of NoSQL databases such as document-oriented databases, in-memory databases, graph databases etc. Bigtable and Dynamo set the precedence for many NoSQL databases that followed. Currently, there are more than 225 recognized NoSQL databases [11].

2.4 Brewer's Conjecture

Dr. Eric Brewer, a professor at the University of California, Berkeley and co-founder of Inktomi Corporation, in 2000, at the ACM conference on Principles of Distributed Computing, presented a conjecture known as CAP theorem [12]. According to the conjecture, when designing a distributed application, it is not possible to achieve, at the same time, all three of the commonly desired properties - consistency, availability and partition tolerance. The conjecture was later proved in the asynchronous network model [13]. However, there was a growing concern about the shortfalls of the CAP theorem. In this section, the CAP theorem is explored in detail. Section 2.4.1 explains the CAP theorem, and section 2.4.2 elaborates on the effects of CAP in NoSQL systems. In section 2.4.3 an overview of the issues in CAP theorem is presented. Section 2.4.4 provides an alternate view, suggested by several experts, in understanding CAP theorem.

2.4.1 CAP theorem

According to Eric Brewer's CAP theorem, it was not possible to have more than two of the three commonly desired properties for a distributed system in a shared-data network with replication:

- Consistency (C) - all the replicated copies of an item will have the same data values.
- Availability (A) - the system is always available for reads and writes
- Partition tolerance (P) - if the nodes get partitioned, the system will tolerate this by continuing to operate.

This can be explained further by considering the trade-off between each of the properties.

- Available / Partition tolerant - This trade-off ensures the system is available for reads and updates, but once an item is updated at one node, the system may become inconsistent because values of that same item at other nodes can be different.
- Consistent / Partition tolerant - To ensure this trade-off, to preserve consistency across a partitioned network, all the other nodes must act as if the data stores at a partition is unavailable. The reason is that the partitions of nodes in the network cannot communicate with each other.
- Consistent / Available - For a system to be both consistent and available, the nodes must communicate with each other and the nodes must be ready to accept read and update requests. This is possible only when the network is never partitioned. Centralized databases are examples of such systems. Once a partition occurs, the system essentially crashes and so becomes unavailable.

In the current scenario, the system designers cannot drop tolerance to the network partitions due the need for large-scale distributed systems. Hence, the choice is restricted to between consistency and availability.

2.4.2 CAP in NoSQL systems

CAP impacts various scalable NoSQL systems. Scalability was one of the contributing factors for the prominence of NoSQL systems, in which case, there is a high probability for the system to be partitioned over a network. As a result, the system has to select a tradeoff between consistency and availability. The requirements of consistency on most relational systems have made most NoSQL systems favor availability because their application requirements do not require strict consistency [14].

Different applications may require different tradeoffs depending on the type of data. An e-commerce application, e.g. amazon.com, prioritizes swift response to the users. This necessitates the system to be highly available (for read and write requests), which requires data to be replicated and thus forfeiting consistency to an extent. Most social media websites, such as Facebook and Twitter, also favor availability over consistency.

A distributed lock manager developed by Google, called chubby lock service, which finds its application in Bigtable, provides strong consistency. By maintaining synchronized logs using a replicated state machine protocol consistency is ensured [15].

2.4.3 Issues in CAP

CAP, since its inception, has paved the way for several researchers to venture into a wider spectrum in designing distributed systems by considering the tradeoffs. This is evident from the number of new systems that has emerged in the past decade.

Consequently, developers of new systems started to question the authenticity of the CAP theorem. The "2 of 3" tradeoff between consistency, availability and partition tolerance was misleading because of the assumption to consider each of the properties as a Boolean rather than a spectrum.

To elucidate, consider the explanation from section 2.4.1 on the tradeoffs. The initial problem is with the definition of consistent/partition tolerant tradeoff, which gives a picture that the system is not available. But, in reality, this tradeoff only sacrifices the degree of availability of the system. Conversely, a system that forfeits consistency is inclined towards forfeiting consistency constantly regardless of the system being partitioned. This means there is no symmetry between availability and consistency in CAP [16].

Secondly, there is no clear distinction between consistent/partition tolerant and consistent/available systems. A consistent/partition tolerant system compromises on the availability and a consistent/available system is not tolerant of any network partition. But, a consistent/available system, if it suffers network partitioning, will become unavailable. Hence, these two categories of systems are inherently similar. This reduces the tradeoff to only two types consistent/partition tolerant or consistent/available and available/partition tolerant.

Due to the aforementioned issues, the validity of the CAP is under question. On the positive side, this ambiguity has opened the doors for system designers and researchers to consider options outside of CAP and to think beyond the limitations of CAP.

2.4.4 CAP - an alternate view

Eric Brewer provided clarifications that addressed the issues relating to the CAP theorem. Brewer accepted that choosing two of the three properties was "misleading"

because "it tended to oversimplify the tensions among properties" [14]. Brewer also redefined the goal of CAP to achieve maximum availability and maximum consistency possible for any given application.

Daniel Abadi, an associate professor at Yale University, addressed the issue of asymmetry in CAP by rewriting CAP as PACELC [16]. According to Abadi, the root of the asymmetry between consistency and availability in CAP is because a system sacrifices consistency either for availability in case of a partition or for latency in case of no partition. Defining PACELC, if a system is partitioned (P), the tradeoff between availability (A) and consistency (C) is considered, if the system is not partitioned (E), the tradeoff between latency (L) and consistency (C) is considered. Considering PACELC as an alternative to CAP gives a better understanding while designing a system and also removes the ambiguity in CAP.

CHAPTER 3

SQL vs. NOSQL SYSTEMS

A paradigm shift in data management has occurred, based on augmenting the use of SQL systems with utilization of NoSQL systems for the applications that are not suitable for SQL. Thus it is important to analyze the characteristics of both SQL and NoSQL systems so that system developers will identify the most appropriate system for their applications. This chapter discusses some of these characteristics and tradeoffs. Section 3.1 reviews the ACID (Atomicity, Consistency, Isolation, and Durability) properties used in SQL systems versus the BASE (Basically available, soft state, and eventually consistent) properties proposed for NoSQL systems. Section 3.2 is a comprehensive analysis of scalability in both types of systems. In section 3.3, the rationales behind the characteristics that contrast functionality/features vs. speed/scalability are examined. Section 3.4 discusses the replication models. In section 3.5, fragmentation of data in both types of systems is appraised. Query language capabilities are discussed in section 3.6.

3.1 ACID vs. BASE

Relational systems gained prominence for their ability to ensure consistency of data. A strong concurrency and recovery model is the result of the implementation of ACID guarantees in the database. The ACID (Atomicity, Consistency, Isolation, and Durability) properties ensure that a transaction is reliably implemented without any undesirable consequences. These qualities can be indispensable for systems where consistency of data is of paramount importance. It is important that systems such

as airline reservation systems, banking systems and commercial transactions such as stock trading systems have strong consistency. These systems need the data to be accurate and can sometimes afford to compromise on the availability and the response time of the system.

ACID properties, although important, are incompatible with very large distributed systems where upto-to-the minute consistency is not required. Social media applications and web searching applications are examples of such systems. Complying with Brewer's conjecture (discussed in section 2.4.1), when a system is designed to span across multiple servers, by enforcing ACID guarantees across a partitioned database, the availability of the system is compromised.

Eric Brewer, to overcome the limitations of ACID, proposed a new model for database development called BASE (Basically available, soft state, and eventually consistent) [12]. An alternative to ACID, BASE ensures availability of the system by trading off the consistency of the database.

BASE is logically opposite to ACID and its application can be found in several NoSQL systems. BASE can be explained as follows:

- Basically Available - guarantees system availability. Each request will fetch a response even though the data might be incorrect or inconsistent.
- Soft state - indicates the state of the system may change over time, even without input.
- Eventually consistent - indicates the system will become consistent over a period of time, provided the system does not receive any inputs during that period.

The availability of the systems in BASE is achieved by embracing partial failure of nodes and ensuring complete uptime. This involves methodical planning on fragmenting the data in a distributed database. In case of a system failure at a database node, the impact is only on a subset of data rather than the entire database. As a

result, the user of the system will experience a system, which responds to the requests on most occasions. Table 3.1, adapted from [12], shows the comparison between ACID and BASE.

Table 3.1. ACID vs. BASE

Characteristics	ACID	BASE
Consistency	Strict	Linient
Preference	Isolation and Consistency	Availability
Scalability	Poor	Very Good
Concurrency	Slow	Fast
Query Language	Powerful	Weak

In the recent years, NoSQL systems are preferred for data storage by several e-commerce applications. For instance, Amazon's e-commerce platform has Dynamo as its underlying storage technology. Dynamo targets applications that operate with weaker consistency and does not provide any isolation guarantees [10]. These systems allow small probability of concurrent transactions to interfere with each other compromising on the ACID guarantees. Dynamo focuses on high availability to its users.

3.2 Scalability

Scalability can be defined as the extent to which the system can expand its capacity without interrupting the system [6]. In the current scenario, scalability of a database is measured by the ability to handle increasingly large amounts of data (usually in petabytes) in addition to providing uninterrupted service to the application. This section discusses the possible kinds of scalability in a distributed environment.

3.2.1 Vertical scalability

Vertical scalability is also known as scaling-up - upgrade by adding more resources to the existing machines, as shown in Figure 3.1 (adapted from [17]). As each machine runs out of capacity, a more powerful box is replaced with that machine. This is achieved by increasing the capacity of individual nodes in system by adding more RAM, processing power and storage disks. Traditional relational systems using SQL favor vertical scaling.

However, the disadvantage of Vertical scalability is that the exponential increase in the cost of the hardware. Figure 3.2, adapted from [18], shows the cost involved in increasing the number of processors.

3.2.2 Horizontal scalability

Horizontal scalability is also known as scaling-out - upgrade by adding new machines, as shown in Figure 3.1 (adapted from [17]). As each machine runs out of capacity, a new machine is added to the distributed system. In horizontal scaling, the capacity of the existing machines is not increased; instead more nodes are added to the cluster.

Several NoSQL systems including Cassandra, MongoDB, Hbase, and Dynamo are horizontally scalable. In order to compete with NoSQL systems, newer relational SQL systems also provide horizontal scalability. For example, Facebook primarily stores user data in MySQL databases sharded over 4000 MySQL server instances [19]. Horizontally scalable systems are cost efficient compared to vertically scalable systems as shown in Figure 3.2.

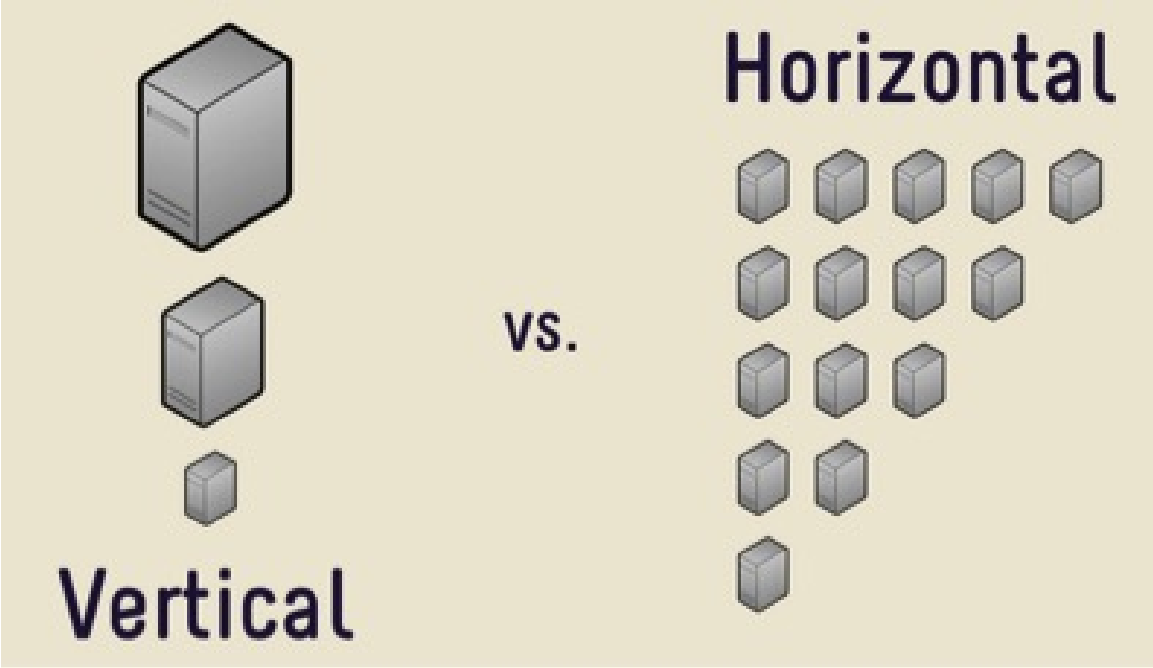

Figure 3.1. Vertical scalability vs. Horizontal scalability.

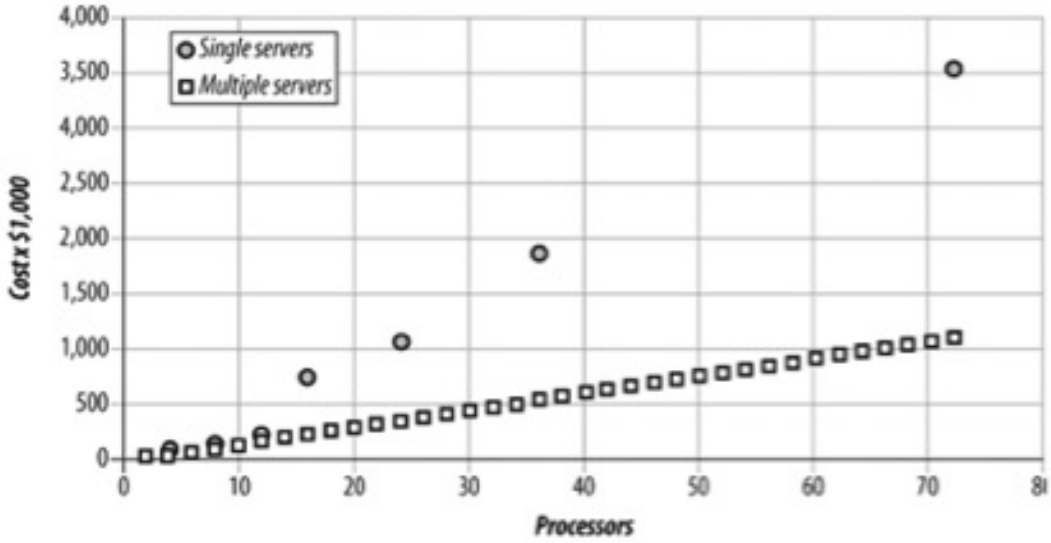

Figure 3.2. Scaling with Single server (scaling-up) and Multiple servers (scaling-out).

3.2.3 Elastic scalability

Elastic scalability is both scaling-out and scaling-in - the ability to grow or shrink by adding or removing more hardware nodes dynamically based on the needs of the system. Elastic scalability is mostly used in cloud platforms. Most popular application of elastic scalability is Netflix. Netflix uses three NoSQL databases: SimpleDB, Hbase and Cassandra. The feature of Cassandra and Hbase to scale horizontally and dynamically, without the need to re-shard or reboot make it elastically scalable [20].

3.3 Speed vs. Features

Technologies are not perfect replacement for one another. When an old system is replaced by a new system, there is always a tradeoff. Relational infrastructure is the most suitable platform to host applications where complex querying capabilities are needed. The option to utilize wide range of features in relational systems comes at the cost of compromising on the speed and scalability of the system. Synonymously, a NoSQL system prioritizes availability and reliability in exchange for limited querying capabilities and limited consistency. In this section, the Speed vs. Features trade off between different systems is reviewed by analyzing Figure 3.3, which is adapted from [21]. In section 3.3.1, the SQL region specified in Figure 3.3 is examined and section 3.3.2 discusses the position of different NoSQL systems in the graph.

3.3.1 SQL region

Relational DBMS (RDBMS), as shown in Figure 3.3, offers a wide variety of features. The complex querying capability, including powerful JOIN operations and the procedural language extensions to SQL (PL/SQL) in relational databases have made expensive operations (involving multiple nodes and multiple tables) look sim-

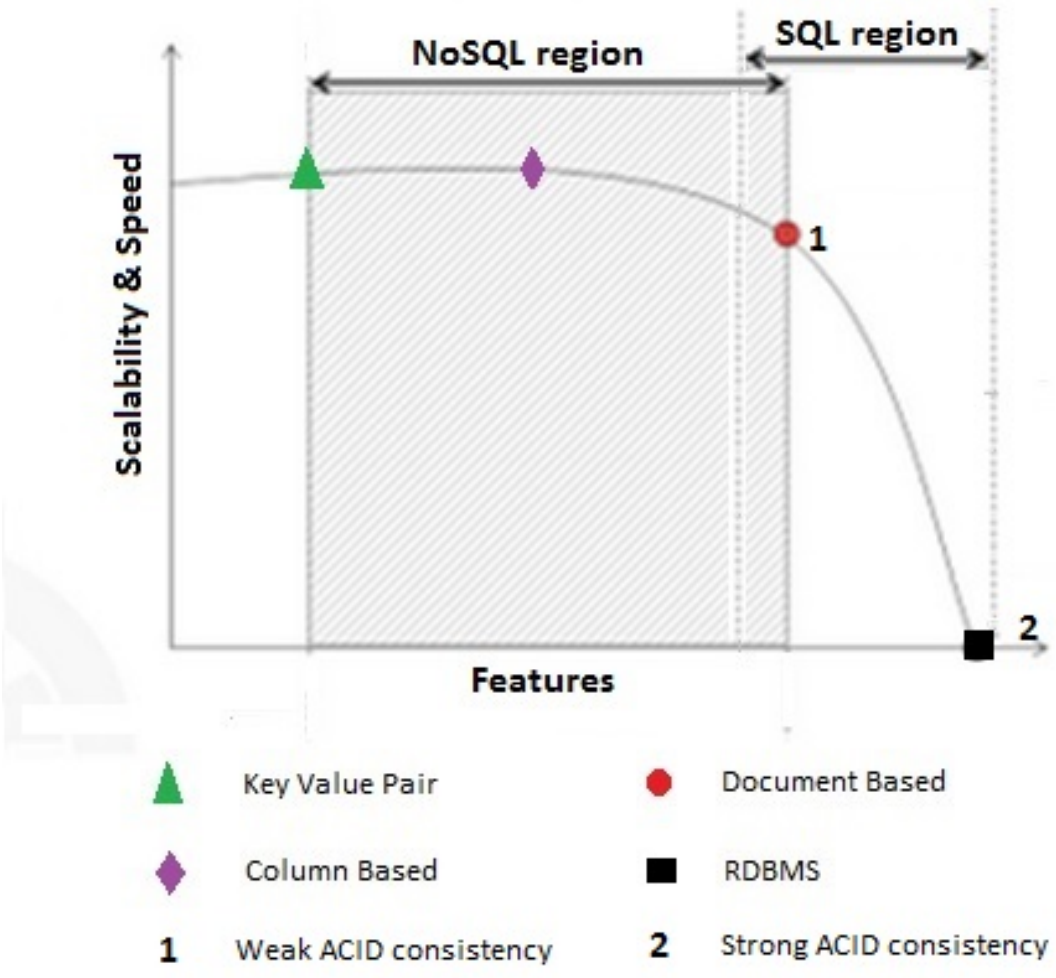

Figure 3.3. Speed and Scalability vs. Features.

ple. Two-phase locking can be implemented to ensure concurrency. This prevents dirty read and lost updates in the database. UNDO/REDO logs and ARIES recovery algorithm help recover databases from failures. Strict scheduling of transactions ensures serializability [6]. The developers and the system architects, who design the system, depending on the requirements, may choose to implement only a subset of available concurrency and recovery (ACID) features.

However, implementing all of the available features will impact the speed and scalability of the system proportionally. Increased latency in the database can be attributed to the uncompromising behavior of the RDBMS to enforce ACID guarantees. The normalization in RDBMS disintegrates the data into multiple tables to reduce data redundancy, but this leads to JOINs to combine back the data from different tables and service the requests. This normalization of data in RDBMS hinders the system's ability to scale across multiple servers. In applications, where the desire for SQL or need for ACID transactions is important, relational systems will still be a unanimous choice.

SQL region in Figure 3.3 marks the area within which a relational database can be plotted, depending on the database policy framed for a particular database. The spectrum ranges from strong ACID consistency with limited scalability and speed, to relaxed ACID consistency to improve scalability and speed.

3.3.2 NoSQL region

Figure 3.3 illustrates, in a straightforward manner, the speed and scalability of the NoSQL databases. The NoSQL space marked in the graph slightly varies, from low to high, based on the features offered by each NoSQL systems but remains steady on speed and scalability. The flexible schemas, lack of constraints and weak consistency are the contributing factors for quicker data access. This phenomenon can be traced to the BASE properties. Nevertheless, NoSQL systems are not without drawbacks. The lack of a strong querying language like SQL and a multi-table correlation mechanism like JOINs has made complex operations impossible or an extremely arduous task for the programmers.

MongoDB [22], a document store database, is a notable mention in Figure 3.3. MongoDB falls on the edge of the NoSQL space with maximum features. This aspect

of comparatively high features in MongoDB can be associated to JSON (JavaScript Object Notation), a minimal readable format for structuring data. MongoDB documents are stored as BSON (Binary JSON), an efficient variation of JSON [6] and can be queried on using MongoDB query language and JSON Query language (JSONiq) [23].

Column-based data stores such as Hbase [24] and hybrid stores such as Cassandra [25] fall in the middle of the NoSQL region between Document store and Key-value store, balancing features and speed/scalability. The systems are designed to satisfy the requirements of large-scale distributed social media applications.

Key-value stores like Memcache, Redis [26], and Dynamo [10] are the NoSQL databases with minimal features but have better scalability and speed. The Key-value stores are part of the NoSQL region but the databases are plotted on the lowest end of the features scale.

3.4 Data Replication

The quest to answer the challenges in managing very large distributed databases, as a result of the surge in the generation of subscriber driven web content, has made implementing data replication in the system inevitable. Enforcing an efficient replication strategy is an integral part in enhancing availability of data [5]. Replication, as a term, may change it's meaning based on the context to which it is applied. One form of replication is knows as backup, which generally refers to synchronizing the data between an on-line database and an off-line backup of the database. This synchronization can be achieved through incremental online backup (hot backup), offline backup (cold backup) and storing of data in tape drives. The main purpose of backup in relational DBMS is to facilitate recovery from failures.

In a distributed NoSQL or SQL database, replication refers to storing the multiple copies of the same data on different nodes to ensure system reliability and availability and to assist in concealing any node failures [27]. This subsection covers the concepts related to replication. In section 3.4.1, different types of replication in a distributed database are presented. The section concludes with an insight into different replication strategies.

3.4.1 Types of replication

For applications whose users are geographically diverse, replication is often the most productive method for data access. Depending on the requirement for reliability of the application, the degree of replication of the database is determined. Replication in a distributed database system can be broadly classified into three types: complete (total) replication, fragment replication and no replication [6].

3.4.1.1 Complete (total) replication

A system, distributed over a network, implementing a fully replicated database is rooted to achieve higher reliability and availability. The term "fully replicated database" refers to the replication of the entire database at every node [6]. A fully replicated database will provide availability in the event of node failures or network partitions. In case of a system that is distributed globally, performance of the system is enhanced by localizing the access to the data i.e. by retrieving the data from the secondary copy present at the location closer to the access points [5]. However, performance of the system in handling update operations is slower as each update has to propagate to all copies. Concurrency control and database recovery are more expensive in case of fully replicated database.

3.4.1.2 Fragment replication

The drawbacks of complete replication can be overcome, to an extent, by replicating only some parts of the database. In partial replication, only specific fragments of data in the database are replicated. Only fragments of data required by the application or the user, to be obtained from the database instantly, is replicated. Fragmentation of a table can be horizontal fragmentation (known as sharding, see section 3.5.1.1), vertical fragmentation, or hybrid fragmentation [6].

3.4.1.3 No replication

Logical opposite of complete (total) replication. A distributed database, with no replication, has a data item stored exactly in one location. In such cases, data needs to be in a denormalized form. No replication may also be referred as "nonredundant allocation" [6]. The probability of an application, with a non-replicated distributed database, to provide an always-available experience to the user is very low.

3.4.2 Replication strategies

When designing a very large distributed system, the system developers need to decide on the replication strategy from a broad range of options. The goal is to choose an option that would be ideal for that specific system. Factors such as type of transactions (local or global transactions), degree of replication (no replication, partial replication or complete replication), degree of consistency (strong vs. weak), extent of replication transparency (restricted vs. full), synchronizing the write operations between different replicas on different nodes, etc. influence the decision in choosing a replication strategy. But the rudimentary factor in choosing a replication strategy is to decide on which node in the distributed system the write operations will be

forwarded first. Based on the above-mentioned factors, the replication strategy can be broadly classified into decentralized replication and centralized replication [5].

3.4.2.1 Decentalized replication

Decentralized replication technique is also known as master-master replication or multiple-master replication. The read and write operations are allowed in the local copy of the database and the changes are then propagated to other sites. It is possible that concurrent users update the value of the same data item stored at different sites (replicas). This could result in the system being temporarily inconsistent. Depending on the desired consistency level of the system, the concurrent writes issue can be addressed. However, a reconciliation method involving undo/redo operations is required to ensure synchronization of data in each site. Systems that implement decentralized replication are highly available and have quick recovery in case of a failure of one node, but weak in consistency.

Decentralized replication is favored in systems that are required to be available for reads and writes even when the system is disconnected from the network. For example, sales forces and personal digital assistants (PDA). The local copy of the database is synchronized periodically when the personal devices/systems are connected to the network.

3.4.2.2 Centalized replication

Centralized replication technique is also known as master-slave replication or single-master replication. In this classification of replication, the writes must be first applied to only one site called the master site and then propagated to other replica sites referred to as slave sites. The propagation of updates from the master site to the slave sites can be immediate (synchronous) or deferred (asynchronous). In case

of deferred propagation, the read requests processed at the slave sites may not return the most recent data.

In some variations of centralized replication known as "primary copy centralization technique", the location of the master copy is dependent on the data item [5]. To elucidate further, the master copy of a data item "a" can be stored in a data center in Asia, while the master copy of a data item "z" can be in a data center located in Europe.

Centralized replication is mostly preferred in systems that require centralized processing [5] e.g. banking systems and billing systems. However, the major drawback in having a single master site is that the availability of the system could be affected if the master site crashes. This scenario is highly possible, if the number of update requests received at the master site exceeds the processing capacity of the system thus creating a bottleneck.

3.4.3 Replication in SQL and NoSQL systems

Relational systems offer distributed capabilities with replication. For example, MySQL [28] offers centralized replication while Oracle [29] offers both centralized and decentralized replications. Centralized replication in MySQL systems is implemented by copying the changes in the master site to other physical locations (slaves). The main purpose of centralized replication in MySQL system is to ensure availability and scalability. The slave sites are mostly used for running analytical queries and perform back-up operations, thus reducing the number of operations performed on the master site. In case of Oracle databases, decentralized replication is made possible by propagating the updates from one master site to another. All the master sites work methodically to ensure data integrity and consistency. Centralized replication in Oracle is supported through materialized views [29].

In NoSQL systems, the replication strategies available for implementation depend on the requirements of the application. Cassandra does not have master-slave architecture. A decentralized replication method is used. Column-based systems such as Hbase supports both centralized and decentralized replication. In addition to that, Hbase offers another variation of replication known as cyclic replication [24]. Document-based systems such as CouchDB [30] use centralized replication. In MongoDB, a variation on the master-slave approach is used.

3.5 Fragmentation

In the current scenario, with the growing customer base, it is impossible to predict the access behavior of the users for a given application. Thus, it becomes important to have a system design that embraces sporadic access behavior. This entails the architects, who design the system, to take into account the arrangement of data across different sites. Fragmentation is breaking up of a database or a database table/file into logical units called fragments [6]. The process of fragmentation in a distributed database system is desirable for the following reasons: Localized data access and controlled network traffic. Frequently accessed data are identified and are made available locally to a specific site where the access requests are high. As a result, network traffic is balanced and managed at a specific site and the efficiency of access to data is enhanced.

There are several factors to be considered in the process of fragmentation. First, transparency of fragmentation, hiding the information about the splitting of data. The user should experience a unified view of the database. Second, deciding on the arrangement (division) of data in a distributed database. This division of data is contingent on the type of system and its underlying characteristics. For relational systems, the data would be in a normalized form and it is imperative to ensure

ACID guarantees. The relation between the schemas and the relation between the tables must be taken into consideration before deciding on the fragmentation strategy. In case of most NoSQL systems, the data would be in a denormalized form. Considering the above-mentioned factors, fragmentation can be broadly categorized into three schemes: sharding (horizontal fragmentation) or row-based fragmentation, column-based fragmentation (vertical fragmentation) and Mixed (hybrid fragmentation). In this section, an insight into different schemes of fragmentation used in SQL and NoSQL systems is presented. Section 3.5.1 discusses about different schemes in fragmentation. Section 3.5.2 gives details on fragmentation in SQL systems. Finally, in section 3.5.3 an insight into fragmentation in NoSQL systems is apprised.

3.5.1 Fragmentation schemes

The complexity involved in reconstructing the data that is fragmented across multiples sites, as requested by the user of the application, majorly influences the decision to choose a fragmentation strategy. Different fragmentation strategies are discussed in the following sections.

3.5.1.1 Sharding

Sharding or row-based fragmentation, also known as horizontal fragmentation, divides the table in a database horizontally. Each divided fragment of the database is a subset of the original table. This scheme divides the table rows into subsets of data that can be stored in different sites.

3.5.1.2 Column-based fragmentation

Column-based fragmentation, popularly referred to as vertical fragmentation, is a column-wise division of a table in a database. In some cases, not all the attributes

(columns) of a table would be required to be stored together in a specific location. Attributes (columns) that are requested together are stored in the vertical fragment. However, it is necessary to include the same key attribute for each vertical fragment to facilitate reconstruction of the original data table.

3.5.1.3 Mixed

Mixed or hybrid fragmentation, is achieved by applying a combination of horizontal and vertical schemes of fragmentation [6]. In mixed fragmentation, the data is disintegrated into comparatively more number of fragments (subset of the subsets) and the cost involved in reconstructing the data into its original form is very high.

3.5.2 Fragmentation in SQL systems

In Relational systems, the database is usually in a normalized form; hence the data is stored into multiple tables. Consistency is maintained by implementing database constraints. To explain fragmentation in SQL systems, consider the example in Table 3.2. Table 3.2 is a country table with attributes providing different information about each country.

Table 3.2. Country details table

Country	Population (in millions)	Capital	Country Code
Australia	23.59	Canberra	61
Japan	127.06	Tokyo	81
England	53.5	London	44
Italy	61.07	Rome	39
Portugal	10.58	Lisbon	351
USA	318.9	Washington D.C	1

Horizontal fragmentation involves breaking the table into a group of rows. For example, the data in Table 3.2 can be divided horizontally based on continent wise location of each country. In that case, the table can be divided into three fragments. In another scenario, the table can be divided based on the population of each country for which there will be two fragments (above 100 million and below 100 million). Each fragment can be placed on a different site. MySQL [28], a relational system, implements horizontal fragmentation by dividing the rows of a table and distributing the fragments across multiple directories and disks.

In vertical fragmentation, the table is divided by the column attributes. The divided fragments must have a common key attribute to facilitate the reconstruction of data. To illustrate with an example, the Table 3.2 can be split into two fragments population and capital. On each of the fragments, it is necessary to include either the country name or the country code.

3.5.3 Fragmentation in NoSQL systems

Most NoSQL systems are designed to handle large volumes of concurrent requests. The scale of data handled by these systems is also very high. For instance, Facebook handles 600 terabytes (10^{12} bytes) of newly generated data everyday [31]. Format of data and modeling of data is different for each system. NoSQL systems mostly follow a schema-less or a partial schema approach. Hence, vertical fragmentation is not prevalent in NoSQL systems. A form of vertical fragmentation is available in column-based NoSQL systems, where column families of the same objects are stored in separate files.

On the contrary, for a very large distributed NoSQL systems, horizontal fragmentation must be implemented to achieve better performance (scalability, availability etc.). MongoDB [22], a document-based NoSQL system, stores data as a collection

of documents. The documents in a collection (a collection in MongoDB is somewhat similar to a table in SQL systems) are divided and stored at different sites. The performance of the system is improved by processing only the documents stored at particular site. Horizontal fragments are called shards in NoSQL systems and horizontal fragmentation is known as sharding. In MongoDB, partitioning of a collection into shards can be achieved in two ways: partition by range and partition using hash function. Voldemort [32], an open source NoSQL system, achieves horizontal partitioning by implementing a variation of the data distribution algorithm known as consistent hashing. Column-based data stores, like Bigtable and Hbase, distributes the data horizontally based on the row range (a range of key values for a set of rows) for a table [9].

3.6 Query Languages

The process involved in fetching data from a database, commonly referred to as query processing, is given prime importance when choosing a database. The ability of the system to translate the high-level database queries into instructions that can be processed by the low-level file system directly influences the performance of the application and the productivity of the user. The number of requests to the disks, and the time taken to process the request to access the data stored in the disks, directly affects the cost of executing the query. The complexity involved in implementing a query processing strategy for a distributed database, due to sharding, replication, and network latency, is higher when compared to a centralized database [5]. Ergo, it is necessary to understand the trade-offs in implementing an efficient querying strategy versus a bad strategy.

The magnitude of successes reached by relational databases is predominantly due to the presence of a non-procedural language, known as Structured Query Lan-

guage (SQL). SQL provides a query language standard that has been implemented on all major relational databases. Although NoSQL databases have found presence on several web applications, one of the major downside is the lack of a standard query language. Most NoSQL databases have a distinct process for retrieving the stored data. This section will highlight the query languages in both types of systems and the impact of having a standard query language.

3.6.1 Query languages in relational databases

SQL, also pronounced as "sequel", is the most commonly used querying language for relational databases. Donald D. Chamberlin [33], to ease the process of data retrieval through simple statements that resembles english sentences, introduced the language as Structured English Query Language (SEQUEL) in 1974. SQL gained popularity for its simple syntax allowing the users to comprehend the language easily and was recognized as a standard querying language for relational systems. SQL is somewhat based on a formal language known as tuple relational calculus.

Relational-query languages, like SQL, provide a platform for the users of the database to specify what data to fetch from the database. The process involved in fetching the data, the internal parsing of the query and the execution plan is transparent to the user. Hence, the optimization of the query processing is done automatically thus improving the efficiency of the system. SQL comes with a comprehensive set of functions to direct the database management activities, example, Data Definition Language (DDL) and Data Manipulation Language (DML).

SQL is capable of processing queries in a distributed relational database. This processing is done from a centralized location in stages. First, the query processor maps the input from the user (usually in the form of a SQL query) to an algebraic query. Second, this stage translates the distributed query accessing the data from

multiple locations into a localized set of operators that can be executed on local fragments [5]. Finally, query optimizer selects the ideal strategy for optimizing the costs (i.e. I/O costs, CPU costs, communication costs).

3.6.2 Query languages in NoSQL databases

NoSQL databases do not have a standard query language similar to SQL. This is mainly due to heterogeneity in the type of data handled by these databases. Unlike relational systems, the choice of NoSQL database to be implemented is exclusively dependent on the kind of application and type of data generated. Consequently, every NoSQL database develops an in-house language to augment its query processing needs. Many NoSQL systems do not have a high-level query language at all but rather an API (Application Program Interface) of operations that can be used by the application programmers. These operations are often called CRUD operations (Create, Read, Update, Delete).

The ramifications of having different query processing techniques can be construed as both positive and negative. On the negative side, this could create a situation where competition among different NoSQL providers would result in less profits. In addition to that, the choice of a NoSQL system creates a long-term dependence on the provider with no feasible alternative system to switch. On the brighter side, with no conventionally perceived limitations on designing a query language, there is a possibility of a path-breaking design. Different querying languages for different systems are as follows:

3.6.2.1 Document stores

MongoDB stores the data in the form of documents and it uses a variation of JSON (JavaScript Object Notation) known a BSON (Binary JSON) for efficient

storage. MongoDB provides a mongo shell for querying. The users can query on the database using MongoDB query language [22].

JSONiq [23], also known as the SQL of NoSQL, is a highly optimizable language designed to query on databases using JSON format. This high-level language facilitates the execution of complex queries on JSON-based NoSQL databases. JSONiq currently finds its application in MongoDB and CouchDB.

Unstructured query language (UnQL), a superset of SQL, is similar to SQL except for working on documents and collections instead of rows and tables. UnQL is best suited to query on JSON format. UnQL finds its application in CouchDB [34].

3.6.2.2 Column stores

Hbase, a column store database, provides an Hbase shell to access the database. This shell can be used for basic querying on the database. Apache Hive [35], a data warehouse software, provides querying facilities over large datasets on Hbase through HiveQL, an Sql-like language.

Cassandra Query Language (CQL) is the primary query language for Cassandra database. It provides the option to access the database using "cqlsh", a basic CQL shell [25].

CHAPTER 4

CATEGORIES OF NOSQL SYSTEMS

The materialization of numerous NoSQL systems, to satisfy the need for Web 2.0, can be attributed to the more flexible data models. A simpler approach to data modeling, when compared to relational systems, is due to the self-describing model of the stored data, so no prior schema is required. For a system to be labeled as NoSQL, it has to be non-relational, distributed and horizontally scalable [11]. Except for the above-mentioned characteristics there is a significant difference in the characteristics among NoSQL database. Each system defines its own concurrency control methods, consistency models, data handling methods, and storage structures. Hence, accurate categorization of these systems is difficult. There is no formal taxonomy to categorize the NoSQL systems, which is the main reason for the existence of different approaches. This chapter presents an overview of generally agreed upon categories of NoSQL systems.

4.1 Document-store

This category of systems store data in the form of semi-structured sets for key-value pairs, known as documents (the unit of storage), encoded into standard format such as XML, JSON (JavaScript Object Notation) and BSON (Binary JSON). All these formats are self-describing, meaning that a set of <attribute name, attribute value> pairs are stored together in a document. A document is treated as a unit similar to an object or a record, except for the fact that it is self-describing. A document is processed without breaking down into atomic key-value pairs; this allows

complex querying like aggregation on the database. These databases are indexed on the document id, which is the primary identifier. Secondary indexing is also supported. Data is stored in disks to ensure persistency. Indexes and frequently accessed data are stored in memory caches to facilitate quicker access [27]. These data stores support a flexible schema-less data model, which is suitable for agile method of application development. Each document can be independent of the other documents in a collection, hence the number and names of attributes in each document is dynamic. The pattern-matching search (both by key and by value), equivalent to the "like" query in SQL, is possible in document stores.

Document stores are suitable for applications that require querying and managing large sets of documents like emails, archival data, consumer data etc. Most prominent databases classified in this category are MongoDB and CouchDB.

4.1.1 MongoDB

MongoDB [22], a portable NoSQL document store database, was created in 2007 by 10gen (now MongoDB Inc.). The motivation behind the development of MongoDB was to have a loosely coupled platform independent database technology. Thus, MongoDB was developed in C++ as an open source, JSON styled, scalable, platform as a service (PaaS) cloud stack. The agility of JSON to provide a standard space, language independent way to store object style data makes application development easier.

MongoDB has a dynamic (schema-less) data model that makes iterating and prototyping in the database simpler. The data in MongoDB is stored as BSON documents, a binary form of simple data structures and arrays, and are grouped into a collection. Each document has a unique identifier and can be indexed (B-tree index) on one or more attributes for efficient access. Single field index, Multi-key

index, compound index, geospatial indexes, text index, and hash index are the index types provided in this database.

Sharding of the documents in a collection is done by using a shard key. The shard key is a partitioning attribute that is present in every record and has a unique value. Two sharding methods are supported - hash partitioning and range partitioning [6]. The database also provides a spectrum of consistency levels and can be tailored to fit the application. For ensuring high availability and reliability of the database, a variation of centralized replication (section 3.4.2.2), known as replica set, is implemented.

MongoDB has a global presence and finds its implementation in a wide range of industries including telecommunications, healthcare, media, hospitality and finance. Verizon, Squarespace, Forbes, New York Times are some of the companies that use this system.

4.1.2 CouchDB

CouchDB [30], developed by Damien Katz in April 2005, is an open source, self contained, web embracing document store inspired by Lotus notes. "Cluster of Unreliable Commodity hardware", aka Couch, aimed to provide a scalable, highly available, and a reliable database when hosted on unreliable hardware. The system prioritized data safety because of which Erlang OTP platform was used for system development. The data is stored as uniquely named documents in a standard JSON format and can be accessed using a RESTful HTTP/JSON API which suits the present-day mobile and web applications. The CRUD (Create, Read, Update, Delete) operations are faster due the underlying B-tree storage structure.

A method similar to hash partitioning is used for horizontal partitioning. A proxy-based clustering framework CouchDB lounge does the partitioning for CouchDB

[36]. The system uses a multiversion concurrency control technique to process concurrent writes. The system supports both centralized and decentralized replication strategy. Incremental replication ensures periodic synchronization of nodes in the cluster. As a result, autonomous functioning of nodes in the cluster (shared nothing architecture) is possible. A notable user of CouchDB can be found in BBC (British Broadcasting Corporation).

4.2 Key-value stores

This category of data management systems uses a storage structure similar to HashMap. Key-value stores prioritize low read-write latency and availability over consistency. This is achieved by using a simple but efficient data structure that accesses data in constant time. An individual record in the database is a key-value pair. The key is a uniquely indexed identifier that is assigned to the value of the record. The value of a record can be stored in different formats like sets, lists, arrays, bitmaps and JSON. This facilitates the storage of unstructured and semi-structured data. These systems do not support querying capabilities on multiple attributes of a data item but may be equipped to handle CRUD operations on a single attribute of the data item [27].

Key-value stores favors persistency in data storage. Replication, sharding, concurrency control are some of the features supported in this category. These systems are ideal for e-commerce applications that require quicker retrieval of data. DynamoDB, Voldemort and Redis are the popular databases in this category.

4.2.1 DynamoDB

Amazon introduced Dynamo [10] in 2007 as a highly available distributed data storage system with seamless scalability. This database set the precedence for the

emergence of several key-value NoSQL systems. Initially it was developed as an in-house storage platform for Amazon's e-commerce applications. In 2012, the system was introduced as DynamoDB through Amazon Web Services, the company's cloud service platform. The intent was to create a highly reliable application that creates an always-accessible experience to its customers. This was achieved by efficiently managing the tradeoff between data consistency and system performance.

DynamoDB is a schema-less data storage that stores data in tables. A table in the database is a collection of independent items. A set of <attributes> = <values> entries forms an item. Individual attributes in an item is a key-value pair. A wide range of data types for the attributes including set, hash, JSON are supported. An attribute or a set of attributes is selected as a primary key and must be specified while creating a table. The primary key indexes the items in a table. In addition to that, secondary indexes can also be created on the items. This makes querying the data using an alternate key possible. The system provides a set of operations for manipulating the data. The feature of DynamoDB to allow search by non-indexed attributes is similar to a full table scan. The items can be considered to be similar to documents.

One of the key features of DynamoDB is its ability to scale, which is achieved by implementing a consistent hashing technique for dynamic sharding of data. Depending on the type of application, the level of concurrency can be set. DynamoDB supports both strict and eventual consistency. Continuous up-time of the system is ensured by replicating the data on multiple nodes. The system supports a centralized replication technique. This commercially licensed cloud storage NoSQL database is implemented in many applications. BMW, infraware, mlbam are some of the customers of DynamoDB [37].

4.2.2 Redis

Redis [26] started as a project, in 2009 by Salvatore Sanfilippo, for making real-time web analytics application more efficient. This schema-less, C-based, key-value memory store supports a basic set of operations like insert, search, delete etc. and data types like lists, sets, sorted sets, hashes etc. The data storage is volatile since the data resides in RAM (Random Access Memory). Optionally, data persistency can be achieved by enabling periodic snapshots of the data. This in-memory data structure supports polyglot persistence [26].

Asynchronous centralized replication is supported and the system provides an option to enable horizontal partitioning. Range partitioning and hash partitioning are the ways in which partitioning can be achieved in this system. To facilitate concurrent access to shared resources, the system implements a distributed lock manager. Twitter, Github, Snapchat, StackOverflow, Flickr are some of the well-known companies using Redis for data storage.

4.3 Tabular stores

Tabular stores, popularly known as column based or wide column stores, are modeled after Google's BigTable [9] which is "a sparse, distributed, persistent multi-dimensional sorted map". The purpose of the systems in this category is to resemble a database that scales across thousands of servers, has a wide range of applicability, and administers very large data (in petabytes). The unit of storage is a map, which is a collection of key-value pairs. The keys in column stores are multidimensional which is usually an uninterrupted array of strings combining the rowkey, column, and a timestamp. This multidimensional lexicographically ordered key is one of the

notable differences between column stores and key-value stores. Bigtable uses Google File System (GFS) as its underlying storage system.

This category of systems implements a wide range of features like replication, horizontal partitioning, concurrency control etc. to support very large scale distributed applications. These systems, for its versioning capabilities, suits applications that have heavy write operations and applications that handle data analytics.

The emergence of tabular stores is because of Google's BigTable and no other wide storage data store was able to match its scalability performance. Hbase and Hypertable are the NoSQL databases belonging to this category and they strictly adhere to the principles of BigTable.

4.3.1 Hbase

Apache Hbase [24], patterned after Google's BigTable, is a Hadoop based open source system running on top of Hadoop Distributed File System (HDFS). The urge to have a system that supports natural language processing on very large amounts of data is the motivation behind the development of this Java based tabular store. Hbase is being used in many applications for its fault tolerant distributed data storage (using HDFS), flexible data model and MapReduce functions.

Hbase has a schema-less data model. Data in Hbase is stored in the form of tables. Each table has multiple row keys that are indexed, lexicographically ordered and multidimensional. In addition to that, a column family or a number of column families must be defined when creating a table. A column family houses a logically grouped set of column qualifiers. Column qualifiers do not need any prior declaration and can be defined dynamically, thus providing an efficient data structure to store large quantities of sparse data. The basic data item is stored under a column qualifier

and is accessed by combination of table name, row key, column family, and column qualifier [6].

Hbase does not modify a basic item but rather writes a different version of the data item. The system is provisioned to store different versions of a data item (synonymous to attribute versioning in temporal databases [6]). Each version of the data item is assigned a timestamp. By default, only the last three versions of a data item are stored in the database and can be configured to a specific number of versions. This feature of Hbase results in high write performance of the database.

Hbase supports sharding of data by Ordered Partitioning. The basic shard in Hbase is a region. The subset of the rows of a column family, ordered by the row key, are assigned to a region. Each region is assigned to only one regionserver, the slave node of a Hbase cluster. Hmaster, the master node of Hbase, is responsible for administrative activities, like monitoring, coordinating and sharding of data. Cluster replication in Hbase is asynchronous and the nodes in the cluster are replicated by using write-ahead log (WAL). Hbase aims to become eventually consistent through replication. Hbase is widely used in social media applications and some of the major applications are Facebook, StumbleUpon, and Yahoo.

4.4 Graph stores

This category of NoSQL systems concentrates on the association (relationship) that connects the data. The data in these systems resembles a relational graph with interconnected key-value pairs. A collection of vertices and edges forms a graph with individual nodes and edges tagged with the information about the type of entity and the nature (label) of the connection [6]. These databases target making data representation more user-friendly by a visual interface. Graph databases are best suited for social networking websites, building recommendation systems, and pattern

mining. Neo4j is the popular database in this category. Other graph databases are InfoGrid, AllegroGraph etc.

4.5 Heterogeneous data stores

This category of data stores are designed by combining the best characteristics of different categories of NoSQL systems with a motive to create a hybrid database with improved performance. There are several databases that can be categorized as heterogeneous and is discussed in the below subsections.

4.5.1 DynamoDB

DynamoDB is predominantly a key-value database. The system makes it possible to perform scan operation by a full table scan or scan by secondary indexing. In addition to that, range querying is possible in DynamoDB. This feature of DynamoDB is characteristic of document store databases. Hence, in some categorization, this database is classified as hybrid.

4.5.2 OrientDB

OrientDB [38] is a distributed, graph based NoSQL system with the characteristics of object oriented databases, document stores, and key-value stores [11]. This Java based system is ACID compliant. The system supports decentralized replication and horizontal partitioning.

4.5.3 Cassandra

Cassandra [25] is the most popular data store that can be categorized as an hybrid NoSQL system. Jeff Hammerbacher from Amazon and Prashant Malik from Facebook designed the system. Cassandra was described as a marriage of Dynamo

and BigTable [39]. The system was developed for the requirement to handle arbitrary reads and writes and manage huge volumes of data.

The underlying architecture of Cassandra is congruent with Google's BigTable. However, the systems are not without differences. Cassandra is a decentralized system, which implements a multiversion concurrency control technique instead of locking. As a result, eventual consistency is achieved. These two features were adapted directly from DynamoDB thus taking the Cassandra closer to key-value store.

Cassandra uses Murmur3Partitioner [25] to partition the data based on the row key. The partitioner creates a 64-bit hash value of the shard key. Secondary indexing is supported in Cassandra. The data model is similar to that of Hbase with few different terminologies. Database in Cassandra is known as keyspace and an additional level of organizing the columns known as super columns is provided.

Cassandra uses Cassandra Query Language (CQL), an SQL-like language, for querying. Cassandra is the choice of data store for many organizations. Most notable companies are Facebook, Twitter, and Reddit.

4.6 Other NoSQL databases

Based on the definition for NoSQL systems, the category of systems that cannot be classified into any of the above mentioned categories are grouped as other databases.

4.6.1 Object databases

Databases that follow the standard set by Object Data Management Group (ODMG) are categorized as object databases. Object Model, object definition language (ODL), and object query language (OQL) are the parts of the standard [6].

Versant Object database and Gemstone systems are some of the databases belonging to this category.

4.6.2 XML databases

These databases are used for storing and manipulating unstructured data. EXtensible Markup Language (XML) is a standard data format that uses a hierarchical data model for persistent data storage. These systems can be queried on by using XML/SQL, Xpath and Xquery [6]. BaseX, Sedna, eXist are some of the databases that can be classified in this category [11].

CHAPTER 5

EXPERIMENTAL SETUP

5.1 Implementation details

The experiments were performed on a cluster consisting of fourteen servers. The details of the nodes are shown in Table 5.1. The Cluster is made of 14 servers connected through 1 Gigabit Ethernet Switch where each server is managed by Centos. A single server consists of Xeon CPU, 4 GB RAM, and 1.4 TB hard disk. One of the node acts as master, which consists of 2.7 TB hard disk. For the experiments of Hbase, Hadoop 2.2.0 was used. CPU utilization and performance monitoring of each node was done through Ganglia, an open source monitoring tool suitable for a distributed environment.

Table 5.1. System Configuration

Node	Disk	Memory	Details
hadoop	2.7 TB	4 GB	Hadoop namenode, Hbase master and Zookeeper
compute-0-2	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-3	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-5	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-7	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-8	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-9	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-10	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-12	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-14	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-15	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-16	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-17	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver
compute-0-19	1.4 TB	4 GB	Hadoop datanode and Hbase regionserver

5.2 Database configurations

For this evaluation, three databases, MongoDB, Cassandra and Hbase, were considered. The following subsections give detailed information about the configuration of each of the three NoSQL databases.

5.2.1 MongoDB

MongoDB [22] version 3.0.6 was used for this benchmarking. The software was installed via x86_64 tarball for linux platforms. Every sharded cluster for a MongoDB must have a config server, which stores the metadata of the cluster. In this experiment, one config server instance was created using the mongod instance. A mongod is a process used for starting standalone MongoDB instances and also used for starting config server processes [22]. Standalone instances of MongoDB were created on each node of the cluster and were added into the cluster using the command "sh.addShard()".

For sharding of data, a range partitioning technique was implemented and the maximum size of each sharded set of data, known as a *chunk*, was set to 64 MB. The collection was sharded based on the indexed ObjectId field, which is unique for each document. The data for this benchmarking was not replicated.

5.2.2 Cassandra

Cassandra [25], version 2.0.16 was used for this benchmarking. The software was installed via the tarball available for download from the Cassandra website. Cassandra uses a Murmur3Partitioner for sharding of data [25]. Murmur3Partitioner is the default partitioner for Cassandra (in recent releases) and it improves the performance of the system by using a quicker hashing function (MurmurHash) for uniform distribution of data. It implements a technique called Consistent Hashing. Each

node in the cluster is assigned a token. The token is generated to suit the partitioner, which uses a maximum possible hash valu from -2^{63} to +2^{63}-1 [25].

Key caching was enabled by default and replication factor was set to 1 (no replication). The distribution of data among the nodes in the cluster and status of each node in the cluster was monitored using nodetool, a Cassandra utility. Table 5.2 provides important configuration details in Cassandra.

Table 5.2. Cassandra Configuration Details

Parameters	Values
initial_token	Token generated based on the Mumur3Partioner hash values
partitioner	Default (org.apache.cassandra.dht.Murmur3Partitioner)
key_cache_size_in_mb	Default (100 MB)
commitlog_sync	Default (periodic)
seed provider	class_name:org.apache.cassandra.locator.SimpleSeedProvider seeds : <IP_address_of_the_provider>
concurrent_reads	Default (32)
concurrent_writes	Default (32)

5.2.3 Hbase

Hbase [24], version 1.1.0, was installed and integrated with hadoop 2.2.0 for efficient distributed storage. Hbase follows master-slave architecture. The master node in Hbase is called a namenode and the slave nodes are called regionservers. The first node was automatically assigned as the master node on installation and the rest of the nodes configured as regionservers. Apache Zookeeper [24], a clone of chubby lock service [9], is an open source coordination service for efficient synchronization, replication, and distribution of data in Hbase, was configured on the master node.

The default auto-sharding feature of Hbase was used and the data was not replicated. Configuration details of Hbase are presented in Table 5.3.

Table 5.3. Hbase Configuration Details

Parameters	Values	Configuration file
hbase.cluster.distributed	true	hbase-site.xml
hbase.rootdir	hdfs://<master>:<port_number>/hbase	hbase-site.xml
hbase.zookeeper.quorum	master	hbase-site.xml
fs.default.name	hdfs://<master>	core-site.xml
dfs.name.dir	<path_of_the_mount>/name	hdfs-site.xml
dfs.data.dir	<path_of_the_mount>/data	hdfs-site.xml

5.3 Benchmarking Tool

For experimentation and analysis, a standard, flexible and customizable benchmarking platform was required. Yahoo! Cloud Service Benchmark [3] (YCSB), an open source, extensible, Java based client satisfied the requirements and was the ideal choice for benchmarking NoSQL databases. YCSB is one of the most popular NoSQL benchmarking tools available and is cited in more than 250 research works. In this section, the architecture of the benchmarking tool is explained. Workload properties and the workloads defined for this experiment are discussed.

5.3.1 Architecture

In 2010, Yahoo! Research presented a paper on Benchmarking Cloud Serving Systems at the 1st ACM Symposium on Cloud Computing [3]. The primary focus was to create a benchmarking tool that simplifies the process of benchmarking different systems that can be deployed on cloud platforms. Figure 5.1, adapted from [3], shows the architecture of the YCSB client.

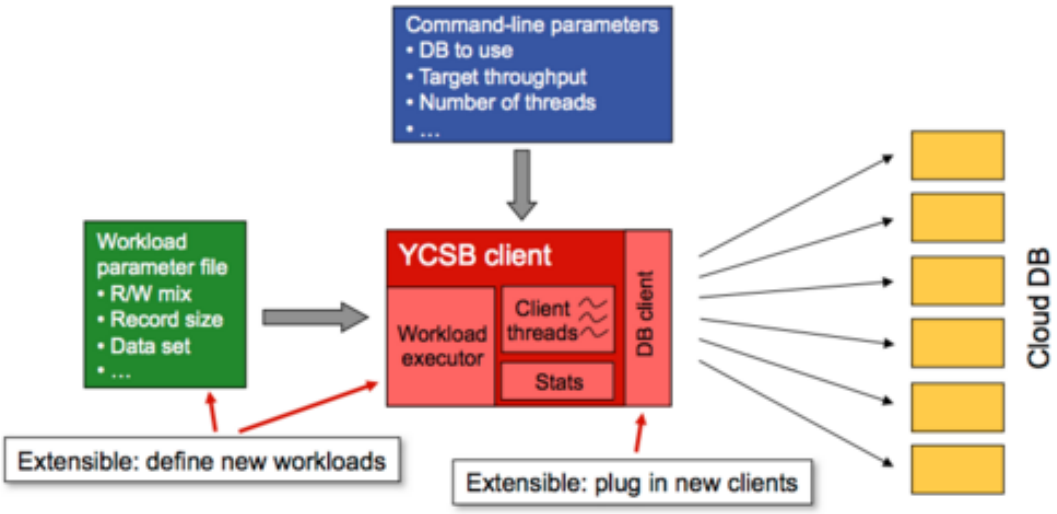

Figure 5.1. Architecture of YCSB client, adapted from [3].

The YCSB client has a workload executor. The role of the workload executor is to handle multiple client threads, execute the basic CRUD operations by reading the workload file, and connecting to different databases using database interface layer. The benchmarking tool can be customized to add new databases. The database interface layer provides a base class, which can be extended to add methods to connect to the database and define the operations to be performed on the database. The source code of the YCSB client is downloadable and can be built using Maven, an open source apache build manager. For this experimentation, YCSB version-0.1.4 was used.

5.3.2 Workload properties

There are different workload properties in YCSB client that can be customized, configured, or enabled to define the benchmarking. The core workload properties that were used in this experiment are mentioned in Table 5.4, adapted from [3].

Table 5.4. Workload Properties

Properties	Description	Values
fieldcount	Number of fields in a record	10 and 20
fieldlength	length of each field (in bytes)	100 and 200
requestdistribution	type of distribution for selecting a record (uniform,zipfian,latest)	Hbase:zipfian MongoDB:zipfian Cassandra:zipfian
maxscanlength	range of records to scan	100
scanlengthdistribution	type of distribution for scanning a record (uniform,zipfian,latest)	Hbase:zipfian MongoDB:zipfian Cassandra:uniform
insertorder	key or hashed	hashed

Table 5.5. Workloads Definition

Workloads	Parameters	Applications
50% Read - 50% Write	readproportion=0.5 updateproportion=0.5	Session stores
100% Read	readproportion=1	Hadoop
100% Blind Write	updateproportion=1	Logging
100% Read-Modify-Write	readmodifywriteproportion=1	User profile updates
100% Scan	scanproportion=1	Search operations

5.3.3 Workloads definition

Five different workloads were defined to execute different CRUD operations on the databases. For a comprehensive analysis, some of the five workloads were customized. In Table 5.5, the defined workloads are listed.

5.4 Test cases

In order to analyze the behavior of the databases comprehensively, several test cases were defined by varying the size of the database, the number of operations performed, and the cluster size. In the following subsections, the scope of this experiment is defined. In section 5.4.1, details about the data set and the size of the database under consideration are presented. Section 5.4.2 provides the reasoning behind defining different operation counts. In section 5.4.3, the details about the different cluster sizes considered for this benchmarking are presented. Finally, a comprehensive view of all the test cases is presented.

5.4.1 Size of the database

NoSQL databases are designed for the purpose of handling large sets of data. Therefore, it is essential to analyze the behavior of each database on its ability to effectively shard and query the data. This behavior can be analyzed by executing the different workload operations on different database sizes. In this benchmark testing, four different dataset sizes were considered. Table 5.6 defines the different data set sizes.

The data generated is synthetic data, which is a set of random characters indexed by a primary identifier. A snapshot of the data from Cassandra can be viewed in Figure 5.2

Table 5.6. Datasets

Record count	Record size (for each record)	Parameters	Total size (of the dataset)
1 Million	1 KB	fieldcount=10 fieldlength=100 (in bytes)	1 GB
1 Million	4 KB	fieldcount=20 fieldlength=200 (in bytes)	4 GB
10 Million	1 KB	fieldcount=10 fieldlength=100 (in bytes)	10 GB
10 Million	4 KB	fieldcount=20 fieldlength=200 (in bytes)	40 GB

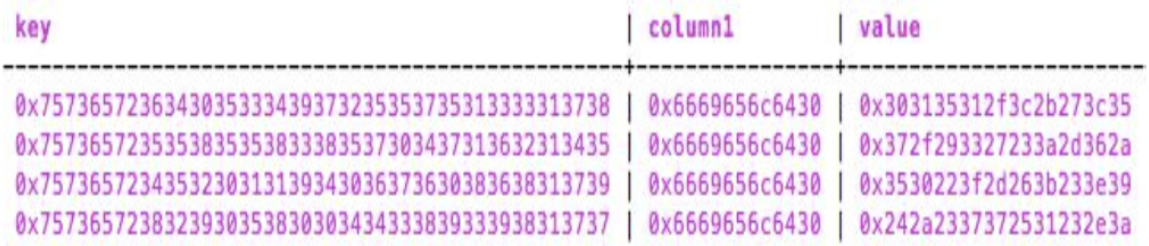

Figure 5.2. Sample data generated by YCSB in Cassandra.

5.4.2 Operation count

The number of operations executed (Read, Write, Scan etc.) on the database was increased proportionally to analyze the performance of the database. By varying this property, the increase or decrease in throughput (defined in Figure 5.3) with respect to the increase in operation count could be captured. Additionally, the mean of the throughputs would assist in smoothing noisy results. The scope of the operation count and definition of throughput is explained in Figure 5.3.

5.4.3 Cluster size

In addition to handling large amounts of data, one of the main factors that influence the reputation of a NoSQL database is its ability to scale across multiple

Operation Count (in thousands)
10K
20K
40K
80K

$$\text{Throughput (ops/sec)} = \frac{\text{Total Time Taken to execute the operations}}{\text{Number of Operations (operation count)}}$$

Figure 5.3. Scope of the operation count and definition of throughput.

nodes in a cluster commonly referred to as horizontal scalability or sharding. The sharding feature of NoSQL databases can be examined by increasing the number of nodes in the cluster. In order to analyze the pattern in performance of the database and to find the ideal number of cluster nodes required for a given dataset size, the experiments need to be executed on a uniformly distributed spectrum of cluster sizes. Hence, six different cluster sizes were considered. Figure 5.4 provides a pictorial representation of the different cluster sizes.

5.4.4 Comprehensive view

A comprehensive view of the scope of the experiments can be viewed in Figure 5.5. In total, 480 test cases were executed on each database with four trials for each test case. The results of each test case will be averaged (mean of four trials) to avoid anomalies.

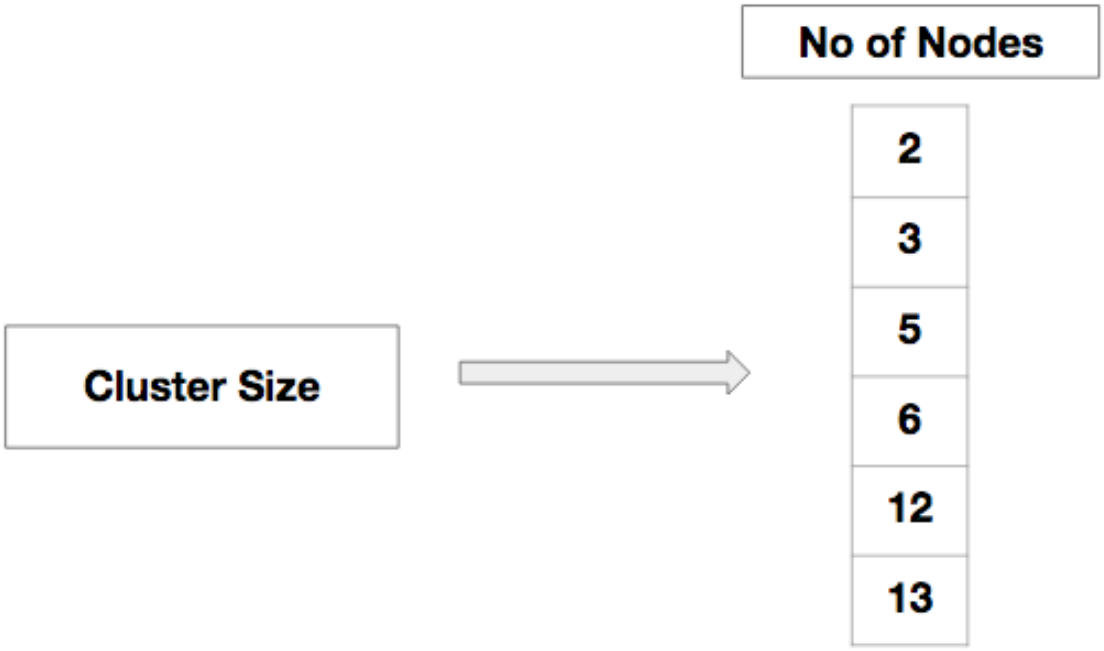

Figure 5.4. Scope of the cluster size.

Data Size
1 GB
4 GB
10 GB
40 GB

Operation Count (in thousands)
10K
20K
40K
80K

Work Loads
50% Read - 50% Write
100% Read
100% Write
100% Scan
100% Read-Modify-Write

No of Nodes in the Cluster
2
3
5
6
12
13

Figure 5.5. Comprehensive view of the scope of the experiments.

CHAPTER 6

RESULTS

In this chapter, the benchmark results of the three databases, MongoDB, Cassandra and Hbase are presented. The test cases defined in previous section (Section 5.4) were executed. For a given cluster size, the workloads were executed on different dataset sizes and different operation counts. A shell script was created to extract the results from the log file. The extracted results were processed into a consolidated format to assist in the analysis. For a given database, the throughput (defined in Figure 5.3) of different dataset sizes is plotted against the cluster size. The benchmark results for individual databases will be presented in order and finally a performance comparison of different databases will be presented. In section 6.1, the performance result of MongoDB is presented. The performance result of Cassandra is evaluated in section 6.2. Apache Hbase's performance is analyzed in section 6.3. The chapter concludes with a comprehensive performance comparison of MongoDB vs. Cassandra vs. Hbase.

6.1 MongoDB

In this section, the results obtained during the benchmark testing of MongoDB are analyzed in detail. The result of each workload is presented individually.

6.1.1 50% Read - 50% Write

In this workload, the number of operations is equally split between read and blind write. In MongoDB, the data is horizontally partitioned into shards, known

as chunks. For this experimentation, the size of each chunk was limited to 64 MB. Data is assigned to a chunk until it reaches the limit and the chunk is flushed to a node in the cluster. Table 6.1 provides the calculation of number of chunks assigned to each node based on the size of the database. This table was used for the analysis to find the optimum number of chunks assigned to a node for which the database showed best results. It was found that MongoDB achieves best throughput when the number of chunks per node was approximately between 10 and 13. The result of this workload is plotted in Figure 6.1.

Table 6.1. MongoDB Chunk Distribution

Data set size	2 nodes	3 nodes	5 nodes	6 nodes	12 nodes	13 nodes
1 GB	8	5.3	3.2	2.6	1.3	1.2
4 GB	32	21	12.8	10.6	5.3	4.9
10 GB	80	53.3	32	26.6	13.3	12.3
40 GB	320	213.3	128	106.6	53.3	49.2

The performance of the database with 1 GB of data decreases with the increase in the number of nodes. This results show that the performance of the database is not guaranteed to always improve by adding more nodes to the cluster. The decrease in the throughput is attributed to the size of the data. Since the size of the database is small, the distribution of chunks in the cluster is more efficient with a small cluster. Hence, the database has a better throughput in a two-node cluster. As the size of the cluster increases, the additional overhead created due to the distribution of data in the cluster and increase in the network communication cost are the reasons for the decrease in throughput.

For 4 GB of data, the throughput forms a bell curve with the database reaching the best performance when the number of nodes in the cluster is 5 or 6. The reason

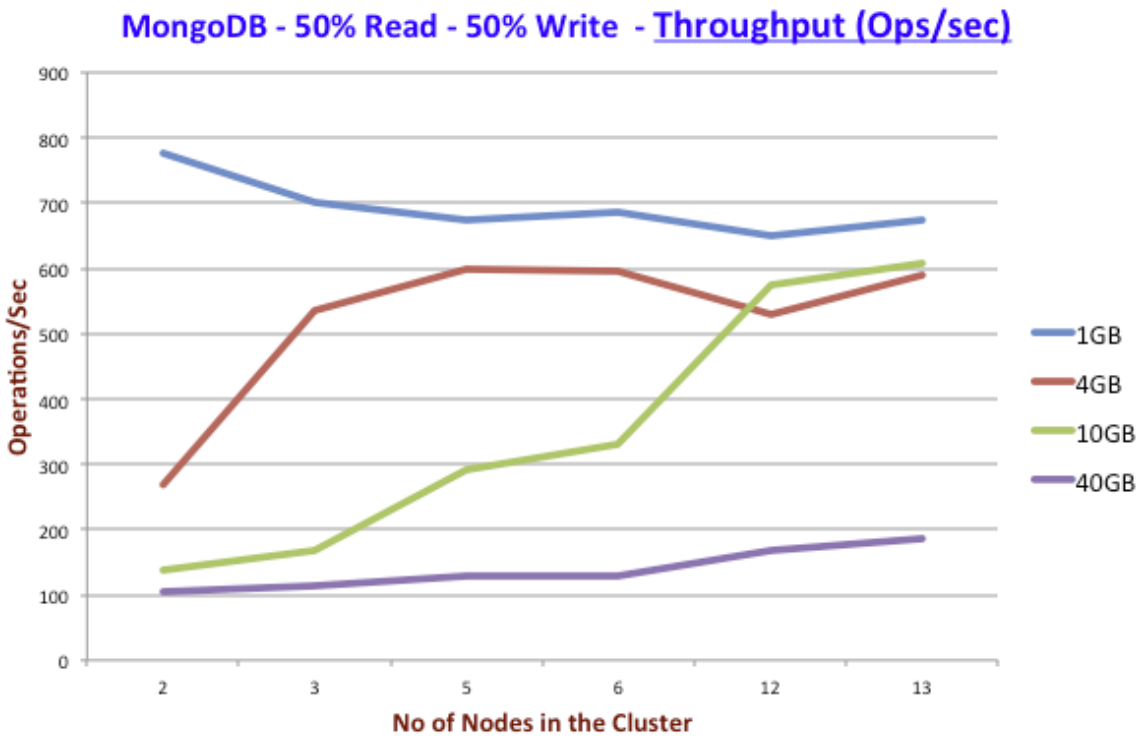

Figure 6.1. MongoDB : 50% Read - 50% Write.

for this behavior can be traced back to the sharding of data in MongoDB. The number of chunks assigned per node is high when the size of the cluster is small. Hence, a low throughput is because of the time taken to locate a record in the nodes. As the number of nodes increase, the distribution of chunks reaches an optimum limit at 5 or 6 nodes achieving high throughput. The throughput decreases at 12 nodes, which is due to over distribution of data.

For MongoDB with 10 GB and 40 GB of data, throughput increases with the increase in the number of nodes in the cluster. With 10 GB of data, the optimum throughput is obtained when the number of nodes in the cluster is between 13 and 14. However, the trend is likely to continue for a database with 40 GB of data and achieve the best performance when the number of nodes in the cluster is approximately 49.

6.1.2 100% Read

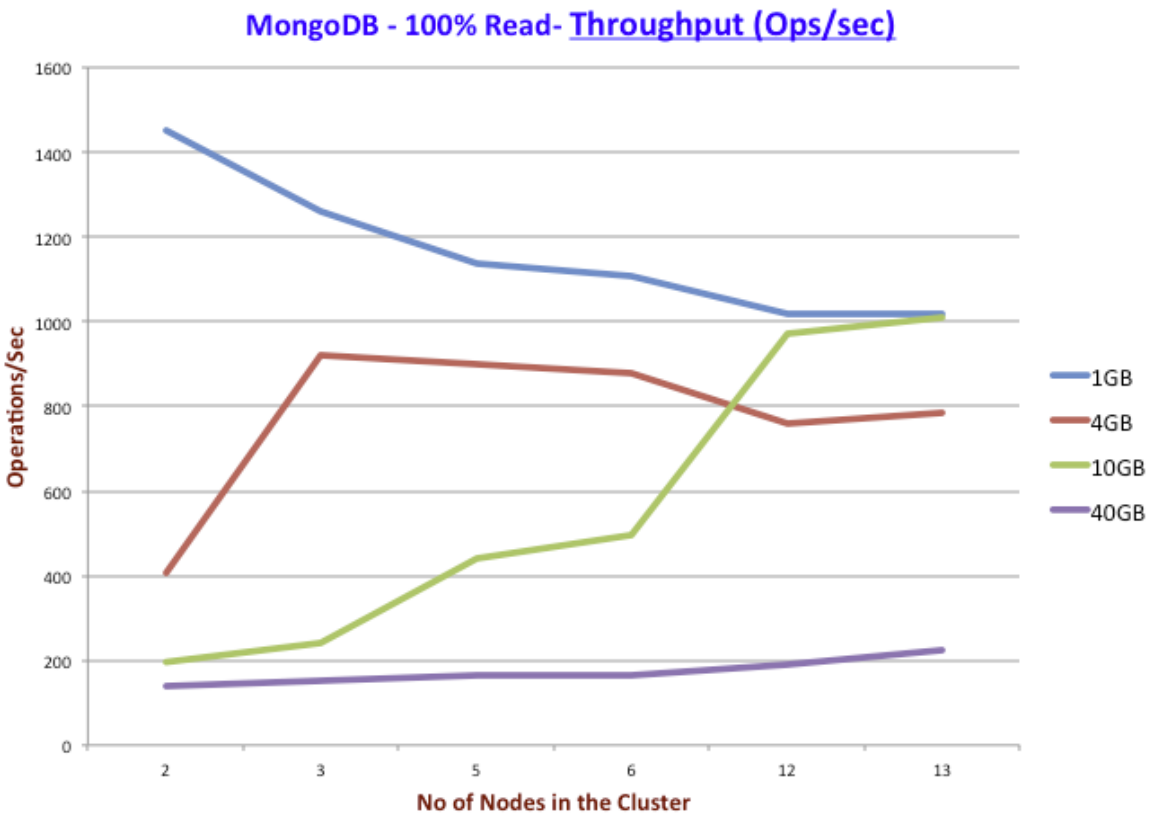

Figure 6.2. MongoDB : 100% Read.

This workload follows the performance pattern similar to a 50% Read - 50% Write workload as shown in Figure 6.2. The throughput decreases with increase in the number of nodes for 1 GB of data. A bell curve pattern for 4 GB of data and an increase in throughput with the increase in the number of nodes for 10 GB and 40 GB of data was observed. The same reasoning provided in section 6.1.1 is applicable here. However, the scale of the throughput for a 100% Read workload is much higher compared to a 50% Read - 50% Write workload. This is because of the structured storage format in MongoDB (section 4.1.1), which ensures efficient access to the data.

6.1.3 100% Blind Write

100% Blind Write, Figure 6.3, follows a similar pattern to 50% Read - 50% Write workload. However, the throughput stabilizes for 4 GB and 10 GB of data when the number of nodes in the cluster reaches 12. The scale of the throughput for 100% Blind Write is lower compared to 50% Read - 50% Write workload. In MongoDB, the time taken to locate the particular attribute within the document to write is more compared to reading a value. As a result, MongoDB performs much better for read operations compared to write operations.

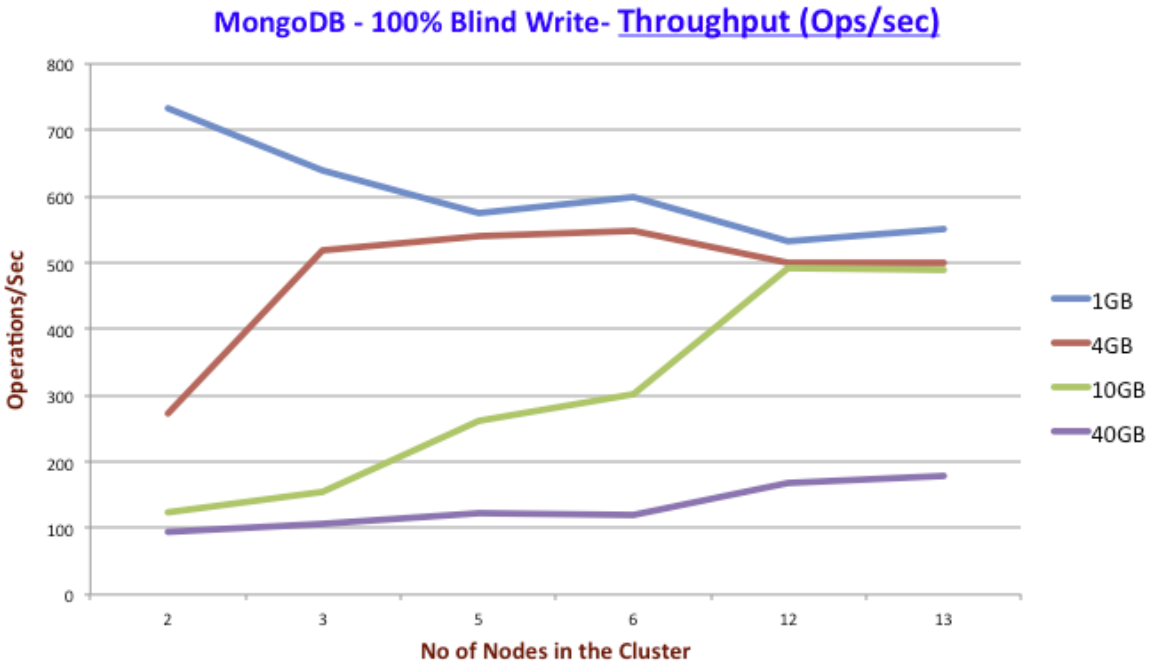

Figure 6.3. MongoDB : 100% Blind Write.

6.1.4 100% Read-Modify-Write

Read-Modify-Write workload is an actual update operation in the database, where the value is read, updated, and written back. This workload follows the same performance pattern as 50% Read - 50% Write workload but has low throughput compared to the above-mentioned workloads. The result of this workload is shown in Figure 6.4.

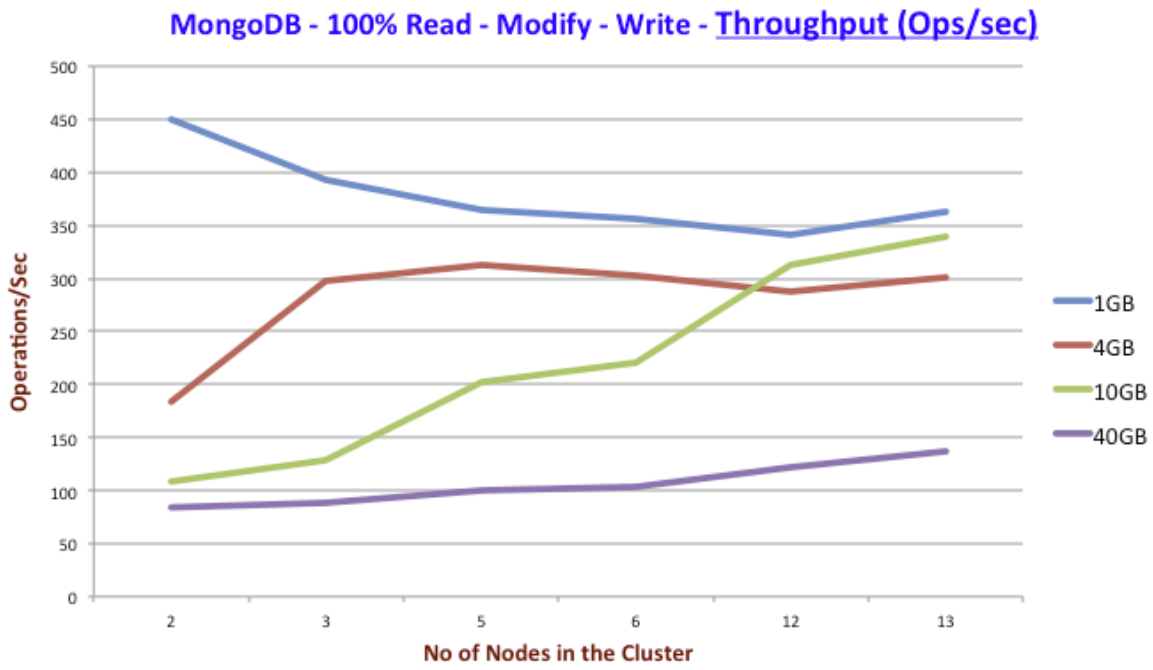

Figure 6.4. MongoDB : 100% Read-Modify-Write.

6.1.5 100% Scan

Each scan operation in this workload is performed over a range of 100 records. To illustrate further, in a scan operation 100 records in a consecutive range of rowIds is read. Hence, the execution time for scan operations is higher compared to other

workloads. The performance of this workload is shown in Figure 6.5. The performance of the database with 1 GB of data decreases exponentially. Once again, this behavior can be attributed to the size of database and the distribution of data. A scan operation is performed over a range of records, increasing the number of nodes in the cluster for a small database makes the range of records for a scan operation split across multiple chunks, which in turn affects the performance of the scan operations adversely. 4 GB of data has optimum performance at 3 nodes. The performance of the database improves slowly with the increase in the size of the cluster for 10 GB and 40 GB of data.

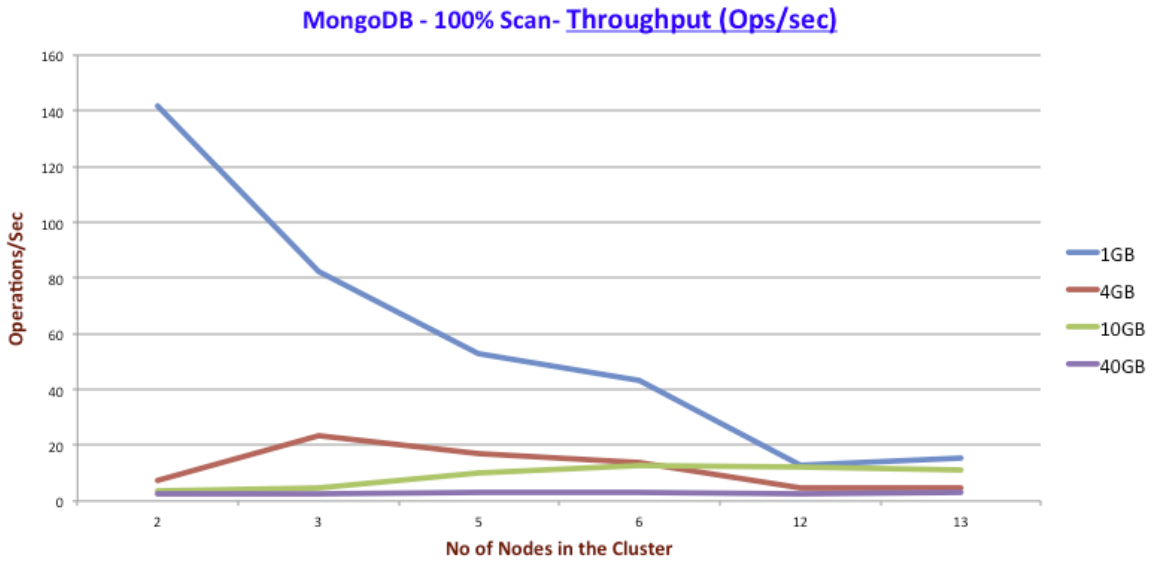

Figure 6.5. MongoDB : 100% Scan.

6.2 Cassandra

In this section, the results obtained during the benchmark testing of Cassandra are presented. Similar to section 6.1, the results of individual workload are analyzed in detail.

6.2.1 50% Read - 50% Write

In this workload, the read and write operations are independent of each other. It can be observed from the graph, in Figure 6.6, with 1 GB and 4 GB of data in the database, the time taken to execute the operations increased with the increase in the number of nodes in the cluster, which is evident from the decrease in throughput. Analyzing the access pattern and the partitioning technique in Cassandra can elucidate this behavior. In Cassandra, the records are accessed based on the key, which is indexed and sorted. The sorted record keys are stored in memory. When the size of the database is small, it is logical to achieve high efficiency with a small cluster size, since the records can be located faster. As and when the cluster size increases, the distribution of data to the nodes in the cluster creates additional network overhead (increased communication costs). Therefore, the time taken to access a record increases as more nodes are added to the cluster.

Cassandra with 10 GB and 40 GB of data in the database achieves better scalability. The performance of the database with 10 GB of data reaches the optimal throughput when the cluster size is at 12 nodes and then starts to decrease. This behavior can be reasoned with the balance between managing the complexity in sharding of data and because of the overhead of the communication protocol (gossip) between different nodes. For 40 GB of data, the performance improves steadily and based on the scalability performance of the database with 10 GB of data, it can be

estimated that the database could achieve its optimal performance when the cluster size is approximately 48 nodes.

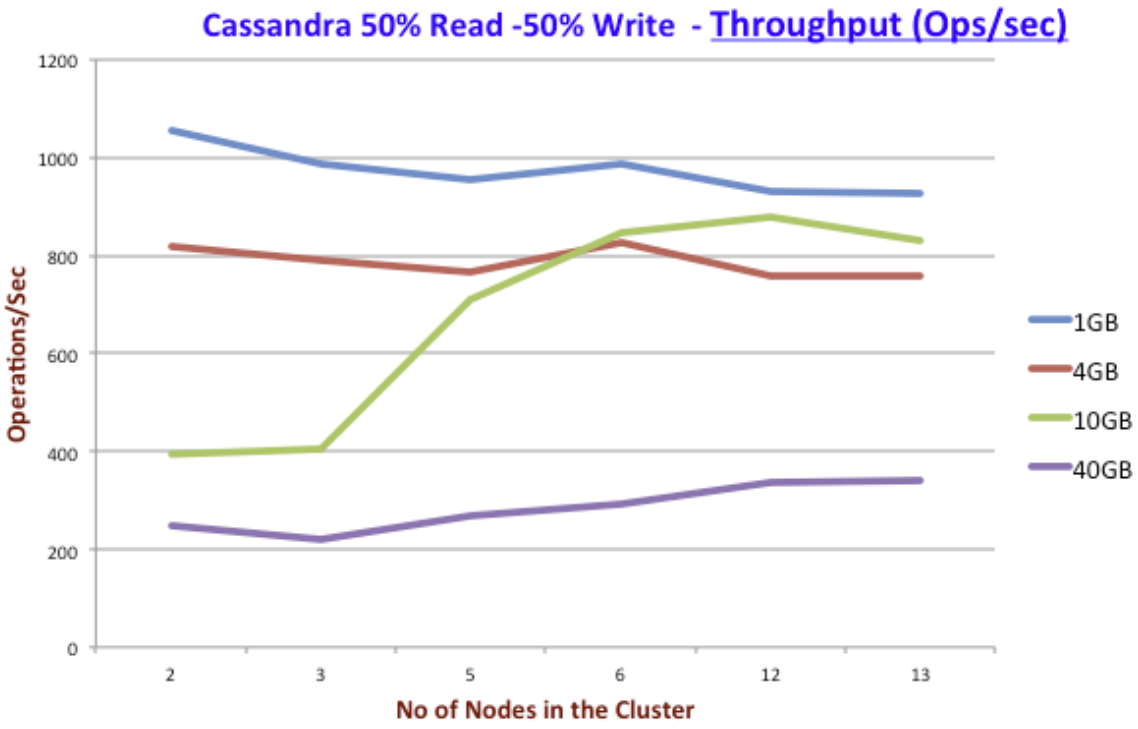

Figure 6.6. Cassandra : 50% Read - 50% Write.

6.2.2 100% Read

The results of 100% Read workload is shown in Figure 6.7. This workload follows a behavioral pattern similar to a 50% Read - 50% Write workload. It is important to note that the overall scale of the throughput was less compared to 50% Read - 50% Write workload. Most notably, Cassandra with 10 GB of data, scales well with increase in the number of nodes to form a bell curve that reaches the best

performance at 12 nodes. The same reasoning provided for the performance of 50% Read - 50% Write workload is applicable here.

The comparative decrease in the scale of the throughput for this workload can be attributed to the architecture of Cassandra and to the process involved in handling read operations. Cassandra follows a master-master or a decentralized architecture, where requests can be processed at every node. In case of a read request, when the requested data is not available in the specific node that processes the request, the processing node coordinates with other nodes, known as seek in Cassandra, to locate the data and process the request. This way of performing seeks slows down the performance of the system. Read performance of Cassandra can be significantly improved by adding more memory (RAM) because of the caching protocol in Cassandra.

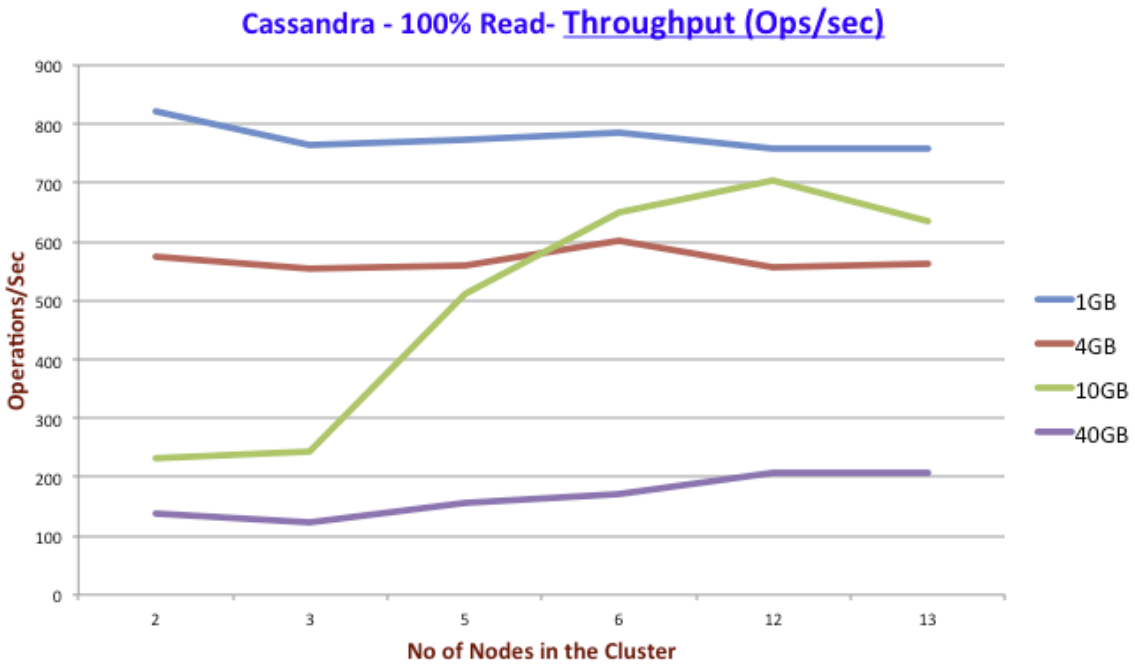

Figure 6.7. Cassandra : 100% Read.

6.2.3 100% Blind Write

Cassandra is designed to handle write operations more efficiently than read operations. This is evident from the results of 100% Blind Write workload, as shown in Figure 6.8. The performance of the database, irrespective of its size, decreased with the increase in the size of the cluster. This workload has the best throughput performance compared to all other workloads with approximately twice the performance of the 100% Read workload.

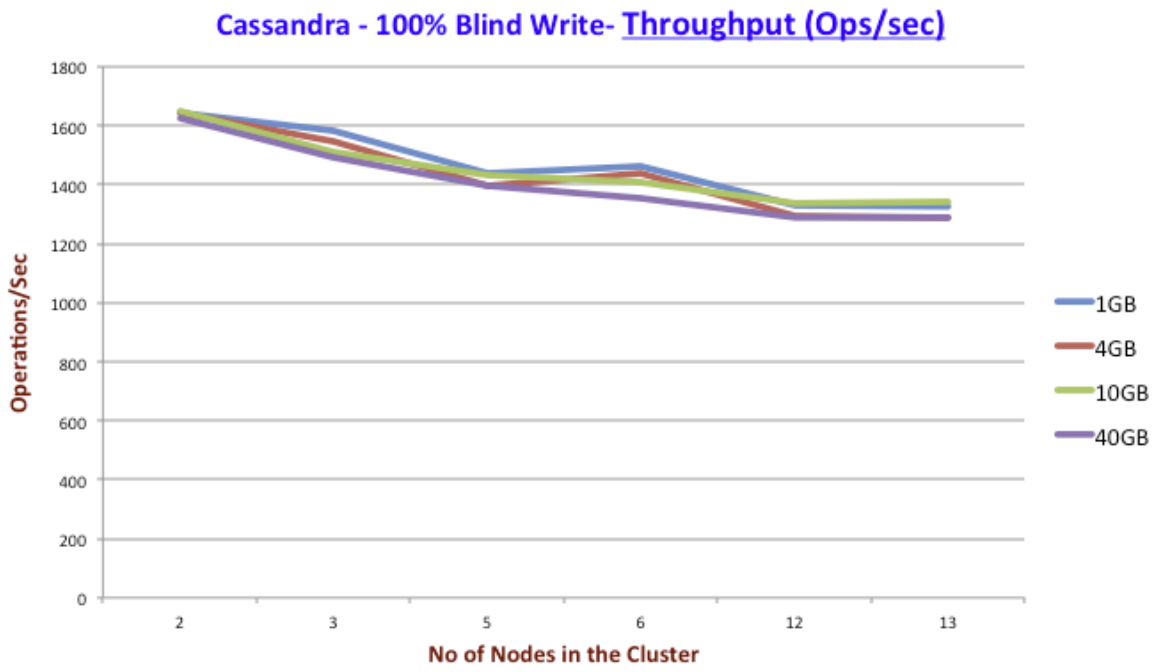

Figure 6.8. Cassandra : 100% Blind Write.

The behavioral pattern and the throughput performance can be explained by analyzing the architecture of Cassandra. The decentralized architecture of Cassandra accepts write operations on all the nodes in the cluster. When a write request is submitted at any node, the request is written to the commitlog that is present in the

memory. This process of writing the data to log file ensures durability of data. When the size of the commitlog reaches the configured limit, the data is written to disk.

The write requests are automatically partitioned and replicated throughout the cluster. There is no disk reads or coordination with other nodes involved while processing a write request, which translates to a higher write performance. The decrease in performance with the increase in the number of nodes is because of the overhead caused by the gossip protocol implemented in Cassandra, where each node has to communicate with other nodes in the cluster constantly.

6.2.4 100% Read-Modify-Write

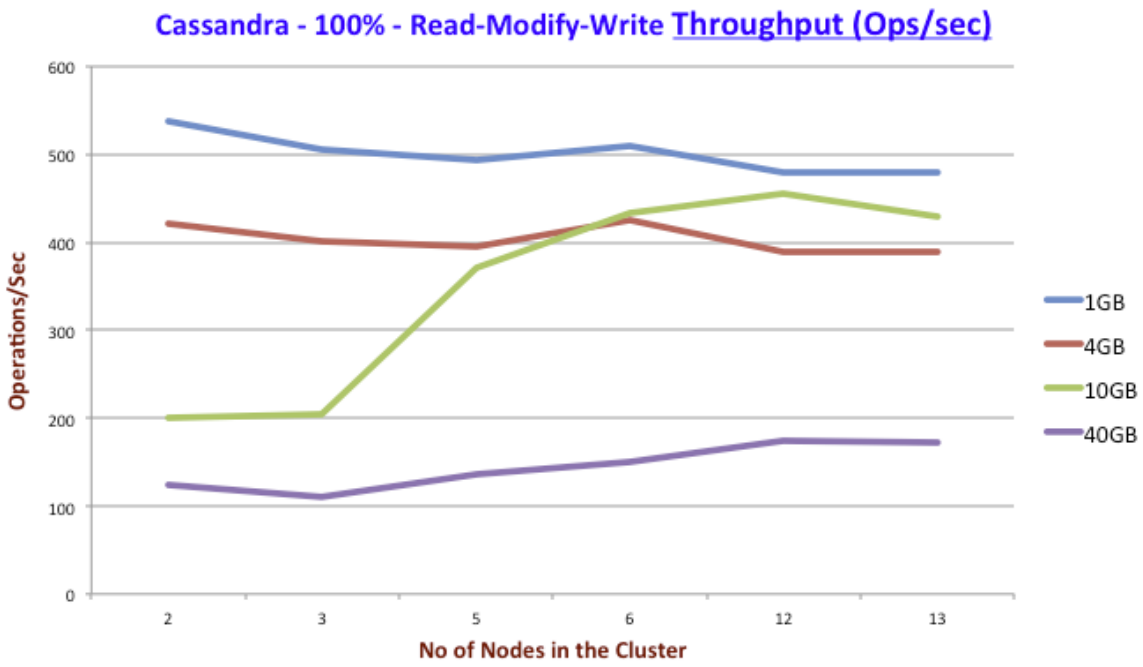

Figure 6.9. Cassandra : 100% Read-Modify-Write.

The performance of Read-Modify-Write workload is slower when compared to the 50% Read - 50% Write and 100% Read workloads. However, the workload follows a behavioral pattern (shown in Figure 6.9) similar to 50% Read - 50% Write and 100% Read workloads. The lower throughput of this workload can be ascribed to the way in which Cassandra handles read and write operations.

6.2.5 100% Scan

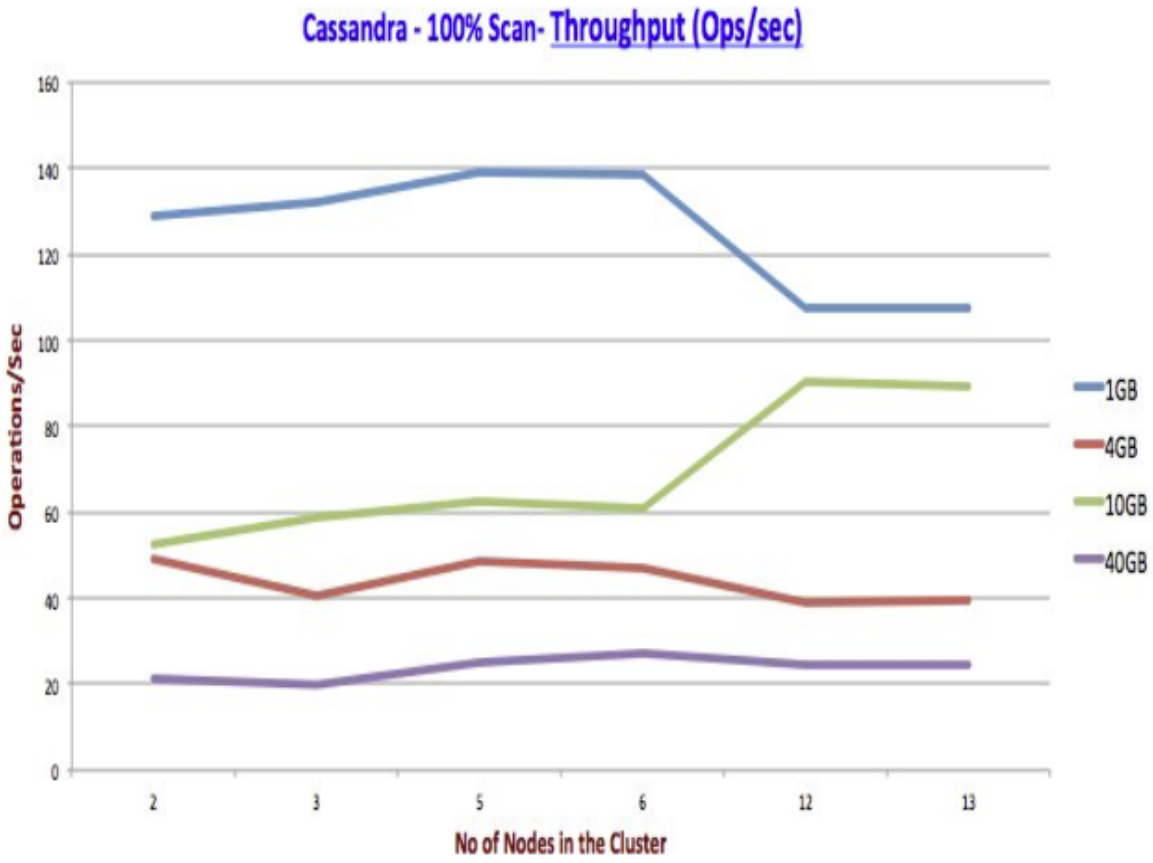

Figure 6.10. Cassandra : 100% Scan.

For executing this test case, the scan distribution method in YCSB client was set to uniform and 100 records were read for each scan. This workload showcased a unique behavior pattern, shown in Figure 6.10, compared to other workloads. The scale of throughput is much lower because this workload reads a range of records. The performance result of Cassandra with 1 GB of data increases slowly until 6 nodes and the performance of the database decreases by 34%. This is because of over distribution of data for a small dataset size. On the contrary, for a database with 10 GB of data, the performance increases with increase in the number of node. It can be reasoned that the distribution of data was optimal for the scan operation when the cluster size is at 12 nodes for this workload.

Most notable observation of this workload is that the database performance was better with 10 GB of data compared to 4 GB of data. This shows that for range (scan) queries in Cassandra, the size of each record in the database influences the performance.

6.3 Hbase

This section analyzes, in detail, the performance of Hbase based on the benchmark testing results obtained. For this benchmark testing (applicable only to Hbase) there were few anomalies observed and those anomalies were not included in this analysis.

6.3.1 50% Read - 50% Write

The results of this workload showed an upward trend, as shown in Figure 6.11. The performance increased with the increase in the number of nodes in the cluster. The throughput decreased with the increase in the size of the database. Hbase is a master-slave type of architecture. In case of a sharded cluster in Hbase, the data

is divided into regions and stored in regionservers. When a request is submitted, the regionservers are directly communicated for processing the requests instead of contacting the Hbase master. This can explain the behavior of Hbase. The increase in the number of nodes increases the system's capability to process more data.

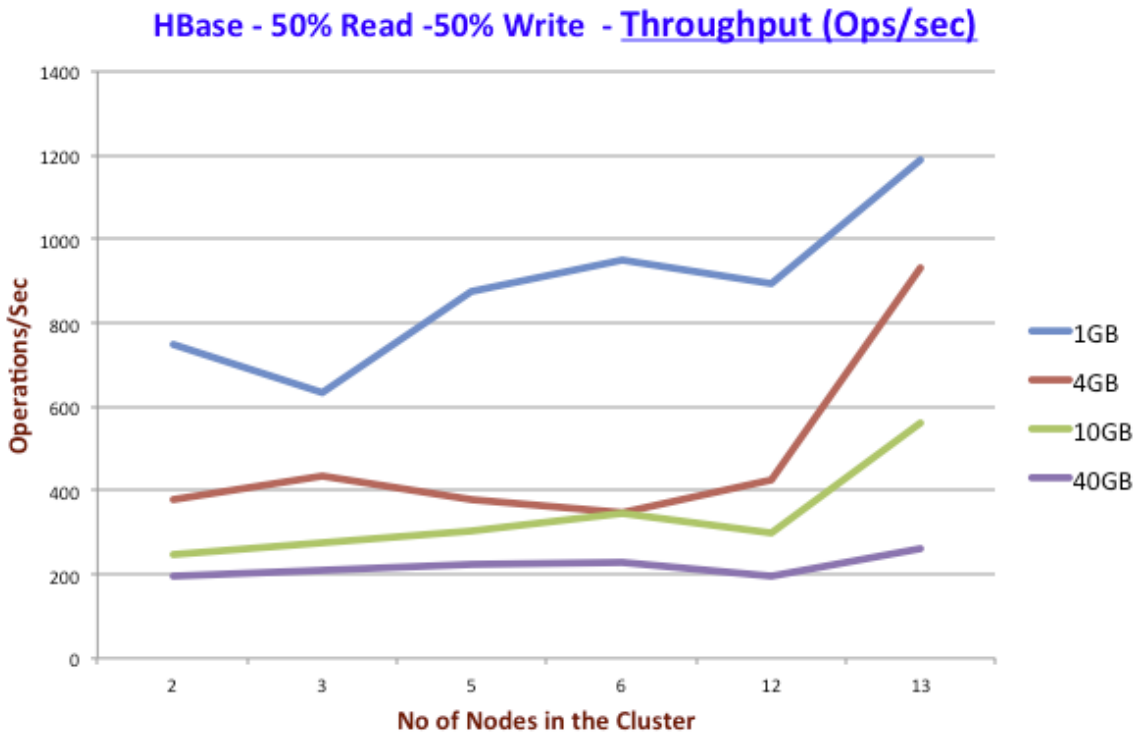

Figure 6.11. Hbase : 50% Read - 50% Write.

6.3.2 100% Read

This workload followed a behavior pattern similar to 50% Read - 50% Write workload and the same reasoning provided for 50% Read - 50% Write workload is applicable here. However, 50% Read - 50% Write workload has a better throughput compared to 100% Read workload. This difference in throughput is because of the

good write performance of Hbase. The results for this workload are shown in Figure 6.12.

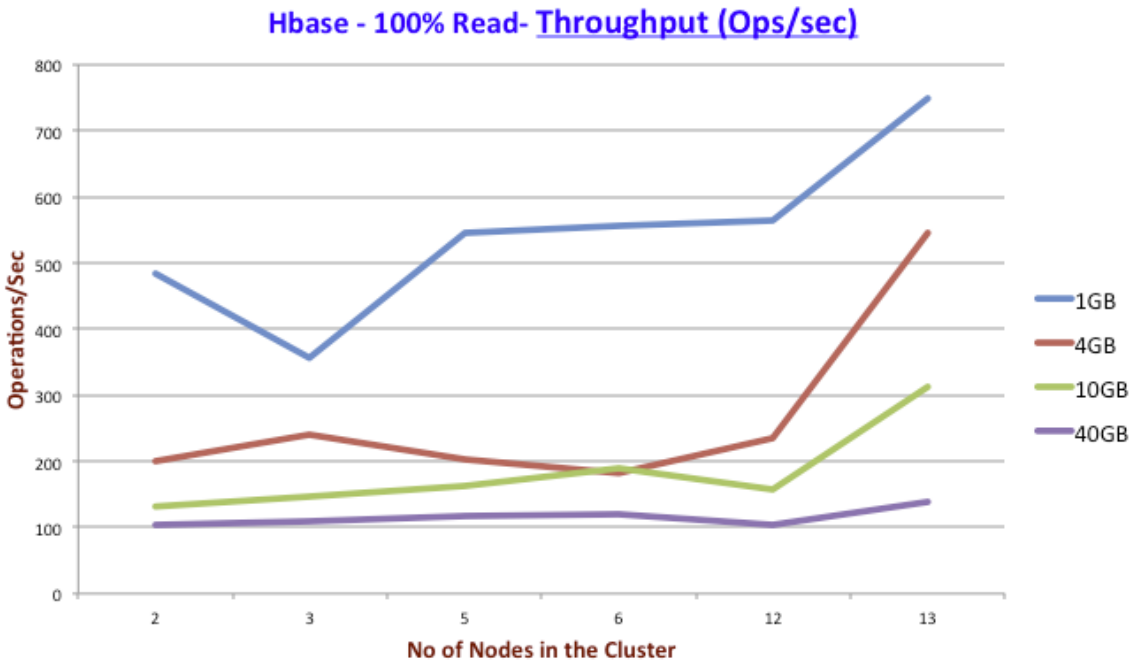

Figure 6.12. Hbase : 100% Read.

6.3.3 100% Blind Write

Hbase handles write operations by creating a new version of data. The versioning capabilities of Hbase are discussed in section 4.3.1. When a write request is submitted to Hbase, the request is first written to the write-ahead log (WAL). The data written to the WAL is placed in MemStore, which holds the modifications to the store in memory. Once the MemStore reaches the configured limit, the data is flushed to the disk. This process of handling the write request in Hbase is the reason for a

very high write performance. The benchmark results of Hbase is shown in Figure 6.13.

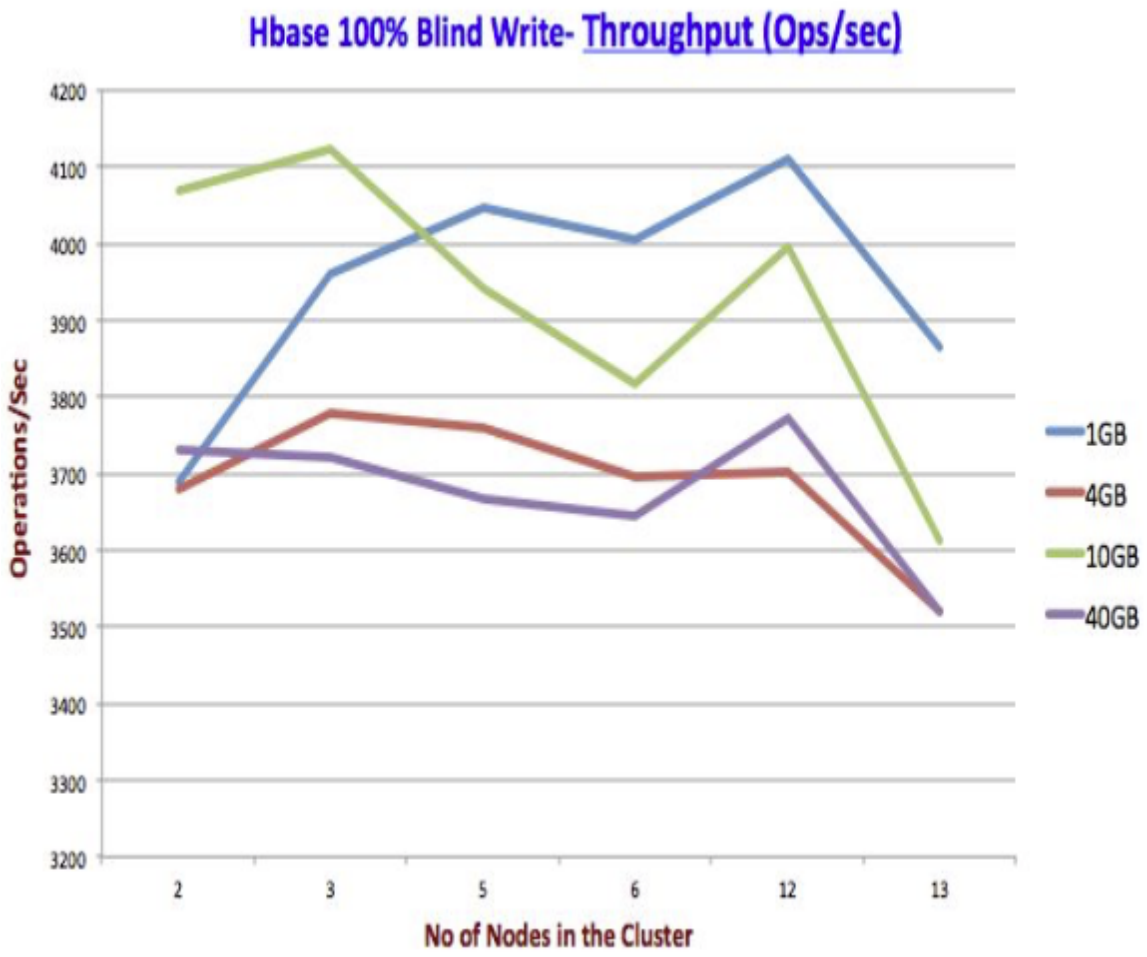

Figure 6.13. Hbase : 100% Blind Write.

The write performance of Hbase increased with the increase in the number of operations. This behavior was unique to Hbase and was not observed in other databases. Approximately 300% increase in throughput was observed with the increase in the number of operations. These results prove that Hbase is designed to

handle large amounts of write requests more efficiently. Figure 6.14 shows the write performance of Hbase with different operation count on different data set sizes.

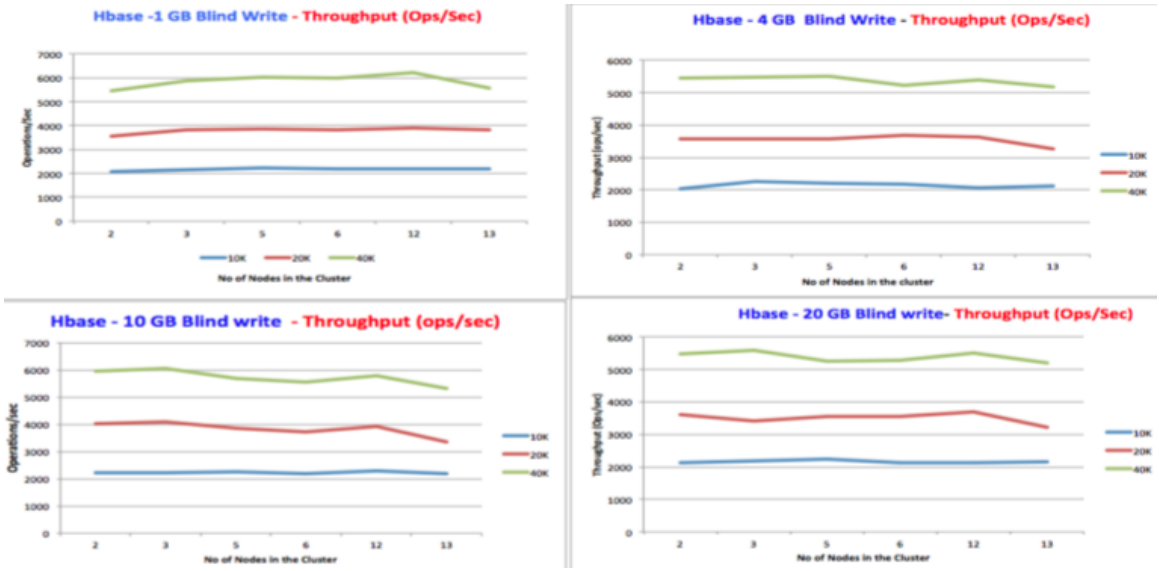

Figure 6.14. Hbase : 100% Blind Write performance w.r.t operation count.

6.3.4 100% Read-Modify-Write

In this workload, the value in a specific column qualifier is read and the new version of the value is written. As shown in Figure 6.15, this workload followed a performance pattern similar to 50% Read - 50% Write workload.

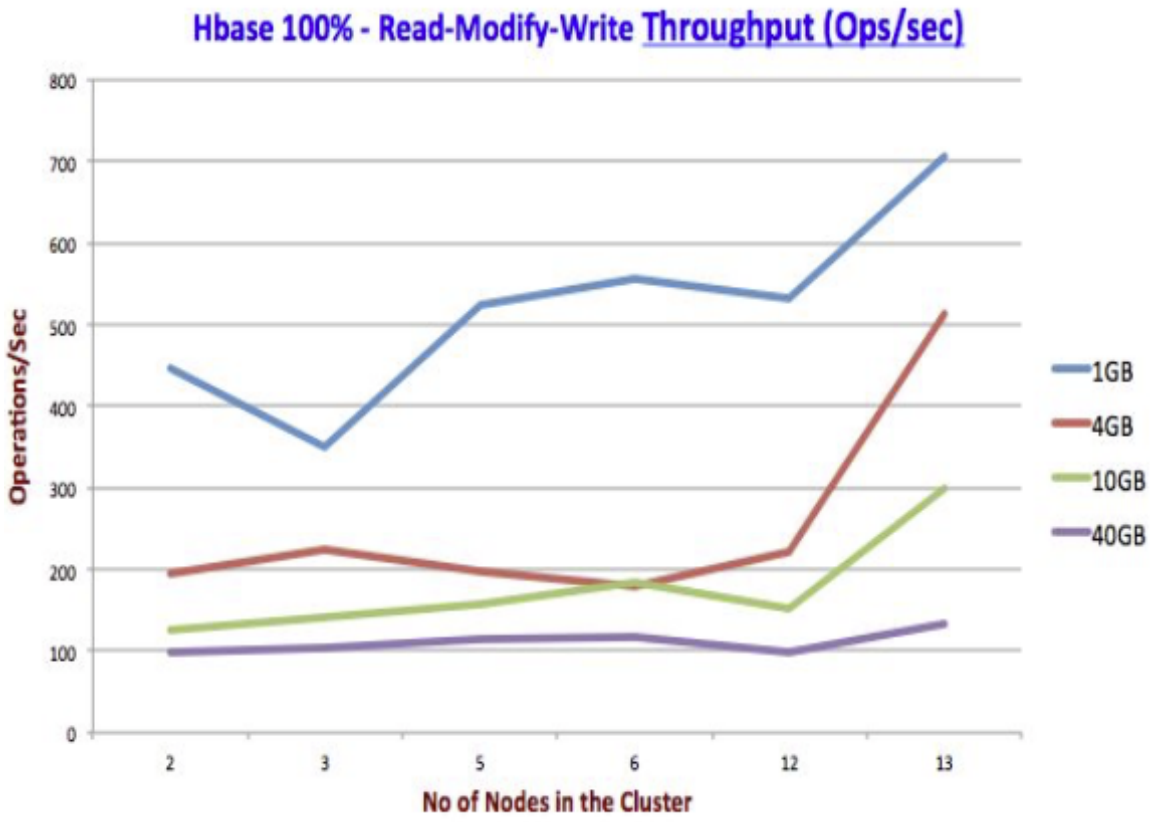

Figure 6.15. Hbase : 100% Read-Modify-Write.

6.3.5 100% Scan

The result of the scan operations in Hbase is shown in Figure 6.16. The throughput decreased with the increase in the size of the database. The performance improved with the increase in the size of the cluster with the database achieving best throughput when the size of the cluster is 13 nodes. The Ordered Partitioning in Hbase could be the reason for this performance.

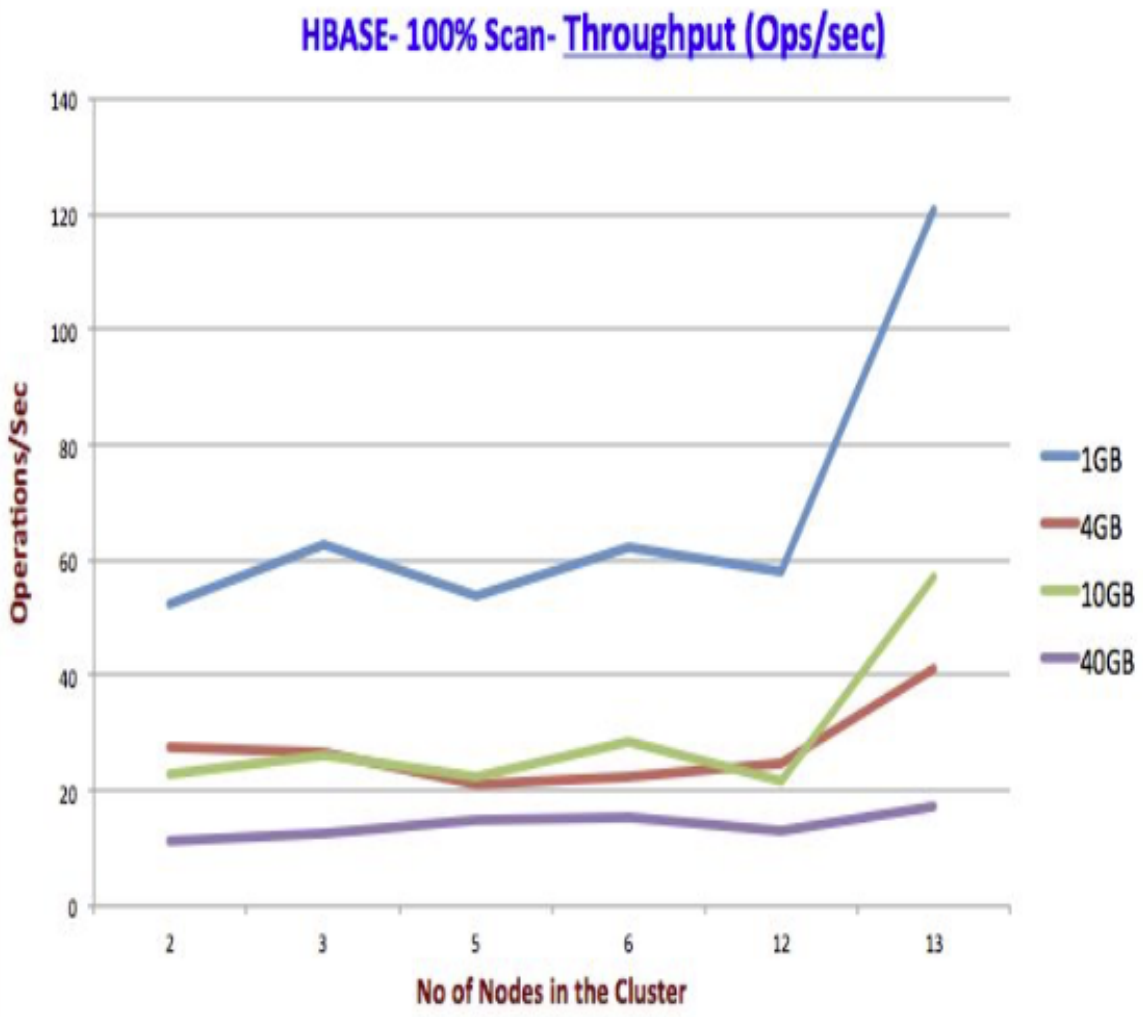

Figure 6.16. Hbase : 100% Scan.

6.4 MongoDB vs. Cassandra vs. Hbase

In the previous sections, the benchmark testing results of three databases, MongoDB, Cassandra and Hbase were presented along with a reasoned explanation on the behavior of each database under different workload conditions. These results have set a basic platform to identify the most suitable database (among MongoDB, Cassandra, and Hbase) for a particular dataset size under a given workload condition. This

section covers the performance results compared between MongoDB, Cassandra and Hbase for different workload conditions. The figures shown in this section will have a separate graph for different data set sizes with throughput of each database plotted against the increasing scale of cluster sizes. The section will conclude with a summary that would assist in choosing the most suitable database for a given workload, dataset, and cluster size combination.

6.4.1 50% Read - 50% Write

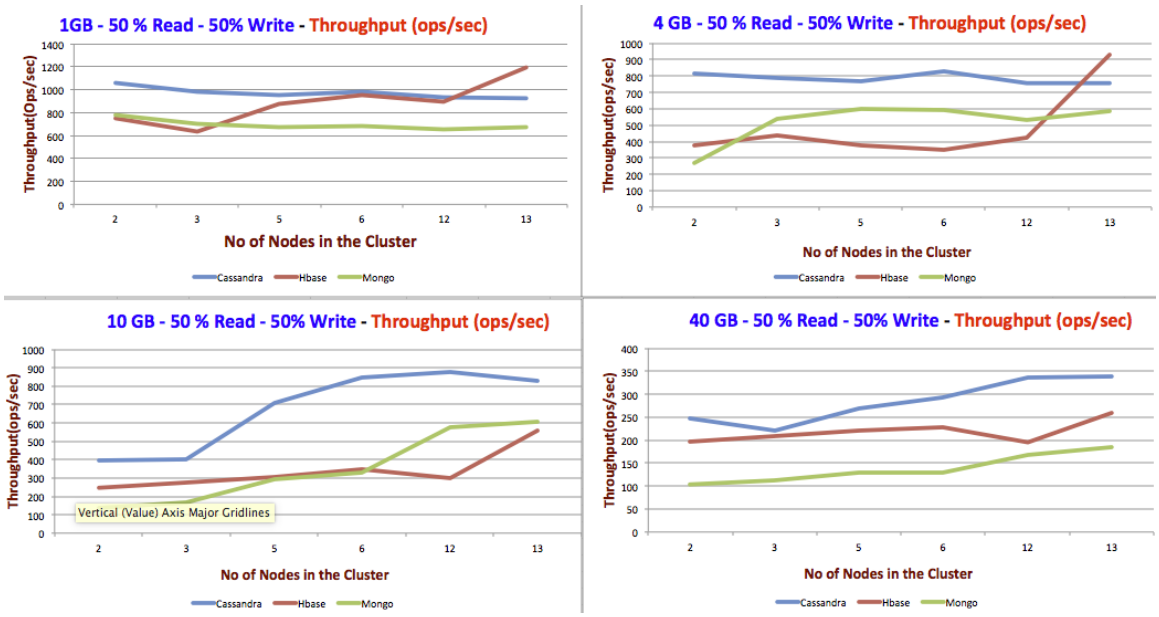

Figure 6.17. MongoDB vs. Cassandra vs. Hbase: 50% Read - 50% Write.

The comparative results for this workload is shown in Figure 6.17. For a small database (1 GB) and medium database (4 GB), with cluster size limited to 12 nodes, the throughput of Cassandra was higher compared to Hbase and MongoDB. However

as the cluster size increased over 12 nodes, the performance of Hbase was approximately 20% better than the performance of other two databases.

Cassandra scaled more efficiently and showed the best performance compared to other two databases, for large data sets. The performance of Cassandra with 10 GB of data was comparatively 92% better than the other two systems. Similarly, for a very large database (40 GB), Cassandra was 30% more efficient than Hbase and MongoDB.

6.4.2 100% Read

Given a circumstance where the database has to execute only read operations, MongoDB's performance was better for a small sets of data (see Figure 6.18). Its throughput was 50% more compared to Cassandra when the data set was up to 4 GB. Efficient storage structure of MongoDB can be attributed for this performance.

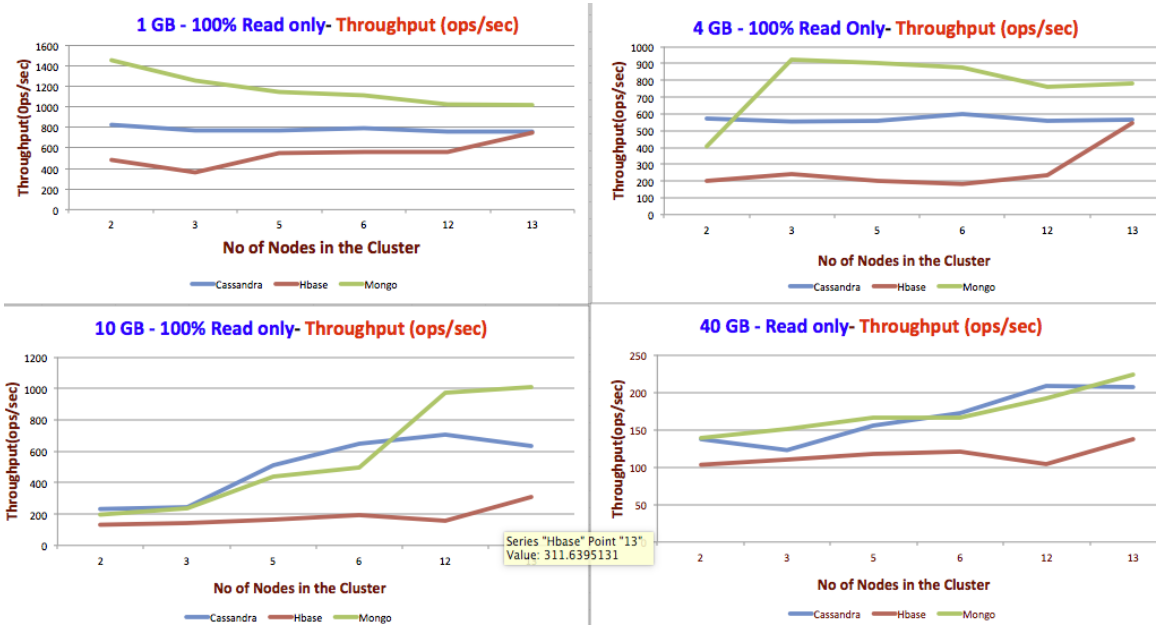

Figure 6.18. MongoDB vs. Cassandra vs. Hbase: 100% Read.

However, the databases showed a varied performance for large data sets. When the cluster size was less than 7 nodes and data size was 10 GB, Cassandra's throughput was slightly better than MongoDB. The results were inconclusive as to which system is better between MongoDB and Cassandra for 40 GB of data.

6.4.3 100% Blind Write

Hbase had the best performance for this workload, irrespective of the size of the database, with 265% better throughput performance than Cassandra, as shown in Figure 6.19. Hbase and Cassandra are both modeled after Google's Bigtable [9]. Though the basic architecture for Hbase and Cassandra are similar, the different approaches in handling the write requests can explain the exponential difference in the performance.

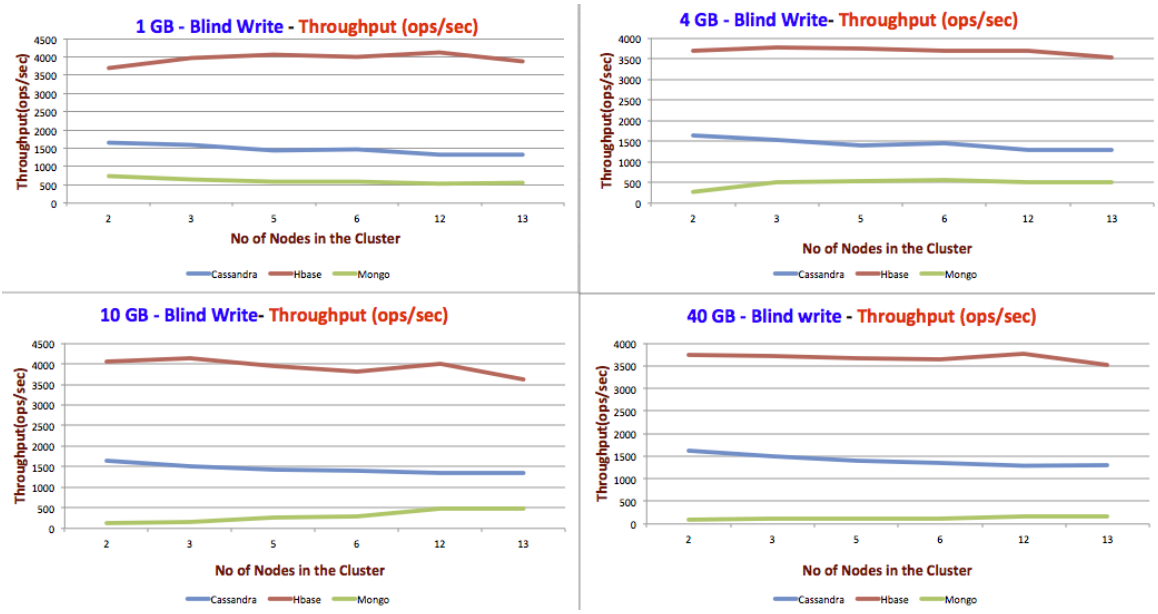

Figure 6.19. MongoDB vs. Cassandra vs. Hbase: 100% Blind Write.

6.4.4 100% Read-Modify-Write

The performance behavior exhibited in this workload was similar to 50% Read - 50% Write workload. The performance comparison of different databases for this workload is shown in Figure 6.20. Cassandra outperformed the other two databases for large sets of data (10 GB and 40 GB). For a small database (1 GB), Hbase achieved better throughput than Cassandra.

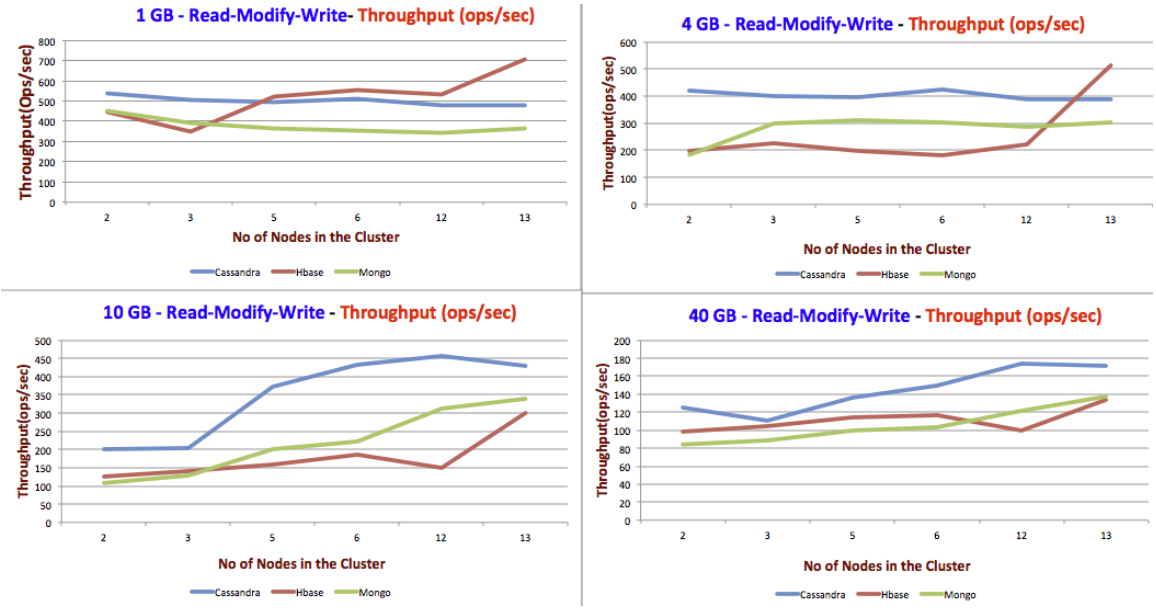

Figure 6.20. MongoDB vs. Cassandra vs. Hbase: 100% Read-Modify-Write.

6.4.5 100% Scan

The scan operations in this workload read 100 records per scan operation (Table 5.4). The performance of a database to execute scan operations is influenced by the partitioning technique implemented in the database. For MongoDB, range partitioning was implemented. A hash partitioning technique called Murmur3Partitioner

(section 4.5.3) was used in Cassandra. Ordered partitioning was used for Hbase. Figure 6.21 shows the performance of the three databases for 100% Scan workload.

Cassandra had the best throughput performance for large data sets, approximately twice the performance of Hbase. In case of small (1 GB) and medium (4 GB) databases, over distribution of data caused a decrease in the performance of Cassandra with increase in the cluster size. Conversely, performance of Hbase increased with the increase in the size of the cluster.

Figure 6.21. MongoDB vs. Cassandra vs. Hbase: 100% Scan.

6.4.6 Summary

The results discussed in this section are summarized in Table 6.2. The information in the table would assist in choosing the suitable database depending on the type of operations to be executed on the database and the size of the database.

Table 6.2. Summarization

Operations	1 GB	4 GB	10 GB	40 GB
50% Read - 50% Write	Cassandra (cluster size <12) Hbase (cluster size >12)		Cassandra	Cassandra
100% Read	MongoDB		Cassandra (cluster size <6) MongoDB (cluster size >6)	MongoDB or Cassandra
100% Blind Write	Hbase	Hbase	Hbase	Hbase
100% Read-Modify-Write	Cassandra (cluster size <5) Hbase (cluster size >5)	Cassandra (cluster size <12) Hbase (cluster size >12)	Cassandra	Cassandra
100% Scan	Cassandra (cluster size <12) Hbase (cluster size >12)		Cassandra	Cassandra

Additionally, the information regarding the scalability of the chosen database for a given data size is provided. This information would be helpful when planning the resources for an application.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Different NoSQL databases have different design features, which are advantageous for specific types of operations. It is necessary to examine the behavior of different NoSQL databases in order to make an informed decision on the choice of database suitable for an application. In this thesis, three of the most popular open source NoSQL databases, MongoDB, Cassandra and Hbase, were analyzed in detail.

A comprehensive analysis on the performance of each database on its ability to scale under different workload conditions and with different dataset sizes and different cluster sizes was illustrated. The performance comparison of the three NoSQL databases was discussed. Finally, a benchmark report, which provides a recommendation on the ideal database for a specific type of database operation, was presented.

7.2 Future Work

For the future work, including different categories of NoSQL databases and different sets of operations can increase the scope of the experiments in this thesis.

The in-memory caching in MongoDB, Cassandra, and Hbase is configurable and it can be increased or decreased based on the requirements of an application. For this experiment, the caching limit was set to default. As a next step, the effect of in-memory caching on the sharding capability of a NoSQL database could be examined.

Implementing an effective replication strategy is important for a large-scale distributed NoSQL database. In this thesis, the focus was on the performance of

the NoSQL system on a defined set of workload conditions. Other important factors such as data replication were not considered. In future, it would be interesting to analyze the difference in performance of a NoSQL database with and without data replication.

REFERENCES

[1] V. Abramova, J. Bernardino, and P. Furtado, "Exprimental Evaluation of NoSQL Databases," *International Journal of Database Management Systems*, vol. 6, no. 3, pp. 1–16, 2014.

[2] B. G. Tudorica and C. Bucur, "A comparison between several NoSQL databases with comments and notes," in *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*, 2011, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5993686

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

[4] IBM. (2011) Bringing big data to the enterprise. [Online]. Available: http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html

[5] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, Third Edition.* springer, 2011. [Online]. Available: http://www.springerlink.com/index/10.1007/978-1-4419-8834-8

[6] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems, Seventh Edition.* Pearson, 2015.

[7] C. Strozzi. (2007) NoSQL A relational Database Management System. [Online]. Available: http://www.strozzi.it

[8] Y. D. Network. (2009) Notes from the NoSQL Meetup. [Online]. Available: https://developer.yahoo.com/blogs/ydn/notes-nosql-meetup-7663.html

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, 2006, pp. 205–218. [Online]. Available: http://research.google.com/archive/bigtable-osdi06.pdf

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon?s Highly Available Key-value Store," in *Proceedings of the Symposium on Operating Systems Principles*, 2007, pp. 205–220. [Online]. Available: http://dl.acm.org/citation.cfm?id=1323293.1294281

[11] S. Edlich. (2015) NoSQL. [Online]. Available: http://nosql-database.org

[12] E. A. Brewer, "Towards Robust Distributed Systems," in *Proceedings of the Symposium on Principles of Distributed Computing*, 2000, p. 7. [Online]. Available: https://www.cs.berkeley.edu/ brewer/cs262b-2004/PODC-keynote.pdf

[13] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, p. 51, 2002.

[14] E. Brewer, "CAP twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[15] S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[16] D. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[17] VerticalVsHorizontal. Vertical vs Horizontal Scalability. [Online]. Available: http://pc-freak.net/blog/vertical-horizontal-server-services-scaling-vertical-horizontal-hardware-scaling/

[18] C. Henderson, *Building Scalable Web Sites*. O'Reilly Media, 2006, no. May. [Online]. Available: http://shop.oreilly.com/product/9780596102357.do

[19] M. Stonebraker and R. Cattell, "10 Rules for Scalable Performance in 'Simple Operation' Datastores," *Communications of the ACM*, vol. 54, no. 6, p. 72, 2011.

[20] Y. Izrailevsky. (2011) NoSQL at Netflix. [Online]. Available: http://techblog.netflix.com/2011/01/nosql-at-netflix.html

[21] V. Malaya. (2013) SQL vs. NoSQL. [Online]. Available: http://sql-vs-nosql.blogspot.com/2013/10/the-base-difference-between-sql-and.html

[22] MongoDB. (2009) MongoDB. [Online]. Available: https://www.mongodb.com/

[23] JSONiq. JSONiq. [Online]. Available: http://www.jsoniq.org/

[24] Apache, "HBase," 2013. [Online]. Available: http://hbase.apache.org/

[25] Cassandra, "The Apache Cassandra Project," 2010. [Online]. Available: http://cassandra.apache.org/

[26] Redis, "Redis," 2009. [Online]. Available: http://redis.io/

[27] S. D. Kuznetsov and A. V. Poskonin, "NoSQL data management systems," *Programming and Computer Software*, vol. 40, no. 6, pp. 323–332, 2014. [Online]. Available: http://link.springer.com/10.1134/S0361768814060152

[28] MySQL. MySQL. [Online]. Available: https://www.mysql.com/

[29] Oracle. Oracle. [Online]. Available: http://www.oracle.com

[30] Apache, "CouchDB," 2005. [Online]. Available: http://couchdb.apache.org/

[31] P. Vagata and K. Wilfong. Scaling the Facebook data warehouse to 300 PB. [Online]. Available: https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/

[32] Voldemort. Project Voldemort. [Online]. Available: http://www.project-voldemort.com/

[33] D. D. Chamberlin and R. Boyce, "SEQUEL: A structured English query language," *Proceedings of the 1974 ACM SIGFIDET*, pp. 249–264, 1974.

[34] UnQL. Unstructured Query language. [Online]. Available: http://www.dataversity.net/unql-a-standardized-query-language-for-nosql-databases/

[35] Apache, "Hive," 2008. [Online]. Available: https://hive.apache.org/

[36] K. Ferguson, V. Raghunathan, R. Leeds, and S. Lindsay. Lounge. [Online]. Available: http://www.oracle.com

[37] Amazon. Amazon DynamoDB. [Online]. Available: https://aws.amazon.com/dynamodb/

[38] OrientDB. OrientDB. [Online]. Available: http://orientdb.com/

[39] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, p. 12, 2011.

BIOGRAPHICAL STATEMENT

Surya Narayanan Swaminathan received his Bachelors in Computer Science and Engineering in 2007 from Sri Venkateswara College of Engineering under Anna University, Chennai, India. He worked for six years in Infosys Technologies. He started his Masters in Computer Science in 2014 in University of Texas at Arlington. His areas of interests are databases, data mining and cloud computing.